DATA STRUCTURES WITH C
(Common to CSE & ISE)

Subject Code: 10CS35 **I.A. Marks : 25**
Hours/Week : 04 **Exam Hours: 03**
Total Hours : 52 **Exam Marks: 100**

PART – A

UNIT - 1 8 Hours
BASIC CONCEPTS: Pointers and Dynamic Memory Allocation, Algorithm Specification, Data Abstraction, Performance Analysis, Performance Measurement

UNIT -2 6 Hours
ARRAYS and STRUCTURES: Arrays, Dynamically Allocated Arrays, Structures and Unions, Polynomials, sparse Matrices, Representation of Multidimensional Arrays

UNIT - 3 6 Hours
STACKS AND QUEUES: Stacks, Stacks Using Dynamic Arrays, Queues, Circular Queues Using Dynamic Arrays, Evaluation of Expressions, Multiple Stacks and Queues.

UNIT - 4 6 Hours
LINKED LISTS: Singly Linked lists and Chains, Representing Chains in C, Linked Stacks and Queues, Polynomials, Additional List operations, Sparse Matrices, Doubly Linked Lists

PART - B

UNIT - 5 6 Hours
TREES – 1: Introduction, Binary Trees, Binary Tree Traversals, Threaded Binary Trees, Heaps.

UNIT – 6 6 Hours
TREES – 2, GRAPHS: Binary Search Trees, Selection Trees, Forests, Representation of Disjoint Sets, Counting Binary Trees, The Graph Abstract Data Type.

UNIT - 7 6 Hours
PRIORITY QUEUES Single- and Double-Ended Priority Queues, Leftist Trees, Binomial Heaps, Fibonacci Heaps, Pairing Heaps.

UNIT - 8 8 Hours
EFFICIENT BINARY SEARCH TREES: Optimal Binary Search Trees, AVL Trees, Red-Black Trees, Splay Trees.

Text Book:
1. Horowitz, Sahni, Anderson-Freed: Fundamentals of Data Structures in C, 2nd Edition, Universities Press, 2007. (Chapters 1, 2.1 to 2.6, 3, 4, 5.1 to 5.3, 5.5 to 5.11, 6.1, 9.1 to 9.5, 10)

Reference Books:
1. Yedidyah, Augenstein, Tannenbaum: Data Structures Using C and C++, 2nd Edition, Pearson Education, 2003.
2. Debasis Samanta: Classic Data Structures, 2nd Edition, PHI, 2009.
3. Richard F. Gilberg and Behrouz A. Forouzan: Data Structures A Pseudocode Approach with C, Cengage Learning, 2005.
4. Robert Kruse & Bruce Leung: Data Structures & Program Design in C, Pearson Education, 2007.

TABLE OF CONTENTS

TABLE OF CONTENTS 1
UNIT – 1: BASIC CONCEPTS 4
1.1-Pointers and Dynamic Memory Allocation: 4
1.2. Algorithm Specification 8
1.3. Data Abstraction 9
1.4. Performance Analysis 10
1.5. Performance Measurement: 11
1.6 RECOMMENDED QUESTIONS 13
UNIT -2 : ARRAYS and STRUCTURES 14
2.1 ARRAY 14
2.2. Dynamically Allocating Multidimensional Arrays 18
2.3. Structures and Unions 20
2.4 .Polynomials 23
2.5. Sparse Matrices 26
2.6. Representation of Multidimensional arrays 29
2.7. RECOMMENDED QUESTIONS 31
UNIT – 3 : STACKS AND QUEUES 32
3.1.Stacks: 32
3.2. Stacks Using Dynamic Arrays 34
3.3. Queues 34
3.4. Circular Queues Using Dynamic Arrays 37
3.5. Evaluation of Expressions: Evaluating a postfix expression 39
3.6. Multiple Stacks and Queues 43
3.7. RECOMMENDED QUESTIONS 44
UNIT – 4 : LINKED LISTS 45
4.1. Singly Linked lists and Chains 45
4.2. Representing Chains in C: 46
4.3. Linked Stacks and Queues 47
4.4. Polynomials: 49
4.5. Additional List operations: 52
4.6. Sparse Matrices 54

4.7. Doubly Linked Lists 58
4.8. RECOMMENDED QUESTIONS 60
UNIT – 5 : TREES – 1 61
5.1 Introduction: 61
5.2 Binary Trees: 63
5.3 Binary tree Traversals: 65
5.4. Threaded Binary trees: 67
5.5. Heaps 67
5.6. RECOMMENDED QUESTIONS 70
UNIT – 6 : TREES – 2, GRAPHS 71
6.1 Binary Search Trees 74
6.2. Selection Trees 76
6.3 Forests 78
6.4 Representation of Disjoint Sets 78
6.5 Counting Binary Trees: 79
6.6 The Graph Abstract Data Type 80
6.7. RECOMMENDED QUESTIONS 81
UNIT – 7 : PRIORITY QUEUES 82
7.1. Single- and Double-Ended Priority Queues: 82
7.2. Leftist tree: 85
7.3. Binomial Heaps 87
7.4. Fibonacci Heaps 91
7.5. Pairing heaps 95
7.6. RECOMMENDED QUESTIONS 97
UNIT – 8 : EFFICIENT BINARY SEARCH TREES 98
8.1. Optimal Binary Search Trees 98
8.2. AVL Trees 99
8.3. Red-black Trees 103
8.5. RECOMMENDED QUESTIONS 119

UNIT – 1: BASIC CONCEPTS

1.1-Pointers and Dynamic Memory Allocation:

In computer science, a pointer is a programming language data type whose value refers directly to (or "points to") another value stored elsewhere in the computer memory using its address. For high-level programming languages, pointers effectively take the place of general purpose registers in low-level languages such as assembly language or machine code, but may be in available memory. A pointer references a location in memory, and obtaining the value at the location a pointer refers to is known as dereferencing the pointer. A pointer is a simple, more concrete implementation of the more abstract reference data type. Several languages support some type of pointer, although some have more restrictions on their use than others.

Pointers to data significantly improve performance for repetitive operations such as traversing strings, lookup tables, control tables and tree structures. In particular, it is often much cheaper in time and space to copy and dereference pointers than it is to copy and access the data to which the pointers point.Pointers are also used to hold the addresses of entry points for called subroutines in procedural programming and for run-time linking to dynamic link libraries (DLLs). In object-oriented programming, pointers to functions are used for binding methods, often using what are called virtual method tables.

Declaring a pointer variable is quite similar to declaring an normal variable all you have to do is to insert a star '*' operator before it.

General form of pointer declaration is -

type* name;

where type represent the type to which pointer thinks it is pointing to.

Pointers to machine defined as well as user-defined types can be made

Pointer Intialization: variable_type *pointer_name = 0;

or

variable_type *pointer_name = NULL;

char *pointer_name = "string value here";

The operator that gets the value from pointer variable is * (indirection operator). This is called the reference to pointer.

P=&a

So the pointer p has address of a and the value that that contained in that address can be accessed by : *p

So the operations done over it can be explained as below:

a++;

a=a+1;

*p=*p+1;

(*p)++:

While "pointer" has been used to refer to references in general, it more properly applies to data structures whose interface explicitly allows the pointer to be manipulated (arithmetically via *pointer arithmetic*) as a memory address, as opposed to a magic cookie or capability where this is not possible.

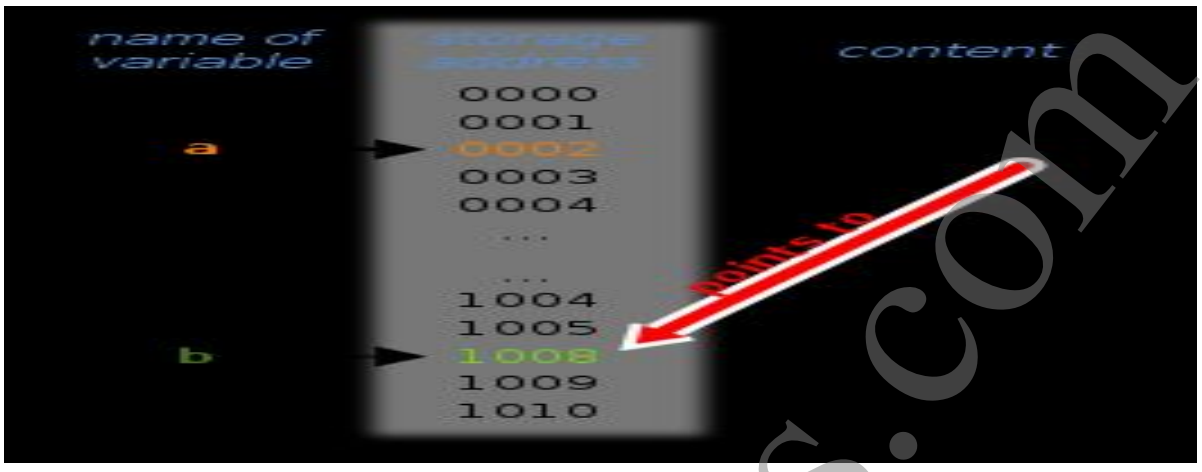


Fig 1: Pointer *a* pointing to the memory address associated with variable *b*. Note that in this particular diagram, the computing architecture uses the same address space and data primitive for both pointers and non-pointers; this need not be the case.

Pointers and Dynamic Memory Allocation:

Although arrays are good things, we cannot adjust the size of them in the middle of the program. If our array is too *small* - our program will fail for large data. If our array is too *big* - we waste a lot of space, again restricting what we can do. The right solution is to build the data structure from small pieces, and add a new piece whenever we need to make it larger. *Pointers* are the connections which hold these pieces together!

Pointers in Real Life

In many ways, telephone numbers serve as pointers in today's society. To contact someone, you do not have to carry them with you at all times. *All you need is their number.* Many different people can all have your number simultaneously. *All you need do is copy the pointer.* More complicated structures can be built by combining pointers. *For example, phone trees or directory information.* Addresses are a more physically correct analogy for pointers, since they really are memory addresses.

Linked Data Structures

All the dynamic data structures we will build have certain shared properties. We need a pointer to the entire object so we can find it. Note that this is a pointer, not a cell. Each cell contains one or more data fields, which is what we want to store. Each cell contains a pointer field to at least one ``next'' cell. Thus much of the space used in linked data structures is not data! We must be able to detect the end of the data structure. This is why we need the NIL pointers.

There are four functions defined in c standard for dynamic memmory allocation - calloc, free, malloc and realloc. But in the heart of DMA there are only 2 of them malloc and free. Malloc stands for memmory allocations and is used to allocate memmory from the heap while free is used to return allocated memmory from malloc back to heap. Both these functions uses a standard library header

<stdlib.h> .Warning !!! - free ( ) function should be used to free memmory only allocated previously from malloc, realloc or calloc. Freeing a random or undefined or compiler allocated memmory can lead to severe damage to the O.S., Compiler and Computer Hardware Itself, in form of nasty system crashes.

The prototype of malloc ( ) function is -

void *malloc (size_t number_of_bytes)

Important thing to nore is malloc return a void pointer which can be converted to any pointer type as explained in previous points. Also size_t is a special type of unsigned integer defined in <stdlib.h> capable of storing largest memmory size that can be allocated using DMA, number_of_bytes is a value of type size_t generally a integer indicating the amount of memmory to be allocated. Function malloc ( ) will be returning a null pointer if memmory allocation fails and will return a pointer to first region of memmory allocated when succsefull. It is also recommended you check the pointer returned for failure in allocation before using the returned memmory for increasing stability of your program, generally programmers provide some error handling code in case of failures. Also this returned pointer never needs a typecast in C since it is a void pointer, it is a good practice to do one since it is required by C++ and will produce a error if you used C++ compiler for compilation.Another commonly used operator used with malloc is sizeof operator which is used to calculate the value of number_of_bytes by determing the size of the compiler as well as user defined types and variables.

The prototype of free ( ) function is -

void free (void *p)

Function free ( ) is opposite of malloc and is used to return memmory previously allocated by other DMA functions. Also only memmory allocated using DMA should be free using free () otherwise you may corrupt your memmory allocation system at minimum.

C Source code shown below shows simple method of using dynamic memmory allocation elegantly –

```
#include <stdio.h>

#include <stdlib.h>

int main ()

{

int *p;

p = (int *) malloc ( sizeof (int) ); //Dynamic Memmory Allocation

if (p == NULL) //Incase of memmory allocation failure execute the error handling code block

{

 printf ("\nOut of Memmory");

  exit (1);

}

*p = 100;
```

```
printf ("\n p = %d", *p); //Display 100 ofcourse.

return 0;

}
```

Dynamic Allocation: To get dynamic allocation, use new:

```
p := New(ptype);
```

New(ptype) allocates enough space to store exactly one object of the type ptype. Further, it returns a pointer to this empty cell. Before a new or otherwise explicit initialization, a pointer variable has an arbitrary value which points to *trouble*!

Warning - initialize all pointers before use. Since you cannot initialize them to explicit constants, your only choices are

NIL - meaning explicitly nothing.

New(ptype) - a fresh chunk of memory.

Pointer Examples

Example: P: = new(node); q := new(node);

p.x grants access to the field *x* of the record pointed to by *p*.

```
p^.info := "music";

q^.next := nil;
```

The pointer value itself may be copied, which does not change any of the other fields.

Note this difference between assigning pointers and what they point to.

```
p := q;
```

We get a real mess. We have completely lost access to music and can't get it back! Pointers are *unidirectional*.

Alternatively, we could copy the object being pointed to instead of the pointer itself.

```
p^ := q^;
```

What happens in each case if we now did:

```
p^.info := "data structures";
```

Where Does the Space Come From?

Can we really get as much memory as we want without limit just by using New?

No, because there are the physical limits imposed by the size of the memory of the computer we are using. Usually Modula-3 systems let the dynamic memory come from the ``other side" of the ``activation record stack" used to maintain procedure calls.Just as the stack reuses memory when a procedure exits, dynamic storage must be recycled when we don't need it anymore.

Garbage Collection

The Modula-3 system is constantly keeping watch on the dynamic memory which it has allocated, making sure that *something* is still pointing to it. If not, there is no way for you to get access to it, so the space might as well be recycled. The *garbage collector* automatically frees up the memory which has nothing pointing to it. It frees you from having to worry about explicitly freeing memory, at the cost of leaving certain structures which it can't figure out are really garbage, such as a circular list.

Explicit Deallocation

Although certain languages like Modula-3 and Java support garbage collection, others like C++ require you to explicitly deallocate memory when you don't need it.

Dispose(p) is the opposite of New - it takes the object which is pointed to by *p* and makes it available for reuse. Note that each dispose takes care of only one cell in a list. To dispose of an entire linked structure we must do it one cell as a time. Note we can get into trouble with *dispose*:

Of course, it is too late to dispose of music, so it will endure forever without garbage collection. Suppose we *dispose(p)*, and later allocation more dynamic memory with *new*. The cell we disposed of might be reused. Now what does *q* point to?

Answer - the same location, but it means something else! So called *dangling references* are a horrible error, and are the main reason why Modula-3 supports garbage collection. A dangling reference is like a friend left with your old phone number after you move. Reach out and touch someone - eliminate dangling references!

Security in Java

It is possible to explicitly dispose of memory in Modula-3 when it is really necessary, but it is strongly discouraged. Java does not allow one to do such operations on pointers at all. The reason is *security*.Pointers allow you access to raw memory locations. In the hands of skilled but evil people, unchecked access to pointers permits you to modify the operating system's or other people's memory contents.

1.2. Algorithm Specification:

A pragmatic approach to algorithm specification and verification is presented. The language AL provides a level of abstraction between a mathematical specification notation and a programming language, supporting compact but expressive algorithm description.

Proofs of correctness about algorithms written in AL can be done via an embedding of the semantics of the language in a proof system; implementations of algorithms can be done through translation to standard programming languages.

The proofs of correctness are more tractable than direct verification of programming language code; descriptions in AL are more easily related to executable programs than standard mathematical specifications. AL provides an independent, portable description which can be related to different proof systems and different programming languages.

Several interfaces have been explored and tools for fully automatic translation of AL specifications into the HOL logic and Standard ML executable code have been implemented. A substantial case study uses AL as the common specification language from which both the formal proofs of correctness and executable code have been produced.

1.3. Data Abstraction

Abstraction is the process by which data and programs are defined with a representation similar to its meaning (semantics), while hiding away the implementation details. Abstraction tries to reduce and factor out details so that the programmer can focus on a few concepts at a time. A system can have several *abstraction layers* whereby different meanings and amounts of detail are exposed to the programmer. For example, low-level abstraction layers expose details of the hardware where the program is run, while high-level layers deal with the business logic of the program.

The following English definition of abstraction helps to understand how this term applies to computer science, IT and objects:

abstraction - a concept or idea not associated with any specific instance[1]

Abstraction captures only those detail about an object that are relevant to the current perspective. The concept originated by analogy with abstraction in mathematics. The mathematical technique of abstraction begins with mathematical definitions, making it a more technical approach than the general concept of abstraction in philosophy. For example, in both computing and in mathematics, numbers are concepts in the programming languages, as founded in mathematics. Implementation details depend on the hardware and software, but this is not a restriction because the computing concept of number is still based on the mathematical concept.

In computer programming, abstraction can apply to control or to data: Control abstraction is the abstraction of actions while data abstraction is that of data structures. Control abstraction involves the use of subprograms and related concepts control flows. Data abstraction allows handling data bits in meaningful ways. For example, it is the basic motivation behind datatype. One can regard the notion of an object (from object-oriented programming) as an attempt to combine abstractions of data and code. The same abstract definition can be used as a common interface for a family of objects with different implementations and behaviors but which share the same meaning. The inheritance mechanism in object-oriented programming can be used to define an abstract class as the common interface.

Data abstraction enforces a clear separation between the *abstract* properties of a data type and the *concrete* details of its implementation. The abstract properties are those that are visible to client code that makes use of the data type—the *interface* to the data type—while the concrete implementation is kept entirely private, and indeed can change, for example to incorporate efficiency improvements over time. The idea is that such changes are not supposed to have any impact on client code, since they involve no difference in the abstract behaviour.

For example, one could define an abstract data type called *lookup table* which uniquely associates *keys* with *values*, and in which values may be retrieved by specifying their corresponding keys. Such a lookup table may be implemented in various ways: as a hash table, a binary search tree, or even a simple linear list of (key:value) pairs. As far as client code is concerned, the abstract properties of the type are the same in each case. Of course, this all relies on getting the details of the interface right in the first place, since any changes there can have major impacts on client code. As one way to look at this: the interface forms a *contract* on agreed behaviour between the data type and client code; anything not spelled out in the contract is subject to change without notice.

Languages that implement data abstraction include Ada and Modula-2. Object-oriented languages are commonly claimed to offer data abstraction; however, their inheritance concept tends to put information in the interface that more properly belongs in the implementation; thus, changes to such information ends up impacting client code, leading directly to the Fragile binary interface problem.

1.4. Performance Analysis:

Performance analysis involves gathering formal and informal data to help customers and sponsors define and achieve their goals. Performance analysis uncovers several perspectives on a problem or opportunity, determining any and all drivers towards or barriers to successful performance, and proposing a solution system based on what is discovered.

A lighter definition is:

Performance analysis is the front end of the front end. It's what we do to figure out what to do. Some synonyms are planning, scoping, auditing, and diagnostics.

What does a performance analyst do?

Here's a list of some of the things you *may* be doing as part of a performance analysis:

- Interviewing a sponsor
- Reading the annual report
- Chatting at lunch with a group of customer service representatives
- Reading the organization's policy on customer service, focusing particularly on the recognition and incentive aspects
- Listening to audiotapes associates with customer service complaints
- Leading a focus group with supervisors
- Interviewing some randomly drawn representatives
- Reviewing the call log
- Reading an article in a professional journal on the subject of customer service performance improvement
- Chatting at the supermarket with somebody who is a customer, who wants to tell you about her experience with customer service

We distinguish three basic steps in the performance analysis process:

- data collection,
- data transformation, and
- data visualization.

  Data collection is the process by which data about program performance are obtained from an executing program. Data are normally collected in a file, either during or after execution, although in some situations it may be presented to the user in real time.

  Three basic data collection techniques can be distinguished:

Profiles record the amount of time spent in different parts of a program. This information, though minimal, is often invaluable for highlighting performance problems. Profiles typically are gathered automatically.

Counters record either frequencies of events or cumulative times. The insertion of counters may require some programmer intervention.

Event traces record each occurrence of various specified events, thus typically producing a large amount of data. Traces can be produced either automatically or with programmer intervention.

The raw data produced by profiles, counters, or traces are rarely in the form required to answer performance questions. Hence, *data transformations* are applied, often with the goal of reducing total data volume. Transformations can be used to determine mean values or other higher-order statistics or to extract profile and counter data from traces. For example, a profile recording the time spent in each subroutine on each processor might be transformed to determine the mean time spent in each subroutine on each processor, and the standard deviation from this mean. Similarly, a trace can be processed to produce a histogram giving the distribution of message sizes. Each of the various performance tools described in subsequent sections incorporates some set of built-in transformations; more specialized transformation can also be coded by the programmer.

Parallel performance data are inherently multidimensional, consisting of execution times, communication costs, and so on, for multiple program components, on different processors, and for different problem sizes. Although data reduction techniques can be used in some situations to compress performance data to scalar values, it is often necessary to be able to explore the raw multidimensional data. As is well known in computational science and engineering, this process can benefit enormously from the use of *data visualization* techniques. Both conventional and more specialized display techniques can be applied to performance data.

As we shall see, a wide variety of data collection, transformation, and visualization tools are available. When selecting a tool for a particular task, the following issues should be considered:

Accuracy. In general, performance data obtained using sampling techniques are less accurate than data obtained by using counters or timers. In the case of timers, the accuracy of the clock must be taken into account.

Simplicity. The best tools in many circumstances are those that collect data automatically, with little or no programmer intervention, and that provide convenient analysis capabilities.

Flexibility. A flexible tool can be extended easily to collect additional performance data or to provide different views of the same data. Flexibility and simplicity are often opposing requirements.

Intrusiveness. Unless a computer provides hardware support, performance data collection inevitably introduces some overhead. We need to be aware of this overhead and account for it when analyzing data.

Abstraction. A good performance tool allows data to be examined at a level of abstraction appropriate for the programming model of the parallel program. For example, when analyzing an execution trace from a message-passing program, we probably wish to see individual messages, particularly if they can be related to send and receive statements in the source program. However, this presentation is probably *not* appropriate when studying a data-parallel program, even if compilation generates a message-passing program. Instead, we would like to see communication costs related to data-parallel program statements.

1.5. Performance Measurement:

Performance measurement is the process whereby an organization establishes the parameters within which programs, investments, and acquisitions are reaching the desired results.

Good Performance Measures:

Provide a way to see if our strategy is working

Focus employees' attention on what matters most to success

Allow measurement of accomplishments, not just of the work that is performed

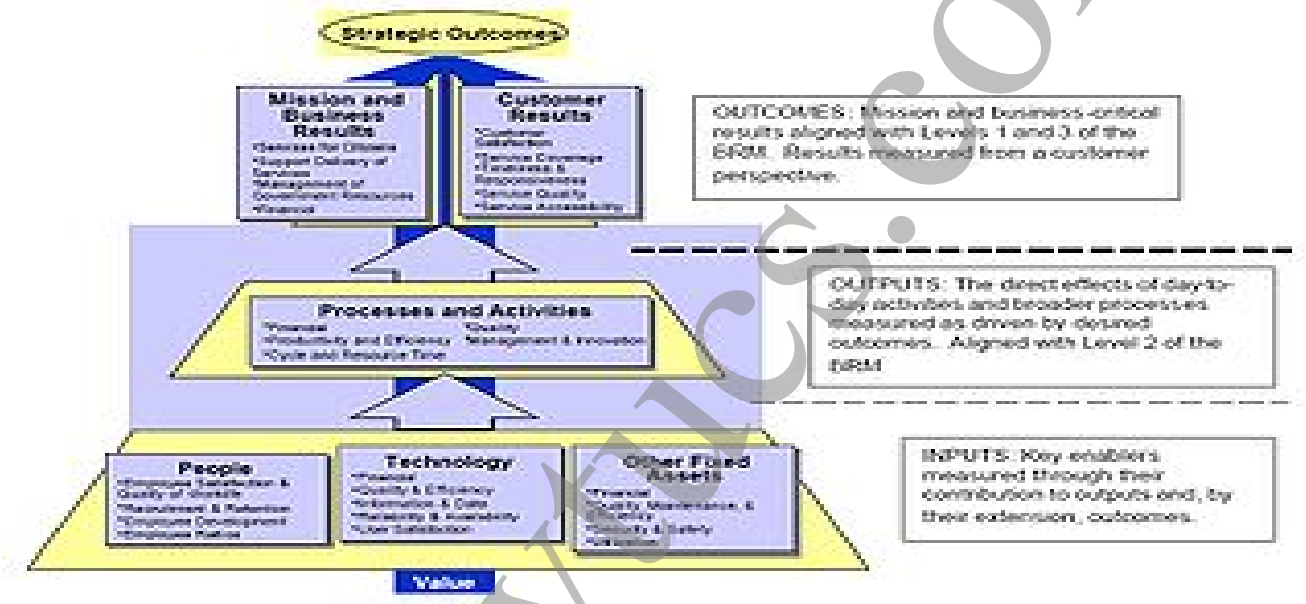
Provide a common language for communication

Are explicitly defined in terms of owner, unit of measure, collection frequency, data quality, expected value(targets), and thresholds

Are valid, to ensure measurement of the right things

Are verifiable, to ensure data collection accuracy

EX-



Performance Reference Model of the Federal Enterprise Architecture,

This process of measuring performance often requires the use of statistical evidence to determine progress toward specific defined organizational objectives. Performance measurement is a fundamental building block of TQM and a total quality organization.

Historically, organisations have always measured performance in some way through the financial performance, be this success by profit or failure through liquidation. However, traditional performance measures, based on cost accounting information, provide little to support organizations on their quality journey, because they do not map process performance and improvements seen by the customer. In a successful total quality organisation, performance will be measured by the improvements seen by the customer as well as by the results delivered to other stakeholders, such as the shareholders.

This section covers why measuring performance is important. This is followed by a description of cost of quality measurement, which has been used for many years to drive improvement activities and raise awareness of the effect of quality problems in an organisation.

A simple performance measurement framework is outlined, which includes more than just measuring, but also defining and understanding metrics, collecting and analysing data, then prioritising and taking improvement actions. A description of the balanced scorecard approach is also covered.

Why measure performance?

'When you can measure what you are speaking about and express it in numbers, you know something about it'.

'You cannot manage what you cannot measure'.

These are two often-quoted statements that demonstrate why measurement is important. Yet it is surprising that organisations find the area of measurement so difficult to manage.

In the cycle of never-ending improvement, performance measurement plays an important role in:

• Identifying and tracking progress against organisational goals

• Identifying opportunities for improvement

• Comparing performance against both internal and external standards

Reviewing the performance of an organisation is also an important step when formulating the direction of the strategic activities. It is important to know where the strengths and weaknesses of the organisation lie, and as part of the *'Plan –Do – Check – Act'* cycle, measurement plays a key role in quality and productivity improvement activities. The main reasons it is needed are:

• To ensure customer requirements *have* been met

• To be able to set sensible *objectives* and comply with them

• To provide *standards* for establishing comparisons

• To provide *visibility* and a "scoreboard" for people to *monitor* their own performance level

• To highlight *quality problems* and determine areas for *priority attention*

• To provide *feedback* for driving the improvement effort

It is also important to understand the impact of TQM on improvements in business performance, on sustaining current performance and reducing any possible decline in performance.

1.6 RECOMMENDED QUESTIONS

1. Define Data Structures?

2.What is a pointer variable?

3. Difference between Abstract Data Type, Data Type and Data Structure?

4. Define an Abstract Data Type (ADT)?

5. Give any 2 advantages and disadvantages of using pointers?

UNIT -2 : ARRAYS and STRUCTURES

2.1 ARRAY:

Definition :Array by definition is a variable that hold multiple elements which has the same data type.

Declaring Arrays :

We can declare an array by specify its data type, name and the number of elements the array holds between square brackets immediately following the array name. Here is the syntax:

1 • data_type array_name[size];

For example, to declare an integer array which contains 100 elements we can do as follows:

1 • int a[100];

There are some rules on array declaration. The data type can be any valid C data types including structure and union. The array name has to follow the rule of variable and the size of array has to be a positive constant integer.We can access array elements via indexes *array_name[index]*. Indexes of array starts from 0 not 1 so the highest elements of an array is *array_name[size-1]*

Initializing Arrays :

It is like a variable, an array can be initialized. To initialize an array, you provide initializing values which are enclosed within curly braces in the declaration and placed following an equals sign after the array name. Here is an example of initializing an integer array.

int list[5] = {2,1,3,7,8};

Array Representation

- Operations require simple implementations.
- Insert, delete, and search, require linear time
- Inefficient use of space

It is appropriate that we begin our study of data structures with the array. The array is often the only means for structuring data which is provided in a programming language. Therefore it deserves a significant amount of attention. If one asks a group of programmers to define an array, the most often quoted saying is: a *consecutive set of memory locations.* This is unfortunate because it clearly reveals a common point of confusion, namely the distinction between a data structure and its representation. It is true that arrays are almost always implemented by using consecutive memory, but not always. Intuitively, an array is a set of pairs, index and value. For each index which is defined, there is a value

associated with that index. In mathematical terms we call this a correspondence or a mapping. However, as computer scientists we want to provide a more functional definition by giving the operations which are permitted on this data structure. For arrays this means we are concerned with only two operations which retrieve and store values. Using our notation this object can be defined as:

structure *ARRAY*(*value, index*)
declare *CREATE*( ) *array*
RETRIEVE(*array,index*) *value*
STORE(*array,index,value*) *array;*
for all *A array, i,j index, x value* **let**
RETRIEVE(*CREATE*,*i*) :: = **error**
RETRIEVE(*STORE*(*A,i,x*),*j*) :: =
if *EQUAL*(*i,j*) **then** *x* **else** *RETRIEVE*(*A,j*)
end
end *ARRAY*

The function CREATE produces a new, empty array. RETRIEVE takes as input an array and an index, and either returns the appropriate value or an error. STORE is used to enter new index-value pairs. The second axiom is read as "to retrieve the *j*-th item where *x* has already been stored at index *i* in *A* is equivalent to checking if *i* and *j* are equal and if so, *x,* or search for the *j*-th value in the remaining array, *A*." This axiom was originally given by J. McCarthy. Notice how the axioms are independent of any representation scheme. Also, *i* and *j* need not necessarily be integers, but we assume only that an EQUAL function can be devised.

If we restrict the index values to be integers, then assuming a conventional random access memory we can implement STORE and RETRIEVE so that they operate in a constant amount of time. If we interpret the indices to be *n*-dimensional, (i_1,i_2, ...,i_n), then the previous axioms apply immediately and define *n*-dimensional arrays. In section 2.4 we will examine how to implement RETRIEVE and STORE for multi-dimensional arrays using consecutive memory locations.

Array and Pointer:

Each array element occupies consecutive memory locations and array name is a pointer that points to the first element. Beside accessing array via index we can use pointer to manipulate array. This program helps you visualize the memory address each array elements and how to access array element using pointer.

```
01 #include <stdio.h>

02

03 void main()

04 {

05

06     const int size = 5;
```

```
07
08    int list[size] = {2,1,3,7,8};
09
10    int* plist = list;
11
12    // print memory address of array elements
13    for(int i = 0; i < size;i++)
14    {
15       printf("list[%d] is in %d\n",i,&list[i]);
16
17    }
18
19    // accessing array elements using pointer
20    for(i = 0; i < size;i++)
21    {
22       printf("list[%d] = %d\n",i,*plist);
23
24       /* increase memory address of pointer so it go to the next
25          element of the array */
26       plist++;
27    }
28
29 }
```

Here is the output

```
list[0]  is  in  1310568
list[1]  is  in  1310572
list[2]  is  in  1310576
list[3]  is  in  1310580
list[4]  is  in  1310584
list[0]  =  2
list[1]  =  1
list[2]  =  3
list[3]  =  7
list[4] = 8
```

You can store pointers in an array and in this case we have an array of pointers. This code snippet uses an array to store integer pointer.

```
1 int *ap[10];
```

Multidimensional Arrays:

An array with more than one index value is called a multidimensional array. The entire array above is called single-dimensional array. To declare a multidimensional array you can do follow syntax

```
1 data_type array_name[][][];
```

The number of square brackets specifies the dimension of the array. For example to declare two dimensions integer array we can do as follows:

```
1 int matrix[3][3];
```

Initializing Multidimensional Arrays :

You can initialize an array as a single-dimension array. Here is an example of initialize an two dimensions integer array:

```
1 int matrix[3][3] =
2 {
3   {11,12,13},
4   {21,22,23},
5   {32,31,33},
6 };
```

Dynamically Allocated Arrays: One-Dimensional Arrays

In C, arrays must have their extents defined at compile-time. There's no way to postpone the definition of the size of an array until runtime. Luckily, with pointers and malloc, we can work around this limitation.

To allocate a one-dimensional array of length *N* of some particular type, simply use malloc to allocate enough memory to hold *N* elements of the particular type, and then use the resulting pointer as if it were an array. For example, the following code snippet allocates a block of *N* ints, and then, using array notation, fills it with the values 0 through *N*-1:

```
int *A = malloc (sizeof (int) * N);
int i;


for (i = 0; i < N; i++)
   A[i] = i;
```

This idea is very useful for dealing with strings, which in C are represented by arrays of chars, terminated with a '\0' character. These arrays are nearly always expressed as pointers in the declaration of functions, but accessed via C's array notation. For example, here is a function that implements strlen:

```
int strlen (char *s)
{
   int i;


   for (i = 0; s[i] != '\0'; i++)
      ;
   return (i)
}
```

2.2. Dynamically Allocating Multidimensional Arrays

We've seen that it's straightforward to call malloc to allocate a block of memory which can simulate an array, but with a size which we get to pick at run-time. Can we do the same sort of thing to simulate multidimensional arrays? We can, but we'll end up using pointers to pointers. If we don't know how many columns the array will have, we'll clearly allocate memory for each row (as many columns wide as we like) by calling malloc, and each row will therefore be represented by a pointer. How will we keep track of those pointers? There are, after all, many of them, one for each row. So we want to simulate an array of pointers, but we don't know how many rows there will be, either, so we'll have to simulate that array (of pointers) with another pointer, and this will be a pointer to a pointer.

This is best illustrated with an example:

```
#include <stdlib.h>
int **array;
array = malloc(nrows * sizeof(int *));
```

```
if(array == NULL)
        {
        fprintf(stderr, "out of memory\n");
        exit or return
        }
for(i = 0; i < nrows; i++)
        {
        array[i] = malloc(ncolumns * sizeof(int));
        if(array[i] == NULL)
                {
                fprintf(stderr, "out of memory\n");
                exit or return
                }
        }
```

array is a pointer-to-pointer-to-int: at the first level, it points to a block of pointers, one for each row. That first-level pointer is the first one we allocate; it has n rows elements, with each element big enough to hold a pointer-to-int, or int *. If we successfully allocate it, we then fill in the pointers (all n rows of them) with a pointer (also obtained from malloc) to n columns number of ints, the storage for that row of the array. If this isn't quite making sense, a picture should make everything clear:

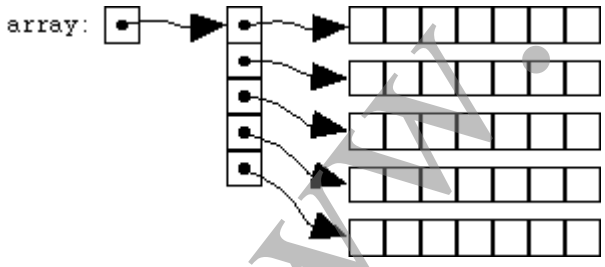


Fig1: representation of array

Once we've done this, we can (just as for the one-dimensional case) use array-like syntax to access our simulated multidimensional array. If we write

```
array[i][j]
```

we're asking for the i'th pointer pointed to by array, and then for the j'th int pointed to by that inner pointer. (This is a pretty nice result: although some completely different machinery, involving two levels of pointer dereferencing, is going on behind the scenes, the simulated, dynamically-allocated two-dimensional ``array'' can still be accessed just as if it were an array of arrays, i.e. with the same pair of bracketed subscripts.). If a program uses simulated, dynamically allocated multidimensional arrays, it

becomes possible to write ``heterogeneous'' functions which *don't* have to know (at compile time) how big the ``arrays'' are. In other words, one function can operate on ``arrays'' of various sizes and shapes. The function will look something like

```
func2(int **array, int nrows, int ncolumns)
{
}
```

This function does accept a pointer-to-pointer-to-int, on the assumption that we'll only be calling it with simulated, dynamically allocated multidimensional arrays. (We must not call this function on arrays like the ``true'' multidimensional array a2 of the previous sections). The function also accepts the dimensions of the arrays as parameters, so that it will know how many ``rows'' and ``columns'' there are, so that it can iterate over them correctly. Here is a function which zeros out a pointer-to-pointer, two-dimensional ``array'':

```
void zeroit(int **array, int nrows, int ncolumns)
{
int i, j;
for(i = 0; i < nrows; i++)
        {
        for(j = 0; j < ncolumns; j++)
                array[i][j] = 0;
        }
}
```

Finally, when it comes time to free one of these dynamically allocated multidimensional ``arrays,'' we must remember to free each of the chunks of memory that we've allocated. (Just freeing the top-level pointer, array, wouldn't cut it; if we did, all the second-level pointers would be lost but not freed, and would waste memory.) Here's what the code might look like:

```
for(i = 0; i < nrows; i++)
        free(array[i]);
free(array);
```

2.3. Structures and Unions:

Structure:

A structure is a user-defined data type. You have the ability to define a new type of data considerably more complex than the types we have been using. A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling. Structures are called “records” in some languages, notably Pascal. Structures help organize complicated data,

particularly in large programs, because they permit a group of related variables to be treated as a unit instead of as separate entities.

Today's application requires complex data structures to support them. A structure is a collection of related elements where element belongs to a different type. Another way to look at a structure is a template – a pattern. For example graphical user interface found in a window requires structures typical example for the use of structures could be the file table which holds the key data like logical file name, location of the file on disc and so on. The file table in C is a type defined structure - FILE. Each element of a structure is also called as field. A field has a many characteristic similar to that of a normal variable. An array and a structure are slightly different. The former has a collection of homogeneous elements but the latter has a collection of heterogeneous elements. The general format for a structure in C is shown

s*truct* {field_list} variable_identifier;

struct struct_name

{

type1 fieldname1;

type2 fieldname2;

.

.

.

typeN fieldnameN;

};

struct struct_name variables;

The above format shown is not concrete and can vary, so different `

flavours of structure declaration is as shown.

struct

{

….

} variable_identifer;

Example

struct mob_equip;

{

long int IMEI;

char rel_date[10];

```
char model[10];
char brand[15];
};
```

The above example can be upgraded with *typedef*. A program to illustrate the working of the structure is shown in the previous section.

```
typedef struct mob_equip;
{
long int IMEI;
char rel_date[10];
char model[10];
char brand[15];
int count;
} MOB; MOB m1;
struct tag
{
…….
};
struct tag variable_identifers;
typedef struct
{
……..
} TYPE_IDENITIFIER;
TYPE_IDENTIFIER variable_identifers;
```

Accessing a structure

A structure variable or a tag name of a structure can be used to access the members of a structure with the help of a special operator '.' –also called as member operator . In our previous example To access the idea of the IMEI of the mobile equipment in the structure mob_equip is done like this Since the structure variable can be treated as a normal variable All the IO functions for a normal variable holds good for the structure variable also with slight. The scanf statement to read the input to the IMEI is given below

```
scanf ("%d",&m1.IMEI);
```

Increment and decrement operation are same as the normal variables this includes postfix and prefix also. Member operator has more precedence than the increment or decrement. Say suppose in example quoted earlier we want count of student then

m1.count++; ++m1.count

Unions

Unions are very similar to structures, whatever discussed so far holds good for unions also then why do we need unions? Size of unions depends on the size of its member of largest type or member with largest size, but this is not son in case of structures.

Example *union* abc1

{

int a;

float b;

char c;

};

The size of the union abc1 is 4bytes as float is largest type. Therefore at any point of time we can access only one member of the union and this needs to remembered by programmer.

Using structure data

Now that we have assigned values to the six simple variables, we can do anything we desire with them. In order to keep this first example simple, we will simply print out the values to see if they really do exist as assigned. If you carefully inspect the printf statements, you will see that there is nothing special about them. The compound name of each variable is specified because that is the only valid name by which we can refer to these variables. Structures are a very useful method of grouping data together in order to make a program easier to write and understand.

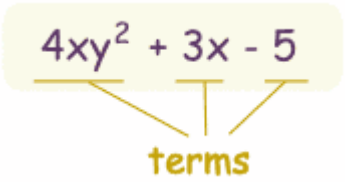
2.4 .Polynomials:

In mathematics, a polynomial (from Greek *poly*, "many" and medieval Latin *binomium*, "binomial"[1] [2] [3]) is an expression of finite length constructed from variables (also known as indeterminates) and constants, using only the operations of addition, subtraction, multiplication, and non-negative integer exponents. For example, $x^2 - 4x + 7$ is a polynomial, but $x^2 - 4/x + 7x^{3/2}$ is not, because its second term involves division by the variable x (4/x) and because its third term contains an exponent that is not a whole number (3/2). The term "polynomial" can also be used as an adjective, for quantities that can be expressed as a polynomial of some parameter, as in "polynomial time" which is used in computational complexity theory.

Polynomials appear in a wide variety of areas of mathematics and science. For example, they are used to form polynomial equations, which encode a wide range of problems, from elementary word problems to complicated problems in the sciences; they are used to define polynomial functions, which appear in settings ranging from basic chemistry and physics to economics and social science; they are used in calculus and numerical analysis to approximate other functions. In advanced mathematics, polynomials are used to construct polynomial rings, a central concept in abstract algebra and algebraic geometry.

A polynomial is made up of terms that are only added, subtracted or multiplied.

A polynomial looks like this:



example of a polynomial
this one has 3 terms

Fig 2:

Polynomial comes form *poly-* (meaning "many") and *-nomial* (in this case meaning "term") ... so it says "many terms"

A polynomial can have:

constants (like 3, -20, or ½)

variables (like *x* and *y*)

exponents (like the 2 in y^2) but only 0, 1, 2, 3, ... etc

That can be combined using:

\+ addition,

\- subtraction, and

× Multiplication

✘ ... but not division! ✘

Those rules keeps polynomials simple, so they are easy to work with!

Polynomial or Not?

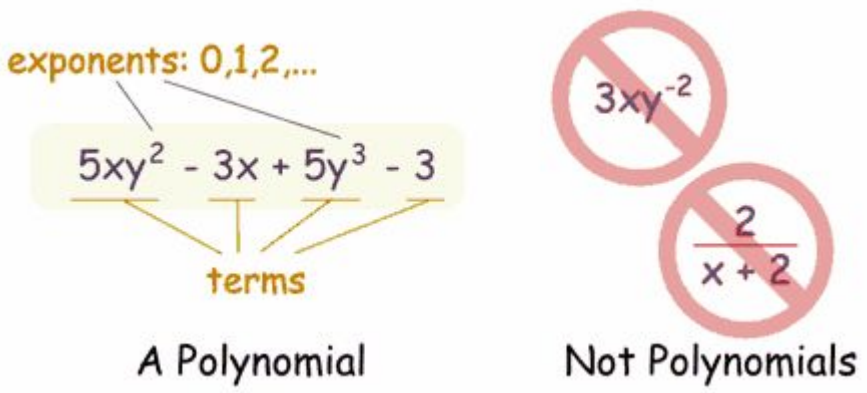


Fig 3:

These are polynomials:

3x

x - 2

$-6y^2 - (^7/_9)\mathbf{x}$

$3xyz + 3xy^2z - 0.1xz - 200y + 0.5$

$512v^5 + 99w^5$

1

(Yes, even "1" is a polynomial, it has one term which just happens to be a constant).

And these are not polynomials

2/(x+2) is not, because dividing is not allowed

1/x is not

$3xy^{-2}$ is not, because the exponent is "-2" (exponents can only be 0,1,2,...)

√x is not, because the exponent is "½" (see fractional exponents)

But these are allowed:

x/2 is allowed, because it is also (½)x (the constant is ½, or 0.5)

also 3x/8 for the same reason (the constant is 3/8, or 0.375)

√2 is allowed, because it is a constant (= 1.4142...etc)

Monomial, Binomial, Trinomial

There are special names for polynomials with 1, 2 or 3 terms:

$3xy^2$	$5x - 1$	$3x + 5y^2 - 3$
Monomial (1 term)	Binomial (2 terms)	Trinomial (3 terms)

How do you remember the names? Think cycles!

Fig 4:(There is also quadrinomial (4 terms) and quintinomial (5 terms), but those names are not often used)

2.5. Sparse Matrices:

A sparse matrix is a matrix that allows special techniques to take advantage of the large number of zero elements. This definition helps to define "how many" zeros a matrix needs in order to be "sparse." The answer is that it depends on what the structure of the matrix is, and what you want to do with it. For example, a randomly generated sparse $n \times n$ matrix with cn entries scattered randomly throughout the matrix is not sparse in the sense of Wilkinson.
Next: Advanced Graphics Up: Sparse matrix computations Previous: Sparse matrix computations

Creating a sparse matrix:

If a matrix A is stored in ordinary (dense) format, then the command S = sparse(A) creates a copy of the matrix stored in sparse format. For example:

```
>> A = [0 0 1;1 0 2;0 -3 0]

A =

     0     0     1
     1     0     2
     0    -3     0

>> S = sparse(A)

S =

   (2,1)        1
   (3,2)       -3
```

(1,3) 1

(2,3) 2

>> whos

Name Size Bytes Class

A 3x3 72 double array

S 3x3 64 sparse array

Grand total is 13 elements using 136 bytes

Unfortunately, this form of the sparse command is not particularly useful, since if A is large, it can be very time-consuming to first create it in dense format. The command S = sparse(m,n) creates an $m \times n$ zero matrix in sparse format. Entries can then be added one-by-one:

>> A = sparse(3,2)

A =

All zero sparse: 3-by-2

>> A(1,2)=1;

>> A(3,1)=4;

>> A(3,2)=-1;

>> A

A =

(3,1) 4

(1,2) 1

(3,2) -1

(Of course, for this to be truly useful, the nonzeros would be added in a loop.)

Another version of the sparse command is S = sparse(I,J,S,m,n,maxnz). This creates an $m \times n$ sparse matrix with entry $(I(k),J(k))$ equal to $S(k), k = 1, \ldots, length(S)$. The optional argument maxnz causes Matlab to pre-allocate storage for maxnz nonzero entries, which can increase efficiency in the case when more nonzeros will be added later to S.

The most common type of sparse matrix is a banded matrix, that is, a matrix with a few nonzero diagonals. Such a matrix can be created with the spdiags command. Consider the following matrix:

>> A

A =

```
 64  -16    0  -16    0    0    0    0    0
-16   64  -16    0  -16    0    0    0    0
  0  -16   64    0    0  -16    0    0    0
-16    0    0   64  -16    0  -16    0    0
  0  -16    0  -16   64  -16    0  -16    0
  0    0  -16    0  -16   64    0    0  -16
  0    0    0  -16    0    0   64  -16    0
  0    0    0    0  -16    0  -16   64  -16
  0    0    0    0    0  -16    0  -16   64
```

This is a 9×9 matrix with 5 nonzero diagonals. In Matlab's indexing scheme, the nonzero diagonals of A are numbers -3, -1, 0, 1, and 3 (the main diagonal is number 0, the first subdiagonal is number -1, the first superdiagonal is number 1, and so forth). To create the same matrix in sparse format, it is first necessary to create a 9×5 matrix containing the nonzero diagonals of A. Of course, the diagonals, regarded as column vectors, have different lengths; only the main diagonal has length 9. In order to gather the various diagonals in a single matrix, the shorter diagonals must be padded with zeros. The rule is that the extra zeros go at the bottom for subdiagonals and at the top for superdiagonals. Thus we create the following matrix:

```
>> B = [
  -16  -16   64    0    0
  -16  -16   64  -16    0
  -16    0   64  -16    0
  -16  -16   64    0  -16
  -16  -16   64  -16  -16
  -16    0   64  -16  -16
    0  -16   64    0  -16
    0  -16   64  -16  -16
    0    0   64  -16  -16
];
```

(notice the technique for entering the rows of a large matrix on several lines). The spdiags command also needs the indices of the diagonals:

```
>> d = [-3,-1,0,1,3];
```

The matrix is then created as follows:

S = spdiags(B,d,9,9);

The last two arguments give the size of S.

Perhaps the most common sparse matrix is the identity. Recall that an identity matrix can be created, in dense format, using the command eye. To create the $n \times n$ identity matrix in sparse format, use I = speye(n).

Another useful command is spy, which creates a graphic displaying the sparsity pattern of a matrix. For example, the above penta-diagonal matrix A can be displayed by the following command; see Figure 6:

>> spy(A)

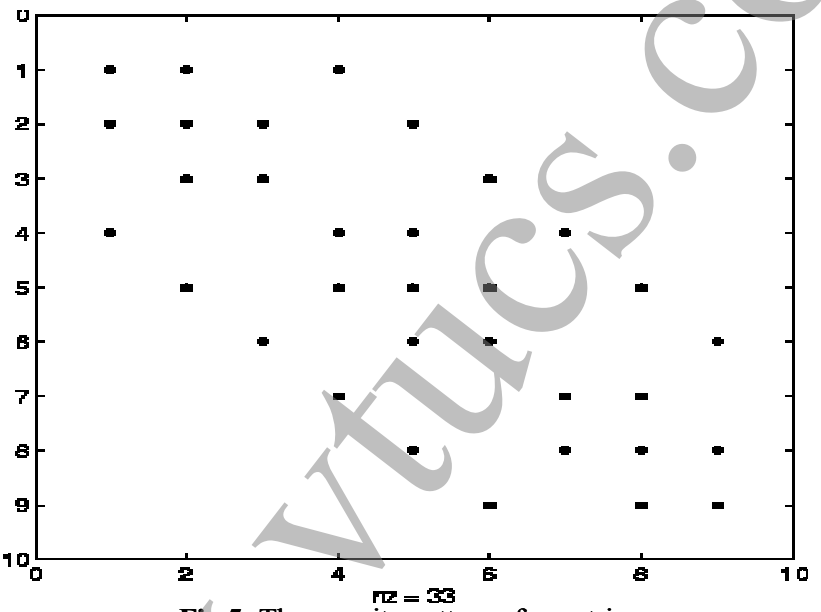


Fig 5: The sparsity pattern of a matrix

2.6. Representation of Multidimensional arrays

For a two-dimensional array, the element with indices *i*,*j* would have address $B + c \cdot i + d \cdot j$, where the coefficients *c* and *d* are the **row** and **column address increments**, respectively.
More generally, in a *k*-dimensional array, the address of an element with indices *i*1, *i*2, …, *ik* is
$B + c1 \cdot i1 + c2 \cdot i2 + \ldots + ck \cdot ik$
This formula requires only *k* multiplications and *k*−1 additions, for any array that can fit in memory. Moreover, if any coefficient is a fixed power of 2, the multiplication can be replaced by bit shifting.
The coefficients *ck* must be chosen so that every valid index tuple maps to the address of a distinct element. If the minimum legal value for every index is 0, then *B* is the address of the element whose indices are all zero. As in the one-dimensional case, the element indices may be changed by changing the base address *B*. Thus, if a two-dimensional array has rows and columns indexed from 1 to 10 and 1 to 20, respectively, then replacing *B* by $B + c1 - -3\ c1$ will cause them to be renumbered from 0 through 9 and 4 through 23, respectively. Taking advantage of this feature, some languages (like FORTRAN 77) specify that array indices begin at 1, as in mathematical tradition; while other languages (like Fortran 90, Pascal and Algol) let the user choose the minimum value for each index.

Compact layouts

Often the coefficients are chosen so that the elements occupy a contiguous area of memory. However, that is not necessary. Even if arrays are always created with contiguous elements, some array slicing operations may create non-contiguous sub-arrays from them.

There are two systematic compact layouts for a two-dimensional array. For example, consider the matrix

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}.$$

In the row-major order layout (adopted by C for statically declared arrays), the elements of each row are stored in consecutive positions:

1	2	3	4	5	6	7	8	9

In Column-major order (traditionally used by Fortran), the elements of each column are consecutive in memory:

1	4	7	2	5	8	3	6	9

For arrays with three or more indices, "row major order" puts in consecutive positions any two elements whose index tuples differ only by one in the *last* index. "Column major order" is analogous with respect to the *first* index. In systems which use processor cache or virtual memory, scanning an array is much faster if successive elements are stored in consecutive positions in memory, rather than sparsely scattered. Many algorithms that use multidimensional arrays will scan them in a predictable order. A programmer (or a sophisticated compiler) may use this information to choose between row- or column-major layout for each array. For example, when computing the product *A·B* of two matrices, it would be best to have *A* stored in row-major order, and *B* in column-major order.

The Representation of Multidimensional Arrays:

N-dimension, A[M_0][M_2]. . .[M_{n-1}]

- **Address of any entry A[i_0][i_1]...[i_{n-1}]**

$$\begin{aligned} & base + i_0 M_1 M_2 \quad M_{n-1} \\ & \quad + i_1 M_2 M_3 \quad M_{n-1} \\ & \quad + i_2 M_3 M_4 \quad M_{n-1} \\ & \quad + i_{n-2} M_{n-1} \\ & \quad + i_{n-1} \\ & = \text{base} + \sum_{j=0}^{n-1} \text{i}_j a_j, \quad where\{ \begin{matrix} a_j = \Pi_{k=j+1}^{n-1}, 0 \le j < n-1 \\ a_{n-1} = 1 \end{matrix} \end{aligned}$$

Array resizing:

Static arrays have a size that is fixed at allocation time and consequently do not allow elements to be inserted or removed. However, by allocating a new array and copying the contents of the old array to it, it

is possible to effectively implement a *dynamic* or *growable* version of an array; see dynamic array. If this operation is done infrequently, insertions at the end of the array require only amortized constant time.
Some array data structures do not reallocate storage, but do store a count of the number of elements of the array in use, called the count or size. This effectively makes the array a dynamic array with a fixed maximum size or capacity; *Pascal strings* are examples of this.

Non-linear formulas

More complicated ("non-linear") formulas are occasionally used. For a compact two-dimensional triangular array, for instance, the addressing formula is a polynomial of degree 2.

Efficiency

Both *store* and *select* take (deterministic worst case) constant time. Arrays take linear (O(*n*)) space in the number of elements *n* that they hold. In an array with element size *k* and on a machine with a cache line size of B bytes, iterating through an array of *n* elements requires the minimum of ceiling(*nk*/B) cache misses, because its elements occupy contiguous memory locations. This is roughly a factor of B/*k* better than the number of cache misses needed to access *n* elements at random memory locations. As a consequence, sequential iteration over an array is noticeably faster in practice than iteration over many other data structures, a property called locality of reference (this does *not* mean however, that using a perfect hash or trivial hash within the same (local) array, will not be even faster - and achievable in constant time).
Memory-wise, arrays are compact data structures with no per-element overhead. There may be a per-array overhead, e.g. to store index bounds, but this is language-dependent. It can also happen that elements stored in an array require *less* memory than the same elements stored in individual variables, because several array elements can be stored in a single word; such arrays are often called **packed** arrays. An extreme (but commonly used) case is the bit array, where every bit represents a single element.

Linked lists allow constant time removal and insertion in the middle but take linear time for indexed access. Their memory use is typically worse than arrays, but is still linear. An alternative to a multidimensional array structure is to use a one-dimensional array of references to arrays of one dimension less.

For two dimensions, in particular, this alternative structure would be a vector of pointers to vectors, one for each row. Thus an element in row *i* and column *j* of an array *A* would be accessed by double indexing (*A*[*i*][*j*] in typical notation). This alternative structure allows *ragged* or *jagged* arrays, where each row may have a different size — or, in general, where the valid range of each index depends on the values of all preceding indices. It also saves one multiplication (by the column address increment) replacing it by a bit shift (to index the vector of row pointers) and one extra memory access (fetching the row address), which may be worthwhile in some architectures.

2.7. RECOMMENDED QUESTIONS

1. How does a structure differ from an union? Mention any 2 uses of structure

2. Explain the single-dimensional array.

3. Explain the declaration & initialization of 2-D array.

4. How transposing of a matrix done.

5. Explain the representation of multidimensional array.

6. Explain the sparse matrix representation.

UNIT – 3 : STACKS AND QUEUES

3.1.Stacks:

A ***stack*** is an ordered collection of items into which new items may be inserted and from which items may be deleted at one end, called the ***top*** of the stack. A stack is a dynamic, constantly changing object as the definition of the stack provides for the insertion and deletion of items. It has single end of the stack as top of the stack, where both insertion and deletion of the elements takes place. The last element inserted into the stack is the first element deleted-***last in first out list (LIFO).*** After several insertions and deletions, it is possible to have the same frame again.

Primitive Operations

When an item is added to a stack, it is ***push***ed onto the stack. When an item is removed, it is ***pop***ped from the stack.

Given a stack *s*, and an item *i*, performing the operation *push(s,i)* adds an item *i* to the top of stack *s.*

push(s, H);

push(s, I);

push(s, J);

Operation *pop(s)* removes the top element. That is, if *i=pop(s)*, then the removed element is assigned to *i.*

pop(s);

Because of the push operation which adds elements to a stack, a stack is sometimes called a ***pushdown list.*** Conceptually, there is no upper limit on the number of items that may be kept in a stack. If a stack contains a single item and the stack is popped, the resulting stack contains no items and is called the ***empty stack.*** Push operation is applicable to any stack. Pop operation cannot be applied to the empty stack. If so, *underflow* happens. A Boolean operation *empty(s)*, returns TRUE if stack is empty. Otherwise FALSE, if stack is not empty.

Representing stacks in C

Before programming a problem solution that uses a stack, we must decide how to represent the stack in a programming language. It is an ordered collection of items. In C, we have ARRAY as an ordered collection of items. But a stack and an array are two different things. The number of elements in an array is fixed. A stack is a dynamic object whose size is constantly changing. So, an array can be declared large enough for the maximum size of the stack. A stack in C is declared as a structure containing two objects:

• An array to hold the elements of the stack.

• An integer to indicate the position of the current stack top within the array.

```
#define STACKSIZE 100

struct stack {

int top;
```

```
int items[STACKSIZE];
};
```

The stack *s* may be declared by

```
struct stack s;
```

The stack items may be int, float, char, etc. The empty stack contains no elements and can therefore be indicated by *top*= -1. To initialize a stack S to the empty state, we may initially execute

s.top= -1.

To determine stack empty condition,

```
if (s.top=-1)
stack empty;
else
stack is not empty;
```

The empty(s) may be considered as follows**:**

```
int empty(struct stack *ps)
{
if(ps->top== -1)
return(TRUE);
else
return(FALSE);
}
```

Aggregating the set of implementation-dependent trouble spots into small, easily identifiable units is an important method of making a program more understandable and modifiable. This concept is known as ***modularization***, in which individual functions are isolated into low-level modules whose properties are easily verifiable. These low-level ***modules*** can then be used by more complex routines, which do not have to concern themselves with the details of the low-level modules but only with their function. The complex routines may themselves then be viewed as modules by still higher-level routines that use them independently of their internal details.

• Implementing pop operation

If the stack is empty, print a warning message and halt execution. Remove the top element from the stack. Return this element to the calling program

```
int pop(struct stack *ps)
{
```

```
if(empty(ps)){
printf("%","stack underflow");
exit(1);
}
return(ps->items[ps->top--]);
}
```

3.2. Stacks Using Dynamic Arrays:

For example:

```
Typedef struct

    {
char        *str;
    }       words;

    main()
    {
 words x[100]; // I do not want to use this, I want to dynamic increase the size of the array as data comesin.
}
```

For example here is the following array in which i read individual words from a .txt file and save them word by word in the array:

Code:

```
char words[1000][15];
```

Here 1000 defines the number of words the array can save and each word may comprise of not more than 15 characters. Now I want that that program should dynamically allocate the memory for the number of words it counts. For example, a .txt file may contain words greater that 1000. Now I want that the program should count the number of words and allocate the memory accordingly. Since we cannot use a variable in place of [1000], I am completely blank at how to implement my logic. Please help me in this regard.

3.3. Queues:

A queue is like a line of people waiting for a bank teller. The queue has a **front** and a **rear**.

When we talk of queues we talk about two distinct ends: the front and the rear. Additions to the queue take place at the rear. Deletions are made from the front. So, if a job is submitted for execution, it joins at the rear of the job queue. The job at the front of the queue is the next one to be executed

• New people must enter the queue at the rear. **push**, although it is usually called an **enqueue** operation.

• When an item is taken from the queue, it always comes from the front. **pop**, although it is usually called a **dequeue** operation.

What is Queue?

• Ordered collection of elements that has two ends as front and rear.

• Delete from front end

• Insert from rear end

• A queue can be implemented with an array, as shown here. For example, this queue contains the integers 4 (at the front), 8 and 6 (at the rear).

Queue Operations

• Queue Overflow

• Insertion of the element into the queue

• Queue underflow

• Deletion of the element from the queue

• Display of the queue

```
struct Queue {
int que [size];
int front;
int rear;
}Q;
```

Example:

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define size 5
struct queue {
int que[size];
int front, rear;
} Q;
```

Example:

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define size 5
struct queue {
int que[size];
int front, rear;
} Q;
int Qfull ( ){
if (Q.rear >= size-1)
return 1;
else
return 0;
}
int Qempty( ){
if ((Q.front == -1)||(Q.front > Q.rear))
return 1;
else
return 0;
}
int insert (int item) {
if (Q.front == -1)
Q.front++;
Q.que[++Q.rear] = item;
return Q.rear;
}
Int delete () {
Int item;
```

```
Item = Q.que[Q.front];
Q.front++;
Return Q.front;
}
Void display () {
Int I;
For (i=Q.front;i<=Q.rear;i++)
Printf(" %d",Q.que[i]);
}
Void main (void) {
Int choice, item;
Q.front = -1; Q.Rear = -1;
do {
Printf("Enter your choice : 1:I, 2:D, 3:Display");
Scanf("%d", &choice);
Switch(choice){
Case 1: if(Qfull()) printf("Cannt Insert");
else scanf("%d",item); insert(item); break;
Case 2: if(Qempty()) printf("Underflow");
else delete(); break;
}
}
}
```

3.4. Circular Queues Using Dynamic Arrays:

Circular Queue

• When an element moves past the end of a circular array, it wraps around to the beginning.

A more efficient queue representation is obtained by regarding the array $Q(1:n)$ as circular. It now becomes more convenient to declare the array as $Q(0:n - 1)$. When rear = $n - 1$, the next element is entered at $Q(0)$ in case that spot is free. Using the same conventions as before, front

will always point one position counterclockwise from the first element in the queue. Again, *front* = *rear* if and only if the queue is empty. Initially we have *front* = *rear* = 1. Figure 3.4 illustrates some of the possible configurations for a circular queue containing the four elements *J*1-*J*4 with *n* > 4. The assumption of circularity changes the ADD and DELETE algorithms slightly. In order to add an element, it will be necessary to move *rear* one position clockwise, i.e.,

if *rear* = *n* - 1 **then** *rear* 0
else *rear* *rear* + 1.

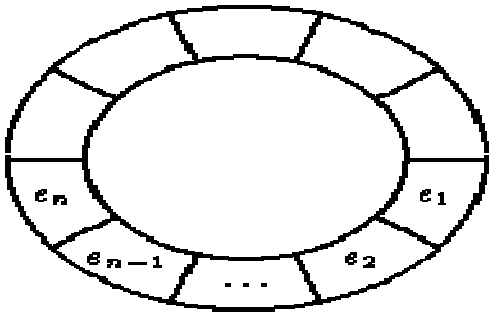


Figure : Circular queue of n elements

Using the modulo operator which computes remainders, this is just *rear (*rear + *1)**mod* n. *Similarly, it will be necessary to move* front *one position clockwise each time a deletion is made. Again, using the modulo operation, this can be accomplished by* front *(*front + *l)**mod* n. *An examination of the algorithms indicates that addition and deletion can now be carried out in a fixed amount of time or* O*(1).*

e.g.

• OOOOO7963 _ 4OOOO7963 (after Enqueue(4))

• After Enqueue(4), rear index moves from 3 to 4.

Queue Full Condition:

if(front == (rear+1)%size) Queue is Full

• Where do we insert:

rear = (rear + 1)%size; queue[rear]=item;

After deletion : front = (front+1)%size;

Example of a Circular Queue

• A Circular Q, the size of which is 5 has three elements 20, 40, and 60 where front is 0

and rear is 2. What are the values of after each of these operations:

Q = 20, 40, 60, - , - front–20[0], rear–60[2]

Insert item 50:

Q = 20, 40, 60, 50, - front-20[0], rear-50[3]

Insert item 10:

Q = 20, 40, 60, 50, 10 front-20[0], rear-10[4]

Q = 20, 40, 60, 50, 10 front-20[0], rear-10[4]

Insert 30

Rear = (rear + 1)%size = (4+1)%5 = 0, hence overflow.

Delete an item

delete 20, front = (front+1)%size = (0+1)%5=1

Delete an item

delete 40, front = (front+1)%size = (1+1)&5=2

Insert 30 at position 0

Rear = (rear + 1)%size = (4+1)%5 = 0

Similarly Insert 80 at position 1

3.5. Evaluation of Expressions: Evaluating a postfix expression:

When pioneering computer scientists conceived the idea of higher level programming languages, they were faced with many technical hurdles. One of the biggest was the question of how to generate machine language instructions which would properly evaluate any arithmetic expression. A complex assignment statement such as

*X A/B ** C + D* E - A * C* (3.1)

might have several meanings; and even if it were uniquely defined, say by a full use of parentheses, it still seemed a formidable task to generate a correct and reasonable instruction sequence. Fortunately the solution we have today is both elegant and simple. Moreover, it is so simple that this aspect of compiler writing is really one of the more minor issues.

An expression is made up of operands, operators and delimiters. The expression above has five operands: *A*,*B*,*C*,*D*, and *E*. Though these are all one letter variables, operands can be any legal variable name or constant in our programming language. In any expression the values that variables take must be consistent with the operations performed on them. These operations are described by the operators. In most programming languages there are several kinds of operators which correspond to the different kinds of data a variable can hold. First, there are the basic arithmetic operators: plus, minus, times, divide, and exponentiation (+,-,*,/,**). Other arithmetic operators include unary plus, unary minus and **mod, ceil**, and **floor**. The latter three may sometimes be library subroutines rather than predefined operators. A second class are the relational operators: . These are usually defined to work for arithmetic operands, but they can just as easily work for character string data. ('CAT' is less than 'DOG' since it precedes 'DOG' in alphabetical order.) The result of an expression which contains relational operators is one of the two

constants: **true** or **false**. Such all expression is called Boolean, named after the mathematician George Boole, the father of symbolic logic.

The first problem with understanding the meaning of an expression is to decide in what order the operations are carried out. This means that every language must uniquely define such an order. For instance, if $A = 4$, $B = C = 2$, $D = E = 3$, then in eq. 3.1 we might want X to be assigned the value

4/(2 ** 2) + (3 * 3) - (4 * 2)
= (4/4) + 9 - 8
= 2.
Let us now consider an example. Suppose that we are asked to evaluate the following postfix expression:

6 2 3 + - 3 8 2 / + * 2 $ 3 +

Symb Opnd1 Opnd2 Value opndstk

6 6

2 6,2

3 6,2,3

+ 2 3 5 6,5

- 6 5 1 1

3 6 5 1 1,3

8 6 5 1 1,3,8

2 6 5 1 1,3,8,2

/ 8 2 4 1,3,4

8

+ 3 4 7 1,7

* 1 7 7 7

2 1 7 7 7,2

$ 7 2 49 49

3 7 2 49 49,3

+ 49 3 52 **52**

Each time we read an operand, we push it onto a stack. When we reach an operator, its operands will be the top two elements on the stack. We can then pop these two elements, perform the indicated operation on them, and push the result on the stack so that it will be available for use as an operand of the next operator. The maximum size of the stack is the number of operands that appear in the input expression. But usually, the actual size of the stack needed is less than maximum, as operator pops the top two operands.

Program to evaluate postfix expression

Along with push, pop, empty operations, we have ***eval, isdigit*** and ***oper*** operations.

eval – the evaluation algorithm

```
double eval(char expr[])
{
int c, position;
double opnd1, opnd2, value;
struct stack opndstk;
opndstk.top=-1;
for (position=0 ;( c=expr [position])!='\0'; position++)
if (isdigit)
push (&opndstk, (double) (c-'0'));
else{
opnd2=pop (&opndstk);
9
opnd1=pop (&opndstk);
value=oper(c, opnd1,opnd2);
push (&opndstk. value);
}
return(pop(&opndstk));
}
```

isdigit – called by eval, to determine whether or not its argument is an operand

```
int isdigit(char symb)
{
return(symb>='0' && symb<='9');
}
```

oper – to implement the operation corresponding to an operator symbol

```
double oper(int symb, double op1, double op2)
{
```

```
switch (symb){
case '+' : return (op1+op2);
case '-' : return (op1-op2);
case '*' : return (op1*op2);
case '/' : return(op1/op2);
case '$' : return (pow (op1, op2);
default: printf ("%s","illegal operation");
exit(1);
}
}
```

Converting an expression from infix to postfix

Consider the given parentheses free infix expression:

A + B * C

Symb Postfix string opstk

1 A A

2 + A +

3 B AB +

4 * AB + *

5 C ABC + *

6 ABC * +

7 ABC * +

Consider the given parentheses infix expression:

(A+B)*C

Symb Postfix string Opstk

1 ((

2 A A (

3 + A (+

4 B AB (+

5) AB+

6 * AB+ *

7 C AB+C *

8 AB+C*

Program to convert an expression from infix to postfix

Along with pop, push, empty, popandtest, we also make use of additional functions such as, *isoperand, prcd, postfix.*

isoperand – returns TRUE if its argument is an operand and FALSE otherwise

prcd – accepts two operator symbols as arguments and returns TRUE if the first has precedence over the second when it appears to the left of the second in an infix string and FALSE otherwise

postfix – prints the postfix string

3.6. Multiple Stacks and Queues:

Up to now we have been concerned only with the representation of a single stack or a single queue in the memory of a computer. For these two cases we have seen efficient sequential data representations. What happens when a data representation is needed for several stacks and queues? Let us once again limit ourselves, to sequential mappings of these data objects into an array $V(1:m)$. If we have only 2 stacks to represent. then the solution is simple. We can use $V(1)$ for the bottom most element in stack 1 and $V(m)$ for the corresponding element in stack 2. Stack 1 can grow towards $V(m)$ and stack 2 towards $V(1)$. It is therefore possible to utilize efficiently all the available space. Can we do the same when more than 2 stacks are to be represented? The answer is no, because a one dimensional array has only two fixed points $V(1)$ and $V(m)$ and each stack requires a fixed point for its bottommost element. When more than two stacks, say n, are to be represented sequentially, we can initially divide out the available memory $V(1:m)$ into n segments and allocate one of these segments to each of the n stacks. This initial division of $V(1:m)$ into segments may be done in proportion to expected sizes of the various stacks if the sizes are known. In the absence of such information, $V(1:m)$ may be divided into equal segments. For each stack i we shall use $B(i)$ to represent a position one less than the position in V for the bottommost element of that stack. $T(i)$, 1 i n will point to the topmost element of stack i. We shall use the boundary condition $B(i) = T(i)$ iff the i'th stack is empty. If we grow the i'th stack in lower memory indexes than the i + 1'st, then with roughly equal initial segments we have

$$B(i) = T(i) = m/n\,(i - 1),\ 1 \quad i \quad n \qquad (3.2)$$

as the initial values of $B(i)$ and $T(i)$, (see figure 3.9). Stack i, 1 i n can grow from $B(i) + 1$ up to $B(i + 1)$ before it catches up with the i + 1'st stack. It is convenient both for the discussion and the algorithms to define $B(n + 1) = m$. Using this scheme the add and delete algorithms become:

procedure *ADD*(*i,X*)
//add element *X* to the *i*'th stack, 1 i *n*//
if $T(i) = B(i + 1)$ **then call** *STACK-FULL* (*i*)
$T(i)$ $T(i) + 1$
$V(T(i))$ X //add *X* to the *i*'th stack//
end *ADD*
procedure *DELETE*(*i,X*)

//delete topmost element of stack *i*//
if *T*(*i*) = *B*(*i*) **then call** *STACK-EMPTY*(*i*)
X V*(*T*(i))*
T(*i*) *T*(*i*) - 1
end *DELETE*

The algorithms to add and delete appear to be a simple as in the case of only 1 or 2 stacks. This really is not the case since the STACK_FULL condition in algorithm ADD does not imply that all *m* locations of *V* are in use. In fact, there may be a lot of unused space between stacks *j* and *j* + 1 for 1 *j n* and *j i*. The procedure STACK_FULL (*i*) should therefore determine whether there is any free space in *V* and shift stacks around so as to make some of this free space available to the *i*'th stack.

Several strategies are possible for the design of algorithm STACK_FULL. We shall discuss one strategy in the text and look at some others in the exercises. The primary objective of algorithm STACK_FULL is to permit the adding of elements to stacks so long as there is some free space in *V*. One way to guarantee this is to design STACK_FULL along the following lines:

a) determine the least *j, i* < *j n* such that there is free space between stacks *j* and *j* + 1, i.e., *T*(*j*) < *B*(*j* + 1). If there is such a *j*, then move stacks *i* + 1, *i* + 2, ...,*j* one position to the right (treating *V*(1) as leftmost and *V*(*m*) as rightmost), thereby creating a space between stacks *i* and *i* + 1.

b) if there is no *j* as in a), then look to the left of stack *i*. Find the largest *j* such that 1 *j* < *i* and there is space between stacks *j* and *j* + 1, i.e., *T*(*j*) < *B*(*j* + 1). If there is such a *j*, then move stacks *j* + 1, *j* + 2, ...,*i* one space left creating a free space between stacks *i* and *i* + 1.

c) if there is no *j* satisfying either the conditions of a) or b), then all *m* spaces of *V* are utilized and there is no free space.

It should be clear that the worst case performance of this representation for the *n* stacks together with the above strategy for STACK_FULL would be rather poor. In fact, in the worst case *O*(*m*) time may be needed for each insertion (see exercises). In the next chapter we shall see that if we do not limit ourselves to sequential mappings of data objects into arrays, then we can obtain a data representation for *m* stacks that has a much better worst case performance than the representation described here.

3.7. RECOMMENDED QUESTIONS

1. Assume A=1, B=2, C=3. Evaluate the following postfix expressions : a) AB+C-BA+C$-
   b) ABC+*CBA-+*

2. Convert the following infix expression to postfix : ((A-(B+C))*D)$(E+F)

3. Explain the different ways of representing expressions

4. State the advantages of using infix & postfix notations

5. State the rules to be followed during infix to postfix conversions

6. State the rules to be followed during infix to prefix conversions

7. Mention the advantages of representing stacks using linked lists than arrays

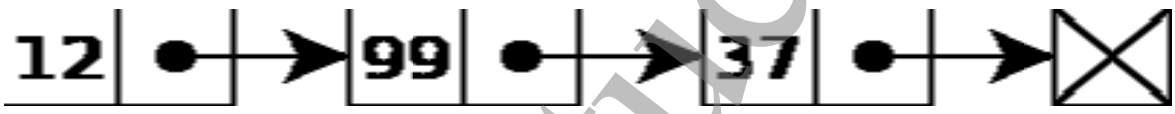
UNIT – 4 : LINKED LISTS

4.1. Singly Linked lists and Chains:

Let us discuss about the drawbacks of stacks and queues. During implementation, *overflow* occurs. No simple solution exists for more stacks and queues. In a sequential representation, the items of stack or queue are *implicitly* ordered by the sequential order of storage.

If the items of stack or queue are *explicitly* ordered, that is, each item contained within itself the address of the next item. Then a new data structure known as *linear linked list.* Each item in the list is called a *node* and contains two fields, an *information field* and a *next address field*. The information field holds the actual element on the list. The next address field contains the address of the next node in the list. Such an address, which is used to access a particular node, is known as a *pointer.*The *null pointer* is used to signal the end of a list. The list with no nodes – *empty list*or *null list.* The notations used in algorithms are:If *p* is a pointer to a node, *node(p)* refers to the node pointed to by *p*. *Info(p)* refersto the information of that node. *next(p)* refers to next address portion. If *next(p)* is notnull, *info(next(p))* refers to the information portion of the node that follows *node(p)* inthe list.

A linked list (or more clearly, "singly linked list") is a data structure that consists of a sequence of nodes each of which contains a reference (i.e., a *link*) to the next node in the sequence.



A linked list whose nodes contain two fields: an integer value and a link to the next node

Linked lists are among the simplest and most common data structures. They can be used to implement several other common abstract data structures, including stacks, queues, associative arrays, and symbolic expressions, though it is not uncommon to implement the other data structures directly without using a list as the basis of implementation.

The principal benefit of a linked list over a conventional array is that the list elements can easily be added or removed without reallocation or reorganization of the entire structure because the data items need not be stored contiguously in memory or on disk. Linked lists allow insertion and removal of nodes at any point in the list, and can do so with a constant number of operations if the link previous to the link being added or removed is maintained during list traversal.

On the other hand, simple linked lists by themselves do not allow random access to the data other than the first node's data, or any form of efficient indexing.

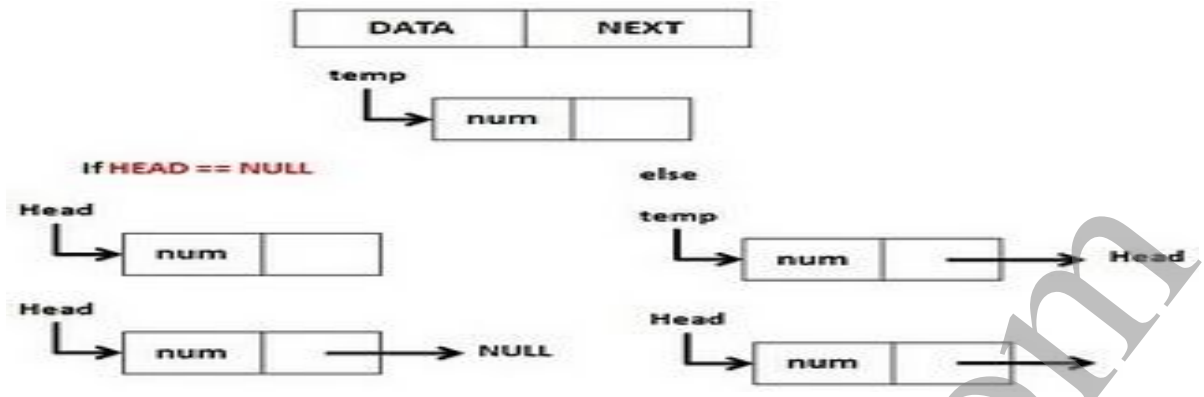


Fig :Inserting and removing nodes from a list

A list is a dynamic data structure. The number of nodes on a list may vary dramatically and dynamically as elements are inserted and removed. For example, let us consider a list with elements 5, 3 and 8 and we need to add an integer 6 to the frontof that list. Then,

p=getnode();

info(p)=6;

next(p)=list;

list=p;

Similarly, for removing an element from the list, the process is almost exactly opposite of the process to add a node to the front of the list. Remove the first node of a nonempty list and store the value of *info field* into a variable *x*. then,

p=list;

list=next(p);

x=info(p);

4.2. Representing Chains in C:

A chain is a linked list in which each node represents one element.

- There is a link or pointer from one element to the next.
- The last node has a NULL (or 0) pointer

An array and a sequential mapping is used to represent simple data structures in the previous chapters

•This representation has the property that successive nodes of the data object are stored a fixed distance apart

(1) If the element aijis stored at location Lij, then aij+1is at the location Lij+1

(2) If the i-thelement in a queue is at location Li, then the (i+1)-th element is at location Li+1% n for the circular representation

(3) If the topmost node of a stack is at location LT , then the node beneath it is at location LT-1, and so on

•When a sequential mapping is used for ordered lists, operations such as insertion and deletion of arbitrary elements become expensive.

In a linked representation–To access list elements in the correct order, with each element we store the address or location of the next element in the list–A linked list is comprised of nodes; each node has zero or more data fields and one or more link or pointer fields.

4.3. Linked Stacks and Queues:

Pushing a Linked Stack

```
Error code Stack :: push(const Stack entry &item)

/* Post: Stack entry item is added to the top of the Stack; returns success or

returns a code of over_ow if dynamic memory is exhausted. */

{

Node *new top = new Node(item, top node);

if (new top == NULL) return over_ow;

top node = new top;

return success;

}
```

Popping a Linked Stack

```
Error code Stack :: pop( )

/* Post: The top of the Stack is removed. If the Stack is empty the method returns

under_ow; otherwise it returns success. */

{

Node *old top = top node;

if (top node == NULL) return under_ow;

top node = old top->next;

delete old top;
```

return success;

}

A queue is a particular kind of collection in which the entities in the collection are kept in order and the principal (or only) operations on the collection are the addition of entities to the rear terminal position and removal of entities from the front terminal position. This makes the queue a First-In-First-Out (FIFO) data structure. In a FIFO data structure, the first element added to the queue will be the first one to be removed. This is equivalent to the requirement that once an element is added, all elements that were added before have to be removed before the new element can be invoked. A queue is an example of a linear data structure.

Queues provide services in computer science, transport, and operations research where various entities such as data, objects, persons, or events are stored and held to be processed later. In these contexts, the queue performs the function of a buffer.

```
#include<malloc.h>
#include<stdio.h>
structnode{
intvalue;
structnode*next;
};

voidInit(structnode*n){
n->next=NULL;
}
voidEnqueue(structnode*root,intvalue){
structnode*j=(structnode*)malloc(sizeof(structnode));
j->value=value;
j->next=NULL;
structnode*temp                                                    ;
temp=root;
while(temp->next!=NULL)
{
temp=temp->next;
}
temp->next=j;
printf("Value Enqueued is : %d\n",value);

}
voidDequeue(structnode*root)
{
if(root->next==NULL)
{
printf("NoElementtoDequeue\n");
}
```

```
else
{
structnode*temp;
temp=root->next;
root->next=temp->next;
printf("ValueDequeuedis%d\n",temp->value);
free(temp);
}
}

voidmain()
{
structnodesample_queue;
Init(&sample_queue);
Enqueue(&sample_queue,10);
Enqueue(&sample_queue,50);
Enqueue(&sample_queue,570);
Enqueue(&sample_queue,5710);
Dequeue(&sample_queue);
Dequeue(&sample_queue);
Dequeue(&sample_queue);
}
```

4.4. Polynomials:

In mathematics, a polynomial (from Greek *poly*, "many" and medieval Latin *binomium*, "binomial"[1] [2] [3]) is an expression of finite length constructed from variables (also known as indeterminates) and constants, using only the operations of addition, subtraction, multiplication, and non-negative integer exponents. For example, $x^2 - 4x + 7$ is a polynomial, but $x^2 - 4/x + 7x^{3/2}$ is not, because its second term involves division by the variable x (4/x) and because its third term contains an exponent that is not a whole number (3/2). The term "polynomial" can also be used as an adjective, for quantities that can be expressed as a polynomial of some parameter, as in "polynomial time" which is used in computational complexity theory.

Polynomials appear in a wide variety of areas of mathematics and science. For example, they are used to form polynomial equations, which encode a wide range of problems, from elementary word problems to complicated problems in the sciences.

A polynomial is a mathematical expression involving a sum of powers in one or more variables multiplied by coefficients. A polynomial in one variable (i.e., a univariate polynomial) with constant coefficients is given by

$$a_n x^n + \ldots + a_2 x^2 + a_1 x + a_0. \quad (1)$$

The individual summands with the coefficients (usually) included are called monomials (Becker and Weispfenning 1993, p. 191), whereas the products of the form $x_1^{a_1} \cdots x_n^{a_n}$ in the multivariate case, i.e., with the coefficients omitted, are called terms (Becker and Weispfenning 1993, p. 188). However, the term "monomial" is sometimes also used to mean polynomial summands *without* their coefficients, and in some older works, the definitions of monomial and term are reversed. Care is therefore needed in attempting to distinguish these conflicting usages.

The highest power in a univariate polynomial is called its order, or sometimes its degree.

Any polynomial $P(x)$ with $P(0) \neq 0$ can be expressed as

$$P(x) = P(0) \prod_{\rho} \left(1 - \frac{x}{\rho}\right), \tag{2}$$

where the product runs over the roots ρ of $P(\rho) = 0$ and it is understood that multiple roots are counted with multiplicity.

A polynomial in two variables (i.e., a bivariate polynomial) with constant coefficients is given by

$$a_{nm} x^n y^m + \ldots + a_{22} x^2 y^2 + a_{21} x^2 y + a_{12} x y^2 + a_{11} x y + a_{10} x + a_{01} y + a_{00}. \tag{3}$$

The sum of two polynomials is obtained by adding together the coefficients sharing the same powers of variables (i.e., the same terms) so, for example,

$$(a_2 x^2 + a_1 x + a_0) + (b_1 x + b_0) = a_2 x^2 + (a_1 + b_1) x + (a_0 + b_0) \tag{4}$$

and has order less than (in the case of cancellation of leading terms) or equal to the maximum order of the original two polynomials. Similarly, the product of two polynomials is obtained by multiplying term by term and combining the results, for example

$$(a_2 x^2 + a_1 x + a_0)(b_1 x + b_0) = a_2 x^2 (b_1 x + b_0) + a_1 x (b_1 x + b_0) + a_0 (b_1 x + b_0) \tag{5}$$

$$= a_2 b_1 x^3 + (a_2 b_0 + a_1 b_1) x^2 + (a_1 b_0 + a_0 b_1) x + a_0 b_0 \tag{6}$$

and has order equal to the sum of the orders of the two original polynomials.

A polynomial quotient

$$R(x) = \frac{P(x)}{Q(x)} \tag{7}$$

of two polynomials $P(x)$ and $Q(x)$ is known as a rational function. The process of performing such a division is called long division, with synthetic division being a simplified method of recording the division. For any polynomial $P(x)$, $P(x) - x$ divides $P(P(x)) - x$, meaning that the polynomial quotient is a rational polynomial or, in the case of an integer polynomial, another integer polynomial (N. Sato, pers. comm., Nov. 23, 2004).

Exchanging the coefficients of a univariate polynomial end-to-end produces a polynomial

$$a_0 x^n + a_1 x^{n-1} + \ldots + a_{n-1} x + a_n = 0 \quad (8)$$

Whose roots are reciprocals $1/x_i$ of the original roots x_i.

Horner's rule provides a computationally efficient method of forming a polynomial from a list of its coefficients, and can be implemented in *Mathematica* as follows.

```
Polynomial[l_List, x_] := Fold[x #1 + #2&, 0, l]
```

The following table gives special names given to polynomials of low orders.

polynomial order	polynomial name
2	quadratic polynomial
3	cubic polynomial
4	quartic
5	quintic
6	sextic

Polynomials of fourth degree may be computed using three multiplications and five additions if a few quantities are calculated first (Press *et al.* 1989):

$$a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 = \left[(A x + B)^2 + A x + C\right]\left[(A x + B)^2 + D\right] + E, \quad (9)$$

where

$$A \equiv (a_4)^{1/4} \quad (10)$$

$$B \equiv \frac{a_3 - A^3}{4 A^3} \quad (11)$$

$$D \equiv 3 B^2 + 8 B^3 + \frac{a_1 A - 2 a_2 B}{A^2} \quad (12)$$

$$C \equiv \frac{a_2}{A^2} - 2 B - 6 B^2 - D \quad (13)$$

$$E \equiv a_0 - B^4 - B^2 (C + D) - C D. \quad (14)$$

Similarly, a polynomial of fifth degree may be computed with four multiplications and five additions, and a polynomial of sixth degree may be computed with four multiplications and seven additions.The use of linked lists is well suited to polynomial operations. We can easily imagine writing a collection of procedures for input, output addition, subtraction and multiplication of polynomials using linked lists as

the means of representation. A hypothetical user wishing to read in polynomials $A(x)$, $B(x)$ and $C(x)$ and then compute $D(x) = A(x) * B(x) + C(x)$ would write in his main program:

```
call READ(A)
call READ(B)
call READ(C)
T  PMUL(A, B)
D  PADD(T, C)
call PRINT(D)
```

Now our user may wish to continue computing more polynomials. At this point it would be useful to reclaim the nodes which are being used to represent T(x). This polynomial was created only as a partial result towards the answer D(x). By returning the nodes of T(x), they may be used to hold other polynomials.

```
procedure ERASE(T)
//return all the nodes of T to the available space list avoiding repeated
calls to procedure RET//
if T = 0 then return
p  T
while LINK (p)  0 do      //find the end of T//
p  LINK (p)
end
LINK (p) AV       // p points to the last node of T//
AV  T     //available list now includes T//
end ERASE
```

Study this algorithm carefully. It cleverly avoids using the RET procedure to return the nodes of T one node at a time, but makes use of the fact that the nodes of T are already linked. The time required to erase $T(x)$ is still proportional to the number of nodes in T. This erasing of entire polynomials can be carried out even more efficiently by modifying the list structure so that the LINK field of the last node points back to the first node as in figure 4.8. A list in which the last node points back to the first will be termed a *circular list*. A *chain* is a singly linked list in which the last node has a zero link field.

Circular lists may be erased in a fixed amount of time independent of the number of nodes in the list. The algorithm below does this.

```
procedure CERASE(T)
//return the circular list T to the available pool//
if T = 0 then return;
X  LINK (T)
LINK(T)  AV
AV  X
end CERASE
```

4.5. Additional List operations:

It is often necessary and desirable to build a variety of routines for manipulating singly linked lists. Some that we have already seen are: 1) INIT which originally links together the AV list; 2) GETNODE and 3) RET which get and return nodes to AV. Another useful operation is one which inverts a chain. This routine is especially interesting because it can be done "in place" if we make use of 3 pointers.

```
procedure INVERT(X)
//a chain pointed at by X is inverted so that if X = (a1, ...,am)
then after execution X = (am, ...,a1)//
p  X;q  0
while p  0 do
r  q;q  p      //r follows q; q follows p//
p  LINK(p)       //p moves to next node//
LINK(q)  r       //link q to previous node//
end
X  q
end INVERT
```

The reader should try this algorithm out on at least 3 examples: the empty list, and lists of length 1 and 2 to convince himself that he understands the mechanism. For a list of m 1 nodes, the **while** loop is executed m times and so the computing time is linear or $O(m)$.

Another useful subroutine is one which concatenates two chains X and Y.

```
procedure CONCATENATE(X, Y, Z)
//X = (a1, ...,am), Y = (b1, ...,bn), m,n  0, produces a new chain
Z = (a1, ...,am,b1 , ...,bn)//
Z  X
if X = 0 then [Z  Y; return]
if Y = 0 then return
p  X
while LINK(p)  0 do      //find last node of X//
p  LINK(p)
end
LINK(p)  Y       //link last node of X to Y//
end CONCATENATE
```

This algorithm is also linear in the length of the first list. From an aesthetic point of view it is nicer to write this procedure using the case statement in SPARKS. This would look like:

```
procedure CONCATENATE(X, Y, Z)
case
: X = 0 :Z  Y
: Y = 0 : Z  X
: else : p  X; Z  X
while LINK(p)  0 do
p  LINK (p)
end
LINK(p)  Y
end
end CONCATENATE
```

Suppose we want to insert a new node at the front of this list. We have to change the LINK field of the node containing x_3. This requires that we move down the entire length of A until we find the last node. It is more convenient if the name of a circular list points to the last node rather than the first.

Now we can write procedures which insert a node at the front or at the rear of a circular list and take a fixed amount of time.

procedure *INSERT__FRONT*(*A, X*)
//insert the node pointed at by X to the front of the circular list
A, where A points to the last node//
if A = 0 **then** [*A X*
LINK (*X*) *A*]
else [*LINK*(*X*) *LINK* (*A*)
LINK(*A*) *X*]
end *INSERT--FRONT*

To insert *X* at the rear, one only needs to add the additional statement *A X* to the **else** clause of *INSERT_-_FRONT*.

As a last example of a simple procedure for circular lists, we write a function which determines the length of such a list.

procedure *LENGTH*(*A*)
//find the length of the circular list *A*//
i 0
if A 0 **then** [*ptr A*
repeat
i i + 1; *ptr* LINK(*ptr*)
until *ptr* = *A*]
return (*i*)
end *LENGTH*

4.6. Sparse Matrices:

A sparse matrix is a matrix populated primarily with zeros (Stoer & Bulirsch 2002, p. 619). The term itself was coined by Harry M. Markowitz.

Conceptually, sparsity corresponds to systems which are loosely coupled. Consider a line of balls connected by springs from one to the next; this is a sparse system. By contrast, if the same line of balls had springs connecting each ball to all other balls, the system would be represented by a dense matrix. The concept of sparsity is useful in combinatorics and application areas such as network theory, which have a low density of significant data or connections.

A sparse matrix is a matrix that allows special techniques to take advantage of the large number of zero elements. This definition helps to define "how many" zeros a matrix needs in order to be "sparse." The answer is that it depends on what the structure of the matrix is, and what you want to do with it. For example, a randomly generated sparse $n \times n$ matrix with cn entries scattered randomly throughout the matrix is not sparse in the sense of Wilkinson (for direct methods) since it takes $O(n^3)$.

Creating a sparse matrix

If a matrix A is stored in ordinary (dense) format, then the command S = sparse(A) creates a copy of the matrix stored in sparse format. For example:

```
>> A = [0 0 1;1 0 2;0 -3 0]

A =

    0    0    1

    1    0    2

    0   -3    0

>> S = sparse(A)

S =

  (2,1)        1

  (3,2)       -3

  (1,3)        1

  (2,3)        2

>> whos

 Name      Size         Bytes  Class


 A         3x3            72  double array

 S         3x3            64  sparse array

Grand total is 13 elements using 136 bytes
```

Unfortunately, this form of the sparse command is not particularly useful, since if A is large, it can be very time-consuming to first create it in dense format. The command S = sparse(m,n) creates an $m \times n$ zero matrix in sparse format. Entries can then be added one-by-one:

```
>> A = sparse(3,2)

A =

   All zero sparse: 3-by-2

>> A(1,2)=1;

>> A(3,1)=4;
```

```
>> A(3,2)=-1;

>> A

A =

  (3,1)       4

  (1,2)       1

  (3,2)      -1
```

(Of course, for this to be truly useful, the nonzeros would be added in a loop.)

Another version of the sparse command is S = sparse(I,J,S,m,n,maxnz). This creates an $m \times n$ sparse matrix with entry $(I(k),J(k))$ equal to $S(k), k = 1, \ldots, length(S)$. The optional argument maxnz causes Matlab to pre-allocate storage for maxnz nonzero entries, which can increase efficiency in the case when more nonzeros will be added later to S.

The most common type of sparse matrix is a banded matrix, that is, a matrix with a few nonzero diagonals. Such a matrix can be created with the spdiags command. Consider the following matrix:

```
>> A

A =

   64   -16     0   -16     0     0     0     0     0

  -16    64   -16     0   -16     0     0     0     0

    0   -16    64     0     0   -16     0     0     0

  -16     0     0    64   -16     0   -16     0     0

    0   -16     0   -16    64   -16     0   -16     0

    0     0   -16     0   -16    64     0     0   -16

    0     0     0   -16     0     0    64   -16     0

    0     0     0     0   -16     0   -16    64   -16

    0     0     0     0     0   -16     0   -16    64
```

This is a 9×9 matrix with 5 nonzero diagonals. In Matlab's indexing scheme, the nonzero diagonals of A are numbers -3, -1, 0, 1, and 3 (the main diagonal is number 0, the first subdiagonal is number -1, the first superdiagonal is number 1, and so forth). To create the same matrix in sparse format, it is first necessary to create a 9×5 matrix containing the nonzero diagonals of A. Of course, the diagonals, regarded as column vectors, have different lengths; only the main diagonal has length 9. In order to gather the various

diagonals in a single matrix, the shorter diagonals must be padded with zeros. The rule is that the extra zeros go at the bottom for subdiagonals and at the top for superdiagonals. Thus we create the following matrix:

```
>> B = [
  -16  -16   64    0    0
  -16  -16   64  -16    0
  -16    0   64  -16    0
  -16  -16   64    0  -16
  -16  -16   64  -16  -16
  -16    0   64  -16  -16
    0  -16   64    0  -16
    0  -16   64  -16  -16
    0    0   64  -16  -16
];
```

(notice the technique for entering the rows of a large matrix on several lines). The spdiags command also needs the indices of the diagonals:

```
>> d = [-3,-1,0,1,3];
```

The matrix is then created as follows:

```
S = spdiags(B,d,9,9);
```

The last two arguments give the size of S.

Perhaps the most common sparse matrix is the identity. Recall that an identity matrix can be created, in dense format, using the command eye. To create the $n \times n$ identity matrix in sparse format, use I = speye(n). Another useful command is spy, which creates a graphic displaying the sparsity pattern of a matrix. For example, the above penta-diagonal matrix A can be displayed by the following command; see Figure 6:

```
>> spy(A)
```

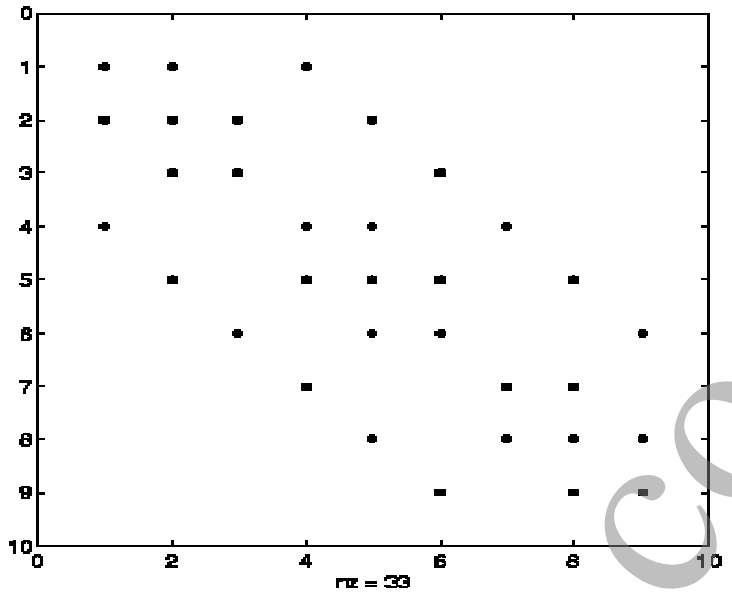


Figure 6: The sparsity pattern of a matrix

4.7. Doubly Linked Lists :

Although a circularly linked list has advantages over linear lists, it still has some drawbacks. One cannot traverse such a list backward. Double-linked lists require more space per node , and their elementary operations are more expensive; but they are often easier to manipulate because they allow sequential access to the list in both directions. In particular, one can insert or delete a node in a constant number of operations given only that node's address. (Compared with singly-linked lists, which require the *previous* node's address in order to correctly insert or delete.) Some algorithms require access in both directions. On the other hand, they do not allow tail-sharing, and cannot be used as persistent data structures.

Operations on Doubly Linked Lists

One operation that can be performed on doubly linked list but not on ordinary linked list is to delete a given node. The following c routine deletes the node pointed by pfrom a doubly linked list and stores its contents in x. It is called by delete(p).

```
delete( p )

{

NODEPTR p, q, r;

int *px;

if ( p = = NULL )

{

printf(“ Void Deletion \n”);
```

```
return;
}
*px = p -> info;
q = p -> left;
r = p -> right;
q -> right = r;
r -> left = q;
freenode( p );
return;
}
```

A node can be inserted on the right or on the left of a given node. Let us consider insertion at right side of a given node. The routine insert right inserts a node with information field x to right of node(p) in a doubly linked list.

```
insertright( p, x) {
NODEPTR p, q, r;
int x;
if ( p = = NULL ) {
printf(" Void Insertion \n");
return;
}
q = getnode();
q -> info = x;
r = p -> right;
r -> left = q;
q -> right = r;
q -> left = p;
p -> left = q;
```

return;

}

4.8. RECOMMENDED QUESTIONS

1.Define Linked Lists

2.List & explain the basic operations carried out in a linked list

3.List out any two applications of linked list and any two advantages of doubly linked list over singly linked list.

4. Write a C program to simulate an ordinary queue using a singly linked list.

5. Give an algorithm to insert a node at a specified position for a given singly linked list.

6. Write a C program to perform the following operations on a doubly linked list:

i) To create a list by adding each node at the front.

ii) To display all the elements in the reverse order.

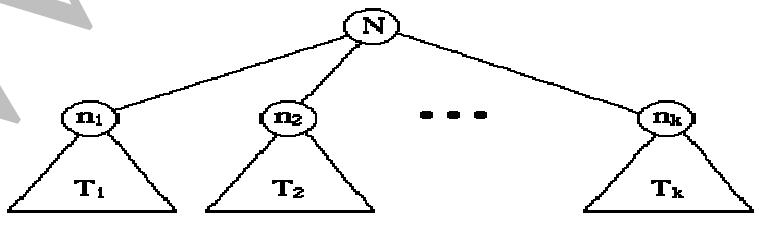
UNIT – 5 : TREES – 1

5.1 Introduction:

A *tree* is a finite set of one or more nodes such that: (i) there is a specially designated node called the *root*; (ii) the remaining nodes are partitioned into *n* 0 disjoint sets T_1, ...,T_n where each of these sets is a tree. T_1, ...,T_n are called the *subtrees* of the root. A tree structure means that the data is organized so that items of information are related by branches. One very common place where such a structure arises is in the investigation of genealogies.

AbstractDataType tree{
instances
A set of elements:
(1) empty or having a distinguished root element
(2) each non-root element having exactly one parent element operations
root()
degree()
child(k)
}
Some basic terminology for trees:

- Trees are formed from *nodes* and *edges*. Nodes are sometimes called *vertices*. Edges are sometimes called *branches*.
- Nodes may have a number of properties including *value* and *label*.
- Edges are used to relate nodes to each other. In a tree, this relation is called "parenthood."
- An edge {a,b} between nodes a and b establishes a as the *parent* of b. Also, b is called a *child* of a.
- Although edges are usually drawn as simple lines, they are really directed from parent to child. In tree drawings, this is top-to-bottom.
- **Informal Definition**: a *tree* is a collection of nodes, one of which is distinguished as "root," along with a relation ("parenthood") that is shown by edges.
- **Formal Definition**: This definition is "recursive" in that it defines tree in terms of itself. The definition is also "constructive" in that it describes how to construct a tree.
    1. A single node is a tree. It is "root."
    2. Suppose N is a node and T_1, T_2, ..., T_k are trees with roots n_1, n_2, ...,n_k, respectively. We can construct a new tree T by making N the parent of the nodes n_1, n_2, ..., n_k. Then, N is the root of T and T_1, T_2, ..., T_k are subtrees.



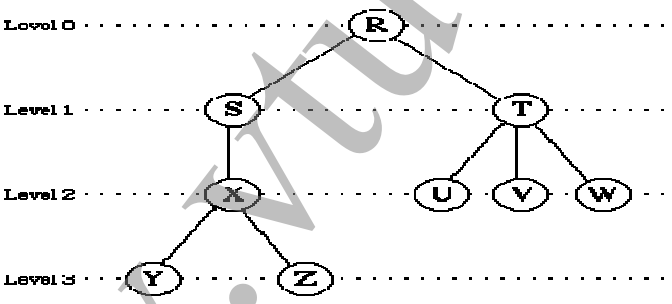
The tree T, constructed using k subtrees

More terminology

- A node is either *internal* or it is a *leaf*.
- A *leaf* is a node that has no children.
- Every node in a tree (except root) has exactly one parent.
- The *degree of a node* is the number of children it has.
- The *degree of a tree* is the maximum degree of all of its nodes.

Paths and Levels

- **Definition**: A *path* is a sequence of nodes $n_1, n_2, ..., n_k$ such that node n_i is the parent of node n_{i+1} for all $1 <= i <= k$.
- **Definition**: The *length* of a path is the number of edges on the path (one less than the number of nodes).
- **Definition**: The *descendents* of a node are all the nodes that are on some path from the node to any leaf.
- **Definition**: The *ancestors* of a node are all the nodes that are on the path from the node to the root.
- **Definition**: The *depth* of a node is the length of the path from root to the node. The depth of a node is sometimes called its *level*.
- **Definition**: The *height of a node* is the length of the longest path from the node to a leaf.
- **Definition**: the *height of a tree* is the height of its root.



A general tree, showing node depths (levels)

In the example above:

- The nodes Y, Z, U, V, and W are leaf nodes.
- The nodes R, S, T, and X are internal nodes.
- The degree of node T is 3. The degree of node S is 1.
- The depth of node X is 2. The depth of node Z is 3.
- The height of node Z is zero. The height of node S is 2. The height of node R is 3.
- The height of the tree is the same as the height of its root R. Therefore the height of the tree is 3.
- The sequence of nodes R,S,X is a path.
- The sequence of nodes R,X,Y is not a path because the sequence does not satisfy the "parenthood" property (R is not the parent of X).

Representation of trees

1. List representation

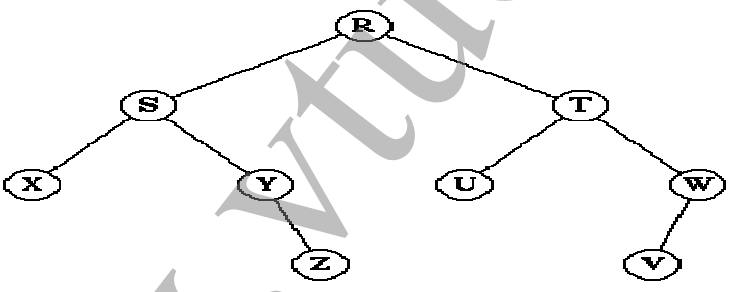
2. left child right sibling representation

3. Representation as a degree two tree

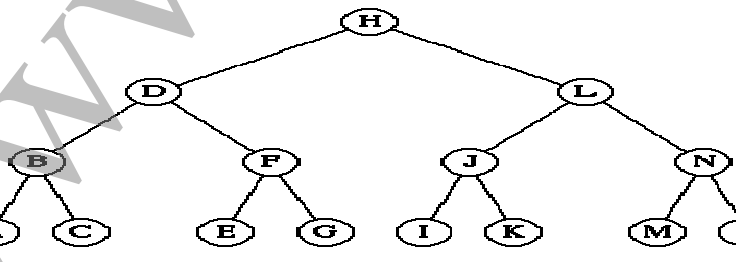
5.2 Binary Trees:

- **Definition**: A binary tree is a tree in which each node has degree of exactly 2 and the children of each node are distinguished as "left" and "right." Some of the children of a node may be empty.
- **Formal Definition**: A binary tree is:
  1. either empty, or
  2. it is a node that has a left and a right subtree, each of which is a binary tree.
- **Definition**: A *full binary tree* (FBT) is a binary tree in which each node has exactly 2 non-empty children or exactly two empty children, and all the leaves are on the same level. (Note that this definition differs from the text definition).
- **Definition**: A *complete binary tree* (CBT) is a FBT except, perhaps, that the deepest level may not be completely filled. If not completely filled, it is filled from left-to-right.
- A FBT is a CBT, but not vice-versa.

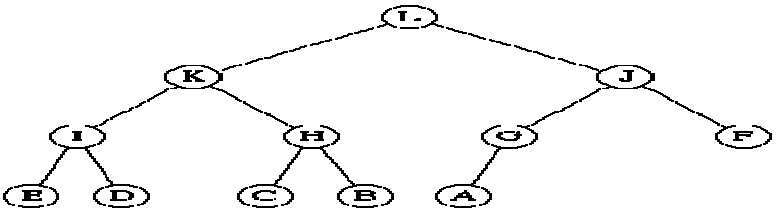
Examples of Binary Trees



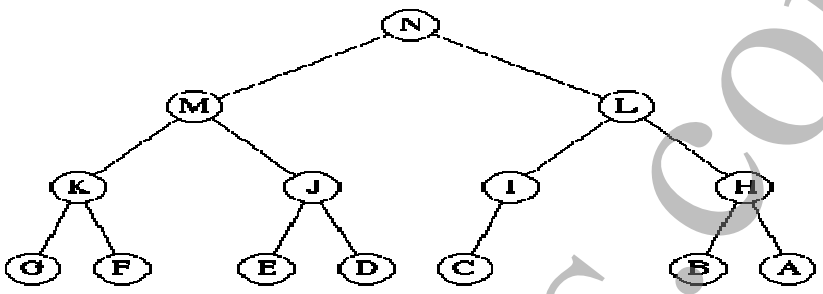
A Binary Tree. As usual, the empty children are not explicitly shown.



A Full Binary Tree. In a FBT, the number of nodes at level **i** is 2^i.



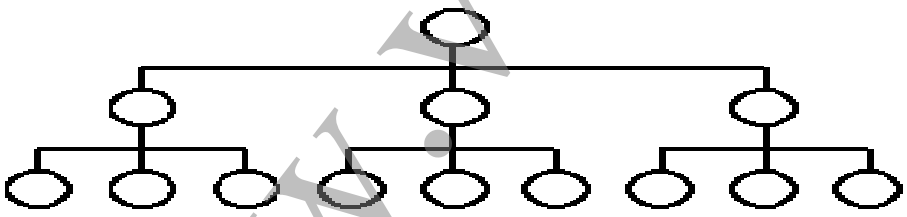
A Complete Binary Tree.



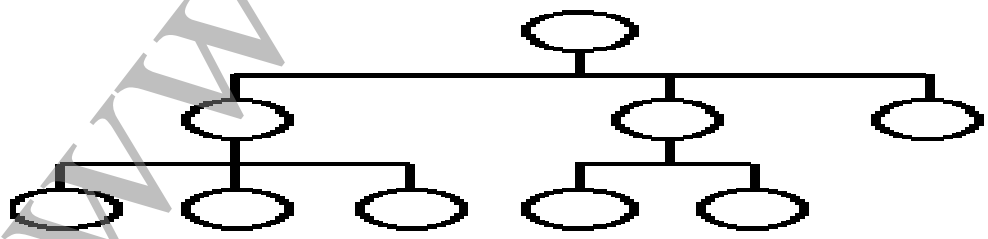
Not a Complete Binary Tree. The tree above is **not** a CBT because the deepest level is not filled from left-to-right.

Properties

Full tree **A tree with all the leaves at the same level, and all the non-leaves having the same degree.**



- **Complete Tree** A full tree in which the 'last' elements are deleted.



- Level h of a full tree has d^{h-1} nodes.

  The first h levels of a full tree have nodes.

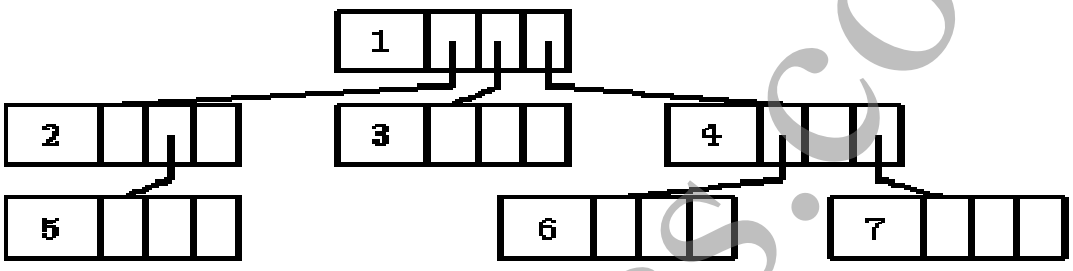
$$1 + d + d^2 + \cdots + d^{h-1} = \frac{d^h - 1}{d - 1}$$

- A tree of height h and degree d has at most d^h - 1 elements

$$N(h) = \begin{cases} dN(h-1) & \text{if } h > 1 \\ 1 & \text{if } h = 1 \end{cases}$$

Representations

1. Nodes consisting of a data field and k pointers

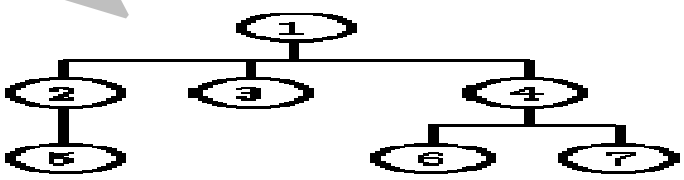


2. Nodes consisting of w data field and two pointers: a pointer to the first child, and a pointer to the next sibling.
3. A tree of degree k assumes an array for holding a complete tree of degree k, with empty cells assigned for missing elements.

5.3 Binary tree Traversals:

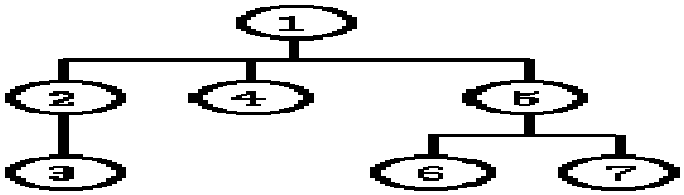
There are many operations that we often want to perform on trees. One notion that arises frequently is the idea of traversing a tree or visiting each node in the tree exactly once. A full traversal produces a linear order for the information in a tree. This linear order may be familiar and useful. When traversing a binary tree we want to treat each node and its subtrees in the same fashion. If we let *L, D, R* stand for moving left, printing the data, and moving right when at a node then there are six possible combinations of traversal: *LDR, LRD, DLR, DRL, RDL,* and *RLD*. If we adopt the convention that we traverse left before right then only three traversals remain: *LDR, LRD* and *DLR*. To these we assign the names inorder, postorder and preorder because there is a natural correspondence between these traversals and producing the infix, postfix and prefix forms of an expression.

Level order



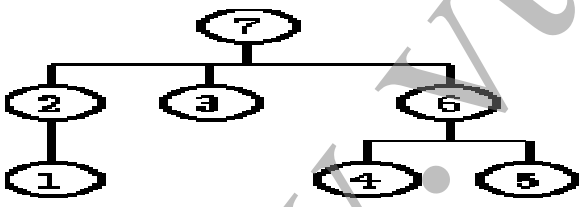
```
x := root()
if( x ) queue (x)
while( queue not empty ){
  x := dequeue()
  visit()
  i=1; while( i <= degree() ){
    queue( child(i) )
  }
}
```

Preorder




```
procedure preorder(x){
  visit(x)
  i=1; while( i <= degree() ){
    preorder( child(i) )
  }
}
```

Postorder




```
procedure postorder(x){
  i=1; while( i <= degree() ){
    postorder( child(i) )
  }
  visit(x)
}
```

Inorder

Meaningful just for binary trees.

```
procedure inorder(x){
  if( left_child_for(x) ) { inorder( left_child(x)  ) }
  visit(x)
  if( right_child_for(x) ) { inorder( right_child(x)  ) }
}
```

Usages for 'visit': determine the height, count the number of elements .

5.4. Threaded Binary trees:

If we look carefully at the linked representation of any binary tree, we notice that there are more null links than actual pointers. As we saw before, there are $n + 1$ null links and $2n$ total links. A clever way to make use of these null links has been devised by A. J. Perlis and C. Thornton. Their idea is to replace the null links by pointers, called threads, to other nodes in the tree. If the RCHILD(*P*) is normally equal to zero, we will replace it *by a pointer to the node which would be printed after P when traversing the tree in inorder*. A null LCHILD link at node *P* is replaced *by a pointer to the node which immediately precedes node P in inorder*.

The tree *T* has 9 nodes and 10 null links which have been replaced by threads. If we traverse *T* in inorder the nodes will be visited in the order *H D I B E A F C G*. For example node *E* has a predecessor thread which points to B and a successor thread which points to *A*.

In the memory representation we must be able to distinguish between threads and normal pointers. This is done by adding two extra one bit fields LBIT and RBIT.

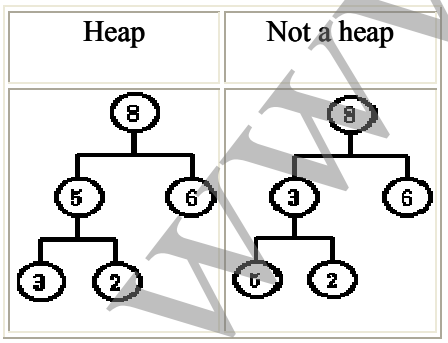
LBIT(*P*) =1 if LCHILD(*P*) is a normal pointer
LBIT(*P*) = 0 if LCHILD(*P*) is a thread
RBIT(*P*) = 1 if RCHILD(*P*) is a normal pointer
RBIT(*P*) = 0 if RCHILD(*P*) is a thread

5.5. Heaps

A **heap** is a complete tree with an ordering-relation R holding between each node and its descendant.

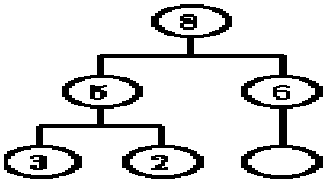
Examples for R: smaller-than, bigger-than

Assumption: In what follows, R is the relation 'bigger-than', and the trees have degree 2.

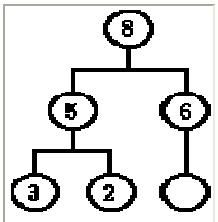
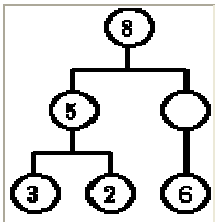
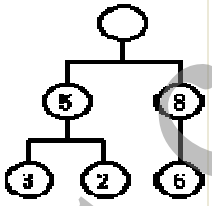


Adding an Element

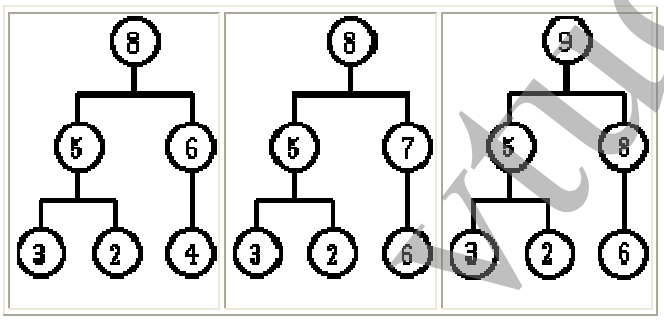
1. Add a node to the tree



2. Move the elements in the path from the root to the new node one position down, if they are smaller than the new element

new element	4	7	9
modified tree			

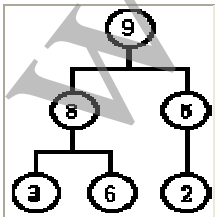
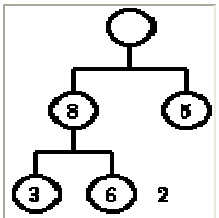
3. Insert the new element to the vacant node



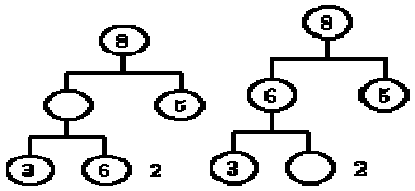
4. A complete tree of n nodes has depth $\lceil \log n \rceil$, hence the time complexity is O(log n)

Deleting an Element

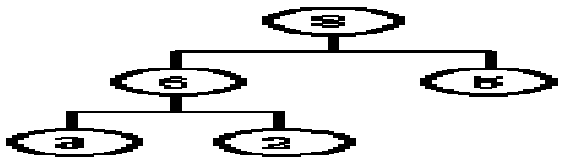
1. Delete the value from the root node, and delete the last node while saving its value.

before	after
	

2. As long as the saved value is smaller than a child of the vacant node, move up into the vacant node the largest value of the children.



3. Insert the saved value into the vacant node



4. The time complexity is O(log n)

Initialization:

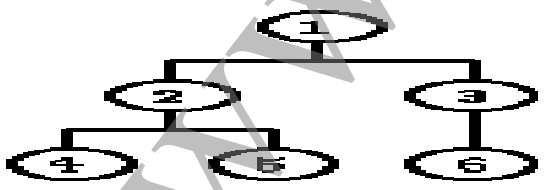
Brute Force

Given a sequence of n values e_1, ..., e_n, repeatedly use the insertion module on the n given values.

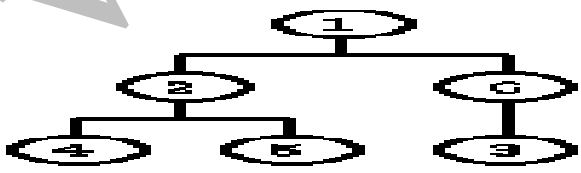
- Level h in a complete tree has at most $2^{h-1} = O(2^n)$ elements
- Levels 1, ..., h - 1 have $2^0 + 2^1 + \ldots + 2^{h-2} = O(2^h)$ elements
- Each element requires O(log n) time. Hence, brute force initialization requires O(n log n) time.

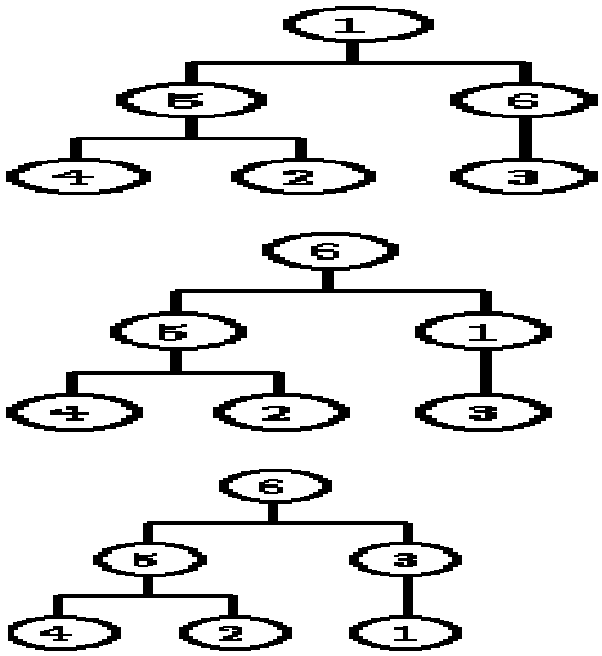
Efficient

- Insert the n elements e_1, ..., e_n into a complete tree



- For each node, starting from the last one and ending at the root, reorganize into a heap the subtree whose root node is given. The reorganization is performed by interchanging the new element with the child of greater value, until the new element is greater than its children.





- The time complexity is O(0 * (n/2) + 1 * (n/4) + 2 * (n/8) + . . .+ (log n) * 1) = O(n(0 .2^{-1} + 1 .2^{-2} + 2 .2^{-3} + . . .+ (log n) .2$^{-\log n}$)) = O(n)

  since the following equalities holds. $\sum_{k=1}^{\infty}(k-1)2^{-k} = 2[\sum_{k=1}^{\infty}(k-1)2^{-k}] - [\sum_{k=1}^{\infty}(k-1)2^{-k}] = [\sum_{k=1}^{\infty}k2^{-k}] - [\sum_{k=1}^{\infty}(k-1)2^{-k}] = \sum_{k=1}^{\infty}[k-(k-1)]2^{-k} = \sum_{k=1}^{\infty}2^{-k} = 1$

Applications:

Priority Queue A dynamic set in which elements are deleted according to a given ordering-relation.
Heap Sort Build a heap from the given set (O(n)) time, then repeatedly remove the elements from the heap (O(n log n)).

5.6. RECOMMENDED QUESTIONS

1. Construct a binary tree for : ((6+(3-2)*5)^2+3)

2. What is threaded binary tree? Explain right in and left in threaded binary trees.

3. Define a tree. Write the procedure to convert general tree to binary tree.

4. Define degree of the node, leaves, root

5. Define internal nodes, parent node, depth and height of a tree

6. State the properties of a binary tree

7. What is meant by binary tree traversal? What are the different binary tree traversal techniques

8. State the merits & demerit of linear representation of binary trees.

9. Define right-in threaded tree & left-in threaded tree

UNIT – 6 : TREES – 2, GRAPHS

Introduction

The first recorded evidence of the use of graphs dates back to 1736 when Euler used them to solve the now classical Koenigsberg bridge problem.Some of the applications of graphs are: analysis of electrical circuits, finding shortest routes, analysis of project planning, identification of chemical compounds, statistical mechanics, genetics, cybernetics, linguistics, social sciences, etc. Indeed, it might well be said that of all mathematical structures, graphs are the most widely used.

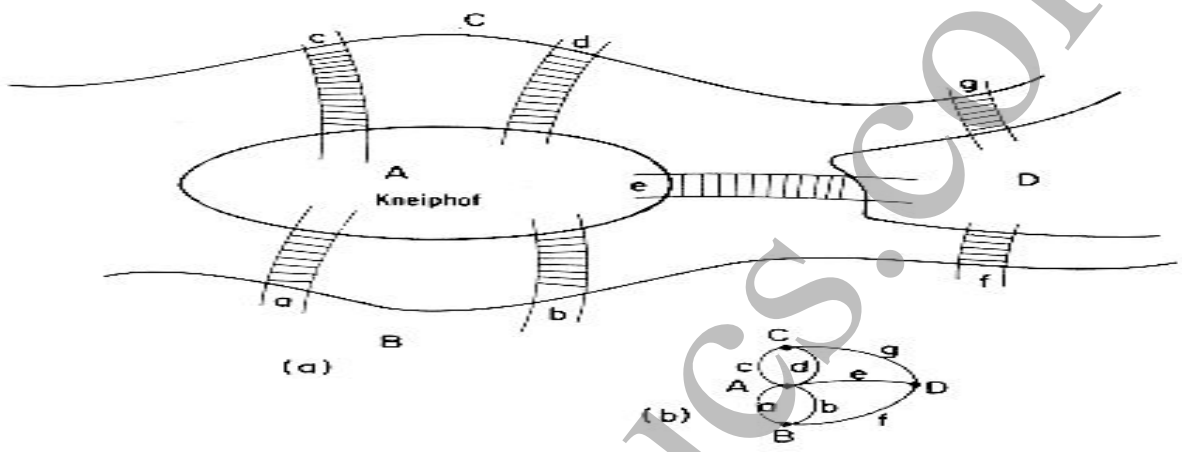


Figure 6.1 Section of the river Pregal in Koenigsberg and Euler's graph.

Definitions and Terminology

A graph, G, consists of two sets V and E. V is a finite non-empty set of *vertices*. E is a set of pairs of vertices, these pairs are called *edges*. $V(G)$ and $E(G)$ will represent the sets of vertices and edges of graph G.

We will also write $G = (V,E)$ to represent a graph.

In an *undirected graph* the pair of vertices representing any edge is unordered . Thus, the pairs (v_1, v_2) and (v_2, v_1) represent the same edge.

In a *directed graph* each edge is represented by a directed pair (v_1, v_2). v_1 is the *tail* and v_2 the *head* of the edge. Therefore $<v_2, v_1>$ and $<v_1, v_2>$ represent two different edges. Figure 6.2 shows three graphs G_1, G_2 and G_3.

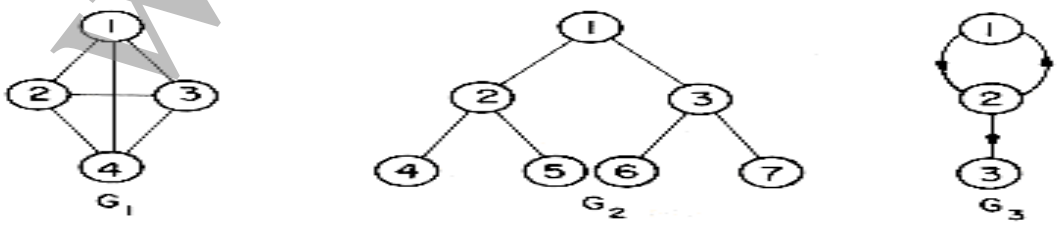


Figure 6.2 Three sample graphs.

The graphs G_1 and G_2 are undirected. G_3 is a directed graph.

$V(G_1) = \{1,2,3,4\}$; $E(G_1) = \{(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)\}$

$V(G_2) = \{1,2,3,4,5,6,7\}$; $E(G_2) = \{(1,2),(1,3),(2,4),(2,5),(3,6),(3,7)\}$

$V(G_3) = \{1,2,3\}$; $E(G_3) = \{<1,2>, <2,1>, <2,3>\}$.

Note that the edges of a directed graph are drawn with an arrow from the tail to the head. The graph G_2 is also a tree while the graphs G_1 and G_3 are not. Trees can be defined as a special case of graphs,

In addition, since $E(G)$ is a set, a graph may not have multiple occurrences of the same edge. When this restriction is removed from a graph, the resulting data object is referred to as a multigraph. The data object of figure 6.3 is a multigraph which is not a graph.

The number of distinct unordered pairs (v_i,v_j) with v_i v_j in a graph with n vertices is $n(n - 1)/2$. This is the maximum number of edges in any n vertex undirected graph.

An n vertex undirected graph with exactly $n(n - 1)/2$ edges is said to be *complete*. G_1 is the complete graph on 4 vertices while G_2 and G_3 are not complete graphs. In the case of a directed graph on n vertices the maximum number of edges is $n(n - 1)$.

If (v_1,v_2) is an edge in $E(G)$, then we shall say the vertices v_1 and v_2 are *adjacent* and that the edge (v_1,v_2) is *incident* on vertices v_1 and v_2. The vertices adjacent to vertex 2 in G_2 are 4, 5 and 1. The edges incident on vertex 3 in G_2 are (1,3), (3,6) and (3,7). If $<v_1,v_2>$ is a directed edge, then vertex v_1 will be said to be *adjacent to* v_2 while v_2 is *adjacent from* v_1. The edge $<v_1,v_2>$ is incident to v_1 and v_2. In G_3 the edges incident to vertex 2 are <1,2>, <2,1> and <2,3>.

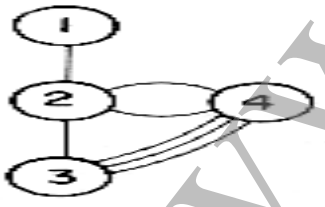


Figure 6.3 Example of a multigraph that is not a graph.

A *subgraph* of G is a graph G' such that $V(G')$ $V(G)$ and $E(G')$ $E(G)$. Figure 6.4 shows some of the subgraphs of G_1 and G_3.

A *path* from vertex v_p to vertex v_q in graph G is a sequence of vertices $v_p,v_{i1},v_{i2}, ...,v_{in},v_q$ such that $(v_p,v_{i1}),(v_{i1},v_{i2}), ...,(v_{in},v_q)$ are edges in $E(G)$. If G' is directed then the path consists of $<v_p,v_{i1}>,<v_i,v_{i2}>, ...,<v_{in},v_q>$, edges in $E(G')$.

The *length* of a path is the number of edges on it.

A *simple path* is a path in which all vertices except possibly the first and last are distinct. A path such as (1,2) (2,4) (4,3) we write as 1,2,4,3. Paths 1,2,4,3 and 1,2,4,2 are both of length 3 in G_1. The first is a simple path while the second is not. 1,2,3 is a simple directed path in G_3. 1,2,3,2 is not a path in G_3 as the edge <3,2> is not in $E(G_3)$.

A *cycle* is a simple path in which the first and last vertices are the same. 1,2,3,1 is a cycle in G_1. 1,2,1 is a cycle in G_3. For the case of directed graphs we normally add on the prefix "directed" to the terms cycle and path.

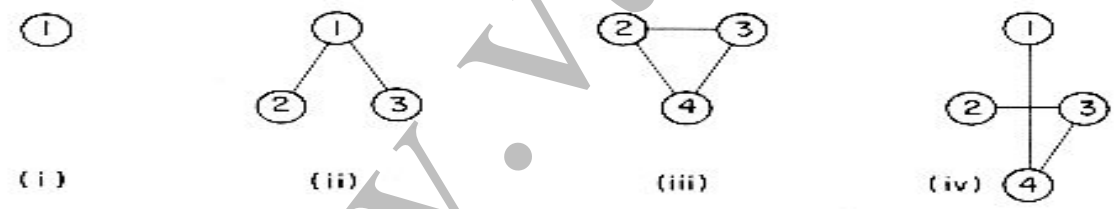
In an undirected graph, G, two vertices v_1 and v_2 are said to be *connected* if there is a path in G from v_1 to v_2 (since G is undirected, this means there must also be a path from v_2 to v_1). An undirected graph is said to be connected if for every pair of distinct vertices v_i, v_i in $V(G)$ there is a path from v_i to v_j in G.

Graphs G_1 and G_2 are connected while G_4 of figure 6.5 is not.

A *connected component* or simply a component of an undirected graph is a *maximal* connected subgraph. G_4 has two components H_1 and H_2 (see figure 6.5).

A *tree* is a connected acyclic (i.e., has no cycles) graph . A directed graph G is said to be *strongly connected* if for every pair of distinct vertices v_i, v_j in $V(G)$ there is a directed path from v_i to v_j and also from v_j to v_i. The graph G_3 is not strongly connected as there is no path from v_3 to v_2.

A *strongly connected component* is a maximal subgraph that is strongly connected. G_3 has two strongly connected components.



(a) Some of the subgraphs of G_1

(b) Some of the subgraphs of G_3

Figure 6.4 (a) Subgraphs of G_1 and (b) Subgraphs of G_3

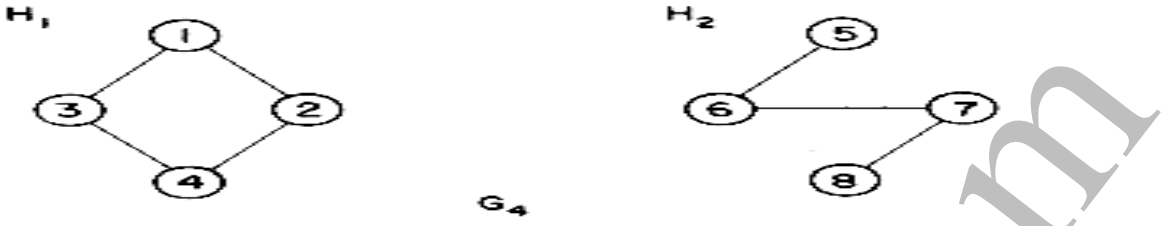


Figure 6.5 A graph with two connected components.

Figure 6.6 strongly connected components of G_3.

The degree of a vertex is the number of edges incident to that vertex. The degree of vertex 1 in G_1 is 3. In case G is a directed graph, we define the *in-degree* of a vertex v to be the number of edges for which v is the head. The *out-degree* is defined to be the number of edges for which v is the tail. Vertex 2 of G_3 has in-degree 1, out-degree 2 and degree 3. If d_i is the degree of vertex i in a graph G with n vertices and e edges, then it is easy to see that $e = (1/2) \sum_{i=1}^{n} d_i$

6.1 Binary Search Trees

Characteristics

Trees in which the key of an internal node is greater than the keys in its left subtree and is smaller than the keys in its right subtree.

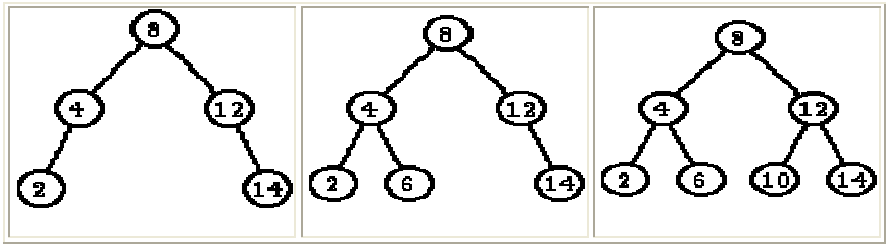
Search

```
search ( tree,key )
  IF empty tree          THEN return not-found
  IF key == value in root  THEN return found
  IF key  > value in root  THEN search (left-subtree,  key)
 search (right-subtree, key)
```

Time: O(depth of tree)

Insertion

	insert 6	insert 10

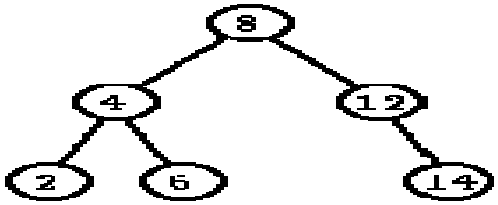


Deletion

The way the deletion is made depends on the type of node holding the key.

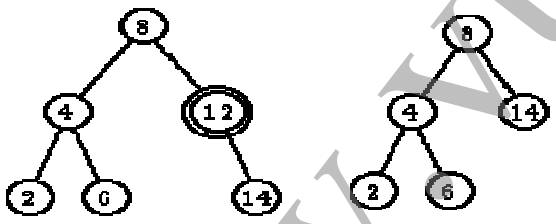
Node of degree 0

Delete the node



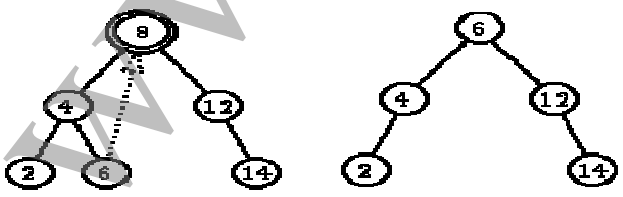
Node of degree 1

Delete the node, while connecting its predecessor to the successor.



Node of degree 2

Replace the node containing the deleted key with the node having the largest key in the left subtree, or with the node having the smallest key in the right subtree.

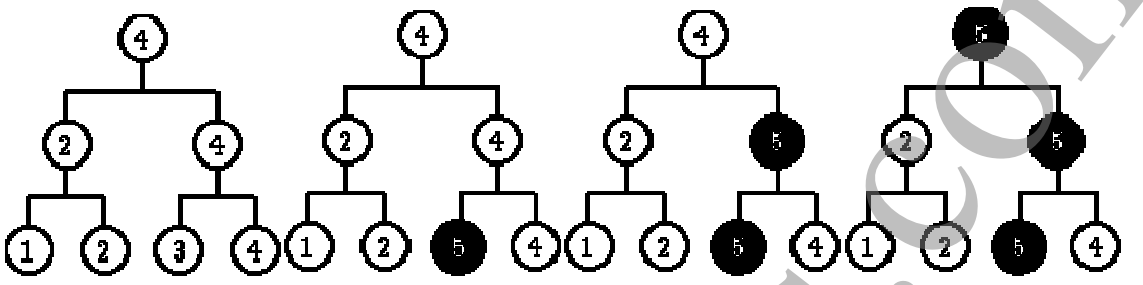


6.2. Selection Trees

A **selection tree** is a complete binary tree in which the leaf nodes hold a set of keys, and each internal node holds the "winner" key among its children.

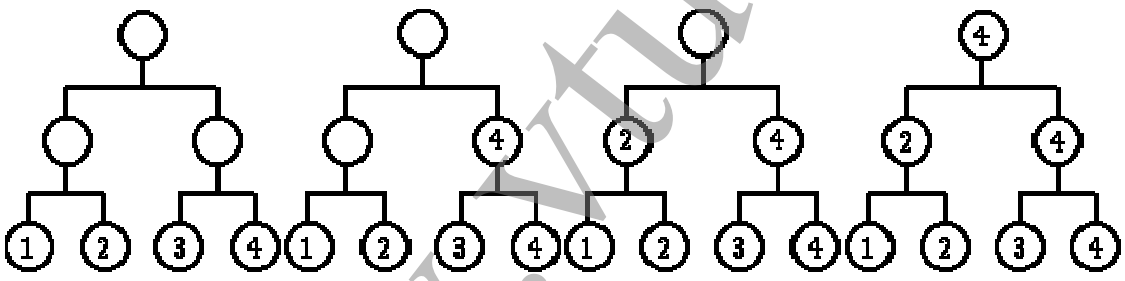
Modifying a Key

It takes O(log n) time to modify a selection tree in response to a change of a key in a leaf.



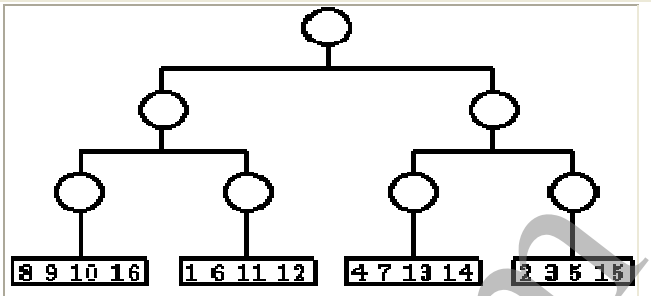
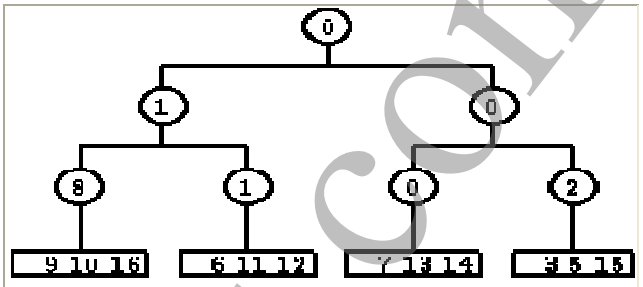
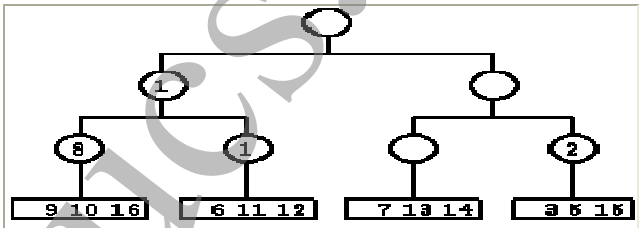
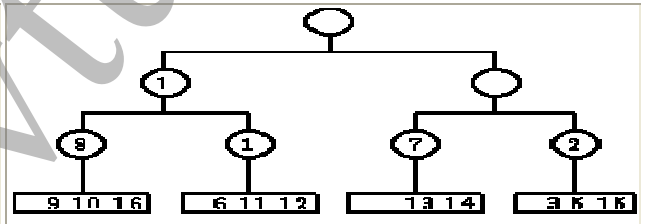
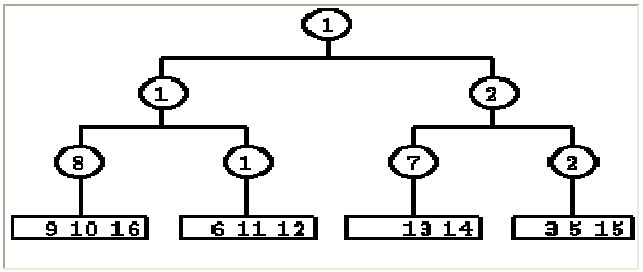
Initialization

The construction of a selection tree from scratch takes O(n) time by traversing it level-wise from bottom up.



Application: External Sort

Given a set of n values	16 9 10 8 6 11 12 1 4 7 14 13 2 15 5 3				n = 16
divide it into M chuncks,	16 9 10 8	6 11 12 1	4 7 14 13	2 15 5 3	M = 4
internally sort each chunk,	8 9 10 16	1 6 11 12	4 7 13 14	2 3 5 15	

construct complete binary tree of M leaves with the chunks attached to the leaves.	
Convert the tree into a selection tree with the keys being fed to the leaves from the chunks	
Remove the winner from the tree	
Feed to the empty leaf the next value from its corresponding chunk	
Adjust the selection tree to the change in the leaf	
Repeat the deletion sub process until all the values are consumed.	

- The algorithm takes $O(M\frac{n}{M}\log\frac{n}{M})$ time to internally sort the elements of the chunks, O(M) to initialize the selection tree, and O(n log M) to perform the selection sort. For M « n the total time complexity is O(n log n).
- To reduce I/O operations, inputs from the chunks to the selection tree should go through buffers.

6.3 Forests

The default interdomain trust relationships are created by the system during domain controller creation. The number of trust relationships that are required to connect *n* domains is *n* –1, whether the domains are linked in a single, contiguous parent-child hierarchy or they constitute two or more separate contiguous parent-child hierarchies.

When it is necessary for domains in the same organization to have different namespaces, create a separate tree for each namespace. In Windows 2000, the roots of trees are linked automatically by two-way, transitive trust relationships. Trees linked by trust relationships form a forest A single tree that is related to no other trees constitutes a forest of one tree.

The tree structures for the entire Windows 2000 forest are stored in Active Directory in the form of parent-child and tree-root relationships. These relationships are stored as trust account objects (class *trustedDomain*) in the System container within a specific domain directory partition. For each domain in a forest, information about its connection to a parent domain (or, in the case of a tree root, to another tree root domain) is added to the configuration data that is replicated to every domain in the forest. Therefore, every domain controller in the forest has knowledge of the tree structure for the entire forest, including knowledge of the links between trees. You can view the tree structure in Active Directory Domain Tree Manager.

6.4 Representation of Disjoint Sets

Set

In computer science, a **set** is an abstract data structure that can store certain values, without any particular order, and no repeated values. It is a computer implementation of the mathematical concept of a finite set. Some set data structures are designed for **static sets** that do not change with time, and allow only query operations — such as checking whether a given value is in the set, or enumerating the values in some arbitrary order. Other variants, called **dynamic** or **mutable sets**, allow also the insertion and/or deletion of elements from the set.

A set can be implemented in many ways. For example, one can use a list, ignoring the order of the elements and taking care to avoid repeated values. Sets are often implemented using various flavors of trees, tries, hash tables, and more.

A set can be seen, and implemented, as a (partial) associative array, in which the value of each key-value pair has the unit type. In type theory, sets are generally identified with their indicator function: accordingly, a set of values of type may be denoted by or . (Subtypes and subsets may be modeled by refinement types, and quotient sets may be replaced by setoids.)

Operations

Typical operations that may be provided by a static set structure *S* are

• element_of(*x*,*S*): checks whether the value *x* is in the set *S*.
• empty(*S*): checks whether the set *S* is empty.
• size(*S*): returns the number of elements in *S*.
• enumerate(*S*): yields the elements of *S* in some arbitrary order.
• pick(*S*): returns an arbitrary element of *S*.
• build(*x*1,*x*2,…,*xn*,): creates a set structure with values *x*1,*x*2,…,*xn*.

The enumerate operation may return a list of all the elements, or an iterator, a procedure object that returns one more value of *S* at each call.

Dynamic set structures typically add:

• create(*n*): creates a new set structure, initially empty but capable of holding up to *n* elements.

• add(*S*,*x*): adds the element *x* to *S*, if it is not there already.
• delete(*S*,*x*): removes the element *x* from *S*, if it is there.
• capacity(*S*): returns the maximum number of values that *S* can hold.
Some set structures may allow only some of these operations. The cost of each operation will depend on the implementation, and possibly also on the particular values stored in the set, and the order in which they are inserted. There are many other operations that can (in principle) be defined in terms of the above, such as:
• pop(*S*): returns an arbitrary element of *S*, deleting it from *S*.
• find(*S*, *P*): returns an element of *S* that satisfies a given predicate *P*.
• clear(*S*): delete all elements of *S*.
In particular, one may define the Boolean operations of set theory:
• union(*S*,*T*): returns the union of sets *S* and *T*.
• intersection(*S*,*T*): returns the intersection of sets *S* and *T*.
• difference(*S*,*T*): returns the difference of sets *S* and *T*.
• subset(*S*,*T*): a predicate that tests whether the set *S* is a subset of set *T*.
Other operations can be defined for sets with elements of a special type:
• sum(*S*): returns the sum of all elements of *S* (for some definition of "sum").
• nearest(*S*,*x*): returns the element of *S* that is closest in value to *x* (by some criterion).
In theory, many other abstract data structures can be viewed as set structures with additional operations and/or additional axioms imposed on the standard operations. For example, an abstract heap can be viewed as a set structure with a min(*S*) operation that returns the element of smallest value.

Implementations

Sets can be implemented using various data structures, which provide different time and space trade-offs for various operations. Some implementations are designed to improve the efficiency of very specialized operations, such as nearest or union. Implementations described as "general use" typically strive to optimize the element_of, add, and delete operation.

Sets are commonly implemented in the same way as associative arrays, namely, a self-balancing binary search tree for sorted sets (which has O(log n) for most operations), or a hash table for unsorted sets (which has O(1) average-case, but O(n) worst-case, for most operations). A sorted linear hash table[1] may be used to provide deterministically ordered sets.

Other popular methods include arrays. In particular a subset of the integers 1..*n* can be implemented efficiently as an *n*-bit bit array, which also support very efficient union and intersection operations. A Bloom map implements a set probabilistically, using a very compact representation but risking a small chance of false positives on queries. The Boolean set operations can be implemented in terms of more elementary operations (pop, clear, and add), but specialized algorithms may yield lower asymptotic time bounds. If sets are implemented as sorted lists, for example, the naive algorithm for union(*S*,*T*) will take code proportional to the length *m* of *S* times the length *n* of *T*; whereas a variant of the list merging algorithm will do the job in time proportional to *m*+*n*. Moreover, there are specialized set data structures (such as the union-find data structure) that are optimized for one or more of these operations, at the expense of others.

6.5 Counting Binary Trees:

Definition: A binary tree has a special vertex called its root. From this vertex at the top, the rest of the tree is drawn downward.Each vertex may have a left child and/or a right child.
Example. The number of binary trees with 1, 2, 3 vertices is:
Example. The number of binary trees with 4 vertices is:

Conjecture: The number of binary trees on n vertices is .
Proof: Every binary tree either:
! Has no vertices (x0) –or–
! Breaks down as one root vertex (x)
along with two binary trees beneath (B(x)2).
Therefore, the generating function for binary trees satisfies B(x) = 1 + xB(x)2. We conclude bn = 1 n+1#2n n $.
Another way: Find a recurrence for bn. Note:
b4 = b0b3 + b1b2 + b2b1 + b3b0.
In general, bn =%n−1 i=0 bibn−1−i .
Therefore, B(x) equals 1 + & n"1 '&n−1 i=0 bibn−1−I (xn = 1 + x & n"1 '&n−1 i=0 bibn−1−I (xn−1 = 1+x & k"0 '&k i=0 bibk−I (xk = 1+x '& k"0 bkxk ('& k"0 bkxk (= 1+xB(x)2.

6.6 The Graph Abstract Data Type:

In computer science, a **graph** is an abstract data type that is meant to implement the graph and hypergraph concepts from mathematics.A graph data structure consists of a finite (and possibly mutable) set of ordered pairs, called **edges** or **arcs**, of certain entities called **nodes** or **vertices**. As in mathematics, an edge (*x*,*y*) is said to **point** or **go from** *x* **to** *y*. The nodes may be part of the graph structure, or may be external entities represented by integer indices or references.

A graph data structure may also associate to each edge some **edge value**, such as a symbolic label or a numeric attribute (cost, capacity, length, etc.).

Algorithms

Graph algorithms are a significant field of interest within computer science. Typical higher-level operations associated with graphs are: finding a path between two nodes, like depth-first search and breadth-first search and finding the shortest path from one node to another, like Dijkstra's algorithm. A solution to finding the shortest path from each node to every other node also exists in the form of the Floyd–Warshall algorithm.A directed graph can be seen as a flow network, where each edge has a capacity and each edge receives a flow. The Ford–Fulkerson algorithm is used to find out the maximum flow from a source to a sink in a graph

Operations

The basic operations provided by a graph data structure *G* usually include:

- adjacent(*G*, *x*, *y*): tests whether there is an edge from node *x* to node *y*.
- neighbors(*G*, *x*): lists all nodes *y* such that there is an edge from *x* to *y*.
- add(*G*, *x*, *y*): adds to *G* the edge from *x* to *y*, if it is not there.
- delete(*G*, *x*, *y*): removes the edge from *x* to *y*, if it is there.
- get_node_value(*G*, *x*): returns the value associated with the node *x*.
- set_node_value(*G*, *x*, *a*): sets the value associated with the node *x* to *a*.

Structures that associate values to the edges usually also provide:

- get_edge_value(*G*, *x*, *y*): returns the value associated to the edge (*x*,*y*).
- set_edge_value(*G*, *x*, *y*, *v*): sets the value associated to the edge (*x*,*y*) to *v*.

Representations

Different data structures for the representation of graphs are used in practice:

- **Adjacency list** – Vertices are stored as records or objects, and every vertex stores a list of adjacent vertices. This data structure allows the storage of additional data on the vertices.
- **Incidence list** – Vertices and edges are stored as records or objects. Each vertex stores its incident edges, and each edge stores its incident vertices. This data structure allows the storage of additional data on vertices and edges.
- **Adjacency matrix** – A two-dimensional matrix, in which the rows represent source vertices and columns represent destination vertices. Data on edges and vertices must be stored externally. Only the cost for one edge can be stored between each pair of vertices.
- **Incidence matrix** – A two-dimensional Boolean matrix, in which the rows represent the vertices and columns represent the edges. The entries indicate whether the vertex at a row is incident to the edge at a column.

6.7. RECOMMENDED QUESTIONS

1. Define forest. Explain the forest traversals.

2. Define Binary search tree. Explain with example?

3. Define a path in a tree, terminal nodes in a tree

4. Why it is said that searching a node in a binary search tree is efficient than that of a simple binary tree?

5. List the applications of set ADT.

6. What do you mean by disjoint set ADT

7. List the abstract operations in the set.

8. Define Graph. What is a directed graph & undirected graph?

9. What is a weighted graph? Define path in a graph?

10. Define outdegree of a graph & indegree of a graph?

UNIT – 7 : PRIORITY QUEUES

Priority Queue:

Need for priority queue:

- In a multi user environment, the operating system scheduler must decide which of several processes to run only for a fixed period for time.
- For that we can use the algorithm of QUEUE, where Jobs are initially placed at the end of the queue.
- The scheduler will repeatedly take the first job on the queue, run it until either it finishes or its time limit is up, and placing it at the and of the queue if it doesn't finish.
- This strategy is generally not approximate, because very short jobs will soon to take a long time because of the wait involved to run.
- Generally, it is important that short jobs finish as fast as possible, so these jobs should have precedence over jobs that have already been running.
- Further more, some jobs that are not short are still very important and should also have precedence.
- This particular application seems to require a special kind of queue, known as a PRIORITY QUEUE.

Priority Queue:

It is a collection of ordered elements that provides fast access to the minimum or maximum element.

Basic Operations performed by priority queue are:

1. Insert operation
2. Deletemin operation

Insert operation is the equivalent of queue's ***Enqueue*** operation.

Deletemin operation is the priority queue equivalent of the queue's ***Dequeue*** operation.

Implementation:

There are three ways for implementing priority queue. They are:

1. Linked list
2. Binary Search tree
3. Binary Heap

7.1. Single- and Double-Ended Priority Queues:

A (single-ended) priority queue is a data type supporting the following operations on an ordered set of values:
1) find the maximum value (FindMax);
2) delete the maximum value (DeleteMax);
3) add a new value x (Insert(x)).
Obviously, the priority queue can be redefined by substituting operations 1) and 2) with FindMin and DeleteMin, respectively. Several structures, some implicitly stored in an array and some using more complex data structures, have been presented for implementing this data type, including max-heaps (or min-heaps)

Conceptually, a max-heap is a binary tree having the following properties:
a) heap-shape: all leaves lie on at most two adjacent levels, and the leaves on the last level occupy the leftmost positions; all other levels are complete.
b) max-ordering: the value stored at a node is greater than or equal to the values stored at its children. A max-heap of size n can be constructed in linear time and can be stored in an n-element array; hence it is referred to as an implicit data structure [g].

When a max-heap implements a priority queue, FindMax can be performed in constant time, while both DeleteMax and Insert(x) have logarithmic time. We shall consider a more powerful data type, the double-ended priority queue, which allows both FindMin and FindMax, as well as DeleteMin, DeleteMax, and Insert(x) operations. An important application of this data type is in external quicksort .
A traditional heap does not allow efficient implementation of all the above operations; for example, FindMin requires linear (instead of constant) time in a max-heap. One approach to overcoming this intrinsic limitation of heaps, is to place a max-heap "back-to-back" with a min-heap.

Definition
A **double-ended priority queue (DEPQ)** is a collection of zero or more elements. Each element has a priority or value. The operations performed on a double-ended priority queue are:

1. isEmpty() ... return true iff the DEPQ is empty
2. size() ... return the number of elements in the DEPQ
3. getMin() ... return element with minimum priority
4. getMax() ... return element with maximum priority
5. put(x) ... insert the element x into the DEPQ
6. removeMin() ... remove an element with minimum priority and return this element
7. removeMax() ... remove an element with maximum priority and return this element

Application to External Sorting

The internal sorting method that has the best expected run time is quick sort (see Section 19.2.3). The basic idea in quick sort is to partition the elements to be sorted into three groups L, M, and R. The middle group M contains a single element called the **pivot**, all elements in the left group L are <= the pivot, and all elements in the right group R are >= the pivot. Following this partitioning, the left and right element groups are sorted recursively.

In an external sort, we have more elements than can be held in the memory of our computer. The elements to be sorted are initially on a disk and the sorted sequence is to be left on the disk. When the internal quick sort method outlined above is extended to an external quick sort, the middle group M is made as large as possible through the use of a DEPQ. The external quick sort strategy is:

1. Read in as many elements as will fit into an internal DEPQ. The elements in the DEPQ will eventually be the middle group of elements.
2. Read in the remaining elements. If the next element is <= the smallest element in the DEPQ, output this next element as part of the left group. If the next element is >= the largest element in the DEPQ, output this next element as part of the right group. Otherwise, remove either the max or min element from the DEPQ (the choice may be made randomly or alternately); if the max element is removed, output it as part of the right group; otherwise, output the removed element as part of the left group; insert the newly input element into the DEPQ.
3. Output the elements in the DEPQ, in sorted order, as the middle group.
4. Sort the left and right groups recursively.

Generic Methods for DEPQs:
General methods exist to arrive at efficient DEPQ data structures from single-ended priority queue (PQ) data structures that also provide an efficient implementation of the remove(theNode) operation (this operation removes the node theNode from the PQ). The simplest of these methods, **dual structure method**, maintains both a min PQ and a max PQ of all the DEPQ elements together with **correspondence pointers** between the nodes of the min PQ and the max PQ that contain the same element. Figure 1 shows a dual heap structure for the elements 6, 7, 2, 5, 4. Correspondence pointers are shown as red arrows.

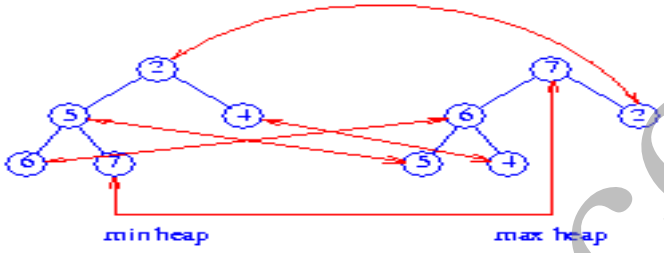


Figure 1 Dual heap

Although the figure shows each element stored in both the min and the max heap, it is necessary to store each element in only one of the two heaps.

The isEmpty and size operations are implemented by using a variable size that keeps track of the number of elements in the DEPQ. The minimum element is at the root of the min heap and the maximum element is at the root of the max heap. To insert an element x, we insert x into both the min and the max heaps and then set up correspondence pointers between the locations of x in the min and max heaps. To remove the minimum element, we do a removeMin from the min heap and a remove(theNode), where theNode is the corresponding node for the removed element, from the max heap. The maximum element is removed in an analogous way.

Total and leaf correspondence are more sophisticated correspondence methods. In both of these, half the elements are in the min PQ and the other half in the max PQ. When the number of elements is odd, one element is retained in a buffer. This buffered element is not in either PQ. In total correspondence, each element a in the min PQ is paired with a distinct element b of the max PQ. (a,b) is a corresponding pair of elements such that priority(a) <= priority(b). Figure 2 shows a total correspondence heap for the 11 elements 2, 3, 4, 4, 5, 5, 6, 7, 8, 9, 10. The element 9 is in the buffer. Corresponding pairs are shown by red arrows.

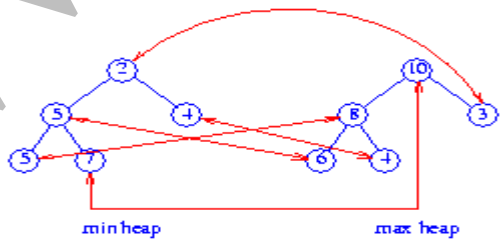


Figure 2 Total correspondence heap

In leaf correspondence, each leaf element of the min and max PQ is required to be part of a corresponding pair. Nonleaf elements need not be in any corresponding pair. Figure 3 shows a leaf correspondence heap.

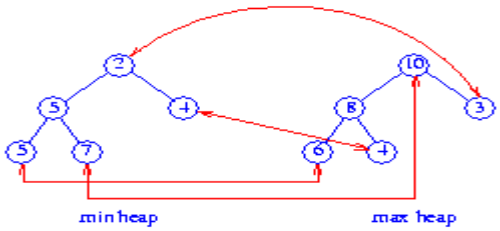


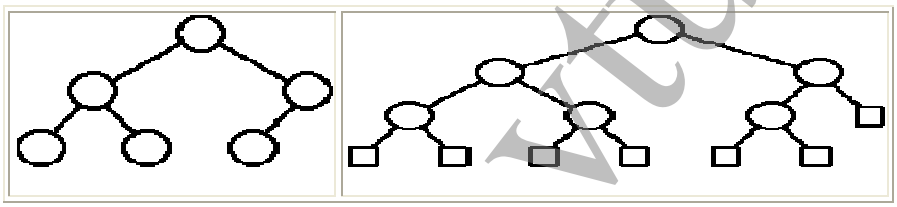
Figure 3 A leaf correspondence heap

Total and leaf correspondence structures require less space than do dual structures. However, the DEPQ algorithms for total and leaf correspondence structures are more complex than those for dual structures. Of the three correspondence types, leaf correspondence generally results in the fastest DEPQ correspondence structures.

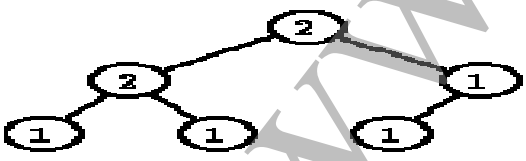
Using any of the described correspondence methods, we can arrive at DEPQ structures from heaps, height biased leftist trees, and pairing heaps. In these DEQP structures, the operations put(x), removeMin(), and removeMax() take O(log n) time (n is the number of elements in the DEPQ, for pairing heaps, this is an amortized complexity), and the remaining DEPQ operations take O(1) time.

7.2. Leftist tree:

Definitions:An **external node** is an imaginary node in a location of a missing child.



Notation. Let the **s-value** of a node be the shortest distance from the node to an external node.

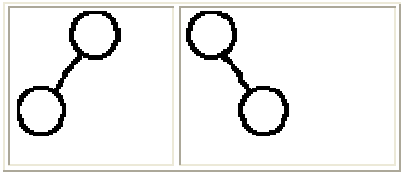


- An external node has the s-value of 0.
- An internal node has the s-value of 1 plus the minimum of the s-values of its internal and external children.

Height-Biased Leftist Trees

In a **height-biased leftist tree** the s-value of a left child of a node is not smaller than the s-value of the right child of the node.

height-biased	non height-biased

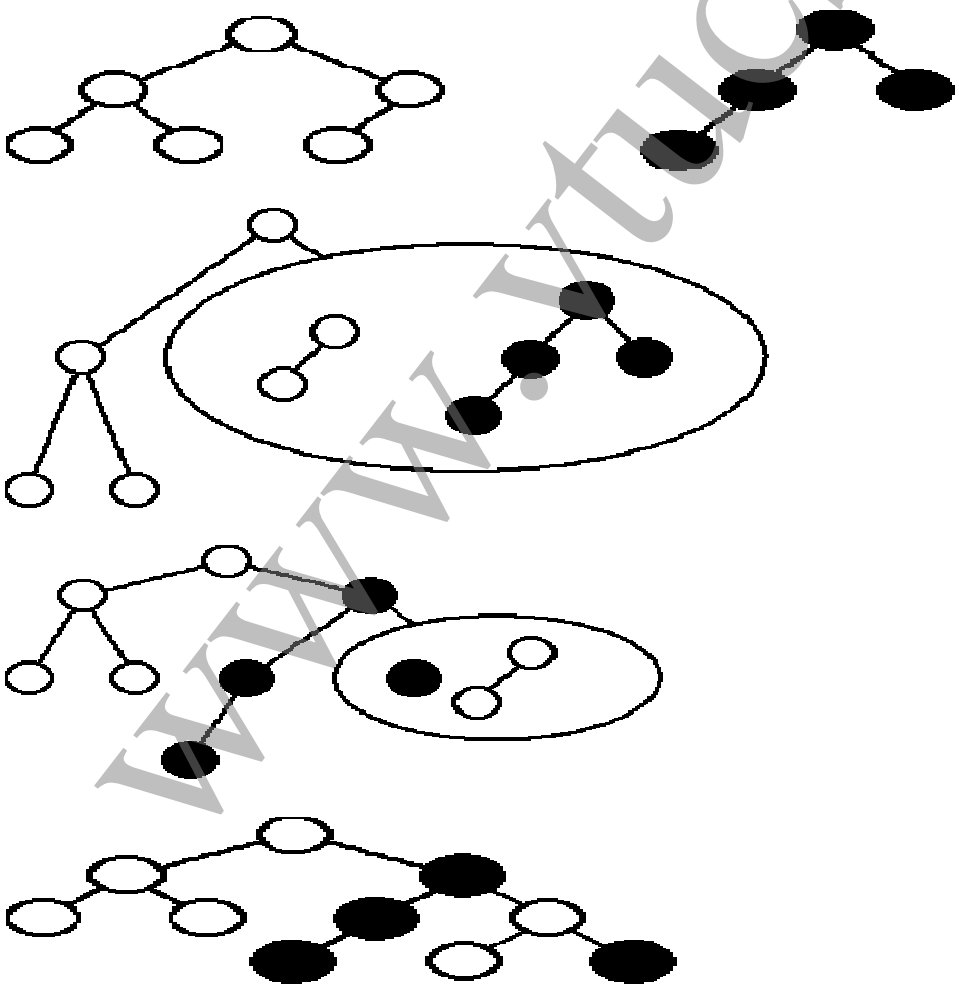


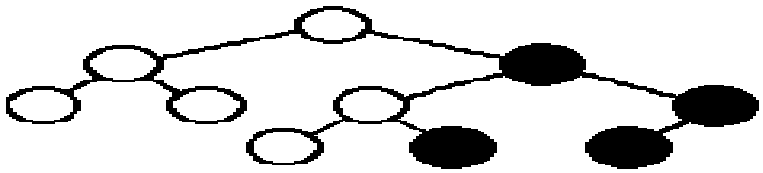
Merging Height-Biased Leftist Trees

Recursive algorithm

- Consider two nonempty height-biased leftist trees A and B, and a relation (e.g., smaller than) on the values of the keys.
- Assume the key-value of A is not bigger than the key-value of B
- Let the root of the merged tree have the same left subtree as the root of A
- Let the root of the merged tree have the right subtree obtained by merging B with the right subtree of A.
- If in the merged tree the s-value of the left subtree is smaller than the s-value of the right subtree, interchange the subtrees.

For the following example, assume the key-value of each node equals its s-value.





Time complexity

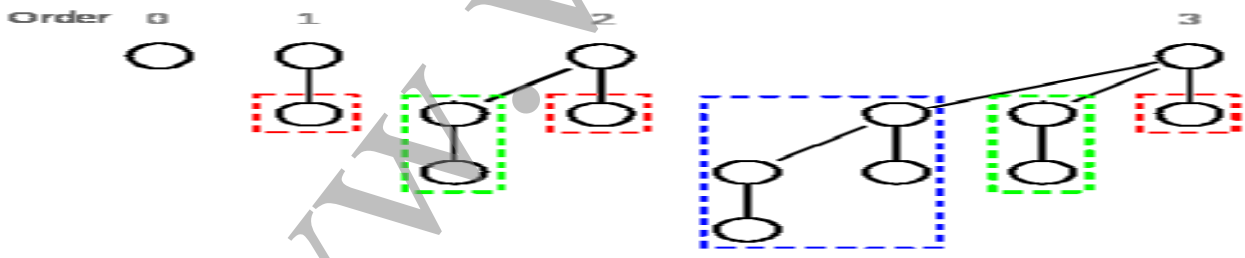
- Linear in the rightmost path of the outcome tree.
- The rightmost path of the the outcome tree is a shortest path
- A shortest path can't contain more than log n nodes.

  Proof If the shortest path can contain more than log n nodes, then the first 1 + log n levels should include $2^0 + 2^1 + \ldots + 2^{\log n} = 2^{1+\log n} - 1 = 2n - 1$ nodes. In such a case, for $n > 1$ we end up with $n > 2n - 1$.

7.3. Binomial Heaps:

Binomial heap is a heap similar to a binary heap but also supports quickly merging two heaps. This is achieved by using a special tree structure. It is important as an implementation of the **mergeable heap** abstract data type (also called meldable heap), which is a priority queue supporting merge operation. A binomial heap is implemented as a collection of binomial trees (compare with a binary heap, which has a shape of a single binary tree). A **binomial tree** is defined recursively:

- A binomial tree of order 0 is a single node
- A binomial tree of order *k* has a root node whose children are roots of binomial trees of orders *k*–1, *k*–2, ..., 2, 1, 0 (in this order).



Binomial trees of order 0 to 3: Each tree has a root node with subtrees of all lower ordered binomial trees, which have been highlighted. For example, the order 3 binomial tree is connected to an order 2, 1, and 0 (highlighted as blue, green and red respectively) binomial tree.

A binomial tree of order *k* has 2^k nodes, height *k*.

Because of its unique structure, a binomial tree of order *k* can be constructed from two trees of order *k*–1 trivially by attaching one of them as the leftmost child of root of the other one. This feature is central to the *merge* operation of a binomial heap, which is its major advantage over other conventional heaps.The name comes from the shape: a binomial tree of order n has $\binom{n}{d}$ nodes at depth d. (See Binomial coefficient.)

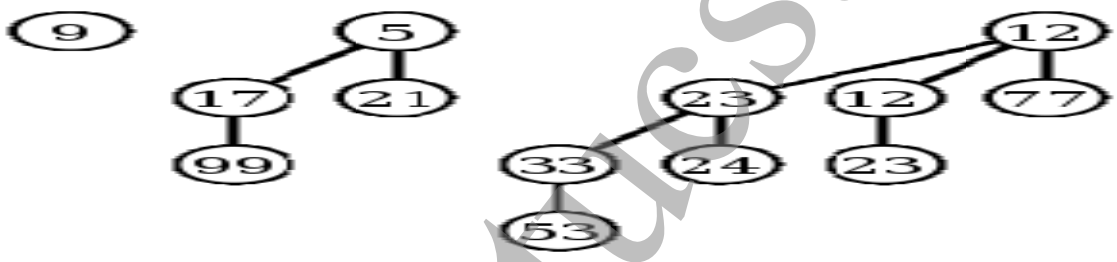
Structure of a binomial heap

A binomial heap is implemented as a set of binomial trees that satisfy the *binomial heap properties*:

- Each binomial tree in a heap obeys the *minimum-heap property*: the key of a node is greater than or equal to the key of its parent.
- There can only be either *one* or *zero* binomial trees for each order, including zero order.

The first property ensures that the root of each binomial tree contains the smallest key in the tree, which applies to the entire heap.

The second property implies that a binomial heap with n nodes consists of at most log n + 1 binomial trees. In fact, the number and orders of these trees are uniquely determined by the number of nodes n: each binomial tree corresponds to one digit in the binary representation of number n. For example number 13 is 1101 in binary, $2^3 + 2^2 + 2^0$, and thus a binomial heap with 13 nodes will consist of three binomial trees of orders 3, 2, and 0 (see figure below).



Example of a binomial heap containing 13 nodes with distinct keys. The heap consists of three binomial trees with orders 0, 2, and 3.

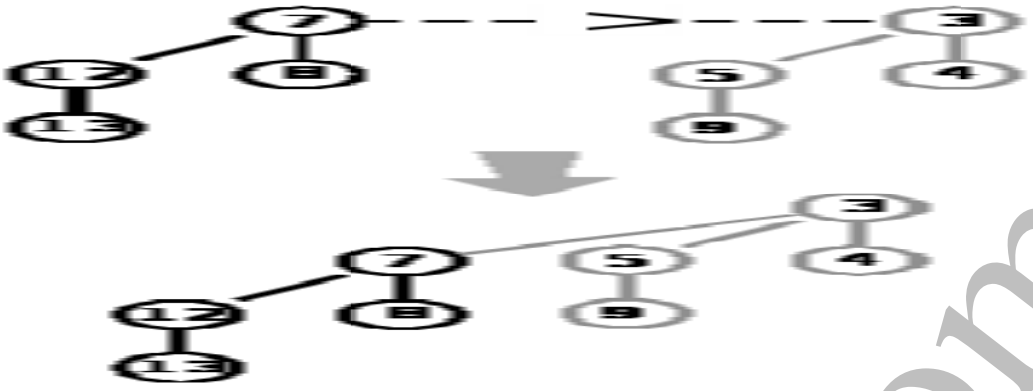
Implementation

Because no operation requires random access to the root nodes of the binomial trees, the roots of the binomial trees can be stored in a linked list, ordered by increasing order of the tree.

Merge

As mentioned above, the simplest and most important operation is the merging of two binomial trees of the same order within two binomial heaps. Due to the structure of binomial trees, they can be merged trivially. As their root node is the smallest element within the tree, by comparing the two keys, the smaller of them is the minimum key, and becomes the new root node. Then the other tree become a subtree of the combined tree. This operation is basic to the complete merging of two binomial heaps.

```
function mergeTree(p, q)
   if p.root.key <= q.root.key
      return p.addSubTree(q)
   else
      return q.addSubTree(p)
```



To merge two binomial trees of the same order, first compare the root key. Since 7>3, the black tree on the left(with root node 7) is attached to the grey tree on the right(with root node 3) as a subtree. The result is a tree of order 3.

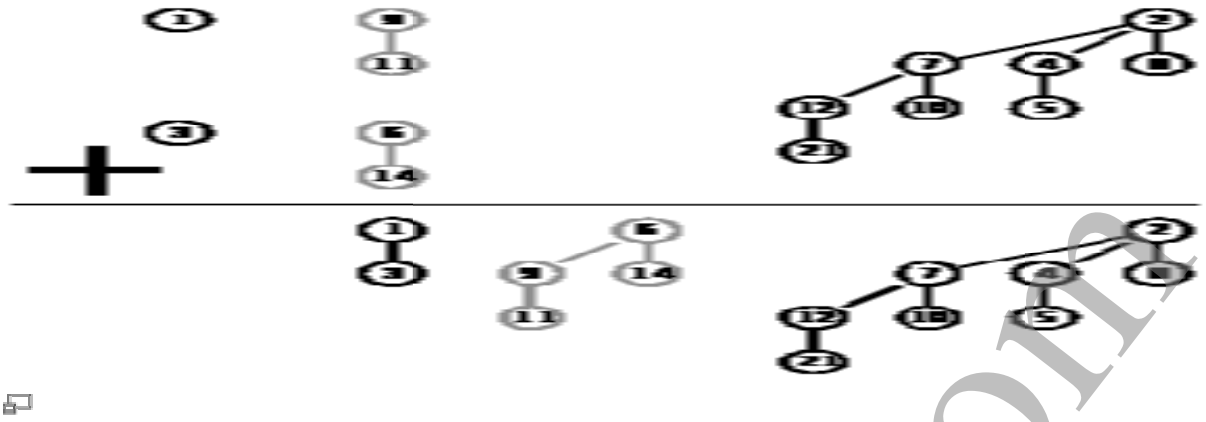
The operation of **merging** two heaps is perhaps the most interesting and can be used as a subroutine in most other operations. The lists of roots of both heaps are traversed simultaneously, similarly as in the merge algorithm.

If only one of the heaps contains a tree of order j, this tree is moved to the merged heap. If both heaps contain a tree of order j, the two trees are merged to one tree of order j+1 so that the minimum-heap property is satisfied. Note that it may later be necessary to merge this tree with some other tree of order j+1 present in one of the heaps. In the course of the algorithm, we need to examine at most three trees of any order (two from the two heaps we merge and one composed of two smaller trees).

Because each binomial tree in a binomial heap corresponds to a bit in the binary representation of its size, there is an analogy between the merging of two heaps and the binary addition of the *sizes* of the two heaps, from right-to-left. Whenever a carry occurs during addition, this corresponds to a merging of two binomial trees during the merge.

Each tree has order at most log n and therefore the running time is $O(\log n)$.

```
function merge(p, q)
   while not( p.end() and q.end() )
      tree = mergeTree(p.currentTree(), q.currentTree())
      if not heap.currentTree().empty()
         tree = mergeTree(tree, heap.currentTree())
         heap.addTree(tree)
      else
         heap.addTree(tree)
      heap.next() p.next() q.next()
```



This shows the merger of two binomial heaps. This is accomplished by merging two binomial trees of the same order one by one. If the resulting merged tree has the same order as one binomial tree in one of the two heaps, then those two are merged again.

Insert

Inserting a new element to a heap can be done by simply creating a new heap containing only this element and then merging it with the original heap. Due to the merge, insert takes O(log *n*) time,however it has an *amortized* time of O(1) (i.e. constant).

Find minimum

To find the **minimum** element of the heap, find the minimum among the roots of the binomial trees. This can again be done easily in *O*(log *n*) time, as there are just *O*(log *n*) trees and hence roots to examine.By using a pointer to the binomial tree that contains the minimum element, the time for this operation can be reduced to *O*(1). The pointer must be updated when performing any operation other than Find minimum. This can be done in *O*(log *n*) without raising the running time of any operation.

Delete minimum

To **delete the minimum element** from the heap, first find this element, remove it from its binomial tree, and obtain a list of its subtrees. Then transform this list of subtrees into a separate binomial heap by reordering them from smallest to largest order. Then merge this heap with the original heap. Since each tree has at most log *n* children, creating this new heap is *O*(log *n*). Merging heaps is *O*(log *n*), so the entire delete minimum operation is *O*(log *n*).

```
function deleteMin(heap)
   min = heap.trees().first()
   for each current in heap.trees()
      if current.root < min then min = current
   for each tree in min.subTrees()
      tmp.addTree(tree)
   heap.removeTree(min)
   merge(heap, tmp)
```

Decrease key

After **decreasing** the key of an element, it may become smaller than the key of its parent, violating the minimum-heap property. If this is the case, exchange the element with its parent, and possibly also with its grandparent, and so on, until the minimum-heap property is no longer violated. Each binomial tree has height at most log *n*, so this takes *O*(log *n*) time.

Delete

To **delete** an element from the heap, decrease its key to negative infinity (that is, some value lower than any element in the heap) and then delete the minimum in the heap.

Performance

All of the following operations work in O(log *n*) time on a binomial heap with *n* elements:

- Insert a new element to the heap
- Find the element with minimum key
- Delete the element with minimum key from the heap
- Decrease key of a given element
- Delete given element from the heap
- Merge two given heaps to one heap

Finding the element with minimum key can also be done in *O*(1) by using an additional pointer to the minimum.

Applications Discrete event simulation, Priority queues

7.4. Fibonacci Heaps:

A **Fibonacci heap** is a heap data structure consisting of a collection of trees. It has a better amortized running time than a binomial heap. Fibonacci heaps were developed by Michael L. Fredman and Robert E. Tarjan in 1984 and first published in a scientific journal in 1987. The name of Fibonacci heap comes from Fibonacci numbers which are used in the running time analysis.

Find-minimum is O(1) amortized time.Operations insert, decrease key, and merge (union) work in constant amortized time. Operations delete and delete minimum work in *O*(log *n*) amortized time. This means that starting from an empty data structure, any sequence of *a* operations from the first group and *b* operations from the second group would take $O(a + b \log n)$ time. In a binomial heap such a sequence of operations would take $O((a + b)\log (n))$ time. A Fibonacci heap is thus better than a binomial heap when *b* is asymptotically smaller than *a*.

Using Fibonacci heaps for priority queues improves the asymptotic running time of important algorithms, such as Dijkstra's algorithm for computing the shortest path between two nodes in a graph.

Structure

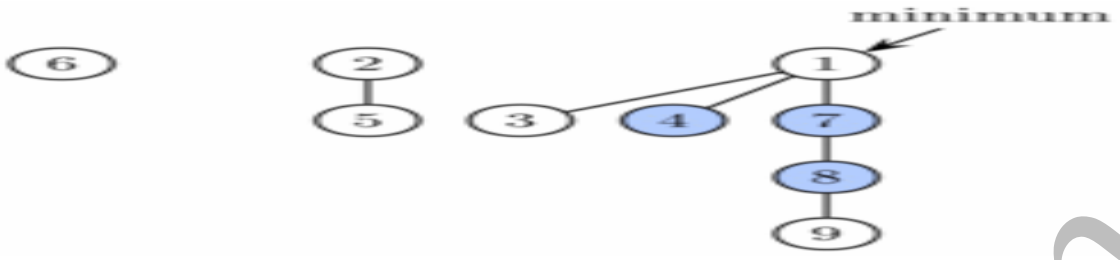


Figure 1. Example of a Fibonacci heap. It has three trees of degrees 0, 1 and 3. Three vertices are marked (shown in blue). Therefore the potential of the heap is 9.

A Fibonacci heap is a collection of trees satisfying the minimum-heap property, that is, the key of a child is always greater than or equal to the key of the parent. This implies that the minimum key is always at the root of one of the trees. Compared with binomial heaps, the structure of a Fibonacci heap is more flexible. The trees do not have a prescribed shape and in the extreme case the heap can have every element in a separate tree. This flexibility allows some operations to be executed in a "lazy" manner, postponing the work for later operations. For example merging heaps is done simply by concatenating the two lists of trees, and operation *decrease key* sometimes cuts a node from its parent and forms a new tree.

However at some point some order needs to be introduced to the heap to achieve the desired running time. In particular, degrees of nodes (here degree means the number of children) are kept quite low: every node has degree at most $O(\log n)$ and the size of a subtree rooted in a node of degree k is at least F_{k+2}, where F_k is the kth Fibonacci number. This is achieved by the rule that we can cut at most one child of each non-root node. When a second child is cut, the node itself needs to be cut from its parent and becomes the root of a new tree (see Proof of degree bounds, below). The number of trees is decreased in the operation *delete minimum*, where trees are linked together.

As a result of a relaxed structure, some operations can take a long time while others are done very quickly. In the amortized running time analysis we pretend that very fast operations take a little bit longer than they actually do. This additional time is then later subtracted from the actual running time of slow operations. The amount of time saved for later use is measured at any given moment by a potential function. The potential of a Fibonacci heap is given by

$$\text{Potential} = t + 2m$$

where t is the number of trees in the Fibonacci heap, and m is the number of marked nodes. A node is marked if at least one of its children was cut since this node was made a child of another node (all roots are unmarked).

Thus, the root of each tree in a heap has one unit of time stored. This unit of time can be used later to link this tree with another tree at amortized time 0. Also, each marked node has two units of time stored. One can be used to cut the node from its parent. If this happens, the node becomes a root and the second unit of time will remain stored in it as in any other root.

Implementation of operations

To allow fast deletion and concatenation, the roots of all trees are linked using a circular, doubly linked list. The children of each node are also linked using such a list. For each node, we maintain its number of

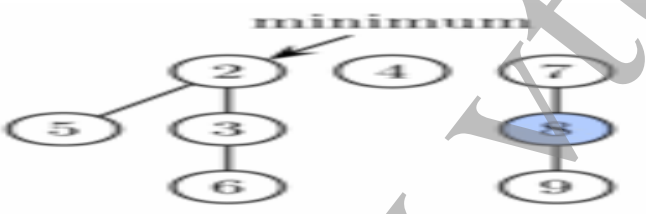
children and whether the node is marked. Moreover we maintain a pointer to the root containing the minimum key.

Operation **find minimum** is now trivial because we keep the pointer to the node containing it. It does not change the potential of the heap, therefore both actual and amortized cost is constant. As mentioned above, **merge** is implemented simply by concatenating the lists of tree roots of the two heaps. This can be done in constant time and the potential does not change, leading again to constant amortized time.

Operation **insert** works by creating a new heap with one element and doing merge. This takes constant time, and the potential increases by one, because the number of trees increases. The amortized cost is thus still constant.

Fibonacci heap from Figure 1 after first phase of extract minimum. Node with key 1 (the minimum) was deleted and its children were added as separate trees.

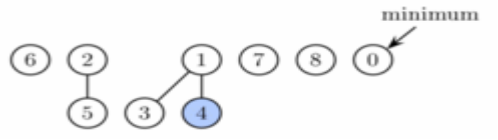
Operation **extract minimum** (same as *delete minimum*) operates in three phases. First we take the root containing the minimum element and remove it. Its children will become roots of new trees. If the number of children was d, it takes time $O(d)$ to process all new roots and the potential increases by $d-1$. Therefore the amortized running time of this phase is $O(d) = O(\log n)$.



Fibonacci heap from Figure 1 after extract minimum is completed. First, nodes 3 and 6 are linked together. Then the result is linked with tree rooted at node 2. Finally, the new minimum is found.

However to complete the extract minimum operation, we need to update the pointer to the root with minimum key. Unfortunately there may be up to n roots we need to check. In the second phase we therefore decrease the number of roots by successively linking together roots of the same degree. When two roots u and v have the same degree, we make one of them a child of the other so that the one with the smaller key remains the root. Its degree will increase by one. This is repeated until every root has a different degree. To find trees of the same degree efficiently we use an array of length $O(\log n)$ in which we keep a pointer to one root of each degree. When a second root is found of the same degree, the two are linked and the array is updated. The actual running time is $O(\log n + m)$ where m is the number of roots at the beginning of the second phase. At the end we will have at most $O(\log n)$ roots (because each has a different degree). Therefore the difference in the potential function from before this phase to after it is: $O(\log n) - m$, and the amortized running time is then at most $O(\log n + m) + O(\log n) - m = O(\log n)$. Since we can scale up the units of potential stored at insertion in each node by the constant factor in the $O(m)$ part of the actual cost for this phase.

In the third phase we check each of the remaining roots and find the minimum. This takes $O(\log n)$ time and the potential does not change. The overall amortized running time of extract minimum is therefore $O(\log n)$.



Fibonacci heap from Figure 1 after decreasing key of node 9 to 0. This node as well as its two marked ancestors are cut from the tree rooted at 1 and placed as new roots.

Operation **decrease key** will take the node, decrease the key and if the heap property becomes violated (the new key is smaller than the key of the parent), the node is cut from its parent. If the parent is not a root, it is marked. If it has been marked already, it is cut as well and its parent is marked. We continue upwards until we reach either the root or an unmarked node. In the process we create some number, say k, of new trees. Each of these new trees except possibly the first one was marked originally but as a root it will become unmarked. One node can become marked. Therefore the potential decreases by at least $k - 2$. The actual time to perform the cutting was $O(k)$, therefore the amortized running time is constant.

Finally, operation **delete** can be implemented simply by decreasing the key of the element to be deleted to minus infinity, thus turning it into the minimum of the whole heap. Then we call extract minimum to remove it. The amortized running time of this operation is $O(\log n)$.

Proof of degree bounds

The amortized performance of a Fibonacci heap depends on the degree (number of children) of any tree root being $O(\log n)$, where n is the size of the heap. Here we show that the size of the (sub)tree rooted at any node x of degree d in the heap must have size at least F_{d+2}, where F_k is the kth Fibonacci number. The degree bound follows from this and the fact (easily proved by induction) that $F_{d+2} \geq \varphi^d$ for all integers $d \geq 0$, where $\varphi = (1+\sqrt{5})/2 \doteq 1.62$. (We then have $n \geq F_{d+2} \geq \varphi^d$, and taking the log to base φ of both sides gives $d \leq \log_\varphi n$ as required.)

Consider any node x somewhere in the heap (x need not be the root of one of the main trees). Define **size**(x) to be the size of the tree rooted at x (the number of descendants of x, including x itself). We prove by induction on the height of x (the length of a longest simple path from x to a descendant leaf), that **size**(x) $\geq F_{d+2}$, where d is the degree of x.

Base case: If x has height 0, then $d = 0$, and **size**(x) $= 1 = F_2$.

Inductive case: Suppose x has positive height and degree $d>0$. Let $y_1, y_2, \ldots, y_d$ be the children of x, indexed in order of the times they were most recently made children of x (y_1 being the earliest and y_d the latest), and let $c_1, c_2, \ldots, c_d$ be their respective degrees. We **claim** that $c_i \geq i-2$ for each i with $2 \leq i \leq d$: Just before y_i was made a child of x, $y_1, \ldots, y_{i-1}$ were already children of x, and so x had degree at least $i-1$ at that time. Since trees are combined only when the degrees of their roots are equal, it must have been that y_i also had degree at least $i-1$ at the time it became a child of x. From that time to the present, y_i can only have lost at most one child (as guaranteed by the marking process), and so its current degree c_i is at least $i-2$. This proves the **claim**.

Since the heights of all the y_i are strictly less than that of x, we can apply the inductive hypothesis to them to get **size**(y_i) $\geq F_{ci+2} \geq F_{(i-2)+2} = F_i$. The nodes x and y_1 each contribute at least 1 to **size**(x), and so we have

$$\mathbf{size}(x) \geq 2 + \sum_{i=2}^{d} \mathbf{size}(y_i) \geq 2 + \sum_{i=2}^{d} F_i = 1 + \sum_{i=0}^{d} F_i.$$

A routine induction proves that $1 + \sum_{i=0}^{d} F_i = F_{d+2}$ for any $d \geq 0$, which gives the desired lower bound on **size**(x).

Worst case

Although the total running time of a sequence of operations starting with an empty structure is bounded by the bounds given above, some (very few) operations in the sequence can take very long to complete (in particular delete and delete minimum have linear running time in the worst case). For this reason Fibonacci heaps and other amortized data structures may not be appropriate for real-time systems. It is possible to create a data structure which the same worst case performance as the Fibonacci heap has amortized performance.[3] However the resulting structure is very complicated, so it is not useful in most practical cases.

7.5. Pairing heaps

Pairing heaps are a type of heap data structure with relatively simple implementation and excellent practical amortized performance. However, it has proven very difficult to determine the precise asymptotic running time of pairing heaps.

Pairing heaps are heap ordered multiway trees. Describing the various heap operations is relatively simple (in the following we assume a min-heap):

- *find-min*: simply return the top element of the heap.
- *merge*: compare the two root elements, the smaller remains the root of the result, the larger element and its subtree is appended as a child of this root.
- *insert*: create a new heap for the inserted element and *merge* into the original heap.
- *decrease-key* (optional): remove the subtree rooted at the key to be decreased then *merge* it with the heap.
- *delete-min*: remove the root and *merge* its subtrees. Various strategies are employed.

- The amortized time per *delete-min* is $O(\log n)$.[1] The operations *find-min*, *merge*, and *insert* take $O(1)$ amortized time[2] and *decrease-key* takes $2^{O(\sqrt{\log \log n})}$ amortized time.[3] Fredman proved that the amortized time per *decrease-key* is at least $\Omega(\log \log n)$.[4] That is, they are less efficient than Fibonacci heaps, which perform *decrease-key* in $O(1)$ amortized time.

Implementation

A pairing heap is either an empty heap, or a pair consisting of a root element and a possibly empty list of pairing heaps. The heap ordering property requires that all the root elements of the subheaps in the list are not smaller that then root element of the heap. The following description assumes a purely functional heap that does not support the *decrease-key* operation.

type PairingHeap[Elem] = Empty | Heap(elem: Elem, subheaps: List[PairingHeap[Elem]])

Operations

find-min

The function *find-min* simply returns the root element of the heap:

```
function find-min(heap)
  if heap == Empty
    error
  else
    return heap.elem
```

merge:

Merging with an empty heap returns the other heap, otherwise a new heap is returned that has the minimum of the two root elements as its root element and just adds the heap with the larger root to the list of subheaps:

```
function merge(heap1, heap2)
  if heap1 == Empty
    return heap2
  elsif heap2 == Empty
    return heap1
  elseif heap1.elem < heap2.elem
    return Heap(heap1.elem, heap2 :: heap1.subheaps)
  else
    return Heap(heap2.elem, heap1 :: heap2.subheaps)
```

Insert:

The easiest way to insert an element into a heap is to merge the heap with a new heap containing just this element and an empty list of subheaps:

```
function insert(elem, heap)
  return merge(Heap(elem, []), heap)
```

delete-min:

The only non-trivial fundamental operation is the deletion of the minimum element from the heap. The standard strategy first merges the subheaps in pairs (this is the step that gave this datastructure its name) from left to right and then merges the resulting list of heaps from right to left:

```
fuction delete-min(heap)
  if heap == Empty
```

```
    error
  elsif length(heap.subheaps) == 0
    return Empty
  elsif length(heap.subheaps) == 1
    return heap.subheaps[0]
  else
    return merge-pairs(heap.subheaps)
```

This uses the auxiliary function *merge-pairs*:

```
function merge-pairs(l)
  if length(l) == 0
    return Empty
  elsif length(l) == 1
    return l[0]
  else
    return merge(merge(l[0], l[1]), merge-pairs(l[2.. ]))
```

That this does indeed implement the described two-pass left-to-right then right-to-left merging strategy can be seen from this reduction:

```
   merge-pairs([H1, H2, H3, H4, H5, H6, H7])
=> merge(merge(H1, H2), merge-pairs([H3, H4, H5, H6, H7]))
     # merge H1 and H2 to H12, then the rest of the list
=> merge(H12, merge(merge(H3, H4), merge-pairs([H5, H6, H7])))
     # merge H3 and H4 to H34, then the rest of the list
=> merge(H12, merge(H34, merge(merge(H5, H6), merge-pairs([H7]))))
     # merge H5 and H5 to H56, then the rest of the list
=> merge(H12, merge(H34, merge(H56, H7)))
     # switch direction, merge the last two resulting heaps, giving H567
=> merge(H12, merge(H34, H567))
     # merge the last two resulting heaps, giving H34567
=> merge(H12, H34567)
     # finally, merge the first merged pair with the result of merging the rest
=> H1234567
```

7.6. RECOMMENDED QUESTIONS

1. Define a priority queue

2. Define a Deque (Double-Ended Queue)

3. What is the need for Priority queue?

4. What are the applications of priority queues?

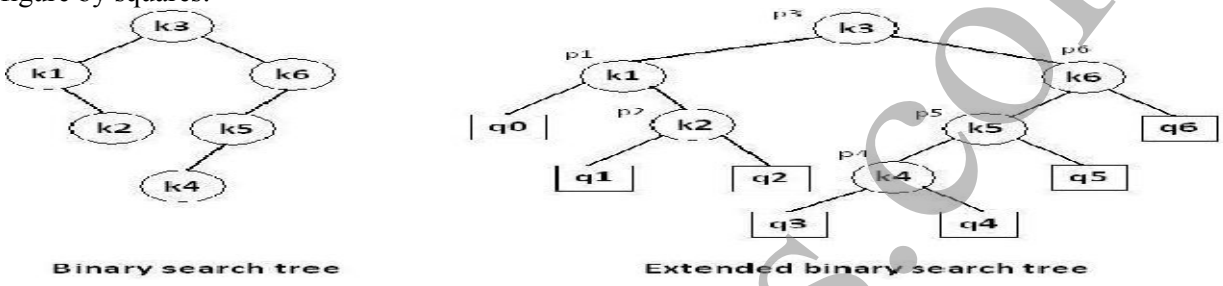
5. What are binomial heaps?

6. Explain the height-biased leftist trees.

UNIT – 8 : EFFICIENT BINARY SEARCH TREES

8.1. Optimal Binary Search Trees:

An optimal binary search tree is a binary search tree for which the nodes are arranged on levels such that the tree cost is minimum. For the purpose of a better presentation of optimal binary search trees, we will consider "extended binary search trees", which have the keys stored at their internal nodes. Suppose "n" keys k1, k2, … , k n are stored at the internal nodes of a binary search tree. It is assumed that the keys are given in sorted order, so that k1< k2 < … < kn. An extended binary search tree is obtained from the binary search tree by adding successor nodes to each of its terminal nodes as indicated in the following figure by squares:



In the extended tree:

- the squares represent terminal nodes. These terminal nodes represent unsuccessful searches of the tree for key values. The searches did not end successfully, that is, because they represent key values that are not actually stored in the tree;
- the round nodes represent internal nodes; these are the actual keys stored in the tree;
- assuming that the relative frequency with which each key value is accessed is known, weights can be assigned to each node of the extended tree (p1 … p6). They represent the relative frequencies of searches terminating at each node, that is, they mark the successful searches.

If the user searches a particular key in the tree, 2 cases can occur:

1 – The key is found, so the corresponding weight 'p' is incremented;
2 – The key is not found, so the corresponding 'q' value is incremented.

GENERALIZATION: the terminal node in the extended tree that is the left successor of k1 can be interpreted as representing all key values that are not stored and are less than k1. Similarly, the terminal node in the extended tree that is the right successor of kn, represents all key values not stored in the tree that are greater than kn. The terminal node that is successed between ki and ki-1 in an inorder traversal represents all key values not stored that lie between ki and ki - 1.

An obvious way to find an optimal binary search tree is to generate each possible binary search tree for the keys, calculate the weighted path length, and keep that tree with the smallest weighted path length. This search through all possible solutions is not feasible, since the number of such trees grows exponentially with "n".

An alternative would be a recursive algorithm. Consider the characteristics of any optimal tree. Of course it has a root and two subtrees. Both subtrees must themselves be optimal binary search trees with respect to their keys and weights. First, any subtree of any binary search tree must be a binary search tree. Second, the subtrees must also be optimal. Since there are "n" possible keys as candidates for the root of the optimal tree, the recursive solution must try them all. For each candidate key as root, all keys less than

that key must appear in its left subtree while all keys greater than it must appear in its right subtree. Stating the recursive algorithm based on these observations requires some notations:
OBST(i, j) denotes the optimal binary search tree containing the keys ki,
ki+1, …, kj;
Wi, j denotes the weight matrix for OBST(i, j)
Wi, j can be defined using the following formula:

$$W_{i,j} = \sum_{k=i+1}^{j} p_k + \sum_{k=i}^{j} q_k$$

Ci, j, $0 \leq i \leq j \leq n$ denotes the cost matrix for OBST(i, j)
Ci, j can be defined recursively, in the following manner:
Ci, i = Wi, j
Ci, j = Wi, j + mini<k≤j(Ci, k - 1 + Ck, j)
Ri, j, $0 \leq i \leq j \leq n$ denotes the root matrix for OBST(i, j)
Assigning the notation Ri, j to the value of k for which we obtain a minimum in the above relations, the optimal binary search tree is OBST(0, n) and each subtree OBST(i, j) has the root kRij and as subtrees the trees denoted by OBST(i, k-1) and OBST(k, j).
*OBST(i, j) will involve the weights qi-1, pi, qi, …, pj, qj.

All possible optimal subtrees are not required. Those that are consist of sequences of keys that are immediate successors of the smallest key in the subtree, successors in the sorted order for the keys. The bottom-up approach generates all the smallest required optimal subtrees first, then all next smallest, and so on until the final solution involving all the weights is found. Since the algorithm requires access to each subtree's weighted path length, these weighted path lengths must also be retained to avoid their recalculation. They will be stored in the weight matrix 'W'. Finally, the root of each subtree must also be stored for reference in the root matrix 'R'.

The following function builds an optimal binary search tree

```
FUNCTION CONSTRUCT(R, i, j)
begin
*build a new internal node N labeled (i, j)
k _ R (i, j)
if i = k then
*build a new leaf node N' labeled (i, i)
else
*N' _ CONSTRUCT(R, i, k)
*N' is the left child of node N
if k = (j – 1) then
*build a new leaf node N'' labeled (j, j)
else
*N'' _ CONSTRUCT(R, k + 1, j)
*N'' is the right child of node N
return N
end
```

8.2. AVL Trees

Definitions

Named after Adelson, Velskii, and Landis.

Trees of height O(log n) are said to be **balanced**. AVL trees consist of a special case in which the subtrees of each node differ by at most 1 in their height. Balanced trees can be used to search, insert, and delete arbitrary keys in O(log n) time. In contrast, height-biased leftist trees rely on non-balanced trees to speed-up insertions and deletions in priority queues.

Height

Claim: **AVL trees are balanced.**

Proof. Let N_h denote the number of nodes in an AVL tree of depth h

$$N_h \geq N_{h-1} + N_{h-2} + 1$$

$$\geq 2N_{h-2} + 1$$

$$\geq 1 + 2(1 + 2N_{h-4})$$

$$= 1 + 2 + 2^2 N_{h-4}$$

$$\geq 1 + 2 + 2^2 + 2^3 N_{h-6}$$

$$\ldots$$

$$\geq 1 + 2 + 2^2 + 2^3 + \ldots + 2^{h/2}$$

$$= 2^{h/2} - 1$$

Hence,

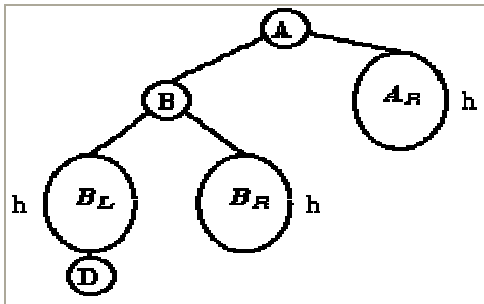
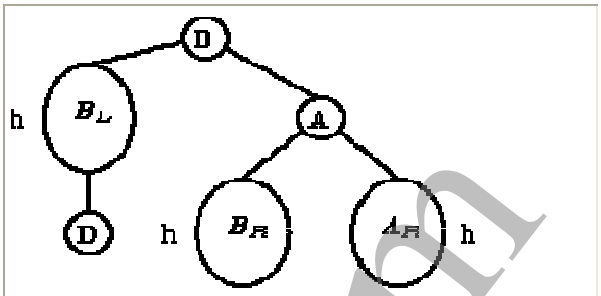
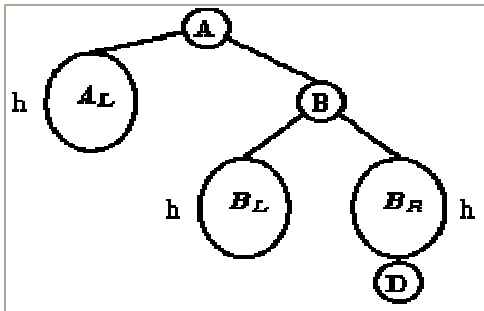
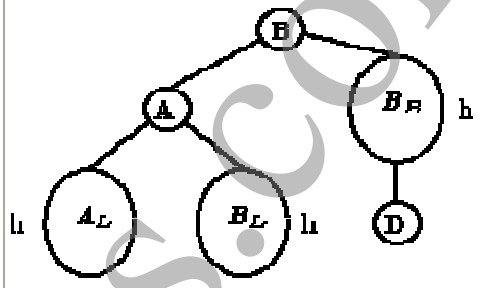
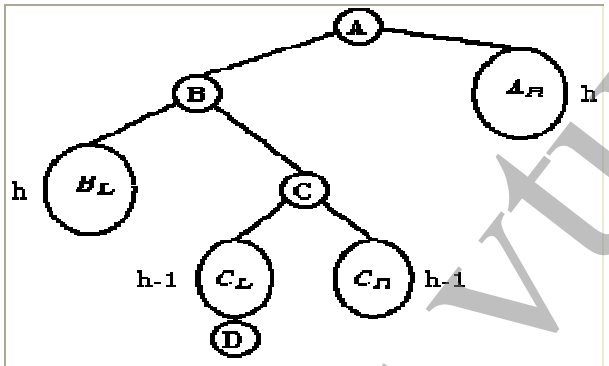
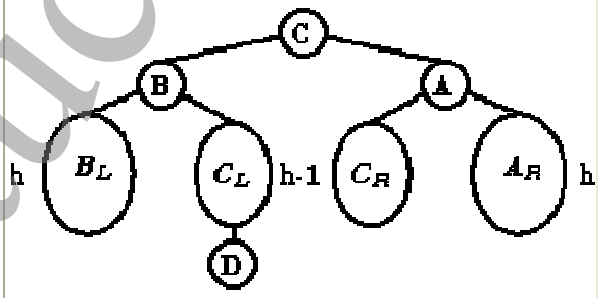
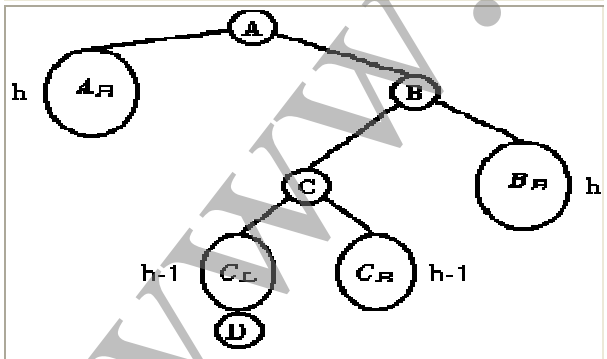
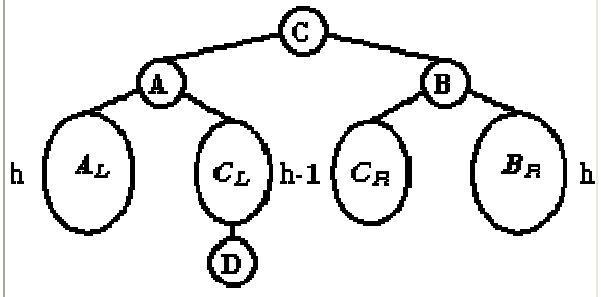
$$2^{h/2} - 1 \leq n$$

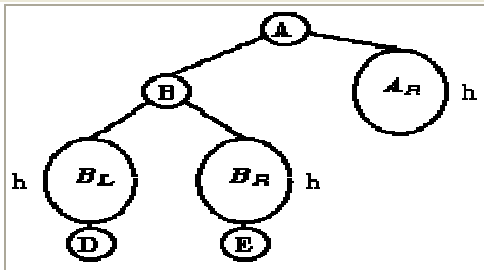
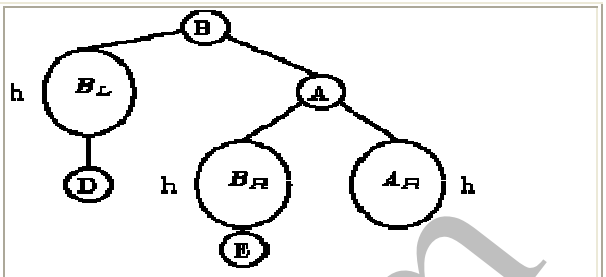
$$h/2 \leq \log_2(n + 1)$$

$$h \leq 2 \log_2(n + 1)$$

A more careful analysis, based on Fibonacci numbers theory, implies the tighter bound of $1.44 \log_2(n + 2)$.

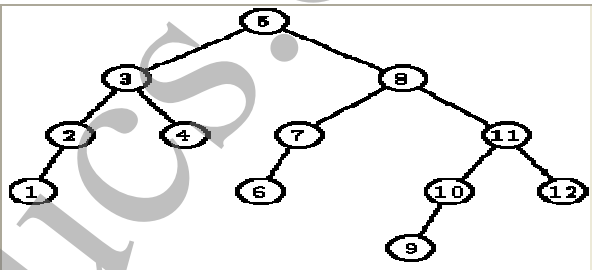
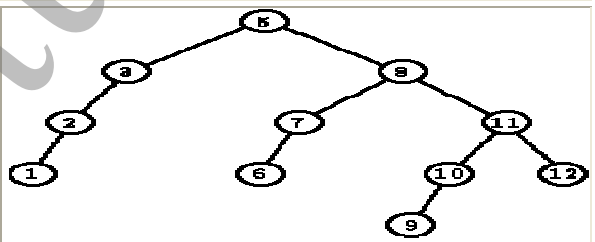
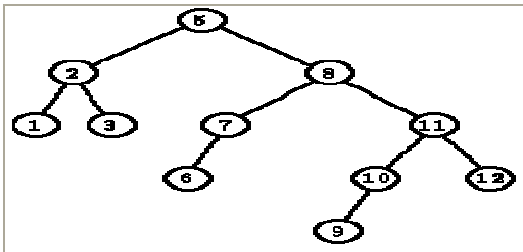
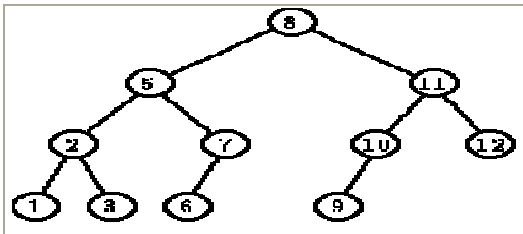
Rotations

LL		
RR		
LR		
RL		

LL & LR ⇒ LL		

Insertions and Deletions

Insertions and deletions are performed as in binary search trees, and followed by rotations to correct imbalances in the outcome trees. In the case of insertions, one rotation is sufficient. In the case of deletions, O(log n) rotations at most are needed from the first point of discrepancy going up toward the root.

	
Delete 4	
Imbalance at '3' implies a LL rotation with '2'	
Imbalance at '5' implies a RR rotation with '8'.	

8.3. Red-black Trees

Properties

A binary search tree in which

- The root is colored black
- All the paths from the root to the leaves agree on the number of black nodes
- No path from the root to a leaf may contain two consecutive nodes colored red

Empty subtrees of a node are treated as subtrees with roots of black color.

The relation $n \geq 2^{h/2} - 1$ implies the bound $h \leq 2 \log_2(n + 1)$.

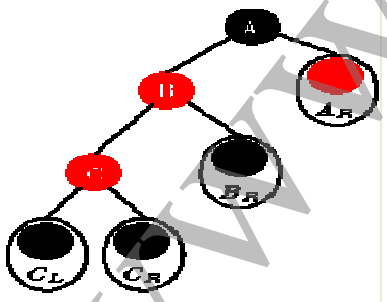
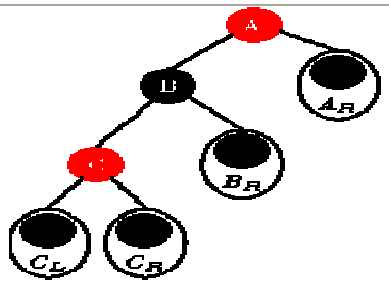
Insertions

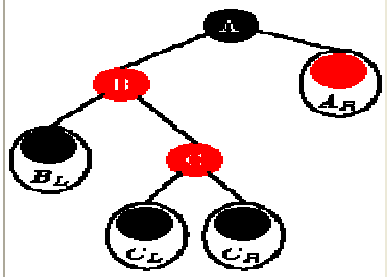
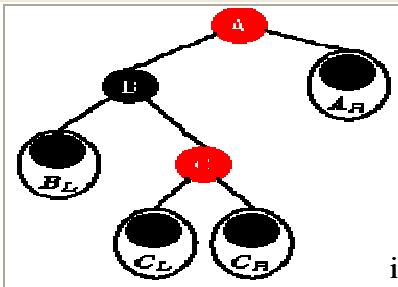
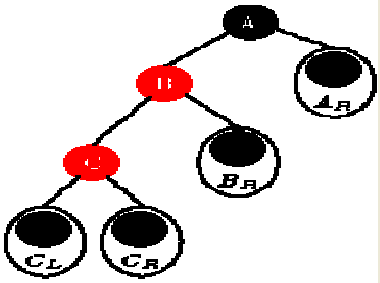
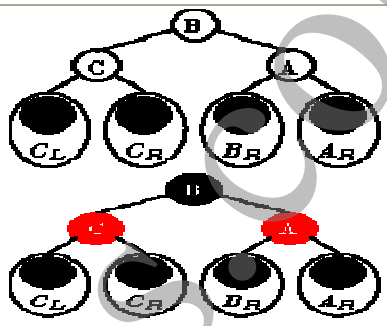
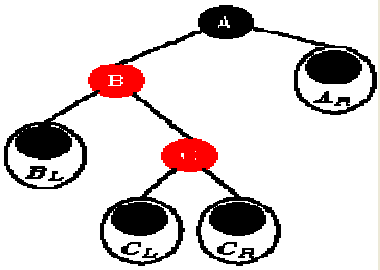
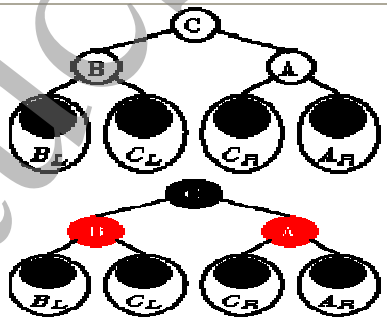
- Insert the new node the way it is done in binary search trees
- Color the node red
- If a discrepancy arises for the red-black tree, fix the tree according to the type of discrepancy.

A discrepancy can result from a parent and a child both having a red color. The type of discrepancy is determined by the location of the node with respect to its grand parent, and the color of the sibling of the parent.

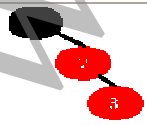
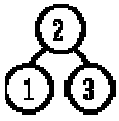
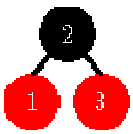
Discrepancies in which the sibling is red, are fixed by changes in color. Discrepancies in which the siblings are black, are fixed through AVL-like rotations.

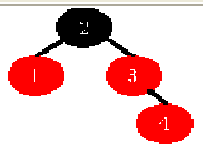
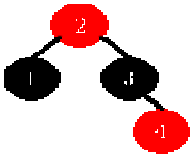
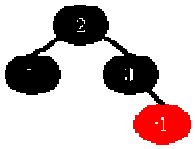
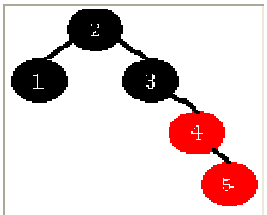
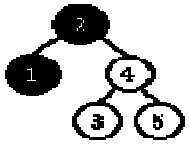
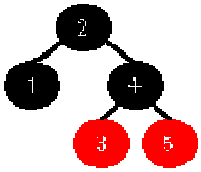
Changes in color may propagate the problem up toward the root. On the other hand, at most one rotation is sufficient for fixing a discrepancy.

LLr		 if 'A' is the root, then it should be repainted to black

LRr		 if 'A' is the root, then it should be repainted to black
LLb		
LRb		

Discrepancies of type RRr, RLr, RRb, and RLb are handled in a similar manner.

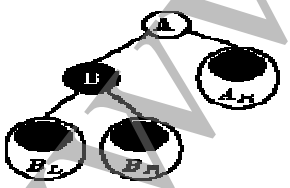
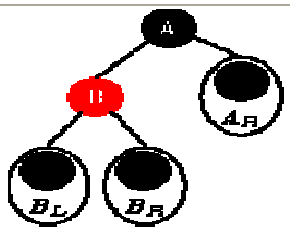
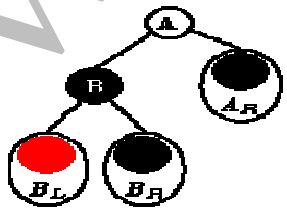
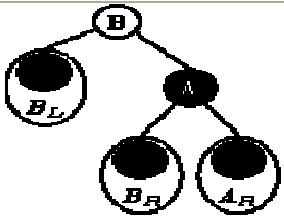
insert 1			
insert 2			
insert 3	 RRb discrepancy		

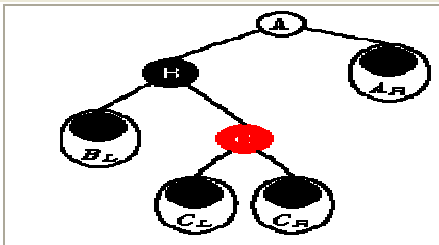
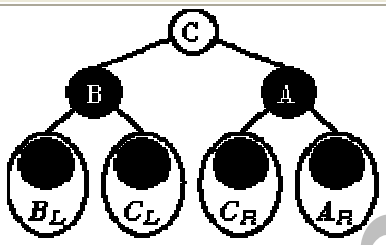
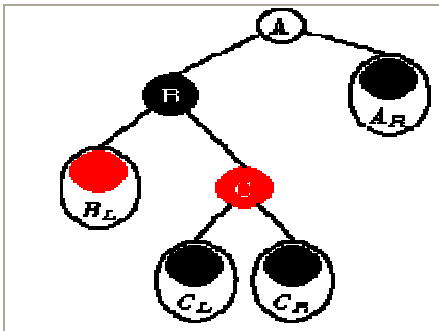
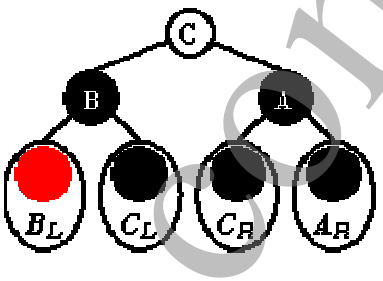
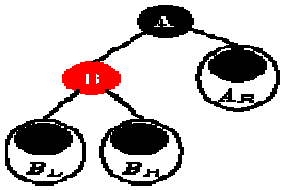
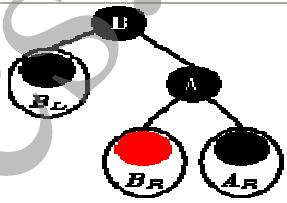
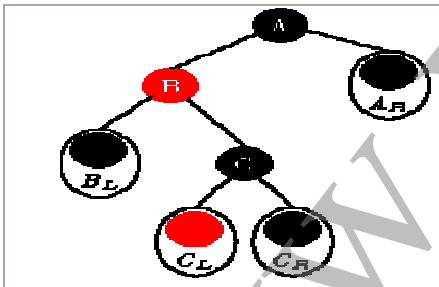
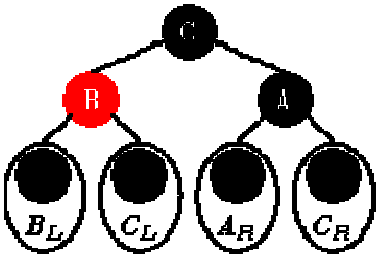
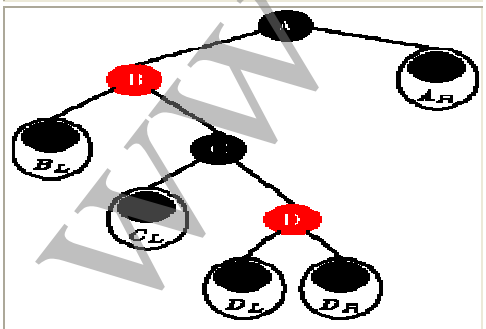
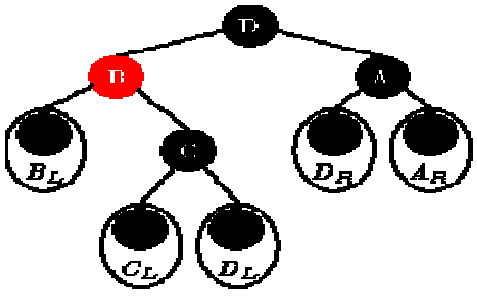
insert 4	 RRr discrepancy		
insert 5	 RRb discrepancy		

Deletions

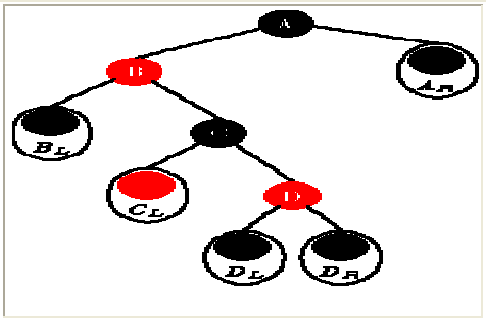
- Delete a key, and a node, the way it is done in binary search trees.
- A node to be deleted will have at most one child. If the deleted node is red, the tree is still a red-black tree. If the deleted node has a red child, repaint the child to black.
- If a discrepancy arises for the red-black tree, fix the tree according to the type of discrepancy. A discrepancy can result only from a loss of a black node.

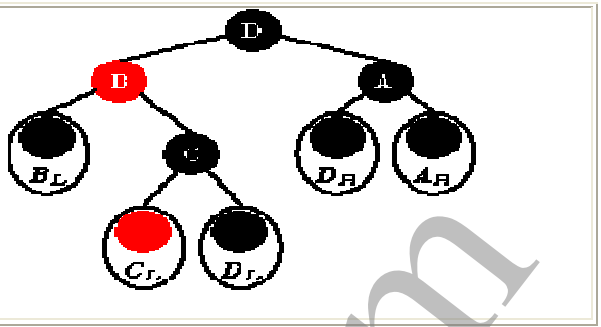
Let A denote the lowest node with unbalanced subtrees. The type of discrepancy is determined by the location of the deleted node (**R**ight or **L**eft), the color of the sibling (**b**lack or **r**ed), the number of red children in the case of the black siblings, and and the number of grand-children in the case of red siblings. In the case of discrepancies which result from the addition of nodes, the correction mechanism may propagate the color problem (i.e., parent and child painted red) up toward the root, and stopped on the way by a single rotation. Here, in the case of discrepancies which result from the deletion of nodes, the discrepancy of a missing black node may propagate toward the root, and stopped on the way by an application of an appropriate rotation.

Rb0		 change of color, sending the deficiency up to the root of the subtree
Rb1		

		
Rb2		
Rr0		 might result in LLb discrepancy of parent and child having both the red color
Rr1		
		

Rr2





Similar transformations apply to Lb0, Lb1, Lb2, Lr0, Lr1, and Lr2.

8.4. Splay Trees :

Splay Trees (self-adjusting search trees):

These notes just describe the bottom-up splaying algorithm, the proof of the access lemma, and a few applications. Every time a node is accessed in a splay tree, it is moved to the root of the tree. The amortized cost of the operation is O(log n). Just moving the element to the root by rotating it up the tree does not have this property. Splay trees do movement is done in a very special way that guarantees this amortized bound. I'll describe the algorithm by giving three rewrite rules in the form of pictures. In these pictures, x is the node that was accessed (that will eventually be at the root of the tree). By looking at the local structure of the tree defined by x, x's parent, and x's grandparent we decide which of the following three rules to follow. We continue to apply the rules until x is at the root of the tree:

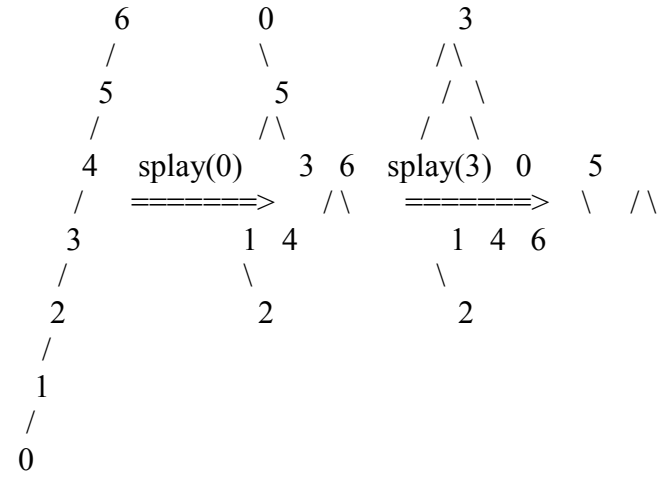
```
                    y          x
Zig (terminal case):    /    ====>    \
                      x              y

                z          z
               /          /          x
Zig-zag:      y    ====>  x   ====>   / \
               \          /          y   z
                x        y

                z                    x
               /          y          \
Zig-zig:      y    ====>  / \  ====>    y
             /            x   z        \
            x                          z
```

Notes (1) Each rule has a mirror image variant, which covers all the cases.

(2) The zig-zig rule is the one that distinguishes splaying from just rotating x to the root of the tree.

(3) Top-down splaying is much more efficient in practice. Code for doing this is on my web site (www.cs.cmu.edu/~sleator).

Here are some examples:



To analyze the performance of splaying, we start by assuming that each node x has a weight w(x) > 0. These weights can be chosen arbitrarily. For each assignment of weights we will be able to derive a bound on the cost of a sequence of accesses. We can choose the assignment that gives the best bound. By giving the frequently accessed elements a high weight, we will be able to get tighter bounds on the running time. Note that the weights are only used in the analysis, and do not change the algorithm at all.

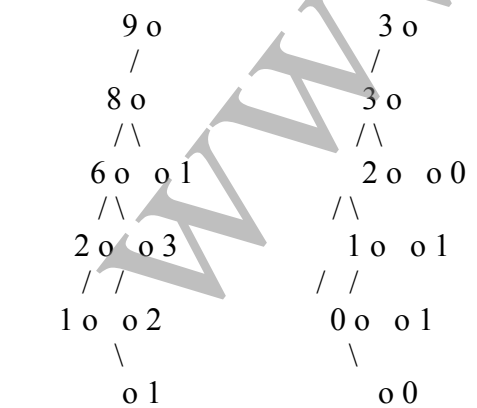
(A commonly used case is to assign all the weights to be 1.)

Before we can state our performance lemma, we need to define two more quantities. The size of a node x (denoted s(x)) is the total weight of all the nodes in the subtree rooted at x. The rank of a node x (denoted r(x)) is the floor(log_2) of the size of x. Restating these:

s(x) = Sum (over y in the subtree rooted at x) of w(y)

r(x) = floor(log(s(x)))

For each node x, we'll keep r(x) tokens on that node. (Alternatively, the potential function will just be the sums of the ranks of all the nodes in the tree.)
Here's an example to illustrate this: Here's a tree, labeled with sizes on the left and ranks on the right.



Notes about this potential:

(1) Doing a rotation between a pair of nodes x and y only effects the ranks of the nodes x and y, and no other nodes in the tree. Furthermore, if y was the root before the rotation, then the rank of y before the rotation equals the rank of x after the rotation.

(2) Assuming all the weights are 1, the potential of a balanced tree is O(n), and the potential of a long chain (most unbalanced tree) is O(n log n).

(3) In the banker's view of amortized analysis, we can think of having r(x) tokens on node x.

Access lemma: The number of splaying steps done when splaying node x in a tree with root t is at most 3(r(t)-r(x))+1.

Proof:

As we do the work, we must pay one token for each splay step we do. Furthermore we must make sure that there are always r(x) tokens on node x as we do our restructuring. We are going to allocate 3(r(t) - r(x)) +1 tokens to do the splay. Our job in the proof is to show that this is enough.

First we need the following observation about ranks, called the Rank Rule. Suppose that two siblings have the same rank, r. Then the parent has rank at least r+1. This is because if the rank is r, then the size is at least 2^r. If both siblings have size at least 2^r, then the total size is at least 2^(r+1) and we conclude that the rank is at least r+1. We can represent this with the following diagram:

```
  x
 / \      Then x >= r+1
r   r
```

Conversly, suppose we find a situation where a node and its parent have the same rank, r. Then the other sibling of the node must have rank < r.So if we have three nodes configured as follows, with these ranks:

```
  r
 / \      Then x < r
x   r
```

Now we can go back to proving the lemma. The approach we take is to show that the 3(r'(x) - r(x)) tokens are sufficient to pay for the a zig-zag or a zig-zig steps. And that 3(r'(x) - r(x)) +1 is sufficient to pay for the zig step. (Here r'() represents the rank function after the step, and r() represents the rank function before the step.)

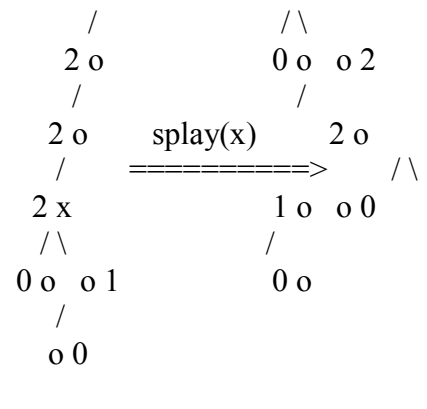
When we sum these costs to compute the amortized cost for the entire splay operation, they telescope to:

3(r(t) - r(x)) +1.

Note that the +1 comes from the zig step, which can happen only once.

Here's an example, the labels are ranks:

```
2 o              2 x
```



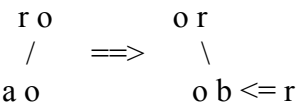
Total: 9 7

We allocated: 3(2-2)+1 = 1
extra tokens from restructuring: 9-7 = 2

3
There are 2 splay steps. So 3 > 2, and we have enough.

It remains to show these bounds for the individual steps. There are three cases, one for each of the types of splay steps.

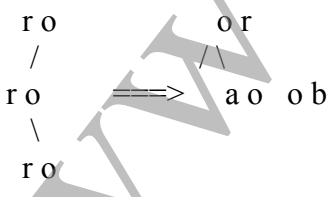
Zig:



The actual cost is 1, so we spend one token on this. We take the tokens on a and augment them with another r-a and put them on b. Thus the total number of tokens needed is 1+r-a. This is at most 1+3(r-a).

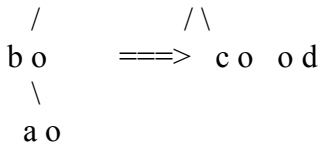
Zig-zag: We'll split it into 2 cases:

Case 1: The rank does not increase between the starting node and ending node of the step.



By the Rank Rule, one of a or b must be < r, so one token is released from the data structure. we use this to pay for the work.
Thus, our allocation of 3(r-r) = 0 is sufficient to pay for this.

Case2: (The rank does increase)



The tokens on c can be supplied from those on b. (There are enough there cause b >= c.)

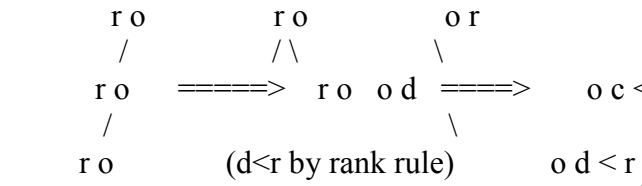
Note that r-a > 0. So:

use r-a (which is at least 1) to pay for the work
use r-a to augment the tokens on a to make enough for d

Summing these two gives: 2(r-a). But 2(r-a) <= 3(r-a), which completes the zig-zag case.

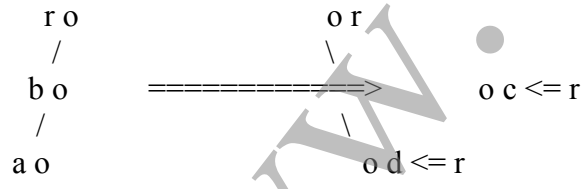
Zig-zig: Again, we split into two cases.

Case 1: The rank does not increase between the starting node and the ending node of the step.



As in the zig-zag case, we use the token gained because d < r to pay for the step. Thus we have 3(r-r)=0 tokens and we need 0.

Case 2: The rank increases during the step.



use r-a (which is at least 1) to pay for the work
use r-a to boost the tokens on a to cover those needed for d
use r-a to boost the tokens on b to cover those needed for c

Summing these gives 3(r-a), which completes the analyzis of the zig-zig case.

This completes the proof of the access lemma.
Balance Theorem: A sequence of m splays in a tree of n nodes takes time O(m log(n) + n log(n)).

Proof: We apply the access lemma with all the weights equal to 1. For a given splay, r(t) <= log(n), and r(x) >= 0. So the amortized cost of the splay is at most:

$3 \log(n) + 1$

We now switching to the world of potentials (potential = total tokens in the tree). To bound the cost of the sequence we add this amount for each splay, then add the initial minus the final potential. The initial potential is at most n log(n), and the final potential is at least 0. This gives a bound of:

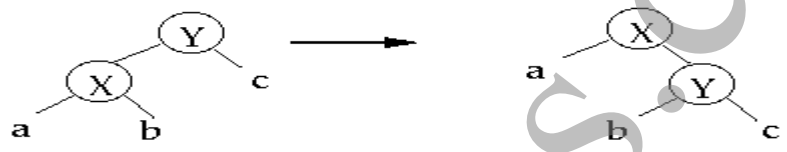
$$m (3 \log(n) + 1) + n \log(n) = O(m \log(n) + n \log(n))$$

Splaying

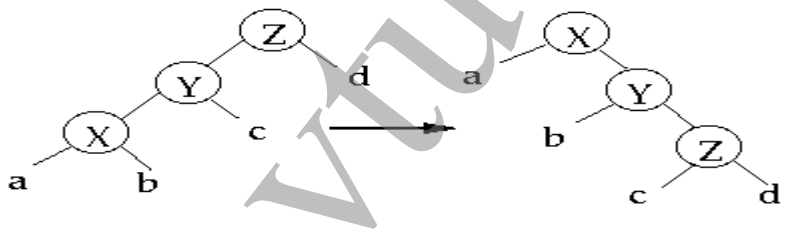
Splay step at x

let p(x) = parent of node x

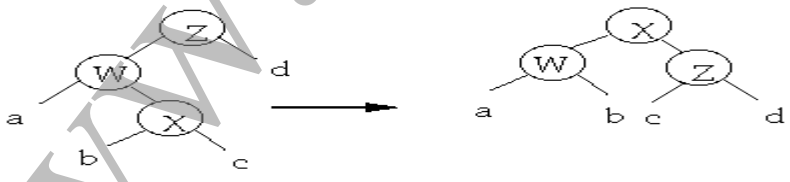
case 1 (zig) p(x) = root of the tree



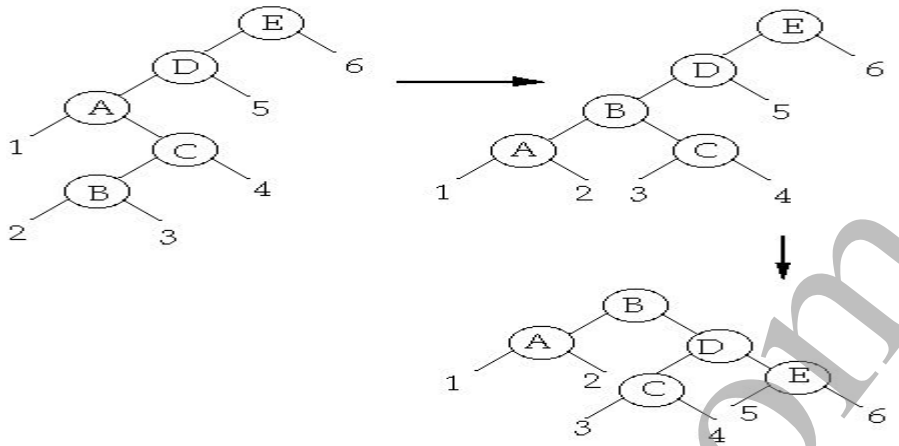
case 2 (zig-zig) p(x) is not the root and x and p(x) are both left (right) children



case 3 (zig-zag) p(x) is not the root and x is a left (right) child and p(x) is a right(left) child



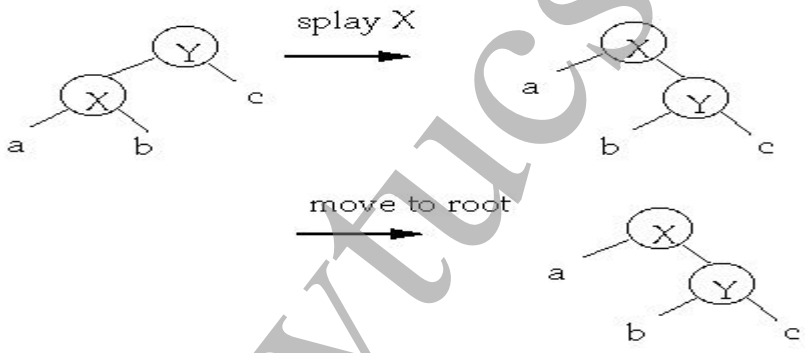
To Splay a node X, repeat the splay step on X until it is the root



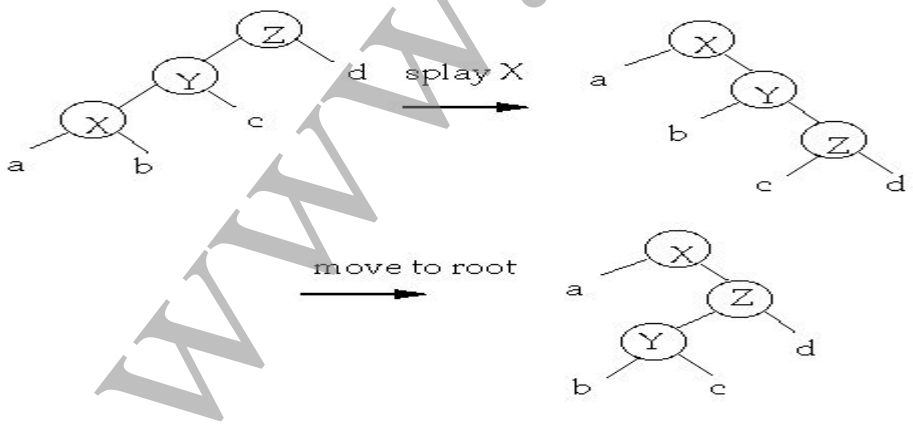
Splay B

Splay vs. Move-to-root

Case 1

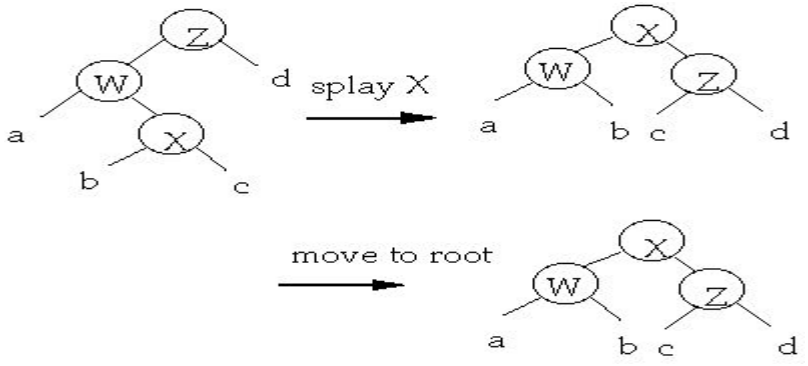


Case 2

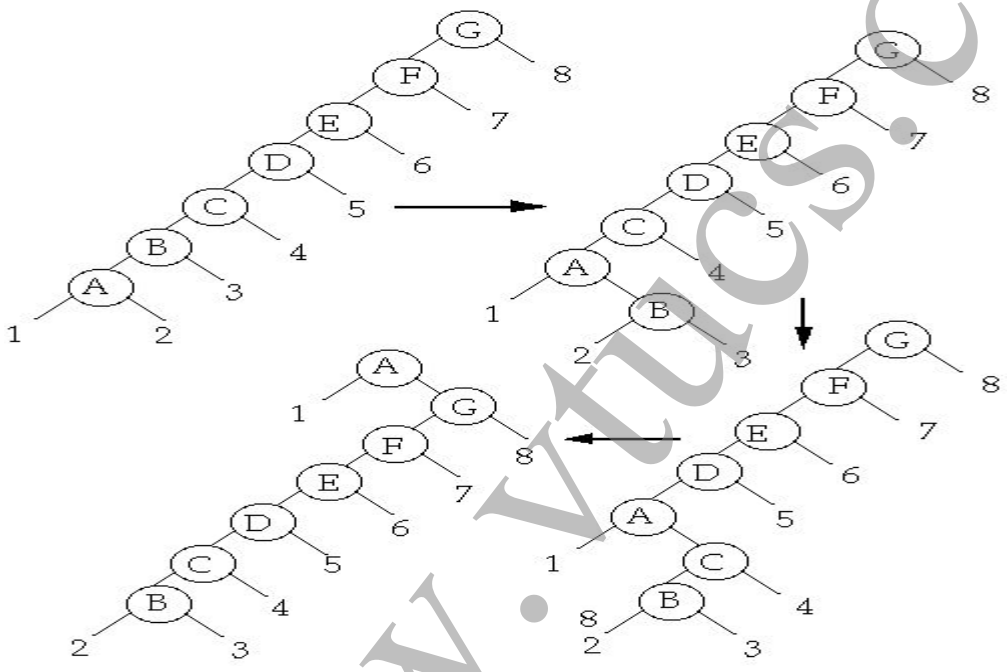


Splay vs. Move-to-root

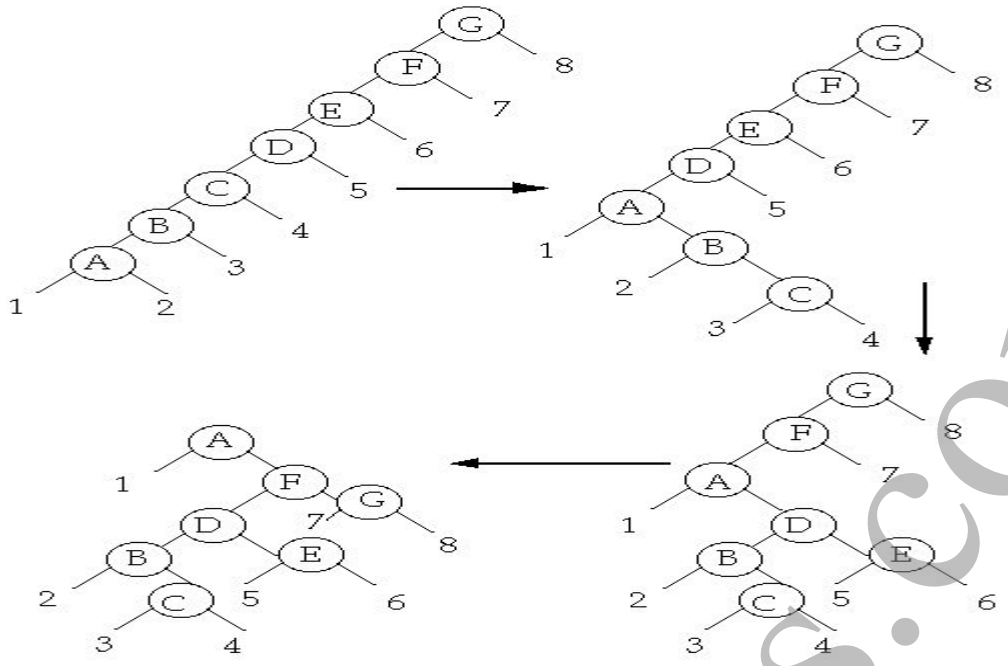
Case 3



Move-to-root A



Splay A



Performance of Splay Tree

Splaying at a node of depth d takes Theta(d) time

ck= actual cost of operation k

$\hat{c}_k$= amortized cost of operation k

Dk = the state of the data structure after applying k'th operation to Dk

$\Phi(D_k)$= potential associated with Dk

$$\hat{c}_k = c_k + \Phi(D_k) - \Phi(D_{k-1})$$

so we get:

$$c_k = \hat{c}_k - \Phi(D_k) + \Phi(D_{k-1})$$

The actual amount of work required is given by:

$$\sum_{k=1}^{m} c_k = \sum_{k=1}^{m} \hat{c}_k - \Phi(D_k) + \Phi(D_{k-1})$$

$$= \Phi(D_0) - \Phi(D_m) + \sum_{k=1}^{m} \hat{c}_k$$

So need the total amortized work and difference in potential

Potential for Splay Trees

Let:

w(x) = weight of node x, a fixed but arbitrary value

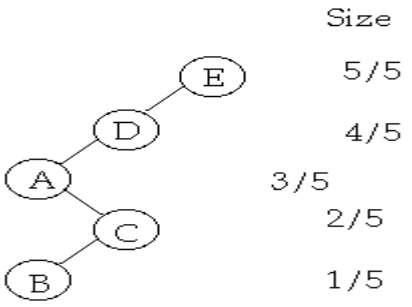
$$\text{size}(x) = \sum_{y \in T_x} w(y)$$

rank(x) = lg(size(x))

$$\Phi(T) = \sum_{x \in T} \text{rank}(x)$$

Example

Let w(x) = 1/n where n is the number of nodes in the tree



$$\Phi(T) = \lg(1/5) + \lg(2/5) + \lg(3/5) + \lg(4/5) + \lg(5/5)$$
$$= -4.70275$$

Lemma The amortized time to splay node x in a tree with root at t is at most 3(r(t) - r(x)) + 1 = O(lg(s(t)/s(x)))

Let s, r denote the size, rank functions before a splay

Let s', r' denote the size, rank functions after a splay

Count rotations

Case 1 (zig) One rotation

Amortized time of this step is:

$\hat{c}_k = c_k + \Phi(D_k) - \Phi(D_{k-1})$

1 + [r'(x) + r'(y)] - r(x) - r(y) only x and y change rank

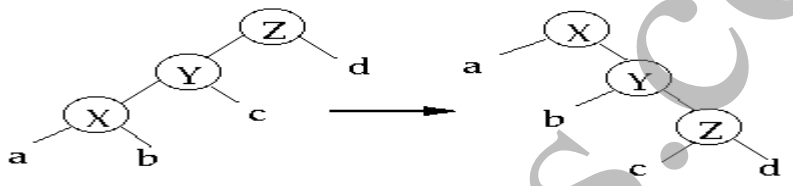
<= 1 + r'(x) - r(x)

r(y) >= r'(y)

<= 1 + 3(r'(x) - r(x))

r'(x) >= r(x)

Case 2 (zig-zig) Two rotations



Amortized time of this step is:

2 + r'(x) + r'(y) + r'(z)
- r(x) - r(y) - r(z) only x, y ,z change rank

= 2 + r'(y) + r'(z) - r(x) - r(y)

r'(x) = r(z)

<= 2 + r'(x) + r'(z) - 2r(x)

r'(x) >= r'(y) and

r(y) >= r(x)

Assume that 2r'(x) - r(x) - r'(z) >= 2

2 + r'(x) + r'(z) - 2r(x)

<= 2r'(x) - r(x) - r'(z) + r'(x) + r'(z) - 2r(x)

= 3r'(x) - 3r(x)

Need to show 2r'(x) - r(x) - r'(z) >= 2

Claim 1 $\max_{a+b\le 1,\ a,b>0} (\lg(a)+\lg(b)) = -2$

Set b = 1-a

We have $$\frac{\partial(\lg(a)+\lg(1-a))}{\partial a} = \left(\frac{c}{a}+\frac{c}{1-a}(-1)\right)\lg(e)$$

Setting this to 0 to find extreme value we get

$$0 = \left(\frac{c}{a}+\frac{c}{1-a}(-1)\right)\lg(e)$$

$$\frac{c}{1-a} = \frac{c}{a}$$

so

$1 - a = a$

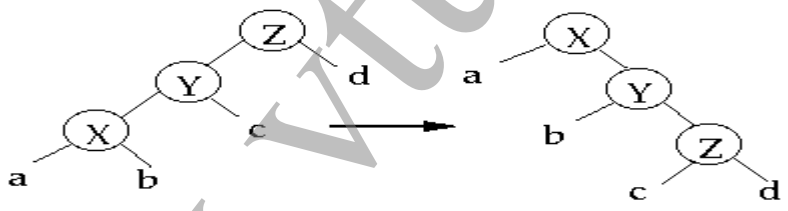
that is a = 1/2 and b = 1/2

but lg(1/2)+lg(1/2)= -2

End claim 1

Claim 2 2r'(x) - r(x) - r'(z) >= 2

Recall that:



We have:

r(x) + r'(z) - 2r'(x) = lg(s(x)) + lg(s'(z)) - 2lg(s'(x))

= lg(s(x)/s'(x)) + lg(s'(z)/s'(x))

Now s(x) + s'(z) <= s'(x)

(Why?)

so

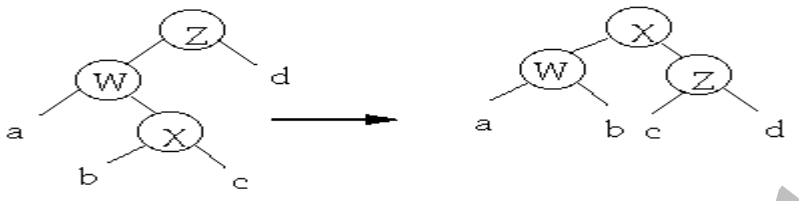
0<= s(x)/s'(x) + s'(z)/s'(x) <= 1

Set s(x)/s'(x) = a and s'(z)/s'(x) =b in claim 1 to get

lg(s(x)/s'(x)) + lg(s'(z)/s'(x)) <= -2

Thus r(x) + r'(z) - 2r'(x) <= -2 or 2r'(x) - r(x) - r'(z) >= 2

Case 3 (zig-zag)



8.5. RECOMMENDED QUESTIONS

1. Define AVL Tree.
2. What are the categories of AVL rotations?
3. What do you mean by balance factor of a node in AVL tree
4. Define splay tree.
5. What is the idea behind splaying?
6. List the types of rotations available in Splay tree.
7. What is the minimum number of nodes in an AVL tree of height h?
8. What is optimal binary search tree.
9. Explain the representation of a Red-black tree.