

What is data?

Facts and statistics collected together for reference or analysis is called data. It refers to the quantities, characters, or symbols on which operations are performed by a computer, which may be stored and transmitted in the form of electrical signals and recorded on magnetic, optical, or mechanical recording media.

Data can exist in a variety of forms; as numbers or text on pieces of paper, as bits and bytes stored in electronic memory, or as facts stored in a person's mind. Strictly speaking, data is the plural of *datum*, a single piece of information. In practice, however, people use *data* as both the singular and plural form of the word.

What is data structure?

Data structure refers to the Interrelationship among data elements that determine how data is recorded, manipulated, stored, and presented by a database.

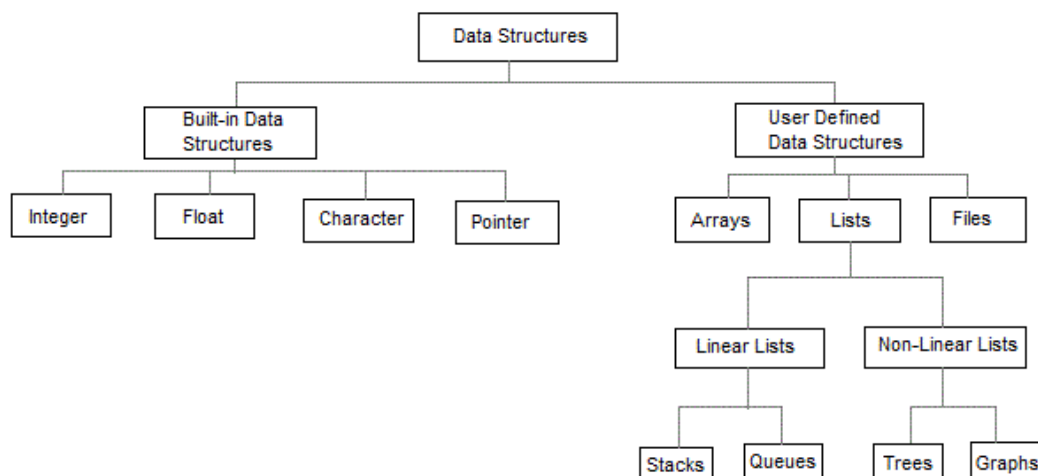
It is a way of collecting and organizing data in such a way that we can perform operations on these data in an effective way. Data Structures is about rendering data elements in terms of some relationship, for better organization and storage. In simple language, Data Structures are structures programmed to store ordered data, so that various operations can be performed on it easily.

Anything that can store data can be called as a data structure hence Integer, Float, Boolean, Char etc., all are data structures. They are known as **Primitive Data Structures**.

Then we also have some complex Data Structures, which are used to store large and connected data. Some examples of **Abstract Data Structure** are:

- Linked List
- Tree
- Graph
- Stack, Queue etc.

All these data structures allow us to perform different operations on data. We select these data structures based on which type of operation is required. We will look into these data structures in more details in our later lessons.



INTRODUCTION TO DATA STRUCTURES

Why data structure is such an important subject?

Sometimes it is difficult for students to view where they can apply knowledge that they get during studies. When we (software engineers, developers and etc.) were in bachelor degree we learnt a lot of different kind of data structures. In this hand book I would like to give some reasons, why it is important to learn complex data structures.

Data structure is a particular way of storing and organizing information in a computer. So that it can be retrieved and used most productively. Different kinds of data structures are meant for different kinds of applications, and some are highly specialized to specific tasks.

Data structures are important for the following reasons:

1. Data structures are used in almost every program or software system.
2. Specific data structures are essential ingredients of many efficient algorithms, and make possible the management of huge amounts of data, such as large integrated collection of databases.
3. Some programming languages emphasize data structures, rather than algorithms, as the key organizing factor in software design.
4. Use of appropriate data structure enables a computer system to perform its task more efficiently, by influencing the ability of computer to store and retrieve data from any location in its memory.
5. Almost every company that is hiring a software engineer would ask about data structures.
6. Choosing the right data structure for the algorithm is one of the key components in the world of software engineering.

What is Abstract Data Type (ADT)?

An **abstract data type**, sometimes abbreviated **ADT**, is a logical description of how we view the data and the operations that are allowed without regard to how they will be implemented. This means that we are concerned only with what the data is representing and not with how it will eventually be constructed. By providing this level of abstraction, we are creating an **encapsulation** around the data. The idea is that by encapsulating the details of the implementation, we are hiding them from the user's view. This is called **information hiding**.

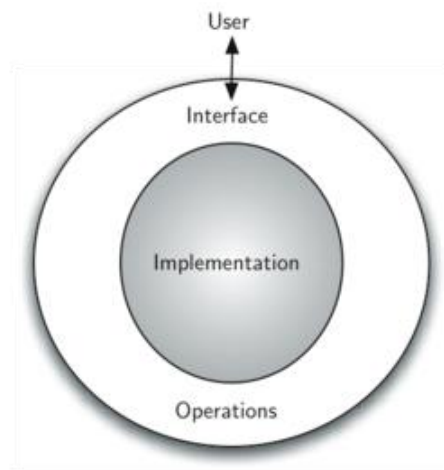
Abstract data types are mathematical models of a set of data values or information that share similar behavior or qualities and that can be specified and identified independent of specific implementations. Abstract data types, or ADTs, are typically used in algorithms. An abstract data type is defined in term of its data items or its associated operations rather than by its implementation.

A stack or a queue is an example of an ADT. It is important to understand that both stacks and queues can be implemented using an array. It is also possible to implement stacks and queues using a linked list. This demonstrates the "abstract" nature of stacks and queues: how they can be considered separately from their implementation.

Figure shows a picture of what an abstract data type is and how it operates. The user interacts with the interface, using the operations that have been specified by the abstract data type. The abstract data type is the shell that the user interacts with. The implementation is hidden one level deeper. The user is not concerned with the details of the implementation.

There are two parts to each ADT:

1. The public or external part, which consists of:
 - the conceptual picture (the user's view of what the object looks like, how the structure is organized)
 - the conceptual operations (what the user can do to the ADT)
2. The private or internal part, which consists of:
 - the representation (how the structure is actually stored)
 - the implementation of the operations (the actual code)



To best describe the term Abstract Data Type, it is best to break the term down into "data type" and then "abstract".

Data type particular kind of data item, as defined by the values it can take, the programming language used, or the operations that can be performed on it. The word **Abstract** in our context stands for *"considered apart from the detailed specifications or implementation"*.

The benefits of using ADTs include:

- Code is easier to understand (e.g., it is easier to see "high-level" steps being performed, not obscured by low-level code).
- Implementations of ADTs can be changed (e.g., for efficiency) without requiring changes to the program that uses the ADTs.
- ADTs can be reused in future programs.

Introduction of Recursion

Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function. *"A function that calls itself directly (or indirectly) to solve a smaller version of its task until a final call which does not require a self-call is a recursive function."*

```
void recursion() {
    recursion(); /* function calls itself */
}
int main() {
    recursion();
}
```

Technically, a *recursive function* is a function that makes a call to itself. In any recursive process it is important to have a terminal condition (which we call the base condition) to assure that the recursive process will end at some point. The important thing to understand about recursion is that, it is the same algorithm we keep applying until we find a base case or termination condition. Recursion is a way to solve problems using a strategy call divide and conquer.

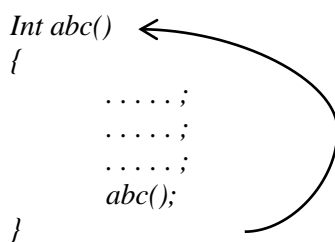
Many problems can be solved recursively, *For example*, games of all types from simple ones like the Towers of Hanoi problem to complex ones like chess.

Recursive function can be categorized as:

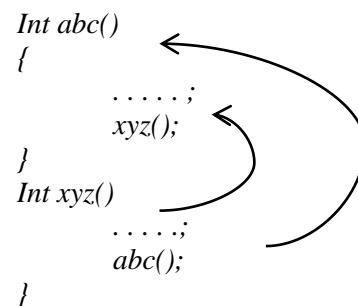
1. **Linear Recursion:** This recursion is the most commonly used. In this recursion a function call itself in a simple manner and by termination condition it terminates. This process called 'Winding' and when it returns to caller that is called 'Un-Winding'. Termination condition also known as Base condition.
2. **Binary Recursion:** Binary Recursion is a process where function is called twice at a time in place of once at a time. Mostly it's using in data structure like operations for tree as traversal, finding height, merging, etc.
3. **Tail Recursion:** In this method, recursive function is called at the last. So it's more efficient than linear recursion method. Means you can say termination point will come (100%) only you have to put that condition.

Recursion is of two types depending on whether a function calls itself from within itself, or whether two functions call one another mutually. The former is called direct recursion and the latter is called indirect recursion. Thus the two types of recursion are:

1. **Direct recursion**
2. **Indirect recursion**



(b) Direct Recursion



(a) Indirect Recursion

Advantages

- Reduce unnecessary calling of function.
- Through Recursion one can solve problems in easy way while its iterative solution is very big and complex.

Disadvantages

- Recursive solution is always logical and it is very difficult to trace.(debug and understand).
- In recursive we must have an If statement somewhere to force the function to return without the recursive call being executed, otherwise the function will never return.
- Recursion takes a lot of stack space, usually not considerable when the program is small and running on a PC.
- Recursion uses more processor time.
-

Iteration, in the context of computer programming, is a process wherein a set of instructions or structures are repeated in a sequence a specified number of times or until a condition is met. When the first set of instructions is executed again, it is called iteration. When a sequence of instructions is executed in a repeated manner, it is called a loop.

Iteration is the repetition of a process in a computer program, usually done with the help of loops. With respect to computing, **iteration** is the process of going through a set of operations that deal with computer code. For example, in a computer program, one form of iteration is a loop. A loop repeats code until a certain condition is met. Each time the computer runs through a loop, it is referred to as iteration.

```
int i = 0; //variable initialization
while(i <= 500)
{
    // loop condition
    printf("C programming is fun");
    i++; //increment
}
```

Difference between Recursion and Iteration

RECURSION	ITERATIONS
Recursive function – is a function that is partially defined by itself	Iterative Instructions –are loop based repetitions of a process
Recursion Uses selection structure	Iteration uses repetition structure
Infinite recursion occurs if the recursion step does not reduce the problem in a manner that converges on some condition.(base case)	An infinite loop occurs with iteration if the loop-condition test never becomes false
Recursion terminates when a base case is recognized	Iteration terminates when the loop-condition fails
Recursion is usually slower than iteration due to overhead of maintaining stack	Iteration does not use stack so it's faster than recursion
Recursion uses more memory than iteration	Iteration consume less memory
Infinite recursion can crash the system	infinite looping uses CPU cycles repeatedly
Recursion makes code smaller	Iteration makes code longer

Example 1: A Recursive Program to find the factorial of a given number.

```
#include<stdio.h>
#include<conio.h>
int factorial(int n);
main() {
    int n,result; clrscr();
    printf("Enter n value:\n");
    scanf("%d",&n);
    result=factorial(n);
    printf("The factorial of given number %d
is:%d\n",n,result);

    getch();
}
int factorial(int n)
{
    if(n<=1)
        return 1;
    else
        return n*factorial(n-1);
}
```

Example 2: A Recursive Program to find the Fibonacci position of a given number.

<pre>#include<stdio.h> #include<conio.h> int fib(int n); main() { int n,result; clrscr(); printf("Enter the nth number in Fibonacci series:\n"); scanf("%d",&n); result=fib(n); printf("The series in Fibonacci number %d is: %d\n",n,result);</pre>	<pre> getch(); } int fib(int n) { if(n==0) return 0; else if(n==1) return 1; else return fib(n-1)+fib(n-2); }</pre>
--	--

Example 3: A Recursive Program to find the GCD of two numbers.

<pre>#include<stdio.h> #include<conio.h> int gcd(int a,int b); main() { int a,b,result; clrscr(); printf("Enter a,b values:\n"); scanf("%d%d",&a,&b); result=gcd(a,b); printf("The GCD of a and b %d %d is:%d\n",a,b,result); getch();</pre>	<pre> } int gcd(int a,int b) { while(a!=b) { if(a>b) return gcd(a-b,b); else return gcd(a,b-a); } return a; }</pre>
--	--

Example 4: A Recursive Program for Tower of Hanoi (TOH) game.

<pre>#include<stdio.h> #include<conio.h> void hanoi(int n,char A,char B,char C); main() { int n; char A='A',B='B',C='C'; clrscr(); printf("Enter number of disks:\n"); scanf("%d",&n); printf("\n Towers of Hanoi problem with %d disks \n"); printf("\n *****\n"); printf("\t Sequence is:\n");</pre>	<pre> hanoi(n,A,B,C); getch(); } void hanoi(int n,char A,char B,char C) { if(n!=0) { hanoi(n-1,A,C,B); printf("Move disk %d from %c -> %c \n",n,A,C); hanoi(n-1,B,A,C); } }</pre>
--	---

Algorithm

A sequential solution of any program that written in human language, called algorithm. Algorithm is a step by step procedure, which defines a set of instructions to be executed in certain order to get the desired output. In programming, algorithms are the set of well-defined instruction in sequence to solve a program. An algorithm should always have a clear stopping point.

Qualities of a good algorithm

1. Inputs and outputs should be defined precisely.
2. Each step in algorithm should be clear and unambiguous.
3. Algorithm should be most effective among many different ways to solve a problem.
4. An algorithm shouldn't have computer code. Instead, the algorithm should be written in such a way that, it can be used in similar programming languages

Example: an algorithm to add two numbers entered by user.

1. *Step 1: Start*
2. *Step 2: Declare variables num1, num2 and sum.*
3. *Step 3: Read values num1 and num2.*
4. *Step 4: Add num1 and num2 and assign the result to sum.*
5. *sum ← num1 + num2*
6. *Step 5: Display sum*
7. *Step 6: Stop*

Algorithm Analysis

Efficiency of an algorithm can be analyzed at two different stages, before implementation and after implementation, as mentioned below –

1. **A priori analysis** – this is theoretical analysis of an algorithm. Efficiency of algorithm is measured by assuming that all other factors e.g. processor speed, are constant and have no effect on implementation.
2. **A posterior analysis** – this is empirical analysis of an algorithm. The selected algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space required are collected.

We shall learn here **a priori** algorithm analysis. Algorithm analysis deals with the execution or running time of various operations involved. Running time of an operation can be defined as no. of computer instructions executed per operation.

What effects run time of an algorithm?

- a. Computer used, the hardware platform
- b. Representation of abstract data types (ADT's)
- c. Efficiency of compiler
- d. Competence of implementer (programming skills)
- e. Complexity of underlying algorithm
- f. Size of the input

Algorithm Complexity

The *complexity* of an algorithm is a function describing the efficiency of the algorithm in terms of the amount of data the algorithm must process.

Suppose X is an algorithm and n is the size of input data, the time and space used by the Algorithm X are the two main factors which decide the efficiency of X .

- **Time Factor** – The time is measured by counting the number of key operations such as comparisons in sorting algorithm
- **Space Factor** – the space is measured by counting the maximum memory space required by the algorithm.

The complexity of an algorithm $f(n)$ gives the running time and/or storage space required by the algorithm in terms of n as the size of input data.

There are two main complexity measures of the efficiency of an algorithm:

1. Space Complexity

Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. Space required by an algorithm is equal to the sum of the following two components –

- A fixed part that is a space required to store certain data and variables that are independent of the size of the problem. For example simple variables & constant used program size etc.
- A variable part is a space required by variables, whose size depends on the size of the problem. For example dynamic memory allocation, recursion stacks space etc.

2. Time Complexity

Time Complexity of an algorithm represents the amount of time required by the algorithm to run to completion. Time requirements can be defined as a numerical function $T(n)$, where $T(n)$ can be measured as the number of steps, provided each step consumes constant time.

For example, addition of two n -bit integers takes n steps. Consequently, the total computational time is $T(n) = c \cdot n$, where c is the time taken for addition of two bits. Here, we observe that $T(n)$ grows linearly as input size increases.

Asymptotic Analysis

The main idea of asymptotic analysis is to have a measure of efficiency of algorithms that doesn't depend on machine specific constants, and doesn't require algorithms to be implemented and time taken by programs to be compared. Asymptotic notations are mathematical tools to represent time complexity of algorithms for asymptotic analysis. The following 3 asymptotic notations are mostly used to represent time complexity of algorithms.

Asymptotic analysis are input bound i.e., if there's no input to the algorithm it is concluded to work in a constant time. Other than the "input" all other factors are considered constant.

Using asymptotic analysis, we can very well conclude the best case, average case and worst case scenario of an algorithm.

Usually, time required by an algorithm falls under three types –

- **Best Case** – Minimum time required for program execution. In the best case analysis, we calculate lower bound on running time of an algorithm. We must know the case that causes minimum number of operations to be executed. In the linear search problem, the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on n). So time complexity in the best case would be 1.
- **Average Case** – Average time required for program execution. In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs. Arguably, average case is the most useful measure. Unfortunately this is typically a very difficult thing to measure.
- **Worst Case** – Maximum time required for program execution. The worst-case running time of an algorithm is an upper bound on the running time for any input. Knowing it gives us a guarantee that the algorithm will never take any longer. For Linear Search, the worst case happens when the element to be searched is not present in the array.

Asymptotic Notations

Asymptotic notation is a way of expressing the cost of an algorithm. Goal of Asymptotic notation is to simplify Analysis by getting rid of unneeded information

Following are the asymptotic notations:

Big Oh Notation, O

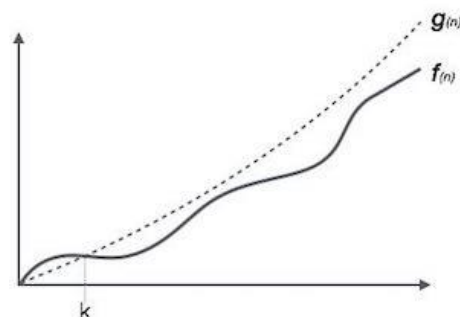
The $O(n)$ is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or longest amount of time an algorithm can possibly take to complete.

A more detailed explanation and definition of Big O analysis would be this: it measures the efficiency of an algorithm based on the **time** it takes for the algorithm to run as a function of the input size. Think of the input simply as what goes into a function – whether it is an array of numbers, a linked list, etc. When doing Big-O analysis, "input" can mean a lot of different things depending on the problem being solved.

We use big-O notation for **asymptotic upper bounds**, since it bounds the growth of the running time from above for large enough input sizes. Because big-O notation gives only an asymptotic upper bound, and not an asymptotically tight bound, we can make statements that at first blush seem incorrect, but are technically correct. For example, it is absolutely correct to say that binary search runs in $z(n) O(n) O(n)$ time. That's because the running time grows no faster than a constant times n . In fact, it grows slower.

For example, for a function $f(n)$

$O(f(n)) = \{ g(n) :$
 There exists $c > 0$ and n_0
 Such that $g(n) \leq c \cdot f(n)$ for all $n > n_0. \}$



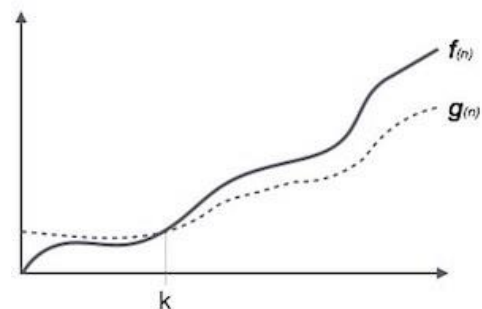
Omega Notation, Ω

The $\Omega(n)$ is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or best amount of time an algorithm can possibly take to complete. We use big- Ω notation for **asymptotic lower bounds**, since it bounds the growth of the running time from below for large enough input sizes.

We can also make correct, but imprecise, statements using big- Ω notation. For example, just as if you really do have a million dollars in your pocket, you can truthfully say "I have an amount of money in my pocket, and it's at least 10 dollars," you can also say that the worst-case running time of binary search is $\Omega(1)$, because it takes at least constant time.

For example, for a function $f(n)$

$\Omega(f(n)) = \{ g(n) \mid$
 There exists $c > 0$ and n_0
 Such that $g(n) \leq c \cdot f(n)$ for all $n > n_0 \}$



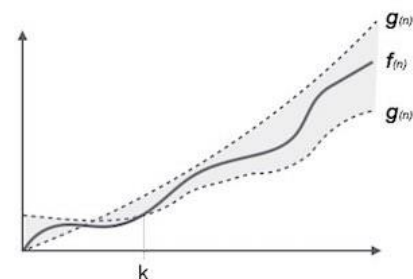
Theta Notation, Θ

The $\Theta(n)$ is the formal way to express both the lower bound and upper bound of an algorithm's running time.

We can also make correct, but imprecise, statements using big- Ω notation. For example, just as if you really do have a million dollars in your pocket, you can truthfully say "I have an amount of money in my pocket, and it's at least 10 dollars," you can also say that the worst-case running time of binary search is $\Omega(1)$, because it takes at least constant time.

It is represented as following –

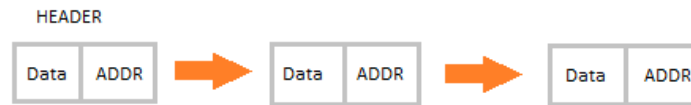
$\Theta(f(n)) = \{ g(n) \mid$ if and only if $g(n) = O(f(n))$
 And
 $g(n) = \Omega(f(n))$ for all $n > n_0 \}$



Linked List

Linked List is a linear data structure and it is very common data structure which consists of group of nodes in a sequence which is divided in two parts. Each node consists of its own data and the address of the next node and forms a chain. Linked Lists are used to create trees and graphs.

Linked list data structure provides better memory management than arrays. Because linked list is allocated memory at run time, so, there is no waste of memory. Performance wise linked list is slower than array because there is no direct access to linked list elements. Linked list is proved to be a useful data structure when the number of elements to be stored is not known ahead of time.



Advantages of Linked Lists

- They are a dynamic in nature which allocates the memory when required.
- Insertion and deletion operations can be easily implemented.
- Stacks and queues can be easily executed.
- Linked List reduces the access time.
- It is less expensive

Disadvantages of Linked Lists

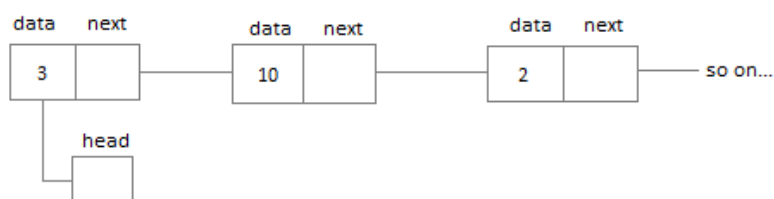
- The memory is wasted as pointers require extra memory for storage.
- No element can be accessed randomly; it has to access each node sequentially.
- Reverse Traversing is difficult in linked list
- Different amount of time is required to access each element.
- It is not easy to sort the elements stored in the linear linked list

Applications of Linked Lists

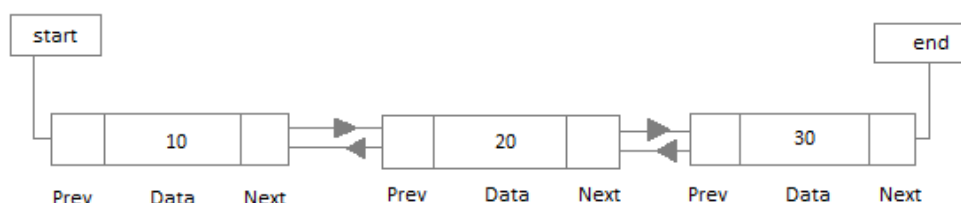
- Linked lists are used to implement stacks, queues, graphs, etc.
- Linked lists let you insert elements at the beginning and end of the list.
- In Linked Lists we don't need to know the size in advance.
- Hash tables use linked lists for collision resolution.
- To implement the symbol table in compiler construction.
- Any "File Requester" dialog uses a linked list.

Types of Linked Lists

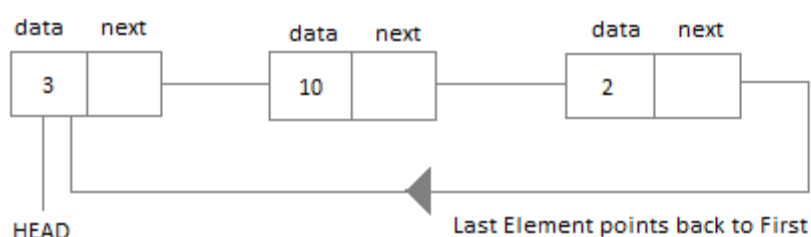
1. **Singly Linked List:** Singly linked lists contain nodes which have a data part as well as an address part i.e. next, which points to the next node in sequence of nodes. The operations we can perform on singly linked lists are insertion, deletion and traversal.
In Singly Linked List we can traverse only in forward direction because LINK part contains address of the next node in the list. It is not possible to traverse in backward direction in Singly Linked List.



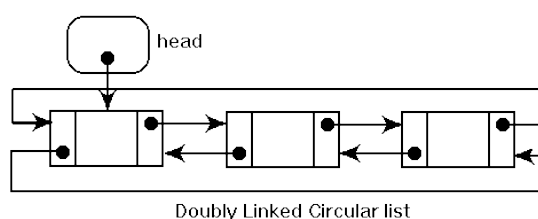
2. **Doubly Linked List:** Doubly Linked List is a collection of variable number of nodes in which each node consists of three parts. First part contains an address of previous node, second part contains a value of the node and third part contains an address of the next node. In a doubly linked list, each node contains two links the first link points to the previous node and the next link points to the next node in the sequence.



3. **Circular Linked List:** Circular Singly Linked List is a collection of variable number of nodes in which each node consists of two parts. First part contains a value of the node and second part contains an address of the next node such that LINK part of the last node contains an address of the first node. In the circular linked list the last node of the list contains the address of the first node and forms a circular chain.



4. **Circular Doubly Linked List:** Circular Doubly Linked List is a collection of variable number of nodes in which each node consists of three parts. First part contains an address of previous node, second part contains a value of the node and third part contains an address of the next node such that RPTR part of the last node contains an address of the first node and LPTR part of the first node contains an address of the last node in the list.



Application of Circular Linked List

- The real life application where the circular linked list is used is our Personal Computers, where multiple applications are running. All the running applications are kept in a circular linked list and the OS gives a fixed time slot to all for running. The Operating System keeps on iterating over the linked list until all the applications are completed.
- Another example can be Multiplayer games. All the Players are kept in a Circular Linked List and the pointer keeps on moving forward as a player's chance ends.

- Circular Linked List can also be used to create Circular Queue. In a Queue we have to keep two pointers, FRONT and REAR in memory all the time, where as in Circular Linked List, only one pointer is required.

Linear Linked List

The element can be inserted in linked list in 2 ways:

- Insertion at beginning of the list.
- Insertion at the end of the list.

We will also be adding some more useful methods like:

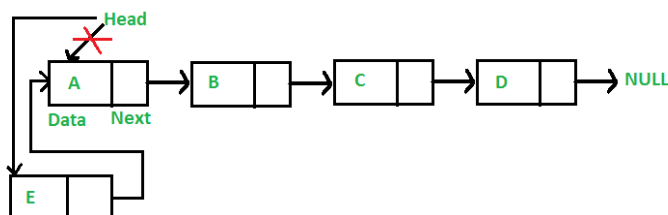
- Checking whether Linked List is empty or not.
- Searching any element in the Linked List
- Deleting a particular Node from the List

Insertion at the Beginning

Steps to insert a Node at beginning:

1. The first Node is the Head for any Linked List.
2. When a new Linked List is instantiated, it just has the Head, which is Null.
3. When Else, the Head holds the pointer to the first Node of the List.
4. We want to add any Node at the front; we must make the head point to it.
5. And the Next pointer of the newly added Node must point to the previous Head, whether it be NULL (in case of new List) or the pointer to the first Node of the List.
6. The previous Head Node is now the second Node of Linked List, because the new Node is added at the front.

```
int LinkedList :: addAtFront (node *n) {
    int i = 0;    //making the next of the new Node point to Head
    n->next = head; //making the new Node as Head
    head = n;
    i++;    //returning the position where Node is added
    return i;
}
```



Inserting at the End

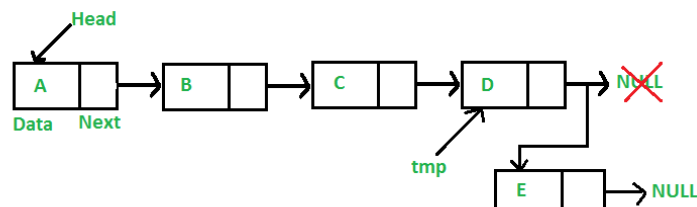
Steps to insert a Node at the end:

1. If the Linked List is empty then we simply, add the new Node as the Head of the Linked List.
2. If the Linked List is not empty then we find the last node, and make it' next to the new Node, hence making the new node the last Node.

```
int LinkedList :: addAtEnd (node *n) {    //If list is empty
    if(head == NULL) {    //making the new Node as Head
```

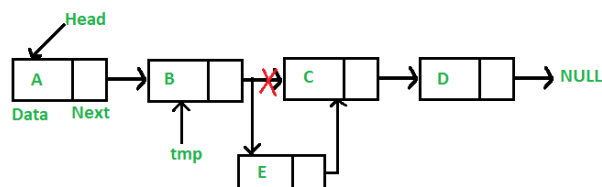
```

        head = n;    //making the next point of the new Node as Null
        n->next = NULL;
    }
    else {           //getting the last node
        node *n2 = getLastNode();
        n2->next = n;
    }
}
    
```



To Insert New Node Before Given Node in Linked List

1. In order to insert a new node before a given node in the linked list we have to follow the below steps:
First we have to check whether a free node is available in the **Availability Stack** or not. If a free node is available then we can allocate memory to a new node.
2. After creating a new node we have to check whether the linked list is empty or not. We have two possibilities:
 - **Linked List is empty (FIRST=NULL).** Hence the list is empty, the specified node is not found in the linked list. In this case we cannot insert a new node before a given node.
 - **Linked List is not empty (FIRST ≠ NULL).** In this case we have to traverse from the first node to the last node in the list until the given node is found. If the node is found in the linked list then we can insert a new node before that node; otherwise, we cannot insert a new node before a given node.



Searching for an Element in the List

In searching, we do not have to do much; we just need to traverse like we did while getting the last node. In this case, we will also compare the **data** of the Node. If we get the Node with the same data, we will return it; otherwise, we will make our pointer point to the next Node, and so on.

```

node* LinkedList :: search(int x) {
    node *ptr = head;
    while(ptr != NULL && ptr->data != x) {
        //until we reach the end or we find a Node with data x, we keep
        moving
        ptr = ptr->next;
    }
    return ptr;
}
    
```

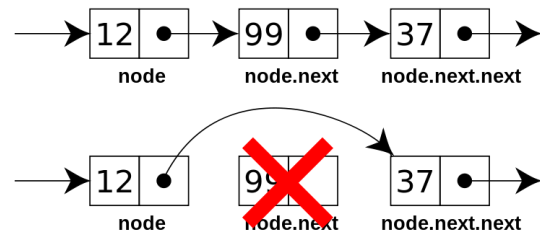
Deleting a Node from the List

Deleting a node can be done in many ways, like we first search the Node with **data** which we want to delete and then we delete it. In our approach, we will define a method which will take the **data** to be deleted as argument, will use the search method to locate it and will then remove the Node from the List.

To remove any Node from the list, we need to do the following:

- If the Node to be deleted is the first node, then simply set the Next pointer of the Head to point to the next element from the Node to be deleted.
- If the Node is in the middle somewhere, then find the Node before it, and make the Node before it point to the Node next to it.

```
node* LinkedList :: deleteNode(int x) {
    //searching the Node with data x
    node *n = search(x);
    node *ptr = head;
    if(ptr == n) {
        ptr->next = n->next;
        return n;
    }
    else {
        while(ptr->next != n) {
            ptr = ptr->next;
        }
        ptr->next = n->next;
        return n; } }
```



Linked lists have following drawbacks:

- Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists.
- Extra memory space for a pointer is required with each element of the list
- Arrays have better cache locality that can make a pretty big difference in performance.

Linked List vs. Array

Both Arrays and Linked List can be used to store linear data of similar types, but they both have some advantages and disadvantages over each other.

Following are the points in favor of Linked Lists.

- The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage, and in practical uses, upper limit is rarely reached.
- Inserting a new element in an array of elements is expensive; because room has to be created for the new elements and to create room existing elements have to shift.

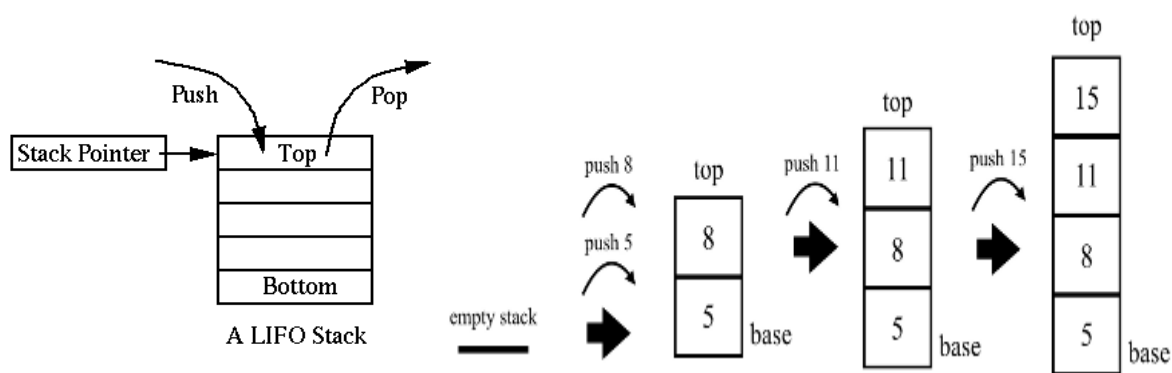
For example, suppose we maintain a sorted list of IDs in an array `id []`.

`id [] = [1000, 1010, 1050, 2000, 2040,]`. And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000).

Deletion is also expensive with arrays unless some special techniques are used. For example, to delete 1010 in `id[]`, everything after 1010 has to be moved.

STACKS

- Stack is abstract data type and linear data structure. It is named stack as it behaves like a real-world stack, for example – deck of cards or pile of plates etc.
- The basic concept can be illustrated by thinking of your data set as a stack of plates or books where you can only take the top item off the stack in order to remove things from it. This structure is used all throughout programming.
- The basic implementation of a stack is also called a LIFO (Last In First Out).
- A Stack is data structure in which addition of new element or deletion of existing element always takes place at a same end. This end is known as the top of the stack. That means that it is possible to remove elements from a stack in reverse order from the insertion of elements into the stack.



- In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.
- A stack can be implemented by means of Array, Structure, Pointer and Linked-List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays which makes it a fixed size stack implementation.

Basic Features of Stack

1. Stack is an ordered list of similar data type.
2. Stack is a **LIFO** structure. (Last in First out).
3. **push()** function is used to insert new elements into the Stack and **pop()** is used to delete an element from the stack. Both insertion and deletion are allowed at only one end of Stack called **Top**.
4. Stack is said to be in **Overflow** state when it is completely full and is said to be in **Underflow** state if it is completely empty.

Basic Operations on Stack

There are basically three operations that can be performed on stacks. They are 1) inserting an item into a stack (push). 2) Deleting an item from the stack (pop). 3) Displaying the contents of the stack.

- **push()** – pushing (storing) an element on the stack.
- **pop()** – removing (accessing) an element from the stack.

These are two primary operations done onto stack. To use a stack efficiently we need to check status of stack as well. For the same purpose, the following functionality is added to stacks –

- **peek()** – get the top data element of the stack, without removing it.
- **isFull()** – check if stack is full.
- **isEmpty()** – check if stack is empty.

At all times, we maintain a pointer to the last PUSHed data on the stack. This pointer always represents the top of the stack, hence named **top**. The **top** pointer provides top value of the stack without actually removing it.

1. PUSH Operation

The process of putting a new data element onto stack is known as **PUSH** Operation. Push operation involves series of steps –

- **Step 1** – Check if stack is full.
- **Step 2** – If stack is full, produce error and exit.
- **Step 3** – If stack is not full, increment **top** to point next empty space.
- **Step 4** – Add data element to the stack location, where top is pointing.
- **Step 5** – return success.

If linked-list is used to implement stack, then in step 3, we need to allocate space dynamically.

Algorithm for PUSH operation

```
begin procedure push: stack, data
if stack is full
return null
endif
top ← top + 1
stack[top] ← data
end procedure
```

2. POP Operation

Accessing the content while removing it from the top of stack is known as pop operation. In array implementation of pop() operation, data element is not actually removed, instead **top** is decremented to a lower position in stack to point to next value. But in linked-list implementation, pop() actually removes data element and de-allocates memory space.

A **POP** operation may involve the following steps –

- **Step 1** – Check if stack is empty.
- **Step 2** – If stack is empty, produce error and exit.
- **Step 3** – If stack is not empty, access the data element at which **top** is pointing.
- **Step 4** – Decrease the value of top by 1.
- **Step 5** – return success.

Algorithm for POP operation

```
begin procedure pop: stack
if stack is empty
return null
endif
data ← stack[top]
top ← top - 1
return data
end procedure
```

Applications of STACK

- The simplest application of a stack is to reverse a word. You push a given word to stack - letter by letter - and then pop letters from the stack.
- Another application is an "undo" mechanism in text editors; this operation is accomplished by keeping all text changes in a stack.
- **Backtracking.** This is a process when you need to access the most recent data element in a series of elements.
- Language processing:

Implementation of STACK

1. **Array Based Implementation:** In an array-based implementation we maintain the following fields: an array *A* of a default size (≥ 1), the variable *top* that refers to the top element in the stack and the *capacity* that refers to the array size. The variable *top* changes from -1 to capacity - 1. We say that a stack is empty when $top = -1$, and the stack is full when $top = capacity - 1$. In a fixed-size stack abstraction, the capacity stays unchanged; therefore when *top* reaches *capacity*, the stack object throws an exception.
2. **Linked List Based Implementation:** Linked List-based implementation provides the best (from the efficiency point of view) dynamic stack implementation. Using a linked list is one way to implement a stack so that it can handle *essentially* any number of elements.

Expression Notation

An expression is defined as a number of operands or data items combined using several operators. There are basically three types of notation for an expression. Another application to stack is calculation of stack.

1. Prefix notation

The infix notation is what we come across in our general mathematics, where the operator is written in-between the operands. For example: the expression to add two numbers *A* and *B* is written in infix notation as:

$$A + B$$

Note that the operator '+' is written in-between the operands *A* and *B*. The reason is why this notation is called infix.

2. Prefix notation

The prefix notation is the notation in which the operator is written before the operands, it is also called the **polish notation** in the honor of mathematician Jan Lukasiewicz who developed this notation. The same expression when written in prefix notation looks like:

+AB

As the operator '+' is written before the operands A and B, this notation is called the prefix notation.

3. Postfix notation

In the postfix notation the operators are written after the operand, so it is called the postfix notation, it also known as suffix notation or reverse polish notation. The above expression if written in postfix expression looks like follows:

AB+

Examples:

Infix	Postfix	Prefix	Notes
$A * B + C / D$	$A B * C D / +$	$+ * A B / C D$	multiply A and B, divide C by D, add the results
$A * (B + C) / D$	$A B C + * D /$	$/ * A + B C D$	add B and C, multiply by A, divide by D
$A * (B + C / D)$	$A B C D / + *$	$* A + B / C D$	divide C by D, add B, multiply by A

Advantage of using postfix notation

Although human beings are quite used to work with mathematical expressions in infix notation, which is rather complex, as using this notation one has to remember a set of non-trivial rules. That set of rules must be applied to expressions in order to determine the final value. These rules include precedence, BODMAS, and associativity.

Using infix notation one cannot tell the order in which operator should be applied by only looking at expression. Whenever an infix expression consists of more than one operator, the precedence rules (BODMAS) should be applied to decide that which operator (and operand associated with that operator) is evaluated first. As compared to postfix notation, which is much easier to work with or evaluate.

In a postfix expression operands appear before the operators, there is no need for operator precedence and other rules. As soon as an operator appears in the postfix expression during scanning of postfix expression the topmost operands are popped off and are calculated applying the encountered operator. Place the result back onto the stack, doing so the stack will contain finally a single value at the end of process.

Notation Conversions

Converting Expressions from one notation to another notation is called notation conversions. We must follow some rules for such conversions and that are explained below:

In $a + b * c$, the expression part $b * c$ will be evaluated first, with multiplication as precedence over addition. We here use parenthesis for $a + b$ to be evaluated first, like $(a + b) * c$.

Operator precedence

Exponential operator	\wedge or $\$$	Highest precedence
Multiplication/Division	$*$, $/$	Next precedence
Addition/Subtraction	$+$, $-$	Least precedence

The rules to be remembered during *infix to postfix* conversion are:

- 1) Parenthesize the expression starting from left to right.
- 2) During parenthesizing the expression, the operands associated with operator having higher precedence are first parenthesized. For example $B * C$ is parenthesized first before $A + B$.
- 3) The sub-expression which has been converted into postfix is to be treated as single operand.
- 4) Once the expression is converted to postfix form remove the parenthesis.

Example: Give the postfix form for $A + [(B + C) + (D + E) * F] / G$

Solution:

Evaluation order is;

```

A + [ { ( BC + ) + ( DE + ) * F } / G ]
A + [ { ( BC + ) + ( DE + F * ) } / G ]
A + [ ( BC + DE + F * + ) / G ]
A + [ BC + DE + F * + G / ]
ABC + DE + F * + G / +      ---postfix form
    
```

Algorithm to convert infix expression to postfix expression

Postfix (Q, P)

Suppose Q is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression P.

1. Push "(" onto the STACK, and add ")" to the end of Q.
2. Scan Q from left to right and repeat steps 3 to 6 for each element of Q until the STACK is empty.
3. If an operand is encountered, add it to P.
4. If a left parenthesis is encountered, push it onto STACK.
5. If an operator \emptyset is encountered, then:
 - a. Add \emptyset to STACK.

[End of if structure]

 - b. Repeatedly pop from the STACK and add to P each operator (On the top of STACK) which has the same precedence as or higher precedence than \emptyset .
6. If a right parenthesis is encountered, then:
 - a. Repeatedly pop from STACK and add to P each operator (on the top of STACK) until a left parenthesis is encountered.
 - b. Remove the left parenthesis. (Do not add the left parenthesis to P)

[End of if structure]
[End of step 2 loop.]
7. Exit.

Algorithm to convert infix expression to prefix expression

1. Reverse the input string.
2. Examine the next element in the input.
3. If it is operand, add it to the output string.
4. If it is opening parenthesis, push it on stack.
5. If it is an operator then,
 - a. If stack is empty, push operator on stack.
 - b. If the top of stack is opening parenthesis push operator on stack.
 - c. If it has same or higher priority the top of stack, push operator on P.
 - d. Else pop the operator for the stack and add it to P, repeat 5.
6. If it is a closing parenthesis, pop operator from stack and add them to P until an opening parenthesis is encountered. Pop and discard opening parenthesis.
7. If there is more input go to step 2.
8. If there is no more input, pop the remaining operators and add them to P.
9. Reverse the output string.

Algorithm to evaluate a postfix Expression

This algorithm finds the VALUE of an arithmetic expressions P written in postfix notations. The following algorithm, which uses a STACK to hold operands, evaluates P.

1. Add a right parenthesis ")" at the end of P.
2. Scan P from left to right and repeat steps 3 and 4 for each element of p until the sentinel "(" is encountered.
3. If an operand is encountered, put in on STACK.
4. If an operator is encountered then,
 - a. Remove the two top elements of STACK, where A is the top element and B is the next to top element.
 - b. Evaluate $B \text{ } \theta \text{ } A$.
 - c. Place the result of (b) back on STACK.

[End of if structure]
[End of step 2 loop]
5. Set Value equal to the top element on stack.
6. Exit.

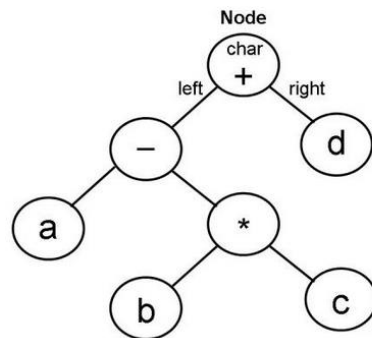
Binary Expression Tree

An expression Tree is a strictly binary tree in which leaf nodes contain operands and non-leaf nodes contain operator, root nodes contains the operator that is applied to result of left and right sub-trees.

Once we obtain the Expression tree for a particular expression, its conversion into different notations (infix, prefix, and postfix) and evaluation become a matter of traversing the Expression tree.

A binary expression tree does not contain parenthesis, the reason for this is that for evaluating an expression using expression tree, structure of tree itself decides order of the operators.

When we run pre-order traversal (visit root, left child and then right child) on the binary expression tree we get prefix notation of the expression, similarly, a post order traversal (visit left child, right child then root.)



AST Tree POSTFIX Expression **abc*-d+**

Prefix → Infix

1. Create the expression tree from the prefix expression.
2. Run in-order traversal on the tree

Prefix → Postfix

1. Create the Expression tree for the prefix expression.
2. Run post-order traversal on the tree.

Post → Infix

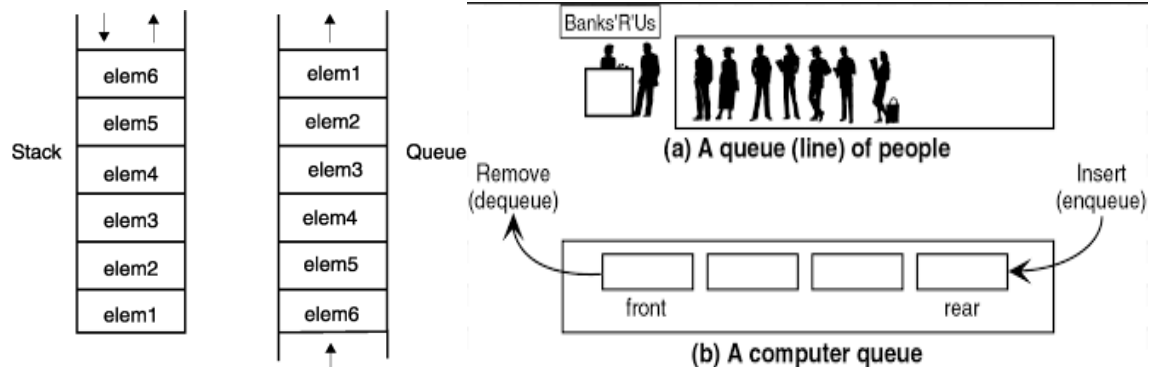
1. Create the expression tree for the postfix expression.
2. Run in-order traversal on the tree.

Post → Prefix

1. Create the expression tree from the postfix expression.
2. Run pre-order traversal on the tree.

QUEUES

- Queue is also an abstract data type or a linear data structure, in which the first element is inserted from one end called **REAR** (also called tail), and the deletion of existing element takes place from the other end called as **FRONT** (also called head).
- This makes queue as FIFO data structure, which means that element inserted first will also be removed first (**First In First Out**)
- The process to add an element into queue is called **Enqueue** and the process of removal of an element from queue is called **Dequeue**.
- A real world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world example can be seen as queues at ticket windows & bus-stops.
- Same as stack, queue can also be implemented using Array, Linked-list, Pointer and Structures. For the sake of simplicity we shall implement queue using one-dimensional array.
- A real life scenario in the form of example for queue will be the queue of people waiting to accomplish a particular task where the first person in the queue is the first person to be served first.
- Other examples can also be noted within a computer system where the queue of tasks arranged in list to perform for the line printer, for accessing the disk storage, or even in time sharing system for the use of CPU.



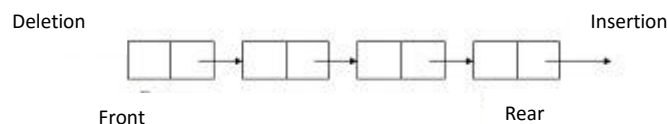
Basic Features of Queues

- Like Stack, Queue is also an ordered list of elements of similar data types.
- Queue is a FIFO (First in First Out) structure.
- Once a new element is inserted into the Queue, all the elements inserted before the new element in the queue must be removed, to remove the new element.
- **Peek ()** function is often used to return the value of first element without De-queuing it.

Different Types of Queues

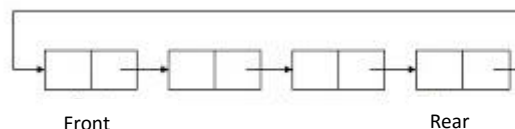
1. Simple or linear queue

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



2. Circular queue

Another common implementation of a queue is a circular buffer. "Buffer" is a general name for a temporary storage location, although it often refers to an array, as it does in this case. A circular buffer, cyclic buffer or ring buffer is a data structure that uses a single, fixed-size buffer as if it were connected end-to-end. This structure lends itself easily to buffering data streams.

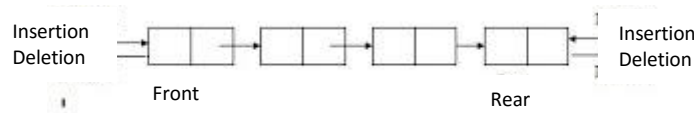


3. Priority queue

The Priority Queue ADT has the same interface as the Queue ADT, but different semantics. The semantic difference is that the item that is removed from the queue is not necessarily the first one that was added. Rather, it is whatever item in the queue has the highest priority.

4. De-queue

A double-ended queue (De-queue) is an abstract data type that generalizes a queue, for which elements can be added to or removed from either the front (head) or back (tail). It is also often called a head-tail linked list, though properly this refers to a specific data structure implementation.



Q. How circular queue is advantageous over sequential queue? Explain with an example.

Answer: The circular queue allows the entire array to store the elements without shifting any data within the queue. The array of circular queue is “wrapped around” which makes the rear of the queue adjacent of front of the queue. This helps in accessing the first or last element efficiently. After deletion operation the empty space of the array can be utilized for storing any new element.

In linear queue the elements can be inserted by incrementing rear pointer. Once rear reaches to a maximum size even though the elements are deleted from front there is a queue full condition hence in linear queue one cannot utilize the array fully. On the other hand, in circular queue the front is adjacent to rear hence circular queue allows an entire array to store the elements. The basic advantage of circular queue over linear queue is that one can efficiently use the array space and there is no wastage of memory.

Basic Operations on Queues

Queue operations may involve initializing or defining the queue, utilizing it and then completing erasing it from memory. Here are basic operations associated with queues –

- enqueue()** – add (store) an item to the queue.
- dequeue()** – remove (access) an item from the queue.

Few more functions are required to make above mentioned queue operation efficient. These are –

- peek()** – get the element at front of the queue without removing it.
- isfull()** – checks if queue is full.
- isempty()** – checks if queue is empty.

In queue, we always **dequeue** (or access) data, pointed by **front** pointer and while **enqueueing** (or storing) data in queue we take help of **rear** pointer.

1. Enqueue Operation

As queue maintains two data pointers, **front** and **rear**, its operations are comparatively more difficult to implement than stack.

The following steps should be taken to **enqueue** (insert) data into a queue –

Step 1 – Check if queue is full.

Step 2 – If queue is full, produce overflow error and exit.

Step 3 – If queue is not full, increment **rear** pointer to point next empty space

Step 4 – Add data element to the queue location, where rear is pointing

Step 5 – return success.

Sometimes, we also check that if queue is initialized or not to handle any unexpected situations.

Algorithm for Enqueue Operation

```
procedure enqueue(data)
  if queue is full
    return overflow
  endif
  rear  $\leftarrow$  rear + 1
  queue[rear]  $\leftarrow$  data
  return true
end procedure
```

2. Dequeue Operation

Accessing data from queue is a process of two tasks – access the data where **front** is pointing and remove the data after access. The following steps are taken to perform **dequeue** operation-

Step 1 – Check if queue is empty.

Step 2 – If queue is empty, produce underflow error and exit.

Step 3 – If queue is not empty, access data where **front** is pointing.

Step 3 – Increment **front** pointer to point next available data element

Step 5 – return success.

Algorithm for Enqueue Operation

```
procedure dequeue
  if queue is empty
    return underflow
  end if
  data = queue[front]
  front  $\leftarrow$  front - 1
  return true
end procedure
```

Applications of QUEUE

Queue, as the name suggests is used whenever we need to have any group of objects in an order in which the first one coming in also gets out first while the others wait for their turn, like in the following scenarios:

1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
2. In real life, Call Center phone systems will use Queues, to hold people calling them in an order, until a service representative is free.
3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive, First come first served.

Implementation: Queue can be implemented using an Array, Stack or Linked List. The easiest way of implementing a queue is by using an Array.

TREES

Introduction

Lists, stacks, and queues are all linear structures: in all three data structures, one item follows another but Trees are non-linear structure:

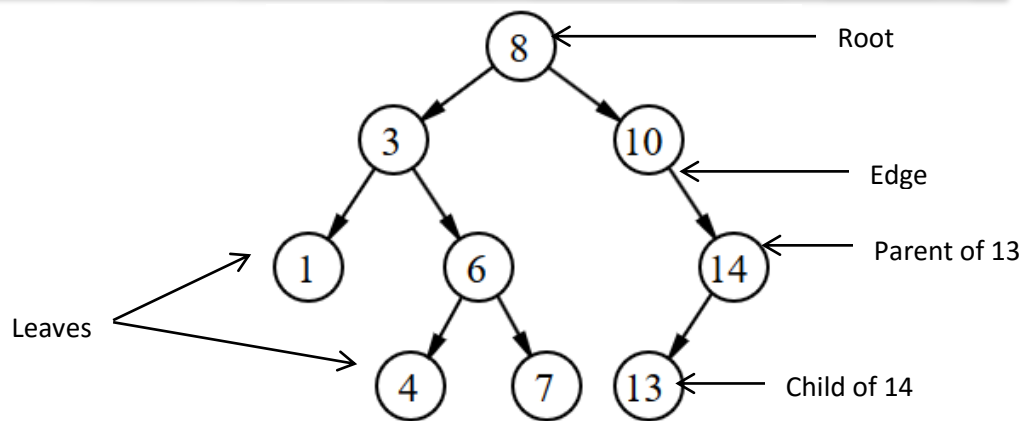
- More than one item can follow another.
- The number of items that follow can vary from one item to another.

A tree can be empty with no nodes or a tree is a structure consisting of one node called the root and zero or one or more sub-trees.

- One node is distinguished as a **root**.
- Every node is connected by a edge *from* exactly one other node; A direction is: *parent -> children*
- Each node can have *arbitrary* number of children. Nodes with no children are called **leaves**, or **external** nodes.
- Nodes, which are not leaves, are called **internal** nodes. Internal nodes have at least one child.
- Nodes with the same parent are called **siblings**.
- The **depth of a node** is the number of edges from the root to the node.
- The **height of a node** is the number of edges from the node to the deepest leaf.
- The **height of a tree** is a height of a root.

Following are important terms with respect to tree.

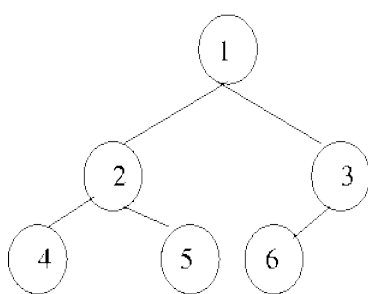
- **Path** – Path refers to sequence of nodes along the edges of a tree.
- **Root** – Node at the top of the tree is called root. There is only one root per tree and one path from root node to any node.
- **Parent** – any node except root node has one edge upward to a node called parent.
- **Child** – Node below a given node connected by its edge downward is called its child node.
- **Leaf** – Node which does not have any child node is called leaf node.
- **Sub-tree** – Sub-tree represents descendants of a node.
- **Visiting** – Visiting refers to checking value of a node when control is on the node.
- **Traversing** – Traversing means passing through nodes in a specific order.
- **Levels** – Level of a node represents the generation of a node. If root node is at level 0, then its next child node is at level 1, its grandchild is at level 2 and so on.
- **Keys** – Key represents a value of a node based on which a search operation is to be carried out for a node.



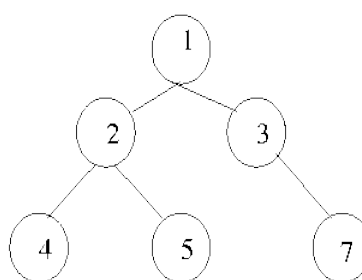
Binary Trees

A tree where each node can have no more than two children is a binary tree. A binary tree is either **empty** or consists of a node called the **root** together with two binary trees called the **left subtree** and the **right subtree**.

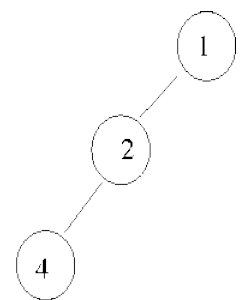
- If h = height of a binary tree,
max no. of leaves = 2^h
- max no. of nodes = $2^{h+1} - 1$
- A binary tree with height h and $2^{h+1} - 1$ nodes (or 2^h leaves) is called a **full binary tree**
- A binary tree in which each node has exactly zero or two children is called a **full binary tree**. In a full tree, there are no nodes with exactly one child.
- A **complete binary tree** is a tree, which is completely filled, with the possible exception of the bottom level, which is filled from left to right.



Complete Binary Tree

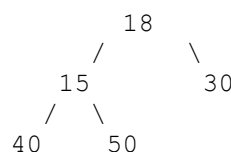


Not Complete



Not Complete

A Binary Tree whose every node is having either 0 or 2 child is called as **Strict Binary Tree**. Strictly binary tree data structure is used to represent mathematical expressions.



Binary Search Trees

Binary Search tree exhibits a special behavior. A binary search tree (BST) is a tree in which all nodes follows the below mentioned properties –

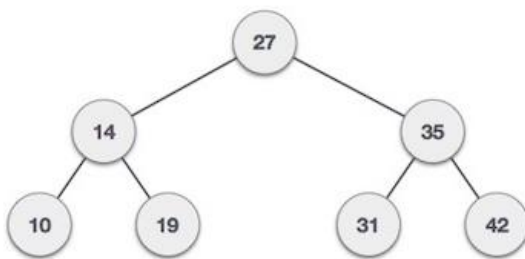
- The left sub-tree of a node has key less than or equal to its parent node's key.
- The right sub-tree of a node has key greater than or equal to its parent node's key.

Thus, a binary search tree (BST) divides all its sub-trees into two segments; *left* sub-tree and *right* sub-tree and can be defined as –

$$\text{left_subtree}(\text{keys}) \leq \text{node}(\text{key}) \leq \text{right_subtree}(\text{keys})$$

BST is a collection of nodes arranged in a way where they maintain BST properties. Each node has key and associated value. While searching, the desired key is compared to the keys in BST and if found, the associated value is retrieved.

An example of Binary Search Tree (BST):



We observe that the root node key (27) has all less-valued keys on the left sub-tree and higher valued keys on the right sub-tree.

BST Basic Operations

The basic operations that can be performed on binary search tree data structure are following

- **Insert** – insert an element in a tree / create a tree.
- **Search** – search an element in a tree.
- **Preorder Traversal** – traverse a tree in a preorder manner.
- **Inorder Traversal** – traverse a tree in an inorder manner.
- **Postorder Traversal** – traverse a tree in a postorder manner.

Insert Operation

The very first insertion creates the tree. Afterwards, whenever an element is to be inserted. First locate its proper location. Start search from root node then if data is less than key value, search empty location in left subtree and insert the data. Otherwise search empty location in right subtree and insert the data.

Algorithm:

```
If root is NULL
then create root node
return
```

```

If root exists then
  compare the data with node.data
  while until insertion position is located
  If data is greater than node.data
    goto right subtree
  else
    goto left subtree
  endwhile
  insert data
end If
    
```

Search Operation

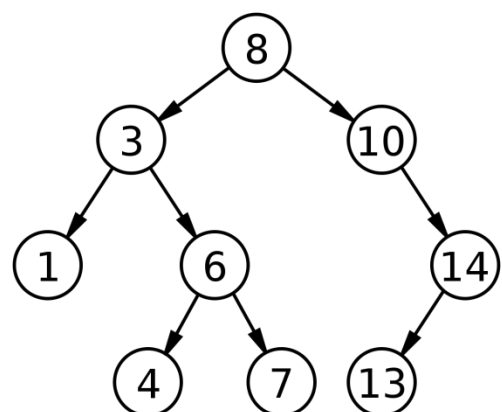
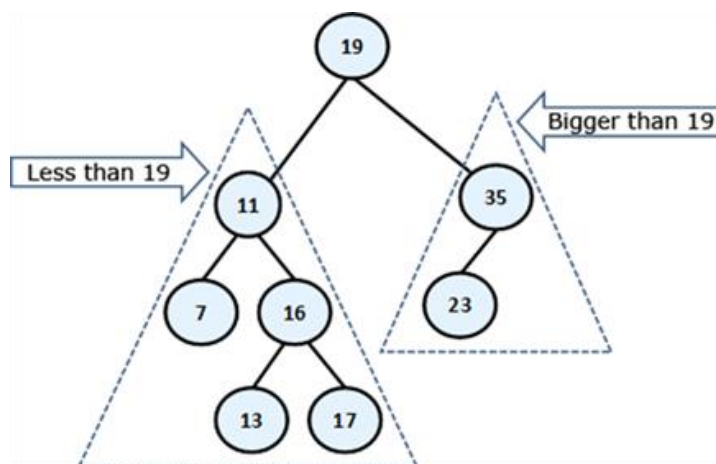
Whenever an element is to be search; Start search from root node then if data is less than key value, search element in left subtree otherwise search element in right subtree. Follow the same algorithm for each node.

Algorithm:

```

If root.data is equal to search.data
  return root
else
  while data not found
  If data is greater than node.data
    goto right subtree
  else
    goto left subtree
  If data found
    return node
  endwhile
  return data not found
end if
    
```

Example of BST



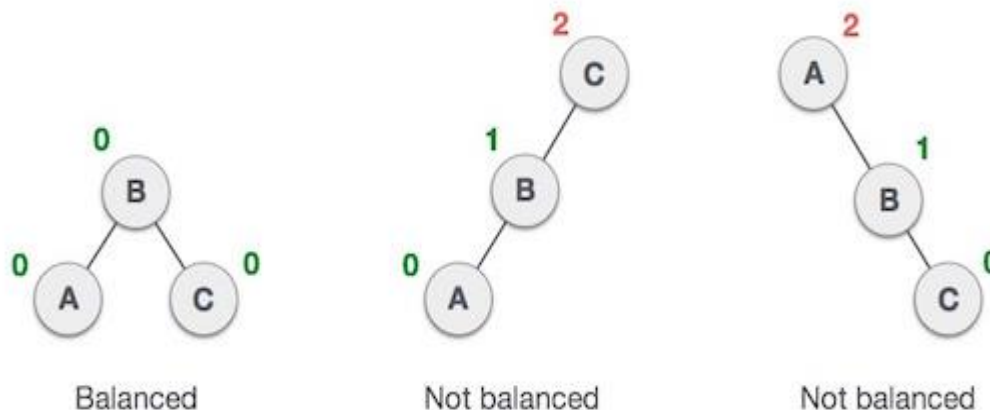
AVL Trees

An **AVL tree** is another balanced binary search tree. Named after their inventors, **Adelson-Velskii** and **Landis**, they were the first dynamically balanced trees to be proposed.

An AVL tree is a binary search tree which has the following properties:

1. The sub-trees of every node differ in height by at most one. This difference is called **Balance Factor**.
2. Every sub-tree is an AVL tree.

Here we see that the first tree is balanced and next two trees are not balanced:



In second tree, the left subtree of **C** has height 2 and right subtree has height 0, so the difference is 2. In third tree, the right subtree of **A** has height 2 and left is missing, so it is 0, and the difference is 2 again. AVL tree permits difference (balance factor) to be only 1.

$$\text{Balance Factor} = \text{height}(\text{left-subtree}) - \text{height}(\text{right-subtree})$$

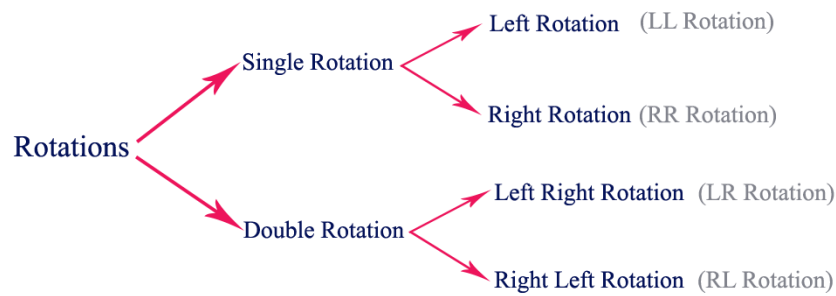
If the difference in the height of left and right sub-trees is more than 1, the tree is balanced using some rotation techniques.

AVL Rotations

To make itself balanced, an AVL tree may perform four kinds of rotations –

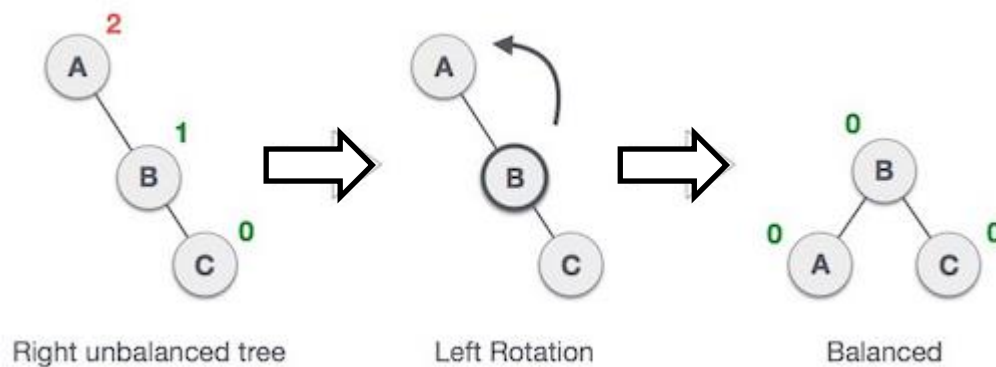
- Left rotation
- Right rotation
- Left-Right rotation
- Right-Left rotation

First two rotations are single rotations and next two rotations are double rotations. Two have an unbalanced tree we at least need a tree of height 2. With this simple tree, let's understand them one by one.



1. Left Rotation

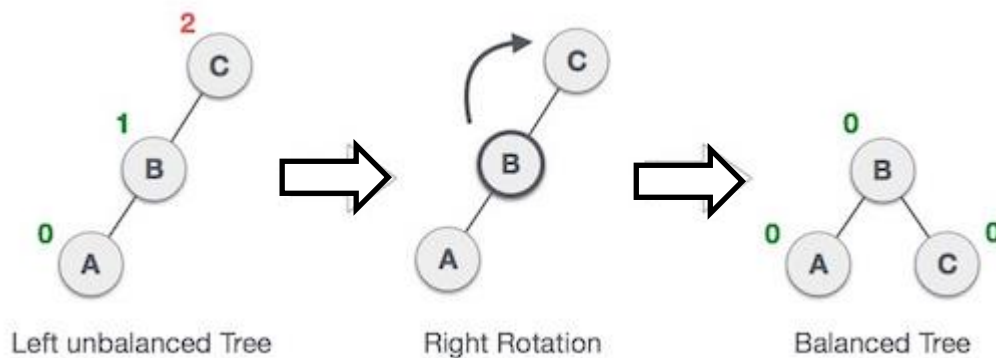
If a tree become unbalanced, when a node is inserted into the right subtree of right subtree, then we perform single left rotation –



In our example, node **A** has become unbalanced as a node is inserted in right subtree of A's right subtree. We perform left rotation by making **A** left-subtree of B.

2. Right Rotation

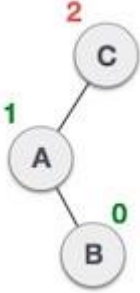
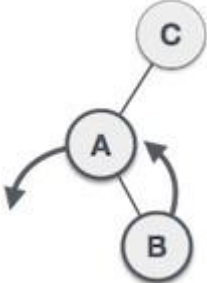
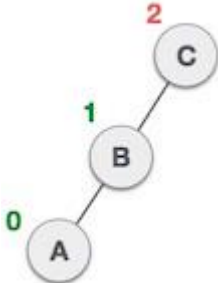
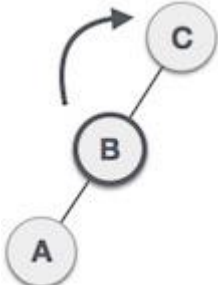
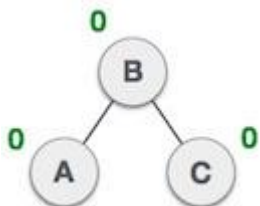
AVL tree may become unbalanced if a node is inserted in the left subtree of left subtree. The tree then needs a right rotation.



As depicted, the unbalanced node becomes right child of its left child by performing a right rotation.

3. Left-Right Rotation

Double rotations are slightly complex version of already explained versions of rotations. A left-right rotation is combination of left rotation followed by right rotation.

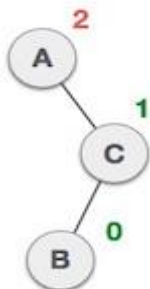
State	Action
	<p>A node has been inserted into right subtree of left subtree. This makes C an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.</p>
	<p>We first perform left rotation on left subtree of C. This makes A, left subtree of B.</p>
	<p>Node C is still unbalanced but now, it is because of left-subtree of left-subtree.</p>
	<p>We shall now right-rotate the tree making B new root node of this subtree. C now becomes right subtree of its own left subtree.</p>
	<p>The tree is now balanced.</p>

4. Right left Rotation

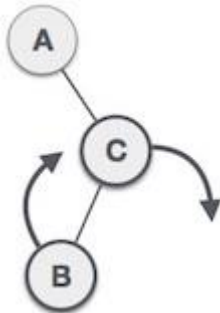
Second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation.

State

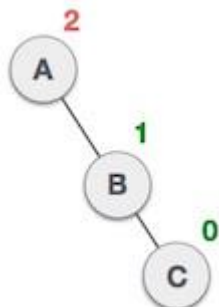
Action



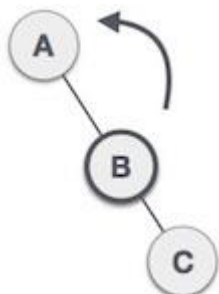
A node has been inserted into left subtree of right subtree. This makes **A** an unbalanced node, with balance factor 2.



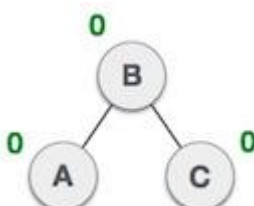
First, we perform right rotation along **C** node, making **C** the right subtree of its own left subtree **B**. Now, **B** becomes right subtree of **A**.



Node **A** is still unbalanced because of right subtree of its right subtree and requires a left rotation.



A left rotation is performed by making **B** the new root node of the subtree. **A** becomes left subtree of its right subtree **B**.



The tree is now balanced.

Splay Trees

A splay tree is just a binary search tree that has excellent performance in the cases where some data is accessed more frequently than others. The tree self-adjusts after lookup, insert and delete operations.

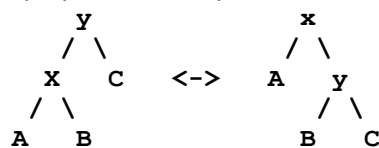
A splay tree does not necessarily have minimum height, nor does it necessarily have logarithmically bounded height.

In fact a splay tree may even be degenerate. However, it is based on the idea that if you recently used something you'll likely need it again soon, so it keeps the most commonly used data near the top where it is accessed most quickly.

There are three kinds of tree rotations that are used to move elements upward in the tree. These rotations have two important effects: they move the node being splayed upward in the tree, and they also shorten the path to any nodes along the path to the splayed node. This latter effect means that splaying operations tend to make the tree more balanced.

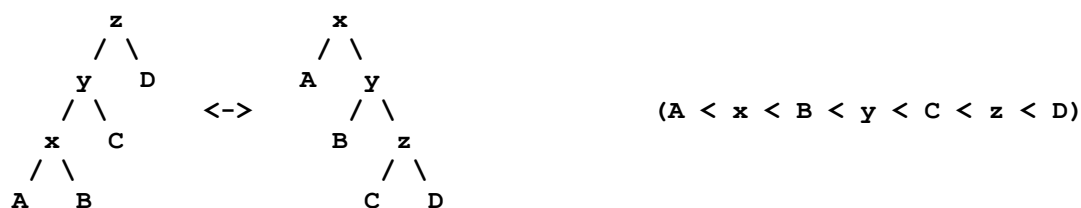
Rotation 1: Simple Rotation

The simple tree rotation used in AVL trees is also applied at the root of the splay tree, moving the splayed node x up to become the new tree root.



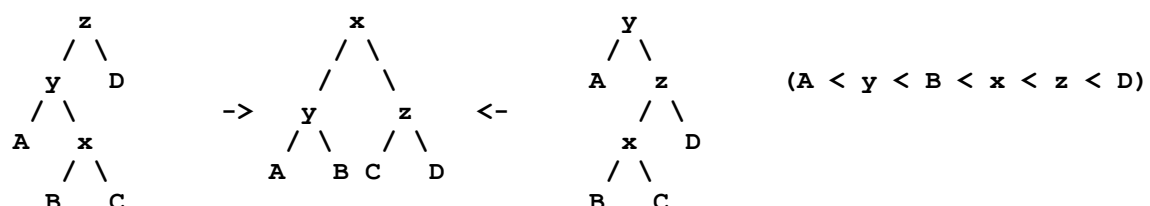
Rotation 2: Zig-Zig and Zag-Zag rotation

In the "zig-zig" case, the splayed node is the left child of a left child or the right child of a right child ("zag-zag").



Rotation 3: Zig-Zag Rotation

In the "zig-zag" case, the splayed node is the left child of a right child or vice-versa. The rotations produce a subtree whose height is less than that of the original tree. Thus, this rotation improves the balance of the tree.



Tree Traversal

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot random access a node in tree. There are three ways which we use to traverse a tree-

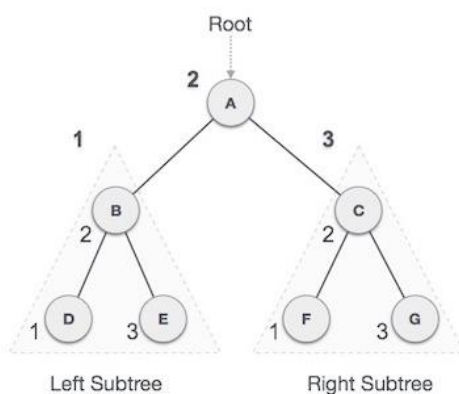
- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

Generally we traverse a tree to search or locate given item or key in the tree or to print all the values it contains.

1. In-Order Traversal

In this traversal method, the left subtree is visited first, then root and then the right subtree. We should always remember that every node may represent a subtree itself.

If a binary tree is traversed **inorder**, the output will produce sorted key values in ascending order.



We start from **A**, and following in-order traversal, we move to its left subtree **B**. **B** is also traversed in-ordered. And the process goes on until all the nodes are visited. The output of in-order traversal of this tree will be –

$D \rightarrow B \rightarrow E \rightarrow A \rightarrow F \rightarrow C \rightarrow G$

Algorithm:

Until all nodes are traversed –

Step 1 – Recursively traverse left subtree

Step 2 – Visit root node

Step 3 – Recursively traverse right subtree

2. Pre-Order Traversal

In this traversal method, the root node is visited first, then left subtree and finally right subtree.

We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-ordered. And the process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be –

$A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$

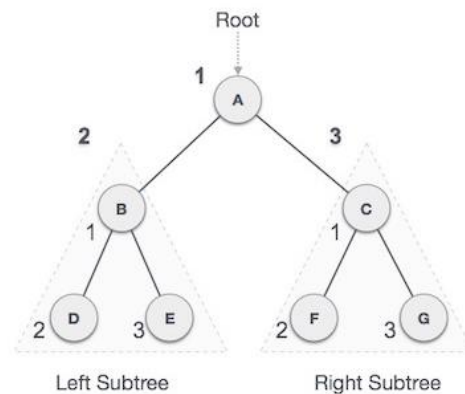
Algorithm:

Until all nodes are traversed –

Step 1 – Visit root node.

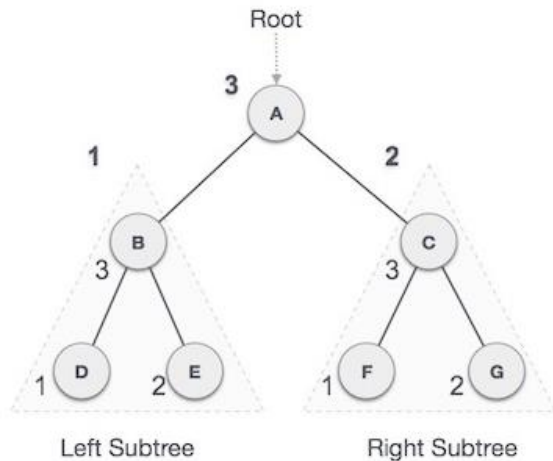
Step 2 – Recursively traverse left subtree.

Step 3 – Recursively traverse right subtree.



3. Post-Order Traversal

In this traversal method, the root node is visited last, hence the name. First we traverse left subtree, then right subtree and finally root.



We start from A, and following pre-order traversal, we first visit left subtree B. B is also traversed post-ordered. And the process goes on until all the nodes are visited. The output of post-order traversal of this tree will be –

$D \rightarrow E \rightarrow B \rightarrow F \rightarrow G \rightarrow C \rightarrow A$

Algorithm:

Until all nodes are traversed –

Until all nodes are traversed –

Step 1 – Recursively traverse left subtree.

Step 2 – Recursively traverse right subtree.

Step 3 – Visit root node.

B-Trees

B-Trees are a variation on binary search trees that allow quick searching in files on disk. Instead of storing one key and having two children, B-tree nodes have n keys and $n+1$ child, where n can be large. **B-tree** is a self-balancing tree data structure that keeps data sorted and allows searches, sequential access, insertions, and deletions in logarithmic time.

B-trees are a way to get better locality by putting multiple elements into each tree node. A B-tree of order m is a search tree in which each non-leaf node has up to m children.

In a B-Tree, the search operation is similar to that of Binary Search Tree. In a Binary search tree, the search process starts from the root node and every time we make a 2-way decision (we go to either

left sub-tree or right sub-tree). In B-Tree also search process starts from the root node but every time we make n -way decision where n is the total number of children that node has. In a B-Tree, the search operation is performed with $O(\log n)$ time complexity.

B-Trees must satisfy several invariants:

1. Every path from the root to a leaf has the same length
2. If a node has n children, it contains $n-1$ keys.
3. Every node (except the root) is at least half full.
4. The elements stored in a given sub-tree all have keys that are between the keys in the parent node on either side of the sub-tree pointer. (This generalizes the BST invariant.)
5. The root has at least two children if it is not a leaf.
6. All leaves on the tree must be at the same

Example of Insertion with Splitting

We will apply the stated rules above:

Insert: 5 9 3 7 1 2 8 6 0 4: Order: 4

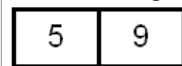
Step 1: Insert 5



Step 2: Insert 9

Root node not full.

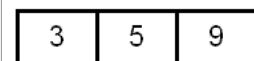
Put 9 to the right of 5



Step 3: Insert 3

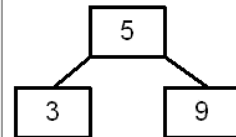
Root node not full.

Put 3 to the left of 5

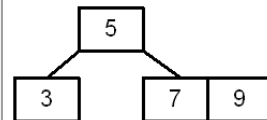


Step 4: Insert 7

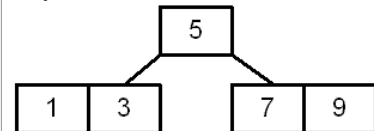
Root node is full, do aggressive node splitting.
Allocate a new (empty) node; make it the root, split pulling median key 5 into the new root:



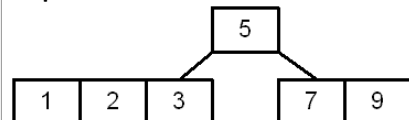
Then insert 7; with 9



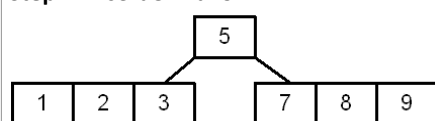
Step 5: Insert 1 with 3



Step 6: Insert 2 with 3



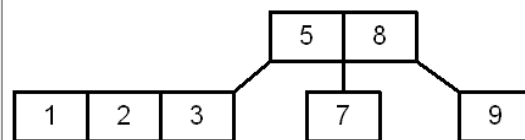
Step 7: Insert 8 with 9



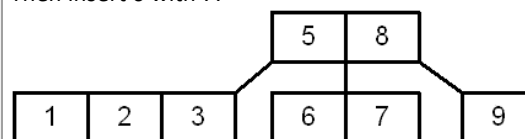
Step 8: Insert 6

It would go in with 7 8 9, but that node is full.

So split it, bringing its median child 8 into the root:

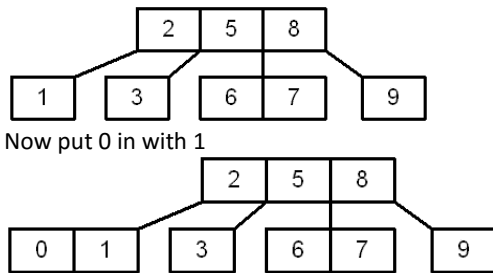


Then insert 6 with 7:



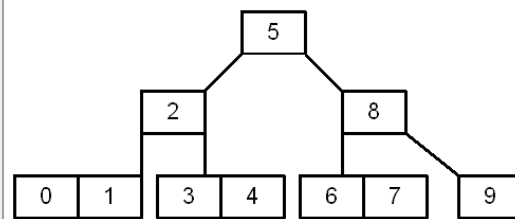
Step 9: Insert 0

0 would go in with 1 2 3, which is full, so split it, sending the median child 2 up to the root:



Step 10: Insert 4

It would be nice to just stick 4 in with 3, but the B-Tree algorithm requires splitting the full root (or any full node encountered - Aggressive Node Splitting). If we don't and one of the leaves becomes full, nowhere for median key of split node since root would be full, aggressive splitting prevents backing the median key up more than one level:



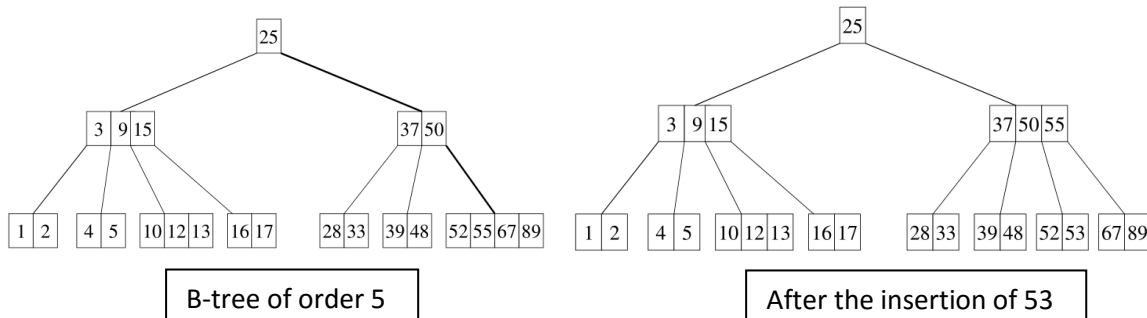
Operations on B-Tree

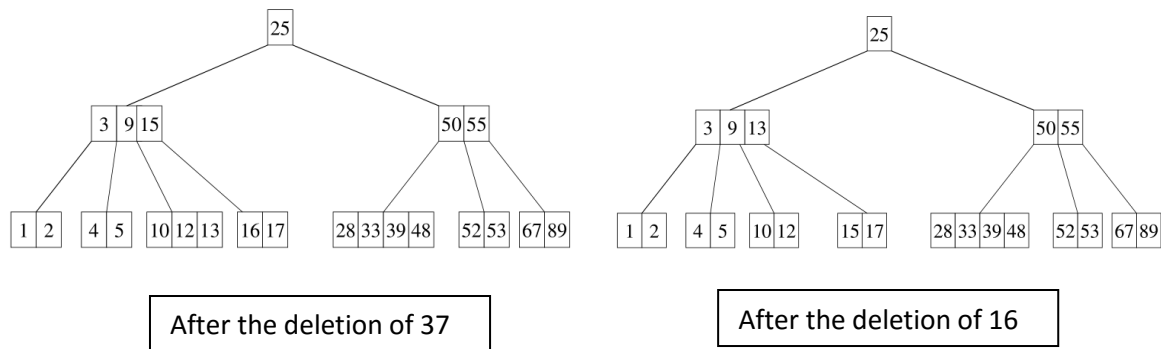
Insertion

- If tree is **Empty**, then create a new node with new key value and insert into the tree as a root node.
- If this makes the parent full, split it as well and push its middle item upwards.
- Continue doing this until either some space is found in an ancestor node, or a new root node is created.
- If the splitting is occurring to the root node, then the middle value becomes new root node for the tree and the height of the tree is increased by one.

Deletion

- Replace the item to be removed by its in-order successor (which is bound to be in a leaf node).
- Remove successor from its leaf node.
- This may cause underflow (fewer than $d/2$ records in that leaf node).
- Depending on how many records the sibling of has, this is fixed either by fusion or by transfer.





HASHING

- Hashing is a technique to convert a range of key values into a range of indexes of an array.
- Hashing is the technique used for performing almost constant time search in case of insertion, deletion and find operation.
- Hash Table is a data structure which store data in associative manner. In hash table, data is stored in array format where each data values have its own unique index value. Access of data becomes very fast if we know the index of desired data.
- Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of size of data. Hash Table uses array as a storage medium and uses hash technique to generate index where an element is to be inserted or to be located from.

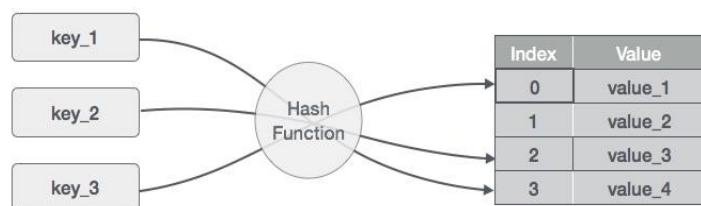
Hash Function

Each index (key) can be used for accessing the value in a constant search time. This mapping key must be simple to compute and must helping in identifying the associated value. Function which helps us in generating such kind of key-value mapping is known as **Hash Function**.

Given a collection of items, a hash function that maps each item into a unique slot is referred to as a **perfect hash function**. If we know the items and the collection will never change, then it is possible to construct a perfect hash function.

A good hash function should:

- Minimize collisions.
- Be easy and quick to compute.
- Distribute key values evenly in the hash table.
- Use all the information provided in the key.



Load Factor

- The load factor of a hash table is the ratio of the number of keys in the table to the size of the hash table.
- The higher the load factor, the slower the retrieval.
- With open addressing, the load factor cannot exceed 1. With chaining, the load factor often exceeds 1.
- Note that 6 of the 11 slots are now occupied. This is referred to as the **load factor**, and is commonly denoted by $\lambda = \text{number of items} / \text{table size}$. For this example, $\lambda = 6/11 = 0.54$.

Example of Hashing

Assume that we have the set of integer items 54, 26, 93, 17, 77, and 31. Here the hash function, sometimes referred to as the “remainder method,” simply takes an item and divides it by the table size, returning the remainder as its hash value ($h(item) = item \% 11$).

Table 4: Simple Hash Function Using Remainders

Item	Hash Value
54	10
26	4
93	5
17	6
77	0
31	9

Fig: Hash Table with six items.

0	1	2	3	4	5	6	7	8	9	10
77	None	None	None	26	93	17	None	None	31	54

For example, if the item 44 had been the next item in our collection, it would have a hash value of 0 ($44 \% 11 = 0$). Since 77 also had a hash value of 0, we would have a problem. According to the hash function, two or more items would need to be in the same slot. This is referred to as a **collision** (it may also be called a “clash”). Clearly, collisions create a problem for the hashing technique.

Basic Operations

Following are basic primary operations of a Hash table which are following.

1. Insert Operation

Whenever an element is to be inserted; Compute the hash code of the key passed and locate the index using that Hash code as index in the array. Use probing methods for empty location if an element is found at computed hash code.

2. Deletion Operation

Whenever an element is to be deleted; Compute the hash code of the key passed and locate the index using that Hash code as index in the array. Use probing methods to get element ahead if an element is not found at computed hash code. When found, store a dummy item there to keep performance of Hash table intact.

3. Search Operation

Whenever an element is to be searched; Compute the hash code of the key passed and locate the element using that Hash code as index in the array. Use probing methods to get element ahead if element not found at computed hash code.

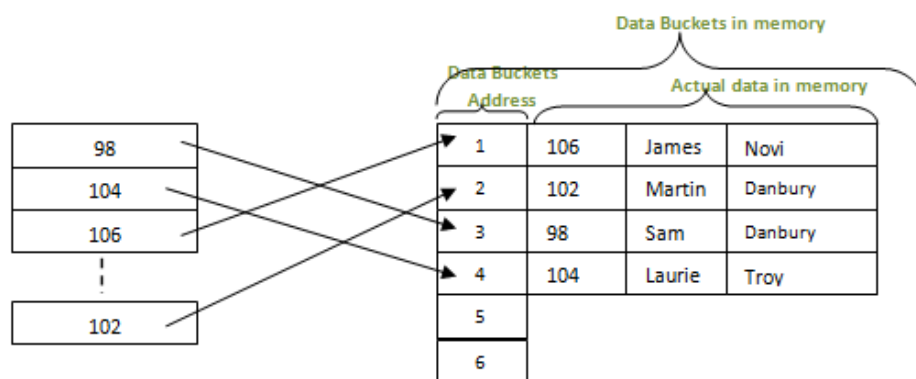
Types of Hashing

There are two types of Hashing:

- 1. Static hashing:** In static hashing, the hash function maps search-key values to a fixed set of locations.
- 2. Dynamic hashing:** In dynamic hashing a hash table can grow to handle more items. The associated hash function must change as the table grows.

Static Hashing

In this method of hashing, the resultant data bucket address will be always same. That means, if we want to generate address for EMP_ID = 103 using mod (5) hash function, it always result in the same bucket address 3. There will not be any changes to the bucket address here. Hence number of data buckets in the memory for this static hashing remains constant throughout. In our example, we will have five data buckets in the memory used to store the data.



Static hashing will be good for smaller databases where record size id previously known. If there is a growth in data, it results in serious problems like bucket overflow.

Note:

- Key-value pairs are stored in a fixed size table called a *hash table*.
 - A hash table is partitioned into many **buckets**.
 - Each bucket has many **slots**.
 - Each slot holds one record.
 - A hash function $f(x)$ transforms the identifier (key) into an address in the hash table
- The **key density (or identifier density)** of a hash table is the ratio n/T
 - n is the number of keys in the table
 - T is the number of distinct possible keys
- The **loading density or loading factor** of a hash table is $a = n/(sb)$
 - s is the number of slots
 - b is the number of buckets

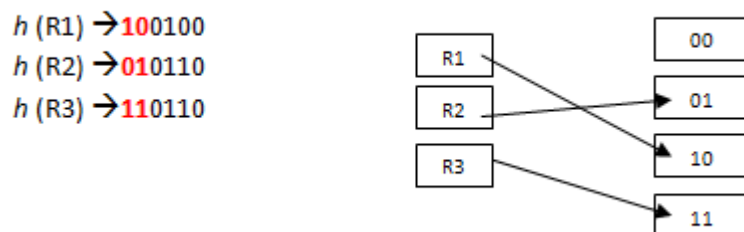
Dynamic Hashing

This hashing method is used to overcome the problems of static hashing – bucket overflow. In this method of hashing, data buckets grows or shrinks as the records increases or decreases. This method of hashing is also known as extendable hashing method. Let us see an example to understand this method.

Consider there are three records R1, R2 and R4 are in the table. These records generate addresses 100100, 010110 and 110110 respectively. This method of storing considers only part of this address – especially only first one bit to store the data. So it tries to load three of them at address 0 and 1.



What will happen to R3 here? There is no bucket space for R3. The bucket has to grow dynamically to accommodate R3. So it changes the address have 2 bits rather than 1 bit, and then it updates the existing data to have 2 bit address. Then it tries to accommodate R3.



Now we can see that address of R1 and R2 are changed to reflect the new address and R3 is also inserted. As the size of the data increases, it tries to insert in the existing buckets. If no buckets are available, the number of bits is increased to consider larger address, and hence increasing the buckets. If we delete any record and if the data can be stored with lesser buckets, it shrinks the bucket size. It does the opposite of what we have seen above. This is how a dynamic hashing works.

Advantages of dynamic hashing

- Performance does not come down as the data grows in the system. It simply increases the memory size to accommodate the data.
- Since it grows and shrinks with the data, memory is well utilized. There will not be any unused memory lying.
- Good for dynamic databases where data grows and shrinks frequently

Disadvantages of dynamic hashing

- As the data size increases, the bucket size is also increased. These addresses will be maintained in bucket address tables. This is because, the address of the data will keep changing as buckets grow and shrink. When there is a huge increase in data, maintaining this bucket address table becomes tedious.
- Bucket overflow situation will occur in this case too. But it might take little time to reach this situation than static hashing.

Differences between static and dynamic hashing

S.N.	Static Hashing	Dynamic Hashing
1.	Numbers of buckets are fixed.	Numbers of Buckets are not fixed.
2.	As the file grows, performance decreases.	Performance does not degrade as the file grows.
3.	Space overhead is more.	Minimum space lies overhead.
4.	Here we do not use Bucket Address table.	Bucket address table is used.
5.	Open hashing and Closed hashing are forms of it.	Extendable hashing and Linear hashing are forms of it.
6.	No complex implementation.	Implementation is complex.
7.	It is a less attractive technique.	It is a highly attractive technique.
8.	Chaining used is overflow chaining.	Overflow chaining is not used

Some Applications of Hash Tables

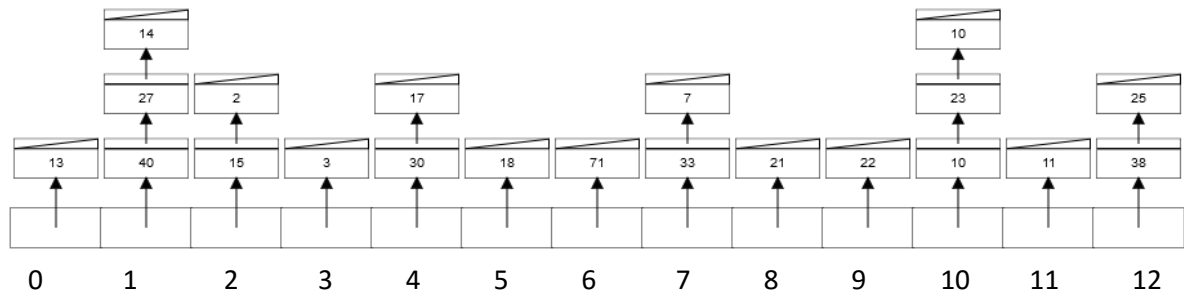
- **Database systems:** Specifically, those that require efficient random access. Generally, database systems try to optimize between two types of access methods: sequential and random. Hash tables are an important part of efficient random access because they provide a way to locate data in a constant amount of time.
- **Symbol tables:** The tables used by compilers to maintain information about symbols from a program. Compilers access information about symbols frequently. Therefore, it is important that symbol tables be implemented very efficiently.
- **Data dictionaries:** Data structures that support adding, deleting, and searching for data. Although the operations of a hash table and a data dictionary are similar, other data structures may be used to implement data dictionaries. Using a hash table is particularly efficient.
- **Network processing algorithms:** Hash tables are fundamental components of several network processing algorithms and applications, including route lookup, packet classification, and network monitoring.
- **Browser Cashes:** Hash tables are used to implement browser caches.

One common method of determining a hash key is the **division method** of hashing. The formula that will be used is:

Hash key = key % number of slots in the table

There are two types of Hash Tables

1. **A Separate-Chained Hash Table (Open Hashing)** is a one-dimensional array of linked lists indexed by integer values that are computed by an index function called a hash function.



2. **An Open-addressed Hash Table (Closed Hashing)** is a one-dimensional array indexed by integer values that are computed by an index function called a hash function

Collision Resolution

When two items hash to the same slot, we must have a systematic method for placing the second item in the hash table. This process is called **collision resolution**. If the hash function is perfect, collisions will never occur. However, since this is often not possible, collision resolution becomes a very important part of hashing.

Linear Probing

When collision occurs, the table is search sequentially for an empty slot. This is accomplished using two values - one as a starting value and one as an interval between successive values in modular arithmetic. The second value, which is the same for all keys and known as the step size, is repeatedly added to the starting value until a free space is found, or the entire table is traversed. The algorithm for this technique is $\text{newLocation} = (\text{startingValue} + \text{step Size}) \% \text{array Size}$

The step size takes the following value: 1, 2, 3, 4,....

Given an ordinary hash function $H(x)$, a linear probing function would be:

$$H(x,i) = (H(x) + i) \bmod (n)$$

The insertion algorithm is as follows:

- use hash function to find index for a record
- If that spot is already use use next available spot in a "higher" index. Treat the hash table as if it is round, if you hit the end of the hash table, go back to the front.

$$89 \% 10 = 9$$

$$18 \% 10 = 8$$

$$49 \% 10 = 9 - 1 \text{ attempt needed} - 1^2 = 1 \text{ spot}$$

$$58 \% 10 = 8 - 3 \text{ attempts} - 3^2 = 9 \text{ spots}$$

$$69 \% 10 = 9 - 2 \text{ attempts} - 2^2 = 4 \text{ spots}$$

[0]	49
[1]	
[2]	
[3]	69
[4]	
[5]	
[6]	
[7]	58
[8]	18
[9]	89

Quadratic Probing

A problem with the linear probe method is that it is possible for blocks of data to form when collisions are resolved. This is known as **primary clustering**.

To resolve the primary clustering problem, **quadratic probing** can be used.

Quadratic probing operates by taking the original hash value and adding successive values of an arbitrary quadratic polynomial to the starting value. The idea here is to skip regions in the table with possible clusters. It uses the hash function of the form:

$$H(k, i) = (h(k) + i^2) \bmod (m) \text{ for } i = 0, 1, 2, \dots, m-1$$

Quadratic Probing is similar to linear probing. The difference is that if you were to try to insert into a space that is filled you would first check $1^2=1$ element away then $2^2 = 4$ elements away, then $3^2 = 9$ elements away then $4^2=16$ elements away and so on.

[0]	72	Add the keys 10, 5, and 15 to the previous table .	[0]	72
[1]			[1]	15
[2]	18	Hash key = key % table size	[2]	18
[3]	43	2 = 10 % 8	[3]	43
[4]	36	5 = 5 % 8	[4]	36
[5]		7 = 15 % 8	[5]	10
[6]	6		[6]	6
[7]			[7]	5

Double Hashing

Double hashing uses the idea of applying a second hash function to the key when a collision occurs. The result of the second hash function will be the numbers of positions from the point of collision to insert.

There are a couple of requirements for the second function:

- It must never evaluate to 0
- Must make sure that all cells can be probed

A popular second hash function is: $\text{Hash}_2(\text{key}) = R - (\text{key} \% R)$ where R is a prime number that is smaller than the size of the table.

The probing sequence is then computed as follows

$$H_i(x) = (h(x) + ih_2(x)) \bmod m$$

Where $h(x)$ is the original function, $h_2(x)$ the second function, i the number of collisions and m the table size.

Table Size = 10 elements	[0]	69
$\text{Hash}_1(\text{key}) = \text{key} \% 10$	[1]	
$\text{Hash}_2(\text{key}) = 7 - (\text{k} \% 7)$	[2]	
Insert keys: 89, 18, 49, 58, 69	[3]	58
$\text{Hash}(89) = 89 \% 10 = 9$	[4]	
$\text{Hash}(18) = 18 \% 10 = 8$	[5]	
$\text{Hash}(49) = 49 \% 10 = 9$ a collision !	[6]	49
= $7 - (49 \% 7)$	[7]	
= 7 positions from [9]	[8]	18
$\text{Hash}(58) = 58 \% 10 = 8$	[9]	89
= $7 - (58 \% 7)$		
= 5 positions from [8]		
$\text{Hash}(69) = 69 \% 10 = 9$		
= $7 - (69 \% 7)$		
= 1 position from [9]		

Rehashing

Once the hash table gets too full, the running time for operations will start to take too long and may fail. To solve this problem, a table at least twice the size of the original will be built and the elements will be transferred to the new table.

The new size of the hash table:

- should also be prime
- will be used to calculate the new insertion spot (hence the name **rehashing**)

This method is applicable:

- Once the table becomes half full
- Once an insertion fails
- Once a specific load factor has been reached, where load factor is the ratio of the number of elements in the hash table to the table size

Extendible Hashing

A new type of dynamic file access called dynamic hashing has recently emerged. It promises the flexibility of handling dynamic files while preserving the fast access times expected from hashing. Such a fast, dynamic file access scheme is needed to support modern database systems.

Some hashing techniques allow the hash function to be modified dynamically to accommodate the growth or shrinking of the database. These are called **dynamic hash functions**.

- **Extendable hashing** is one form of dynamic hashing.
- Extendable hashing splits and combines buckets as database size changes.
- This imposes some performance overhead, but space efficiency is maintained.
- As reorganization is on one bucket at a time, overhead is acceptably low.

➤ Advantages

- Extendable hashing provides performance that does not degrade as the file grows.
- Minimal space overhead - no buckets need be reserved for future use. Bucket address table only contains one pointer for each hash value of current prefix length.

➤ Disadvantages

- Extra level of indirection in the bucket address table
- Added complexity

Priority Queues

Introduction

A **priority queue** is a collection of elements such that each element has been assigned a priority and the order in which element are deleted and processed comes from the following rules:

- An element of higher priority is processed before any element of lower priority.

- Two elements with the same priority are processed according to the order in which they were added to the queue.

A prototype of priority is processed first, and programs with the same priority form a standard queue.

There can be two types of implementation of priority queue:

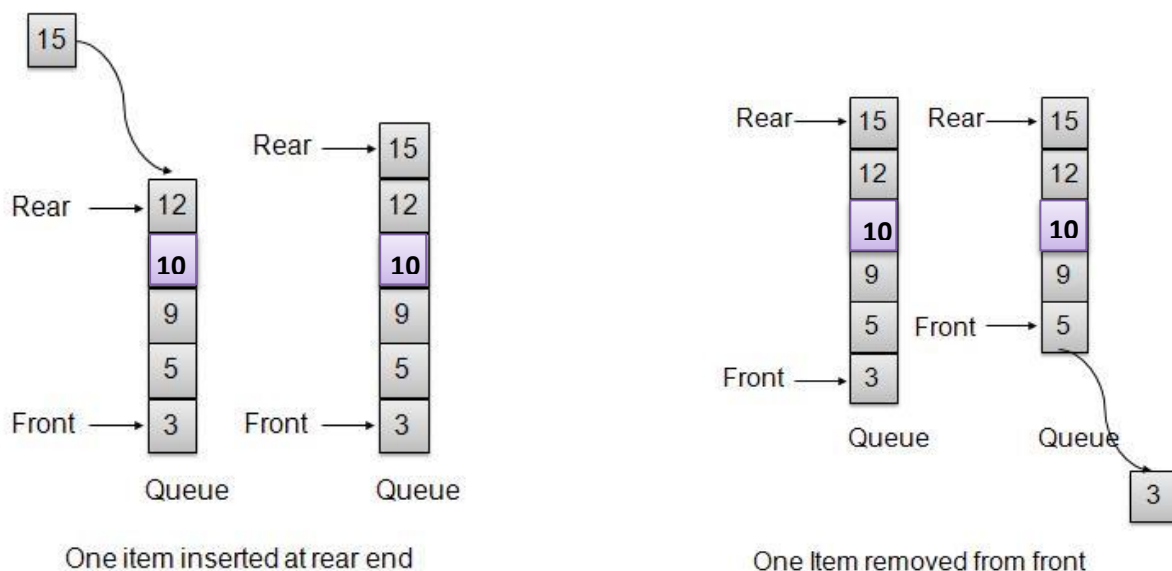
- Ascending Priority Queue
- Descending Priority Queue

A collection of items into which items can be inserted arbitrarily and from which only the smallest item can be removed is called **Ascending Priority Queue**.

In **Descending Priority Queue** only the largest item is deleted. The elements of priority Queue need not be numbers or characters that can be composed directly. They may be complex structures that are ordered by one or several fields. Sometimes the field on which the element of a priority queue is ordered is not even part of the elements themselves.

The priority queue is a data structure in which intrinsic ordering of the elements determines the result of its basic operations.

Insertion/Enqueue: The insertion in priority queues is the same as in non-priority queues. Whenever an element is inserted into queue, priority queue inserts the item according to its order. Here we're assuming that data with high value has low priority.



Deletion/Dequeue: Deletion requires a search for the element of highest priority and deletes the element with highest priority. The following methods can be used for deletion/removal from a given priority queue:

- An empty indicator replaces deleted elements.
- After each deletion, elements can be moved up in the array decrementing the rear.
- The array in the queue can be maintained as an ordered circular array 3.

A priority queue is different from a "normal" queue, because instead of being a "**first-in-first-out**" data structure, values come out in order by **priority**. A priority queue might be used, for example, to handle the jobs sent to the Computer Science Department's printer: Jobs sent by the department chair should be printed first, then jobs sent by professors, then those sent by graduate students, and finally those sent by undergraduates.

A priority queue can be implemented using many of the data structures that we've already studied (an array, a linked list, or a binary search tree). However, those data structures do not provide the most efficient operations. To make all of the operations very efficient, we'll use a new data structure called a **heap**.

Simple Implementations

Using Array: If we use an array to store the values in the priority queue, we can either store the values in sorted order (which will make the insert operation slow, and the removeMax operation fast), or in arbitrary order (which will make the insert operation fast, and the removeMax operation slow).

Using Linked List: Using a linked list is similar to using an array; again, we can keep the list sorted (which makes insert $O(N)$) or unsorted (which makes removeMax $O(N)$). Note that if the list is sorted we must use linear search to find the place to insert the new value.

Using Binary Search Tree: If we use a BST then the largest value can be found in time proportional to the height of the tree by going right until we reach a node with no right child. Removing that node is also $O(\text{tree height})$, as is inserting a new value.

Heap (Binary heap)

A **heap** is a binary tree (in which each node contains a Comparable key value), with two special properties:

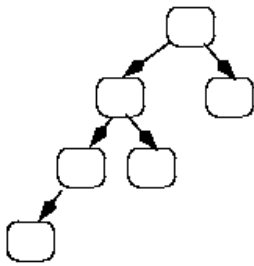
The **ORDER** property:

For every node n , the value in n is **greater than or equal to** the values in its children (and thus is also greater than or equal to all of the values in its subtrees).

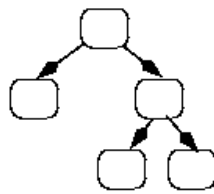
The **SHAPE** property:

1. All leaves are either at depth d or $d-1$ (for some value d).
2. All of the leaves at depth $d-1$ are to the **right** of the leaves at depth d .
3. (a) There is at most 1 node with just 1 child. (b) That child is the **left** child of its parent, and (c) it is the **rightmost** leaf at depth d .

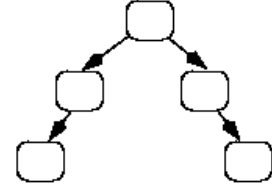
Here are some binary trees, some of which violate the shape properties, and some of which respect those properties:



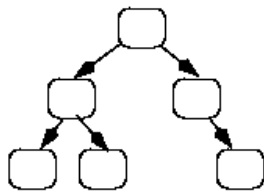
NO: violates shape property 1



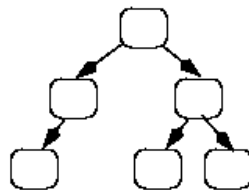
NO: violates shape property 2



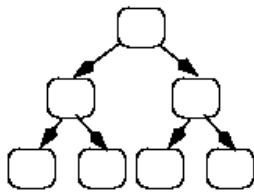
NO: violates shape property 3(a)



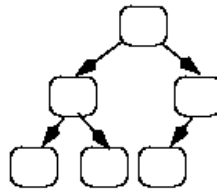
NO: violates shape property 3(b)



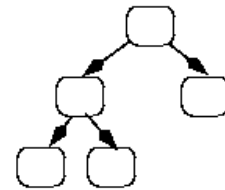
NO: violates shape property 3(c)



YES!

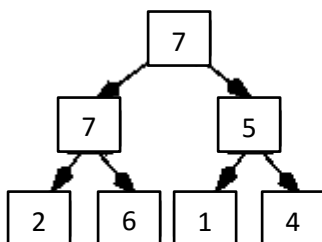


YES!

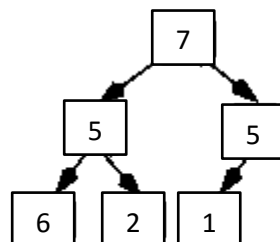


YES!

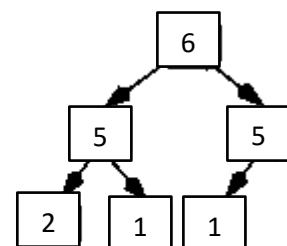
And here are some more trees; they all have the **shape** property, but some violate the order property:



YES!



NO (6 is > 5)



YES!

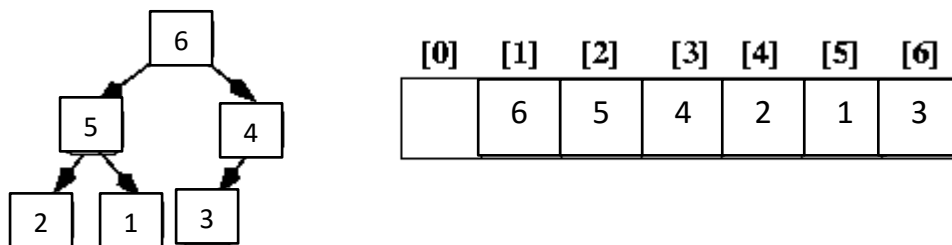
- In a heap the highest (or lowest) priority element is always stored at the root, hence the name "heap".
- Since a heap is a complete binary tree, it has a smallest possible height - a heap with N nodes always has $O(\log N)$ height.
- A heap is useful data structure when we need to remove the object with the highest (or lowest) priority. A common use of a heap is to implement a priority queue.
- All nodes are either greater than equal to (**Max-Heap**) or less than equal to (**Min-Heap**) to each of its child nodes. This is called **heap property**.

Implementing Priority Queues Using Heaps

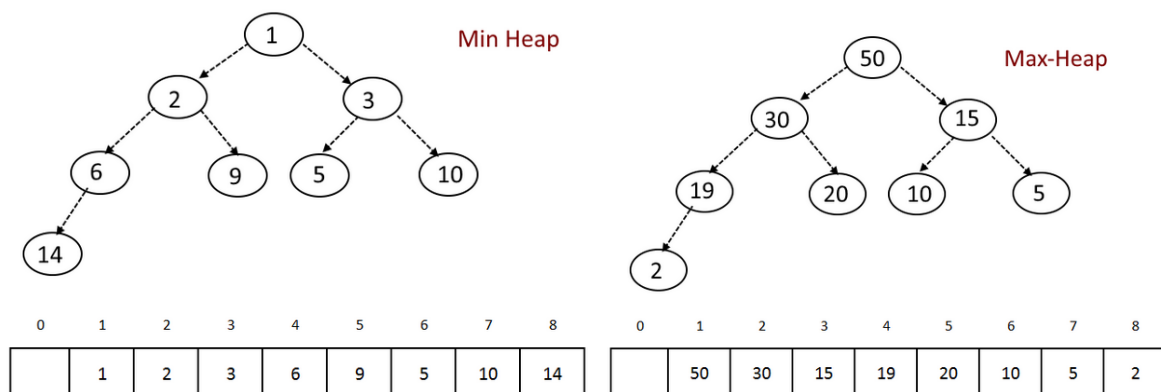
Now let's consider how to implement priority queues using a heap. The standard approach is to use an **array** (or an ArrayList), starting at position 1 (instead of 0), where each item in the array corresponds to one node in the heap:

- The root of the heap is always in array[1].
- Its **left** child is in array[2].
- Its **right** child is in array[3].
- In general, if a node is in array[k], then its left child is in array[k*2], and its right child is in array[k*2 + 1].
- If a node is in array[k], then its **parent** is in array[k/2] (using integer division, so that if k is odd, then the result is truncated; e.g., 3/2 = 1).

Here's an example, showing both the conceptual heap (the binary tree), and its array representation:



Note that the heap's "shape" property guarantees that there are never any "holes" in the array.



Basic Operation on Heap

Heap majorly has 3 operations -

- Insert Operation
- Delete Operation
- Extract-Min (OR Extract-Max)

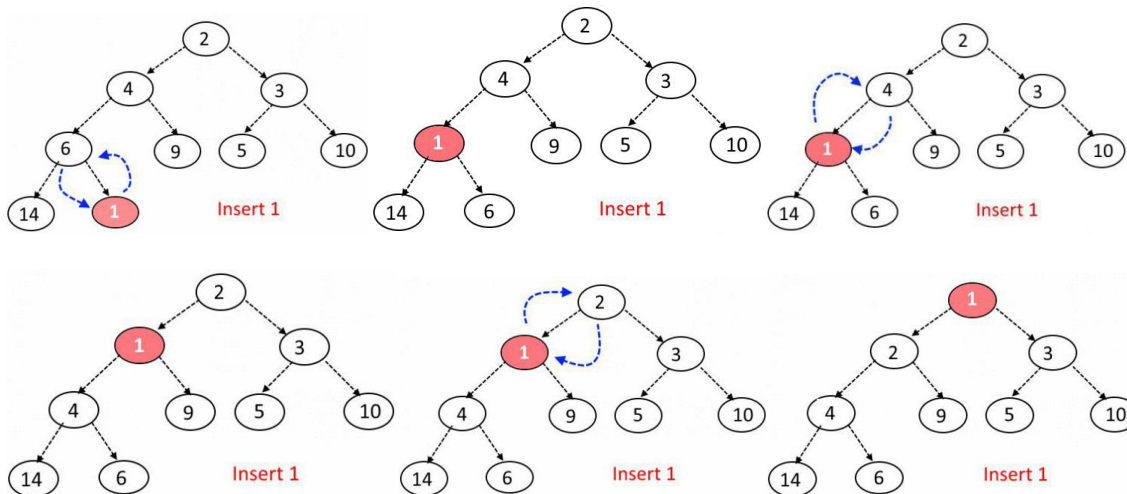
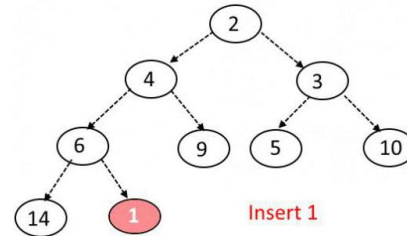
Insert Operation

- Add the element at the bottom leaf of the Heap.
- Perform the Bubble-Up operation.

- All Insert Operations must perform the **bubble-up** operation (it is also called as **up-heap**, **percolate-up**, **sift-up**, **trickle-up**, **heapify-up**, or **cascade-up**)

Bubble up Operation

- If inserted element is smaller than its parent node in case of Min-Heap OR greater than its parent node in case of Max-Heap, swap the element with its parent.
- Keep repeating the above step, if node reaches its correct position, STOP.

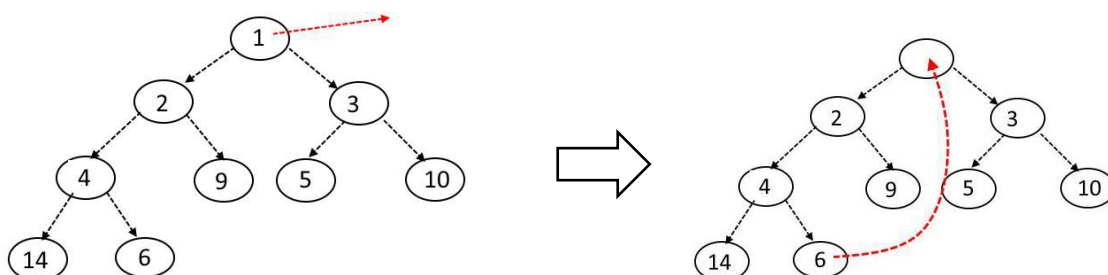


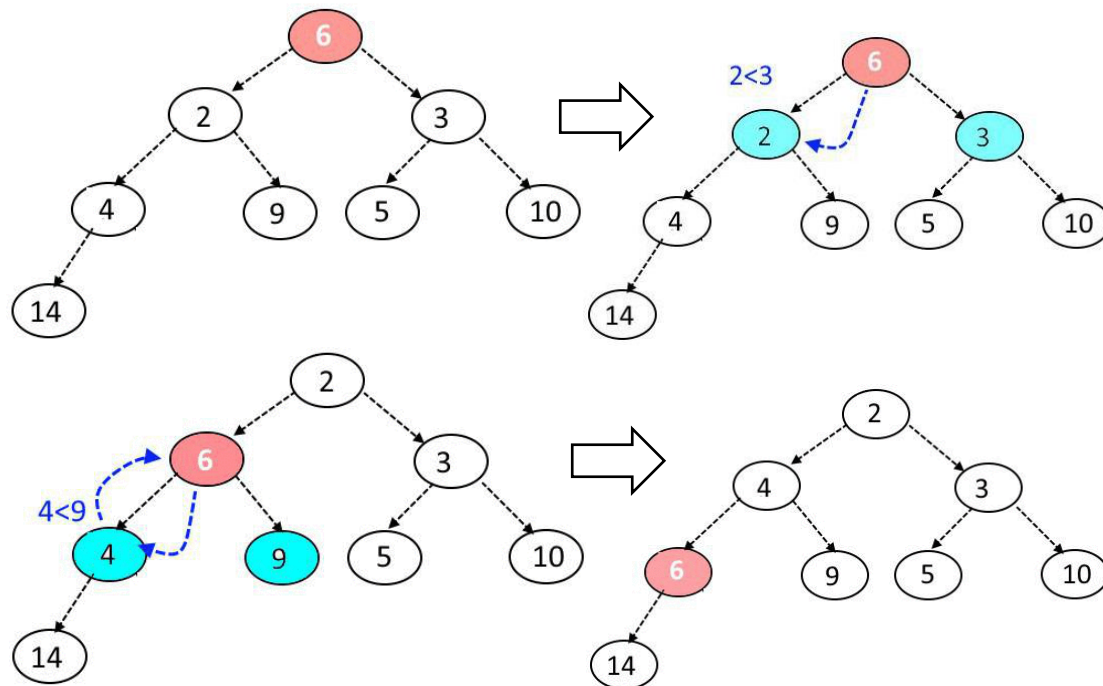
Extract-Min OR Extract-Max Operation:

- Take out the element from the root. (It will be minimum in case of Min-Heap and maximum in case of Max-Heap).
- Take out the last element from the last level from the heap and replace the root with the element.
- Perform **Sink-Down** operation.
- All delete operation must perform Sink-Down Operation (also known as bubble-down, percolate-down, sift-down, trickle down, heapify-down, and cascade-down).

Sink-Down Operation:

- If replaced element is greater than any of its child node in case of Min-Heap OR smaller than any if its child node in case of Max-Heap, swap the element with its smallest child (Min-Heap) or with its greatest child (Max-Heap).
- Keep repeating the above step, if node reaches its correct position, STOP.





Delete Operation

- Find the index for the element to be deleted.
- Take out the last element from the last level from the heap and replace the index with this element.
- Perform *Sink-Down*

Application of Priority Queues

1. **Operating systems.** One place where a priority queue is often needed is in the process manager of an operating system.
2. **Time-activated events.** We may have a collection of events that we are scheduled to begin at varying times. We can use a priority queue to prepare for the event slated to begin next.
3. **Managing responsibilities.** We could use a priority queue to inform you as to which task you should concern yourself with next.
4. **Sorting.** We can sort with a priority queue. This is the idea underlying the Heap sort algorithm.
5. **Graph Algorithm:** Find Shortest path.
6. **Event simulation:** Instead of checking for events at each time click, look up next event to happen.

D-heaps

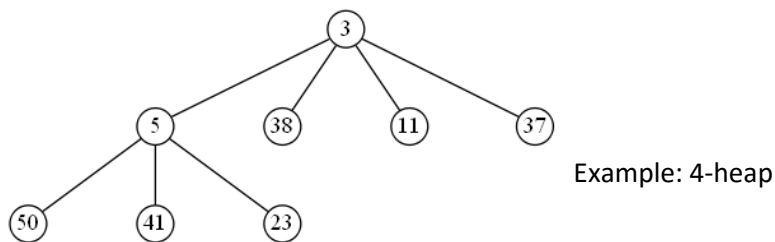
The d-ary heap or d-heap is a priority queue data structure, a generalization of the binary heap in which the nodes have d children instead of 2. Thus, a binary heap is a 2-heap, and a ternary heap is a 3-heap.

A d -ary heap is like a binary heap, but (with one possible exception) non-leaf nodes have d children instead of 2 children. The d -ary heap consists of an array of n items, each of which has a priority

associated with it. D-heaps are easy to implement but two heaps can't be easily merged. It was invented by Johnson in 1975.

The rules for adding items to and removing items from a d -heap are exactly the same as for adding and removing in a binary heap.

- To remove an item, replace the root node with the last node in the heap and sift down.
- To add an item, add the item to the end of the heap and bubble up.



The number of items in a full d -heap of n levels is $(1-d^n) / (1-d)$. A seven level full binary heap contains 127 items. A seven-level full 4-heap would contain 5,461 $(16383/3)$ items.

The primary implementation difference between a binary heap and a d -heap is in the code in the *SiftDown* method that finds the smallest child. The binary heap has only two nodes to check, so that can be done directly. In a d -heap, you have to write a loop that checks the d child nodes.

Leftist Heaps

Leftist tree always keeps the left branches of all roots being the longer and in worst case; they are as long as the right branches. In other word, all right branches of all roots are shortest. Leftist heaps are one of the possible implementations of priority queues. When we decide which branch to go, if we know one branch is always shorter than the other, then we should take it. This is because shorter branch means potentially less nodes and the number of future possible operations intends to be smaller.

In order to maintain this property, each node has a **rank**, which indicates **the length of the path between the node and the right most leaf**. For example,

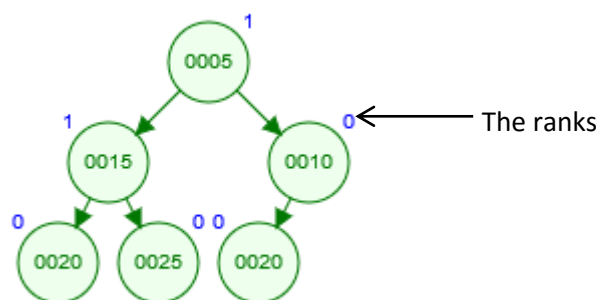


Fig: The leftist heap

When inserting a new node into a tree, a new one-node tree is created and merged into the existing tree. To delete a minimum item, we remove the root and the left and right sub-trees are then merged. Both these operations take $O(\log n)$ time. For insertions, this is slower than binomial heaps.

Leftist Property:

For all node x : $\text{rank}[\text{right}[x]] \leq \text{rank}[\text{left}[x]]$

We can see that by always putting the higher rank to the left can make the right branch being shortest all the time. Actually, this strategy is how this design of tree based heap got the name *leftist*, i.e., more nodes intend to be on the left side.

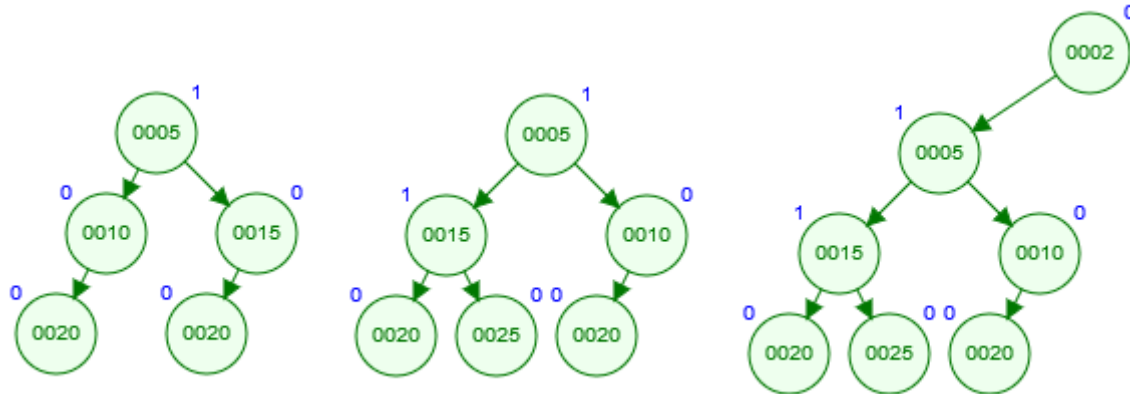


Fig: Initial leftist heap

Fig: After the insertion of 25

Fig: After the insertion of 2

A **leftist Heap** is a binary tree with properties

1. $\text{key}(i) \geq \text{key}(\text{parent}(i))$
2. The root contains the minimum key. As with array heaps, we could have the maximum at the top, simply by changing this property and modifying the code accordingly.
3. $\text{rank}(\text{right}(i)) \leq \text{rank}(\text{left}(i))$
4. The shortest path to a descendant external node is through the right child.

Skew Heaps

Skew heaps are one of the possible implementations of priority queues. A skew heap is a self-adjusting form of a leftist heap, which may grow arbitrarily unbalanced because they do not maintain balancing information.

The only difference between a skew heap and a leftist heap is the union operation is treated differently in skew heaps. The swapping of the children of a visited node on the right path is performed unconditionally; the rank value is not maintained.

The purpose of the swapping is to keep the length of the right path bounded, even though the length of the right path can grow to $\Omega(n)$, it is quite effective. The reasoning behind this is that insertions are made on the right side and therefore creating a “heavy” right side. Then by swapping everything unconditionally a relatively “light” right side is created. So, the good behavior of skew heaps is due to always inserting to the right and unconditionally swapping all nodes.

The self-adjustment in skew heaps has important advantages over the balancing in leftist heaps:

- Reduction of memory consumption (no balancing information needs to be stored)
- Reduction of time consumption (saving the updates of the rank information)
- Unconditional execution (saving one test for every processed node)

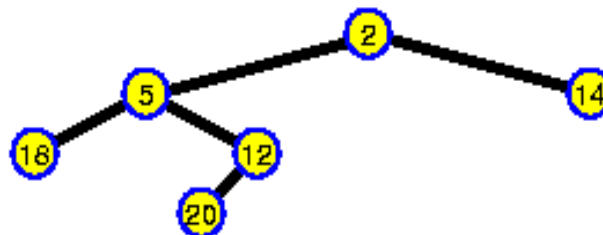
The relation between **skew heaps** and leftist heaps is the same as that between splay trees and AVL trees: a skew heap is a self-adjusting variant of a leftist heap, which may grow arbitrarily unbalanced,

but for which we nevertheless can prove $O(\log n)$ amortized time for the operations. This is achieved without maintaining balancing information.

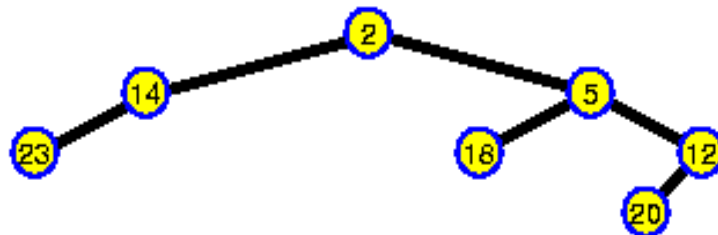
A node is said to be heavy, if the size of its right subtree is larger (not equal) than the size of its left subtree. A node which is not heavy is said to be **light**.

SKEW-HEAP OPERATIONS

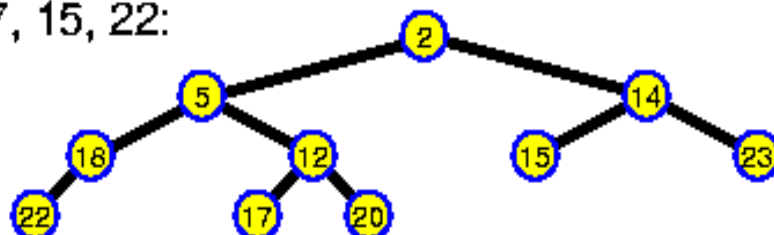
After inserting 20, 12, 5, 2, 14, 18:



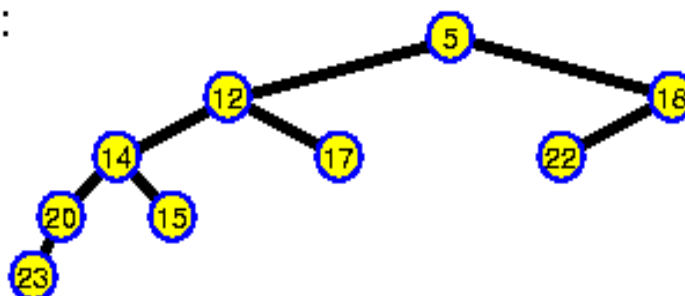
After inserting 23:



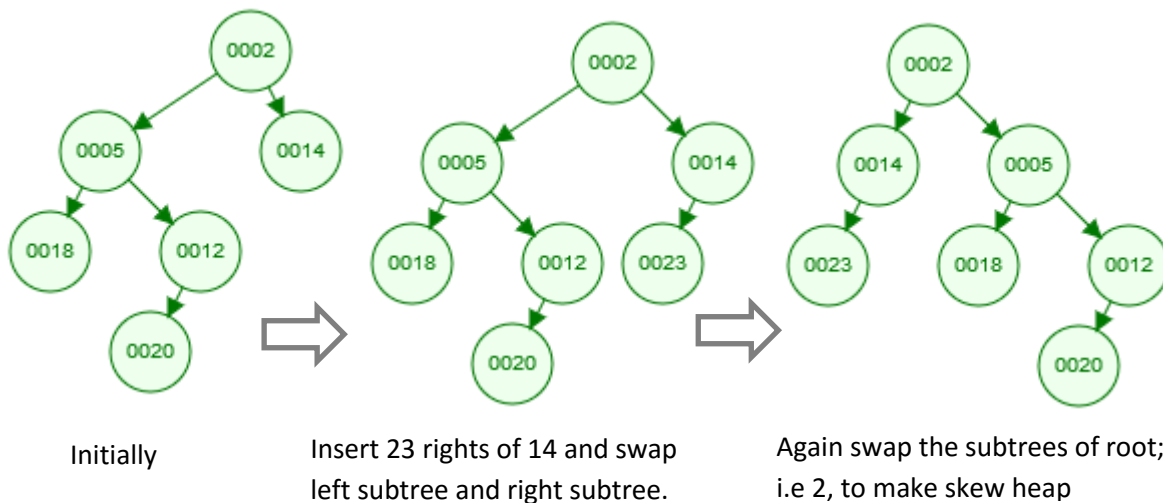
After inserting 17, 15, 22:



After deletemin:



Steps while inserting 23 are shown below:



Binomial Queues

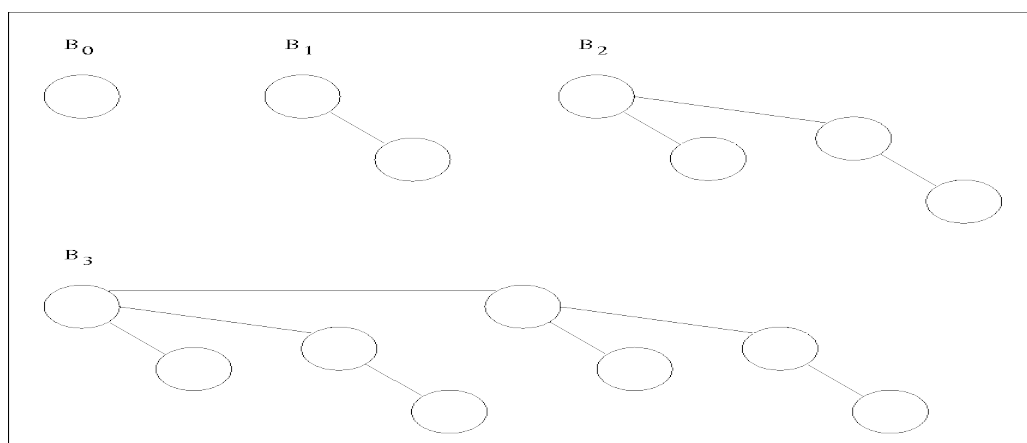
A binomial queue is a priority queue that is implemented not as a single tree but as a collection of heap-ordered trees. A collection of trees is called a *forest*. Each of the trees in a binomial queue has a very special shape called a binomial tree. Binomial trees are general trees. I.e., the maximum degree of a node is not fixed.

A binomial queue of N elements has one power-of-2 heap for each 1 bit in the binary representation of N . For example, a binomial queue of 13 nodes comprises an 8-heap, a 4-heap, and a 1-heap.

A binomial queue is a set of power-of-2 heaps, no two of the same size. The structure of a binomial queue is determined by that queue's number of nodes, by correspondence with the binary representation of integers.

Binomial tree:

- Support merge, insert, and delete min operations in $O(\log n)$ worst-case time.
- A binomial queue is a forest of heap-ordered trees.
- Each of the heap-ordered trees is of a constrained form known as a *binomial tree*.
- A *binomial tree* of height 0 is a one-node tree; a binomial tree B_k of height k is formed by attaching a binomial tree B_{k-1} to the root of another binomial tree, B_{k-1} .
- A binomial tree B_k consists of a root with children B_0, B_1, \dots, B_{k-1} .
- B_k has exactly 2^k nodes.



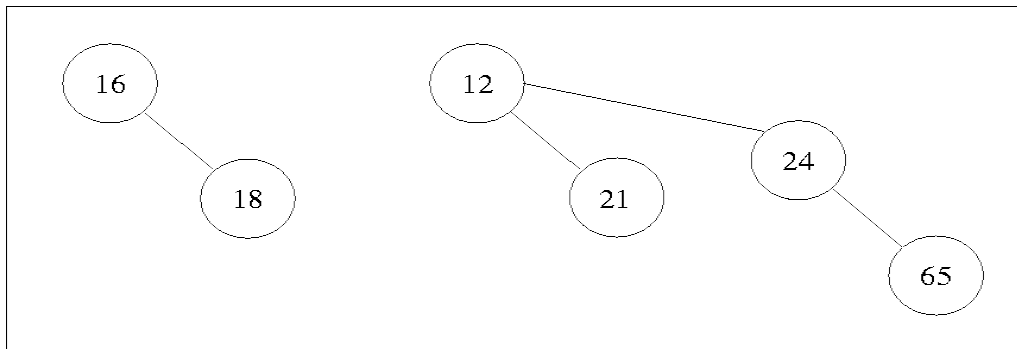
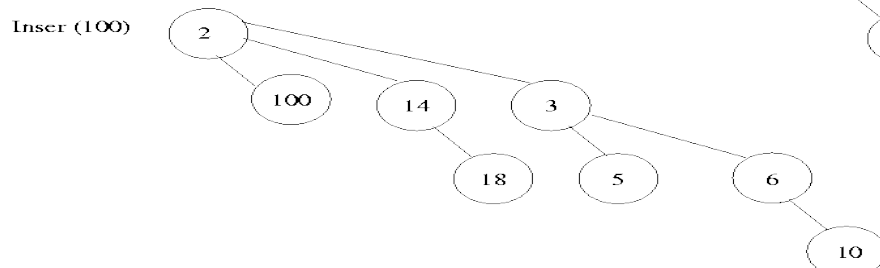
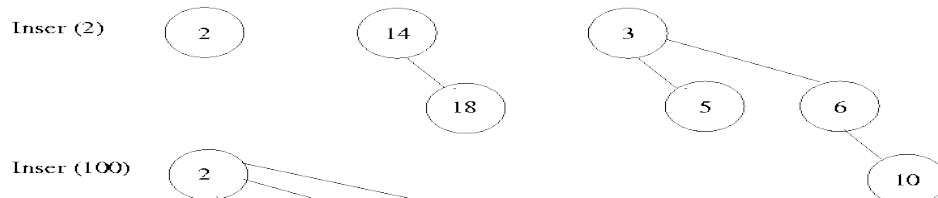
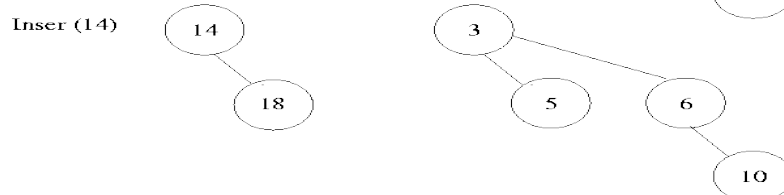
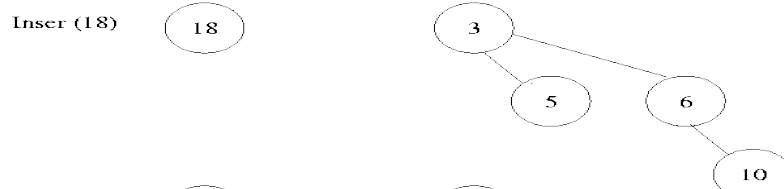
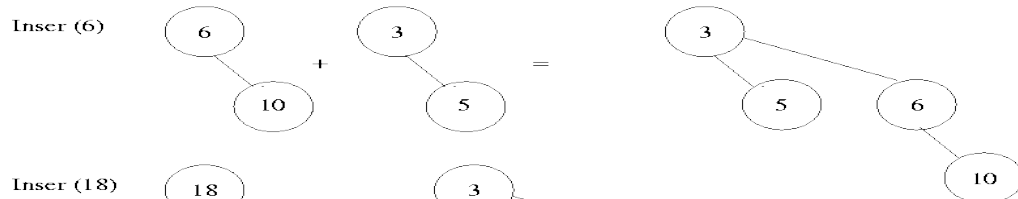
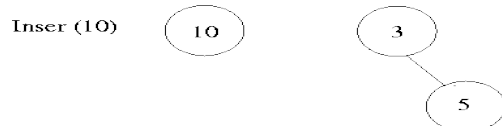


Fig: A Binomial Queue H1 with six elements.

Inser (5)  An example of Binomial Queue



Sorting

In general sorting means rearrangement of data according to a defined pattern. The task of a sorting algorithm is to transform the original unsorted sequence to the sorted sequence. We can distinguish two types of sorting. If the number of objects is small enough to fit into the main memory, sorting is called **internal sorting**. If the number of objects is so large that some of them reside on external storage during the sort, it is called **external sorting**.

Sorting is nothing but storage of data in sorted order; it can be in ascending or descending order. The term Sorting comes into picture with the term Searching. There are so many things in our real life that we need to search, like a particular record in database, roll numbers in merit list, a particular telephone number, any particular page in a book etc.

Sorting arranges data in a sequence which makes searching easier. Every record which is going to be sorted will contain one key. Based on the key the record will be sorted.

If a sorting algorithm, after sorting the contents, does not change the sequence of similar content in which they appear, it is called **stable sorting**.

If a sorting algorithm, after sorting the contents, changes the sequence of similar content in which they appear, it is called **unstable sorting**.

A sorting algorithm is said to be **adaptive**, if it takes advantage of already 'sorted' elements in the list that is to be sorted. That is, while sorting if the source list has some element already sorted, adaptive algorithms will take this into account and will try not to re-order them.

A **non-adaptive algorithm** is one which does not take into account the elements which are already sorted. They try to force every single element to be re-ordered to confirm their sortedness.

Sorting Efficiency

There are many techniques for sorting. Implementation of particular sorting technique depends upon situation. Sorting techniques mainly depend on two parameters. First parameter is the execution time of program, which means time taken for execution of program. Second is the space which means memory that is taken by the program. Like searching, the efficiency of a sorting algorithm is related to the number of items being processed.

First, it will be necessary to compare two values to see which is smaller (or larger). In order to sort a collection, it will be necessary to have some systematic way to compare values to see if they are out of order. The total number of comparisons will be the most common way to measure a sort procedure. Second, when values are not in the correct position with respect to one another, it may be necessary to exchange them. This exchange is a costly operation and the total number of exchanges will also be important for evaluating the overall efficiency of the algorithm.

There are many types of Sorting techniques, differentiated by their efficiency and space requirements. Following are some sorting techniques which we will cover in this document.

- | | |
|-------------------|----------------|
| 1. Insertion Sort | 4. Merge Sort |
| 2. Shell Sort | 5. Quick Sort |
| 3. Heap Sort | 6. Bucket sort |

Insertion Sort

It is a simple Sorting algorithm which sorts the array by shifting elements one by one. Following are some of the important characteristics of Insertion Sort.

1. It has one of the simplest implementation
2. It is efficient for smaller data sets, but very inefficient for larger lists.
3. Insertion Sort is adaptive, that means it reduces its total number of steps if given a partially sorted list, hence it increases its efficiency.
4. It is better than Selection Sort and Bubble Sort algorithms.
5. Its space complexity is less, like Bubble Sorting; insertion sort also requires a single additional memory space.
6. It is **Stable**, as it does not change the relative order of elements with equal keys
7. Worst Case Time Complexity: $O(n^2)$, Best Case Time Complexity : $O(n)$, Average Time Complexity : $O(n^2)$, Space Complexity : $O(1)$

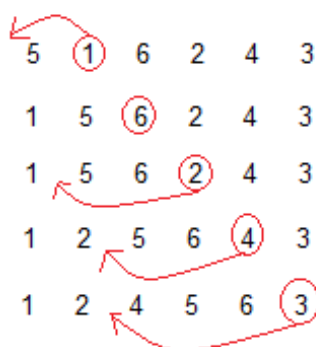
Algorithm

- Step 1 – If it is the first element, it is already sorted. Return 1;
- Step 1 – Pick next element
- Step 1 – Compare with all elements in the sorted sub-list
- Step 1 – Find appropriate position
- Step 1 – Insert the value
- Step 1 – Repeat until list is sorted

How insertion sorting works?

5	1	6	2	4	3
---	---	---	---	---	---

Lets take this Array.



(Always we start with the second element as key.)

As we can see here, in insertion sort, we pick up a key, and compares it with elements ahead of it, and puts the key in the right place

5 has nothing before it.

1 is compared to 5 and is inserted before 5.

6 is greater than 5 and 1.

2 is smaller than 6 and 5, but greater than 1, so its is inserted after 1.

And this goes on...

A Lower Bound for simple sorting Algorithms

A lower bound is a mathematical argument saying you can't hope to go faster than a certain amount. More precisely, every algorithm within a certain model of computation has a running time at least that amount. This doesn't necessarily mean faster algorithms are completely impossible, but only that if you want to go faster, you can't stick with the abstract model, you have to look more carefully at the problem.

Lower bounds are useful for two reasons: First, they give you some idea of how good an algorithm you could expect to find (so you know if there is room for further optimization). Second, if your lower bound is slower than the amount of time you want to actually spend solving a problem, the lower bound tells you that you'll have to break the assumptions of the model of computation somehow. Quick sort is an example of randomization and average case analysis. Bucket sort shows how abstraction is not always a good idea.

The number of yes/no decisions (= the number of bits) necessary to distinguish between the $n!$ Permutations is $\log_2(n!)$; it is a lower bound for the complexity of any sorting algorithm. This lower bound is known as the information theoretic lower bound.

This means that $\Omega(n \log(n))$ is a lower bound for the time complexity of any sorting algorithm that is based on comparisons. The sorting algorithms Heap sort and Merge sort have an upper bound of $O(n \log(n))$ steps. Therefore, they are optimal since they attain the lower bound.

The Shell Sort

The **shell sort**, sometimes called the “diminishing increment sort,” improves on the insertion sort by breaking the original list into a number of smaller sublists, each of which is sorted using an insertion sort. It is a complex and fast sorting algorithm.

The basic idea of Shell Sort algorithm can be described as these steps:

1. Set a step size h that is smaller than the number of elements to be sorted and greater than 1.
2. Group the entire collection of data elements in h groups by putting elements that are h steps away from each other into the same group.
3. Sort each group by exchanging locations of elements in the same group.
4. Repeat step 2 and 3 with a smaller step size h until h becomes 1.

This idea of sorting is based on the following facts:

- Sorting h groups of n/h elements in each group takes much less time than sorting n elements.
- If a collection is sorted with a step size h , it is called partially sorted, because any elements will be at most $n-h$ steps away from its final sorted position.
- If a collection is sorted with a step size h , the insertion sort will be much more efficient than the original collection.
- Insertion sort method should be used for each sorting step.

Algorithm

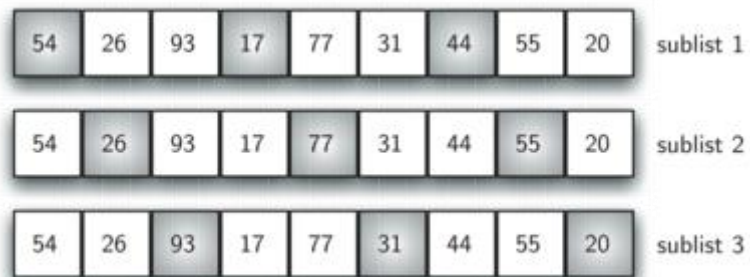
Step 1 – Initialize the value of h

Step 2 – Divide the list into smaller sub-list of equal interval h

Step 3 – Sort these sub-lists using **insertion sort**

Step 3 – Repeat until complete list is sorted

Example:



Sorting each sublist and reassembling the dataset; we get,

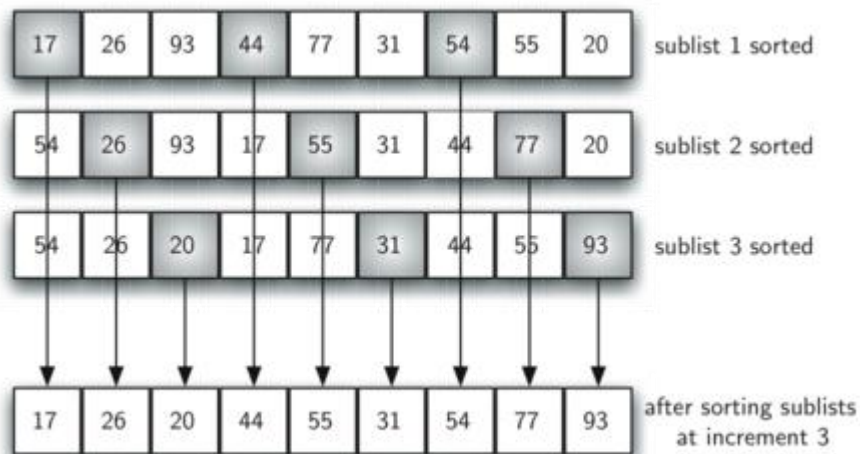


Fig: A shell sort after sorting each sublist.

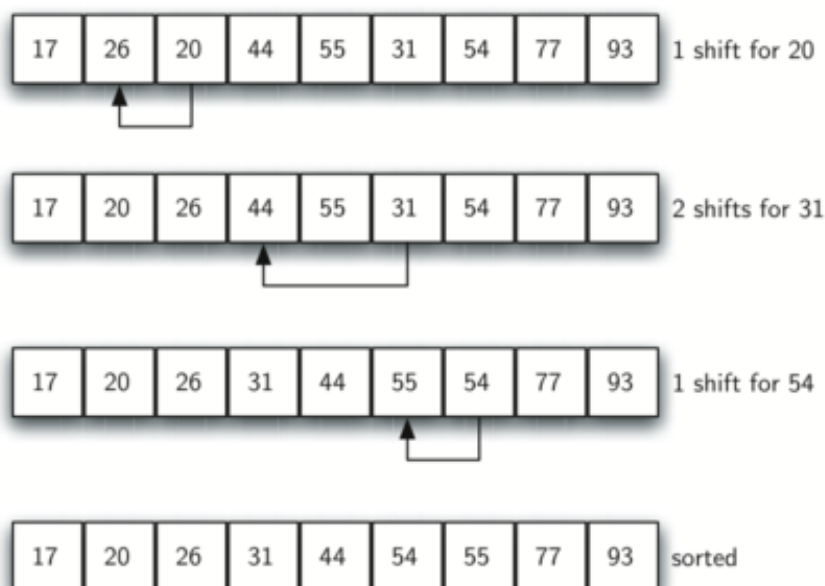


Fig: A final insertion sort with increment

Heap Sort

Heap Sort is one of the best sorting methods being in-place and with no quadratic worst-case scenarios. Heap sort algorithm is divided into two basic parts:

- Creating a Heap of the unsorted list.
- Then a sorted array is created by repeatedly removing the largest/smallest element from the heap, and inserting it into the array. The heap is reconstructed after each removal.

Initially on receiving an unsorted list, the first step in heap sort is to create a Heap data structure (Max-Heap or Min-Heap). Once heap is built, the first element of the Heap is either largest or smallest (depending upon Max-Heap or Min-Heap), so we put the first element of the heap in our array. Then we again make heap using the remaining elements, to again pick the first element of the heap and put it into the array. We keep on doing the same repeatedly until we have the complete sorted list in our array.

- Worst Case Time Complexity : $O(n \log n)$, Best Case Time Complexity : $O(n \log n)$, Average Time Complexity : $O(n \log n)$, Space Complexity : $O(1)$
- Heap sort is not a Stable sort, and requires a constant space for sorting a list.
- Heap Sort is very fast and is widely used for sorting.

Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element.

Heap Sort Algorithm for sorting in increasing order:

1. Build a max heap from the input data.
2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.
3. Repeat above steps until size of heap is greater than 1.

Merge Sort

Merge Sort follows the rule of **Divide and Conquer**. But it doesn't divide the list into two halves. In merge sort the unsorted list is divided into N sublists, each having one element, because a list of one element is considered sorted. Then, it repeatedly merges these sublists, to produce new sorted sublists, and at last one sorted list is produced.

Merge Sort is quite fast, and has a time complexity of **$O(n \log n)$** . It is also a stable sort, which means the "equal" elements are ordered in the same order in the sorted list. Merge sort is a recursive algorithm that continually splits a list in half. If the list is empty or has one item, it is sorted by definition.

Once the two halves are sorted, the fundamental operation, called a **merge**, is performed. Merging is the process of taking two smaller sorted lists and combining them together into a single, sorted, new list.

Algorithm

Step 1 – if it is only one element in the list it is already sorted, return.

Step 2 – divide the list recursively into two halves until it can no more be divided.

Step 3 – merge the smaller lists into new list in sorted order.

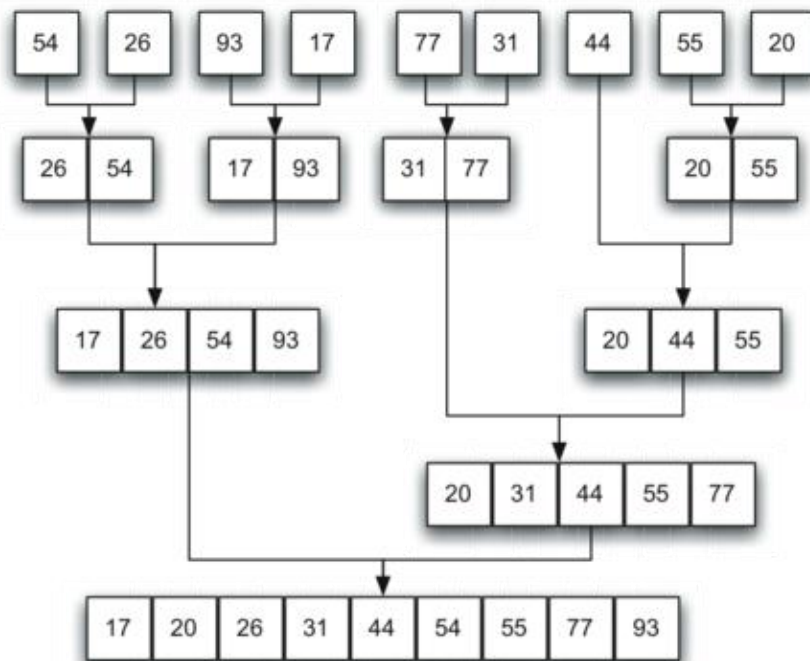


Fig: An example of merge sort.

Quick Sort

Quick Sort, as the name suggests, sorts any list very quickly. Quick sort is not stable search, but it is very fast and requires very less additional space. It is based on the rule of **Divide and Conquer** (also called *partition-exchange sort*). This algorithm divides the list into three main parts:

1. Elements less than the Pivot element
2. Pivot element
3. Elements greater than the pivot element

Quicksort, also known as *partition-exchange sort*, uses **these steps**.

- 1) Choose any element of the array to be the pivot.
- 2) Divide all other elements (except the pivot) into two partitions.
 - All elements less than the pivot must be in the first partition.
 - All elements greater than the pivot must be in the second partition.
- 3) Use recursion to sort both partitions.
- 4) Join the first sorted partition, the pivot, and the second sorted partition.

The best pivot creates partitions of equal length (or lengths differing by 1). The worst pivot creates an empty partition (for example, if the pivot is the first or last element of a sorted array). The runtime of Quicksort ranges from $O(n \log n)$ with the best pivots, to $O(n^2)$ with the worst pivots, where n is the number of elements in the array.

Quick sort is in-place sorting algorithm whereas merge sort is not in-place. In-place sorting means internal sorting; it does not use additional storage space to perform sorting. In merge sort, to merge

the sorted arrays it requires a temporary array and hence it is not in-place i.e. it is external sorting. Hence quick sort is superior to merge sort in terms of space.

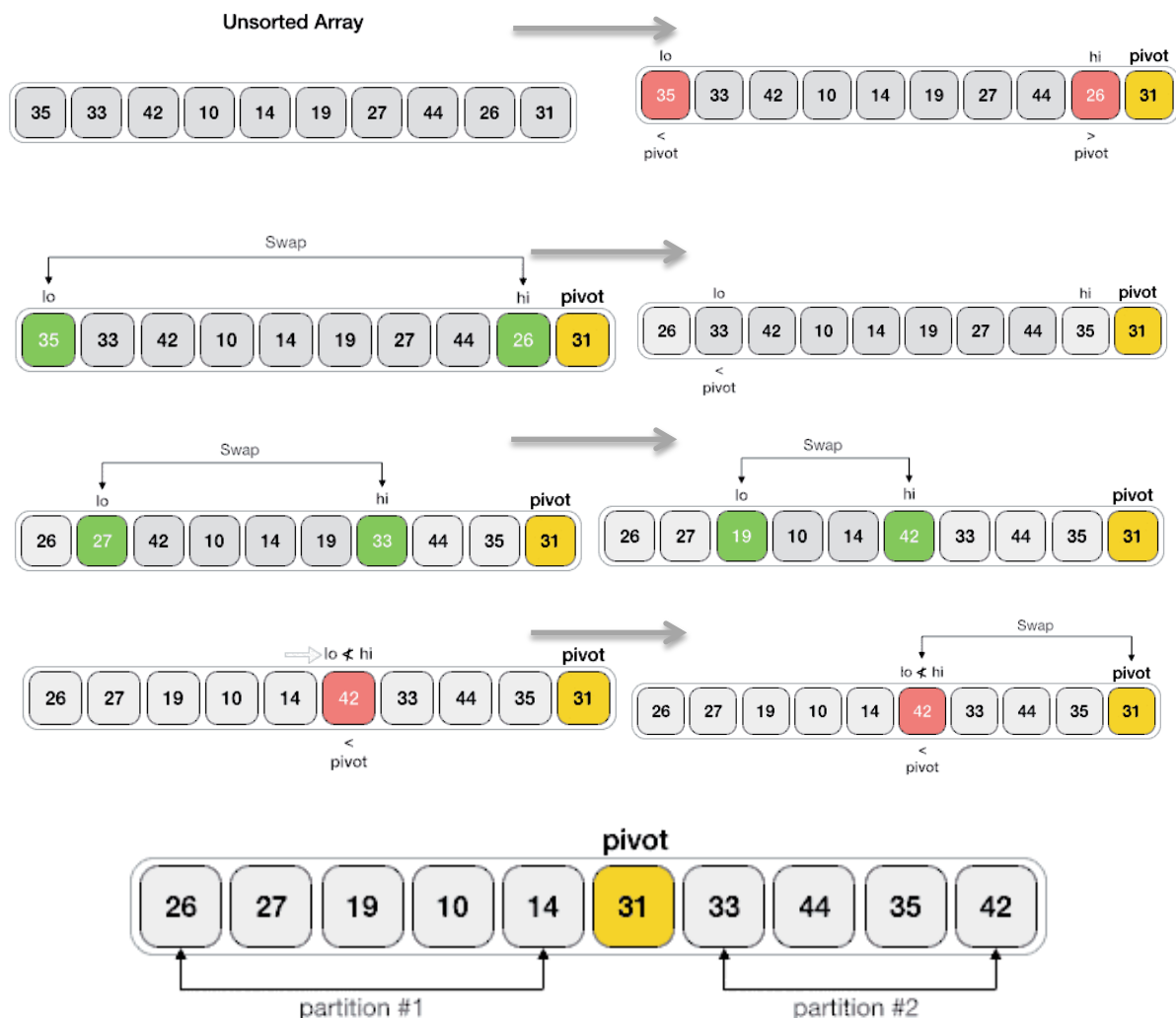
Quick Sort Pivot Algorithm

- Step 1** – Choose the highest index value has pivot
- Step 2** – Take two variables to point left and right of the list excluding pivot
- Step 3** – left points to the low index
- Step 4** – right points to the high
- Step 5** – while value at left is less than pivot move right
- Step 6** – while value at right is greater than pivot move left
- Step 7** – if both step 5 and step 6 does not match swap left and right
- Step 8** – if $\text{left} \geq \text{right}$, the point where they met is new pivot

Quick Sort Algorithm

- Step 1** – Make the right-most index value pivot
- Step 2** – partition the array using pivot value
- Step 3** – quicksort left partition recursively
- Step 4** – quicksort right partition recursively

Example:



Sorting Large Objects

There's no one algorithm that's clearly the "best" algorithm. It depends on a bunch of factors. When comparing various sorting algorithms, there are several things to consider.

The first is usually runtime. When dealing with increasingly large sets of data, inefficient sorting algorithms can become too slow for practical use within an application.

A second consideration is memory space. Faster algorithms that require recursive calls typically involve creating copies of the data to be sorted. In some environments where memory space may be at a premium (such as an embedded system) certain algorithms may be impractical.

A third consideration is stability. Stability, simply defined, is what happens to elements that are comparatively the same. In a stable sort, those elements whose comparison key is the same will remain in the same relative order after sorting as they were before sorting.

The following chart compares sorting algorithms on the various criteria outlined above:

Only a few items => INSERTION SORT
 Items are mostly sorted already => INSERTION SORT
 Concerned about worst-case scenarios => HEAP SORT
 Interested in a good average-case result => QUICKSORT
 Items are drawn from a **dense universe** => BUCKET SORT
 Desire to write as little code as possible => INSERTION SORT

Bucket sort

Bucket sort is a sorting algorithm that works by partitioning an array into a number of buckets. Each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sorting algorithm. It is a **distribution sort**, and is a cousin of **radix sort** in the most to least significant digit flavor.

Bucket sort is a generalization of **pigeonhole sort**. Since bucket sort is **not a comparison sort**, the $(n \log n)$ lower bound is inapplicable. The computational complexity estimates involve the number of buckets.

This means that more auxiliary memory is required for the buckets at the cost of running time than more comparison sorts. It runs in $O(n+k)$ time in the average case where n is the number of elements to be sorted and k is the number of buckets.

Bucket sort's best case occurs when the data being sorted can be distributed between the buckets perfectly.

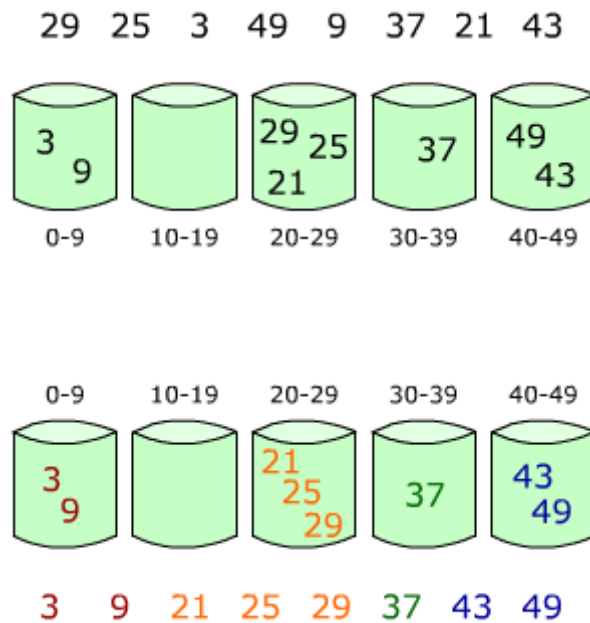
Bucket sort performs at its worst, $O(n^2)$, when all elements are allocated to the same bucket. Since individual buckets are sorted using another algorithm, if only a single bucket needs to be sorted, bucket sort will take on the complexity of the inner sorting algorithm.

Bucket sort works as follows:

- Set up an array of initially empty buckets.

- Go over the original array, putting each object in its bucket.
- Sort each non-empty bucket.
- Visit the buckets in order and put all elements back into the original array.

Example of Bucket Sort:



External Sorting

This term is used to refer to sorting methods that are employed when the data to be sorted is too large to fit in primary memory.

External sorting is a term for a class of sorting algorithms that can handle massive amounts of data. External sorting is required when the data being sorted do not fit into the main memory of a computing device (usually RAM) and instead they must reside in the slower external memory (usually a hard drive).

Characteristics of External Sorting:

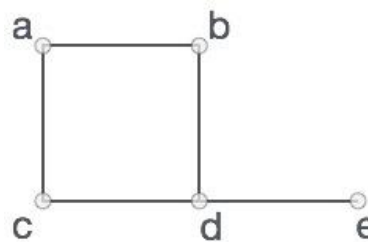
- During the sort, some of the data must be stored externally. Typically the data will be stored on tape or disk.
- The cost of accessing data is significantly greater than either bookkeeping or comparison costs.
- There may be severe restrictions on access. For example, if tape is used, items must be accessed sequentially.

Most external sort routines are based on merge sort. They typically break a large data file into a number of shorter, sorted **runs**. These can be produced by repeatedly reading a section of the data file into RAM, sorting it with ordinary quicksort, and writing the sorted data to disk.

Graphs

Introduction

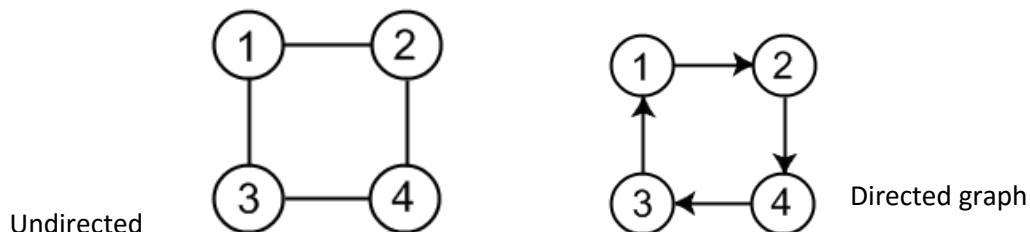
A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as **vertices**, and the links that connect the vertices are called **edges**.



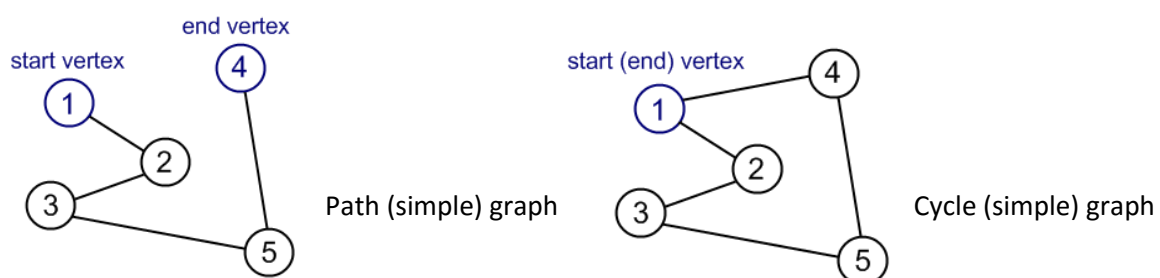
Formally, a graph is a pair of sets (V, E) , where V is the set of vertices and E is the set of edges, connecting the pairs of vertices. Take a look at the following graph –

In the above graph,
 $V = \{a, b, c, d, e\}$
 $E = \{ab, ac, bd, cd, de\}$

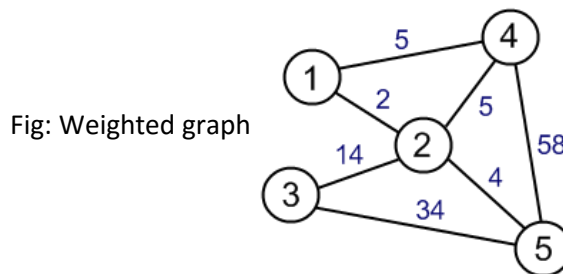
All graphs are divided into two big groups: directed and undirected graphs. The difference is that edges in directed graphs, called *arcs*, have a direction. These kinds of graphs have much in common with each other, but significant differences are also present.



Sequence of vertices, such that there is an edge from each vertex to the next in sequence, is called **path**. First vertex in the path is called the *start vertex*; the last vertex in the path is called the *end vertex*. If start and end vertices are the same, path is called **cycle**. Path is called *simple*, if it includes every vertex only once. Cycle is called *simple*, if it includes every vertex, except start (end) one, only once. Let's see examples of path and cycle.



The last definition I give here is a weighted graph. Graph is called *weighted*, if every edge is associated with a real number, called edge weight. For instance, in the road network example, weight of each road may be its length or minimal time needed to drive along.



Mathematical graphs can be represented in data-structure. We can represent a graph using an array of vertices and a two dimensional array of edges. Before we proceed further, let's familiarize ourselves with some important terms –

Vertex – each node of the graph is represented as a vertex. In example given below, labeled circle represents vertices. So A to G are vertices. We can represent them using an array as shown in image below. Here A can be identified by index 0. B can be identified using index 1 and so on.

Edge – Edge represents a path between two vertices or a line between two vertices. In example given below, lines from A to B, B to C and so on represent edges. We can use a two dimensional array to represent array as shown in image below. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.

Adjacency – Two node or vertices are adjacent if they are connected to each other through an edge. In example given below, B is adjacent to A; C is adjacent to B and so on.

Path – Path represents a sequence of edges between two vertices. In example given below, ABCD represents a path from A to D.

Basic Operations

Following are basic primary operations of a Graph:

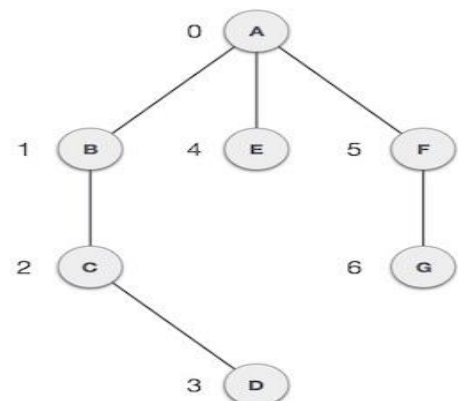
Add Vertex – add a vertex to a graph.

Add Edge – add an edge between two vertices of a graph.

Display Vertex – display a vertex of a graph.

Degree: The number of edges incident on a vertex determine its degree. The degree of vertex u , is written as $\text{degree}(u)$. If $\text{degree}(u) = 0$, this means that vertex u does not belongs to any edge then vertex u is called isolated vertex.

Complete Graph: A graph G is said to complete or fully connected if there is path from every vertex to every other vertex. A complete graph with n vertices will have $n*(n-1)/2$ edges.



Tree: A graph is a tree if it has two properties:

1. It is connected, and
2. There are no cycles in the graph.

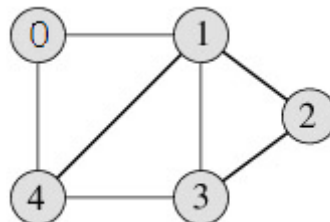
Graph Representation

Following two are the most commonly used representations of graph.

1. Adjacency Matrix
2. Adjacency List

There are other representations also like, Incidence Matrix and Incidence List. The choice of the graph representation is situation specific. It totally depends on the type of operations to be performed and ease of use.

Following is an example undirected graph with 5 vertices.



Adjacency Matrix:

Adjacency Matrix is a 2D array of size $V \times V$ where V is the number of vertices in a graph. Let the 2D array be $adj[i][j]$, a slot $adj[i][j] = 1$ indicates that there is an edge from vertex i to vertex j . Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If $adj[i][j] = w$, then there is an edge from vertex i to vertex j with weight w .

The adjacency matrix for the above example graph is:

	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

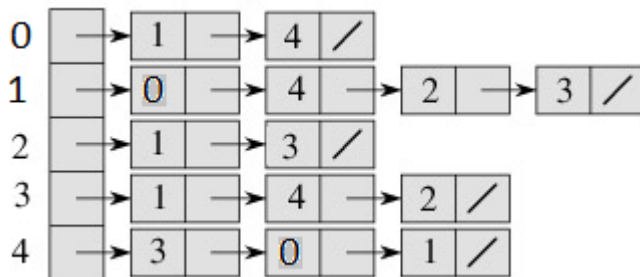
Table: Adjacency Matrix Representation of the above graph

Advantages: Representation is easier to implement and follow. Removing an edge takes $O(1)$ time. Queries like whether there is an edge from vertex 'u' to vertex 'v' are efficient and can be done $O(1)$.

Disadvantages: Consumes more space $O(V^2)$. Even if the graph is sparse(contains less number of edges), it consumes the same space. Adding a vertex is $O(V^2)$ time.

Adjacency List:

An array of linked lists is used. Size of the array is equal to number of vertices. Let the array be `array[]`. An entry `array[i]` represents the linked list of vertices adjacent to the i th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be stored in nodes of linked lists. Following is adjacency list representation of the above graph.



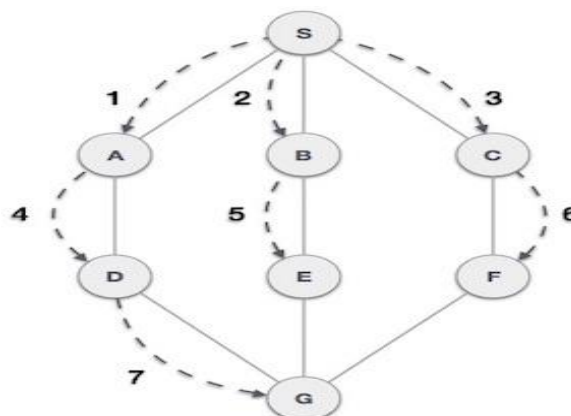
Graph Traversal

To traverse a graph is to process every node in the graph exactly once. Because there are many paths leading from one node to another, the hardest part about traversing a graph is making sure that you do not process some node twice. There are two general solutions to this difficulty:

1. Breadth First Traversal (Breadth First Search –BFS)

The breadth-first-search algorithm starts at a vertex i and visits, first the neighbors of i , then the neighbors of the neighbors of i , then the neighbors of the neighbors of the neighbors of i , and so on.

Breadth First Search algorithm(BFS) traverses a graph in a breadth wards motion and uses a queue to remember to get the next vertex to start a search when a dead end occurs in any iteration.



As in example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs following rules.

Rule 1 – Visit adjacent unvisited vertex. Mark it visited. Display it. Insert it in a queue.

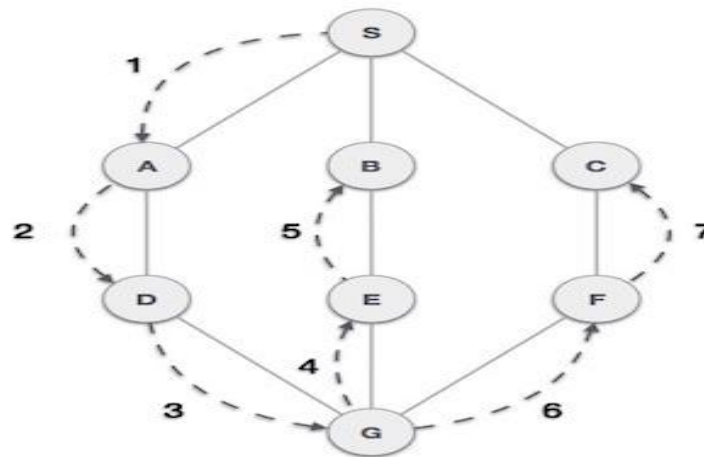
Rule 2 – If no adjacent vertex found, remove the first vertex from queue.

Rule 3 – Repeat Rule 1 and Rule 2 until queue is empty.

2. Depth First Traversal (Depth First Search – DFS)

The depth-first-search algorithm is similar to the standard algorithm for traversing binary trees; it first fully explores one subtree before returning to the current node and then exploring the other subtree. Another way to think of depth-first-search is by saying that it is similar to breadth-first search except that it uses a stack instead of a queue.

Depth First Search algorithm(DFS) traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search when a dead end occurs in any iteration.



As in example given above, DFS algorithm traverses from A to B to C to D first then to E, then to F and lastly to G. It employs following rules.

- **Rule 1** – Visit adjacent unvisited vertex. Mark it visited. Display it. Push it in a stack.
- **Rule 2** – If no adjacent vertex found, pop up a vertex from stack. (It will pop up all the vertices from the stack which do not have adjacent vertices.)
- **Rule 3** – Repeat Rule 1 and Rule 2 until stack is empty.

Applications of Depth First Search

1. **Minimum spanning Tree:** For an unweighted graph, DFS traversal of the graph produces the minimum spanning tree and all pair shortest path tree.
2. **Detecting cycle in a graph:** A graph has cycle if and only if we see a back edge during DFS. So we can run DFS for the graph and check for back edges.
3. **Path Finding:** We can specialize the DFS algorithm to find a path between two given vertices u and z; Call DFS (G, u) with u as the start vertex.
4. **Topological Sorting:** Topological Sorting is mainly used for scheduling jobs from the given dependencies among jobs.
5. **To test if a graph is bipartite:** We can augment either BFS or DFS when we first discover a new vertex, color it opposed its parents, and for each other edge, check it doesn't link two vertices of the same color.
6. **Finding Strongly Connected Components of a graph:** A directed graph is called strongly connected if there is a path from each vertex in the graph to every other vertex.
7. **Solving puzzles with only one solution:** such as mazes. (DFS can be adapted to find all solutions to a maze by only including nodes on the current path in the visited set.)

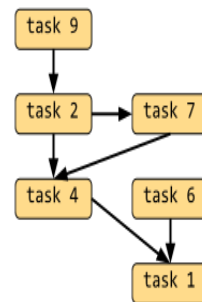
Topological Sort

In the field of computer science, a **topological sort** (sometimes abbreviated **toposort**) or **topological ordering** of a directed graph is a linear ordering of its vertices such that for every directed edge uv from vertex u to vertex v , u comes before v in the ordering.

There are many problems involving a set of tasks in which some of the tasks must be done before others. For example, consider the problem of taking a course only after taking its prerequisites.

Topological sort is a method of arranging the vertices in a directed acyclic graph (DAG), as a sequence, such that no vertex appears in the sequence before its predecessor.

TOPOLOGICAL SORT



RIGHT



WRONG



For each edge (u,v) u must be before v !

Topological sort is not unique

- The following are all topological sort of the graph shown)

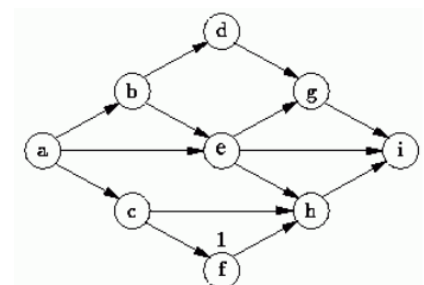
$s1 = \{a, b, c, d, e, f, g, h, i\}$

$s2 = \{a, c, b, f, e, d, h, g, i\}$

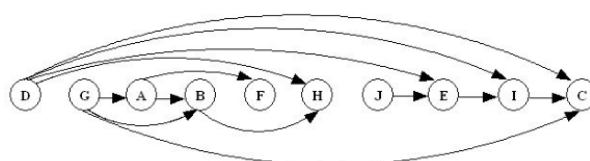
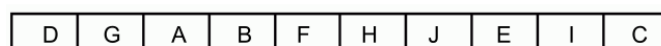
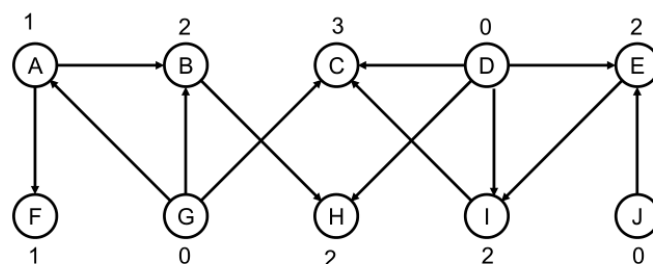
$s3 = \{a, b, d, c, e, g, f, h, i\}$

$s4 = \{a, c, f, b, e, h, d, g, i\}$

etc.



Topological sort example



Shortest Path Algorithm

Dijkstra's algorithm solves the single-source shortest-path problem when all edges have non-negative weights. It is a greedy algorithm and similar to Prim's algorithm. Algorithm starts at the source vertex, s , it grows a tree, T , that ultimately spans all vertices reachable from S . Vertices are added to T in order of distance i.e., first S , then the vertex closest to S , then the next closest, and so on.

Dijkstra's Algorithm solves the single-source shortest-paths problem on a weighted, directed graph $G = (V, E)$ for the case in which all edge weights are non-negative. Here we assume that $W(u, v) > 0$ for each edge $(u, v) \in E$. Dijkstra's Algorithm maintains a set of S of vertices whose final shortest-path weights from source have already been determined.

Dijkstra's Algorithm

Step 1: Label the initial vertex of the graph with the weight zero.

Step 2: Calculate the weights of all vertices adjacent to the initial vertex corresponding to the weights of the edges incident on the initial vertex.

Step 3: Label these vertices with smallest possible value of their weights.

Step 4: Calculate the weights of all those vertices which are adjacent to the vertices with minimum weights determined in step 3.

Step 5: Label these vertices with minimum weight.

Step 6: Continue this process until all the vertices of weighted graph are labeled.

Step 7: Trace the path of cumulative minimum weight from the initial vertex to desire vertex.

The **relaxation** process updates the costs of all the vertices, v , connected to a vertex, u , if we could improve the best estimate of the shortest path to v by including (u, v) in the path to v .

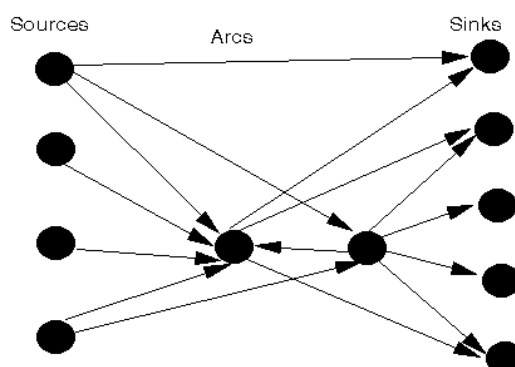
The shortest path between two vertices is a path with the shortest length (least number of edges). Call this the link-distance.

Breadth-first-search is an algorithm for finding shortest (link-distance) paths from a single source vertex to all other vertices.

BFS processes vertices in increasing order of their distance from the root vertex.

Network Flow Problems

To illustrate the general network flow problem consider the diagram below where we have a number of *sources* of material and a number of *sinks* (or demand points) for material. Typically each source has an upper limit on the amount of material it can supply and each demand point has an associated number indicating the amount of material it needs.



Between the sources and the sinks are intermediate *nodes* through which material can be shipped (flows) to other intermediate nodes or to the sinks. We also have *arcs* (essentially directed from the sources to the sinks) where each arc has associated with it:

- An upper limit (or *capacity*) on the amount of material which can flow down the arc; and
- A *cost* per unit of material sent down the arc.

Hence the problem is one of deciding how to supply the sinks from the sources at minimum cost. This problem is known as the *minimum cost network flow problem*.

In graph theory, a flow network (also known as a transportation network) is a directed graph where each edge has a capacity and each edge receives a flow. The amount of flow on an edge cannot exceed the capacity of the edge. Often in operations research, a directed graph is called a network. The vertices are called nodes and the edges are called arcs. A flow must satisfy the restriction that the amount of flow into a node equals the amount of flow out of it, unless it is a source, which has only outgoing flow, or sink, which has only incoming flow. A network can be used to model traffic in a road system, circulation with demands, fluids in pipes, currents in an electrical circuit, or anything similar in which something travels through a network of nodes.

Minimum Spanning Tree (MST)

A **minimum spanning tree** is a spanning tree of a connected, undirected graph. It connects all the vertices together with the minimal total weighting for its edges.

A *spanning tree* of a graph is just a sub-graph that contains all the vertices and is a tree. A graph may have many spanning trees; for instance the complete graph on four vertices has sixteen spanning trees. Among all the spanning trees of a weighted and connected graph, the one (possibly more) with the least total weight is called a minimum spanning tree (MST). If each edge has a distinct weight then there will be only one, unique minimum spanning tree.

A minimum spanning tree has $(V - 1)$ edges where V is the number of vertices in the given graph. A *minimum spanning tree (MST)* or minimum weight spanning tree for a weighted, connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.

Kruskal's Algorithm

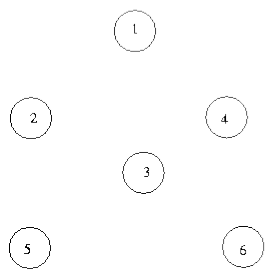
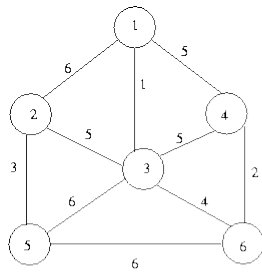
Kruskal's algorithm is a minimum-spanning-tree algorithm which finds an edge of the least possible weight that connects any two trees in the forest. It is a greedy algorithm in graph theory as it finds a minimum spanning tree for a connected weighted graph adding increasing cost arcs at each step.

Algorithm

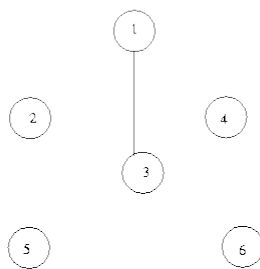
1. T (the final spanning tree) is defined to be the empty set;
2. For each vertex v of G , make the empty set out of v ;
3. Sort the edges of G in ascending (non-decreasing) order;
4. for each edge (u, v) from the sored list of step 3.
 - If u and v belong to different sets
 - Add (u, v) to T ;
 - Get together u and v in one single set;
5. Return T

The algorithm is a Greedy Algorithm. The Greedy Choice is to pick the smallest weight edge that does not because a cycle in the MST constructed so far.

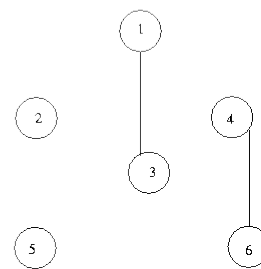
Example of finding a minimum spanning tree using Kurskal's Algorithm



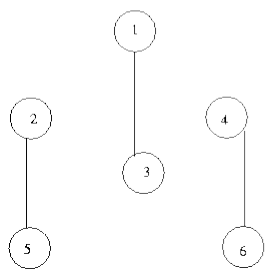
Initial Configuration



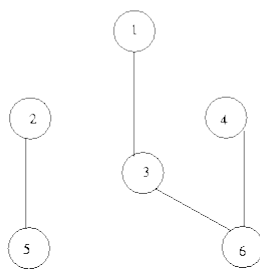
step1. choose (1,3)



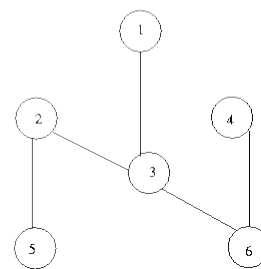
step2. choose (4,6)



step3. choose (2,5)



step4. choose (3,6)



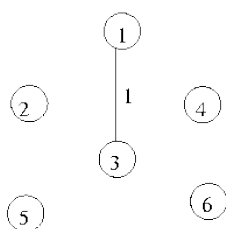
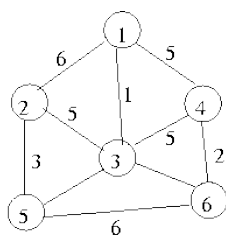
step5. choose (2,3)

Prim's Algorithm

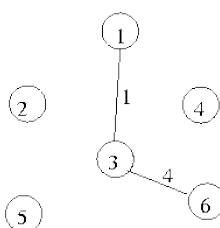
In computer science, **Prim's algorithm** is a greedy algorithm that finds a minimum spanning tree for a weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. The algorithm operates by building this tree one vertex at a time, from an arbitrary starting vertex, at each step adding the cheapest possible connection from the tree to another vertex.

The algorithm may informally be described as performing the following steps:

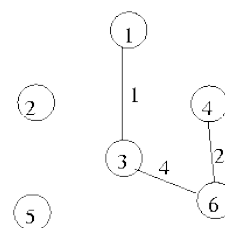
1. Initialize a tree with a single vertex, chosen arbitrarily from the graph.
2. Grow the tree by one edge: of the edges that connect the tree to vertices not yet in the tree, find the minimum-weight edge, and transfer it to the tree.
3. Repeat step 2 (until all vertices are in the tree).



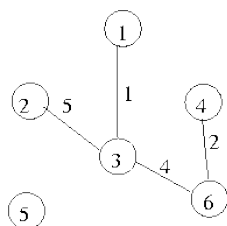
Iteration 1. $U = \{1\}$



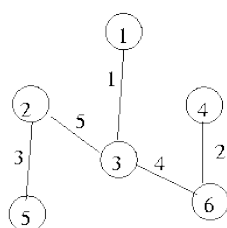
Iteration 2. $U = \{1, 3\}$



Iteration 3. $U = \{1, 3, 6\}$



Iteration 4. $U = \{1, 3, 6, 4\}$



Iteration 5. $U = \{1, 3, 6, 4, 2\}$

Like Kruskal's algorithm, Prim's algorithm is also a Greedy algorithm. It starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets, and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.

The idea behind Prim's algorithm is simple; a spanning tree means all vertices must be connected. So the two disjoint subsets (discussed above) of vertices must be connected to make a *Spanning Tree*.

Algorithm Design Techniques

Greedy Algorithm

Greedy algorithms are simple and straightforward. They are shortsighted in their approach in the sense that they take decisions on the basis of information at hand without worrying about the effect these decisions may have in the future. They are easy to invent, easy to implement and most of the time quite efficient. Many problems cannot be solved correctly by greedy approach. Greedy algorithms are used to solve optimization problems.

A **greedy algorithm** always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution. Greedy

algorithms do not always yield optimal solutions, but for many problems they do. An algorithm that always takes the best immediate, or local, solution while finding an answer. Greedy algorithms find the overall, or globally, *optimal solution* for some *optimization problems*, but may find less-than-optimal solutions for some instances of other problems.

Greedy algorithms work by recursively constructing a set of objects from the smallest possible constituent parts. Recursion is an approach to problem solving in which the solution to a particular problem depends on solutions to smaller instances of the same problem. The advantage to using a greedy algorithm is that solutions to smaller instances of the problem can be straightforward and easy to understand. The disadvantage is that it is entirely possible that the most optimal short-term solutions may lead to the worst possible long-term outcome.

A greedy algorithm is a mathematical process that looks for simple, easy-to-implement solutions to complex, multi-step problems by deciding which next step will provide the most obvious benefit. *Prim-Jarnik algorithm and Kruskal's algorithm are greedy algorithms that find the globally optimal solution, a minimum spanning tree.* Greedy algorithms are often used in ad hoc mobile networking to efficiently route packets with the fewest number of hops and the shortest delay possible. They are also used in machine learning, business intelligence (BI), artificial intelligence (AI) and programming.

Divide and Conquer Algorithm

Divide-and-conquer is a top-down technique for designing algorithms that consists of dividing the problem into smaller sub-problems hoping that the solutions of the sub-problems are easier to find and then composing the partial solutions into the solution of the original problem. Binary Search, Quick sort and Merge sort are an extremely well-known instance of divide-and-conquer paradigms.

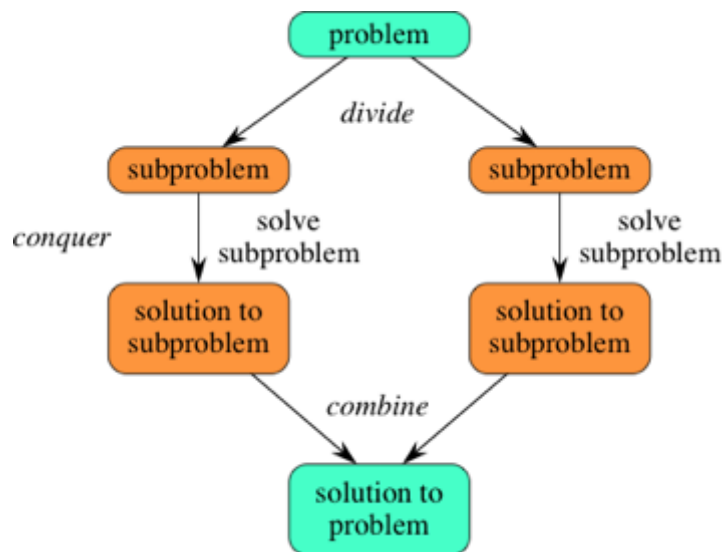
Divide-and-conquer paradigm consists of following major phases:

- Breaking the problem into several sub-problems that are similar to the original problem but smaller in size,
- Solve the sub-problem recursively (successively and independently), and then
- Combine these solutions to sub-problems to create a solution to the original problem.

In divide and conquer approach, the problem in hand, is divided into smaller sub-problems and then each problem is solved independently. When we keep on dividing the sub-problems into even smaller sub-problems, we may eventually reach at a stage where no more division is possible. Those "atomic" smallest possible sub-problems (fractions) are solved. The solution of all sub-problems is finally merged in order to obtain the solution of original problem.

We should think of a divide-and-conquer algorithm as having three parts:

1. **Divide** the problem into a number of sub-problems that are smaller instances of the same problem.
2. **Conquer/solve** the sub-problems by solving them recursively. If they are small enough, solve the sub-problems as base cases.
3. **Combine** the solutions to the sub-problems into the solution for the original problem.



Dynamic Programming

Dynamic programming approach is similar to divide and conquer in breaking down the problem in smaller and yet smaller possible sub-problems. But unlike, divide and conquer, these sub-problems are not solved independently. Rather, results of these smaller sub-problems are remembered and used for similar or overlapping sub-problems.

In mathematics, management, science, economics, computer, science, and bioinformatics, **dynamic programming** (also known as **dynamic optimization**) is a method for solving a complex problem by breaking it down into a collection of simpler sub-problems, solving each of those sub-problems just once, and storing their solutions - ideally, using a memory-based data structure.

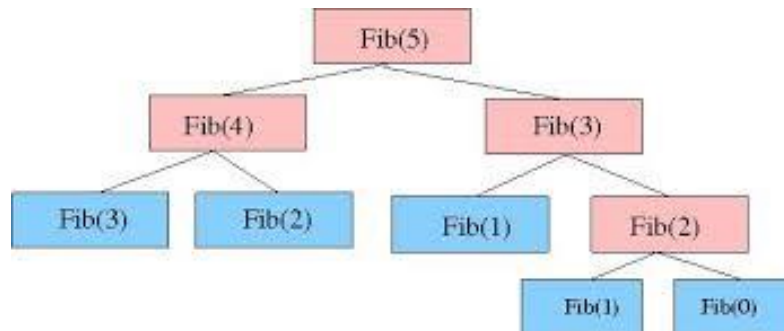
Dynamic programming (usually referred to as **DP**) is a very powerful technique to solve a particular class of problems. It demands very elegant formulation of the approach and simple thinking and the coding part is very easy. The idea is very simple, If you have solved a problem with the given input, then save the result for future reference, so as to avoid solving the same problem again. Shortly '*Remember your Past*':. If the given problem can be broken up in to smaller sub-problems and these smaller sub-problems are in turn divided in to still-smaller ones, and in this process, if you observe some overlapping sub-problems, then it's a big hint for DP.

As compared to divide-and-conquer, dynamic programming is more powerful and subtle design technique. Dynamic programming takes advantage of the duplication and arranges to solve each sub-problem only once, saving the solution (in table or in a globally accessible place) for later use. The underlying idea of dynamic programming is: avoid calculating the same stuff twice, usually by keeping a table of known results of sub-problems. Unlike divide-and-conquer, which solves the sub-problems top-down, a dynamic programming is a bottom-up technique. The dynamic programming technique is related to divide-and-conquer, in the sense that it breaks problem down into smaller problems and it solves recursively. However, because of the somewhat different nature of dynamic programming problems, standard divide-and-conquer solutions are not usually efficient.

Bottom-up means

- Start with the smallest sub-problems.
- Combining theirs solutions obtain the solutions to sub-problems of increasing size.
- Until arrive at the solution of the original problem

Fibonacci number series, Tower of Hanoi, Shortest path by Dijkstra etc. can be solved by using dynamic programming algorithm.



Divide & Conquer	Dynamic Programming
1. Partitions a problem into independent smaller sub-problems	1. Partitions a problem into overlapping sub-problems
2. Doesn't store solutions of sub-problems. (Identical sub-problems may arise - results in the same computations are performed repeatedly.)	2. Stores solutions of sub-problems: thus avoids calculations of same quantity twice
3. Top down algorithms: which logically progresses from the initial instance down to the smallest sub-instances via intermediate sub-instances.	3. Bottom up algorithms: in which the smallest sub-problems are explicitly solved first and the results of these used to construct solutions to progressively larger sub-instances

Randomized algorithms

A **randomized algorithm** is an algorithm that employs a degree of randomness as part of its logic. The algorithm typically uses uniformly random bits as an auxiliary input to guide its behavior, in the hope of achieving good performance in the "average case" over all possible choices of random bits. For many applications, a randomized algorithm is either the simplest or the fastest algorithm available, and sometimes both.

A randomized algorithm is one that receives, in addition to its input data, a stream of random bits that it can use for the purpose of making random choices. Even for a fixed input, different runs of a randomized algorithm may give different results; thus it is inevitable that a description of the properties of a randomized algorithm will involve probabilistic statements. For example, even when the input is fixed, the execution time of a randomized algorithm is a random variable.

Backtracking algorithms

Backtracking is a form of recursion. Recursion is the key in backtracking programming. As the name suggests we backtrack to find the solution. We start with one possible move out of many available moves and try to solve the problem if we are able to solve the problem with the selected move then we will print the solution else we will backtrack and select some other move and try to solve it. If none of the moves work out we will claim that there is no solution for the problem.

Pick a starting point.

While (Problem is not solved)

For each path from the starting point

Check if selected path is safe, if yes select it

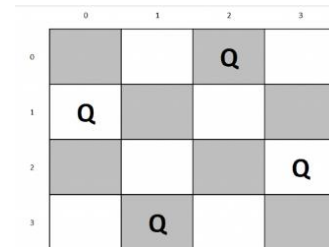
And make recursive call to rest of the problem

If recursive calls returns true, then return true.

Else undo the current move and return false.

End For

If none of the move works out, return false, NO SOLUTION.



The classic textbook example of the use of backtracking is the eight queens puzzle, that asks for all arrangements of eight chess queens on a standard chessboard so that no queen attacks any other. Backtracking is an important tool for solving constraint satisfaction problems, such as crosswords, verbal arithmetic, Sudoku, and many other puzzles.

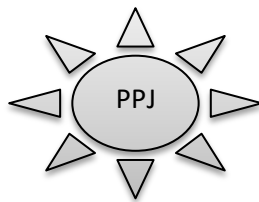
- It is a step-by-step representation of a solution to a given problem ,which is very easy to understand
- It has got a definite procedure.
- It easy to first develop an algorithm & then convert it into a flowchart &then into a computer program.
- It is independent of programming language.
- It is easy to debug as every step is got its own logical sequence.
- But it is time consuming computer program

The Summary of all data structures we have studied here is as follows:

Array	Quick inserts Fast access if index known	Slow search Slow deletes Fixed size
Ordered Array	Faster search than unsorted array	Slow inserts Slow deletes Fixed size
Stack	Last-in, first-out access	Slow access to other items
Queue	First-in, first-out access	Slow access to other items
Linked List	Quick inserts Quick deletes	Slow search
Binary Tree	Quick search	Deletion algorithm is complex

	Quick inserts Quick deletes <i>(If the tree remains balanced)</i>	
Red-Black Tree	Quick search Quick inserts Quick deletes <i>(Tree always remains balanced)</i>	Complex to implement
2-3-4 Tree	Quick search Quick inserts Quick deletes <i>(Tree always remains balanced)</i> <i>(Similar trees good for disk storage)</i>	Complex to implement
Hash Table	Very fast access if key is known Quick inserts	Slow deletes Access slow if key is not known Inefficient memory usage
Heap	Quick inserts Quick deletes Access to largest item	Slow access to other items
Graph	Best models real-world situations	Some algorithms are slow and very complex

“A good programmer is someone who always looks both ways before crossing a one-way street.”__ PPJ



DATA STRUCTURES LAB SYLLABUS

There are 10 exercises based on C or C ++

Exercise 1: Implementation of Stack.

Exercise 2: Implementation of Linear and Circular queue.

Exercise 3: Solution of TOH and Fibonacci recursion.

Exercise 4: Implementation of Linked list: Singly, and doubly linked list.

Exercise 5: Implementation of tree: AVL tree, Balancing of AVL.

Exercise 6: Implementation of Merge sort.

Exercise 7: Implementation of Search: Sequential, Tree and Binary.

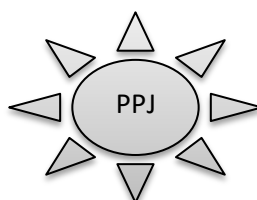
Exercise 8: Implementation of Graphs: Graph traversals.

Exercise 9: Implementation of Heap.

Exercise 10: Implementation of Hashing.

"Bad programmers worry about the code. Good programmers worry about data structures and their relationships."__ PPJ

"The gap between theory and practice is not as wide in theory as it is in practice."__ PPJ



1. Implementation of stack

1. A C program that implements stack (its operations) using arrays.

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>

#define size 5
struct stack {
    int s[size];
    int top;
} st;

int stfull() {
    if (st.top >= size - 1)
        return 1;
    else
        return 0;
}

void push(int item) {
    st.top++;
    st.s[st.top] = item;
}

int stempty() {
    if (st.top == -1)
        return 1;
    else
        return 0;
}

int pop() {
    int item;
    item = st.s[st.top];
    st.top--;
    return (item);
}

void display() {
    int i;
    if (stempty())
        printf("\nStack Is Empty!");
    else {
        for (i = st.top; i >= 0; i--)
            printf("\n%d", st.s[i]);
    }
}
```

```
int main() {
    int item, choice;
    char ans;
    st.top = -1;

    printf("\n\tImplementation Of Stack");
    printf("\n\t_____");
    clrscr();
    do {
        printf("\nMain Menu");
        printf("\n1.Push \n2.Pop \n3.Display \n4.exit");
        printf("\nEnter Your Choice : ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("\nEnter The item to be pushed :");
                scanf("%d", &item);
                if (stfull())
                    printf("\nStack is Full!");
                else
                    push(item);
                break;
            case 2:
                if (stempty())
                    printf("\nEmpty stack! Underflow !!");
                else {
                    item = pop();
                    printf("\nThe popped element is %d. ",
item);
                }
                break;
            case 3:
                display();
                break;
            case 4:
                exit(0);
        }
        printf("\n Do You want To Continue?");
        ans = getch();
    } while (ans == 'Y' || ans == 'y');
    return 0;
}
```

Output: (Execute Yourself.)

II. A C program that implements stack (its operations) using Linked List.

```
#include<stdio.h>
#include<conio.h>
#include<process.h>
#include<stdlib.h>
struct stack
{
    int info;
    struct stack *next;
};
void push(int);
void pop(void);
int topmost(void);
void display();
struct stack *getnode(int x);

/* pointer declaration to point to the top of the
stack */
struct stack *top;
void main()
{
    int item, ch;
    top = NULL;

    clrscr();
    while(1)
    {
        printf("\n1.push");
        printf("\n2.pop");
        printf("\n3.Top ELEMENT");
        printf("\n4.display");
        printf("\n5.exit");
        printf("\nEnter your choice\n");
        scanf("%d",&ch);

        switch(ch)
        {
            case 1: printf("Enter the element to be
pushed\n");
scanf("%d",&item);
push(item);
printf("\nPress enter");
getch();
break;

            case 2: pop();
printf("\nPress enter");
```

```
getch();
break;

            case 3: item=topmost();
printf("\n Topmost element  is %d",item);
printf("\nPress enter");
getch();
break;

            case 4: display();
printf("\nPress enter");
getch();
break;

            default : exit(0);
        }
    }

    void push( int x)
    {
        struct stack *p;
        p = getnode(x);

        /* checking for an empty stack */
        if ( top == NULL)
        {
            top = p;
        }
        else
        {
            p->next = top;
            top = p;
        }
    } /* end of push( )*/

    struct stack *getnode(int x)
    {
        struct stack *p;
        p=(struct stack *) malloc(sizeof(struct stack));
        p->info=x;
        p->next = NULL;
        return(p);
    }

    void pop( void)
    {
        struct stack *temp;
        int x;
```

```

/* check for an empty stack */

if (top == NULL)
{
    printf ("Cannot remove nodes from an empty
    stack");
    getch();
}
else
{
    temp = top;
    x = top->info;
    top = top->next;
    free( temp);
    printf("\nDeleted item is %d",x);
    return;
}
}
int topmost( void)
{
    return top->info;
}

void display()
{
    struct stack *p;
    p=top;
    if (top == NULL)
    {
        printf("\nStack is Empty");
        return;
    }
    printf("\nContents of the stack are\n");
    for(p=top; p!= NULL; p=p->next)
    printf("%d\t",p->info);
}
    
```

Output: (Execute Yourself.)

2. Solution of TOH and Fibonacci Recursion

I. Solution of TOH (Tower of Hanoi) using recursion.

```

#include<stdio.h>
#include<conio.h>
void hanoi(int n,char A,char B,char C);
main()
    
```

```

{
    int n;
    char A='A',B='B',C='C';
    clrscr();
    printf("\n Enter number of disks:\n");
    scanf("%d",&n);
    printf("\n Towers of Hanoi problem with %d
    disks \n",n);
    printf("\n
    *****\n");
    printf("\t Sequence is: \n");
    hanoi(n,A,B,C);
    getch();
}

void hanoi(int n,char A,char B,char C)
{
    if(n!=0)
    {
        hanoi(n-1,A,C,B);
        printf("Move disk %d from %c -> %c
        \n",n,A,C);
        hanoi(n-1,B,A,C);
    }
}
    
```

Output: (Execute Yourself.)

II. Solution of Fibonacci Sequence Using recursion.

```

#include<stdio.h>
#include<conio.h>
int fib(int n);
void main()
{
    int n,result;
    clrscr();
    printf("Enter the nth number in fibonacci
    series:\n");
    scanf("%d",&n);
    result=fib(n);
    printf("The series in fibonacci number %d is:
    %d\n",n,result);
    getch();
}
int fib(int n)
{
    if(n==0)
    
```

```

printf(" Zero position is not specified in
fibonacci sequence. ");
else if(n==1)
return 0;
else if(n==2)
return 1;
else
return fib(n-1)+fib(n-2);
}

```

Output: (Execute Yourself.)

3. Implementation of Linear and Circular Queue.

I. A C – Program to Implement Linear Queue using Array.

```

#include<stdio.h>
#include<conio.h>
#define max 10
int a[max];
int rear=-1;
int front=-1;
void insertion(int a[],int val);
void deletion(int a[]);
void peep(int a[]);
void display(int a[]);

main()
{
int ch,val;
clrscr();
while(1)
{
printf("\n 1.insertion \n 2.deletion \n 3.peep \n
4.display \n 5.exit \n");
printf("enter your choice:");
scanf("%d",&ch);
switch(ch)
{
case 1:printf("Enter the value to insert in
queue:\n");
scanf("%d",&val);
insertion(a,val);
break;
case 2:deletion(a);
break;
case 3:peep(a);

```

```

break;
case 4:display(a);
break;
case 5:exit(1);
break;
default:printf("invalid choice");
}
}
}
void insertion(int a[],int val)
{
if(rear==max-1)
{
printf("Queue overflow");
}
else
{
if(front==-1)
front=0;
rear=rear+1;
a[rear]=val;
}
}
void deletion(int a[])
{
if(front==-1 && rear==-1)
{
printf("Queue underflow");
}
else
{
int val;
val=a[front];
if(front==rear)
{
front=rear=-1;
}
else
{
front=front+1;
}
printf("The deleted element from the queue
is:%d",val);
}
}
void peep(int a[])
{
if(front==-1 && rear==-1)
{
printf("queue is empty");

```

```

    }
    else
    {
        printf("The maximum element in the queue
is:%d",a[rear]);
    }
}
void display(int a[])
{
    int i;
    if(front== -1 && rear== -1)
    {
        printf("queue is empty");
    }
    else
    {
        for(i=front;i<=rear;i++)
        {
            printf("%d\t",a[i]);
        }
    }
}

```

Output: (Execute Yourself.)

II. A C - Program to Implement Circular Queue using Array.

```

#include<stdio.h>
#define max 5
int q[10],front=0,rear=-1;
void main()
{
    int ch;
    void insert();
    void delet();
    void display();
    clrscr();
    printf("\nCircular Queue operations\n");

    printf("1.insert\n2.delete\n3.display\n4.exit\n");
    while(1)
    {
        printf("Enter your choice:");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1: insert();
                    break;

```

```

            case 2: delet();
                    break;
            case 3:display();
                    break;
            case 4:exit();
            default:printf("Invalid option\n");
        }
    }
}

void insert()
{
    int x;
    if((front==0&&rear==max-1) || (front>0&&rear==front-1))
        printf("Queue is overflow\n");
    else
    {
        printf("Enter element to be insert:");
        scanf("%d",&x);
        if(rear==max-1&&front>0)
        {
            rear=0;
            q[rear]=x;
        }
        else
        {
            if((front==0&&rear== -1) || (rear!=front-1))
                q[++rear]=x;
        }
    }
}

void delet()
{
    int a;
    if((front==0)&&(rear== -1))
    {
        printf("Queue is underflow\n");
        getch();
        exit();
    }
    if(front==rear)
    {
        a=q[front];
        rear=-1;
        front=0;
    }
    else
        if(front==max-1)
        {

```

```

        a=q[front];
        front=0;
    }
    else a=q[front++];
    printf("Deleted element is:%d\n",a);
}

void display()
{
    int i,j;
    if(front==0&&rear==1)
    {
        printf("Queue is underflow\n");
        getch();
        exit();
    }
    if(front>rear)
    {
        for(i=0;i<=rear;i++)
            printf("\t%d",q[i]);
        for(j=front;j<=max-1;j++)
            printf("\t%d",q[j]);
        printf("\nrear is at %d\n",q[rear]);
        printf("\nfront is at %d\n",q[front]);
    }
    else
    {
        for(i=front;i<=rear;i++)
        {
            printf("\t%d",q[i]);
        }
        printf("\nrear is at %d\n",q[rear]);
        printf("\nfront is at %d\n",q[front]);
    }
    printf("\n");
}
getch();

```

Output: (Execute Yourself.)

4. Implementation of Linked List

I. A C - Program to Implement Singly Linked Lists.

```

#include<conio.h>
#include<stdio.h>
struct node

```

```

{
    int i;
    struct node *next;
};

void main()
{
    struct node *first;
    struct node *last;
    struct node *temp;
    int ch,user,add,cnt=0,t=0;
    struct node *p;

    clrscr();
    printf("\n\t 1.CREATION");
    printf("\n\t 2.INSERT AT STARTING");
    printf("\n\t 3.INSERT AT MIDDLE(USER'S CHOICE)");
    printf("\n\t 4.INSERT AT END");
    printf("\n\t 5.DELETE 1ST NODE");
    printf("\n\t 6.DELETE LAST NODE");
    printf("\n\t 7.DELETE USER'S CHOICE");
    printf("\n\t 8.DISPLAY");
    printf("\n\t 10.EXIT");
    printf("\n\t Enter your Choice: ");
    scanf("%d",&user);

    while(user!=10)
    {
        if(user==1)
        {
            printf("\n\t ENTER DATA : ");
            first=(struct node*)malloc(sizeof(struct node));
            scanf("%d",&ch);

            first->i=ch;
            first->next=0;
            p=first;
            cnt=1;
        }
        if(user==2)
        {
            p=(struct node*)malloc(sizeof(struct node));
            printf("\n\t ENTER DATA FOR 1ST NODE : ");
            scanf("%d",&p->i);
            p->next=first;
            first=p;
            cnt++;
        }
        if(user==4)

```



```

    {
        p=(struct node*)malloc(sizeof(struct
node));
        printf("\n\t ENTER DATA FOR LAST
NODE : ");
        scanf("%d",&p->i);
        temp=first;
        while(temp->next!=0)
        {
            temp=temp->next;
        }
        temp->next=p;
        p->next=0;
        cnt++;
    }
    if(user==3)
    {
        printf("\n\t ENTER ANY ADDRESS BETWEEN 1 and
%d : ",cnt);
        scanf("%d",&add);

        t=1;
        p=first;
        while(t!=add)
        {
            p=p->next;
            t++;
        }
        temp=(struct node*)malloc(sizeof(struct node));
        printf("\n\t ENTER DATA FOR NODE : ");
        scanf("%d",&temp->i);
        temp->next=p->next;
        p->next=temp;
        cnt++;
    }
    if(user==5)
    {
        p=first;
        first=p->next;
        free(p);
    }
    if(user==6)
    {
        p=first;
        while(p->next->next!=0)
        {
            p=p->next;
        }
        p->next=0;
        free(p->next->next);
    }

```

```

    }
    if(user==8)
    {
        p=first;
        while(p!=0)
        {
            printf("\n\t %d",p->i);
            p=p->next;
        }
    }
    if(user==7)
    {
        printf("\n\t ENTER ANY ADDRESS BETWEEN 1 and
%d: ",cnt);
        scanf("%d",&add);

        t=1;
        p=first;
        while(t<add-1)
        {
            p=p->next;
            t++;
        }
        temp=p->next;
        p->next=temp->next;
        free(temp);
        cnt--;
    }
    printf("\n\t 1.CREATION");
    printf("\n\t 2.INSERT AT STARTING");
    printf("\n\t 3.INSERT AT MIDDLE(USER'S
CHOICE)");
    printf("\n\t 4.INSERT AT END");
    printf("\n\t 5.DELETE 1ST NODE");
    printf("\n\t 6.DELETE LAST NODE");
    printf("\n\t 7.DELETE MIDDLE NODE(USER'S
CHOICE)");
    printf("\n\t 8.DISPLAY");
    printf("\n\t 10.EXIT");
    printf("\n\t Enter your choice: ");
    scanf("%d",&user);
    }
    getch();
}

```

Output: (Execute Yourself.)

II. A C - Program to implement doubly Linked Lists.

```
#include<stdio.h>
#include<stdlib.h>
/* Linked list structure */
struct linkedlist {
    struct linkedlist *prev;
    int data;
    struct linkedlist *next;
} *node = NULL, *first = NULL, *last = NULL,
*node1 = NULL, *node2 = NULL;
/* Function for create/insert node at the
beginning of Linked list */
void insert_beginning() {
    /* Dynamic memory allocation */
    node = (struct linkedlist*)malloc(sizeof(struct
linkedlist));
    printf("Enter value for the node:\n");
    scanf("%d",&node->data);
    if(first == NULL) {
        node->prev = NULL;
        node->next = NULL;
        first = node;
        last = node;
        printf("Linked list Created!\n");
    }
    else {
        node->prev = NULL;
        first->prev = node;
        node->next = first;
        first = node;
        printf("Data Inserted at the
beginning of the Linked list!\n");
    }
}

/* Function for create/insert node at the end of
Linked list */
void insert_end() {
    /* Dynamic memory allocation */
    node = (struct
linkedlist*)malloc(sizeof(struct linkedlist));
    printf("Enter value for the node:\n");
    scanf("%d",&node->data);
    if(first == NULL) {
        node->prev = NULL;
        node->next = NULL;
        first = node;
```

```
        last = node;
        printf("Linked list Created!\n");
    }
    else {
        node->next = NULL;
        last->next = node;
        node->prev = last;
        last = node;
        printf("Data Inserted at the end of
the Linked list!\n");
    }
}

/* Function for Display Linked list */
void display() {
    node = first;
    printf("List of data in Linked list in Ascending
order!\n");
    while(node != NULL) {
        printf("%d\n",node->data);
        node = node->next;
    }
    node = last;
    printf("List of data in Linked list in Descending
order!\n");
    while(node != NULL) {
        printf("%d\n",node->data);
        node = node->prev;
    }
}

/* Function for create/insert node in Linked list */
void delete() {
    int count = 0, number, i;
    node = node1 = node2 = first;
    for(node = first; node != NULL; node = node-
>next)
        printf("Enter value for the node:\n");
    count++;
    display();
    printf("\n%d nodes available here!\n",
count);
    printf("Enter the node number which you
want to delete:\n");
    scanf("%d", &number);
    if(number != 1) {
        if(number < count && number > 0) {
            for(i = 2; i <= number; i++)
                node = node->next;
            for(i = 2; i <= number-1; i++)
                node1 = node1->next;
```

```

        for(i = 2; i <= number+1; i++)
            node2 = node2->next;
            node2->prev = node1;
            node1->next = node2;
            node->prev = NULL;
            node->next = NULL;
            free(node);
    }
    else if(number == count) {
        node = last;
        last = node->prev;
        last->next = NULL;
        free(node);
    }
    else
        printf("Invalid node
number!\n");
    }
    else {
        node = first;
        first = node->next;
        first->prev = NULL;
        free(node);
    }
    printf("Node has been deleted
successfully!\n");
}
int main() {
    int op = 0;
    while(op != 5) {
        printf("1. Insert at the beginning\n2. Insert at the
end\n3. Delete\n4. Display\n5. Exit\n");
        printf("Enter your choice:\n");
        scanf("%d", &op);
        switch(op) {
            case 1:
                insert_beginning();
                break;
            case 2:
                insert_end();
                break;
            case 3:
                delete();
                break;
            case 4:
                display();
                break;
            case 5:
                printf("Bye Bye!\n");

```

```

        exit(0);
        break;
        default:
            printf("Invalid choice!\n");
        }
    }
    return 0; }

```

Output: (Execute Yourself.)

5. Implementation and Balancing of AVL tree. (C ++ Program)

```

#include<iostream.h>
#include<conio.h>
#include<stdlib.h>
struct node
{
    /* defining node */
    int element;
    node *left;
    node *right;
    int height;
};
typedef struct node *nodeptr;
template<class T>
class avltree
{
    public:
        void insert(T,nodeptr &); /* defining
functions */
        void del(T,nodeptr &);
        void find(T,nodeptr &);
        int avlheight(nodeptr);
};

template<class T>
void avltree<T>::insert(T x,nodeptr &p)
{
    if(p==NULL)
    {
        p=new node;
        p->element=x; /* for first element */
        p->left=NULL;
        p->right=NULL;
        p->height=0;
        if(p==NULL)
            cout<<"Out of Space";
    }
    else
    {

```

```

        if(x<p->element)
        {
            /* traversing for insertion */
            insert(x,p->left);
        }
        if((avlheight(p->left)-avlheight(p->right))==2)
        {
            if(x<p->left->element);
            p=singlrotaleft(p);
        }
        else
        {
            p=dblrotaleft(p);
        }
    }
    else if(x>p->element)
    {
        insert(x,p->right);
    }
    if((avlheight(p->right)-avlheight(p->left))==2)
    {
        if(x<p->right->element);
        p=singlrotarget(p);
    }
    else
    {
        p=dblrotarget(p);
    }
}
else
    cout<<"Element Already
Exists"<<x;
}
int m,n,d;
m=avlheight(p->left);
n=avlheight(p->right);
d=max(m,n);
p->height=d+1;
}
template<class T>

void avltree<T>::del(T x,nodeptr &p)
{
    nodeptr d;
    if(p==NULL)
        cout<<"Element Not Found";
    else if(x<p->element)
        del(x,p->left);    /* searching element to
delete */
    else if(x>p->element)
        del(x,p->right);
    else if((p->left==NULL)&&(p->right==NULL))
    {
        d=p;
        free(d);
        p=NULL;
        cout<<"Element Deleted";
    }
}

```

```

    }
    else if(p->left==NULL)
    {
        d=p;
        free(d);    /* freeing the memory */
        p=p->right;
        cout<<"Element Deleted";
    }
    else if(p->right==NULL)
    {
        d=p;
        p=p->left;
        free(d);
        cout<<"Element Deleted";
    }
    else
        p->element=deletemin(p->right);
}
template<class T>
void avltree<T>::find(T x,nodeptr &p)
{
    /* finding an element */

    if(p==NULL)
        cout<<"Element Not Found";
    else
        if(x<p->element)
            find(x,p->left);
        else
            if(x>p->element)
                find(x,p->right);
            else
                cout<<"Element found";
    }
}
int avltree<T>::avlheight(nodeptr p)
{
    int t;
    if (P==Null)
        Return -1
    else
    {
        T = p->height
        Return t;
    }
}
int main()
{
    clrscr();
    Nodeptr root,root1,min,max;
    int a,choice,findele,delete,leftele,rightele,flag;
    Char ch='y';
    AV Ltree<int>bst;
}

```

Output: (Execute Yourself.)

```
#include<stdio.h>
#include<conio.h>
void mergesort(int a[],int low,int high);
void combine(int a[],int low,int mid,int high);
void main()
{
    int a[10],n,i;
    clrscr();
    printf("Enter n value for array elements:");
    scanf("%d",&n);
    printf("Enter array elements:\n");
    for(i=0;i<n;i++)
    {
        scanf("%d",&a[i]);
    }
    mergesort(a,0,n-1);
    printf("Sorted array elements are:\n");
    for(i=0;i<n;i++)
    {
        printf("\n%d",a[i]);
    }
    getch();
}

void mergesort(int a[],int low,int high)
{
    int mid;
    if(low<high)
    {
        mid=(low+high)/2;
        mergesort(a,low,mid);
        mergesort(a,mid+1,high);
        combine(a,low,mid,high);
    }
}

void combine(int a[],int low,int mid,int high)
{
    int b[10],i=low,h=low,j=mid+1,k;
    while(h<=mid && j<=high)
    {
        if(a[h]<=a[j])
        {
            b[i]=a[h];
            h=h+1;
        }
    }
}
```

```

else
{
    b[i]=a[j];
    j=j+1;
}
i=i+1;
}
if(h>mid)
{
    for(k=j;k<=high;k++)
    {
        b[i]=a[k];
        i=i+1;
    }
}
else
{
    for(k=h;k<=mid;k++)
    {
        b[i]=a[k];
        i=i+1;
    }
}
for(k=low;k<=high;k++)
{
    a[k]=b[k];
}
}

```

Output: (Execute Yourself.)

7. Implementation of Search: Sequential, Tree and Binary.

I. Sequential Search or Linear Search (using recursion and non-recursion)

```

#include<stdio.h>
#include<conio.h>
void lsnr(int a[],int n,int x);
void lsr(int a[],int n,int x);
main()
{
    int a[10],n,i,x,ch;
    clrscr();
    printf("Enter the no.of elements of an array:\n");
    scanf("%d",&n);
    printf("Enter the array elements:\n");
    for(i=0;i<n;i++)

```

```

{
    scanf("%d",&a[i]);
}
printf("Enter the searching element:\n");
scanf("%d",&x);
printf("enter the choice:\n 1.nonrecursion \n
2.recursion\n");
scanf("%d",&ch);
switch(ch)
{
    case 1:printf("non recursion process:\n");
            lsnr(a,n,x);
            break;
    case 2:printf("recursion process:\n");
            lsr(a,n,x);
            break;
}
getch();
}
void lsnr(int a[],int n,int x)
{
    int i,pos=0;
    for(i=0;i<n;i++)
    {
        if(x==a[i])
        {
            printf("the element is found at %d",i);
            pos=1;
            break;
        }
    }
    if(pos==0)
        printf("The searching element is not found");
}
void lsr(int a[],int n,int x)
{
    int i,pos=0;
    if(x==a[n])
    {
        printf("the element is found at %d",n);
        pos=1;
    }
    else
    {
        if(pos==0 && n==0)
            printf("element is not found");
        else
            lsr(a,n-1,x);
    }
}
}

```

II. Implementation Binary Search (using recursion and non-recursion)

```
#include<stdio.h>
#include<conio.h>
void bsnr(int[],int,int,int);
void bsr(int[],int,int,int);
main()
{
    int a[10],n,i,x,low,high,ch;
    clrscr();
    printf("enter n values for array:\n");
    scanf("%d",&n);
    printf("enter array elements:\n");
    for(i=0;i<n;i++)
    {
        scanf("%d",&a[i]);
    }
    printf("enter searching element:\n");
    scanf("%d",&x);
    printf("enter ur choice:\n 1.nonrecursion \n 2.recursion\n");
    scanf("%d",&ch);

    switch(ch)
    {
        case 1:printf("nonrecursion process:\n");
                bsnr(a,0,n-1,x);
                break;
        case 2:printf("recursion process:\n");
                bsr(a,0,n-1,x);
                break;
    }
    getch();
}

void bsnr(int a[],int low,int high,int x)
{
    int pos=0,mid;
    while(low<=high)
    {
        mid=(low+high)/2;
        if(x==a[mid])
        {
            printf("element is found at position of %d\n",mid);
            pos=1;
            break;
        }
    }
}
```

```

    }
    else if(x<a[mid])
    {
        high=mid-1;
    }
    else
    {
        low=mid+1;
    }
}
if(pos==0)
{
    printf("element not found");
}
}

void bsr(int a[],int low,int high,int x)
{
    int mid,pos=0;
    if(low<=high)
    {
        mid=(low+high)/2;
        if(x==a[mid])
        {
            printf("element is found at position %d\n",mid);
            pos=1;
        }
        else if(x<a[mid])
        {
            bsr(a,low,mid-1,x);
        }
        else
        {
            bsr(a,mid+1,high,x);
        }
    }
    else
    printf("element is not found:");
}
}
```

Output: (Execute Yourself.)

III. Implementation Binary Search (using recursion and non-recursion) (C++ Program)

```
#include<iostream.h>
#include<process.h>
class bstree
{
    struct node
    {
        int data;
        node *left,*right;
    }*root,*p,*prev;
public:bstree()
    {
        root=NULL;
    }
    void insert();
    void search();
    void deletion();
    void display();
    void preorder(node *);
};

void bstree::insert()
{
    node *temp=new node;
    cout<<"\n enter element to insert:";
    cin>>temp->data;
    temp->left=NULL;
    temp->right=NULL;
    prev=NULL;
    p=root;
    while(p!=NULL)
    {
        prev=p;
        if(temp->data<p->data)
            p=p->left;
        else
            p=p->right;
    }
    if(prev==NULL)
    {
        root=temp;
    }
    else
    {
        if(temp->data<prev->data)
            prev->left=temp;
        else
            prev->right=temp;
    }
}
```

```
else
    prev->right=temp;
} }
void bstree::search()
{
    int x;
    cout<<"\n enter the element to search:";
    cin>>x;
    node *s;
    s=root;
    int count=1;
    while(s!=NULL)
    {
        if(s->data==x)
        {
            cout<<"\n element is found at
location:"<<s<<"\t"<<count;
            return;
        }

        else
        {
            if(x<s->data)
                s=s->left;
            count++;
            else
                s=s->right;
            count++;
        }
    }
    cout<<"\n element not found";
    return;
}

void bstree::deletion()
{
    node *p,*prev,*newnode,*ptr,*temp;
    int key;
    char v;
    cout<<"\n enter the element to be deleted:";
    cin>>key;
    prev=NULL;
    p=root;
    while((p->data!=key)&&(p!=NULL))
    {
        prev=p;
        if(key<p->data)
        {
            p=p->left;
            v='l';
        }
        else
        {
            p=p->right;
            v='r';
        }
    }
    if(p==NULL)
        return;
    if(v=='l')
    {
        if(prev->left==p)
            prev->left=NULL;
        else
            prev->left=p->left;
    }
    else
    {
        if(prev->right==p)
            prev->right=NULL;
        else
            prev->right=p->right;
    }
    newnode=new node;
    newnode->data=p->data;
    newnode->left=p->left;
    newnode->right=p->right;
    delete p;
    p=newnode;
    if(v=='l')
        prev->left=p;
    else
        prev->right=p;
}
```



```

{
    p=p->right;
    v='r';
} }
if(p==NULL)
{
    cout<<"\n node to be deleted not found";
}
ptr=p;
if(ptr->left==NULL)
{
    newnode=ptr->right;
    delete ptr;
}
else if(ptr->right==NULL)
{
    newnode=ptr->left;
    delete ptr;
}
else if((ptr->left!=NULL)&&(ptr->right!=NULL))
{
    temp=ptr->right;
    while(temp->left!=NULL)
    {
        temp=temp->left;
    }
    temp->left=ptr->left;
    newnode=ptr->right;
    delete ptr;
}
if(prev==NULL)
{
    root=newnode;
}
if(v=='l')
    prev->left=newnode;(ptr->left=newnode)
else if(v=='r')
    prev->right=newnode;(ptr->right=newnode)
}

```

```

void bstree::display()
{
    if(root==NULL)
        cout<<"\n Tree is not created";
    else
    {
        cout<<"\n The tree is:";
    }
    preorder(root);
}

```

```

}
}
void bstree::preorder(node *temp)
{
    if(temp!=NULL)
    {
        cout<<"t"<<temp->data;
        preorder(temp->left);
        preorder(temp->right);
    } }

void main()
{
    clrscr();
    int cho;
    bstree b;
    while(1)
    {
        cout<<"\n 1.insert \n 2.search \n 3.deletion \n
4.display \n 5.exit";
        cout<<"\n enter ur choice:";
        cin>>cho;
        switch(cho)
        {
            case 1:b.insert();
                break;
            case 2:b.search();
                break;
            case 3:b.deletion();
                break;
            case 4:b.display();
                break;
            case 5:exit(1);
                break;
            default:cout<<"\n ur choice is wrong";
        }
    }
}

```

Output: (Execute Yourself.)

8. To Implement Operations on Graph. (C++ Program)

```
#include<iostream.h>
#include<conio.h>
#include<process.h>
#define max 10
class graph
{
private:
    int g[max][max];
public:
    graph(); //constructor
    void create();
    void insertvertex();
    void deletevertex();
    void findvertex();
    void insertedge();
    void deleteedge();
    void display();
};

graph::graph()
{
    for(int i=0;i<max;i++)
        for(int j=0;j<max;j++)
            g[i][j]=0;
}

void graph::create()
{
    int v1,v2;
    char ans='y';
    do
    {
        cout<<"\n Enter the vertex v1 and v2:";
        cin>>v1>>v2;
        g[v1][v2]=1;
        g[v2][v1]=1;
        cout<<"\n Do you want to insert more vertex?";
        ans=getch();
    }while(ans=='y');
}

void graph::insertvertex()
{
    int v1,v2;
    char ans='y';
    cout<<"\n Enter the vertex to be inserted:";
```

```
        cin>>v1;
        do
        {
            cout<<"\n Enter neighbouring vertex:";
            cin>>v2;
            g[v1][v2]=1;
            g[v2][v1]=1;
            cout<<"\n more neighbouring vertex?";
            ans=getch();
        }while(ans=='y');
    }

    void graph::deletevertex()
    {
        int i,v;
        cout<<"\n Enter the vertex to be deleted:";
        cin>>v;
        for(i=0;i<max;i++)
        {
            g[v][i]=0;
            g[i][v]=0;
        }
        cout<<"\n The vertex is deleted:";
    }

    void graph::findvertex()
    {
        int i,v;
        int flag=1;
        cout<<"\n Enter the vertex to be searched in the graph:";
        cin>>v;
        for(i=0;i<max;i++)
        {
            if(g[v][i]==1)
            {
                flag=0;
                cout<<"\n neighbouring vertex is:"<<i;
            }
        }
        if(flag==1)
            cout<<"\n vertex is not present in the graph";
    }

    void graph::insertedge()
    {
        int v1,v2;
        cout<<"\n Enter the edge to be inserted by v1 and v2:";
        cin>>v1>>v2;
        g[v1][v2]=1;
```

```

    g[v2][v1]=1;
}
void graph::deleteedge()
{
    int v1,v2;
    cout<<"\n Enter the edge to be deleted by v1 and v2:";
    cin>>v1>>v2;
    g[v1][v2]=0;
    g[v2][v1]=0;
}
void graph::display()
{
    cout<<"\n";
    for(int i=0;i<max;i++)
    {
        for(int j=0;j<max;j++)
        {
            cout<<" "<<g[i][j];
        }
        cout<<"\n";
    } }

void main()
{
    graph g; //object creation
    int choice;
    char ch='y';
    clrscr();
    cout<<"\n Program for graph operations";
    g.create();
    g.display();
    do
    {
        cout<<"\n Enter ur choice:";
        cin>>choice;
        cout<<"\n 1.insertvertex \n 2.deletevertex \n
3.findvertex \n 4.insertedge \n 5.deleteedge \n
6.exit";
        switch(choice)
        {
            case 1:g.insertvertex();
                    g.display();
                    break;
            case 2:g.deletevertex();
                    g.display();
                    break;
            case 3:g.findvertex();
                    g.display();
                    break;

```

```

            case 4:g.insertedge();
                    g.display();
                    break;
            case 5:g.deleteedge();
                    g.display();
                    break;
            case 6:exit(0);
        }
        cout<<"\n Do you want to go main menu?";
        ch=getch();
    } while(ch=='y'); }

```

9. Implementation of Heap sort (C- Program)

```

#include<stdio.h>
#include<conio.h>
void makeheap(int a[],int n);
void heapsort(int a[],int n);
main()
{
    int a[10],n,i;
    clrscr();
    printf("Enter n value for array:\n");
    scanf("%d",&n);
    printf("Enter array elements:\n");
    for(i=0;i<n;i++)
    {
        scanf("%d",&a[i]);
    }
    makeheap(a,n);
    printf("The maxheap array elements are:\n");
    for(i=0;i<n;i++)
    {
        printf("%d\n",a[i]);
    }
    heapsort(a,n);
    printf("The sorted array elements are:\n");
    for(i=0;i<n;i++)
    {
        printf("%d\n",a[i]);
    }
    getch(); }

void makeheap(int a[],int n)
{
    int i,j,f,temp;
    for(i=1;i<n;i++)
    {

```

```

temp=a[i];
j=i;
f=(j-1)/2;
while(a[f]<temp && j>0)
{
    a[j]=a[f];
    j=f;
    f=(j-1)/2;
}
a[j]=temp;
} }
void heapsort(int a[],int n)
{
    int i,j,k,temp;
    for(i=n-1;i>0;i--)
    {
        temp=a[i];
        a[i]=a[0];
        k=0;
        if(i==1)
            j=-1;
        else
            j=1;
        if(i>2 && a[2]>a[1])
            j=2;
        while(j>0 && temp<a[j])
        {
            a[k]=a[j];
            k=j;
            j=2*k+1;
            if(j+1<=i-1 && a[j]<a[j+1])
                j++;
            if(j>i-1)
                j=-1;
        }
        a[k]=temp;
    } }

```

Output: (Execute Yourself.)

10. Implementation of Hashing. (C++ Program)

```

#include<iostream.h>
#include<conio.h>
#include<stdlib.h>
#include<math.h>
#include<process.h>
#define max 10

```

```

class hash
{
    private:
        struct dct
        {
            int k;
            int val;
        }a[max];
    public:
        void init();
        void insert(int,int,int);
        int divmethod(int);
        int mulmethod(int);
        void display();
        void size();
};

void hash::init()
{
    for(int i=0;i<max;i++)
    {
        a[i].k=-1;
        a[i].val=-1;
    }
}

void hash::insert(int index,int key,int value)
{
    int i,flag=0,count=0;
    if(a[index].k==-1)//if the location indicated by
    hash key is empty
    {
        a[index].k=key;
        a[index].val=value;
    }
    else
    {
        i=0;
        while(i<max)
        {
            if(a[i].k!=-1)
                count++;
            i++;
        }
        if(count==max)//checking for the hash full
        {
            cout<<"\n Hash Table is full";
        }
        for(i=index+1;i<max;i++)//moving linearly down
        if(a[i].k==-1)//searching for empty location
        {
            a[i].k=key;

```

```

        a[i].val=value; //placing the number at empty
location
        flag=1;
        break;
    }
    /*From key position to the end of array we have
searched empty location
    and now we want to check empty location in
the upper part of the array*/
    for(i=0;i<index&&flag==0;i++)//array from 0th
to keyth location will be searched
        if(a[i].k==key)
        {
            a[i].k=key;
            a[i].val=value;
            flag=1;
            break;
        }
    } //outer else
} //end
int hash::divmethod(int num)
{
    int record;
    record=num%10;
    return record;
}
int hash::mulmethod(int num)
{
    int record;
    int p=1;
    double a=0.6180;
    record=floor(num*p*a);
    return record;
}
void hash::display()
{
    int i;
    cout<<"\n The Hash Table is...\n";
    cout<<"\n -----";
    for(i=0;i<max;i++)
    {
        cout<<"\n "<<i<<" "<<a[i].k<<" "<<a[i].val;
    }
    cout<<"\n -----";
}
void hash::size()
{
    int i,len=0;
    for(i=0;i<max;i++)
    {

```

```

        if(a[i].k!=-1)
            len++;
    }
    cout<<"\n The size of the dictionary is:"<<len;
}
void main()
{
    clrscr();
    int index,key,value,choice;
    hash h;//object creation
    cout<<"\n The dictionary funtions using hashing
is:";
    h.init();
    while(1)
    {
        cout<<"\n 1.insertion(divmethod) \n
2.insertion(mulmethod) \n 3.display \n 4.size \n
5.exit";
        cout<<"\n enter ur choice:";
        cin>>choice;
        switch(choice)
        {
            case 1:cout<<"\n Enter the key:";
                    cin>>key;
                    cout<<"\n Enter the value:";
                    cin>>value;
                    index=h.divmethod(value);
                    h.insert(index,key,value);
                    break;
            case 2:cout<<"\n Enter the key:";
                    cin>>key;
                    cout<<"\n Enter the value:";
                    cin>>value;
                    index=h.mulmethod(key);
                    h.insert(index,key,value);
                    break;
            case 3:h.display();
                    break;
            case 4:h.size();
                    break;
            case 5:exit(0);
        }
    } getch(); }

```

Output: (Execute Yourself.)

"Talk is cheap. Show me the code."

***** *THANK YOU* *****

Some Important Interview Questions

1. What is data-structure?

Data structure is a way of defining, storing & retrieving of data in a structural & systematic way. A data structure may contain different type of data items

2. What are various data-structures available?

Data structure availability may vary by programming languages. Commonly available data structures are list, arrays, stack, queues, graph, tree etc.

3. What is algorithm?

Algorithm is a step by step procedure, which defines a set of instructions to be executed in certain order to get the desired output.

4. Why we need to do algorithm analysis?

A problem can be solved in more than one ways. So, many solution algorithms can be derived for a given problem. We analyze available algorithms to find and implement the best suitable algorithm.

5. What are the criteria of algorithm analysis?

An algorithm are generally analyzed on two factors – time and space. That is, how much execution time and how much extra space required by the algorithm.

6. What is asymptotic analysis of an algorithm?

Asymptotic analysis of an algorithm refers to defining the mathematical foundation /framing of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case and worst case scenario of an algorithm

7. What are asymptotic notations?

Asymptotic analysis can provide three levels of mathematical binding of execution time of an algorithm – Best case is represented by $O(n)$ notation, Worst case is represented by $\Omega(n)$ notation, Average case is represented by $\Theta(n)$ notation.

8. What is linear data structure?

A linear data-structure has sequentially arranged data items. The next time can be located in the next memory address. It is stored and accessed in a sequential manner. Array and list are example of linear data structure

9. What are common operations that can be performed on a data-structure?

The following operations are commonly performed on any data-structure –

Insertion – adding a data item

Deletion – removing a data item

Traversal – accessing and/or printing all data items

Searching – finding a particular data item

Sorting – arranging data items in a pre-defined sequence

10. Briefly explain the approaches to develop algorithms.

There are three commonly used approaches to develop algorithms –

Greedy Approach – finding solution by choosing next best option

Divide and Conquer – diving the problem to a minimum possible sub-problem and solving them independently

Dynamic Programming – diving the problem to a minimum possible sub-problem and solving them combined.

11. Give some examples greedy algorithms.

The below given problems find their solution using greedy algorithm approach –

Travelling Salesman Problem, Prim's Minimal Spanning Tree Algorithm, Kruskal's Minimal Spanning Tree Algorithm, Dijkstra's Minimal Spanning Tree Algorithm, Graph - Map Coloring, Graph - Vertex Cover, Knapsack Problem, Job Scheduling Problem.

12. What are some examples of divide and conquer algorithms?

The below given problems find their solution using divide and conquer algorithm approach – Merge Sort, Quick Sort, Binary Search, Strassen's Matrix Multiplication, Closest pair (points)

13. What are some examples of dynamic programming algorithms?

The below given problems find their solution using divide and conquer algorithm approach – Fibonacci number series, Knapsack problem, Tower of Hanoi, All pair shortest path by Floyd-Warshall, Shortest path by Dijkstra, Project scheduling

14. What is a linked-list?

A linked-list is a list of data-items connected with links i.e. pointers or references. Most modern high-level programming language does not provide the feature of directly accessing memory location, therefore, linked-list are not supported in them or available in form of inbuilt functions

15. What is stack?

In data-structure, stack is an Abstract Data Type (ADT) used to store and retrieve values in Last In First Out method.

16. Why do we use stacks?

Stacks follows LIFO method and addition and retrieval of a data item takes only $O(n)$ time. Stacks are used where we need to access data in the reverse order or their arrival. Stacks are used commonly in recursive function calls, expression parsing, depth first traversal of graphs etc.

17. What operations can be performed on stacks?

The below operations can be performed on a stack –

push() – adds an item to stack, **pop()** – removes the top stack item, **peek()** – gives value of top item without removing it, **isempty()** – checks if stack is empty, **isfull()** – checks if stack is full.

18. What is a queue in data-structure?

Queue is an abstract data structure, somewhat similar to stack. In contrast to stack, queue is opened at both ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

19. Why do we use queues?

As queues follows FIFO method, they are used when we need to work on data-items in exact sequence of their arrival. Every operating system maintains queues of various processes. Priority queues and breadth first traversal of graphs are some examples of queues.

20. What operations can be performed on Queues?

The below operations can be performed on a stack –

enqueue() – adds an item to rear of the queue, **dequeue()** – removes the item from front of the queue, **peek()** – gives value of front item without removing it, **isempty()** – checks if stack is empty, **isfull()** – checks if stack is full.

21. What is linear searching?

Linear search tries to find an item in a sequentially arranged data type. These sequentially arranged data items known as array or list are accessible in incrementing memory location. Linear search compares expected data item with each of data items in list or array. The average case time complexity of linear search is $O(n)$ and worst case complexity is $O(n^2)$. Data in target arrays/lists need not to be sorted.

22. What is binary search?

A binary search works only on sorted lists or arrays. This search selects the middle which splits the entire list into two parts. First the middle is compared. This search first compares the target value to the mid of the list. If it is not found, then it takes decision on whether.

23. What is bubble sort and how bubble sort works?

Bubble sort is comparison based algorithm in which each pair of adjacent elements is compared and elements are swapped if they are not in order. Because the time complexity is $O(n^2)$, it is not suitable for large set of data

24. Tell me something about 'insertion sort'?

Insertion sort divides the list into two sub-lists, sorted and unsorted. It takes one element at time and finds it appropriate location in sorted sub-list and insert there. The output after insertion is a sorted sub-list. It iteratively works on all the elements of unsorted sub-list and inserts them to sorted sub-list in order.

25. What is selection sort?

Selection sort is in-place sorting technique. It divides the data set into two sub-lists: sorted and unsorted. Then it selects the minimum element from unsorted sub-list and places it into the sorted list. This iterates unless all the elements from unsorted sub-list are consumed into sorted sub-list.

26. How insertion sort and selection sorts are different?

Both sorting techniques maintains two sub-lists, sorted and unsorted and both take one element at a time and places it into sorted sub-list. Insertion sort works on the current element in hand and places it in the sorted array at appropriate location maintaining the properties of insertion sort. Whereas, selection sort searches the minimum from the unsorted sub-list and replaces it with the current element in hand.

27. What is merge sort and how it works?

Merge sort is sorting algorithm based on divide and conquer programming approach. It keeps on dividing the list into smaller sub-list until all sub-list has only 1 element. And then it merges them in a sorted way until all sub-lists are consumed. It has run-time complexity of $O(n \log n)$ and it needs $O(n)$ auxiliary space.

28. What is shell sort?

Shell sort can be said a variant of insertion sort. Shell sort divides the list into smaller sublist based on some gap variable and then each sub-list is sorted using insertion sort. In best cases, it can perform upto $O(n \log n)$.

29. How quick sort works?

Quick sort uses divide and conquer approach. It divides the list in smaller 'partitions' using 'pivot'. The values which are smaller than the pivot are arranged in the left partition and greater values are arranged in the right partition. Each partition is recursively sorted using quick sort.

30. What is a graph?

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as vertices, and the links that connect the vertices are called edges.

31. How depth first traversal works?

Depth First Search algorithm(DFS) traverses a graph in a depth ward motion and uses a stack to remember to get the next vertex to start a search when a dead end occurs in any iteration.

32. How breadth first traversal works?

Breadth First Search algorithm(BFS) traverses a graph in a breadth wards motion and uses a queue to remember to get the next vertex to start a search when a dead end occurs in any iteration.

33. What is a tree?

A tree is a minimally connected graph having no loops and circuits.

34. What is a binary tree?

A binary tree has a special condition that each node can have two children at maximum.

35. What is a binary search tree?

A binary search tree is a binary tree with a special provision where a node's left child must have value less than its parent's value and node's right child must have value greater than its parent value.

36. What is tree traversal?

Tree traversal is a process to visit all the nodes of a tree. Because, all nodes are connected via edges (links) we always start from the root (head) node. There are three ways which we use to traverse a tree –

In-order Traversal, Pre-order Traversal, Post-order Traversal

37. What is an AVL Tree?

AVL trees are height balancing binary search tree. AVL tree checks the height of left and right sub-trees and assures that the difference is not more than 1. This difference is called Balance Factor.

$\text{BalanceFactor} = \text{height}(\text{left-sutree}) - \text{height}(\text{right-sutree})$

38. What is a spanning tree?

A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. A spanning tree does not have cycles and it cannot be disconnected.

39. How many spanning trees can a graph has?

It depends on how connected the graph is. A complete undirected graph can have maximum n^{n-1} number of spanning trees, where n is number of nodes

40. How Kruskal's algorithm works?

This algorithm treats the graph as a forest and every node as an individual tree. A tree connects to another only and only if it has least cost among all available options and does not violate MST properties.

41. How Prim's algorithm finds spanning tree?

Prim's algorithm treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph.

42. What is a minimum spanning tree (MST)?

In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight that all other spanning trees of the same graph.

43. What is a heap in data structure?

Heap is a special balanced binary tree data structure where root-node key is compared with its children and arranged accordingly. A min-heap, a parent node has key value less than its children and a max-heap parent node has value greater than its children.

44. What is a recursive function?

A recursive function is one which calls itself, directly or calls a function that in turn calls it. Every recursive function follows the recursive properties – base criteria where function stops calling itself and progressive approach where the function tries to meet the base criteria in each iteration.

45. What is tower of Hanoi?

Tower of Hanoi is a mathematical puzzle which consists of three towers (pegs) and more than one ring. All rings are of different size and stacked upon each other where the large disk is always below the small disk. The aim is to move the tower of disk from one peg to another, without breaking its properties.

46. What is hashing?

Hashing is a technique to convert a range of key values into a range of indexes of an array. By using hash tables, we can create associative data storage where data index can be found by providing its key values.

47. Which sorting algorithm is considered the fastest?

There are many types of sorting algorithms: quick sort, bubble sort, quick sort, radix sort, merges sort, etc. Not one can be considered the fastest because each algorithm is designed for a particular data structure and data set. It would depend on the data set that you would want to sort.

48. What are multidimensional arrays?

Multidimensional arrays make use of multiple indexes to store data. It is useful when storing data that cannot be represented using a single dimensional indexing, such as data representation in a board game, tables with data stored in more than one column.

49. Are linked lists considered linear or non-linear data structures?

It actually depends on where you intend to apply linked lists. If you based it on storage, a linked list is considered non-linear. On the other hand, if you based it on access strategies, then a linked list is considered linear.

50. How does dynamic memory allocation help in managing data?

Aside from being able to store simple structured data types, dynamic memory allocation can combine separately allocated structured blocks to form composite structures that expand and contract as needed.

51. Differentiate NULL and VOID.

Null is actually a value, whereas Void is a data type identifier. A variable that is given a Null value simply indicates an empty value. Void is used to identify pointers as having no initial size.

52. What is the advantage of the heap over a stack?

Basically, the heap is more flexible than the stack. That's because memory space for the heap can be dynamically allocated and de-allocated as needed. However, memory of the heap can at times be slower when compared to that stack.

53. What is Data abstraction?

Data abstraction is a powerful tool for breaking down complex data problems into manageable chunks. This is applied by initially specifying the data objects involved and the operations to be performed on these data objects without being overly concerned with how the data objects will be represented and stored in memory.

54. How do signed and unsigned numbers affect memory?

In the case of signed numbers, the first bit is used to indicate whether positive or negative, which leaves you with one bit short. With unsigned numbers, you have all bits available for that number. The effect is best seen in the number range (unsigned 8 bit number has a range 0-255, while 8-bit signed number has a range -128 to +127).

55. What are dynamic data structures?

Dynamic data structures are structures that expand and contract as a program runs. It provides a flexible means of manipulating data because it can adjust according to the size of the data.

56. In what data structures are pointers applied?

Pointers that are used in linked list have various applications in data structure. Data structures that make use of this concept include the Stack, Queue, Linked List and Binary Tree.

57. What is the minimum number of queues needed when implementing a priority queue?

The minimum number of queues needed in this case is two. One queue is intended for sorting priorities while the other queue is intended for actual storage of data

58. Differentiate linear from nonlinear data structure.

Linear data structure is a structure wherein data elements are adjacent to each other. Examples of linear data structure include arrays, linked lists, stacks and queues. On the other hand, non-linear data structure is a structure wherein each data element can connect to more than two adjacent data elements. Examples of nonlinear data structure include trees and graphs.

59. What is Huffman's algorithm?

Huffman's algorithm is associated in creating extended binary trees that has minimum weighted path lengths from the given weights. It makes use of a table that contains frequency of occurrence for each data element.

60. How to implement a stack using queue?

A stack can be implemented using two queues. Let stack to be implemented be 's' and queues used to implement be 'q1' and 'q2'. Stack 's' can be implemented in two ways:

Method 1 (By making push operation costly)

Method 2 (By making pop operation costly)

61. How to implement a queue using stack?

A queue can be implemented using two stacks. Let queue to be implemented be q and stacks used to implement q be stack1 and stack2. Queue can be implemented in two ways:

Method 1 (By making enqueue operation costly)

Method 2 (By making dequeue operation costly)

62. Which Data Structure Should be used for implementing LRU cache?

We use two data structures to implement an LRU Cache.

Queue which is implemented using a doubly linked list. The maximum size of the queue will be equal to the total number of frames available (cache size). The most recently used pages will be near front end and least recently pages will be near rear end.

A Hash with page number as key and address of the corresponding queue node as value.

63. What is time complexity of Binary Search?

Time complexity of binary search is $O(\log n)$.

64. When does the worst case of Quicksort occur?

In quicksort, we select a pivot element, and then partition the given array around the pivot element by placing pivot element at its correct position in sorted array. The worst case of quicksort occurs when one part after partition contains all elements and other part is empty. For example, if the input array is sorted and if last or first element is chosen as a pivot, then the worst occurs.

65. What actions are performed when a function is called?

When a function is called

i) arguments are passed

ii) local variables are allocated and initialized

ii) transferring control to the function

66. What actions are performed when a function returns?

When function returns;

- i) Return address is retrieved
- ii) Function's data area is freed
- iii) Branch is taken to the return address

67. Is it necessary to sort a file before searching a particular item?

If less work is involved in searching a element than to sort and then extract, then we don't go for sort

If frequent use of the file is required for the purpose of retrieving specific element, it is more efficient to sort the file.

Thus it depends on situation.

68. What are the issues that hamper the efficiency in sorting a file?

The issues are

- i) Length of time required by the programmer in coding a particular sorting program
- ii) Amount of machine time necessary for running the particular program
- iii) the amount of space necessary for the particular program

69. Parenthesis is never required in Postfix or Prefix expressions, why?

70. Parenthesis is not required because the order of the operators in the postfix /prefix expressions determines the actual order of operations in evaluating the expression

71. Convert the expression $((A + B) * C - (D - E) ^ (F + G))$ to equivalent Prefix and Postfix notations.

Prefix Notation:

$^ - * + ABC - DE + FG$

Postfix Notation:

$AB + C * DE - - FG + ^$

72. There are 8, 15, 13, and 14 nodes in 4 different trees. Which one of them can form a full binary tree?

The answer is the tree with 15 nodes. In general, there are $2^n - 1$ nodes in a full binary tree.

By the method of elimination:

Full binary trees contain odd number of nodes, so there cannot be full binary trees with 8 or 14 nodes.

Moreover, with 13 nodes you can form a complete binary tree but not a full binary tree. Thus, the correct answer is 15.

73. List out the advantages of using a linked list

It is not necessary to specify the number of elements in a linked list during its declaration

Linked list can grow and shrink in size depending upon the insertion and deletion that occurs in the list

Insertions and deletions at any place in a list can be handled easily and efficiently

A linked list does not waste any memory space

74. State the advantages of using postfix notations

- Need not worry about the rules of precedence
- Need not worry about the rules for right to left associativity
- Need not need parenthesis to override the above rules

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

- It is not necessary to specify the number of elements to be stored in a stack during its declaration, since memory is allocated dynamically at run time when an element is added to the stack
- Insertions and deletions can be handled easily and efficiently
- Linked list representation of stacks can grow and shrink in size without wasting memory space, depending upon the insertion and deletion that occurs in the list
- Multiple stacks can be represented efficiently using a chain for each stack

76. Define a priority queue.

Priority queue is a collection of elements, each containing a key referred as the priority for that element. Elements can be inserted in any order (i.e., of alternating priority), but are arranged in order of their priority value in the queue. The elements are deleted from the queue in the order of their priority (i.e., the elements with the highest priority is deleted first). The elements with the same priority are given equal importance and processed accordingly.

77. Difference between Abstract Data Type, Data Type and Data Structure

- An Abstract data type is the specification of the data type which specifies the logical and mathematical model of the data type.
- A data type is the implementation of an abstract data type.
- Data structure refers to the collection of computer variables that are connected in some specific manner.i.e) Data type has its root in the abstract data type and a data structure comprises a setoff computer variables of same or different data types.

78. State the difference between primitive and non-primitive data types

Primitive data types are the fundamental data types. Eg) int, float, double, char Non-primitive data types are user defined data types. E.g.) Structure, Union and enumerated data types.

79. Define a full binary tree.

A full binary tree is a tree in which all the leaves are on the same level and every non-leaf node has exactly two children.

80. Define a complete binary tree.

A complete binary tree is a tree in which every non-leaf node has exactly two children not necessarily to be on the same level.

81. What is an expression tree?

An expression tree is a tree which is built from infix or prefix or post fix expression. Generally, in such a tree, the leaves are operands and other nodes are operators.

82. What is the idea behind splaying?

Splaying reduces the total accessing time if the most frequently accessed node is moved towards the root. It does not require maintaining any information regarding the height or balance factor and hence saves space and simplifies the code to some extent.

83. Define Hashing.

Hashing is the transformation of string of characters into a usually shorter fixed length value or key that represents the original string. Hashing is used to index and retrieve items in a database because it is faster to find the item using the short hashed key than to find it using the original value.

84. What do you mean by hash table?

The hash table data structure is merely an array of some fixed size, containing the keys. A key is a string with an associated value. Each key is mapped into some number in the range 0 to tablesize-1 and placed in the appropriate cell.

85. What do you mean by hash function?

A hash function is a key to address transformation which acts upon a given key to compute the relative position of the key in an array. The choice of hash function should be simple and it must distribute the data evenly. A simple hash function is
 $\text{Hash key} = \text{key} \bmod \text{table size}.$

86. What do you mean by collision in hashing?

When an element is inserted, it hashes to the same value as an already inserted element, and then it produces collision.

87. What do you mean by Probing?

Probing is the process of getting next available hash table array cell.

88. What do you mean by secondary clustering?

Although quadratic probing eliminates primary clustering, elements that hash to the same position will probe the same alternative cells. This is known as secondary clustering.

89. What do you mean by rehashing?

If the table gets too full, the running time for the operations will start taking too long and inserts might fail for open addressing with quadratic resolution. A solution to this is to build another table that is about twice as big with the associated new hash function and scan down the entire original hash table, computing the new hash value for each element and inserting it in the new table. This entire operation is called rehashing.

90. Define Graph.

A graph G consists of a nonempty set V which is a set of nodes of the graph, a set E which is the set of edges of the graph, and a mapping from the set for edge E to a set of pairs of elements of V . It can also be represented as $G = (V, E)$

91. What is a simple graph?

A simple graph is a graph, which has not more than one edge between a pair of nodes than such a graph is called a simple graph.

92. What is meant by strongly connected in a graph?

An undirected graph is connected, if there is a path from every vertex to every other vertex. A directed graph with this property is called strongly connected.

93. What is an undirected acyclic graph?

When every edge in an acyclic graph is undirected, it is called an undirected acyclic graph. It is also called as undirected forest.

94. Define bi-connectivity in graph.

A connected graph G is said to be bi-connected, if it remains connected after removal of any one vertex and the edges that are incident upon that vertex. A connected graph is bi-connected, if it has no articulation points.

95. In what areas do data structures applied?

Data structures are important in almost every aspect where data is involved. In general, algorithms that involve efficient data structure is applied in the following areas: numerical analysis, operating system, A.I., compiler design, database management, graphics, and statistical analysis, to name a few.

96. What is the minimum number of nodes that a binary tree can have?

A binary tree can have a minimum of zero nodes, which occurs when the nodes have NULL values. Furthermore, a binary tree can also have 1 or 2 nodes

97. What are dynamic data structures?

Dynamic data structures are structures that expand and contract as a program runs. It provides a flexible means of manipulating data because it can adjust according to the size of the data.

98. What is a bubble sort and how do you perform it?

A bubble sort is one sorting technique that can be applied to data structures such as an array. It works by comparing adjacent elements and exchanges their values if they are out of order. This method lets the smaller values "bubble" to the top of the list, while the larger value sinks to the bottom.

99. Differentiate linear from nonlinear data structure.

Linear data structure is a structure wherein data elements are adjacent to each other. Examples of linear data structure include arrays, linked lists, stacks and queues. On the other hand, non-linear data structure is a structure wherein each data element can connect to more than two adjacent data elements. Examples of nonlinear data structure include trees and graphs.

100. How do signed and unsigned numbers affect memory?

In the case of signed numbers, the first bit is used to indicate whether positive or negative, which leaves you with one bit short. With unsigned numbers, you have all bits available for that number. The effect is best seen in the number range (unsigned 8 bit number has a range 0-255, while 8-bit signed number has a range -128 to +127).