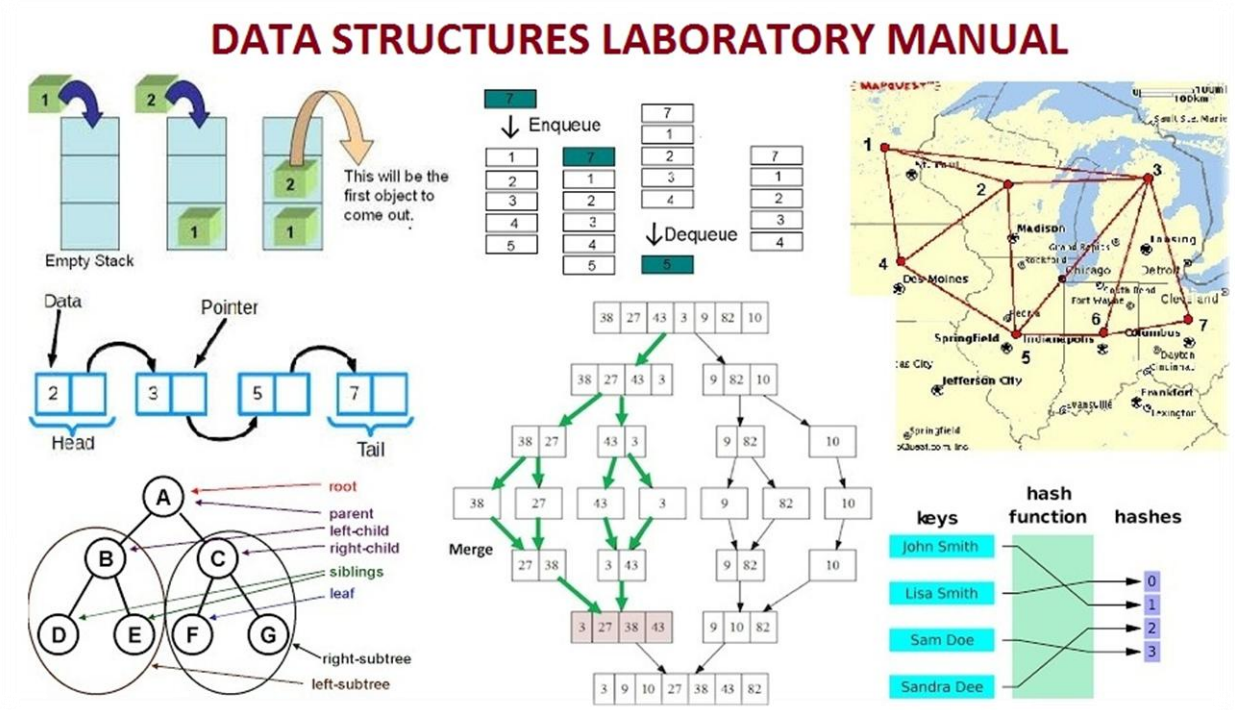
MUFFAKHAM JAH COLLEGE
OF
ENGINEERING AND TECHNOLOGY
(Affiliated Osmania University)
Banjara Hills, Hyderabad, Telangana State

DEPARTMENT
OF
INFORMATION TECHNOLOGY



1. Institution Vision

To be part of universal human quest for development and progress by contributing high calibre, ethical and socially responsible engineers who meet the global challenge of building modern society in harmony with nature.

2. Institution Mission

- To attain excellence in imparting technical education from the undergraduate through doctorate levels by adopting coherent and judiciously coordinated curricular and co-curricular programs
- To foster partnership with industry and government agencies through collaborative research and consultancy
- To nurture and strengthen auxiliary soft skills for overall development and improved employability in a multi-cultural work space
- To develop scientific temper and spirit of enquiry in order to harness the latent innovative talents
- To develop constructive attitude in students towards the task of nation building and empower them to become future leaders
- To nourish the entrepreneurial instincts of the students and hone their business acumen.
- To involve the students and the faculty in solving local community problems through economical and sustainable solutions.

3. Department vision

Fostering a bright technological future by enabling the students to function as leaders in software industry and serve as means of transformation to empower society through ITeS.

4. Department Mission

To create an ambience of academic excellence through state of art infrastructure and learner-centric pedagogy leading to employability in multi-disciplinary fields.

5. Program Educational Objectives

1. The Program Educational Objectives of Information Technology Program are as follows:
2. Graduates will demonstrate technical competence and leadership in their chosen fields of employment by identifying, formulating, analyzing and creating efficient IT solutions.
3. Graduates will communicate effectively as individuals or team members and be successful in varied working environment.
4. Graduates will demonstrate lifelong learning through continuing education and professional development.
5. Graduates will be successful in providing viable and sustainable solutions within societal, professional, environmental and ethical context.

6. Program Outcomes

PO1: Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

PO2: Problem analysis: Identify, formulate, research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences

PO3: Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

PO4: Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

PO5: Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

PO6: The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

PO7: Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

PO8: Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO9: Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

PO10: Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

PO11: Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO 12: Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

7. Program Specific Outcomes

PSO1: Work as Software Engineers for providing solutions to real world problems using Structured, Object Oriented Programming languages and open source software.

PSO2: Function as Systems Engineer, Software Analyst and Tester for IT and ITeS.

8. Introduction to Data Structures

A **Data Structure** is a particular way of storing and organizing data in a computer so that it can be stored, retrieved, or updated efficiently.

Data structures are generally based on the ability of a computer to fetch and store data at any place in its memory, specified by an address — a bit string that can be itself stored in memory and manipulated by the program.

Relationship between Data Structures and Algorithms*:* The term data structure is used to describe the way data is stored, and the term algorithm is used to describe the way data is processed. Data structures and algorithms are interrelated. Choosing a data structure affects the kind of algorithm you might use, and choosing an algorithm affects the data structures we use.

Data structure is a representation of logical relationship existing between individual elements of data. In other words, a data structure defines a way of organizing all data items that considers not only the elements stored but also their relationship to each other. The term data structure is used to describe the way data is stored.

A data structure is said to be ***linear*** if its elements form a sequence or a linear list. The linear data structures like an array, stacks, queues and linked lists organize data in linear order. A data structure is said to be ***non linear*** if its elements form a hierarchical classification where, data items appear at various levels.

Trees and Graphs are widely used non-linear data structures. Tree and graph structures represent hierarchical relationship between individual data elements. Graphs are nothing but trees with certain restrictions removed.

Data structures are divided into two types:

• Primitive data structures.

• Non-primitive data structures.

Primitive types

- Boolean
- Character
- Integer
- Double
- Float

Composite types

- Structures
- Unions
- Tagged union

Abstract data types

- Container
- Deque
- List
- Priority queue
- Queue
- Set
- Stack
- String
- Tree

Linear data structures

Arrays

- Array
- Dynamic array
- Sparse array
- Matrix
- Sparse matrix

Lists

- Linked list
- Doubly linked list
- Circularly Linked List
- Circular Doubly Linked List
- Skip list

Non Linear Data Structures

Binary trees

- Binary tree
- Binary search tree
- Self-balancing binary search tree
- Randomized binary search tree
- Weight-balanced tree
- Threaded binary tree
- AVL tree
- Red-black tree
- Splay tree

B-trees

- B-tree
- B+ tree
- 2-3 tree
- 2-3-4 tree

Graphs

- Undirected Graphs
- Directed Graphs
- Weighted Graphs
- Connected Graphs
- Multigraphs
- Special Graphs, etc

The collections of data you work with in a program have some kind of structure or organization. No matter how complex your data structures are they can be broken down into two fundamental types: Contiguous or Non-Contiguous

In contiguous structures, terms of data are kept together in memory (either RAM or in a file). An array is an example of a contiguous structure. Since each element in the array is located next to one or two other elements. In contrast, items in a non-contiguous structure and scattered in

memory, but we link to each other in some way. A linked list is an example of a non-contiguous data structure. Here, the nodes of the list are linked together using pointers stored in each node. Figure 1, below illustrates the difference between contiguous and non-contiguous structures. Figure 2, and 3 shows the examples of such data structures.

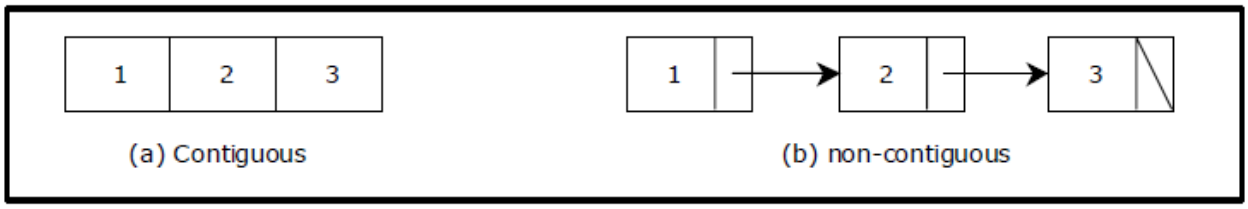


Figure: 1. Contiguous and Non-Contiguous data structures

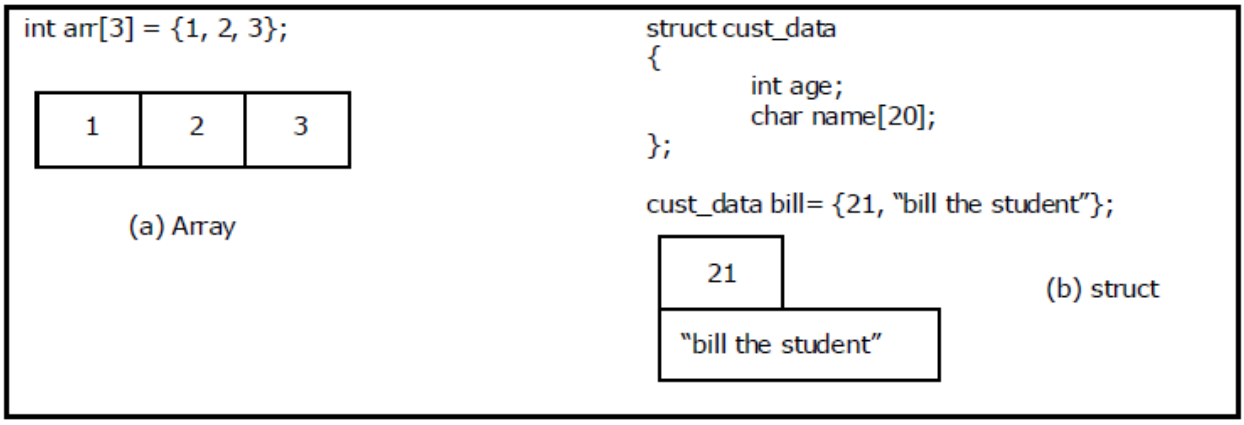


Figure 2: Examples of Contiguous data structures

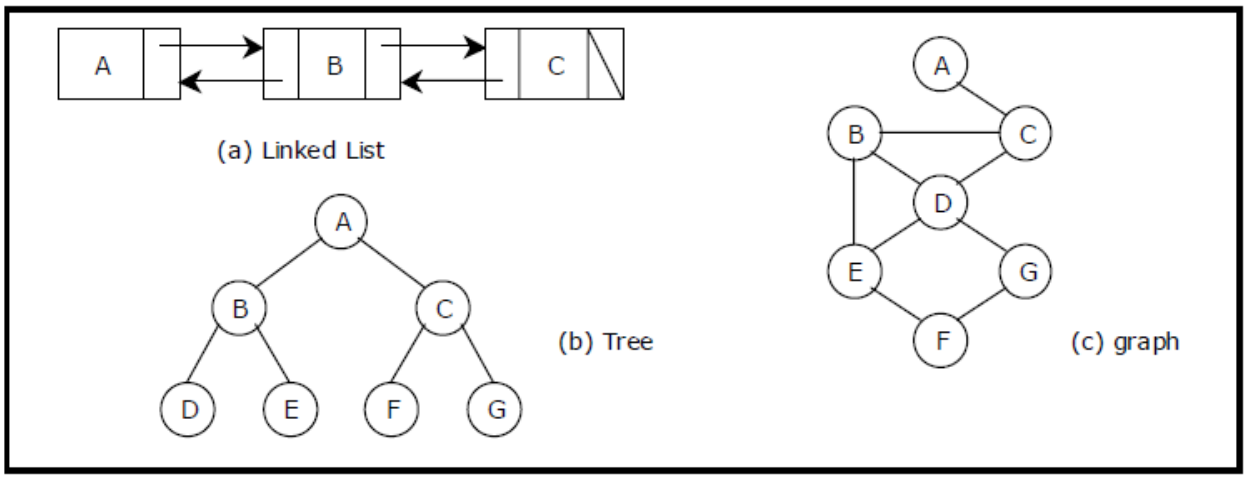


Figure 3: Examples of Non-Contiguous data structures

Linear Data Structures: The linear data structures as mentioned above are: Arrays, Lists, Stacks, Queues, Dequeues, etc.

Stacks: A stack is a data structure in which elements are added to or deleted from a single end called as Top of the stack. The elements last inserted is the first to be removed, therefore stack is said to follow the Last In First Out principle, LIFO. In this context, the insert operation is more commonly known as Push and deletes operation as Pop. Other operations on a stack are: to find size of stack, return top most element, search for an element in the stack etc.

Stacks are used in: Function calls, Recursion, Evaluation of expressions by compilers, undo mechanism in an editor, etc

Figure 4 illustrates the operations carried out on a stack.

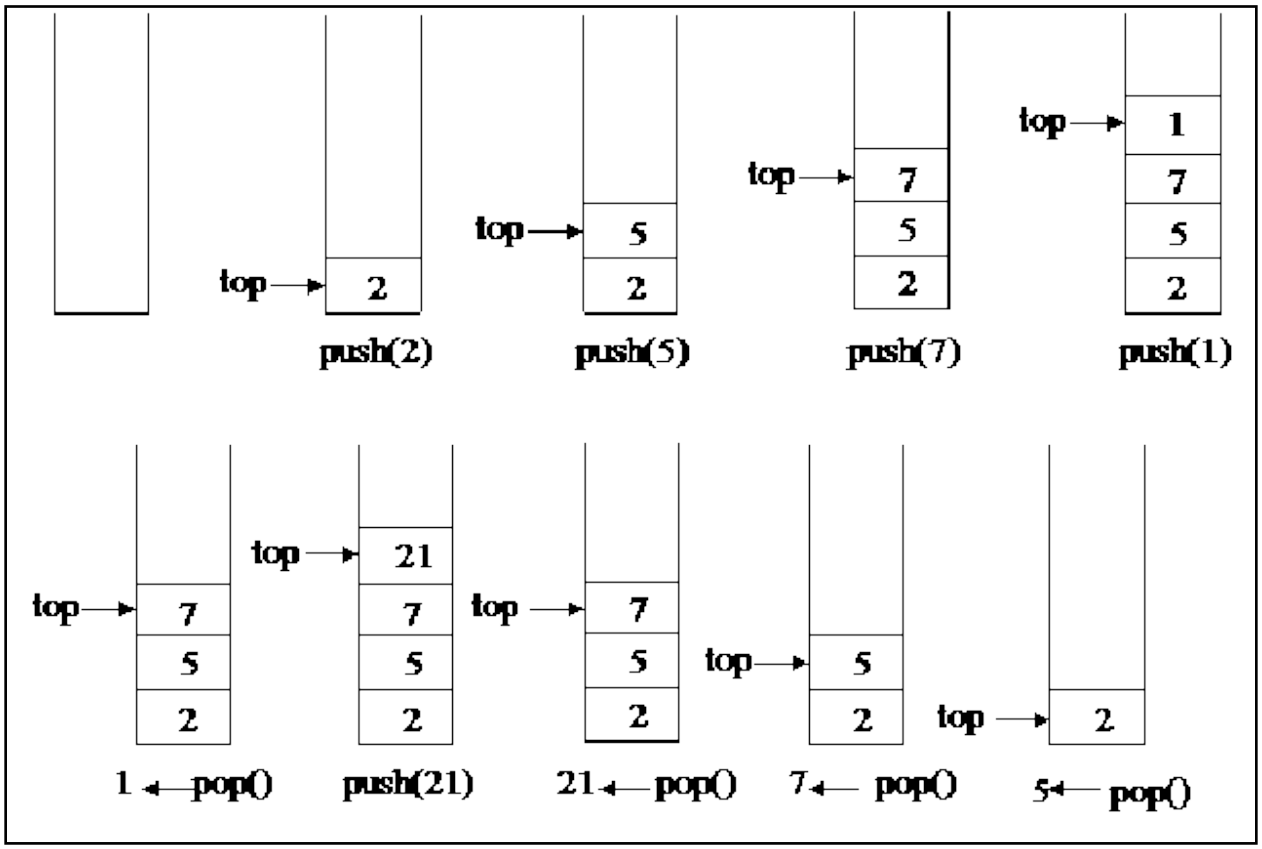


Figure 4. Operations on a stack

The stack data structures can be represented using Arrays or Linked Lists. When represented using an array, the initial value of Top, the index for top most element, when stack is empty, is -1, a nonexistent index.

The ADT for an array based stack is given below:

ADT Stack

{

Private:

An array;

Top Index, capacity;

Public:

Stack(capacity);

Int Push(item);

Void Pop(&item);

}

When elements are pushed into stack, the Top gradually increases towards right. Each pop operation causes the Top index to decrease by one. Figure 5 illustrates the array representation.

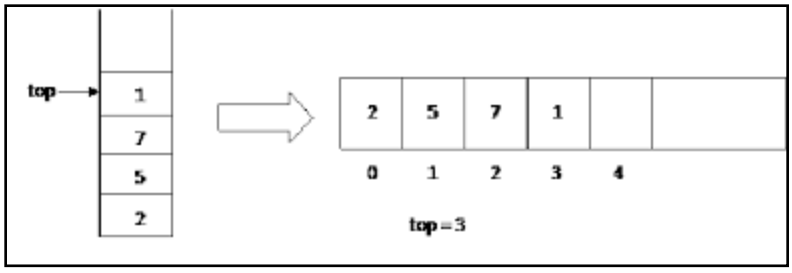


Figure 5: Array Representation of Stack

Stacks can also be represented using linked representation, wherein the elements of the stack are not stored contiguously using an array, rather the elements are stored using a node structure that can be stored in the memory non-contiguously. Each node contains information about the data element and the address of where the next element is stored in the memory.

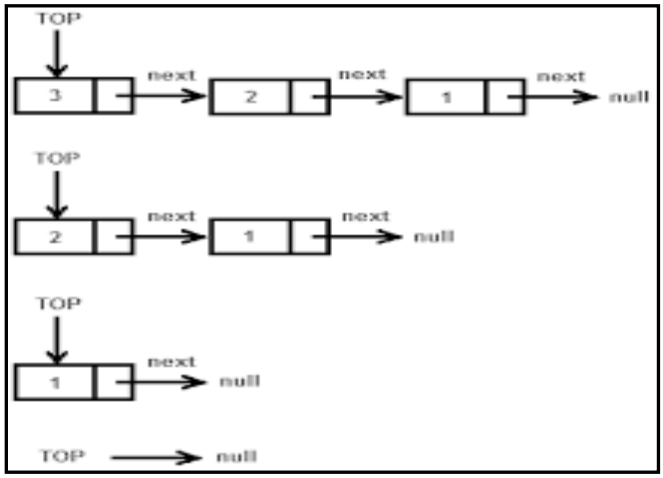


Figure 6: Linked Stack

The ADT for a linked stack is:

```
ADT Node
{
        Private:
                T data; //template type data
                Node<T> *link;
};
ADT LinkedStack
{
        Private:
                Node<T> *Top;
                Int size;
        Public:
                Push(T item); Pop(); bool isEmpty();
}
```

QUEUES: A queue is a linear data structure in which elements can be inserted and deleted from two different ends. The end at which elements are inserted into queue is referred to as Rear and the end from which elements are deleted is known as Front. The element first inserted will be the first to delete, therefore queue is said to follow, First In First Out, FIFO principle. Queues

are used: in processor scheduling, request processing systems, etc. A queue can be implemented using arrays or linked representation.

```
ADT Queue
{
        Private:
                T queue[];
                Int front, rear;
                Int size;
                Int capacity;
        Public:
                Void Push( T item);
                Void Pop(T &item);
                Bool IsEmpty();
                T getFront();
                T GetRear();
};
```

A queue can be implemented using arrays with different constraints on the front and rear pointers.

Front fixed and Rear Variable: in this representation, rear pointer is variable, whereas front pointer is fixed, which means that front always point to the array index 0. Hence each deletion on a queue causes the elements from index 1 to rear to move one position down, backward movement of data.

Front and Rear Variable: in this representation, front is made variable, and every deletion on a queue causes the front pointer to migrate or advance to next location. At one particular point, the front may be greater than 0, rear is at capacity, does it mean that queue is full? Whenever front is non zero and rear reaches capacity, the queue is not full but it can accommodate elements at starting indices of array from 0 to front-1. But to use these locations, we need to either shift elements from front to rear to positions starting from 0, which results in increased time complexity of delete operation. To overcome this wastage of space and increase in time complexity, we imagine the queue to be circular, such that two ends of the queue meet each other.

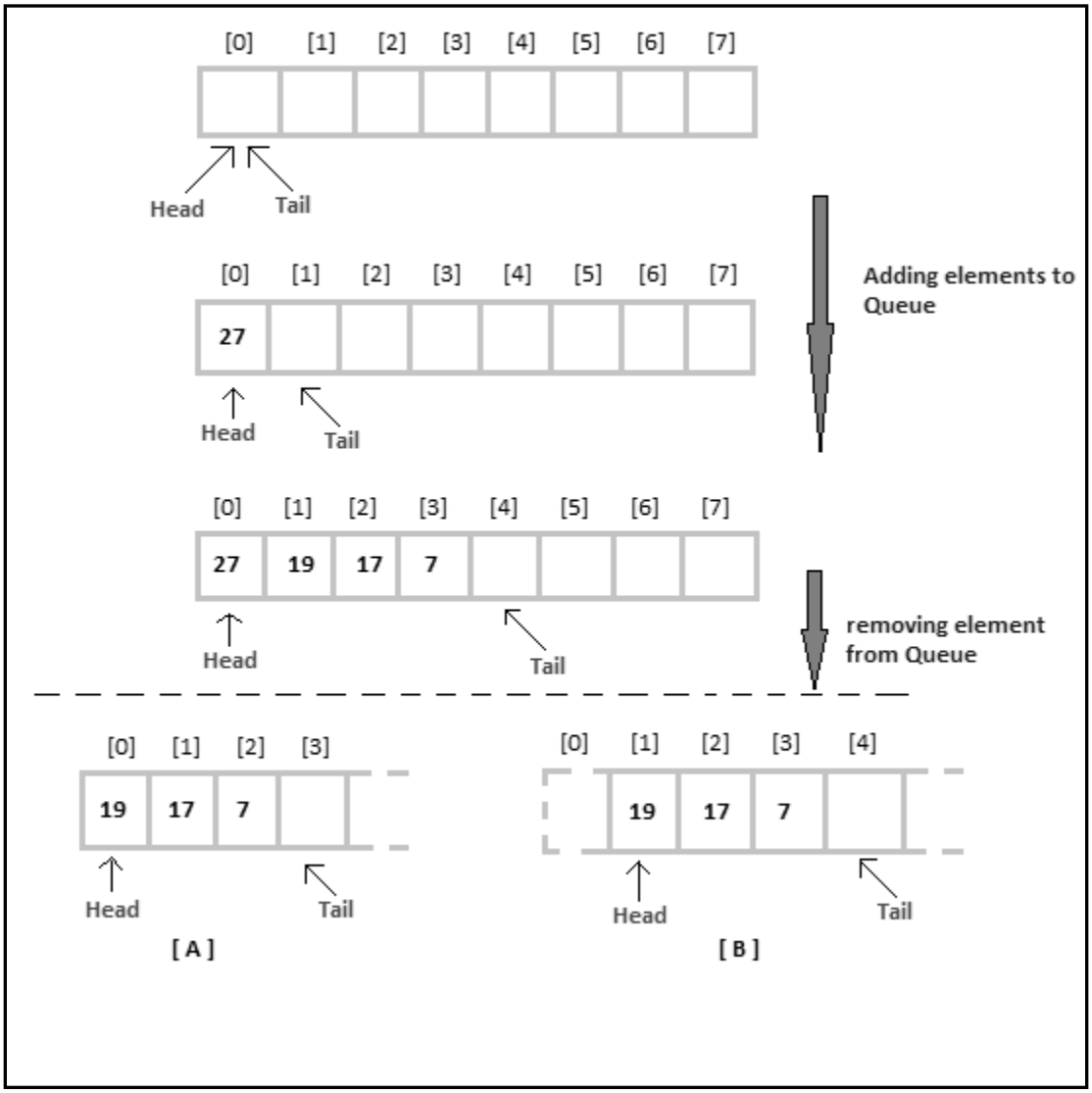


Figure 7: Insert and delete operations on a queue when Front (Head) is variable.

The circular representation of queue requires that the front pointer points one position counter clockwise from the first element inserted in the queue, therefore, if a queue is store in an array of size n, then only n-1 elements can be stored in such a queue. The front is made to point so because it will be difficult to distinguish between queue empty and queue full conditions. Below figure depicts a circular queue.

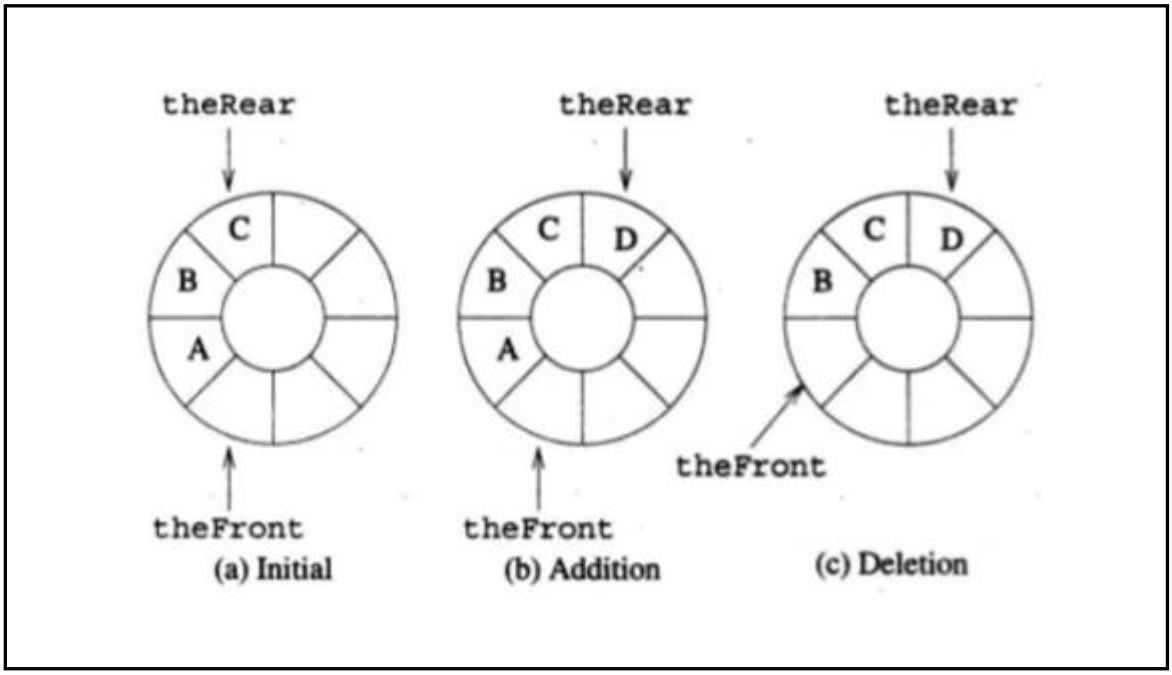


Figure 8: Circular Queue.

Linked Queues: Linked representation of queue is depicted in the below diagram. The node Front points to first element inserted in the queue and node Rear Points to last element inserted in the queue. To insert an element into a linked queue, simply create a new node and let Rear pointer update its link field to address of newly inserted node, and let the newly inserted node be renamed as Rear. To delete an element from a Linked Queue, the Front pointer is advanced to point to the next element.

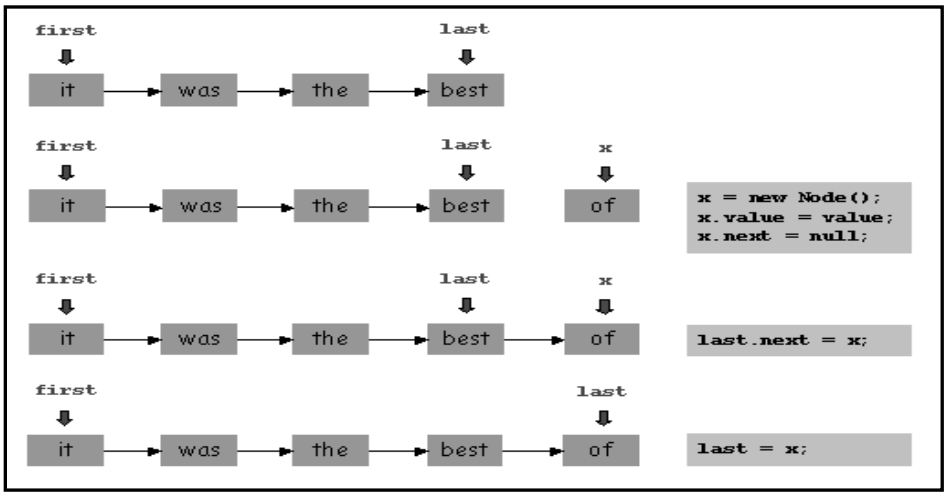


Figure 9: Linked representation of a Queue.

Linked Lists:

A linked list is a collection of nodes which contain data elements as well as the information about where the next element in the list is stored. Each element is stored using a node format, which contains two fields: data and link. Data field contains the information and Link filed contains a pointer to the address of next element in the list.

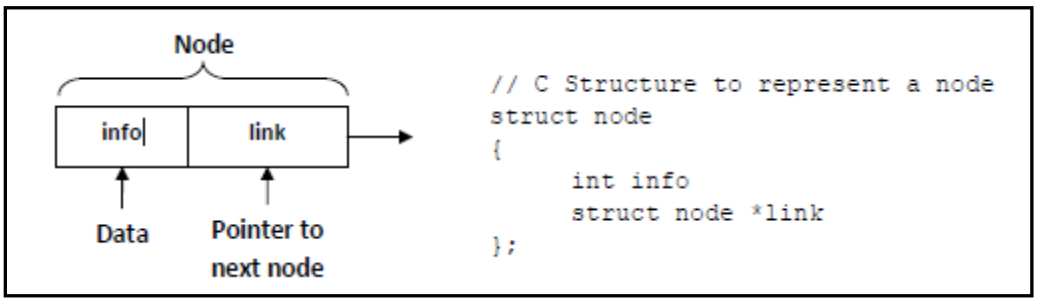


Figure: Linked List Node Structure

Operations on Linked Lists:

Insert: Insert at first position, insert at last position, insert into ordered list

Delete: Delete an element from first, last or any intermediate position

Traverse List: print the list

Copy the linked List, Reverse the linked list, search for an element in the list, etc

Types of Linked Lists:

Singly Linked List:

- It is a basic type of linked list.
- Each node contains data and pointer to next node and last node's pointer is NULL.
- Limitation of SLL is that we can traverse the list in only one direction, forward direction.

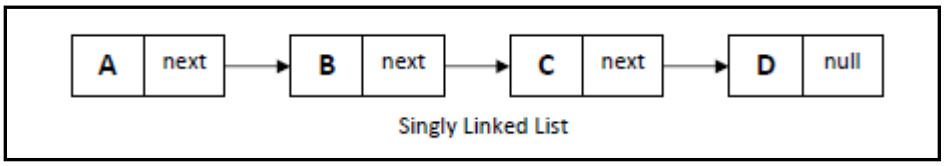


Figure : Single Linked List

Circular Linked List:

- CLL is a SLL where last node points to first node in the list
- It does not contain null pointers like SLL
- We can traverse the list in only one direction

- Its advantage is that when we want to go from last node to first node, it directly points to first node

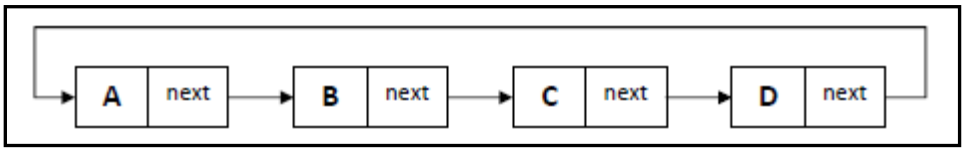
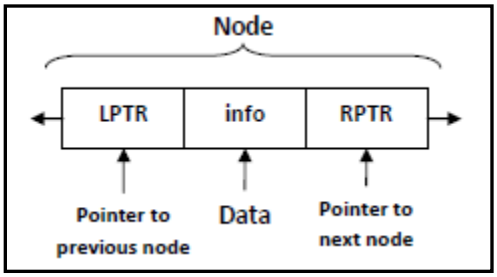


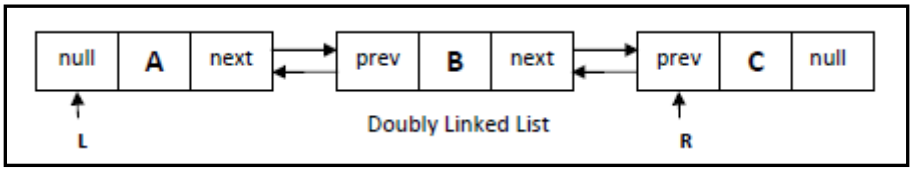
Figure: CLL

Doubly Linked List:

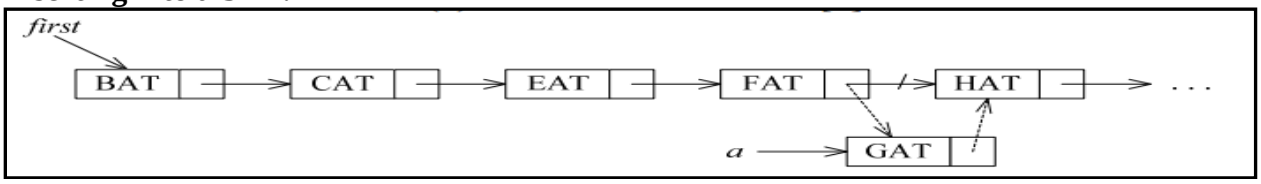
- Each node of doubly linked list contains data and two pointer fields, pointer to previous and next node.
-



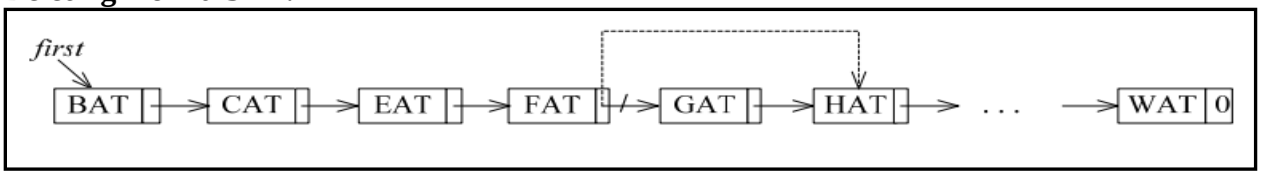
- Advantage of DLL is that we can traverse the list any direction, forward or reverse.
- Other advantages of DLL are that we can delete a node with little trouble, since we have pointers to the previous and next nodes. A node on a SLL cannot be removed unless we have pointer to its predecessor.



Inserting into a SLL:



Deleting from a SLL:



Advantages of an array

1. We can access any element of an array directly means random access is easy
2. It can be used to create other useful data structures (queues, stacks)
3. It is light on memory usage compared to other structures

Disadvantages of an array

1. Its size is fixed
2. It cannot be dynamically resized in most languages
3. It is hard to add/remove elements
4. Size of all elements must be same.
5. Rigid structure (Rigid = Inflexible or not changeable)

Advantages of Linked List

1. Dynamic size
2. It is easy to add/remove/change elements
3. Elements of linked list are flexible, it can be primary data type or user defined data types

Disadvantages of Linked List

1. 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.
2. It cannot be easily sorted
3. We must traverse 1/2 the list on average to access any element
4. More complex to create than an array
5. Extra memory space for a pointer is required with each element of the list

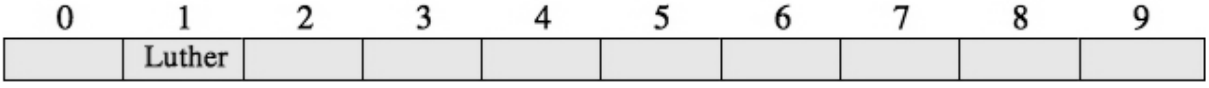
Hashing: Hashing is an important concept in Computer Science. A *Hash Table* is a data structure that allows you to store and retrieve data very quickly. There are three components that are involved with performing storage and retrieval with Hash Tables:

- **A hash table**. This is a fixed size table that stores data of a given type.
- **A hash function**: This is a function that converts a piece of data into an integer. Sometimes we call this integer a **hash value**. The integer should be at least as big as the hash table. When we store a value in a hash table, we compute its hash value with the hash function, take that value modulo the hash table size, and that's where we store/retrieve the data.
- **A collision resolution strategy**: There are times when two pieces of data have hash values that, when taken modulo the hash table size, yield the same value. That is called a *collision*. You need to handle collisions. We will detail four collision resolution strategies: Separate chaining, linear probing, quadratic probing, and double hashing.

An example helps to illustrate the basic concept. Let's suppose that our hash table is of size 10, and that we are hashing strings. We'll talk about hash functions later, but let's suppose that we have four strings that we want to store in our hash table: "Luther," "Rosalita", "Binky" and "Dontonio." Let's also suppose that we have a hash function that converts these strings into the following values:

- "Luther" has a hash value of 3249384281.
- "Rosalita" has a hash value of 2627953124.
- "Binky" has a hash value of 216319842.
- "Dontonio" has a hash value of 2797174031.

So, we start with an empty hash table, and let's store "Luther". "Luther"'s hash value modulo 10 is 1, so "Luther" goes into index 1 of the hash table:



Similarly, "Rosalita" goes into index 4, and "Binky" into index 2. If we insert them into the hash table, the table looks as follows:

0	1	2	3	4	5	6	7	8	9
	Luther	Binky		Rosalita					

To find a string in the hash table, we calculate its hash value modulo ten, and we look at that index in the hash table. For example, if we want to see if "Luther" is in the hash table, we look in index 1.

Now, suppose we want to look for "Dontonio" in the hash table. Since its hash value is 2797174031, we look in index 1. It's not there. Now suppose we wanted to insert "Dontonio." Well, that's a problem, because "Luther" is already at index one. That is a collision.

Properties of hash tables and hash functions: First, we define the **load factor** of a hash table to be:

(Number of data elements in the hash table) / (Size of the hash table)

The selection of a hash function is very important. Important properties of Hash Functions are as follows:

- It should be quick to compute, often constant time, or linear in the size of the data that you are hashing.
- The hash values that it computes should be uniformly distributed from zero to one minus the hash table size. That minimizes collisions.
- A hash function is a function.
    - That is, if $k_1 = k_2$, then $h(k_1) = h(k_2)$.
- A good hash function is one that
    - produces unpredictable hash indices (is *random*).
    - produces widely and evenly separated hash indices (is *uniform*).
    - is fast to compute (is *efficient*).

If the keys are integers and the hash table is an array of size 127, then the function Hash (Key) defined by Hash(Key) = key % 127 maps numbers to their modulus in the finite field of size 127.

Collision Resolution: what to do when two keys are hashed to the same location in the hash table. There are different ways of handling situations in which two keys map to the same location (called collision):

- open addressing (also known as closed hashing) - finds an alternative location for the (key, value) pair, if the first location is occupied. (Linear Probing, Quadratic Probing)
- Closed addressing (also known as open hashing) - allows multiple (key, value) pairs to be stored in a single array location. (Separate Chaining)

Trees: A tree is a structure in which each node can have multiple successors (unlike the linear structures that we have been studying so far, in which each node always had at most one successor). The first node in a tree s called a root, it is often called the top level node (YES, in computer science root of a tree is at the top of a tree). In a tree, there is always a unique path from the root of a tree to every other node in a tree – this has an important consequence: there are no cycles in a tree (think of a cycle as a closed path that allows us to go in a cycle infinitely many times).

The nodes at the end of each path that leads from root downwards are called leaves. The other way to think about it is that leaves are the nodes that point to null. In a linear structure there was only one such node indicating the end of the list. In trees we have many such nodes.

Given a node in a tree, its successors (nodes connected to it in a level below) are called its children.

Descendants of a node are its children, and the children of its children, and the children of the children of its children.

Given a node in a tree, its predecessor (node that connects to it in a level above - there is only one such node) is called its parent.

Ancestors of a node are its parent, and the parent of the parent, and ... - all the nodes along the path from itself to the root

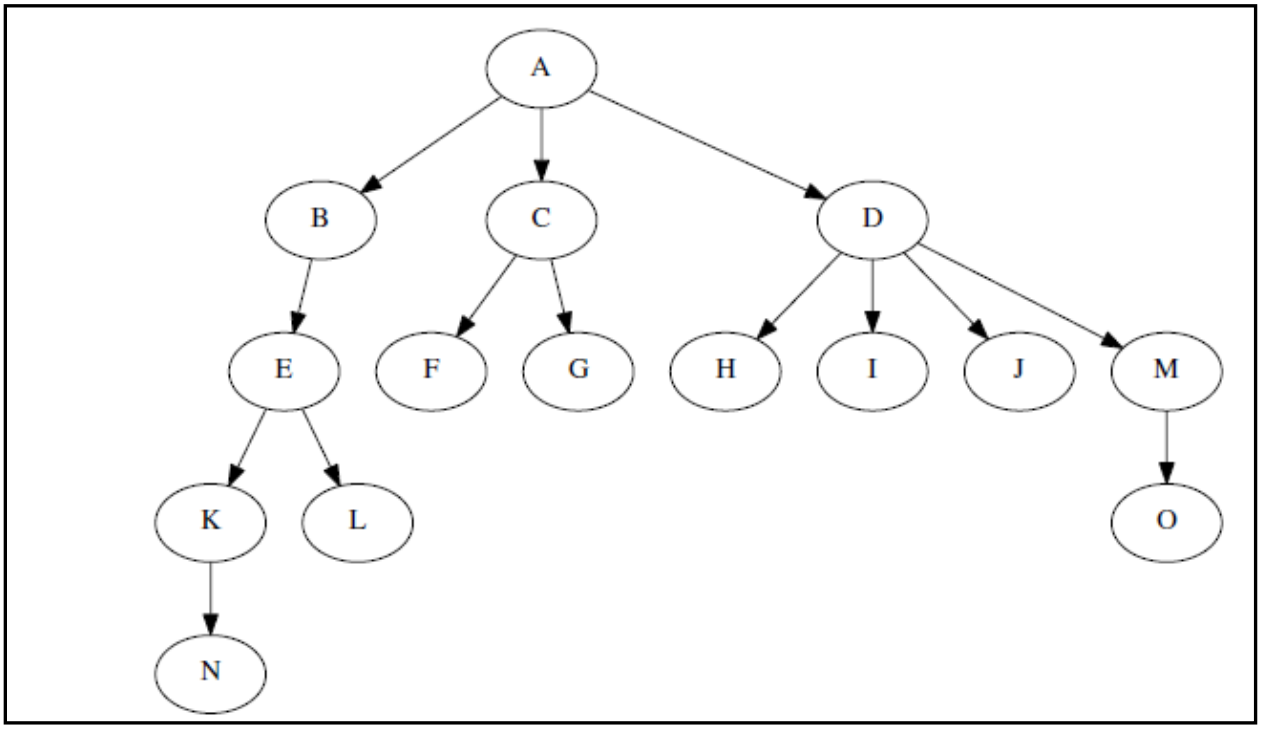


Figure 10: An example of a Tree with 15 nodes

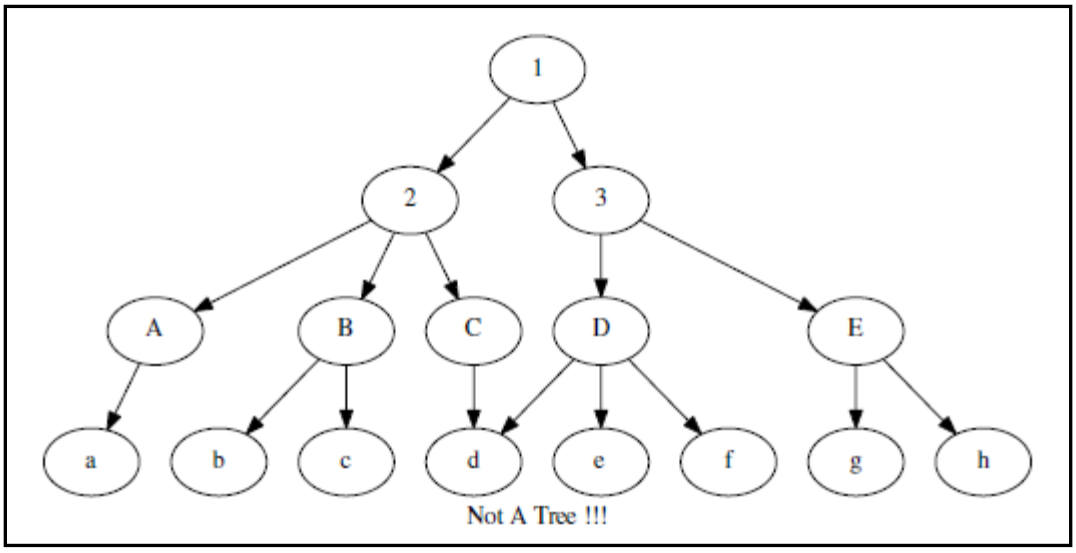


Figure 11: This is not a Tree, as it has multiple paths between a pair of nodes.

Binary Trees: A binary tree is a special kind of tree in which each node can have at most two children: they are distinguished as a left child and a right child. The subtree rooted at the left child of a node is called its left subtree and the subtree rooted at the right child of a node is called its right subtree.

Level of a node refers to the distance of a node from the root. The level of the root is 1. Children of the root are at level 2, "grandchildren" or children of the children of the root are at level 3, etc. The height of a tree is the largest level of any node in the tree. In this case the term depth is used to indicate the largest level.

Maximum number of nodes on a level *i* of a binary tree is 2^{i-1}. Also the maximum number of nodes in a binary tree of depth k is $2^k - 1$, k>0.

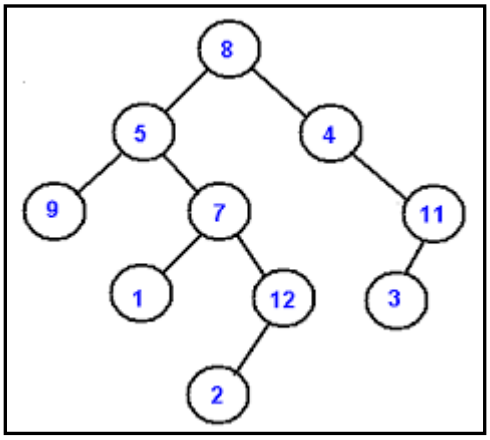


Figure 12: a Binary Tree

Binary Search Tree: A binary search tree is a binary tree which may be empty. If not empty, it satisfies the following properties:

1. Every element has a key and no two elements have the same key .
2. The keys (if any) in the left subtree are smaller than the key in the root
3. The keys (if any) in the right subtree are greater than the key in the root
4. The left and right subtrees are binary search trees.

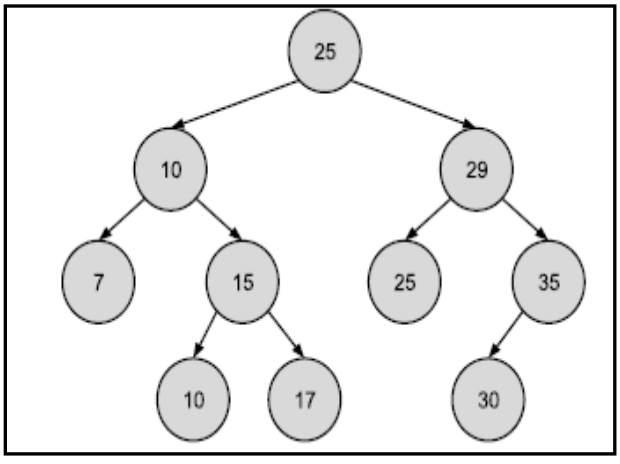


Figure 13: Binary Search Tree on Numbers

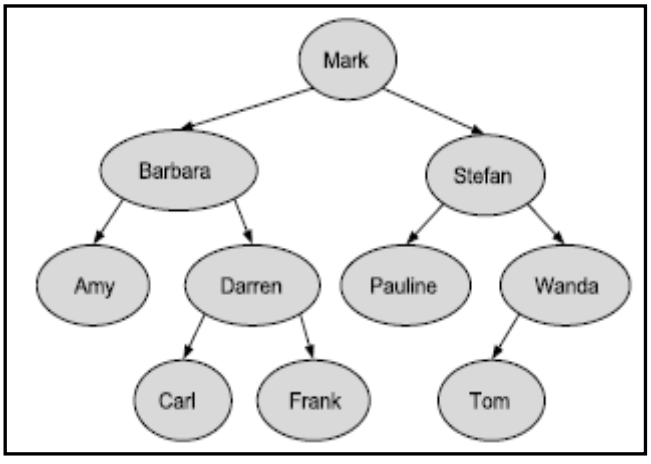


Figure 14: Binary Search Tree on Strings

Searching a Binary Search Tree: Suppose we wish to search for an element with key x. We being at root. If the root is 0, then the search tree contains no elements and the search terminates unsuccessfully. Otherwise, we compare x with key in root. If x equals key in root, then search terminates successfully. If x is less than key in root, then no element in right sub tree can have key value x, and only left subtree is to be searched. If x is larger than key in root, the no element in left subtree can have the key x, and only right subtree is to be searched. The subtrees can be searched recursively.

Insertion into a Binary Search Tree: To insert an element x, we must first verify that its key is different from those of existing elements. To do this, a search is carried out. If search is unsuccessful, then element is inserted at point where the search terminated.

Deleting from a Binary Search Tree: Deletion from a leaf element is achieved by simply removing the leaf node and making its parent's child field to be null. Other cases are deleting a node with one subtree and two subtrees.

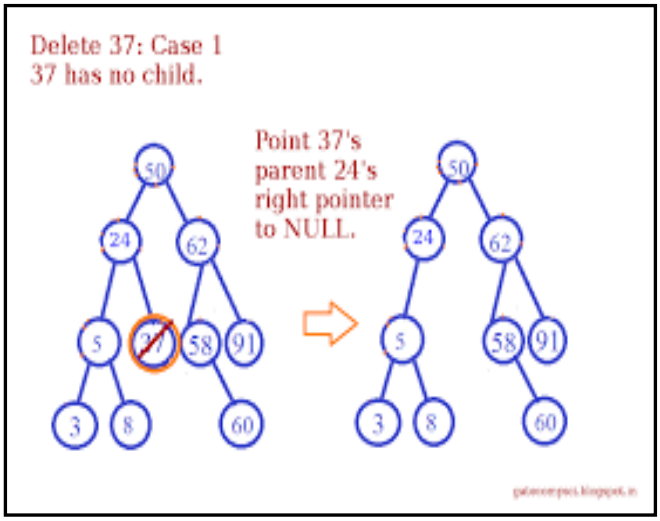


Figure 15: Deleting a leaf node

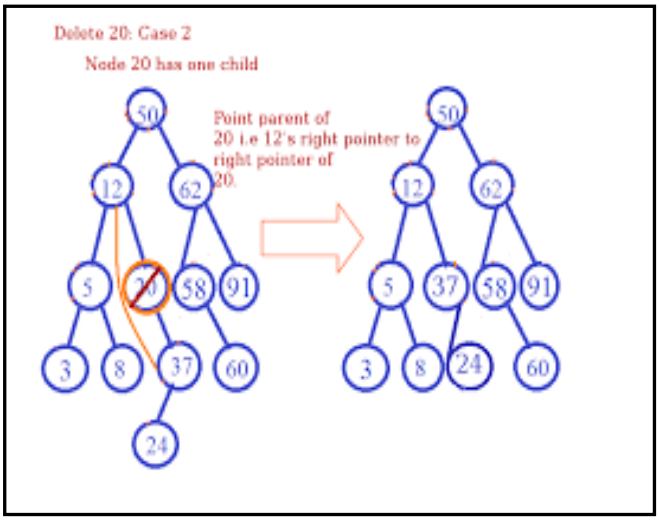


Figure 16: Deleting a non leaf node.

AVL Trees: Consider a Binary Search Tree for months of an year:

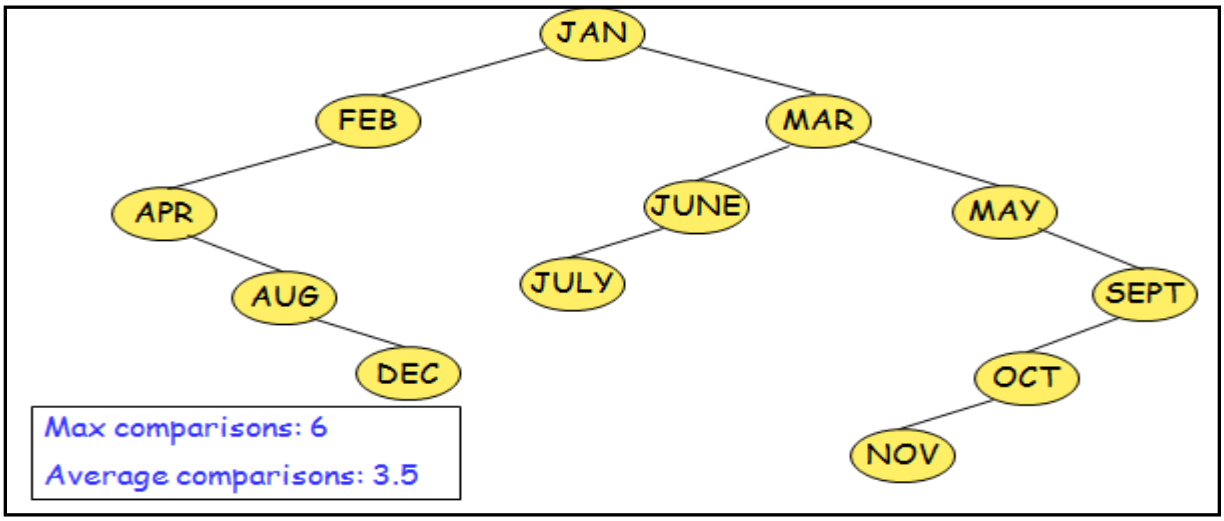


Figure 16: BST when months are inserted in chronological order

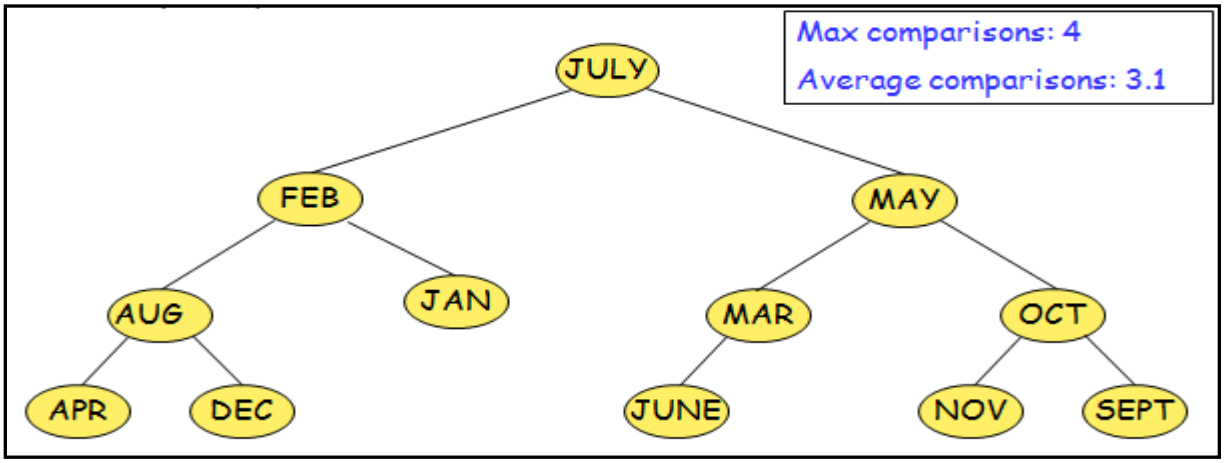


Figure 17: BST when inserted in order: JULY, FEB, MAY, AUG, DEC, MAR, OCT, APR, JAN, JUNE, SEPT, NOV

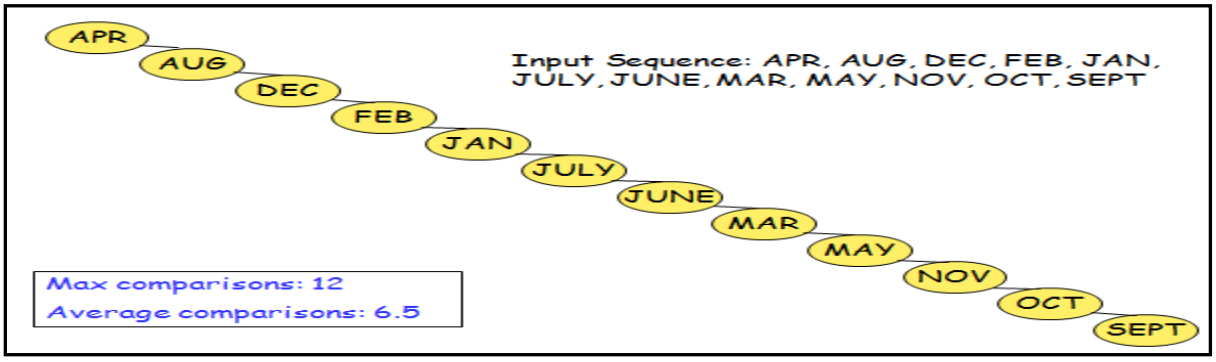


Figure 18: A degenerate BST.

From the above three examples, we know that the average and maximum search time will be minimized if the binary search tree is maintained as a complete binary search tree at all times. However, to achieve this in a dynamic situation, we have to pay a high price to restructure the tree to be a complete binary tree all the time.

In 1962, Adelson-Velskii and Landis introduced a binary tree structure that is balanced with respect to the heights of subtrees. As a result of the balanced nature of this type of tree, dynamic retrievals can be performed in O(log n) time if the tree has n nodes. The resulting tree remains height-balanced. This is called an AVL tree.

AVL Tree: An empty tree is height-balanced. If *T* is a nonempty binary tree with T_L and T_R as its left and right subtrees respectively, then *T* is height-balanced iff

(1) T_L and T_R are height-balanced, and

(2) $|h_L - h_R| \leq 1$ where h_L and h_R are the heights of T_L and T_R, respectively.

(3) The Balance factor, *BF(T)* , of a node *T* is a binary tree is defined to be $h_L - h_R$, where h_L and h_R, respectively, are the heights of left and right subtrees of *T*. For any node *T* in an AVL tree, *BF(T)* = -1, 0, or 1.

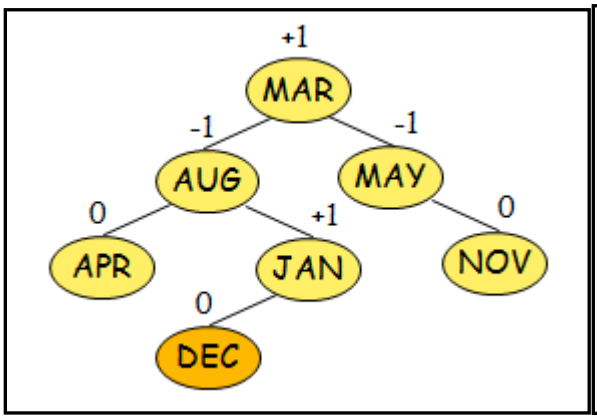


Figure 18: An AVL Tree

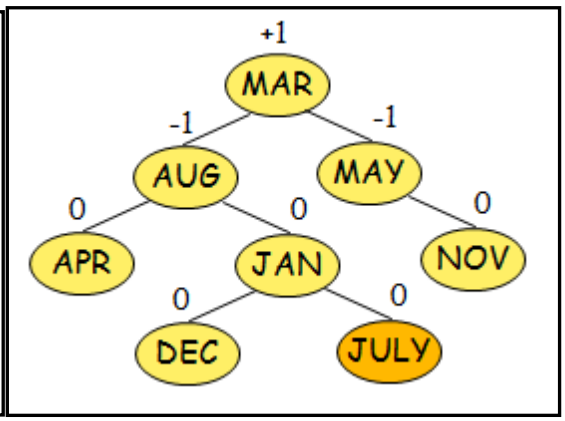


Figure 19: An AVL Tree

Following an insertion, an AVL Tree may no longer be an AVL Tree. To restore the balance factors of the nodes in AVL Tree, any one of the four types of rotations carried out: LL Rotation, RR Rotation, LR Rotation, and RL rotation.

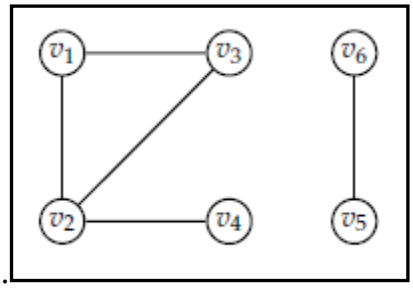
Graphs:

Let V be a *finite* set, and denote by $E(V) = \{\{u, v\} \mid u, v \in V, u \neq v\}$

DEFINITION. A pair $G = (V, E)$ with $E \subseteq E(V)$ is called a **graph** (**on** V). The elements of V are the **vertices** of G, and those of E the **edges** of G. The vertex set of a graph G is denoted by VG and its edge set by EG. Therefore $G = (VG, EG)$.

In literature, graphs are also called *simple graphs*; vertices are called *nodes* or *points*; edges are called *lines* or *links.*

A graph G can be represented as a plane figure by drawing a line (or a curve) between the points u and v (representing vertices) if $e = uv$ is an edge of G. The figure on the right is a geometric representation of the graph G with $VG = \{v1, v2, v3, v4, v5, v6\}$ and $EG = \{v1v2, v1v3, v2v3, v2v4, v5v6\}$.



Graph Representations:

Let $VG = \{v1, \ldots, vn\}$ be ordered. The **adjacency matrix** of G is the $n \times n$-matrix M with entries $Mij = 1$ or $Mij = 0$ according to whether $vivj \in G$ or $vivj \notin G$. For instance, the above graph has an adjacency matrix given below. Notice that the adjacency matrix is always symmetric (with respect to its diagonal consisting of zeros).

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \end{pmatrix}$$

Sorting:

Insertion Sort: Insertion sort iterates, consuming one input element each repetition, and growing a sorted output list. An iteration of insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.

Sorting is typically done in-place, by iterating up the array, growing the sorted list behind it. At each array-position, it checks the value there against the largest value in the sorted list (which happens to be next to it, in the previous array-position checked). If larger, it leaves the element in place and moves to the next. If smaller, it finds the correct position within the sorted list, shifts all the larger values up to make a space, and inserts into that correct position.

The resulting array after *k* iterations has the property where the first *k* + 1 entries are sorted ("+1" because the first entry is skipped). In each of the iterations, the first remaining entry of the input is removed, and inserted into the result at the correct position, thus extending the result.

3 7 4 9 5 2 6 1

3 7 4 9 5 2 6 1

3 **7** 4 9 5 2 6 1

3 **4** 7 9 5 2 6 1

3 4 7 **9** 5 2 6 1

3 4 **5** 7 9 2 6 1

2 3 4 5 7 9 6 1

2 3 4 5 **6** 7 9 1

1 2 3 4 5 6 7 9

Quick Sort: The basic idea of Quick sort is to repeatedly divide the array into smaller pieces (these are called partitions), and to recursively sort those partitions. Quick sort divides the current partition by choosing an element - the pivot - finding which of the other elements are smaller or larger, sorting them into two different sub-partitions (one for the values smaller than the pivot, one for those larger than the pivot).

(66, 77, 11, 88, 22, 33, 44, 55)

R1	R2	R3	R4	R5	R6	R7	R8	Left	Right
[66	77	11	88	22	33	44	55]	1	8
[33	55	11	44	22]	66	[88	77]	1	5
[11	22]	33	[44	55]	66	[88	77]	1	2
[11	22]	33	[44	55]	66	[88	77]	4	5
[11	22]	33	[44	55]	66	[88	77]	7	8
[11	22]	33	[44	55]	66	[77	88]	8	8

Merge Sort: Merge Sort is a sorting algorithm which produces a sorted sequence by sorting its two halves and merging them. Merge Sort algorithm (Like Quick Sort) is based on a divide and conquers strategy. First the sequence to be sorted is decomposed into two halves (Divide). Each half is sorted independently (Conquer). Then the two sorted halves are merged to a sorted sequence (Combine)

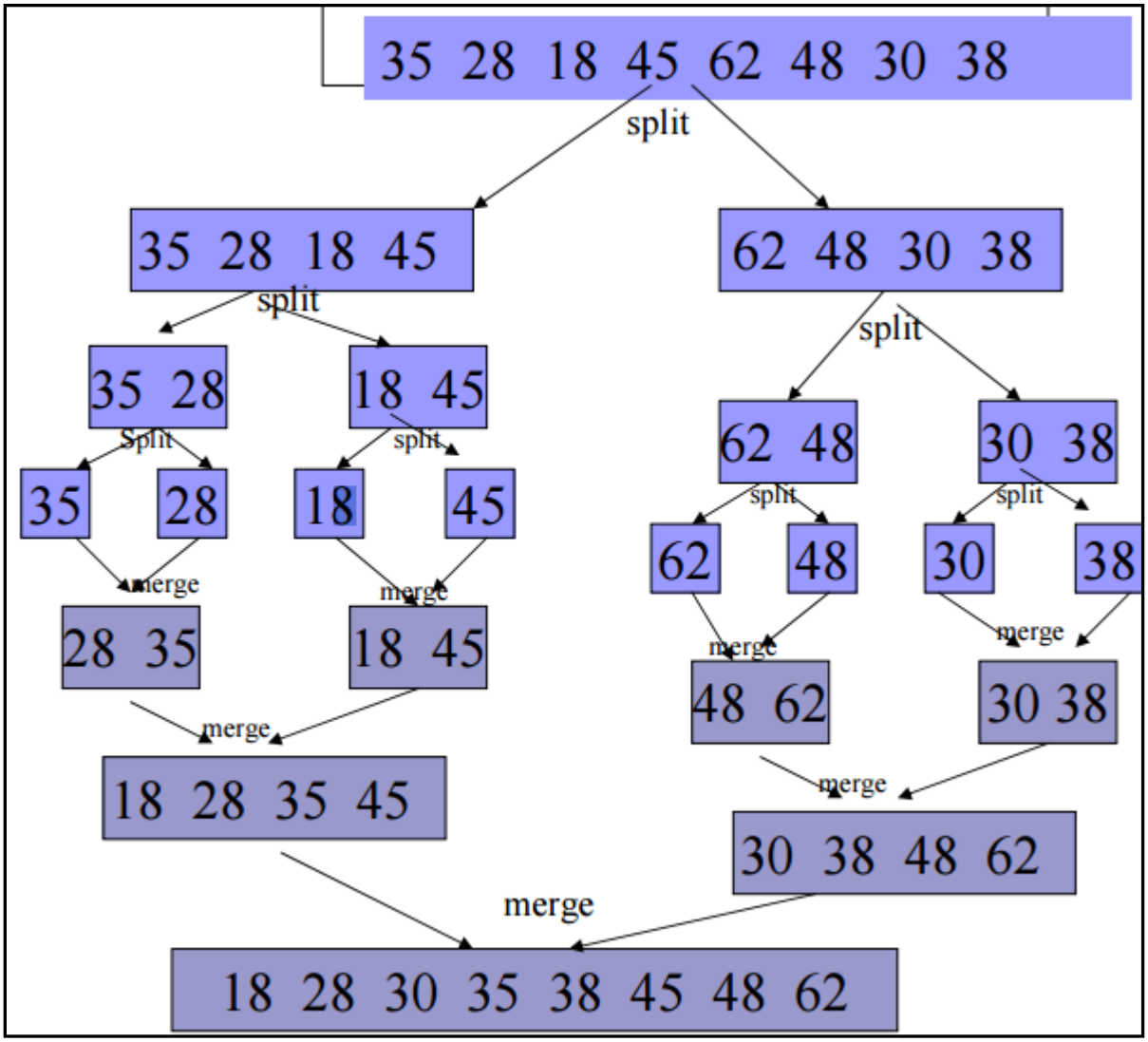


Figure: Working of Merge Sort

9. Program 1: Overview of C++ and Programs to demonstrate C++ classes and Templates

Program Objective: To briefly summarize the concepts of C++ necessary in Data Structures Laboratory such as: Classes, Member Functions, Inheritance, Friend Functions, Templates, Exception handling, Instantiation, etc.

Program Description: In order to give a quick review of aforementioned topics in C++, following programs are demonstrated in the lab.

- Rectangle Class (class Declaration, Member function definition, Invocation)
- Linear Search (template based)
- Binary Search (Template based)
- Max and Min of an Array (Template Based)

Rectangle Class Pseudo Code:

- Declare a class Rect, with data members: len, breadth, area, and perimeter.
- Declare member functions to accept input and compute area and perimeters: Rect(l,b), ComputeArea(), ComputePeri(), Display()
- Write a main function:
  - instantiate the class by passing appropriate arguments to the constructor
  - invoke ComputeArear(), ComputePeri() and Display() functions

Validation: Sample Output: Enter the Length and Breadth of rectangle: 2 4
Area of Rectangle is : 8
Perimeter of Rectangle: 12

Linear Search Pseudo code: Given a list of n elements and a key, objective is to search for the existence of the key in the list, if found return the index of key in list, else return -1 indicating non-existence of key in list.

- Declare a class LinearSearch, with data members: T List[20], T Key, int size;
- Declare member functions: LinearSearch(int n), void Search()

```
Void LinearSearch(int n)
{
    Cout<<"\nEnter "<<n<<" elements in the array\n";
    For(int i=0;i<n;i++)
            Cin>>list[i];
    Cout<<"\nEnter the key to be searched\n";
    Cin>>key;
}
Void Search()
{
    Int flag=0,i;
    For(i=0;i<n;i++;)
    {
            If(list[i]==key)
            {
                    Flag=1; break;
            }
    }
    If(flag==1) cout<<"\nElement found at "<<i<<"\n";
    Else cout<<"\nElement not found\n";
}
```

Validation:

```
Enter 1- for int
2- for float
3- for char
1
Enter no of array elements
5
Enter array elements
8   22      5       13      45
Enter element to be searched
13
Element Found at 4 position
```

Binary Search Pseudo Code: Given an array of elements, and a key, objective is to search the key in array and the list must be a sorted sequence of elements. Binary search searches for a key by dividing the array into two sub arrays at the index (low+high)/2, and compares the key value with element at computed index, if matched, returns, else if the key is smaller than that element, then binary search is carried out in sub array (0:mid-1), or if key is greater then binary search is done on sub array (mid+1:n). This process is repeated recursively until the key is found.

Declare a class with a constructor and the member functions: sort, display, search.

→Accept elements in a dynamically allocated array via constructor.

a=new T[n];

cout<<"\n Enter array elements";

for(int i=0;i<n;i++)

cin>>a[i];

→ Use sort function to sort elements in an array.

→ Use display function to display the sorted array.

→ Use search function to search the desired key element using the following logic:

int l=0,h=size-1,mid;

```
cout<<"Enter element to be searched";
cin>>ele;
while(l<=h)
{
        mid=(l+h)/2;
        for(i=0;i<size;i++)
        {
                if(a[mid]==ele)
                {
                        return 1;
                        break;
                }
                else if(a[mid]>ele)
                h=mid-1;
                else
                l=mid+1;
        }
}
```

Validation: Enter elements and key from the keyboard, and run the program for both successful and unsuccessful search.

Program to find Max and Min of an Array:

Description: Given an array, we have to find out the maximum element and the minimum element of the array.

Implementation logic:

→ Declare a class with a constructor to accept array elements and a member function to find the max and min number and to display them.

→Accept elements in a dynamically allocated array via constructor.

```
a=new T[n];
cout<<"\n Enter array elements";
for(int i=0;i<n;i++)
        cin>>a[i];
```

Member function details:

→ Initially assign high and low to first element in an array.

```
low=a[0];
high=a[0];
```

→ Compare high and low with each array element and after each comparison assign larger value to high and lower value to low.

```
for(int i=0;i<n;i++)
{
        if(high<a[i])
                high=a[i];
        if(low>a[i])
                low=a[i];
}
```

Validation: run the program by entering array elements of integer type and character type.

Sample output:

```
 Enter  1-for int
        2- for float
        3- for char
1
Enter the size of an array
5
Enter array elements
1        45       23       78       5
The max is 78 and the min is 1
```

10. Program 2: Implementation of Array ADT and String ADT

Program Objective: Implementation of One Dimensional Array as an Abstract Data Type

Program Description: The objective is to write a program to perform various operations on an Array of elements, such as, create an array of elements, store elements, retrieve elements using overloading of stream insertion and extraction operators, and to know if array is empty or full, etc.. The ADT for Array1D is as follows:

```
class Array1D
{
        private:
                int capacity;
                int size;
                int *array;
        public:
                Array1D(int Arraycap=10);
                ~Array1D(){ delete [] array;}
                int Getsize();
                bool isEmpty() const;
                bool isFull() const;
                void insert(int pos,int vol);
                void Display(int pos,int &x);
        friend istream& operator>>(istream& is, Array1D &a);
        friend ostream& operator<<(ostream& os, Array1D &a);
};
```

- Implement the constructor to dynamically allocate memory to the array, and also implement the Destructor to release the memory allocated to array.
- GetSize() function returns the number of elements currently stored in array
- isEmpty() checks if the array is empty and returns true if so, else false
- isFull() checks if the array is full before every insert operation on array.
- Insert(pos, val) is a function that will insert an item *val* at the index *pos*.
- Stream insertion and extraction operators are to be overloaded

Validation: Run the program with elements of type integers, characters and float .

Sample Output:

Enter the number of elements to be stored in the array ADT

5

Enter the elements: 10, 4, 5, 2, 54

Array is not Empty.

Retrieving the elements from array: 10, 4, 5, 2, 54

String ADT: Implementation of various operations on an array of characters or String.

Program Description: The objective is to perform various operations on a string of characters: create, display, reverse, find substring, find a given pattern in a given string, count the occurrences of characters of the string, concatenate two strings, find length of string etc.

Pseudo Code:

- Declare a class StringADT, with data members as: *str, size
- Implement the constructor and dynamically allocate memory to str and assign passed string to str
- Compute length of string by using logic: for(int i=0;str[i]!='\0';i++) len++;
- Overload the == operator to find if two passed strings are identical or not (without using strcmp)
    - ➔ Compute length of first string, l1
    - ➔ Compute length of second string, l2
    - ➔ Compare if l1 and l2 are same, if not, strings are not identical
    - ➔ If i1==i2, then compare every character of first string corresponding character of second string, for every match increase a flag value,
    - ➔ At the end compare the flag value with either l1 orl2, if same, strings are identical, else not.
- Implement the Concat function to concatenate two string
    - ➔ Compute length of first string, l1
    - ➔ Compute length of second string, l2
    - ➔ Check if any of the string is empty, if so then no concatenation can happen
    - ➔ Concatenation is done as follows: for(i=l1,j=0;j<l2;j++,i++) s[i]=t.s[j];
- Implement substr(int I, int j) to obtain a substring from index i to index j.

```
void substr(int i, int j)
{
      if(i>Length() || (i+j-1)>Length())
            throw "invalid range specified, retry";
      char str[20];
      int l=i,k;
      for(k=i;k<=(i+j-1);k++)
      {
            str[k]=s[l];
            l++;
      }
      str[k]='\0';
      cout<<"\nThe substring is ...."<<str<<"\n";
}
```

- Implement the Find(pattern) function to know if a given pattern exists in string

```
int Find(String pat)
{
      for(int start=0;start<=Length()-pat.Length();start++)
      {
            int j;
      for(j=0;j<pat.Length() && s[start+j]==pat.s[j];j++)
      {
            if((j+1)==pat.Length()) return start;
      }
      }
      return -1;
}
```

Validation: validate the above program by entering appropriate strings, indices, and patterns.

ENter the string..

narendermodi

The entered String is ...

narendermodi

ENter one more string

modi

The entered String is ...

modi

The size of string 1 is...12

The size of string 2 is...4

in == lengths are as...12 4

The entered strings are not identical

The concatenated string is...narendermodimodi

11. Program 3: Programs for Stack, Queue and Circular Queue.

Program Objective: Write programs to demonstrate the fundamental data structures: Stacks and Queues

Description: The objective is to implement the stack data structure. Operations to be carried out on a stack are push, pop, isEmpty, isFull, TopElement, Display(), etc.

Declare a class with a constructor and the following member functions: push, pop, display, op.

→Dynamically allocate memory to the array via constructor and initialize top to -1 in it.

a=new T[n];

top=-1;

Member function details:

→ **push():** check if the stack is full if so display "stack full" else increment top and push the element in position pointed by the top.

```
if(top==n-1)// Stack full condition
        cout<<"stack is full";
else
{
        ++top;
        a[top]=ele;
}
```

→ **pop():** check if the stack is empty if so display "stack empty. No elements to pop" else pop the element pointed by the top, store it in a variable of same type and decrement top to point the top most element in the stack.

```
T val;
if(top==-1)
        return -1;
else
{
        val=a[top];
        top--;
        return val;
}
```

→ **Display():** Check if the stack is empty if not display the elements in the stack.
→ **op():** Give the following options to the user:

```
cout<<"\n Enter 1-for push\n 2-for pop\n 3-for display\n 4-to exit\n";
cin>>ch;
```

Carry out the operation chosen by the user.

Validation: Execute the program and carry out multiple push, pop and display operations to understand the working of the program

Sample Output:

Enter 1-for int
2- for float
3- for char
3
Enter the no of elements in stack 4
Enter 1- for push
2- for pop
3- for display
4- for exit
1
Enter element to push a
Enter another choice 1
Enter element to push b
Enter another choice 1
Enter element to push c
Enter another choice 2
poped value is c
Enter another choice 3
Stack contents are
b a Enter another choice 4

Queues: program to demonstrate various operations on a queue data structure

Description: The objective is to implement a queue data structure, which supports creation, pushing of elements, pop of elements, return front and rear elements, etc. The queue data structure is implemented using arrays.

→ Declare a class with a constructor and the following member functions: push, pop, display, and op.

→Dynamically allocate memory to the array via constructor and initialize rear=-1 and front=0 in it.

a=new T[n];

Member function details:

→ Push(): check if the Queue is full if so display "Queue full" else increment rear and push the element in position pointed by the rear.

if(r=n-1)/***Queue full condition***/
cout<<"Queue is full";

```
else
{
    ++r;
    a[r]=ele;
}
```

→ Pop(): check If the Queue is empty if so display "Queue empty. No element to pop"

Else pop the element pointed by the front, store it in a variable of same type and If rear and front are pointing to the same position (queue empty) then set them to the positions show below Else just increment front. Do not forget to return the element.

```
if(r==-1)/***Queue Empty condition***/
return -1;
else
{
    T ele=a[f];
    if(f==r)/***Queue Empty***/
    {
            f=0;r=-1;/*** Set f and r***/
    }
    else
            ++f;
    return ele;
}
```

→ Display(): Check If("r==-1") the Queue is empty if not display the elements in it.

→ op(): Give the following options to the user:

```
cout<<"\n Enter 1-for enqueue\n 2-for dequeue\n 3-for display\n 4-to exit\n";
cin>>ch;
```

Carry out the operation chosen by the user.

Sample Output:

```
Enter 1-for int
2-for float
3-for char
3
Enter the size of the array: 4
Enter 1-for enqueue
2-for dequeue
3-for display
4-for exit
1
Enter element j
Enter another choice 1
Enter element a
Enter another choice 2
The front element in the queue is j
```

Enter another choice 3
Contents are
a
Enter another choice 4

Circular Queues:

→ Declare a class with a constructor and the following member functions: Push, Pop, display, op.

→Dynamically allocate memory to the array via constructor and initialize rear and front to -1 in it.

a=new T[n];

Member function details:

→ Push(): check If the Circular queue is full if so display "Queue full" Else assign rear to (rear+1)%n and push the element in position pointed by the rear.

If front is equal to -1 then set it to 0.

```
if((r+1)%n=f)/***Circular queue full condition***/
cout<<"Queue is full";
else
{
    r=(r+1)%n;
    a[r]=ele;
}
if(f==-1)
    f=0;
```

→ Pop(): check If the Queue is empty if so display "Queue empty. No element to pop"

Else pop the element pointed by the front i.e. store it in a variable of same type and If rear and front are pointing to the same position (queue empty) then set them to the positions show below Else just assign (f+1)%n to front . Do not forget to return the element.

```
if(r==-1)/***Circular queue Empty condition***/
return -1;
else
{
    T ele=a[f];
    if(f==r)/***Circular queue Empty***/
            f=r=-1;
    else
            f=(f+1)%n;
    return ele;
```

→ Display(): Check If("r==-1||f==-1") the Circular queue is empty if not check

If front is less than or equal to rear if so display elements present at the indices starting from front to rear.

If front is greater than rear then display elements that are

1. Present at the indices starting from front to n-1(n is size of the array).

2. Present at the indices starting from 0 to rear.

```
int i;
if(r==-1|| f==-1)
cout<<"\nQueue is empty";
else
{
    cout<<"\ncontents are\n";
    if(f<=r)
    {
            for(i=f;i<=r;i++)
                    cout<<a[i]<<endl;
    }
    if(f>r)
    {
            for(i=f;i<=n-1;i++)
                    cout<<a[i]<<endl;
            for(i=0;i<=r;i++)
                    cout<<a[i]<<endl;
    }
}
```

→ op(): Give the following options to the user:

```
cout<<"\n Enter 1-for enqueue\n 2-for dequeue\n 3-for display\n 4-to exit\n";
cin>>ch;
```

Carry out the operation chosen by the user.

12. Program 4: Program to convert an Infix expression into Postfix and Evaluate Postfix Expression

Program Objective: The objective is to write a program to demonstrate the execution of expressions by a compiler using stack data structure.

Program Description: In order to execute the arithmetic, logical, or relational expressions, we have to convert these expressions for the efficient execution into a notation known as Reverse Polish Notation or Postfix notation, wherein the operators come after the operands. Data structure used to carry out this conversion and evaluation is Stack.

Pseudo Code:

→Initialize stack contents to the special symbol #

→ Scan the left most symbol in the given infix expression and denote as the current input symbol

→ While the current symbol is not # DO

→ Remove and output all stack symbols whose precedence values are greater than or equal to precedence of the current input symbol

→ Push current input symbol on to the symbol.

→ Scan the left most symbol in the infix expression and let it be the current input symbol.

→ Remove an output all stack symbol until the top element becomes #.

```
int pre(char x)
{
    if(x=='+'||x=='-')
            return 1;
    else if(x=='*'||x=='/')
            return 2;
    else if(x=='#')
            return 0;
    else
            return 3;
}
int main()
{
    try
    {
            Stack<char>s(30);
            char inf[30],pf[30],x;
            int i=0,j=0,l;
```

```
        cout<<"Enter an infix expression\n";
        cin>>inf;
        l=strlen(inf);
        inf[l]='#';
        inf[l+1]='\0';
        s.push('#');
        while(!s.isEmpty())
        {
                if(pre(inf[i])>pre(s.Top()))
                {
                        s.push(inf[i++]);
                }
                else
                {
                        s.pop(x);
                        pf[j++]=x;
                }
        }
        pf[j-1]='\0';
        cout<<pf;
}
catch(char *e) {        cout<<e<<endl;      } }
```

Program Validation: Execute the program and provide an arithmetic expression, and obtain the postfix notation of that expression. Sample output is as follows:
Enter an infix expression: a+b*c
The postfix notation of expression is : bc*a+
Evaluation of Postfix Expression: The objective is to evaluate a given postfix expression using stack data structure. For example, 2+3*5 is equivalent to expression 3 5 * 2+ in postfix.

To evaluate 3 5 *2+,

- Create a stack of integers, i.e., Stack<int> s(30)
- Enter a postfix expression, i.e. 3 5 * 2 +
- Scan the postfix expression from left to right until \0 is encountered and do the following
    - If an operand is encountered, then push it onto the stack
    - If an operator is encountered, then pop two topmost elements from the stack and carryout the operator on these two operands and push the result onto stack
- Display the result.

Validation: to validate the program, enter a postfix expression from the keyboard, such as 23+ or 23*5+ etc.

13. Program 5: Program to implement a Linear List and Singly Linked List

Program Description: the objective is to write a program to implement the linear list using arrays and a linked list using linked representation. The operations to be carried out on the list are: create, insert (prepend, append, insert after), delete (first position, intermediate, last position), merge two lists, sort, display, reverse, etc.

Linear List using Arrays: A list can be created using a one-dimensional array as follows:

→ Declare a list of template type, and size of list

→ Define a member function IsEmpty() to check for the empty list and a Member function size() to find the no of elements in the list.

→ get(index): This member function returns an element if present at the given index of the list else a false value is returned.

Return list[index];

→ indexof(x): This member function returns the index of the element x and returns a false value if x is not present in the list.

```
For(int i=0;i<size;i++)
{
        If(list[i]==x) then return i+1;
}
Return -1;
```

→ erase(index): This member function removes or deletes the index element.

To remove or delete this element we first need to ascertain that the list contains an element with this index and then delete the element. Then, we have to move the elements from index index-1 to n-1 one position down.

→ insert(index,x): This member function inserts an element x as the index element. To insert new element as index element we first need to move element at position index through n-1 one position up then insert new element in position index and increment n by one.

Sample output:

```
Enter 1-for int
2-for float
3-for char
1
Enter length of array 4
Enter
```

```
1.To insert element
2.To erase element
3.To print listsize
4.To get index of given element
5.To get element's index
6.Exit
7.Display
1
Enter index and element to be inserted in the list 0
12
Enter another choice 1
Enter index and element to be inserted in the list 1
23
Enter another choice 1
Enter index and element to be inserted in the list 2
45
Enter another choice 2
Enter index at which element to be deleted 2
Enter another choice 3
The size of the list is 2
Enter another choice 4
Enter element to find its index 23
The index of the element is 1
Enter another choice 5
Enter index to find the element 0
The element at the given index is 12
Enter another choice 7
12  23
```

Singly Linked List: an SLL is a collection of nodes wherein every element contains the information about address of the next element in the list. Last element contains its link field as NULL. Since there is only one link field, it is known as singly linked list. The structure of the node is as follows:

```
Template<class T>
Class Node
{
        Public:
                T data;
                Node<T> *link;
};
```

- Write a class SLL with following data members: size, Node<T> *first, *last;
- Write the constructor SLL() to initialize: first=last=NULL, size=0
- Implement the destructor to destroy all nodes of a list one after the other
- Implement the insert function with following possibilities:
    - Create a temporary node to be inserted in list: temp;

- Inserting in an empty list: if (size==0) then first=last=temp;
- If list is not empty, insert at first position: temp->link=first; first=temp;
- If list is not empty, insert at last position: last->link=temp; last=temp;
- To insert at any intermediate position:

```
Node<T> *p=first;
for(int i=0;i<pos-1;i++)
      p=p->link;
temp->link=p->link;
p->link=temp;
```

- Increase the list size by one: size++;

- Implement the Delete function with following possibilities:
  - If list is empty, throw an exception
  - Else, if pos=1, then advance the first pointer to second node, delete the first node
  - If pos==size, let a pointer p point to last but one node, make p->link=NULL, delete the last node, and make p as the new last node.
  - If deleting an element from any intermediate position, let p point to a node that precedes the node to be deleted q, let p->link=q->link, delete q
  - Decrease the list size by 1.
- Implement the main function as follows:

```
int main()
{
        try
        {       int x,e;
                Chain<int> a;
                a.Insert(1,100);
                a.Insert(2,200);
                a.Insert(3,300);
                a.Insert(4,400);
                a.Insert(5,500);
                a.Insert(6,600);
                a.Display();
                a.Delete(2,x);
                cout<<"\nThe deleted element is ..."<<x<<"\n";
                cout<<"the list after deletion is\n";
                a.Display();
                cout<<"\nInserting at the Backkk...\n";
                cout<<"\nEnter an element to be inserted at back of list (Append)";
                cin>>e;
                a.InsertBack(e);
                cout<<"\n The list after InsertBack function is called...\n";
                a.Display();
                Node<int> *n=a.GetNode();
                cout<<"This is first node's data"<<n->data;
        }catch(char *e)
```

```
{
        cout<<e<<"\n";
}
return 0;
}
```

Program Validation: Execute the program with appropriate data items to obtain a linked list of elements. Sample output is as follows:

```
"SLL.cpp" 163L, 3133C written
[asrar@it DS]$ g++ SLL.cpp
][asrar@it DS]$ ./a.out

100->200->300->400->500->600->

The deleted element is ...200

the list after deletion is

100->300->400->500->600->

Inserting at the Backkk...

Enter an element to be inserted at back of list (Append)23

 The list after InsertBack function is called...

100->300->400->500->600->23->This is first node's data100[asrar@it DS]$
```

14. Program 6: Programs to implement Stacks and Queues using Linked Representation

Program Objective: The objective is to implement a stack and a queue using linked representation

Program Description: The stacks and queues can be implemented using linked representations. Every element of the stack and queue is stored using a node structure defined below:

```
Template<class T>
        Class Node
        {
                Public:
                        T data;
                        Node<T> *link;
        };
```

Pseudo code for Linked Stack:

- Write a class LinkedStack, with data members as : Node<T> *Top, size;
- Member functions: LinkedStack(), Push(T item), Pop(T &item), Display(), isEmpty(),etc
- Push(T item): create a temporary node temp to be pushed onto stack
    - check if stack is empty, i.e. if(Top==NULL) Top=temp;
    - if stack is not empty, then let : temp->link=Top;Top=temp;
    - increase the size of the stack
- Pop(T &itme): check if size is zero, then return an exception as stack underflowing
    - Let a temporary node temp: temp=Top;
    - Top=Top->link;
    - Item=temp->data; delete temp;
    - Decrease the size of the stacks: size--;
- Implement the main function as follows:

```
int main()
{
        try
        {
                int x;
                Stack<int> s;
                s.push(10);
                s.push(20);
                s.push(30);
                s.push(40);
                s.display();
                s.pop(x);
                cout<<"\nPopped element is "<<x<<endl;
        }
        catch ( char *c)
        { cout<<c;}
```

```
}
```

Linked Queue:

- Write a class LinkedQ with data members as: Node<T> *front, *rear, size
- Member functions: LinkeDQ(), Push(T item), Pop(T &item), Display(), isEmpty(),etc
- Push(T item): create a temporary node temp to be appended at the rear end
    - check if queue is empty, i.e. if(front==rear==NULL) front=rear=temp;
    - if queue is not empty, then let : rear->link=temp;rear=temp;
    - increase the size of the queue
- Pop(T &itme): check if size is zero, then return an exception as queue is empty
    - Let a temporary node temp: temp=front;
    - front=front->link;
    - Item=temp->data; delete temp;
    - Decrease the size of the queue: size--;
- Implement the main function as follows:

```
int main()
{
	try
	{
		int x;
		LQ<int> q;
		q.Push(10);
		q.Push(20);
		q.Push(30);
		q.Push(40);
		q.Push(50);
		q.Push(60);
		q.Display();
		q.Pop(x);
		cout<<"\n The deleted elememt is "<<x<<"\n";
		cout<<"\n The queue after deletion is \n";
		q.Display();
	}
	catch(char *c)
	{
		cout<<c;
	}
	return 0;
}
```

15. Program 7: Programs to implement Doubly Linked List and Circularly Linked List

Doubly Linked List: A DLL node contains three fields in it: data, next and previous. The next field points the address of next element in the list and previous points to previous node in the list. Since every node has two link fields and nodes are connected using two linked fields, hence it is known as doubly linked list. The node structure used in DLL is as follows:

```
template<class T>
class Node
{
        public:
                T data;
                Node<T> *next;
                Node<T> *prev;

};
```

- Declare a class DLL with following data members: Node<T> *first, size
- Write the constructor DLL() to initialize: first=NULL, size=0
- Implement the destructor to destroy all nodes of a list one after the other
- Implement the insert function with following possibilities:
    - Create a temporary node to be inserted in list: temp;
    - Temp->data=x; temp->next=NULL; temp->prev=NULL;
    - Inserting in an empty list: if (size==0) then first=temp;
    - If list is not empty, insert at first position: temp->link=first; first->prev=temp; first=temp;
    - If list is not empty, insert at last position, let p point to the last node: temp->prev=p; p->next=temp;
    - To insert at any intermediate position:

```
Node<T> *p=first;
for(int i=1;i<pos-1;i++)
        p=p->next;
        //p points to a node after which insertion takes place
temp->next=p->next;
temp->prev=p;
p->next->prev=temp;
p->next=temp;
```

- Implement the Delete function with following possibilities:
    - If list is empty, throw an exception
    - Else, if pos=1, then advance the first pointer to second node, delete the first node
    - If pos==size, let a pointer p point to last but one node, make p->link=NULL, delete the last node, and make p as the new last node.

- If deleting an element from any intermediate position:

```
for(int i=0;i<pos-2;i++)
      p=p->next;//p points to a node to left of node to be dleted
      q=p->next;
      p->next=q->next;
      q->next->prev=p;
}
x=q->data;
delete q;
size--;
```

- Decrease the list size by 1.

- Implement the main function as follows:

```
int main()
{
	try
	{
		int pos,x;
		DLL<int> d;
		d.Insert(1,100);
		d.Insert(2,200);
		d.Insert(3,300);
		d.Insert(4,400);
		d.Insert(5,500);
		d.Insert(6,600);
		d.Display();
		cout<<"\nEnter the position to delete a ndoe\n";
		cin>>pos;
		d.Delete(pos,x);
		cout<<"\nThe Deleted node's data is "<<x<<endl;
		cout<<"\n The list after deletion is \n";
		d.Display();
	}
	catch(char *c)
	{
		cout<<c<<endl;
	}
	return 0;
}
```

16: Program 8: Program for Polynomial Arithmetic using Linked Lists

Program Objective: The objective is to write a program to represent a polynomial of degree n using Linked Lists and to carry out the operations such as : add two polynomials, multiply two polynomials, display etc.

Program Description: A polynomial is an expression that contains more than two terms. A term is made up of coefficient and exponent. An example of polynomial is:

$P(x) = 4x^3+6x^2+7x+9$

A polynomial can be represented using a linked list. A linked list node can be defined such that it contains two parts- one is the coefficient and second is the corresponding exponent. The node definition may be given as shown below:

```
struct Term
{
        int coef;
        int exp;
        Term Set(int c, int e)
        {
                coef=c;
                exp=e;
                return *this;
        }
        int DispCoef()
        {
                return coef;
        }
        int DispExp()
        {
                return exp;
        }
};
```

- Declare a class Polynomial with data members as: chain<Term> Poly;
- Member functions: GetPoly(), Polynomial operator+(Polynomial &b), Polynomial Operator*(Polynomial &b), Display(), etc
- GetPoly():
    1. Accept the number of terms in the polynomial: n
    2. Prompt the use to enter n terms consisting of exp and coef;
    3. Add each term to polynomial
- Addition of two polynomials:
    - Declare a third resultant polynomial, c

- Declare two pointers Node<Term> *a and Node<Term> *b to point to first terms of each of the two polynomials to be added
- The link list representing the two polynomials are traverse till end of one of them is reached. While doing this they are compared on term basis.
- If the exponents are equal add the terms and store it in 3rd polynomial.
- If the exponents are not equal, store the term with larger exponent in the 3rd polynomial.
- During the traversal if the end of one of the list is reached then just append the remaining terms of the second polynomial to the 3rd polynomial

- mul(list<T> p1,list<T> p2): Here each term of the first polynomial is multiplied with every term of second polynomial and the new term obtained is added to the 3rd polynomial.
  - Scan the polynomial one from left to right term by term, and multiply each term of first polynomial with every term of second polynomial, and add the result to the third polynomial
  - Display the resultant product polynomial

Program Validation: The program should generate the results of polynomial addition and multiplication for the two input polynomials

```
Enter   1-for int
        2-for float                 1
Enter
        1-To enter first polynomial expression
        2-To enter second polynomial expression
        3-To add
        4-To multiply
        5-To display first expression
        6-To display second expression
        7-To exit                   1
Enter the coefficient 2
Enter the exponent 2
Enter another choice 1
Enter the coefficient 3
Enter the exponent 1
Enter another choice 2
Enter the coefficient 4
Enter the exponent 1
Enter another choice 2
Enter the coefficient 5
Enter the exponent 0
Enter another choice 5 :        2x^2+3x^1
Enter another choice 6:         4x^1+5x^0
Enter another choice 3:         2x^2+7x^1+5x^0
Enter another choice 4:         8x^3+22x^2+15x^1
Enter another choice 7
```

17: Program 9: Program to implement Hashing

Program Objective: Objective is to write a program to demonstrate Hashing

Program Description: The program implements hash table with modulo as the hash function. To avoid collisions in case of identical keys for two different elements, we use Linear Probing collision resolution technique.

Pseudo code:

- Declare a constant: TableSize=128
- Declare a class for Hash Entry :

```
Class HashEntry
{
        Public:
                Int key;
                Int value;
                HashEntry(int key, int value)
                {
                        This->key=key;
                        This->value=value;
                }
};
```

- Declare a class : HashMap to store the hash table entries with data member: **table
- Dynamically allocate memory for this array of type HashEntry
- Implement a function : HashFunction(int key) using modulo method: key%TableSize
- Implement a function Insert(key, value):
    - Compute the hash function on key to get index into hash table
    - If the element exists at the generated key (collision), then go for linear probing to find next available location in hash table
    - Insert the item at the index returned by above step
- Implement a function Search(key) as follows:
    - Compute hash function on the key
    - Check if the key at the index is same as input key, if not may be linear probing has stored the key at next index, so recomputed hash index
    - If not found return -1, else return the corresponding element
- Implement a function Delete(key): to delete a particular element whose key is supplied as input
- Write a main function using switch cases to perform operations like: Insert, Search, Delete, etc based on the user's choice

Program Validation: Ensure that the program works as indicated in the source code by running it for several combinations of key –element pairs and carry out all operations. The sample output for a given input is as follows:

```
----------------------
Operations on Hash Table
----------------------
1.Insert element into the table
2.Search element from the key
3.Delete element at a key
4.Exit
Enter your choice: 1
Enter element to be inserted: 12
Enter key at which element to be inserted: 1

----------------------
Operations on Hash Table
----------------------
1.Insert element into the table
2.Search element from the key
3.Delete element at a key
4.Exit
Enter your choice: 1
Enter element to be inserted: 24
Enter key at which element to be inserted: 2
----------------------
Operations on Hash Table
----------------------
1.Insert element into the table
2.Search element from the key
3.Delete element at a key
4.Exit
Enter your choice: 1
Enter element to be inserted: 36
Enter key at which element to be inserted: 3
----------------------
Operations on Hash Table
----------------------
1.Insert element into the table
2.Search element from the key
3.Delete element at a key
4.Exit
Enter your choice: 1
Enter element to be inserted: 48
Enter key at which element to be inserted: 4
----------------------
Operations on Hash Table
----------------------
1.Insert element into the table
2.Search element from the key
3.Delete element at a key
4.Exit
Enter your choice: 1
Enter element to be inserted: 60
Enter key at which element to be inserted: 5
----------------------
Operations on Hash Table
----------------------
1.Insert element into the table
2.Search element from the key
3.Delete element at a key
4.Exit
Enter your choice: 2
Enter key of the element to be searched: 3
Element at key 3 : 36
----------------------
Operations on Hash Table
----------------------
1.Insert element into the table
2.Search element from the key
3.Delete element at a key
4.Exit
Enter your choice: 2
Enter key of the element to be searched: 5
Element at key 5 : 60
----------------------
```

18: Program 10: Selection Sort, Insertion Sort, Heap Sort, and Shell Sort

Program Objective: The objective is to write a program to demonstrate the working of Selection sort on a given array of integers.

Program Description: The selection sort works by finding or selecting the smallest or largest element and placing it first or last in the sorted array. Then it finds the next smaller (greater) element and places it next in the sorted array.

Pseudo code:

- Declare a class Selection: with data members: T *a, int n
- Member functions: SelectionSort(), IndexOfMax(), swap(int &, int &), constructor
- indexofmax(int[],int):
    - initialize index=0;
    - for(int i=0;i<n;i++) if(a[index]<a[i]) index=I; return index;
- SelectionSort (): The index returned by indexOfMax() is used to find the largest no and swap the largest element with last element in unsorted array

Program Validation: Run the program by passing an array of integers, characters, etc and obtain a sorted sequence. The sample output for a given input is as follows:

Enter the number of elements 5

Enter the elements

4 6 3 87 2

sorted elements are

2 3 4 6 87

Program Objective: The objective is to write a program to demonstrate the working of Insertion sort on a given array of integers.

Program Description: Insertion sort iterates, consuming one input element each repetition, and growing a sorted output list. An iteration of insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.

Pseudo Code:

- Declare a class Insertion: with data members: T *a, int size
- Write the constructor to dynamically allocate memory to array and initialize the size

- Insert(T e, int i) // insert e into a sorted array of size i

```
{
        arr[0]=e; // store e at index 0
        While(e<arr[i])
        {
                Arr[i+1]=arr[i];
                Decrement I;
        }
        arr[i+1]=e;
}
```

- InsertionSort()

```
{
        For(int j=2;j<=n;j++)
        {
                T temp=arr[j];
                Insert(temp, j-1);
        }
}
```

Program Validation: Run the program with an array of inputs and obtain a sorted sequence of array elements in ascending order. The sample output for a set of given inputs is as follows:

Enter the elements in the array: 5 4 3 2 1

The Sorted Sequence is : 1 2 3 4 5

Program Objective: The objective is to sort the given sequence of elements using Max Heap data structure.

Program Description: A Max heap is a complete binary tree that is also a Max Tree, in which value at the parent is as large as value at its children. In a max heap, the root of heap stores the largest element. In sorting data using max heap, we remove the root, place it at position A[n] and replace the root by last child (w.r.t. level order) of max tree. Now since the root has changed., we need to check if the heap property is satisfied by new root. If not then reheapify the heap. This process of removing the root and reheapification continues till heap becomes empty.

HEAPSORT(A)

1. BUILD-MAX-HEAP(A)

2. for i ← length[A] downto 2

3. do exchange A[1], A[i]

4. heap-size[A] ← heap-size[A] − 1

5. MAX - HEAPIFY(A, 1)

Program Validation: Run the program with a set of inputs and obtain a sorted sequence of elements. Sample output for a given set of integers is as follows:

Enter the element 1
5
Enter the element 2
4
Enter the element 3
3
Enter the element 4
2
Enter the element 5
1
Before Sorting...
The array elements are...

5 4 3 2 1

int SOrt, value of x is...5

int SOrt, value of x is...4

int SOrt, value of x is...3

int SOrt, value of x is...2

int SOrt, value of x is...1

The array elements are...

1 2 3 4 5

19: Program 11: Quick Sort and Merge Sort

Program Objective: objective is to write a program to sort the given set of elements using Quick Sort.

Program Description: quick sort works by selecting one of the element (mostly the first element) as the pivot element and rearranges the remaining elements such that elements to the left of pivot are smaller than or equal to pivot and elements to the right of the pivot are larger than or equal to pivot. Once the pivot element is placed at its right position, then remaining elements to its left and right are recursively sorted using quick sort

Pseudo Code:

Create a class QucikSort with data members as: T *a, int n

Member functions: Constructor, QSort(low, high), Partition(a, low, high), and Display()

```
QucikSort(int size)
{
        n=size;
        a=new T [n];
        read the elements into aray;
}
Void QSort(int low, int high)
{
        If(low<high)
        {
                Obtain a partition of array (a, low, high+1) in j;
                QSort(low, j-1);
                QSort(j+1, high);
        }
}
int partition(a, low, high)
{
        Designate first element as pivot element;
        Let i be low and j be high;
        do
        {
                do
                {
                        i++;
                }while(a[i]<p);
                do
                {
                j--;
                }while(a[j]>p);
```

```
                if(i<j)
                {
                        T temp=a[i];
                        a[i]=a[j];
                        a[j]=temp;
                }

        }while(i<=j);
        T temp=a[j];
        a[j]=a[low];
        a[low]=temp;
        return j;
}
```

Display() { display the elements of sorted array;}

Program Validation: Run the program with an array as input and obtain the sorted array as result. Sample output for given input is as follows:

```
enter the element 0
65
 enter the element 1
45
 enter the element 2
66
 enter the element 3
34
 enter the element 4
778
 Array Before Sorting is....
65->45->66->34->778->
Array after sorting is...
34->45->65->66->778->
```

Program Objective: The objective is to sort a given set of elements using Merge Sort.

Program Description: The program accepts an array, sorts the array using merge sort. The given array is divided at its midpoint, (low+high)/2, and then arr[low:mid] and arr[mid+1:high] are sorted using merge sort recursively. The arrays are subdivided into sub arrays as long as they have a single element, then the merge process follows merging the subarrays into a sorted sequence.

Pseudo Code:

```
Algorithm MergeSort(low, high)
{
```

```
        If(low<high)
        {
                //divide p into sub problems
                Mid=floor((low+high)/2);
                MergeSort(low,mid);
                MergeSort(mid+1,high);
                Merge (low, mid, high);
        }
}
Algorithm Merge(low, mid, high)
{
        h=low, i=low, j=mid+1;
        while(h<=mid and j<=high) do
        {
                If(a[h]<=a[j]) then
                {
                        B[i]=a[h];
                        h++;
                }
                Else
                {
                        b[i]=a[j];
                        j++;
                }
                i++;
        }
        If(h>mid)       for(k=j;k<high;k++) { b[i]=a[k]; i++; }
        else            for(k=h;k<=mid;k++) { b[i]=a[k]; i++;}
}
```

Program Validation: Execute the program with an array of integers and obtain a sorted sequence of elements: sample output for a given set of inputs is as follows:

```
Enter the size of array..
6
Enter the 6 elements into array
65
45
57
78
45
23
Elements before sorting are..
65      45      57      78      45      23
 Elemeents after sorting are..
low is 0 mid is 2 high is5
low is 0 mid is 1 high is2
low is 0 mid is 0 high is1
in Mergelow is 0 mid is 0 high is1
in Mergelow is 0 mid is 1 high is2
low is 3 mid is 4 high is5
low is 3 mid is 3 high is4
in Mergelow is 3 mid is 3 high is4
in Mergelow is 3 mid is 4 high is5
in Mergelow is 0 mid is 2 high is5
23      45      45      57      65      78
```

20: Program 12: Tree Traversals and Graph Search Methods

Program Objective: The objective is to implement a binary search tree and its traversals such as Inorder, Preorder, and Postorder on a given binary tree.

Program Description: The binary trees in this program are represented using an array.

- root of the tree (A): array position 1
- root's left child (B): array position 2
- root's right child (C): array position 3
- left child of node in array position K: array position 2K
- right child of node in array position K: array position 2K+1

Pseudo Code:

```
#include<iostream.h>
template<class T>
class BTNode
{
	public:
		T data;
		BTNode<T> *lchild;
		BTNode<T> *rchild;
		BTNode()
		{
			lchild=rchild=0;
		}
		BTNode(T elem)
		{
			data=elem;
			rchild=lchild=0;
		}
		BTNode(BTNode<T> *lc,T elem,BTNode<T> *rc)
		{
			data=elem;
			lchild=lc;
			rchild=rc;
		}
};
template<class T>
class BinaryTree
{
	private:
		BTNode<T> *root;
		void Inorder(BTNode<T> *root);
		void Preorder(BTNode<T> *root);
```

```
		void Postorder(BTNode<T> *root);
		void Visit(BTNode<T> *root);
		int Height(BTNode<T> *root);
	public:
		BinaryTree()
		{
			root=NULL;
		}
		~BinaryTree(){}
		bool IsEmpty()
		{
			return root==NULL;
		}
		void MakeTree(T elem,BinaryTree<T> *l,BinaryTree<T> *r);
		void Inorder();
		void Preorder();
		void Postorder();
		int Height();
};
template<class T>
void BinaryTree<T>::Visit(BTNode<T> *root)
{
	cout<<root->data;
}
template<class T>
void BinaryTree<T>::Inorder(BTNode<T> *root)
{
	if(root!=NULL)
	{
		Inorder(root->lchild);
		Visit(root);
		Inorder(root->rchild);
	}
}
template<class T>
void BinaryTree<T>::Preorder(BTNode<T> *root)
{
	if(root!=NULL)
	{
		Visit(root);
		Preorder(root->lchild);
		Preorder(root->rchild);
	}
}
template<class T>
void BinaryTree<T>::Postorder(BTNode<T> *root)
```

```
{
        if(root!=NULL)
        {
                Postorder(root->lchild);
                Postorder(root->rchild);
                Visit(root);
        }
}
template<class T>
void BinaryTree<T>::MakeTree(T elem,BinaryTree<T> *l,BinaryTree<T> *r)
{
        root=new BTNode<T>(l->root,elem,r->root);
}
template<class T>
int BinaryTree<T>::Height()
{
        int h;
        h=Height(root);
        return h;
}
template<class T>
int BinaryTree<T>::Height(BTNode<T> *root)
{
        if(root==0)
        return 0;
        int lh=Height(root->lchild);
        int rh=Height(root->rchild);
        if(lh>rh)
        return ++lh;
        else
        return ++rh;
}
template<class T>
void BinaryTree<T>::Preorder()
{
        Preorder(root);

}
template<class T>
void BinaryTree<T>::Inorder()
{
        Inorder(root);
}
template<class T>
void BinaryTree<T>::Postorder()
{
```

```
        Postorder(root);
}
int main()
{
        BinaryTree<char> ll;
        BinaryTree<char> lr;
        BinaryTree<char> l;
        l.MakeTree('a',&ll,&lr);
        BinaryTree<char> rl;
        BinaryTree<char> rr;
        BinaryTree<char> r;
        r.MakeTree('b',&rl,&rr);
        BinaryTree<char> t;
        t.MakeTree('+',&l,&r);
        cout<<"Inorder traversal is";
        t.Inorder();
        cout<<"Preorder traversal is";
        t.Preorder();
        cout<<"Postorder traversal is";
        t.Postorder();
return 0;
}
```

Program Validation: Sample output for an expression tree +

a b

Inorder traversal isa + b

Preorder traversal is+ a b

Postorder traversal isa b +

Program Objective: The objective is to write a program to represent a graph using Adjacency Matrix and perform Graph Search Methods such as Breadth First Search BFS and Depth First Search DFS.

Program Description: This program represents the undirected graph using an Adjacency Matrix. An adjacency matrix is a 0-1 matrix, where a 1 indicates that edge (i,j) belongs to edge set of graph and a 0 indicates edge (i,j) does not belong to edge set of graph. To carry out BFS we use a Queue data structure and for DFS we use stacks.

Pseudo Code:

- Declare a class Graph: with following data members:

```
int **a;
int n;
int e;
int *visit;
```

- Member functions: Constructor, and following functions

```
bool isEmpty();
int NumberOfVertices();
int NumberOfEdges();
int Degree(int u);
bool EdgeExists(int u, int v);
void InsertEdge(int u, int v);
void DeleteEdge(int u, int v);
void DisplayAdjMatrix();
void BFS(int u);
void DFS();
```

- DFS Functions:

```
void Graph::DFS(int v)
    {
        visit[v]=1;
        int w;
        for(int i=1;i<=n;i++)
        {
            if(a[v][i]==1 && visit[i]==0)
            {
            //w is found such that it is adj and not visited yet
                w=i;
                cout<<w<<"\t";
                visit[w]=1;
                DFS(w);
            }
        }
    }
```

- Degree Function:

```
int Graph::Degree(int u)
{
    int sum=0;
    for(int j=1;j<=n;j++)
        sum=sum+a[u][j];//row sum
    return sum;
}
```

- Insert Edge function

```
void Graph::InsertEdge(int u, int v)
{
	if(u<1 || u>n || v<1 || v>n || a[u][v]==1)
		throw "Bad Input, No such Element in Vertex Set";
	a[u][v]=1;
	a[v][u]=1;
}
```

- BFS Function

```
void Graph:: BFS(int v)
{
	int visit[n+1];
	Queue<int> q(100);
	for(int i=1;i<=n;i++)
	{
		visit[i]=0;
	}
	for(int i=1;i<=n;i++)
		cout<<visit[i]<<"\t";
	cout<<endl;
	visit[v]=1;
	q.push(v);
	while(!q.isEmpty())
	{
		int w=q.Front();
		cout<<w<<" ";
		q.pop();
		for(int i=1;i<=n;i++)
		{
			if(a[w][i]==1 && visit[i]==0)
			{
				q.push(i);
				visit[i]=1;
			}
		}
	}
}
```

Program Validation: Execute the program by providing information about the graph using a two dimensional matrix and obtain the Traversals on the graph using DFS and BFS methods

Enter the number of Vertices in Graph

4

Enter the number of distinct unordered pairs

4

enter the Vertex u of Edge 1

1

Enter the Vertex V of Edge 1

2

enter the Vertex u of Edge 2

1

Enter the Vertex V of Edge 2

3

enter the Vertex u of Edge 3

2

Enter the Vertex V of Edge 3

4

enter the Vertex u of Edge 4

3

Enter the Vertex V of Edge 4

4

The Adjacency Matrix of G is **********

0 1 1 0

1 0 0 1

1 0 0 1

0 1 1 0

The Breadth First Search Traversal of Graph si....

1 2 3 4

The Depth First Search Traversal of Grpah is ,....

Enter the starting vertex to begin DFS

1

2 4 3

21. Program 13: AVL Trees

Program Objective: The objective is to represent the AVL trees using Linked representation and perform operations like: Insert, Delete, Rotation caused due to Insert and delete, traversals, etc.

Program Description: The program demonstrates the operations on an AVL tree using linked representation. It performs rotations caused by insert operations like: LL , RR, RL and LR .

```
include<iostream>
#include<cstdio>
#include<sstream>
#include<algorithm>
#define pow2(n) (1 << (n))
using namespace std;

/*
 * Node Declaration
 */
struct avl_node
{
   int data;
   struct avl_node *left;
   struct avl_node *right;
}*root;

/*
 * Class Declaration
 */
class avlTree
{
   public:
      int height(avl_node *);
      int diff(avl_node *);
      avl_node *rr_rotation(avl_node *);
      avl_node *ll_rotation(avl_node *);
      avl_node *lr_rotation(avl_node *);
      avl_node *rl_rotation(avl_node *);
      avl_node* balance(avl_node *);
      avl_node* insert(avl_node *, int );
      void display(avl_node *, int);
      void inorder(avl_node *);
      void preorder(avl_node *);
      void postorder(avl_node *);
      avlTree()
      {
         root = NULL;
      }
};

/*
 * Main Contains Menu
 */
int main()
{
```

```
    int choice, item;
    avlTree avl;
    while (1)
    {
        cout<<"\n---------------------"<<endl;
        cout<<"AVL Tree Implementation"<<endl;
        cout<<"\n---------------------"<<endl;
        cout<<"1.Insert Element into the tree"<<endl;
        cout<<"2.Display Balanced AVL Tree"<<endl;
        cout<<"3.InOrder traversal"<<endl;
        cout<<"4.PreOrder traversal"<<endl;
        cout<<"5.PostOrder traversal"<<endl;
        cout<<"6.Exit"<<endl;
        cout<<"Enter your Choice: ";
        cin>>choice;
        switch(choice)
        {
        case 1:
            cout<<"Enter value to be inserted: ";
            cin>>item;
            root = avl.insert(root, item);
            break;
        case 2:
            if (root == NULL)
            {
                cout<<"Tree is Empty"<<endl;
                continue;
            }
            cout<<"Balanced AVL Tree:"<<endl;
            avl.display(root, 1);
            break;
        case 3:
            cout<<"Inorder Traversal:"<<endl;
            avl.inorder(root);
            cout<<endl;
            break;
        case 4:
            cout<<"Preorder Traversal:"<<endl;
            avl.preorder(root);
            cout<<endl;
            break;
        case 5:
            cout<<"Postorder Traversal:"<<endl;
            avl.postorder(root);
            cout<<endl;
            break;
        case 6:
            exit(1);
            break;
        default:
            cout<<"Wrong Choice"<<endl;
        }
    }
    return 0;
}

/*
 * Height of AVL Tree
```

```
 */
int avlTree::height(avl_node *temp)
{
   int h = 0;
   if (temp != NULL)
   {
      int l_height = height (temp->left);
      int r_height = height (temp->right);
      int max_height = max (l_height,
r_height);
      h = max_height + 1;
   }
   return h;
}

/*
 * Height Difference
 */
int avlTree::diff(avl_node *temp)
{
   int l_height = height (temp->left);
   int r_height = height (temp->right);
   int b_factor= l_height - r_height;
   return b_factor;
}

/*
 * Right- Right Rotation
 */
avl_node       *avlTree::rr_rotation(avl_node
*parent)
{
   avl_node *temp;
   temp = parent->right;
   parent->right = temp->left;
   temp->left = parent;
   return temp;
}
/*
 * Left- Left Rotation
 */
avl_node       *avlTree::ll_rotation(avl_node
*parent)
{
   avl_node *temp;
   temp = parent->left;
   parent->left = temp->right;
   temp->right = parent;
   return temp;
}

/*
 * Left - Right Rotation
 */
avl_node       *avlTree::lr_rotation(avl_node
*parent)
{
   avl_node *temp;
   temp = parent->left;
   parent->left = rr_rotation (temp);
   return ll_rotation (parent);
}
```

```
/*
 * Right- Left Rotation
 */
avl_node        *avlTree::rl_rotation(avl_node
*parent)
{
    avl_node *temp;
    temp = parent->right;
    parent->right = ll_rotation (temp);
    return rr_rotation (parent);
}

/*
 * Balancing AVL Tree
 */
avl_node            *avlTree::balance(avl_node
*temp)
{
    int bal_factor = diff (temp);
    if (bal_factor > 1)
    {
        if (diff (temp->left) > 0)
            temp = ll_rotation (temp);
        else
            temp = lr_rotation (temp);
    }
    else if (bal_factor < -1)
    {
        if (diff (temp->right) > 0)
            temp = rl_rotation (temp);
        else
            temp = rr_rotation (temp);
    }
    return temp;
}

/*
 * Insert Element into the tree
 */
avl_node    *avlTree::insert(avl_node    *root,
int value)
{
    if (root == NULL)
    {
        root = new avl_node;
        root->data = value;
        root->left = NULL;
        root->right = NULL;
        return root;
    }
    else if (value < root->data)
    {
        root->left = insert(root->left, value);
        root = balance (root);
    }
    else if (value >= root->data)
    {
        root->right = insert(root->right, value);
        root = balance (root);
    }
    return root;
```

```
}

/*
 * Display AVL Tree
 */
void avlTree::display(avl_node *ptr, int level)
{
   int i;
   if (ptr!=NULL)
   {
      display(ptr->right, level + 1);
      printf("\n");
      if (ptr == root)
      cout<<"Root -> ";
      for (i = 0; i < level && ptr != root; i++)
         cout<<"        ";
      cout<<ptr->data;
      display(ptr->left, level + 1);
   }
}

/*
 * Inorder Traversal of AVL Tree
 */
void avlTree::inorder(avl_node *tree)
{
   if (tree == NULL)
      return;
   inorder (tree->left);
   cout<<tree->data<<"  ";
   inorder (tree->right);
}
/*
 * Preorder Traversal of AVL Tree
 */
void avlTree::preorder(avl_node *tree)
{
   if (tree == NULL)
      return;
   cout<<tree->data<<"  ";
   preorder (tree->left);
   preorder (tree->right);

}

/*
 * Postorder Traversal of AVL Tree
 */
void avlTree::postorder(avl_node *tree)
{
   if (tree == NULL)
      return;
   postorder ( tree ->left );
   postorder ( tree ->right );
   cout<<tree->data<<"  ";
}
```

Program Validation: Sample output of the program is as follows:

```
$ g++ avl_tree.cpp
$ a.out
----------------------
```

```
AVL Tree Implementation
 --------------------
1.Insert Element into the tree
2.Display Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice: 2
Tree is Empty
 ---------------------
AVL Tree Implementation
 ---------------------
1.Insert Element into the tree
2.Display Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice: 1
Enter value to be inserted: 8
 ---------------------
AVL Tree Implementation
 ---------------------
1.Insert Element into the tree
2.Display Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice: 2
Balanced AVL Tree:
 Root -> 8
 ---------------------
AVL Tree Implementation
 ---------------------
1.Insert Element into the tree
2.Display Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice: 1
Enter value to be inserted: 5
 ---------------------
AVL Tree Implementation
 ---------------------
1.Insert Element into the tree
2.Display Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice: 2
Balanced AVL Tree:
 Root -> 8
            5
 ---------------------
AVL Tree Implementation
 ---------------------
1.Insert Element into the tree
2.Display Balanced AVL Tree
```

```
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice: 1
Enter value to be inserted: 4
 ---------------------
AVL Tree Implementation
---------------------
1.Insert Element into the tree
2.Display Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice: 2
Balanced AVL Tree:
            8
Root -> 5
            4
---------------------
AVL Tree Implementation
 ---------------------
1.Insert Element into the tree
2.Display Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice: 1
Enter value to be inserted: 11
 ---------------------
AVL Tree Implementation
---------------------
1.Insert Element into the tree
2.Display Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice: 2
Balanced AVL Tree:
                  11
            8
Root -> 5
            4
---------------------
AVL Tree Implementation
 ---------------------
1.Insert Element into the tree
2.Display Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice: 1
Enter value to be inserted: 15
 ---------------------
AVL Tree Implementation
 ---------------------
1.Insert Element into the tree
2.Display Balanced AVL Tree
```

3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice: 2
Balanced AVL Tree:

```
        15
    11
        8
```

Root -> 5

```
    4
```

AVL Tree Implementation

1.Insert Element into the tree
2.Display Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice: 1
Enter value to be inserted: 3

AVL Tree Implementation

1.Insert Element into the tree
2.Display Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice: 2
Balanced AVL Tree:

```
        15
    11
        8
```

Root -> 5

```
    4
        3
```

AVL Tree Implementation

1.Insert Element into the tree
2.Display Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice: 1
Enter value to be inserted: 6

ANNEXURE– I: Data Structures Laboratory – OU Syllabus

Instrucitons	3 Periods per week
Duration of University Examination	3 Hours
University Examinations	50 Marks
Sessional	25 Marks

List of Experiments:

1. Implementation of Array as an ADT
2. Implementation of String as an ADT
3. Implementation of Stacks and Queues
4. Infix to Postfix conversion and postfix evaluation
5. Polynomial Arithmetic using Linked Lists
6. Implementation of Binary Search and Hashing
7. Implementation of Selection, Shell, Merge and Quick Sort
8. Implementation of Tree Traversal on Binary Trees
9. Implementation of Heap Sort
10. Implementation of Operations on AVL Trees
11. Implementation of Traversals on Graphs
12. Implementation of Splay Trees