CS231: DATA STRUCTURES LAB

LAB MANUAL

Department of Computer Science and Engineering

College of Engineering Trivandrum

Kerala

Vision and Mission of the Institution

Vision

National level excellence and international visibility in every facet of engineering research and education.

Mission

I. To facilitate quality transformative education in engineering and management.
II. To foster innovations in technology and its application for meeting global challenges.
III. To pursue and disseminate quality research.
IV. To equip, enrich and transform students to be responsible professionals for better service to humanity.

Vision and Mission of the Department

Vision

To be a centre of excellence in education and research in the frontier areas of Computer Science and Engineering.

Mission

I. To facilitate quality transformative education in Computer Science and Engineering.
II. To promote quality research and innovation in technology for meeting global challenges.
III. To transform students to competent professionals to cater to the needs of the society.

Program Educational Objectives (PEOs)

Consistent with the stated vision and mission of the institute and the Department, the faculty members of the Department strive to train the students to become competent technologists with integrity and social commitment.
Within a short span of time after graduation, the graduates shall:

PEO1. Achieve professional competency in the field of Computer Science and Engineering.
PEO2. Acquire domain knowledge to pursue higher education and research.
PEO3. Become socially responsible engineers with good leadership qualities and effective interpersonal
skills.

Program Specific Outcomes (PSOs):

PSO1: Model computational problems by applying mathematical concepts and design solutions using suitable data structures and algorithmic techniques.

PSO2: Design and develop solutions by following standard software engineering principles and implement by using suitable programming languages and platforms.

PSO3: Develop system solutions involving both hardware and software modules
Program Outcomes (POs)

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, review 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.

PO12: 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.

Course Summary

Prerequisite and Course Description

CS205 Data Structure is the prerequisite.

Course Educational Objectives

1. To implement basic linear and non-linear data structures and their major operations.

2. To implement applications using these data structures.

3. To implement algorithms for various sorting techniques.

Course Outcomes(COs)

On completion of the course student will be able to
CO1. Build applications using linear data structures.
CO2. Build applications using non-linear data structures.
CO3. Apply sorting and searching techniques for various real world applications

CO - PO Mapping

Course outcome	PO1	PO2	PO3	PO 4	PO5	PO 6	PO7	PO8	PO 9	PO1 0	PO1 1	PO1 2	PS O 1	PS O2	PS O3
CO1	3	3	3	3		1	1	1	2	1		1	3	3	3
CO2	3	3	3	3		1	1	1	2	1		1	3	3	3
CO3	3	3	3	3		1	1	1	2	1		1	3	3	3
CO	3	3	3	3		1	1	1	2	1		1	3	3	3

Evaluation Plan

Students will be evaluated based on the following criteria:

Practical record/Output = 60
Regular lab viva = 10
Final written test/Quiz = 30

Syllabus

List of Exercises/Experiments : (Minimum 12 are to be done)

1. Implementation of Stack and Multiple stacks using one dimensional array. **

2. Application problems using stacks: Infix to post fix conversion, postfix and pre-fix evaluation, MAZE problem etc. **

3. Implementation of Queue, DEQUEUE and Circular queue using arrays.

4. Implementation of various linked list operations. **

5. Implementation of stack, queue and their applications using linked list.

6. Implementation of trees using linked list

7. Representation of polynomials using linked list, addition and multiplication of polynomials. **

8. Implementation of binary trees using linked lists and arrays- creations, insertion, deletion and traversal. **

9. Implementation of binary search trees – creation, insertion, deletion, search

10. Application using trees

11. Implementation of sorting algorithms – bubble, insertion, selection, quick (recursive and non-recursive), merge sort (recursive and non-recursive), and heap sort.**

12. Implementation of searching algorithms – linear search, binary search.**

13. Representation of graphs and computing various parameters (in degree, out degree etc.) - adjacency list, adjacency matrix.

14. Implementation of BFS, DFS for each representation.

15. Implementation of hash table using various mapping functions, various collision and

overflow resolving schemes.**

16. Implementation of various string operations.

17. Simulation of first-fit, best-fit and worst-fit allocations.

18. Simulation of a basic memory allocator and garbage collector using doubly linked list.

** mandatory.

Cycle – I
Searching , Sorting and String operations

Sl. No.	List of Programs
1	Write a menu- driven program to implement a) Linear search b) Binary search
2	Write a program to implement a) Bubble sort b) Insertion sort c) Selection sort d) Quick sort e) Merge sort
3	Write a menu-driven program to implement the following string operations without using library functions : a) String length b) String concatenation c) String copy d) String comparison e) Substring search and replacement f) Count the number of vowels, consonants and number of words in a sentence

Cycle – II
Stacks, Queues and Applications (using arrays)

Sl. No.	List of Programs
1	Write a menu driven program to implement stack operations using array a) Push b) Pop c) Display
2	Write a menu driven program to implement queue operations using array a) Insert b) Delete c) Display
3	Wr Write a menu driven program to implement Input restricted DEQUEUE and output restricted DEQUEUE using array.
4	Wr Write a menu driven program to implement circular queue using array.
5	Wr Write a program to implement infix to postfix conversion using stack

6 Wr	Write a menu-driven program for the following a) Postfix evaluation b) Prefix evaluation
7	Write a program to implement multiple stacks using array

Cycle – III

Linked Lists and applications

Sl. No.	List of Programs
1	Write a menu-driven program to implement the following singly linked list operations a) Insertion at head b) Deletion at head c) Insertion at tail d) Deletion at tail e) Insertion at a position f) Deletion at a position g) Insertion before a data h) Deletion before a data
2	Write a menu driven program to implement stack operations using linked list a) Push b) Pop c) Display
3 Wr	Write a menu driven program to implement polynomial addition and multiplication using linked list

Cycle – IV

Trees ,Graphs and Hashing

Sl. No.	List of Programs
1	Write a menu driven program to implement binary search tree and perform the following operations: a) Creation b) Insertion c) Deletion d) Search e) Traversal
2	Write a program to implement BFS and DFS traversal algorithms on a graph

3	Wr	Write a program to implement hash table using various mapping functions, various collision and overflow resolving schemes

IMPORTANT DATA STRUCTURES

ARRAY

Array, is a data structure consisting of a collection of elements (values or variables), each identified by at least one *array index* or *key*. An array is stored such that the position of each element can be computed from its index tuple by a mathematical formula. The simplest type of data structure is a linear array, also called one-dimensional array.

For example, an array of 10 32-bit integer variables, with indices 0 through 9, may be stored as 10 words at memory addresses 2000, 2004, 2008, ... 2036, so that the element with index i has the address $2000 + 4 \times i$.

The memory address of the first element of an array is called first address or foundation address.

Because the mathematical concept of a matrix can be represented as a two-dimensional grid, two-dimensional arrays are also sometimes called matrices. In some cases the term "vector" is used in computing to refer to an array, although tuples rather than vectors are the more mathematically correct equivalent. Tables are often implemented in the form of arrays, especially lookup tables; the word *table* is sometimes used as a synonym of *array*.

Arrays are among the oldest and most important data structures, and are used by almost every program. They are also used to implement many other data structures, such as lists and strings. They effectively exploit the addressing logic of computers. In most modern computers and many external storage devices, the memory is a one-dimensional array of words, whose indices are their addresses. Processors, especially vector processors, are often optimized for array operations.

Arrays are useful mostly because the element indices can be computed at run time. Among other things, this feature allows a single iterative statement to process arbitrarily many

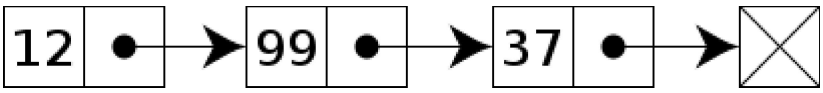
elements of an array. For that reason, the elements of an array data structure are required to have the same size and should use the same data representation. The set of valid index tuples and the addresses of the elements (and hence the element addressing formula) are usually, but not always fixed while the array is in use.

The term *array* is often used to mean array data type, a kind of data type provided by most high-level programming languages that consists of a collection of values or variables that can be selected by one or more indices computed at run-time. Array types are often implemented by array structures; however, in some languages they may be implemented by hash tables, linked lists, search trees, or other data structures.

The term is also used, especially in the description of algorithms, to mean associative array or "abstract array", a theoretical computer science model (an abstract data type or ADT) intended to capture the essential properties of arrays. Arrays have better cache locality as compared to linked lists.

LINKED LIST

Linked list is a linear collection of data elements, whose order is not given by their physical placement in memory. Instead, each element POINTSto the next. It is a DATA STRUCTURE consisting of a collection of NODES which together represent a SEQUENCE. In its most basic form, each node contains: DATA, and a REFERENCE (in other words, a *link*) to the next node in the sequence. This structure allows for efficient insertion or removal of elements from any position in the sequence during iteration. More complex variants add additional links, allowing more efficient insertion or removal of nodes at arbitrary positions. A drawback of linked lists is that access time is linear (and difficult to PIPELINE). Faster access, such as random access, is not feasible. ARRAYS have better CACHE LOCALITY compared to linked lists.



A linked list whose nodes contain two fields: an integer value and a link to the next node. The last node is linked to a terminator used to signify the end of the list.

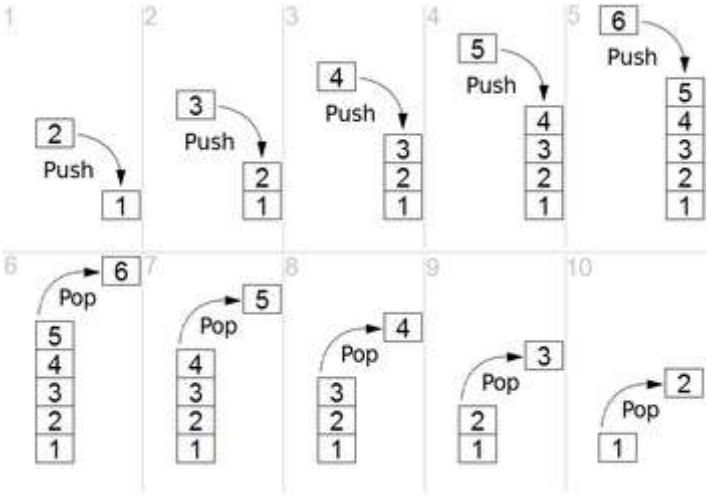
Linked lists are among the simplest and most common data structures. They can be used to implement several other common ABSTRACT DATA TYPES, including LISTS, STACKS, QUEUES, ASSOCIATIVE ARRAYS, and S-EXPRESSIONS, though it is not uncommon to implement those data structures directly without using a linked list as the basis.

The principal benefit of a linked list over a conventional ARRAY is that the list elements can be easily inserted or removed without reallocation or reorganization of the entire structure because the data items need not be stored CONTIGUOUSLY in memory or on disk, while restructuring an array at RUN-TIME is a much more expensive operation. Linked lists allow insertion and removal of

nodes at any point in the list, and allow doing so with a constant number of operations by keeping the link previous to the link being added or removed in memory during list traversal.

On the other hand, since simple linked lists by themselves do not allow RANDOM ACCESS to the data or any form of efficient indexing, many basic operations—such as obtaining the last node of the list, finding a node that contains a given datum, or locating the place where a new node should be inserted—may require iterating through most or all of the list elements. The advantages and disadvantages of using linked lists are given below. Linked list are dynamic, so the length of list can increase or decrease as necessary. Each node does not necessarily follow the previous one physically in the memory.

STACK



Simple representation of a stack runtime with *push* and *pop* operations.

A *stack* is an abstract data type that serves as a collection of elements, with two principal operations:

1. **push**, which adds an element to the collection, and
2. **pop**, which removes the most recently added element that was not yet removed.

The order in which elements come off a stack gives rise to its alternative name, **LIFO (last in, first out)**. Additionally, a peek operation may give access to the top without modifying the stack. The name "stack" for this type of structure comes from the analogy to a set of physical items stacked on top of each other, which makes it easy to take an item off the top of the stack, while getting to an item deeper in the stack may require taking off multiple other items first.
Considered as a linear data structure, or more abstractly a sequential collection, the push and pop operations occur only at one end of the structure, referred to as the *top* of the stack. This makes it possible to implement a stack as a singly linked list and a pointer to the top element. A stack may be implemented to have a bounded capacity. If the stack is full and does not contain enough space to accept an entity to be pushed, the stack is then considered to be in an overflow state. The pop operation removes an item from the top of the stack.

Applications

Expression Evaluation and Conversion

Calculators employing reverse Polish notation use a stack structure to hold values. Expressions can be represented in prefix, postfix or infix notations and conversion from one form to another may be accomplished using a stack. Many compilers use a stack for parsing the syntax of expressions, program blocks etc. before translating into low level code. Most programming languages are context-free languages, allowing them to be parsed with stack based machines.

Backtracking

Another important application of stacks is backtracking. Consider a simple example of finding the correct path in a maze. There are a series of points, from the starting point to the destination. We start from one point. To reach the final destination, there are several paths. Suppose we choose a random path. After following a certain path, we realise that the path we have chosen is wrong. So we need to find a way by which we can return to the beginning of that path. This can be done with the use of stacks. With the help of stacks, we remember the point where we have reached. This is done by pushing that point into the stack. In case we end up on the wrong path, we can pop the last point from the stack and thus return to the last point and continue our quest to find the right path. This is called backtracking.
The prototypical example of a backtracking algorithm is depth-first search, which finds all vertices of a graph that can be reached from a specified starting vertex. Other applications of backtracking involve searching through spaces that represent potential solutions to an optimization problem. Branch and bound is a technique for performing such backtracking searches without exhaustively searching all of the potential solutions in such a space.

Compile time Memory Management

A number of programming languages are stack-oriented, meaning they define most basic operations (adding two numbers, printing a character) as taking their arguments from the stack, and placing any return values back on the stack. For example, PostScript has a return stack and an operand stack, and also has a graphics state stack and a dictionary stack. Many virtual machines are also stack-oriented, including the p-code machine and the Java Virtual Machine.
Almost all calling conventions—the ways in which subroutines receive their parameters and return results—use a special stack (the "call stack") to hold information about procedure/function calling and nesting in order to switch to the context of the called function and restore to the caller function when the calling finishes. The functions follow a runtime protocol between caller and called to save arguments and return value on the stack. Stacks are an important way of supporting nested or recursive function calls. This type of stack is used implicitly by the compiler to support CALL and RETURN statements (or their equivalents) and is not manipulated directly by the programmer.

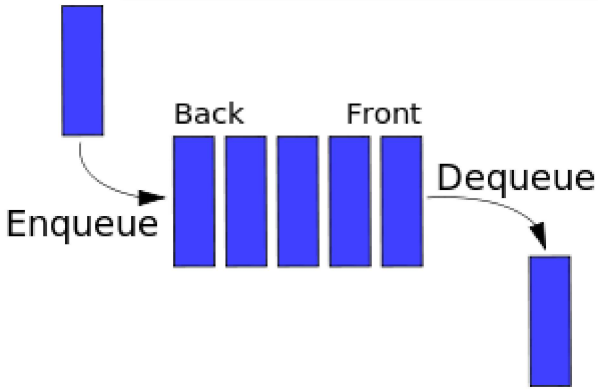
QUEUE

A **queue** is a COLLECTION in which the entities in the collection are kept in order and the principal (or only) operations on the collection are the addition of entities to the rear terminal position, known as *enqueue*, and removal of entities from the front terminal position, known as *dequeue*. This makes the queue a FIRST-IN-FIRST-OUT (FIFO) DATA STRUCTURE. In a FIFO data structure, the first element added to the queue will be the first one to be removed. This is equivalent to the requirement that once a new element is added, all elements that were added before have to be removed before the new element can be removed. Often a *PEEK* or *front* operation is also entered, returning the value of the front element without dequeuing it. A queue is an example of a LINEAR DATA STRUCTURE, or more abstractly a sequential collection.

Queue

TIME COMPLEXITY in BIG O NOTATION

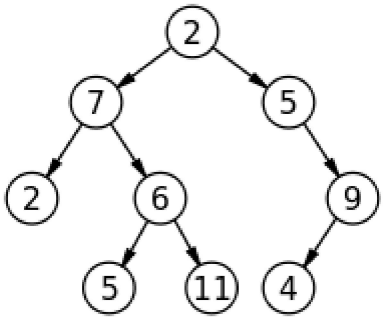
Algorithm	Average	Worst case
Space	O(*n*)	O(*n*)
Search	O(*n*)	O(*n*)
Insert	O(1)	O(1)
Delete	O(1)	O(1)



Representation of a FIFO (first in, first out) queue

Queues provide services in COMPUTER SCIENCE, TRANSPORT, and OPERATIONS RESEARCH where various entities such as data, objects, persons, or events are stored and held to be processed later. In these contexts, the queue performs the function of a BUFFER.
Queues are common in computer programs, where they are implemented as data structures coupled with access routines, as an ABSTRACT DATA STRUCTURE or in object-oriented languages as classes. Common implementations are CIRCULAR BUFFERS.

TREE



A simple unordered tree; in this diagram, the node labelled 7 has two children, labelled 2 and 6, and one parent, labelled 2. The root node, at the top, has no parent.

A **tree** is a widely used abstract data type (ADT)—or data structure implementing this ADT—that simulates a hierarchical tree structure, with a root value and subtrees of children with a parent node, represented as a set of linked nodes.

A tree data structure can be defined recursively as a collection of nodes (starting at a root node), where each node is a data structure consisting of a value, together with a list of references to nodes (the "children"), with the constraints that no reference is duplicated, and none points to the root.
Alternatively, a tree can be defined abstractly as a whole (globally) as an ordered tree, with a value assigned to each node. Both these perspectives are useful: while a tree can be analyzed mathematically as a whole, when actually represented as a data structure it is usually represented and worked with separately by node (rather than as a set of nodes and an adjacency list of edges between nodes, as one may represent a digraph, for instance). For example, looking at a tree as a whole, one can talk about "the parent node" of a given node, but in general as a data structure a given node only contains the list of its children, but does not contain a reference to its parent (if any).

TERMINOLOGY USED IN TREES

Root

The top node in a tree.

Child

A node directly connected to another node when moving away from the root.

Parent

The converse notion of a child.

Siblings

A group of nodes with the same parent.

Descendant

A node reachable by repeated proceeding from parent to child. Also known as subchild.

Ancestor

A node reachable by repeated proceeding from child to parent.

Leaf

External node (not common)

A node with no children.

Branch node

Internal node

A node with at least one child.

Degree

For a given node, its number of children. A leaf is necessarily degree zero.

Edge

The connection between one node and another.

Path

A sequence of nodes and edges connecting a node with a descendant.

Level

The level of a node is defined as: 1 + the number of edges between the node and the root.

Depth

The depth of a node is defined as: the number of edges between the node and the root.

Height of node

The height of a node is the number of edges on the longest path between that node and a leaf.

Height of tree

The height of a tree is the height of its root node.

Forest

A forest is a set of n ≥ 0 disjoint trees.

BINARY SEARCH TREE

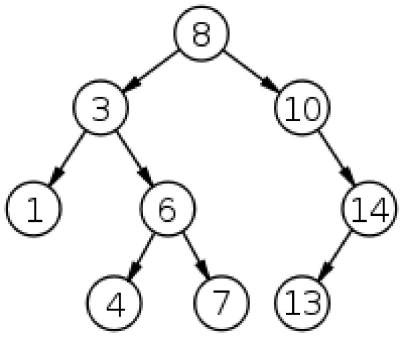
Binary search trees(BST), sometimes called **ordered** or **sorted binary trees**, are a particular type of container: data structures that store "items" (such as numbers, names etc.) in memory. They allow fast lookup, addition and removal of items, and can be used to implement either dynamic sets of items, or lookup tables that allow finding an item by its *key* (e.g., finding the phone number of a person by name).

Binary search tree

Type	Tree
Invented	1960
Invented by	P.F. Windley, A.D. Booth, A.J.T. Colin, and T.N. Hibbard

Time complexity in big O notation

Algorithm	Average	Worst case
Space	O(*n*)	O(*n*)
Search	O(log *n*)	O(*n*)
Insert	O(log *n*)	O(*n*)
Delete	O(log *n*)	O(*n*)



A binary search tree of size 9 and depth 3, with 8 at the root. The leaves are not drawn.

Binary search trees keep their keys in sorted order, so that lookup and other operations can use the principle of binary search: when looking for a key in a tree (or a place to insert a new key), they traverse the tree from root to leaf, making comparisons to keys stored in the nodes of the tree and deciding, on the basis of the comparison, to continue searching in the left or right subtrees. On average, this means that each comparison allows the operations to skip about half of the tree, so that each lookup, insertion or deletion takes time proportional to the logarithm of the number of items stored in the tree. This is much better than the linear time required to find items by key in an (unsorted) array, but slower than the corresponding operations on hash tables.

Several variants of the binary search tree have been studied in computer science; this article deals primarily with the basic type, making references to more advanced types when appropriate.

DEFINITION

A binary search tree is a rooted binary tree, whose internal nodes each store a key (and optionally, an associated value) and each have two distinguished sub-trees, commonly denoted *left* and *right*. The tree additionally satisfies the binary search property, which states that the key in each node must be greater than or equal to any key stored in the left sub-tree, and less than or equal to any key stored in the right sub-tree. The leaves (final nodes) of the tree contain no key and have no structure to distinguish them from one another.

Frequently, the information represented by each node is a record rather than a single data element. However, for sequencing purposes, nodes are compared according to their keys rather than any part of their associated records. The major advantage of binary search trees over other data structures is that the related sorting algorithms and search algorithms such as in-order traversal can be very efficient; they are also easy to code.

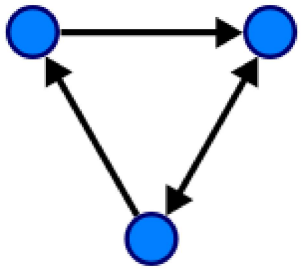
Binary search trees are a fundamental data structure used to construct more abstract data structures such as sets, multisets, and associative arrays.

- When inserting or searching for an element in a binary search tree, the key of each visited node has to be compared with the key of the element to be inserted or found.
- The shape of the binary search tree depends entirely on the order of insertions and deletions, and can become degenerate.
- After a long intermixed sequence of random insertion and deletion, the expected height of the tree approaches square root of the number of keys, $\sqrt{n}$, which grows much faster than log n.
- There has been a lot of research to prevent degeneration of the tree resulting in worst case time complexity of O(n) (for details see section Types).

Order Relation

Binary search requires an order relation by which every element (item) can be compared with every other element in the sense of a total preorder. The part of the element which effectively takes place in the comparison is called its *key*. Whether duplicates, i.e. different elements with same key, shall be allowed in the tree or not, does not depend on the order relation, but on the application only.
In the context of binary search trees a total preorder is realized most flexibly by means of a three-way comparison subroutine.

GRAPH



A directed graph with three vertices (blue circles) and three edges (black arrows).

A gr**aph** is an abstract data type that is meant to implement the undirected graph and directed graph concepts from mathematics; specifically, the field of graph theory.

A graph data structure consists of a finite (and possibly mutable) set of *vertices* or *nodes* or *points*, together with a set of unordered pairs of these vertices for an undirected graph or a set of ordered pairs for a directed graph. These pairs are known as *edges*, *arcs*, or *lines* for an undirected graph and as *arrows*, *directed edges*, *directed arcs*, or *directed lines* for a directed graph. The vertices may be part of the graph structure, or may be external entities represented by integer indices or references.

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

REPRESENTATIONS

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

Adjacency list

Vertices are stored as records or objects, and every vertex stores a list of adjacent vertices. This data structure allows the storage of additional data on the vertices. Additional data can be stored if edges are also stored as objects, in which case each vertex stores its incident edges and each edge stores its incident vertices.

Adjacency matrix

A two-dimensional matrix, in which the rows represent source vertices and columns represent destination vertices. Data on edges and vertices must be stored externally. Only the cost for one edge can be stored between each pair of vertices.

Incidence matrix

A two-dimensional Boolean matrix, in which the rows represent the vertices and columns represent the edges. The entries indicate whether the vertex at a row is incident to the edge at a column.

Adjacency lists are generally preferred because they efficiently represent sparse graphs. An adjacency matrix is preferred if the graph is dense, that is the number of edges $|E|$ is close to the number of vertices squared, $|V|^2$, or if one must be able to quickly look up if there is an edge connecting two vertices.

HASH TABLES

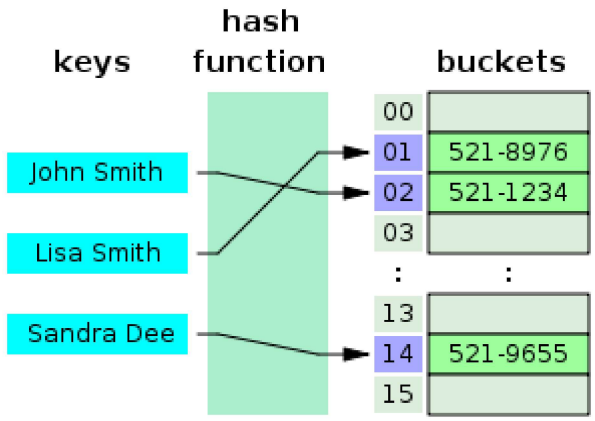
Hash table (**hash map**) is a data structure that implements an associative array abstract data type, a structure that can map keys to values. A hash table uses a hash function to compute an *index* into an array of *buckets* or *slots*, from which the desired value can be found.

Hash table

Type	Unordered associative array
Invented	1953

Time complexity in big O notation

Algorithm	Average	Worst case
Space	O(*n*)[1]	O(*n*)
Search	O(1)	O(*n*)
Insert	O(1)	O(*n*)
Delete	O(1)	O(*n*)



A small phone book as a hash table

Ideally, the hash function will assign each key to a unique bucket, but most hash table designs employ an imperfect hash function, which might cause hash *collisions* where the hash function generates the same index for more than one key. Such collisions must be accommodated in some way.

In a well-dimensioned hash table, the average cost (number of instructions) for each lookup is independent of the number of elements stored in the table. Many hash table designs also allow arbitrary insertions and deletions of key-value pairs, at (amortized) constant average cost per operation.

In many situations, hash tables turn out to be on average more efficient than search trees or any other table lookup structure. For this reason, they are widely used in many kinds of computer software, particularly for associative arrays, database indexing, caches, and sets.

HASHING

The idea of hashing is to distribute the entries (key/value pairs) across an array of *buckets*. Given a key, the algorithm computes an *index*that suggests where the entry can be found:

```
index = f(key, array_size)
```

Often this is done in two steps:

```
hash = hashfunc(key)
index = hash % array_size
```

In this method, the *hash* is independent of the array size, and it is then *reduced* to an index (a number between `0` and `array_size - 1`) using the modulo operator (`%`).

In the case that the array size is a power of two, the remainder operation is reduced to masking, which improves speed, but can increase problems with a poor hash function.

MEMORY MANAGEMENT

Different types of variable memory management techniques are:

- **Best Fit Allocation:**

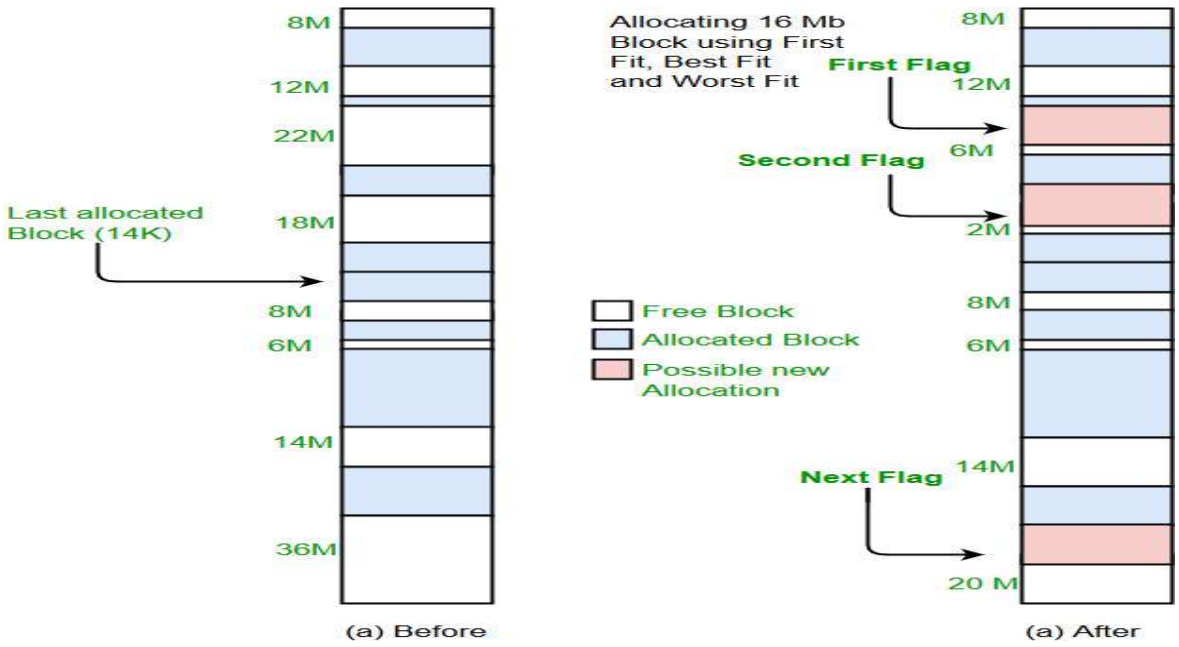
In this method, the OS has to search the entire list, or it can keep it sorted and stop when it hits an entry which has a size larger than the size of new process. This algorithm produces the smallest left over block. However, it requires more time for searching all the list or sorting it

If sorting is used, merging the area released when a process terminates to neighboring free blocks, becomes complicated. Although, best fit minimizes the wastage space, it consumes a lot of processor time for searching the block which is close to required size. Also, Best-fit may perform poorer than other algorithms in some cases.

- **First-Fit Allocation:**

In the first fit algorithm, the allocator keeps a list of free blocks (known as the free list) and, on receiving a request for memory, scans along the list for the first block that is large enough to satisfy the request. If the chosen block is significantly larger than that requested, then it is usually split, and the remainder added to the list as another free block.

The first fit algorithm performs reasonably well, as it ensures that allocations are quick. When recycling free blocks, there is a choice as to where to add the blocks to the free list—effectively in what order the free list is kept. Its advantage is that it is the fastest search as it searches only the first block i.e. enough to assign a process.It may have problems of not allowing processes to take space even if it was possible to allocate..



An example of memory allocation using various techniques

- **Worst Fit:**

This policy allocates the process to the largest available free block of memory. This leads to elimination of all large blocks of memory, thus requests of processes for large memory cannot be met.

Merits

It is also fast but slower than first fit. Its algorithm is simple

It is easy to implement

Demerits

External Fragmentation ,large amount of space may be wasted.

It requires sorting and searching free holes

It consumes time

It does not provides the optimal solution

Cycle1 : Searching , Sorting and String operations

Exp I : Write a menu- driven program to implement a) Linear search b) Binary search

Problem Description : Searching algorithm is any algorithm which is used to retrieve information stored within some data structure. Search algorithms can be classified into two . Linear search algorithms check every element in the data structure in a linear fashion to find the search value(key). Binary, or half interval searches, repeatedly target the center of the search structure and divide the search space in half . The only prerequisite for binary search is that the search array must be a sorted one.

ALGORITHMS:

1. Linear Search

Input Specification : An Input array A with size n and a search value, key

Output Specification : If the element is present in the array, return the index position of the search key, else return an ERROR message

LINEAR SEARCH(A,n,key)

```
1. BEGIN
2. i=0, pos=-1
3. WHILE(i<n)
4.         IF(a[i]==item)   //comparing each element with given element
5.                 pos=i
6.                 break
7.         Increment i
8. END WHILE
9. IF (pos == -1) print “ERROR – KEY NOT FOUND”
10. ELSE return pos+1
11. END LINEAR SEARCH
```

2. Binary Search

Input Specification : An Input sorted array A with size n and a search value, key

Output Specification : If the element is present in the array, return the index position of the search key, else return an ERROR message

BINARY SEARCH(A,n,key)

```
1. BEGIN
```

```
2.  low=0, high=n, pos=-1
3.  WHILE(low<=high)
4.          mid=(low+high)/2                    //finding the middle element
5.          IF(a[mid]==item)
6.                  pos=mid
7.                  break
8.          ELSE  IF(a[mid]>item)
9.                  high=mid-1
10. ELSE
11.         low=mid+1
12. END WHILE
13. IF (pos == -1) print “ERROR – KEY NOT FOUND”
14.       ELSE return pos+1
15. END binary_search
```

SAMPLE INPUT & OUTPUT

a) **LINEAR SEARCH**

Input 1 : 3 8 1 9 6 4 2, key = 9

Output : 3

Input 2 : 3 8 1 9 6 4 2, key = 7

Output : ERROR

b) **BINARY SEARCH**

Input 1 : 1 2 3 4 6 8 9, key = 4

Output : 3

Input 2 : 1 2 3 4 6 8 9, key = 7

Output : ERROR

Exp2 : Sorting Algorithms

Problem Description : Sorting algorithm is any algorithm which is used to arrange data in either ascending order or descending order. There are a number of

sorting algorithms which vary in their effiiciency. Commonly used sorting algorithms are Bubble sort, Selection sort, Insertion sort, Quick sort, Heap sort and merge sort.

Input Specification : An array of size n

Output Specification : The input array arranged in either ascending or descending order

ALGORITHMS

1. **Bubble Sort**

BUBBLE SORT(A,n)

```
1.  BEGIN
2.  i=0
3.  WHILE(i<n)
4.          j=0
5.          WHILE(j<n)
6.                  IF(A[j]>A[j+1])
7.                          t=A[j]                  //swapping
8.                          A[j]=A[j+1]
9.                          A[j+1]=t
10.                 END IF
11.                 j = j + 1
12.         END WHILE
13.         i = i + 1
14. END WHILE
15. END BUBBLE SORT
```

2. Insertion Sort

INSERTION SORT(A, n)

```
1.  BEGIN
2.  FOR j=2 to n do
3.          key= A[j]
4.          i= j-1
5.          WHILE i>0 & A[i] > key
6.                  A[i+1]= A[i]
7.                  i= i-1
8.          ENDWHILE
9.          A[i+1]= key
10. END FOR
11. END INSERTION SORT
```

3. Selection Sort

SELECTION SORT(A,n)

```
1.  BEGIN
2.  FOR  i = 1 to n-1 do
3.          min = i
4.          FOR  j = i+1 to n do
5.                  IF A[j] < A[min]
6.                          min = j
7.                  END IF
8.           END FOR
9.           IF min != i
10.                 swap A[min] and A[i]
11.          END IF
12. END FOR
13. END SELECTION SORT
```

4. Quick Sort

QUICK SORT(A, p, r) // p and r are the lower and upper indices of the array to be sorted

```
1.  BEGIN
2.  IF p<r,
3.          q= partition(A, p, r)
4.          QUICKSORT(A, p, q-1)
5.          QUICKSORT(A, q+1, r)
6.  ENDIF
7.  END QUICKSORT
```

PARTITION(A, p, r) // p and r are the indices of the array to be partitioned

```
1.  BEGIN
2.  key= A[p]
3.  i= p
4.  j= r+1
```

```
5.  WHILE true
6.          do i= i+1, until A[i] > key
7.          do j= j-1, until A[j] <= key
8.          IF i < j
9.                  exchange A[i], A[j]
10.         ELSE
11.                 exchange A[j], key
12.         ENDIF
13.         return j
14. ENDWHILE
15. END PARTITION
```

5. Heap Sort

HEAP SORT(A, n)

```
1.  BEGIN
2.  BUILDHEAP(A)
3.  FOR i= n down to 2, do
4.          exchange A[i] and A[1]
5.          n= n-1
6.          HEAPIFY(A, 1)
7.  END FOR
8.  END HEAP SORT
```

BUILDHEAP(A) //Function to create the heap tree

```
1.  BEGIN
2.  FOR i= floor(n/2) down to 2, do
3.          HEAPIFY(A, i)
4.  END FOR
5.  END BUILDHEAP
```

HEAPIFY(A,i) //Function to perform heapify operation

```
1.  BEGIN
2.  l= 2*i
3.  r= 2*i + 1
4.  IF l <= n & A[l] > A[i]
5.          largest= l
6.  ELSE
7.          largest= i
8.  ENDIF
9.  IF r<= n & A[r]> A[largest]
10.         largest= r
11. ENDIF
12. IF largest != i
13.         exchange A[i] & A[largest]
```

```
14.         HEAPIFY(A, largest)
15. ENDIF
16. END HEAPIFY
```

6. Merge Sort

MERGE SORT(A,p,r) // p and r are the lower and upper indices of the array to be sorted

```
1.  BEGIN
2.  IF p<r,
3.          q= (p+r)/2
4.          MERGE SORT(A, p, q)
5.          MERGE SORT(A, q+1, r)
6.          MERGE(A, p, q, r)
7.  ENDIF
8.  END MERGE SORT
```

MERGE (A,p,q,r) // merges two arrays A[p...q] and A[q+1...r]

```
1.  j= p
2.  m= q+1
3.  WHILE j<= q && m<= r
4.          IF a[j]<= a[m]
5.                  c[k]= a[j++]
6.          ELSE
7.                  c[k]= a[m++]
8.          ENDIF
9.          k++
10. ENDWHILE
11. IF j > q,
12.         FOR  i= m to r, do
13.                 c[k]= a[i]
14.                 k++
15.         END FOR
16. ELSE
17.         FOR  i= j to q, do
18.                 c[k]= a[i]
19.                 k++
20.         END FOR
21. END IF
22. FOR  i= p to r, do
23.         a[i]= c[i]
24. END FOR

25. END MERGE
```

SAMPLE INPUT & OUTPUT (Same for all algorithms)

Input **: n = 8 A = 9 7 2 8 10 4 3**

Output **: A = 2 3 4 7 8 9 10**

Exp3 : String Operations

Problem Description :

A string is an array of characters terminated by a null character('\0'). C library has a rich collection of functions for handling strings. This experiment is intended to perform various string handling operations without any library functions.

ALGORITHMS

1. To find the length of the string

Input Specification : An input string, str

Output Specification : Length of the string ,n

LENGTH(str)

1. BEGIN
2. i=1
3. WHILE str[i] != '\0'
4. i = i + 1
5. END WHILE
6. n = i – 1
7. RETURN n
8. END LENGTH

SAMPLE INPUT

Input : Malayalam

Output : 9

2. To concatenate two strings

Input Specification : Two strings str1 and str2

Output Specification : A new string str1 which is obtained by concatenating str1 and str2

CONCATENATE(str1, str2)

1. BEGIN
2. i =0 , j=0
3. WHILE str1[i] != '\0'
4. i= i+1

5. END WHILE
6. WHILE str2[j] !='\0'
7. str1[i] = str2[j]
8. i = i+1
9. j = j +1
10. END WHILE
11. str1[i] = '\0'
12. RETURN str1
13. END CONCATENATE

SAMPLE INPUT & OUTPUT

Input str1 ="hello" str2 ="world"

Output str1 = "helloworld"

3. **Copy a string**

By string copy, we mean copying one string to another string character by character. The size of the destination string should be greater than equal to the size of the source string

Input Specification : Input string str1

Output Specification : A new string str2 which contains the content of str1

STRING COPY(str1)

1. BEGIN
2. i = 0
3. WHILE str1[i] != '\0'
4. str2[i] = str1[i]
5. i = i+1
6. END WHILE
7. str2[i] ='\0'
8. RETURN str2
9. END STRING COPY

SAMPLE INPUT & OUTPUT

Input : str1 = “daffodils”

Output : str2 = “daffodils”

4. **Compare two strings**

Input Specification : Two strings str1 and str2

Output Specification : Returns 0 if str1 is equal to str2, returns 1 if str1 is greater than str2 and returns -1 if str1 is less than str2

COMPARESTRING(str1, str2)

```
1.  BEGIN
2.  i = 0, j = 0
3.  WHILE str1[i] ! =’\0’ && str2[j] ! =’\0’
4.          IF str1[i] < str2[j]
5.                  RETURN -1
6.          ELSE IF str1[i] > str2[j]
7.                   RETURN 1
8.           ELSE  i= i+1, j= j+1
9.           END IF
10.  END WHILE
11. IF str1[i] !=’\0’
12.        RETURN 1
13. ELSE IF st2[j] !=’\0’
14.        RETURN -1
15. ELSE RETURN 0
16. END IF
17. END COMPARESTRING
```

SAMPLE INPUT & OUTPUT

Input : str1 = “daffodils” **,** str2 = “daffodils”

Output : 0

Input : str1 = “sand” str2 =”sandstone”

Output : -1

Input : str1 = “dryer” str2 =”dry”

Output : 1

Input : str1 =”java” str2 =”pascal”

Output : -1

5. **Substring search and replacement**

Input Specification **:** An input string str, a search string pattern and a replace string key

Output specification **:** A new string str1 obtained by replacing key in the place of the string pattern in the original string str**.**

SUBSTRREPLACE(str, pattern,key)

```
1.  BEGIN
2.  i = m = c = j = 0
3.  WHILE str[c] ! = '\0'
4.        IF  str[m] == pattern[i]
5.            i=i+1, m=m+1
6.            IF (pattern[i] == '\0')
7.                FOR  k = 0 TO  key[k] != '\0'
8.                    str1[j] = key[k]
9.                    i=0, c=m, k=k+1,j=j+1
10.               END FOR
11.           ELSE
12.              str1[j]  = str[c]
13.              j=j+1, c= c+1, m=c,i=0
14.       END IF
15.   END WHILE
16.   str1[j] = '\0'
17. END SUBSTRREPLACE
```

6. Count the number of vowels, consanants and words in a string

I***nput Specification*** **:** An input string str

Output specification **: Count of vowels,Vcount, Count of consonants, Ccount, Count of words, Wcount.**

Algorithm CountWords(str)

1. BEGIN
2. i = 0, Vcount =0, Ccount=0, Wcount=0
3. WHILE str[i] != '\0'
4. IF str[i] is an alphabet
5. IF str[i] IN {'a','e','i','o','u','A','E','I','O','U'}
6. Vcount = Vcount +1
7. ELSE
8. Ccount = Ccount + 1
9. END IF
10. ELSE IF str[i] IN {' ', '\n', '\t'}
12. Wcount = Wcount+1
13. END IF
14. END WHILE
15. END CountWords

Additional Exercises

1. Implement insertion and deletion operation in an array data structures
2. Implement reversal of each word in a string
3. Write a C program to sort an array of integers in ascending order using radix sort
4. Write a C program to sort a given list of strings

Viva Questions

1. What is the advantage of binary search method over linear search?

2. What are the drawbacks of binary search method?

3. compare various sorting methods by their performance.

4. How strings are implemented in C?

CYCLE2 : Stacks,Queues and Applications(using Arrays)

Exp1 : Stack Operations using Array

Problem Description : Stack is a linear data structure that performs operations in LIFO(last in first out)manner. 'top' is the variable that keeps track of the stack. The following three operations can be performed on stack :

(1) PUSH : This operation inserts an element into the stack. For inserting 'top' is incremented and then the element is inserted.

(2) POP : This operation deletes an element from the stack. For deleting 'top' is decremented.

(3) DISPLAY : This operation displays the content of the stack.

ALGORITHMS

1. Push

Input Specification : An array, stk of maximum size n , top value top and the item to be pushed

Output Specification : Modified stack stk with item added on the top.

PUSH(stk,top,item)

1. BEGIN
2. IF top== n - 1
3. print ' stack is full '
4. ELSE
5. top=top+1 //top incremented
6. stack[top]=item //inserting the element
7. END IF
8. END PUSH

2. **Pop**

Input Specification : An array, stk of size n , top value top

Output Specification : item, the topmost element of the stack.

POP(stk,top)

1. BEGIN
2. IF(top== -1)
3. print ‘ stack is empty ’
4. ELSE
5. top=top-1
6. END POP

3. Display(stk,top)

Input Specification : An array, stk of size *n* , top value *top*

Output Specification : All elements of the stack.

DISPLAY(stk,top)

1. BEGIN
2. IF top == -1
3. print ‘ stack is empty ’
4. ELSE
5. WHILE (top>=0)
6. print stack [top]
7. top=top-1
8. END WHILE
9. END IF
10. END DISPLAY

Sample Input &Output

Let the stack Stk be initially empty

Stk[] ; top = -1 ; n = 5

>>Push(Stk, -1,10)

Stk [10] ; top =0

>> Push (Stk, 0, 20)

Stk[10 20] top = 1

>> Push (Stk, 1, 30)

Stk[10 20 30] top = 2

>> Push (Stk, 2, 40)

Stk[10 20 30 40] top = 3

>> Push (Stk, 3, 50)

Stk[10 20 30 40 50] top = 4

>> Display(Stk, 4)

Stk[10 20 30 40 50] top = 4

Output : 10 20 30 40 50

>> Push(Stk, 4, 70)

Stk[10 20 30 40 50] top = 4

Output : "Stack Full"

>>Pop(Stk, 4)

Stk[10 20 30 40] top = 3

Output : 50

>>Pop(Stk, 3)

Stk[10 20 30] top = 2

Output : 40

>> Display(Stk, 2)

Stk[10 20 30] top =2

Output : 10 20 30

>>Push(Stk,2,90)

Stk[10 20 30 90] top = 3

>>Pop(Stk, 3)

Stk[10 20 30] top = 2

Output : 90

>>Pop(Stk, 2)

Stk[10 20] top = 1

Output : 30

>>Pop(Stk,1)

Stk[10] top = 0

Output : 20

>>Pop(Stk,0)

Stk[10] top = -1

Output : 10

>>Pop(Stk,-1)

Stk[ ] top = -1

Output : “Stack Empty”

>> Display(Stk, -1)

Stk[ ] top = -1

Output : “Stack Empty”

****

Exp2 : Queue Operations using Array

Problem Description : Queue is a linear data structure that performs operations in FIFO(first in first out)manner. 'front' and 'rear' are the variables that keeps track of the queue. The insertion operation in a queue is called an 'enqueue' operation and the deletion operation is called a dequeue operation. To enqueue, the condition of full queue is checked and if not satisfied ,the operation is performed at the rear . To dequeue, the condition of empty queue is checked and if not satisfied, the operation is performed at the front .

ALGORITHMS

1. Enqueue

Input Specification : An array, *Que* of size *n* , with front and rear values represented as *front* and *rear* respectively and the item to be inserted, *item*

Output Specification : Modified queue *Que* with *item* added on the *rear.*

Enqueue(Que,item,front,rear)

```
1.  BEGIN
2.  IF  rear == n
3.          print 'queue full'
4.  ELSE
5.          IF front== -1                       //If it is the 1st element
6.                  front=rear=0
7.          ELSE
8.                  rear=rear+1                 // entering the element at the rear
9.          END IF
10.         Que[rear]=item
11. END IF
12. END Enqueue
```

2. Dequeue

Input Specification : An array, Que of size n , front and rear values

Output Specification : Modified queue Que with item deleted from the front.

Dequeue(Que, front)

```
1.  BEGIN
2.  IF front== -1                               //If no element is present
3.          print'queue empty'
```

```
4.          return
5.  END IF
6.  IF front == rear AND front != -1
7.          item=Que[front]
8.          front=rear= -1
9.  ELSE                                    //deleting the element
10.         item=Que[front]
11.         front=front+1
12. END IF
13. END Dequeue
```

3. Display

Input Specification : An array, Que of size n , front and rear values

Output Specification : All elements of the queue

Display(Que,front,rear)

```
1.  BEGIN
2.  IF front== -1
3.          print'queue empty'
4.  ELSE
5.         i=front
6.         WHILE(i<=rear)
7.              print 'Que[i]'
8.              i=i+1
9.         END WHILE
10. END IF
11. END Display
```

Sample Input &Output

Let the queue Que be initially empty

Que[] ; front = -1 ; rear = -1 ; n = 4

>>Enqueue(Que, -1,-1, 10)

Que[10] ; front = 0 ; rear = 0

>>Enqueue(Que, 0, 0, 20)

Que[10 20] ; front = 0 ; rear = 1

>>Enqueue(Que, 0, 1, 30)

Que[10 20 30] ; front = 0 ;rear = 2

>>Enqueue(Que, 0, 2, 40)

Que[10 20 30 40] ; front = 0 ;rear = 3

>>Enqueue(Que, 0, 3, 50)

Que[10 20 30 40 50] ; front = 0 ;rear = 4

>>Display(Que,0,4)

Que[10 20 30 40 50] ; front = 0 ;rear = 4

Output : 10 20 30 40 50

>>Dequeue(Que, 0)

Que[20 30 40 50] ; front =1 ; rear = 4

Output : 10

>>Dequeue(Que, 1)

Que[30 40 50] ; front =2 ; rear = 4

Output : 20

>>Display(Que, 2, 4)

Que[30 40 50] ; front =2 ; rear = 4

Output : 30 40 50

>>Enqueue(Que, 2, 4, 90)

Que[30 40 50] ; front =2 ; rear = 4

Output : “Queue is full “

>>Dequeue(Que, 2)

Que[40 50] ; front =3 ; rear = 4

Output : 30

>>Dequeue(Que, 3)

Que[50] ; front =4 ; rear = 4

Output : 30

>>Dequeue(Que, 3)

Que[ ]; front = -1; rear = -1

Output : 50

>>Dequeue(Que, -1)

Que[ ]; front = -1; rear = -1

Output : “Queue is empty”

>>Display(Que, -1, -1)

Que[]; front = -1; rear = -1

Output : “Queue is empty”

>>Enqueue(Que, -1,-1, 80)

Que[80] ; front = 0 ; rear = 0

****

Exp3 : Implementation of Input Restricted Deque and Output Restricted Deque

Problem Description : A deque(double ended queue) is a data structure which permits insertion and deletion from both ends of the queue. There are two variants of deque

1. An input restricted deque is a deque, which allows insertion at only 1 end, rear end, but allows deletion at both ends, rear and front end of the lists

2. An output-restricted deque is a deque, which allows deletion at only one end, front end, but allows insertion at both ends, rear and front ends, of the lists.

The possible operations performed on deque are

1. Add an element at the rear end 2. Add an element at the front end 3. Delete an element from the front end 4. Delete an element from the rear end. Only 1st, 3rd and 4th operations are performed by input-restricted deque 1st ,2nd and 3rd operations are performed by output-restricted deque.

1. Insertion at the rear end

Input Specification : A deque , dq , with maximum size MAX, and item the element to be inserted

Output Specification : Modified deque

Algorithm InsertRearDQ(dq,item)

1. BEGIN

2. IF (rear==MAX) //Check for overflow

3. Print("Queue is Overflow")

4. Exit

5. ELSE

6. rear=rear+1;

7. dq[rear]=item;

/* Set rear and front pointer

8. IF rear==0 rear=1 END IF

9. IF front=0 front=1 END IF

10. END IF

11. END InsertRearDQ

2. Insertion at the front end

Input Specification : A deque , dq , with maximum size MAX, and item, the element to be inserted

Output Specification : Modified deque

Algorithm InsertFrontDQ(dq,item)

1. BEGIN
2. IF (front<=1) //Check for the front position
3. Print (“Cannot add item at front end”)
4. Exit
5. ELSE //Insert at front
6. front=front-1
7. Dq[front]=item
8. END IF
9. END InsertFrontDQ

3. Deletion from the front end

Input Specification : A deque , dq , with maximum size MAX

Output Specification : Modified deque and the deleted value, item

Algorithm DeleteFrontDQ(dq)

1. BEGIN
2. IF front== 0 // Check for front pointer
3. print(" Queue is Underflow”)
4. exit
5. ELSE
6. item =dq[front]
7. IF front== rear // Set front and rear pointer
8. front=0, rear=0
9. ELSE
10. front=front+1

```
11.     END IF
12.  END IF
13. END DeleteFrontDQ
```

4. Deletion from the rear end

Input Specification : A deque , dq , with maximum size MAX

Output Specification : Modified deque and the deleted value, item

```
Algorithm DeleteFrontDQ(dq)
1. BEGIN
2. IF rear == 0  //Check for the rear pointer
3.       print("Cannot delete value at rear end")
4.       Exit
5. ELSE
6.      item = dq[rear]  // Perform deletion
7.      IF front== rear  // Set front and rear pointer
8.            front=0 ,  rear=0
9.      ELSE
10.           rear=rear-1
11.    END IF
12.  END IF
13. END  DeleteFrontDQ
```

Exp3 : Circular Queue Operations using Array

Problem Description : As in ordinary queues, the insertion is performed at rear end and deletion is performed at front end. Circular Queues solves the disadvantage posed by ordinary queues.

ALGORITHMS

1. Enqueue

Input Specification : An array, *CQue* of size *n* , with front and rear values represented as *front* and *rear* respectively and the item to be inserted, *item*

Output Specification : Modified queue *CQue* with *item* added on the *rear.*

Algorithm **Enqueue(CQue,item,front,rear)**

```
1.  BEGIN
2.  IF front equals -1 and rear equals -1
3.          front = front+1, rear = rear+1
4.          CQue[rear] = item
5.  ELSE IF rear modulus n+1 equals front          //If it is the 1st element
6.          display "QUEUE FULL"
7.  ELSE
8.          rear = rear modulus n+1
9.          CQue[rear] = item
10. END IF
11. END Enqueue
```

2. Dequeue

Input Specification : An array, CQue of size n , front and rear values

Output Specification : Modified queue CQue with item deleted from the front.

Algorithm **Dequeue(CQue, front)**

```
1.  BEGIN
2.  IF front== -1                          //If no element is present
3.          print'queue empty'
4.          return
5.  END IF
6.  IF front == rear AND front != -1
7.          item=CQue[front]
8.          front=rear= -1
9.  ELSE                                   //deleting the element
10.         item=CQue[front]
11.         front=front+1
12. END IF
13. END Dequeue
```

3. Display

Input Specification : An array, *CQue* of size *n* , front and rear values

Output Specification : All elements of the queue

Algorithm **Display(Que,front,rear)**

```
1.  BEGIN
2.  IF front== -1
3.          print'queue empty'
```

```
4.  ELSE
5.        i=front
6.        WHILE(i<=rear)
7.             print 'CQue[i]'
8.             i=i+1
9.        END WHILE
10. END IF
11. END Display
```

Sample Input &Output

Let the queue Que be initially empty

CQue[] ; front = -1 ; rear = -1 ; n = 4

>>Enqueue(CQue, -1,-1, 10)

CQue[10] ; front = 0 ; rear = 0

>>Enqueue(CQue, 0, 0, 20)

CQue[10 20] ; front = 0 ; rear = 1

>>Enqueue(CQue, 0, 1, 30)

CQue[10 20 30] ; front = 0 ;rear = 2

>>Enqueue(CQue, 0, 2, 40)

CQue[10 20 30 40] ; front = 0 ;rear = 3

>>Enqueue(Que, 0, 3, 50)

Que[10 20 30 40 50] ; front = 0 ;rear = 4

>>Display(Que,0,4)

Que[10 20 30 40 50] ; front = 0 ;rear = 4

Output : 10 20 30 40 50

>>Dequeue(Que, 0)

Que[20 30 40 50] ; front =1 ; rear = 4

Output : 10

>>Dequeue(Que, 1)

Que[30 40 50] ; front =2 ; rear = 4

Output : 20

>>Display(Que, 2, 4)

Que[30 40 50] ; front =2 ; rear = 4

Output : 30 40 50

>>Enqueue(Que, 2, 4, 90)

Que[30 40 50] ; front =2 ; rear = 4

Output : “Queue is full “

>>Dequeue(Que, 2)

Que[40 50] ; front =3 ; rear = 4

Output : 30

>>Dequeue(Que, 3)

Que[50] ; front =4 ; rear = 4

Output : 30

>>Dequeue(Que, 3)

Que[ ]; front = -1; rear = -1

Output : 50

>>Dequeue(Que, -1)

Que[ ]; front = -1; rear = -1

Output : “Queue is empty”

>>Display(Que, -1, -1)

Que[]; front = -1; rear = -1

Output : “Queue is empty”

>>Enqueue(Que, -1,-1, 80)

Que[80] ; front = 0 ; rear = 0

****

Exp 5 : Infix to postfix conversion

Problem Description : The conversion is performed by reading each element in the expression and comparing the ISP(In Stack Priority)value of the stack top and ICP(Incoming Priority)value of the character read from the expression. A stack is used which holds the operators on the basis of its priority value. A symbol is pushed on to the stack IF its in-coming priority is greater than in-stack priority value of the top most element in the stack.Similarly, a symbol is popped from the stack IF its in-stack priority is greater than or equal to the in-coming priority value of the in-coming element.

Input Specification:

E : simple arithmetic expression in infix notation delimited at the end by '#',

ISP(X): Return the in-stack priority value for a symbol X.

ICP(X): This function returns the incoming priority value of a symbol X.

Output(X): Append the symbol X into the resultant expression.

Let us assume that a stack of capacity SIZE is known and TOP is the current pointer in it. PUSH and POP are usual operations of the stack.

Output Specification: An arithmetic expression in postfix notation

Data Structures: Array representation of a stack with TOP as the pointer to the top-most element.

Algorithm **InfixtoPostfix(E)**

1. BEGIN
2. TOP=0,PUSH('(') //initialize the stack
3. WHILE(TOP>0) do
4. item=E.Readsymbol() //scan the next symbol in infix expression
5. x=POP() //get the next item from stack
6. CASE: item= operand //IF the symbol is an operand

```
7.                  PUSH(x)              //the stack will remain same
8.                  Output(item)   //add the symbol into output expression
9.        CASE :item=')'                 //scan reaches to its END
10.                 WHILE x != '(' do       //till the left match is not found
11.                       Output(x)
12.                       x =POP()
13.                 END WHILE
14.       CASE :ISP(x)≥ICP(item)
15.                 WHILE(ISP(x)≥ICP(item)) do
16.                       Output(x)
17.                       x =POP()
18.                 END WHILE
19.                 PUSH(x)
20.                 PUSH(item)
21.       CASE: ISP(x)≤ICP(item)
22.                 PUSH(x)
23.                 PUSH(item)
24.         Otherwise
25.                  Print "Invalid expression"
26.         END CASE
27.  END WHILE
28.  END
```

Sample Input &Output

>>>> Infix: (A+ (B*C)/D)

Postfix: ABC*D/+

Exp 6 : Evaluation of prefix and postfix evaluation

Program Description : The prefix and postfix expressions can be evaluated without converting to its equivalent infix expression using stack data structure.

Postfix evaluation

Input Specification : An expression E in postfix notation

Output Specification : Value of the expression, e

Algorithm **PostFixEvaluation(E)**

1. Append a special delimiter '#' at the END of the expression

2. Item=O.Readsymbol()

3. WHILE(item≠'#') do

4. IF (item = operand) then

5. PUSH(item)

6. ELSE

7. op = item

8. y =POP()

9. x = POP()

10. t = x op y

11. PUSH(t)

12. ENDIF

13. item = E. ()

14. END WHILE

15. value = POP()

16. Return(value)

Prefix evaluation

The prefix expression can be reversed to obtain its postfix equivalent and then the above algorithm **PostFixEvaluation** can be used for the evaluation

Sample Input &Output

>>> 3 4 5*6/+

Output 6

>>> +/6*5 4 3

Output 6

Exp7: Multiple stacks in an array

Program Description : The array is divided in to the required number of stacks each having a top and bottom pointer.Insertion and deletion is made through the top END. Every stack can grow from boundary [stack_no]+1 to boundary[stack_no + 1]. During insertion if no free space is available in the required stack, a search is made for free space in any of the stack on the right side. If found the elements are shifted one position to the right side to make space for insertion and the pointers are updated. Else a similar search and updation is performed on the left side of the given stack.

Deletion is similar to usual stack deletion.

1. Initialization

Input Specification : the size of the array, m , and the number of stacks to be implemented n.

Output Specification : top and bottom values of n stacks,t[] and b[] initialized

Algorithm **StackInitialize(m,n)**

```
1. BEGIN
2. FOR  k = 1 TO  n
3.          b[k] = (m/n)*(k -1)
4.          t[k] = b[k]
5. END FOR
6. b[k+1] = m-1
7. END
```

2. Push

Input Specification : s[], multiple stack of max size m and divided into n stack, item , item to be inserted, i : to which stack to insert

Output Specification : Modified array, s

Algorithm **PushIntoStacki(s, item, i)**

```
1.  flag = 0 ,flag1 = 0
2.  IF t[i]<b[i+1]      /* Reqiured stack is not full  */
3.          t[i]  =  t[i+1]
4.                  s[t[i]]  =  item
```

```
5.                      flag = 1
6.   ELSE        /* Required stack is full */
7.        j =i+1
8.        WHILE j<= n                /* checking for space on the stacks on right side */
9.             IF t[j]<b[j+1]  /* shifting elements on space to right to make space for insertion */
10.                    FOR k = t[j] TO b[i+1]
11.                         s[k+1]  =  s[k]
12.                    END FOR
13.                    s[k+1]  = item
                       /* incrementing bottom and top pointers */
14.                    FOR k = i+1 TO    j
15.                         b[k]  =  b[k]+1
16.                         t[k] = t[k]+1
17.                    END FOR
18.                    t[i] = t[i]+1
19.                    flag1 =1 , flag = 1
20.                    break from WHILE loop
21.            END IF
22.       END WHILE
23.       IF flag = 0
24.            j = i-1
25.            WHILE j>0
                       /* checking for space on the stacks on left side */
26.                    IF t[j]<b[j+1]
            /* shifting elements on space to left to make space for insertion */
27                          FOR k = b[j+1]+1 TO   t[i]-1
28.                              s[k-1]  =  s[k]
29.                         END FOR
```

```
30.                         s[t[i]]  =  item
                            /* decrementing bottom and top pointers */
32.                         FOR k = j+1 TO  i-1
33.                             b[k]  =  b[k] - 1
34.                             t[k] =  t[k] – 1
35.                          END FOR
36.                         b[i]  =  b[i] -1
37.                         flag1 = 1
38.                         break from WHILE
39.                   END IF
40.                   j = j-1
41.              END WHILE
42.     END IF
43.  END IF
44.   END  PushIntoStacki
```

3. Pop

Input Specification : s[], multiple stack of max size m and divided into n stack, i : to which stack from the item is to be deleted.

Output Specification : Modified array, s

***Algorithm* PopFromStacki(s, i)**

```
1.   BEGIN
2.    IF t[i]  =  b[i]
3.          display “stack empty”
4.         exit
5.   END IF
6.    item  = s[t[i]]
7.    t[i] = t[i]-1
8.    END PopFromStack
```

3. Display

Input Specification : the array s containing multiple stacks

Output Specification : All elements of different stacks contained in the array.

Algorithm **DisplayAllStacks(s)**

```
1. BEGIN
2.    FOR i = 0 to n
3.        display stack     "i+1 "
4.        IF b[i] !=  t[i]
5.                  FOR j = b[i]+1 TO  t[i]
6.                            print s[j]
7.                  END FOR
8.        ELSE
9.                  print" empty"
10.       END IF
11.  END FOR
12.  END DisplayAllStacks(s)
```

Sample Input And Output

m = 20

n = 4

PUSH :

Enter the element to be inserted : 1

Enter the stack no : 3

Enter the element to be inserted : 5

Enter the stack no : 1

Enter the element to be inserted : 9

Enter the stack no : 1

Enter the element to be inserted : 15

Enter the stack no : 4

DISPLAY :

Stack 1 : 5 9

Stack 2: empty

Stack 3: 1

Stack 4: 15

POP :

Enter stack no : 1

The popped item is : 9

Enter stack no : 4

The popped item is : 15

DISPLAY :

Stack 1 : 5

Stack 2: empty

Stack 3: 1

Stack 4: empty

Additional Exercises

1. Write a C program to reverse the elements in the stack using recursion

2. Implement multiple queues in a single array.

3. Implement stack operations using queue data structure.

4. Implement queue operations using stack data structure.

5. Find an algorithm to convert infix expression to prefix

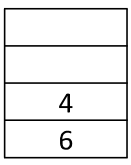
(Hint:first reverse the given expression,then convert to its postfix form and again reverse that expression).

6..How can we convert infix expression to prefix/postfix form without using stack data structure?(Hint: construct an expression tree and perform the required traversal of binary tree(preorder,postorder,inorder))

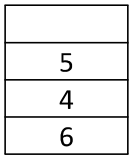
Viva Questions

1. If the elements "A", "B", "C" and "D" are placed in a queue and are deleted one at a time, in the middle is?
   A. ABCD B. DCBA C. DCAB D. ABDC
2. A data structure in which elements can be inserted or deleted at/from both the ends but not in the middle is?
   A. Queue B. Circular queue C. Deque D. Priority queue
3. A circular queue is implemented using an array of size 10. The array index starts with 0, front is 6, and rear is 9. The insertion of next element takes place at the array index.
   A. 8 B. 7 C. 0 D. 10
4. Convert the expression ((A+B)*C-(D-E))^(F+G) to equivalent prefix and postfix notations
5. Identify the wrong stack permutation for the input sequence 1, 2, 3, 4, 5.
   A. 5, 4, 3, 2, 1 B. 4, 3, 5, 2, 1 C. 2, 3, 5, 4, 1 D. 3, 4, 1, 5, 2
6. A circular queue of characters is implemented using a linear array[1,6]. The array currently contain array[D - - A B C]. if one character is deleted and two characters inserted, what will be the postions of rear and front pointers
   1. 1, 6 B. 3, 4 C. 3, 5 D. 5, 4
7. Consider the following arithmetic expression 2*3-(4+5) which of the stack configuration is not possible while evaluating the postfix equivalent expression of the above infix.

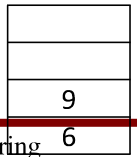
A.



B.



C.



D.

8. The initial configuration of circular queue is as follows

a	-	-	b	C

What is the status of Queue contents after the following sequence of steps

- Enqueue ‘x’
- Dequeue
- Enqueue ‘y’
- Dequeue
- Enqueue ‘z’
- Dequeue

A. x y z - - B. x – y – z- C. x - - y z D. – x y z –

CYCLE3 : Linked Lists and Applications

Exp 1 : Insertion and deletion operations in a single linked list

PROGRAM DESCRIPTION :

Single linked list contains nodes which includes one or more data fields and a pointer field. Data fields contains the data to be stored and the pointer field contains the address to the next node. Linked lists are resource efficient compared to arrays because the memory required for the storage of nodes are dynamically allocated.

1. Insertion at the beginning of a list

Input Specification : a HEAD pointer to the beginning of the list and val, the data to be inserted to the beginning of the list.

Output Specification : A modified list containing a new node having val as the data item, added to the beginning of the list.

Algorithm **InsertBeginningSL(HEAD, val)**

1. BEGIN
2. new=Getnode( ) //creates new node & returns pointer to that memory
3. new→DATA= val
4. new→LINK = HEAD
5. HEAD = new //newly created node made as head
6. END InsertBeginningSL

2. Insertion at the end of the list

Input Specification : a HEAD pointer to the beginning of the list and val, the data to be inserted to the end of the list.

Output Specification : A modified list containing a new node having val as the data item, added to the end of the list.

Algorithm **InsertEndSL(HEAD, val)**

1. BEGIN

2. new = Getnode( )

3. ptr = HEAD

4. WHILE (ptr→LINK != NULL)

5. ptr = ptr→LINK

6. END WHILE

7. ptr→LINK=new //last node points to the newly created node

8. new→DATA = val

9. new→LINK = NULL

10. END InsertEndSL

3. Insertion at any specified position

Input Specification : a HEAD pointer to the beginning of the list and val, the data to be inserted after the node containing value 'key'

Output Specification : A modified list containing a new node having val as the data item, added in the specified position in the list.

Algorithm **InsertAnyPosSL(HEAD, val, key)**

1. BEGIN

2. new = Getnode( )

3. ptr=HEAD

4. WHILE (ptr→DATA != key) //finds the position for the element to be inserted

5. ptr = ptr→LINK

6. END WHILE

7. new→LINK = ptr→LINK

8. ptr→LINK = new

9. new→DATA = val

10. END InsertAnyPosSL

4. Deletion from the beginning of a list

Input Specification : a HEAD pointer to the beginning of the list

Output Specification : A modified list obtained by deleting the first element of the list

Algorithm **DeleteBeginningSL(HEAD)**

1. BEGIN

2. ptr = HEAD

3. HEAD = HEAD→LINK //second node made as head

4.. free (ptr)

5. END Delete**BeginningSL**

5. Deletion from the end of a list

Input Specification : a HEAD pointer to the beginning of the list

Output Specification : **A modified list obtained by deleting the last element of the list**

Algorithm **DeleteEndSL(HEAD)**

1. BEGIN

2. ptr = HEAD

3. WHILE (ptr→LINK !=NULL)

4. temp = ptr

5. ptr = ptr→LINK

6. END WHILE

7. free (ptr) //last node deleted

8. temp→LINK = NULL

9. END DeleteEnd**SL**

6. Deletion from any specified position

Input Specification : a HEAD pointer to the beginning of the list and val, 'key' the data in the node which is to be deleted.

Output Specification : A modified list obtained after deleting a node from the specified position

***Algorithm Deletefrom*AnyPosSL(HEAD, key)**

1. BEGIN

2. ptr = HEAD

3. WHILE (ptr→DATA != key) //finds the node to be deleted

4. temp = ptr

5. ptr = ptr→LINK

6. END WHILE

7. temp→LINK = ptr→LINK

8. free (ptr)

9. END DeletefromAnyPosSL

7. Traversing the list

Input Specification : a HEAD pointer to the beginning of the list

Output Specification : All elements in the list

Algorithm **TraverseSL(HEAD)**

```
1. BEGIN
2. IF (HEAD== NULL)
3.       Print 'list is empty'
4. ELSE
5.       ptr=HEAD
6.       WHILE (ptr !=NULL)              //loop displays elements in the list
7.              print ptr→DATA
8.              ptr=ptr→LINK
9.        END WHILE
10. END IF
11. END  TraverseSL
```

Exp2 : Implement Stack data structure using Linked List

Program Description : The linear data structure Stack , which operates based on LIFO policy is implemented using linked structure.

1. Push

Input Specification : A linked list representation of Stack, pointed by TOP,and item , the value to be pushed.

Output Specification : Stack with a new value pushed.

Algorithm **PushSL** is similar to the **InsertEndSL** in Exp1.

2. Pop

Input Specification : A linked list representation of Stack, pointed by TOP

Output Specification : Stack with the topmost element popped

Algorithm **PopSL** is similar to the **DeleteFromBeginningSL** in Exp1.

3. Display

Input Specification : A linked list representation of Stack, pointed by TOP

Output Specification : All stack elements

Algorithm **DisplaySL** is similar to the **TraverseSL** in Exp1.

****

Exp3 : Implement polynomial addition and multiplication using Linked List

Program Description : The linked list node structure in representing a polynomial is as shown below :

Polynomial addition:

In order to add two polynomial we have to compare we have to compare their terms starting at the first node and moving towards the END one by one.There may be three cases during the comparision:

Case 1: Exponents of two terms are equal,then the coefficients of two terms are added

Case 2: Exponent of the current term of pointer 1 > exponent of pointer 2.Then a duplicate of current term of pointer 1 is added to the resultant polynomial

Case 3: Exponent of the current term of pointer 1 < exponent of pointer 2.Then a duplicate of current term of pointer 2 is added to the resultant polynomial

Polynomial Multiplication:

During polynomial multiplication we multiply each term in polynomial 1 with every term of second polynomial.Each product node is inserted in proper position in the resultant polynomial.If a node with same exponent is already in the resultant polynomial the coefficients of two nodes are added together else it is inserted to the correct position depENDing on the exponent value ie the position where the exp of product term is greater than exp of a node in the resultant polynomial.If the exp is the least it is appENDed at the END.

1. Polynomial Addition

Input Specification : Two polynomials in linked list representation whose header pointers are *Phead* and Qhead , to be added

Output Specification : The sum polynomial in linked list representation whose header pointer is *Rhead*

Data Structure: Single linked list for representing a term in a single variable polynomial

Algorithm PolyAddSL(Phead, Qhead)

1. BEGIN
2. Pptr = Phead , Qptr = Qhead

```
3.  Rptr=Rhead = NULL
4.  WHILE (Pptr ≠ NULL)   and   (Qptr ≠ NULL)   do
5.              CASE  Pptr → exp ==  Qptr → exp              //exponents are equal
6.                      new = GetNode()
                        /* Coefficients are added */
7.                      new→coeff = Pptr→coeff + Qptr→coeff
8.                      new→exp = Pptr→exp
9.                      new→link = NULL
10.                     Pptr = Pptr→link , Qptr→link
11.             CASE  Pptr→exp > Qptr→exp :
12.                     new = GetNode()
13.                     new→coeff = Pptr→coeff
14.                     new→exp = Pptr→exp
15.                     new→link = NULL
16.                     Pptr = Pptr→link
17.             CASE : Pptr→exp < Qptr→exp
18.                     new = GetNode()
19.                     new→coeff = Qptr→coeff
20.                     new→exp = Qptr→exp
21.                     new→link = NULL
22.                     Qptr = Qptr→link
23.             END CASE
24.             IF Rptr == NULL THEN
25.                     Rptr = Rhead = new
26.             ELSE  Rptr→link = new
27.                     Rptr = new
28.             END IF
29.     END WHILE
```

```
30.  IF Pptr≠NULL and Qptr=NULL
31.       WHILE Pptr≠NULL   do,        //inserting remaining terms of Pptr to Rptr
32.            new = GetNode()
33.            new→coeff = Pptr→coeff
34.            new→exp = Pptr→exp
35.            new→link = NULL
36.            Pptr = Pptr→link
37.            IF Rptr == NULL THEN
38.                 Rptr = Rhead = new
39.             ELSE  Rptr→link = new
40.                 Rptr = new
41.            END IF
42.       END WHILE
43.  END IF
44.  IF Pptr=NULL and Qptr≠NULL
45.       WHILE Qptr≠NULL   do,     //inserting remaining terms of Qptr to Rptr
46.            new = GetNode()
47.            new→coeff = Qptr→coeff
48.            new→exp = Qptr→exp
49.            new→link = NULL
50.            Qptr = Qptr→link

51.            IF Rptr == NULL THEN
52.                 Rptr = Rhead = new
53.             ELSE  Rptr→link = new
54.                 Rptr = new
55.            END IF
56.       END WHILE
```

57. END IF

58. return Rhead

59. END PolyAddSL

2. Polynomial Multiplication

Input Specification : Two polynomials in linked list representation whose header pointers are Phead and Qhead , to be multiplied

Output Specification : The product polynomial in linked list representation whose header pointer is Rhead

Data Structure: Single linked list for representing a term in a single variable polynomial

Algorithm PolyMultiplySL(Phead, Qhead)

1. BEGIN
2. Pptr = Phead , Qptr = Qhead
3. IF (Pptr→link = NULL) or (Qptr→lnk = NULL)
4. exit //no valid operation possible
5. ENDIF
6. Rptr = Rhead = NULL
7. WHILE(Pptr≠NULL) //for each term of P
8. WHILE(Qptr≠NULL)
9. c=Pptr→coeff * Qptr→coeff
10. x=Pptr→exp + Qptr→exp

    /* Search for the equal exponent value in R */
11. Rptr = Rhead
12. IF (Rptr == NULL)
13. new = GetNode();
14. new→exp = x , new→coeff = c, new→link = NULL
15. Rhead = new
16. ELSE
17. WHILE (Rptr≠NULL) and (Rptr→exp>x)

```
18.                                 Rptr1 = Rptr
19.                                 Rptr = Rptr→link
20.                           END WHILE
21.                           IF (Rptr == NULL) Or  (Rptr→exp < x)
22.                                 new = GetNode()
23.                                 new→exp = x , new→coeff = c
24.                                 Rptr1→ link = new, new→link = Rptr
25.                           ELSE IF (Rptr→exp = x) then //node with same coefficient
26.                                 Rptr→coeff = Rptr→coeff+c
27.                           END IF
28.                     END IF
29.                     Qptr = Qptr→link
30.               END WHILE
31.               Pptr = Pptr→link
32.       END WHILE
33.       return Rhead
33.       END  PolyMultiplySL
```

Additional Exercises

1. Implementation of circular linked list – Insertion, deletion, splitting of a circular linked list list into two circular lists, concatenation of two circular lists to form a single list.

2. Using linked list representation, implement a Deque.

Viva Questions

1. Linked lists are best suited ….
   A. for relatively permanent collections of data.
   B. for the size of the structure and the data in structure are constantly changing.
   C. data structure.
   D. for none of the above situation.
2. In linear data structure data elements are adjacent to each other
   A. True
   B. False
3. The situation when in a linked list START=NULL is …
   B. Underflow B. Overflow C. Houseful D. None of the mentioned
4. What is the time complexity of searching for an element in a circular linked list?

   A. O(n) B. O(nlogn) C. O(1) D. None of the mentioned

5. What are the time complexities of finding 8^{th} element from beginning and 8^{th} element from end in a singly linked list?
Let n be the number of nodes in linked list, you may assume that n>8.

   A. O(1) and O(n) B. O(1) and O(1) C. O(n) and O(1) D. O(n) and O(n)

CYCLE 4 :Trees, Graphs and Hashing

Exp1 : Binary Search Tree Operations using Linked List

PROGRAM DESCRIPTION:

A binary search tree is a binary tree which satisfies the following BST propery:

All elements stored in the left subtree of any node 'X' are less than the element stored at 'X' and All elements stored in the right subtree of 'X' are greater than the element at 'X'

Inorder traversal: elements are visited in the fashion ' left child→root →right child '

BST Insertion:

To insert an element we first search for the element.During search if the element >current data we go to the right subtree else to the left subtree and search is continued .If the element is already present in tree a 'duplicate insertion' error is raised .If element is not present then our search procedure ENDs in a null link.This is the correct position where the element is to be inserted.

BST Deletion:

There are three possible cases in deletion:

Case 1 : deleting a leaf node(NODE with no children) –the required node is simply deleted from the tree and the link to node from the parent node is made null

Case 2 : deleting a node with one child – the node is deleted and the single child(with its subtree) is linked to the parent of the removed tree.

Case 3 : deleting a node with two child – to delete we first find its inorder successor(left most node in the right subtree of the node to be deleted) node.

The node to be deleted with inorder successor and then delete inorder successor.

1. Searching on a BST

Input Specification : A BST pointed by Root, and the item to be searched, key

Output Specification : If the search is successful, a pointer to the node containing the searched value is returned else NULL is returned

Algorithm SearchBSTLL(Root, key)

1.BEGIN

2. ptr =ROOT,flag = FALSE //start from the root node

2. WHILE (ptr! = NULL) and (flag = FALSE) do

3. CASE: ITEM<ptr→DATA //go to the left subtree

4. ptr1 =ptr

5. ptr =ptr→LCHILD

6. CASE: ITEM>ptr→DATA //go to the right subtree

7. ptr1 = ptr

8. ptr = ptr→RCHILD

9. CASE: ptr→DATA = ITEM //NODE exits

10. flag = TRUE

11. Print "Search Successfull"

12. return ptr //Quit execution

13. END CASE

14. END WHILE

2. Inorder Traversal

Input Specification : A BST pointed by Root, and a pointer ptr to the root node

Output Specification : Listing of all elements in the BST

Algorithm InorderLL(ptr)

1. BEGIN

2. IF ptr!=NULL

3. InorderLL(ptr→LCHILD)

4. print 'ptr→DATA'

5. InorderLL(ptr→RCHILD)

6. END IF

7. END InorderLL

3. Preorder Traversal

Input Specification : A BST pointed by Root, and a pointer ptr to the root node

Output Specification : Listing of all elements in the BST

Algorithm PreorderLL(ptr)

1. BEGIN

2. IF ptr!=NULL

3. print 'ptr→DATA'

4. PreorderLL(ptr→LCHILD)

5. PreorderLL(ptr→RCHILD)

6. END IF

7. END PreorderLL

4. Postorder Traversal

Input Specification : A BST pointed by Root, and a pointer ptr to the root node

Output Specification : Listing of all elements in the BST

Algorithm PostorderLL(ptr)

1. BEGIN

2. IF ptr!=NULL

3. PostorderLL(ptr→LCHILD)

4. PostorderLL(ptr→RCHILD)

5. print 'ptr→DATA'

6. END IF

7. END PostorderLL

5. Insertion in a BST

Input Specification : A BST pointed by Root, and the item to be inserted, key

Output Specification : Modified BST

Algorithm InsertBSTLL(Root, key)

1. BEGIN

2. IF Root == NULL

3. new = GetNode()

4. new→DATA = ITEM //avail a node and then initialize it

```
5.      new→LCHILD = NULL
6.      new→RCHILD = NULL
7. ELSE
8.      ptr = SearchBSTLL(Root, key)
9.      IF ptr ≠NULL
10.             print 'Insertion Not Possible – Data already exists"
11.             Return
12.     ELSE
13.             new = GetNode()
14.             new→DATA = ITEM //avail a node and then initialize it
15.             new→LCHILD = NULL,  new→RCHILD = NULL
16.            IF(ptr→DATA<key) then              //insert as the right child
17.                     ptr→RCHILD = new
18.             ELSE
19.                     ptr→LCHILD = new          //insert as the left child
20.             ENDIF
21.   END IF
22. END  InsertBSTLL
```

6. Deletion from a BST

Input Specification : A BST pointed by Root, and the item to be deleted, key

Output Specification : Modified BST

```
Algorithm DeleteBSTLL(Root, key)
1.  BEGIN
2.  ptr = SearchBSTLL(Root, key)
3. IF ptr == NULL
4.       Print "ITEM does not exist. No deletion"
5.       Exit
6.  END IF
7.  IF(ptr->LCHILD =NULL)and(ptr->RCHILD = NULL) then     //node has no child
8.       CASE =1
9.  ELSE IF(ptr->LCHILD!= NULL)and(ptr->RCHILD!= NULL) then       //node contain
both left and right child
10.              CASE =3
11.      ELSE                                  //node contains only one child
12.              CASE =2
13.      ENDIF
         /*DELETION CASE 1 */

14 . IF (CASE = 1) then
15.      IF(parent->LCHILD =ptr)then       //IF the node is a left child
16.              parent->LCHILD = NULL     //set the pointer of its parent
17.      ELSE
18.              parent->RCHILD = NULL
19.      ENDIF
```

```
20.    ReturNode(ptr)              //return the deleted node to the memory bank
21.  ENDIF
            /*DELETION: CASE 2 */
22.  IF(CASE = 2) then              //when the node contain only one child
23.       IF(parent->LCHILD = ptr) then  //IF the node is a left child
24.            IF (parent->LCHILD = NULL) then
25.                 parent->LCHILD = ptr->RCHILD
26.            ELSE
27.                 parent->LCHILD = ptr->LCHILD
28.            END IF
29.       ELSE
30.            IF(parent->RCHILD = ptr) then
31.                 IF(ptr->LCHILD = NULL) then
32.                      parent->RCHILD = ptr->RCHILD
33.                 ELSE
34.                      parent->RCHILD = ptr->LCHILD
35.                 END IF
36.            END IF
37.       END IF
38.  ReturnNode(ptr)                    //return the deleted node to the
memory bank
39.  ENDIF
     /*DELETION CASE 3 */
```

```
40. IF(CASE = 3)                    //when the node contain both left and right child
41.         ptr1 = InorderSuccSL(ptr)       //find the inorder successor of the node
42.         item1 = ptr->DATA
43.         DeleteBSTLL(ptr1, item1)        //delete the inorder successor
44.         ptr->DATA = item1       /replace the data with the data of the inorder  st
                                                                    successor
45. END IF
46. END DeleteBSTLL
```

Algorithm InorderSuccSL(ptr)

Input Specification : Pointer to the node ptr whose inorder successor is to be found

Output Specification : Pointer to the inorder successor of ptr

```
1. BEGIN
2.   ptr1 = ptr->RCHILD                          //move to the right subtree
3.   IF (ptr1! = NULL) then          //IF the right subtree is not empty
4.      WHILE (ptr->LCHILD! = NULL) do              //move to the left most END
5.              ptr1 = ptr->LCHILD
6.      END WHILE
7.   END IF
8.   Return(ptr1)                    //return the pointer of the inorder successor
9.   END InorderSuccSL
```

Exp2 : Graph Traversals using Linked List

Problem Description :A Graph is a non linear data structure represented as G = (V,E) where V is the set of vertices and E is the set of edges. The graph data strucuture can be traversed in two ways: Depth First Search(DFS) and Breadth First Search(BFS).

1. Depth First Search

Input Specification: A Graph represented using linked list pointed by Gptr, v, the starting vertex

Output Specification: List of all vertices in the graph in an array VISIT

Data Structures used : A stack named OPEN

Algorithm **DFSLL(Gptr, v)**

```
1. START
2. IF Gptr == NULL
3.      Print " Graph is empty"
4.      Exit
5. END IF
6.  u  = v                                  //Start from v
7.  OPEN.push(u)                    //Push the starting vertex into OPEN
8.  WHILE (OPEN.TOP != NULL)
9.          u = OPEN.POP()
10.         IF(Search_SL(VISIT,u) = FALSE) then
11.                 InsertEnd_SL(VISIT,u)
```

```
12.              ptr = GPTR[u]
13.              WHILE(ptr-->LINK !=NULL)
14.                        vptr = ptr->LINK
15.                       OPEN.PUSH(vptr->LABEL)
16.              END WHILE
17.       END IF
18.   END WHILE
19.   RETURN  VISIT
20    END  DFSLL
```

2. Breadth First Search

Input Specification: A Graph represented using linked list pointed by Gptr, v, the starting vertex

Output Specification: List of all vertices in the graph in an array VISIT

Data Structures used : A queue named OPENQ

```
 Algorithm BFSLL(Gptr, v)
1. START
2. IF Gptr == NULL
3.       Print " Graph is empty"
4.       Exit
5. END IF
6.  u  = v                                  //Start from v
7.  OPENQ.enqueue(u)                        //Enter the starting vertex into OPENQ
```

```
8.  WHILE (OPENQ.STATUS != EMPTY)
9.          u = OPENQ.dequeue()
10.          IF(Search_SL(VISIT,u) = FALSE) then
11.                 InsertEnd_SL(VISIT,u)
12.                 ptr = GPTR[u]
13.                WHILE(ptr-->LINK !=NULL)
14.                          vptr = ptr->LINK
15.                        OPENQ.enqueue (vptr->LABEL)
16.                END WHILE
17.       END IF
18.   END WHILE
19.   RETURN  VISIT
20    END BFSLL
```

Additional Exercises

1. 1. Write a program to delete a tree.

hint : use postorder traversal so that children will be deleted before deleting parent node

2. Write a program to find the mirror image of a tree. Mirror image of a binary tree is another binary tree with left and right child of all its non-leaf nodes interchanged.

3. Tree operations

1. Iterative versions of inorder, preorder and post order

2. Create a copy of a binary tree

3. Check whether two trees are similar

4. Display the elements in level order,i.e, list the elements in the root, followed by elements in level 2, then the elements in level 3 and so on. (Hint : Use stack)

4. Graph operations

1. Implementation using arrays and linked representation

2. Calculating the indegree and outdegree of each vertex

Viva Questions

1. The maximum number of binary trees that can be formed with three nodes is:
   A. 1 B. 3 C. 5 D. 4
2. A binary tree T has 20 leaves. The number of nodes in T having two children is
   A. 18 B. 19 C. 21 D. 20
3. Draw the expression tree for 2* 3/(2-1)+5*(4-1)
4. Consider the node of a complete binary tree whose value is stored in DATA[i] for an array implementation. If a node has a right child where will be the right child value stored
   A. DATA[i/2] B. DATA[i+2] C. DATA[2*i+1] D.DATA[2*i+2]
5. Suppose T is a binary tree with 14 nodes. What is the minimum possible depth of T.

A. 0 B. 4 C. 5 D. 3

6. Suppose the numbers 7, 5, 1, 8, 3, 6, 0, 9, 4, 2 are inserted in that order into an initially empty binary search tree. The binary search tree uses the usual ordering on natural numbers.
   a)Draw the binary Search Tree
   b)What is the in-order traversal sequence of the resultant tree?
   c) What is its Height

7. Draw the binary search tree and find the number of nodes of left and right subtree of the binary tree if the data is inserted in the following order: 45, 15, 8, 5 6, 5, 65, 47, 12, 18, 10, 73, 50, 16, 61