

DEPARTMENT OF COMPUTER ENGINEERING

LAB MANUAL



DATA STRUCTURES (CSL301)

S.E. (COMPUTER ENGINEERING)
SEMESTER-III
(R-2019)-Ver1

COMPUTER ENGINEERING DEPARTMENT

DEPARTMENT'S VISION

To be a centre of Excellence in Computer Engineering to fulfill the rapidly growing needs of the Society.

DEPARTMENT'S MISSION

M1: To Impart quality education to meet the professional challenges in the area of Computer Engineering.

M2: To create an environment for research, innovation, professional and social development.

M3: To nurture lifelong learning skills for achieving professional growth.

M4 To strengthen the alumni and industrial interaction for overall development of students.

PROGRAMME EDUCATIONAL OBJECTIVES(PEOs)

PEO1: Practice Computer engineering in core and multi-disciplinary domains.

PEO2: Exhibit leadership skills for professional growth.

PEO3: Pursue higher Studies for career advancement

PROGRAMME OUTCOMES (POs)

PO1: Engineering knowledge: Apply the knowledge of mathematics science engineering fundamentals and an mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems 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 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 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, 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 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: long learning: Recognize the need for and have the preparation and ability to engage in independent and lifelong learning in the broadest context of technological change.

PROGRAMME SPECIFIC OUTCOMES (PSOs)

PSO1: To apply computational and logical skills to solve Computer engineering problems.

PSO2: To develop interdisciplinary skills and acquaint with cutting edge technologies in software industries.

GENERAL INSTRUCTIONS (Do's And Don'ts)

1. Wearing ID-Card is compulsory.
2. Keep your bag at the specified place.
3. Shut down the system after use.
4. Place the chairs in proper position before leaving the laboratory.
5. Report failure/Non-working of equipment to Faculty In-charge / Technical Support staff immediately.
6. Know the location of the fire extinguisher and the FIRST-AID Box and how to use then in case of an emergency.
7. Do not eat or drink in the laboratory.
8. Do not litter in the laboratory.
9. Avoid stepping on electrical wires or any other computer cables.
10. Do not open the system unit casing or monitor casing particularly when the power is turned ON.
11. Do not insert metal objects such as clips, pins and needles into the computer casing. They may cause fire.
12. Do not remove anything from laboratory without permissions.
13. Do not touch, connect or disconnect any plug or cable without permission.

COURSE OUTCOMES (CO)

CO1: Students will be able to implement various linear and nonlinear data structures.

CO2: Students will be able to handle operations like insertion, deletion, searching and traversing on various data structures.

LAB ARTICULATION MATRIX (MAPPING WITH PO & PSO)

	Course Outcome	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2
CSL 303.1	Students will be able to implement various linear and nonlinear data structures.	3	3	3	2	2			3	3	3			2	
CSL 303.2	Students will be able to handle operations like insertion, deletion, searching and traversing on various data structures.	3	3	3	2	2			3	3	3			2	
Average		1.25	3	3	3	3	2	2		3	3	3	2	3	2.333333

Laboratory Assessment

Academic Year: _____

Class/Sem: Div: _____

Batch: _____

Student Name: _____

Roll No: _____

Course Name: _____

Course Code: _____

1. Preparedness and Efforts/ Preparation and Knowledge

3: Well prepared and puts efforts.

2: Not prepared but puts efforts or prepared but doesn't put efforts

1: Neither prepared nor puts efforts

2. Presentation of output/ Accuracy and Neatness of Documentation

3: Uses all perfect Instructions / Interrupts/Presented well.

2: Uses some perfect Instructions / Interrupts/moderate presentation.

1: Doesn't use any of the proper Instruction / Interrupt/ Not presented properly.

3. Results/ Participation in Practical Performance

3: Participate and gets proper results

2: Participate but doesn't get proper result or gets result but with the help of faculty in-charge

1: Neither Participate nor gets the results

4. Punctuality

3: Get the experiment checked in-time and is always in-time to the lab sessions

2: Some time delays the experiment checking or is late to the lab sessions for few times

1: Most of the time delays experiment checking and / or comes late for lab sessions

5. Lab Ethics

3: Follows proper lab ethics by keeping the lab clean and placing things at their right place

2: Sometimes doesn't follow the lab ethics

1: Most of the times makes the lab untidy and keeps things at wrong place

EVALUATION:

Performance Indicator/ Expt. No	1	2	3	4	5	6	7	8	9	10	Total
	CO1, 2	CO1, 2	CO1 ,2	CO1 ,2	CO1 ,2	CO1, 2	CO1, 2	CO1, 2	CO1, 2	CO1 ,2	
1. Preparedness and Efforts/ Preparation and Knowledge											
2. Presentation of output/ Accuracy and Neatness of Documentation											
3. Debugging and results/ Participation in Practical Performance											
4. Punctuality											
5. Lab Ethics											
Total											
Average											

Exceed Expectations (3), Meet Expectations (2). Below Expectations (1)

Faculty In-charge

EXPERIMENT LIST

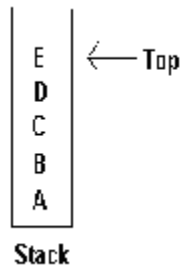
1. Implement Stack ADT using array.
2. Convert an Infix expression to Postfix expression using stack ADT.
3. Evaluate Postfix Expression using Stack ADT.
4. Implement Linear Queue ADT using array.
5. Implement Circular Queue ADT using array.
6. Implement Singly Linked List ADT.
7. Implement Circular Linked List ADT.
8. Implement Stack / Linear Queue ADT using Linked List.
9. Implement Binary Search Tree ADT using Linked List.
10. Implement Graph Traversal techniques: a) Depth First Search b) Breadth First Search

Experiment No. 1

Aim: Implementation of stack using arrays.

Theory:

A *stack* is an ordered list in which all insertions and deletions are made at one end, called the *top*. Just like a plate stack in canteens where plates are added to the top and later removed from the top.



The restrictions on a stack imply that if the elements A,B,C,D,E are added to the stack, in that order, then the first element to be removed/deleted must be E. Equivalently we say that the last element to be inserted into the stack will be the first to be removed. For this reason stacks are sometimes referred to as Last In First Out (LIFO) lists.

The main function that has to be included in stack are:

1. `isfull()`: This is to check whether the stack is full or not. Returns true if stack is full.
2. `isempty()`: This is to check whether the stack is empty or not. Returns true if stack is empty.
3. `push()`: It is used to enter element at the top of the stack.
4. `pop()`: It is used to remove first element from the top of stack.
5. `stacktop()`: This function is used to return the value at the stack top.

After implementation of the program, apply the following test cases, set the size of the stack to 5.

-

1. Insert 6 elements and show five elements (1,2,3,4,5,6) are inserted successfully and for 6th element, you get an overflow message.
2. Pop 6 times and check that elements are displayed in the order 5,4,3,2,1, underflow.

Conclusion: Stack is a linear data structure thus it can be easily implemented using arrays. All the operations- push(), pop() and test cases- overflow and underflow were tested successfully.

Upload your program (rollno_expt1) with output for the test cases [here](#).



Experiment No. 2

Aim: Convert an Infix expression to Postfix expression using stack ADT.

Theory: Stacks are widely used in the design and implementation of compilers. For example, they are used to convert arithmetic expressions from infix notation to postfix notation. An infix expression is one in which operators are located between their operands. This is how we are accustomed to writing expressions in standard mathematical notation. In postfix notation, the operator immediately follows its operands.

Examples of infix expressions are:

$a * b$
 $f * g - b$

The corresponding postfix expressions are:

$a b *$
 $f g * b -$

In a postfix expression, a binary operator is applied to the two immediately preceding operands. A unary operator is applied to the one immediately preceding operand. A useful feature of postfix is that the order in which operators are applied is always unambiguous.

Because of the existence of this unique interpretation, some compilers first convert arithmetic expressions from the standard infix notation to postfix to simplify code generation. To convert an infix expression to postfix, we must use a stack to hold operators that cannot be processed until their operands are available.

Algorithm

Step 1: Take the infix expression as input from the user.

Step 2: Repeat until each character in the infix notation is scanned

IF (is encountered, push it on the stack

IF an operand (whether a digit or a character) is encountered, add it postfix expression.

IF) is encountered, then

a. Repeatedly pop from stack and add it to the postfix expression until a (is encountered.

b. Discard the (. That is, remove the (from stack and do not add it to the postfix expression

IF an operator is encountered, then

a. Repeatedly pop from stack and add each operator (popped from the stack) to the postfix expression which has the same precedence or a higher precedence than

b. Push the operator to the stack

[END OF IF]

Step 3: Repeatedly pop from the stack and add it to the postfix expression until the stack is empty

Step 4: EXIT

Working of the algorithm to convert infix to postfix expression:

Assume that we are given as input the infix string $a + b * c - d$. This is equivalent to the parenthesized infix arithmetic expression $(a + (b * c)) - d$. To convert this to postfix format, we scan the input from left to right. If the next symbol in the input stream is an operand (for example, a) it is placed directly into the output stream without further processing.

Input: $a + b * c - d$ Output: a opStack: empty ↑

If the next symbol in the input string is an operator (for example, +), then we compare the precedence of this operator to the one on top of a stack called opStack, short for operator stack, which is initialized to empty at the start of the scan. If opStack is empty or if the precedence of the next operator in the input stream is greater than the precedence of the one on top of opStack, then we push the operator from the input string onto opStack and continue.

Input: $a + b * c - d$ Output: a opStack: + ↑

Let's explain what is happening a little more intuitively. We are trying to locate the highest-precedence operation to determine which operation to perform next. As long as the operators being scanned are increasing in precedence, we cannot do anything with our infix expression. Instead, we stack them and continue scanning the input string.

Input: $a + b * c - d$ Output: a b opStack: +

↑
Input: $a + b * c - d$ Output: a b opStack: + *

↑
Input: $a + b * c - d$ Output: a b c opStack: + *

↑
Input: $a + b * c - d$ Output: a b c opStack: + *

↑

At this point, we encounter the $-$ operator, whose precedence is lower than the $*$, the operator on top of the stack. We now pop operators from opStack until either (1) opStack is empty, or (2) the precedence of the operator on top of opStack is strictly less than that of the current operator in the input string. In this case, we pop both the $*$ and $+$ operators and place them into the output stream:

Input: $a + b * c - d$ Output: $a b c * +$ opStack: $+$
 \uparrow
 Input: $a + b * c - d$ Output: $a b c * +$ opStack: empty

The operator $-$ is pushed on top of the stack.

This scanning process continues until we come to the end of the input. Then, any remaining operators on opStack are popped into the output and the algorithm terminates.

Input: $a + b * c - d$ Output: $a b c * +$ opStack: $-$
 \uparrow
 Input: $a + b * c - d$ Output: $a b c * + d$ opStack: $-$
 \uparrow
 Input: $a + b * c - d$ Output: $a b c * + d -$ opStack: empty
 \uparrow

The algorithm has produced the expression $a b c * + d -$, the correct postfix representation of the original input. In this example, a stack was the right structure to hold operators from the input stream because of the nature of the algorithm. Operators must be saved until we determine their proper location, and then they are placed into the output stream in the reverse order of their occurrence in the input. Again, this is a perfect match with the LIFO model of a stack.

Implement and check the correctness of the program against following test cases:

Test case1- Provide input as: $a+b*c$. Check if the output is $abc*+$

Test case2- Provide input as: $(a+b)*c$. Check if the output is $ab+c*$

Test case3- Provide input as: $a+b*c/d$. Check if the output is $abc*d/+$

Conclusion: Postfix expressions are used in compiler construction. The transformation of infix to postfix expression studied and implemented successfully. The program was tested with all test cases (order of operators, with and without brackets) .

Upload your program (rollno_expt2) with output for the test cases [here](#).



Experiment No. 3

Aim: Implementation of evaluation of postfix expression using stack.

Theory:

The postfix expression is easily evaluated by the computers using stack. For the evaluation of postfix expression, computer need not remember the precedence of operators and brackets.

Algorithm

1. scan every character of the postfix expression and repeat step 1 and 2 until the end of the expression.
2. If an operand is encountered the push it on top of stack
Else if an operator 'o' is encountered then
 - a. Pop the top two operands from stack op1, op2.
 - b. Perform the operation op1 o op2.
 - c. Push the result of step b on top of stack
3. Set result equal to the topmost element of the stack

Example:

Input- 12+3* stack:1
↑12+3* stack:1 2
↑
12+3* stack:3
↑
12+3* stack:3 3
↑
12+3* stack:9
↑

Output: 3

Implement and check the correctness of the program against following test cases:

Test case1- Provide input as: 1 2 + 3 * . Check if the output is 9

Test case2- Provide input as: 9 4 1 - /. Check if the output is 3

Test case3- Provide input as: 2 3 5 * 5 / + Check if the output is 5

Conclusion: The postfix expression is evaluated using stack and the program is successfully tested.

Upload your program (rollno_expt3) with output for the test cases [here](#).



Experiment No. 4

Aim: Implement Linear Queue ADT using array.

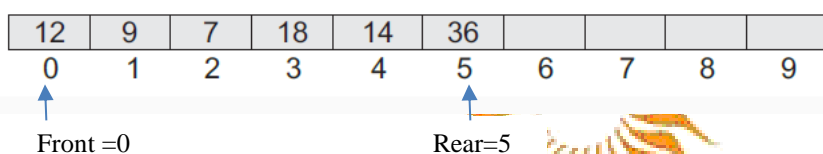
Theory:

A queue is a **FIFO** (First-In, First-Out) data structure in which the element that is inserted first is the first one to be taken out. The elements in a queue are added at one end called the **REAR** and removed from the other end called the **FRONT**.

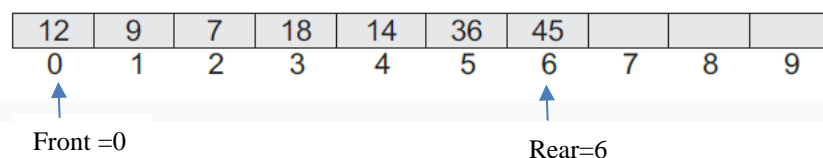
Queues can be implemented by using either arrays or linked lists. In this experiment, we will study and implement linear queue using arrays.

ARRAY REPRESENTATION OF QUEUES

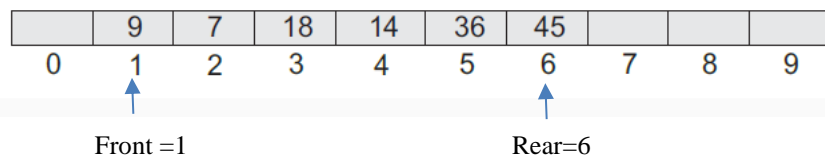
Queues can be easily represented using linear arrays. As stated earlier, every queue has **front** and **rear** variables that point to the position from where deletions and insertions can be done, respectively. The array representation of a queue is shown in Figure below with **FRONT = 0** and **REAR = 5**.



Suppose we want to add another element with value 45, then **REAR** would be incremented by 1 and the value would be stored at the position pointed by **REAR**.



If we want to delete an element from the queue, then the value of **FRONT** will be incremented. Deletions are done from only this end of the queue.



Linear Queue ADT

Queue Q, Front = -1, Rear = -1

Operations

1. Insert(Q, val) – Insert new element at the end of the queue.

Handwritten notes:
1st element is inserted
R = -1
F = 0
R = 0
Q[0] = 12
R++
Q[R] = 9

2. Remove (Q)- Removes element from the front of the queue.
3. isEmpty(Q)- returns true if queue is empty else returns false
4. IsFull(Q)- returns true if queue is full else returns false

Insert operation

Step 1: IF REAR == MAX-1

Write OVERFLOW

Goto step 4

[END OF IF]

Step 2: IF FRONT = -1 and REAR = -1

SET FRONT = REAR = 0

ELSE

SET REAR = REAR + 1

[END OF IF]

Step 3: SET QUEUE[REAR] = NUM

Step 4: EXIT

Remove operation

Step 1: IF FRONT == -1 OR FRONT > REAR

Write UNDERFLOW

ELSE

SET VAL = QUEUE[FRONT]

SET FRONT = FRONT + 1

[END OF IF]

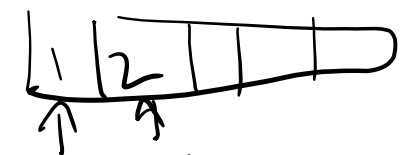
Step 2: EXIT

Implement linear queue using arrays in C and check the correctness of the program against following test cases:

set the size of the queue to 5. -

Test case1-

1. Insert 6 elements and show five elements (1,2,3,4,5,6) are inserted successfully and for 6th element, you get an overflow message.



F=0 R=1



F=1 R=1

remove
remove



R=1 F=2

2. Remove elements 6 times and check that elements are displayed in the order 1 2 3 4 5 underflow.

Conclusion: The linear queue is implemented and tested against overflow and underflow conditions.



Experiment No. 5

Aim: Implement Circular Queue ADT using array.

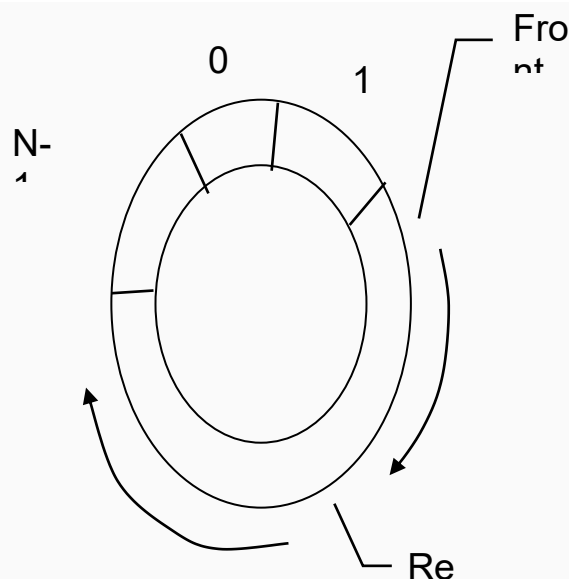
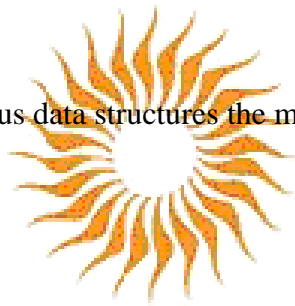
Theory:

A circular queue is one in which the insertion of new element is done at the very first location of the queue if the last location of the queue is full. Suppose if we have a Queue of n elements then after adding the element at the last index i.e. $(n-1)$ th, as queue is starting with 0 index, the next element will be inserted at the very first location of the queue which was not possible in the simple linear queue. That's why linear queue leads to wastage of the memory, and this flaw of linear queue is overcome by circular queue.

Operations on a Queue:

Just like various other homogeneous data structures the main operations that can be performed on a circular queue are:

1. Insertion
2. Deletion
3. Traverse



In a standard queue data structure re-buffering problem occurs for each de-queue operation. To solve this problem by joining the front and rear ends of a queue to make the queue as a circular queue

Circular queue is a linear data structure. It follows FIFO principle.

- In circular queue the last node is connected back to the first node to make a circle.
- Circular linked list follow the First In First Out principle
- Elements are added at the rear end and the elements are deleted at front end of the queue
- Both the front and the rear pointers points to the beginning of the array.
- It is also called as “Ring buffer”.
- Items can inserted and deleted from a queue in $O(1)$ time.

Circular Queue can be created in three ways they are

- Using single linked list
- Using double linked list
- Using arrays

Algorithm:

Insert Circular ():

Description: Here **QUEUE** is an array with **N** locations. **FRONT** and **REAR** points to the front and rear elements of the **QUEUE**. **ITEM** is the value to be inserted.

1. If (**FRONT** == 1 and **REAR** == **N**) or (**FRONT** == **REAR** + 1) Then

2. Print: Overflow

3. Else

4. If (**REAR** == 0) Then

[Check if **QUEUE** is empty]

(a) Set **FRONT** = 1

(b) Set **REAR** = 1

5. Else If (**REAR** == **N**) Then

[If **REAR** reaches end if **QUEUE**]

6. Set **REAR** = 1

7. Else

8. Set **REAR** = **REAR** + 1

[Increment **REAR** by 1]

[End of Step 4 If]

9. Set $QUEUE[REAR] = ITEM$
10. Print: ITEM inserted
[End of Step 1 If]
11. Exit.

Delete Circular ():

Description: Here **QUEUE** is an array with **N** locations. **FRONT** and **REAR** points to the front and rear elements of the **QUEUE**.

1. If $(FRONT == 0)$ Then [Check for Underflow]
2. Print: Underflow
3. Else
4. $ITEM = QUEUE[FRONT]$
5. If $(FRONT == REAR)$ Then [If only element is left]
 - (a) Set $FRONT = 0$
 - (b) Set $REAR = 0$
6. Else If $(FRONT == N)$ Then [If FRONT reaches end of QUEUE]
7. Set $FRONT = 1$
8. Else
9. Set $FRONT = FRONT + 1$ [Increment FRONT by 1]
[End of Step 5 If]
10. Print: ITEM deleted
[End of Step 1 If]
11. Exit



Implement and check the correctness of the program against following test cases:

Set size of array to 5.

Test case1- Insert 6 elements and show five elements (1,2,3,4,5,6) are inserted successfully and for 6th element, you get an overflow message.

Test case2- Remove elements 6 times and check that elements are displayed in the order 1 2 3 4 5 underflow.

Test case3- Insert 5 elements (7,2,9,4,5) and display the queue. Check if the output is 7 2 9 4 5 .

Conclusion: The circular queue is a linear data structure implemented using array and tested against overflow and underflow conditions.

Experiment No. 6

Aim: Implement Singly Linked List ADT and operations- insertbeg, insertend, delete and display.

Theory:

Linked list is dynamic data structure in which elements can be inserted and deleted anywhere without shifting. In contrast to fixed size in arrays, linked list can allocate memory to data elements during run time also.

A linked list consists of nodes that are dynamically created by allotting memory during run time. The node has two parts: data and pointer to next node.

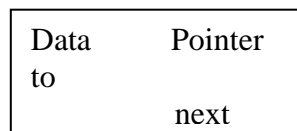


Figure 1. Linked List Node Structure

The linked list consists of node connected sequentially as shown in figure 2.

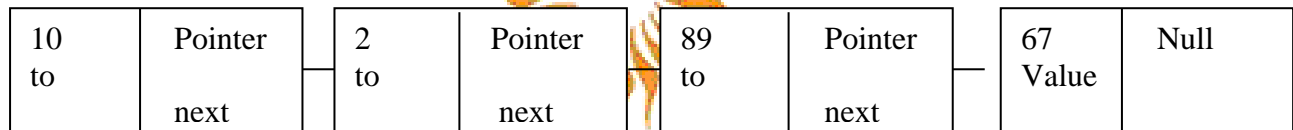


Figure 2. Linked List

The structure of the node in C is-

struct node

```
{
int data;
struct node *next;
};
```

Types of Linked List

Following are the various flavours of linked list.

- **Simple Linked List** – Item Navigation is forward only.
- **Doubly Linked List** – Items can be navigated forward and backward way.
- **Circular Linked List** – Last item contains link of the first element as next and first element has link to last element as prev.

Basic Operations

Following are the basic operations supported by a list.

- **Insertion** – add an element at the beginning, in between or end of the list.
- **Deletion** – delete an element anywhere in the list.
- **Display** – displaying complete list.
- **Search** – search an element using given key.

Advantages:

- i. Dynamic memory allocation
- ii. Quick insertion and deletion

Disadvantages:

- i. Sequential access.



Implement and check the correctness of the program against following test cases:

Test case1- Call insertbeg() operation to insert four elements: 4, 5, 6, 7 in the same order and display the linked list. Check if the output is- 7 6 5 4.

Test case2- Call insertend() operation to insert four elements: 8, 9, 10, 11 in the same order and display the linked list. Check if the output is- 7 6 5 4 8 9 10 11.

Test case3- Call delete() operation to delete element: 4 and display the linked list. Check if the output is- 7 6 5 8 9 10 11.

Test case4- Call delete() operation to delete element: 7 and display the linked list. Check if the output is- 6 5 8 9 10 11.

Test case5- Call delete() operation to delete element: 12 and check if element not found message is printed.

Conclusion: Implemented insertion, deletion and traversal operations of Singly linked list in C.

Experiment No. 7

Aim: Implement Circular Linked List ADT and operations- insertbeg, insertend, delete and display.

Theory:

In circular linked list, the last node contains a pointer to the first node of the list.

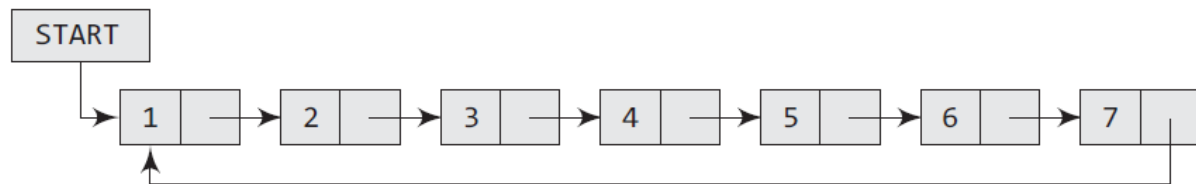


Figure 6.26 Circular linked list

The operations to be performed are-

1. Insert
2. Delete
3. Traverse.

Traverse operations-

```
void traverse (struct node *start)
{ struct node *ptr;
ptr=start;
while (ptr->next != start)
{
printf ("%d", ptr->data);
ptr=ptr->next;
}
printf ("%d", ptr->data);
}
```

Insert operation-

Step 1: IF AVAIL = NULL
Write OVERFLOW

Go to Step 11

[END OF IF]

Step 2: SET = AVAIL

Step 3: SET AVAIL = AVAIL NEXT

Step 4: SET NEWNODE->DATA = VAL

Step 5: SET PTR = START

Step 6: Repeat Step 7 while PTR -> NEXT != START

Step 7: PTR = PTR NEXT

[END OF LOOP]

Step 8: SET NEWNODE-> NEXT = START

Step 9: SET PTR->NEXT = NEWNODE

Step 10: SET START = NEWNODE

Step 11: EXIT

Implement linear queue using arrays in C and check the correctness of the program against following test cases:

Test case1- Call insertbeg() operation to insert four elements: 4, 5, 6, 7 in the same order and display the linked list. Check if the output is- 7 6 5 4.

Test case2- Call insertend() operation to insert four elements: 8, 9, 10, 11 in the same order and display the linked list. Check if the output is- 7 6 5 4 8 9 10 11.

Test case3- Call delete() operation to delete element: 4 and display the linked list. Check if the output is- 7 6 5 8 9 10 11.

Test case4- Call delete() operation to delete element: 7 and display the linked list. Check if the output is- 6 5 8 9 10 11.

Test case5- Call delete() operation to delete element: 12 and check if element not found message is printed.

Conclusion: Implemented insertion, deletion and traversal operations of circular linked list in C and successfully tested the correctness of the program.

Experiment No. 8


Aim: Implement Stack Queue ADT using Linked List.

Theory:

Instead of using array, we can also use linked list to implement stack. Linked list allocates the memory dynamically. However, time complexity in both the scenario is same for all the operations i.e. push, pop and peek.

Adding a node to the stack (Push operation)

Adding a node to the stack is referred to as **push** operation. Pushing an element to a stack in linked list implementation is different from that of an array implementation. In order to push an element onto the stack, the following steps are involved.

- 
1. Create a node first and allocate memory to it.
 2. If the list is empty then the item is to be pushed as the start node of the list. This includes assigning value to the data part of the node and assign null to the address part of the node.
 3. If there are some nodes in the list already, then we have to add the new element in the beginning of the list (to not violate the property of the stack). For this purpose, assign the address of the starting element to the address field of the new node and make the new node, the starting node of the list.

Deleting a node from the stack (POP operation)

Deleting a node from the top of stack is referred to as **pop** operation. Deleting a node from the linked list implementation of stack is different from that in the array implementation. In order to pop an element from the stack, we need to follow the following steps :

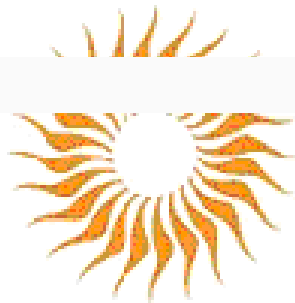
1. **Check for the underflow condition:** The underflow condition occurs when we try to pop from an already empty stack. The stack will be empty if the head pointer of the list points to null.
2. **Adjust the head pointer accordingly:** In stack, the elements are popped only from one end, therefore, the value stored in the head pointer must be deleted and the node must be freed. The next node of the head node now becomes the head node.

Implement and check the correctness of the program against following test cases:

Test case1- Call push operation to insert four elements: 4, 5, 6, 7 in the same order and display the stack. Check if the output is- 7 6 5 4.

Test case2- Call pop operation five times and check if the output is- 7 6 5 4 stack empty.

Conclusion: The stack ADT is successfully implemented using linked list and the test cases are tested.

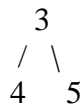


Experiment No. 9

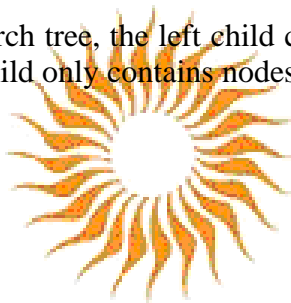
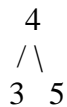
Aim: Implement Binary Search Tree ADT and operations- insert and traverse (inorder, postorder, preorder) using Linked List.

Theory:

Binary tree: In short, a [binary tree](#) is a tree where each node has up to two leaves. In a binary tree, a left child node and a right child node contain values which can be either greater, less, or equal to parent node.



Binary Search Tree: In binary search tree, the left child contains nodes with values less than the parent node and where the right child only contains nodes with values greater than or equal to the parent.



Operations on binary search tree

Operations, such as *find*, on a binary search tree require comparisons between nodes. These comparisons are made with calls to a *comparator*, which is a function that computes the total order (linear order) on any two keys. This *comparator* can be explicitly or implicitly defined, depending on the language in which the binary search tree was implemented. A common *comparator* is the less-than function, for example, $a < b$, where a and b are keys of two nodes a and b in a binary search tree.

Searching

Searching a binary search tree for a specific key can be a [recursive](#) or an [iterative](#) process.

We begin by examining the [root node](#). If the tree is *null*, the key we are searching for does not exist in the tree. Otherwise, if the key equals that of the root, the search is successful and we return the node. If the key is less than that of the root, we search the left subtree. Similarly, if the key is greater than that of the root, we search the right subtree. This process is repeated until the key is found or the remaining subtree is *null*. If the searched key is not found before a *null* subtree is

reached, then the item must not be present in the tree. This is easily expressed as a recursive algorithm:

```
function Find-recursive(key, node): // call initially with node = root  
    if node = Null or node.key = key then  
        return node  
    else if key < node.key then  
        return Find-recursive(key, node.left)  
    else  
        return Find-recursive(key, node.right)
```

The same algorithm can be implemented iteratively:

```
function Find(key, root):  
    current-node := root  
    while current-node is not Null do  
        if current-node.key = key then  
            return current-node  
        else if key < current-node.key then  
            current-node := current-node.left  
        else  
            current-node := current-node.right  
    return Null
```



Because in the worst case this algorithm must search from the root of the tree to the leaf farthest from the root, the search operation takes time proportional to the tree's *height*. On average, binary search trees with n nodes have $O(\log n)$ height. However, in the worst case, binary search trees can have $O(n)$ height, when the unbalanced tree resembles a [linked list](#) ([degenerate tree](#)).

Insertion and search

Insertion begins as a search would begin; if the key is not equal to that of the root, we search the left or right sub-trees as before. Eventually, we will reach an external node and add the new key-value pair (here encoded as a record 'newNode') as its right or left child, depending on the node's key. In other words, we examine the root and recursively insert the new node to the left subtree if its key is less than that of the root, or the right subtree if its key is greater than or equal to the root.

Here's how a typical binary search tree insertion might be performed in a binary tree :

```
void insert(Node*& root, int data) {  
    if (!root)  
        root = new Node(data);  
    else if (data < root->data)  
        insert(root->left, data);
```

```

else if (data > root->data)
    insert(root->right, data);
return;
}

```

The part that is rebuilt uses $O(\log n)$ space in the average case and $O(n)$ in the worst case (see [big-O notation](#)). In either version, this operation requires time proportional to the height of the tree in the worst case, which is $O(\log n)$ time in the average case over all trees, but $O(n)$ time in the worst case. Another way to explain insertion is that in order to insert a new node in the tree, its key is first compared with that of the root. If its key is less than the root's, it is then compared with the key of the root's left child. If its key is greater, it is compared with the root's right child. This process continues, until the new node is compared with a leaf node, and then it is added as this node's right or left child, depending on its key. There are other ways of inserting nodes into a binary tree, but this is the only way of inserting nodes at the leaves and at the same time preserving the BST structure.

Tree traversal is the process of visiting all the nodes in a tree in a specific order. There are three types of tree traversals.

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



Pre-order Traversal:

- Visit the root node.
- Recursively traverse left sub-tree(in pre-order).
- Recursively traverse right sub-tree(in pre-order).

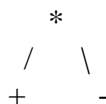
Algorithm for Pre-Order tree traversal:

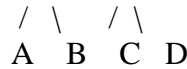
```

void preorder(Treenode T) {
    if (T == NULL)
        return;
    printData(T->data);
    preorder(T->left);
    preorder(T->right);
    return;
}

```

Example:

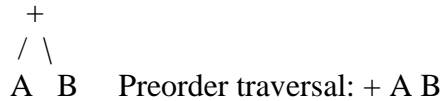




Step 1: Visit root node and store the key '*' to output.

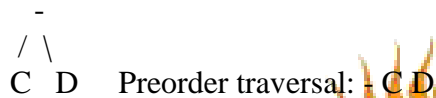
Output: *

Step 2: Recursively traverse left subtree of '*'



Output: * + A B

Step 3: Recursively traverse right subtree of '*'



Output: * + A B - C D

Preorder: * + A B - C D



Post Order Traversal:

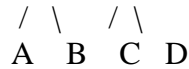
- Recursively traverse left subtree(in post order).
- Recursively traverse right subtree(in post order).
- Visit the root node.

Algorithm for Post order tree traversal:

```
void postorder(Treenode T) {
    if (T == NULL)
        return;
    postorder(T->left);
    postorder(T->right);
    printData(T->data);
    return;
}
```

Example:





Preorder: A B + C D - *

Inorder Traversal:

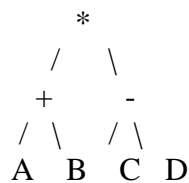
- Recursively traverse left subtree(inorder).
- Visit the root node.
- Recursively traverse right subtree(inorder).

Algorithm for Inorder tree traversal:

```
void inorder(Treenode T) {
    if (T == NULL)
        return;
    inorder(T->left);
    printData(T->data);
    inorder(T->right);
    return;
}
```



Example:



In-order: A + B * C - D

Implement and check the correctness of the program against following test cases:

Test case1- Call insert operation to insert six elements: 10, 5, 15, 2, 7, 20 in the same order.

Test case2- Call three traversals and check if the output is-

Inorder- 2 5 7 10 15 20

Postorder- 2 7 5 20 15 10

Preorder- 10 5 2 7 15 20

Conclusion: Binary search tree implemented using insert, traverse and search operations.

Experiment No. 10

Aim: Implement Graph Traversal techniques: a) Depth First Search b) Breadth First Search

Theory:

A graph is a finite set of nodes with edges between nodes. Formally, a graph G is a structure (V, E) consisting of

- a finite set V called the set of nodes, and
- a set E that is a subset of $V \times V$. That is, E is a set of pairs of the form (x, y) where x and y are nodes in V

Graph traversal refers to the problem of visiting all the nodes in a graph in a particular manner. Tree traversal is a special case of graph traversal. In contrast to tree traversal, in general graph traversal, each node may have to be visited more than once, and a root-like node that connects to all other nodes might not exist.

Depth-first Search:

A Depth-first search (DFS) is a technique for traversing a finite undirected graph. DFS visits the child nodes before visiting the sibling nodes, that is, it traverses the depth of the tree before the breadth. Usually a stack is used in the search process.

Breadth-first Search:

A Breadth-first search (BFS) is another technique for traversing a finite undirected graph. BFS visits the sibling nodes before visiting the child nodes. Usually a queue is used in the search process.

Algorithm:

Depth-First Search:

Step 1. Select the root node.
Step 2. Mark all nodes unvisited.
for $(i=0; i < n; i++)$ visited $[i]=0$;
Step 3. Insert root node into the queue
push(stack, root)
Step 4. while (stack not empty)
{

```
u= pop(stack)
If (u is not visited)
{
Mark u as visited, visited[u]=1
Print u;
push all adjacent nodes of u that are not visited.
}
}
```

Breadth-First Search:

```
Step 1. Select the root node.
Step 2. Mark all nodes unvisited.
for (i=0; i<n; i++) visited[i]=0;
Step 3. Insert root node into the queue
insert(queue, root)
Step 4. while (queue not empty)
{
u= delete(queue)
If (u is not visited)
{
Mark u as visited, visited[u]=1
Print u;
Insert all adjacent nodes of u that are not visited.
}
}
```



Implement and check the correctness of the program against following test cases:

Test case1- Create a graph using five nodes defined as - $V = \{a, b, c, d, e\}$

$E = \{ab, ac, bd, cd, de\}$

Perform DFS and BFS using root node a.

Check if the output is

DFS- a c d e b

BFS- a b c d e

Conclusion: Graph is implemented using adjacency matrix and the DFS and BFS traversal implemented successfully



