LAB MANUAL



**BCSE2361: DATA STRUCTURES & ALGORITHMS**

**LAB MANUAL**

|  |  |  |
| --- | --- | --- |
| L T P C | | **0 0 2 3** |
| Course Type | | **CORE** |
| Semester Offered | | **III** |
| Academic Year | | **2022-2023** |
| Slot | |  |
| Class Room | |  |
| **Faculty Details:** | | |
| Name | **Abdul Mazid** | |
| Website link |  | |
| Designation | **Assistant Professor** | |
| School | **School of Computing Science and Engineering** | |
| Cabin No |  | |
| Intercom | **-------** | |
| Open Hours |  | |
|  |  | |

Table of Contents

|  |  |  |
| --- | --- | --- |
| 1. | Course details  1.1.Courseobjective (s) |  |
|  | 1.2. Pre-requisites  1.3.LearningOutcomes |  |
| 2. | 1.4.Syllabus &References  List of Experiments |  |
| 3. | Experimental Setup details for the course. |  |
| 4. | Experiment details |  |

4.1.1. Implementation of Stack, Queue, Linked list Algorithms

4.1.2. Implementation of trees, Graphs Algorithm

4.1.3. Implementation of Searching Algorithm.

4.1.4. Implementation of Sorting Algorithms.

5. Guidelines for continuous assessment

5.1. Format for continuous assessment

5.2. Format for internal end semester assessment

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Name of The Course | Data Structures and Algorithms Lab | L | T | P | C |
| Course Code | BCSE2361 | 0 | 0 | 2 | 3 |
| Version No |  | | | | |
| Prerequisite |  | | | | |
| Co requisite |  | | | | |
| Anti- requisite |  | | | | |

Course Objectives:

1. Introduce the fundamentals and abstract concepts of Data Structures.
2. Introduce searching, sorting techniques
3. Learn how concepts of data structures are useful in problem solving.

Course Outcomes:

Upon successful completion of this course, students will be able to

|  |  |
| --- | --- |
| CO1 | Understand the comparison and use of Recursion and Loops |
| CO2 | Understand the application of linear data structure(s) to solve various problems |
| CO3 | Understand the application of non linear data structure(s) to solve various problems |
| CO4 | Understand the shortest path algorithms involving complicated data structures like Graphs. |
| CO5 | Become expert in calculating and comparing complexities of various searching and sorting algorithms. |
| CO6 | Gain understanding of latest trends and research areas in the course |

Text Books

1. Horowitz and Sahani, “Fundamentals of Data Structures”, Galgotia Publication

Reference Books

1. Aaron M. Tenenbaum, Yedidyah Langsam and Moshe J. Augenstein “Data Structures Using C and C++” , PHI
2. Jean Paul Trembley and Paul G. Sorenson, “An Introduction to Data Structures with applications”, McGraw Hill
3. R. Kruse etal, “Data Structures and Program Design in C”, Pearson Education
4. Lipschutz, “Data Structures” Schaum’s Outline Series, TMH
5. G A V Pai, “Data Structures and Algorithms”, TMH

Course Content

|  |  |  |
| --- | --- | --- |
| Unit 1 | Module 1 | 9 Hours |
| Elementary Data Organization, Algorithm, Efficiency of an Algorithm, Time and Space Complexity, Asymptotic notations: Big-Oh, Time-Space trade-off. Abstract Data Types (ADT)Arrays: Definition, Single and Multidimensional Arrays, Representation of Arrays: Row Major Order, and Column Major Order, Application of arrays, Sparse Matrices and their representations. Linked lists: Array Implementation and Dynamic Implementation of Singly Linked Lists, Doubly Linked List, Circularly Linked List, Operations on a Linked List. Insertion, Deletion, Traversal, Polynomial Representation and Addition, Generalized Linked List | | |
| Unit II | Module 2 | 8 Hours |
| Primitive Stack operations: Push & Pop, Array and Linked Implementation of Stack in C, Application of stack: Prefix and Postfix Expressions, Evaluation of postfix expression, Recursion, Tower of Hanoi Problem, Simulating Recursion, Principles of recursion, Tail recursion, Removal of recursion | | |
| Unit III | Module 3 | 8 Hours |
| Binary Trees, Binary Tree Representation: Array Representation and Dynamic Representation, Complete Binary Tree, Algebraic Expressions, Extended Binary Trees, Array and Linked Representation of Binary trees, Tree Traversal algorithms: Inorder, Preorder and Postorder, Threaded Binary trees, Traversing Threaded Binary trees, Huffman algorithm. | | |
| Unit-IV | Module 4 | 7 Hours |
| Terminology, Sequential and linked Representations of Graphs: Adjacency Matrices, Adjacency List, Adjacency Multi list, Graph Traversal : Depth First Search and Breadth First Search, Connected Component, Spanning Trees, Minimum Cost Spanning Trees: Prims and Kruskal algorithm. Transitive Closure and Shortest Path algorithm: Dijikstra Algorithm | | |
| Unit-V | Module 5 | 8 Hours |
| Sequential search, Binary Search, Comparison and Analysis Internal Sorting: Insertion Sort, Selection, Bubble Sort, Shell sort. | | |
| Unit-VI | Advancements and Research | 6 Hours |
| The advances and the latest trends in the course as well as the latest applications of the areas covered in the course. The latest research conducted in the areas covered in the course. Discussion of some latest papers published in IEEE transactions and ACM transactions, Web of Science and SCOPUS indexed journals as well as high impact factor conferences as well as symposiums. Discussion on some of the latest products available in the market based on the areas covered in the course and patents filed in the areas covered in the course. | | |

Continuous Assessment Pattern

|  |  |  |  |
| --- | --- | --- | --- |
| Internal Assessment (IA) | Mid Term Test (MTE) | End Term Test (ETE) | Total Marks |
| 20 | 30 | 50 | 100 |

**Mode of Evaluation:** Continuous Lab evaluation using File, Viva and Lab performance

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Theory** | | **Laboratory** | | **Theory and laboratory** |
| **Components** | **Internal** | **SEE** | **Internal** | **SEE** |
| **Marks** | 50 | 50 | 70 | 30 |
| **Total Marks** | 100 | | 100 | |
| **Scaled Marks** | 75 | | 25 | | 100 |

**Course Lab Outcomes Assessment**

The laboratory component strongly contributes towards the program outcome **Design/development of solutions-PO (3)**.This evaluation method will be used for the evaluation of lab and program outcomes of this course.

**Direct Measurement Report**

BCSE 2361 POutcome (3) Report Form

**Measure**– percent of students scoring at least 70%marks in lab.

**Target–** 70%ofstudents

|  |  |  |  |
| --- | --- | --- | --- |
| **S. No.** | **Course Outcomes** | **Mapped Program Outcomes** | **Mapping** |
| 1 | To become expert in identifying and using linear and non linear data structures. | 1,2,3,4,13, 15 | Exp No. 1,2, 3 Mid- term internal evaluation, End-term external evaluation |
| 2 | To become expert in application of linked list data structure to solve various problems. | 1,2,15 | Exp No. 4, 5, 6 Mid- term internal evaluation, End-term external evaluation |
| 3 | To become proficient in analyzing order and time space tradeoff under complexity | 1,2,4, 5, 15 | Exp No. 1,2,3, 18, 19, 20 End- term internal evaluation, End-term external evaluation |
| 4 | To become expert in application of stack, queue, tree and graph data structure to solve various problems. | 1,2, 4, 15 | Exp No. 7-17, End- term internal evaluation, End-term external evaluation |
| 5 | To become expert in calculating complexties of various serching and sorting algorithms. | 1, 2, 4, 15 | Exp No. 18-22, End- term internal evaluation, End-term external evaluation |
| 6 | Able to effectively choose the data structure that efficiently model the information in a problem | 1,2, 4, 15, 16 | Exp No. 20, 22, End- term internal evaluation, End-term external evaluation |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Program Outcome→ | | Engineering Knowledge | Problem analysis | Design/development of solutions | Conduct investigations of complex problems | Modern tool usage | The engineer and society | Environment and sustainability | Ethics | Individual or team work | Communication | Project management and finance | Life-long Learning | integrate Computer Science & Engineering into other fields | understanding of the various technologies and tools associated with Big Data | integrate knowledge and experience in a real life with an understanding and implementation of Data Structures, Algorithms and Advanced Programming | Integrate & understanding  of the  knowledge of the  system sciences and  and intelligence sciences and their applications |
| Course Code | Course Name | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| BCSE2361 | DS Lab | 3 | 3 | 1 | 3 | 1 |  |  |  |  |  |  |  | 1 |  | 3 | 3 |

1=addressed to small extent

2= addressed significantly

3=major part of course

|  |
| --- |
| LAB |
| THEORY |

LISTOFEXPERIMENTS

|  |  |
| --- | --- |
| **S. No.** | **Title of Lab Experiments** |
|  | Write a program to compute minimum/maximum of a given array. |
|  | 1. Write a program to sort given set of numbers in ascending/descending order using Bubble Sort and analyze its complexity. |
|  | 1. Write a menu-based program to perform array operations: deletion of an element from the specified position, inserting an element at the specified position, printing the array elements. |
|  | 1. Write a program to search an element in the array using linear search. |
|  | Write a program to search an element in a 2-dimensional array. |
|  | Write a program to perform following operations in matrix:   1. Addition 2. Subtraction 3. Multiplication 4. Transpose |
|  | Write a menu-based program to perform following operations on single linked list:   * 1. To insert a node at the beginning of the list.   2. To insert a node at the end of the list.   3. To insert a node after a given node in the list.   d. To delete the first node from the list.  e. To delete the last node from the list.  f. To delete a node after a given node from the list.  g. To delete a node at a given position from the list. |
|  | Write a menu-based program to perform following operations on double linked list:   1. To insert a node at the beginning of the list. 2. To insert a node at the end of the list. 3. To insert a node after a given node in the list.   d. To delete the first node from the list.  e. To delete the last node from the list.  f. To delete a node after a given node from the list.  g. To delete a node at a given position from the list. |
|  | Write a menu-based program to perform following operations on circular linked list:   1. To insert a node at the beginning of the list. 2. To insert a node at the end of the list. 3. To insert a node after a given node in the list. 4. To delete the first node from the list.   e. To delete the last node from the list.  f. To delete a node after a given node from the list.  g. To delete a node at a given position from the list. |
|  | Write a menu-based program to implement stack operations: PUSH, POP using array implementation of stack. |
|  | Write a menu-based program using functions to implement stack operations: PUSH, POP using linked implementation of stack. |
|  | Write a program to convert infix expression into postfix expression and then to evaluate resultAnti- postfix expression. |
|  | Write a program to solve Towers of Hanoi Problem. |
|  | Write a menu-based program to implement linear queue operations: INSERTION, DELETION using array implementation of queue. |
|  | Write a menu-based program to implement linear queue operations: INSERTION, DELETION using linked list implementation of queue. |
|  | Write a menu-based program to implement circular queue operations: INSERTION, DELETION. |
|  | Write a program to traverse a binary tree using PRE-ORDER, IN-ORDER, POST-ORDER traversal techniques. |
|  | Write a menu-based program to perform operations for a binary search tree (BST).   1. Search an element 2. Find minimum 3. Find maximum 4. Insertion 5. Deletion |
|  | Write a program to traverse a graph using breadth-first search (BFS), depth-first search (DFS). |
|  | Write a program to sort a given set of numbers in ascending/descending order using insertion sort and also search a number using binary search. |
|  | Write a program to sort a given set of numbers in ascending/descending order using Quick sort and selection sort. Also record the time taken by these two programs and compare them. |
|  | Write a program to sort a given set of numbers in ascending/descending order using Merge sort. |
| **Value Added Experiments** | |
|  | Implement the Towers of Hanoi Problem using graphical view. |
|  | Design, develop, and execute a program in C to convert a given valid parenthesized infix arithmetic expression to postfix expression and then to print both the expressions and then to evaluate resultant expression using Stack. The expression consists of single character operands and the binary operators + (plus), - (minus), \* (multiply) and / (divide). |

**EXPERIMENTAL SETUP DETAILS FOR THE COURSE**

**Software Requirements**

TurboC2.0/Turbo C++3.0+

**Hardware Requirements**

No specific requirements. Any computer Hardware capable of running DOS can be used for this course.

**EXPERIMENT DETAILS**

**Experiment No. 1**

Title: WAP to compute minimum/ maximum of a given array.

**Algorithm for straight forward maximum and minimum**

StraightMaxMin(a,n,max,min)

// set max to the maximum and min to the minimum of a[1:n].

{

     max := min := a[1];

     for i := 2 to n do

     {

           if(a[i] > max) then max := a[i];

           if(a[i] > min) then min := a[i];

     }

}

**Experiment No: 2**

**Title:** Write a menu-based program to perform array operations: deletion of an element from the specified position, inserting an element at the specified position, printing the array elements.

**Insertion Operation**

Insert operation is to insert one or more data elements into an array. Based on the requirement, new element can be added at the beginning, end or any given index of array.

Let LA is a Linear Array (unordered) with N elements and K is a positive integer such that K<=N. Below is the algorithm where ITEM is inserted into the Kth position of LA –

1. Start

2. Set J=N

3. Set N = N+1

4. Repeat steps 5 and 6 while J >= K

5. Set LA[J+1] = LA[J]

6. Set J = J-1

7. Set LA[K] = ITEM

8. Stop

**Deletion Operation**

Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

Consider LA is a linear array with N elements and K is a positive integer such that K<=N. Below is the algorithm to delete an element available at the Kth position of LA.

1. Start

2. Set J=K

3. Repeat steps 4 and 5 while J < N

4. Set LA[J-1] = LA[J]

5. Set J = J+1

6. Set N = N-1

7.Stop

**Experiment No: 3**

**Title:** Write a program to perform following operations in matrix:

1. Addition
2. Subtraction
3. Multiplication
4. Transpose

**Addition of Matrix**

1. Input a matrix say m1[n] with n elements

2. Input another matrix say m2[n] with n elements

3. for(i=0;i<r1;i++)do  
4. {  
5. for(j=0;j<c1;j++)do  
6.  m3[i][j]=m1[i][j]+m2[i][j];  
7.  }  
8. Print matrix m3

9. Stop

**Subtraction of Matrix**

1. Input a matrix say m1[n] with n elements

2. Input another matrix say m2[n] with n elements

3. for(i=0;i<r1;i++)do  
4. {  
5. for(j=0;j<c1;j++)do  
6.  m3[i][j]=m1[i][j]-m2[i][j];  
7.  }  
8. Print matrix m3

9. Stop

**Multiplication of Matrix**

1. Input a matrix say m1[n] with n elements

2. Input another matrix say m2[n] with n elements

3. for(i=0;i<r1;i++)do  
4. for(j=0;j<c2;j++)do  
5. {  
6. m3[i][j]=0;  
7. for(k=0;k<c1;k++)do  
8. m3[i][j]=m3[i][j]+m1[i][k]\*m2[k][j];  
9. }  
10. Print matrix m3

11. Stop

**Transpose of Matrix**1. Input a matrix say m1[n] with n elements

2. for(i=0;i<r1;i++)do  
3. {  
4. for(j=0;j<c1;j++)do  
5. {  
6. m2[j][i]=m1[i][j];  
7. }  
8. }  
9. Print matrix m2

10. Stop

**Experiment No: 4**

**Title:** Write a menu-based program to perform following operations on single linked list:

* 1. To insert a node at the beginning of the list.
  2. To insert a node at the end of the list.
  3. To insert a node after a given node in the list.

d. To delete the first node from the list.

e. To delete the last node from the list.

f. To delete a node after a given node from the list.

g. To delete a node at a given position from the list.

**Linked List**

linked list is a dynamic and linear data structure.

The Concept Of Link List

1. Link list used for the dynamic memory allocation.
2. Array and link list both are the linear data structure.
3. When we want to represent several lists by using arrays of varying size, either we have to represent.
4. each list using a separate array of maximum size or we have to represent each of the lists using one single array.
5. The first one will lead to wastage of storage, and the second will involve a lot of data movement.
6. A linked list is a list of elements in which the elements of the list can be placed anywhere in memory, and these elements are linked with each other using an explicit link field, that is, by storing the address of the next element in the link field of the previous element.

Structure of node

struct node

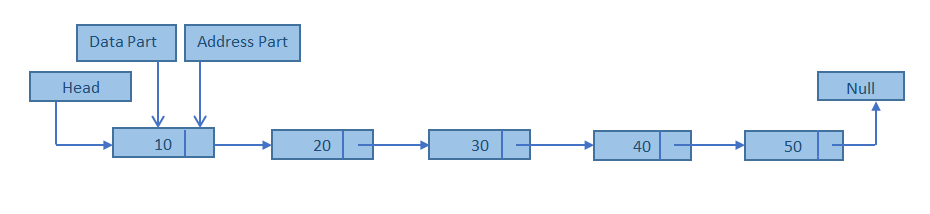
{

int data;

struct node \*next;

};

**Representation of link list**

Link list consists a series of structure. Each structure consists of a data field and address field. Data field consists data part and the address field contains the address of the successors.****

**Advantage of Link list**

1. Link list is an example of dynamic data structure. They can grow and shrink during the execution of program.
2. Efficient memory utilization. Memory is not pre allocated like static data structure. The allocation of memory depends upon the user ,i.e no need to pre-allocate memory
3. Insertion and deletion easily performed.
4. Linear Data Structures such as Stack,Queue can be easily implemeted using Linked list
5. Faster Access time,can be expanded in constant time without memory overhead

**Points to Remember**

1. Linked lists are used when the quantity of data is not known prior to execution.
2. In linked lists, data is stored in the form of nodes and at runtime, memory is allocated for creating nodes.
3. Due to overhead in memory allocation and deallocation, the speed of the program is lower.
4. The data is accessed using the starting pointer of the list.

**Algorithm**: Insert at begin.

void insert\_beg()

{

Input :the new node

temp=(struct node \*)malloc(sizeof(struct node));

Enter the value of node

temp->next=head;

head=temp;

}

**Algorithm**: Insert at end

void insert\_end()

{

Input: the new node

temp=(struct node \*)malloc(sizeof(struct node));

Enter the value at node

temp->next=NULL;

tail->next=temp;

tail=temp;

}

**Algorithm**: Insert at given position.

void insert\_pos(int n)

{

Enter the existing node after which you want insert a new node :

temp=head;

for(i=1;i<=n;i++) do

{

if(temp->info==pos) then

{

temp2=temp;

temp=temp->next;

Input: the new node :

item=(struct node \*)malloc(sizeof(struct node));

Enter the value of the node.

temp2->next=item;

item->next=temp;

c++;

}

temp=temp->next;

}

if(c==0) then

Output: the node not found

}

**Algorithm**: Delete from begin.

void del\_beg()

{

head=head->next;

}

**Algorithm**: Delete from end.

void del\_end(int n)

{

struct node \*temp;

int i;

temp=head;

for(i=1;i<=n;i++)

{

if(temp->next==tail)

tail=temp;

temp=temp->next;

}

tail->next=NULL;

}

**Algorithm**: Delete from given position.

void del\_pos(int n)

{

Enter the existing node after which you want delete a node :

temp=head;

for(i=1;i<=n;i++) do

{

if(temp->info==pos) then

{

temp2=temp;

temp2=temp2->next;

temp2=temp2->next;

temp->next=temp2;

c++;

}

temp=temp->next;

}

if(c==0) then

output: the node not found

}

**Algorithm**: Display the list.

void display(int n)

{

temp=head;

printf("the list is : \n");

for(int i=1;i<=n;i++)

printf("%d-> ",temp->info);

temp=temp->next;

}

**Experiment No. 5**

Title: Write a program to performs following operations on doubly linked lists:

1. To insert a node at the beginning of the list.
2. To insert a node at the end of the list.
3. To insert a node after a given node of the list.

d. To delete a node at the beginning of the list.

e. To delete a node at the end of the list.

f. To delete a node after a given node of the list.

A doubly linked list is a list that contains links to next and previous nodes. Unlike singly linked lists where traversal is only one way, doubly linked lists allow traversals in both ways. A generic doubly linked list node can be designed as:

typedef struct node { void\* data;

struct node\* next; struct node\* prev;

} node;

node\* head = (node\*) malloc(sizeof(node));

The design of the node allows flexibility of storing any data type as the linked list data. For example,

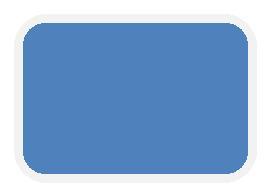
head data = malloc(sizeof(int)); \*((int\*)(head data)) = 12;

or

head data = malloc(strlen(“guna”)+1); strcpy((char\*)(head data), “guna”);

**Inserting to a Doubly Linked Lists**

Suppose a new node, **newnode** needs to be inserted after the node current



|  |  |  |
| --- | --- | --- |
| current |  | newnode |
|  |  |  |



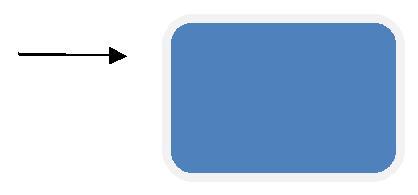
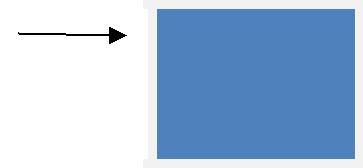
The following code can then be written

newnode next = current next; current next = newnode;

newnode prev = current; (newnode next) prev = newnode;

**Deleting a Node from a Doubly Linked Lists**

Suppose a new node, **current** needs to be deleted



current



The following code can then be written

node\* N = current prev

N next = current next;

(N next) prev = N;

**Insertion At Begining in doubly linked list**

**Algorithm**

1. InsertAtBegDll(info,prev,next,start,end)
2. create a new node and address in assigned to ptr.
3. check[overflow] if(ptr=NULL)
4. write:overflow and exit
5. set Info[ptr]=item;
6. if(start=NULL)
7. set prev[ptr] = next[ptr] = NULL
8. set start = end = ptr
9. else
10. set prev[ptr] = NULL
11. next[ptr] = start
12. set prev[start] = ptr
13. set start = ptr
14. [end if]
15. Exit.

**Insertion At Location in doubly linked list**

**Algorithm**

1. InsertAtlocDll(info,prev,next,start,end,loc,size)

1. set nloc = loc-1 , n=1

1. create a new node and address in assigned to ptr.

1. check[overflow] if(ptr=NULL)

1. write:overflow and exit

1. set Info[ptr]=item;

1. if(start=NULL)

1. set prev[ptr] = next[ptr] = NULL

1. set start = end = ptr

1. else if(nloc<=size)

1. repeat steps a and b while(n != nloc)
2. a. loc = next[loc]

1. b. n = n+1

1. [end while]

1. next[ptr] = next[loc]

1. prev[ptr] = loc

1. prev[next[loc]] = ptr

1. next[loc] = ptr

1. else

1. set prev[ptr] = end

1. next[end] = ptr

1. set ptr[next] = NULL

1. set end = ptr

1. [end if]

1. Exit.

**Insertion At Last in doubly linked list**

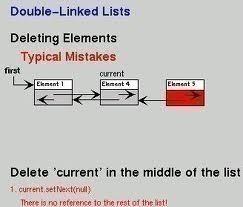
**Algorithm**

1. InsertAtEndDll(info,prev,next,start,end)
2. create a new node and address in assigned to ptr.
3. check[overflow] if(ptr=NULL)
4. write:overflow and exit
5. set Info[ptr]=item;
6. if(start=NULL)
7. set prev[ptr] = next[ptr] = NULL
8. set start = end = ptr
9. else
10. set prev[ptr] = end
11. next[end] = ptr
12. set ptr[next] = NULL
13. set end = ptr
14. [end if]
15. Exit.

**Deleting from the Beginning of the List**

An element from the beginning of the list can be deleted by performing the following steps:

* Assign the value of head (address of the first element of the list) to a temporary variable (say temp)
* There are two further cases:
  1. If there is only one element in the existing list, both head and tail are set to NULL.



* 1. If there is more than one element in the list then
     + Assign NULL to the prev pointer field of the second node.
     + Assign the address of the second node to head.
* Deallocate the memory occupied by the node pointed to by temp.

**Deleting from the End of the List**

An element from the end of the list can be deleted by performing the following steps:

* Assign the value of tail (address of the last element of the list) to a temporary variable (say temp)
* Further there are two cases:
  1. If there is only one element in the existing list, set both head and tail to NULL.
  2. If there is more than one element in the list then
     + Assign NULL to the next pointer field of the second last node.
     + Assign the address of the second last node to tail.
* Deallocate the memory occupied by the node pointed to by temp.

**Freeing up the Entire List**

The doubly linked list can be deleted either from the beginning or from the end. To delete from the beginning, use the following steps:

* Assign the head pointer to a temporary variable, say temp.
* Advance the head pointer to the next node.
* Deallocate the memory occupied by the node pointed to by temp.

Repeat the above steps until the entire list is deleted. Finally, set the tail pointer to NULL.

**Experiment No.6**

Title: Write a program to perform following operations on circular linked lists:

1. To insert a node at the beginning of the list.
2. To insert a node at the end of the list.
3. To insert a node after a given node of the list.

d. To delete a node at the beginning of the list.

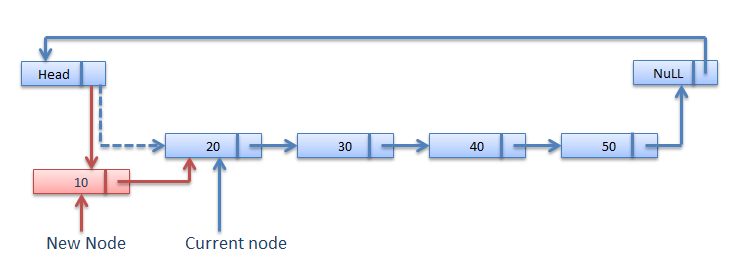
e. To delete a node at the end of the list.

f. To delete a node after a given node of the list

**Insertion At Beginning in Circular linked list**

Procedure for insertion a node at the beginning of list

Step1. Create the new node  
Step2. Set the new node’s next to itself (circular!)   
Step3. If the list is empty,return new node.  
Step4. Set our new node’s next to the front.  
Step5. Set tail’s next to our new node.   
Step6. Return the end of the list.



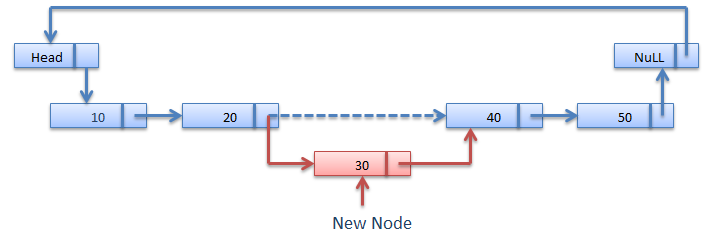
**Algorithm for Insertion at the front of Circular linked list**

1. node\* AddFront(node\* tail, int num){
2. node \*temp = (node\*)malloc(sizeof(node));
3. temp->data = num;
4. temp->next = temp;
5. if (tail == NULL)
6. return temp;
7. temp->next = tail->next;
8. tail->next = temp;
9. return tail;
10. }

**Insertion At Location in Circular linked list**

C function for insertion at given Location

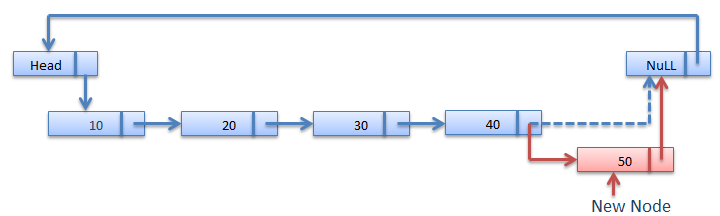
1. void insert\_location(struct link \*node)
2. {
3. int node\_no=1,insert\_no,flag=0,count;
4. node=start->next;
5. ptr=start;
6. count=i;
7. printf("\n Enter position where you want to insert new node:-");
8. scanf("%d",&insert\_no);
10. while(count)
11. {
12. if(node\_no==insert\_no)
13. {
14. new1=(struct link \*)malloc(sizeof(struct link));
15. printf("\n Insert data for new node:-");
16. scanf("%d",&new1->data);
17. ptr->next=new1;
18. new1->next=node;
19. flag=1;
20. break;
21. }
22. else
23. {
24. ptr=ptr->next;
25. node=node->next;
26. }
27. node\_no++;
28. count--;
29. }
30. if(flag==0)
31. {
32. printf("\n Position not found");
33. }
34. else
35. {
36. i++;
37. }
38. }



**Insertion At Last in Circular linked list**

Procedure for insertion a node at the Last of list

Step1. Create the new node  
Step2. Set the new node’s next to itself (circular!)   
Step3. If the list is empty,return new node.  
Step4. Set our new node’s next to the front.  
Step5. Set tail’s next to our new node.   
Step6. Return the end of the list.



**Algorithm for Insertion at the last of Circular linked list**

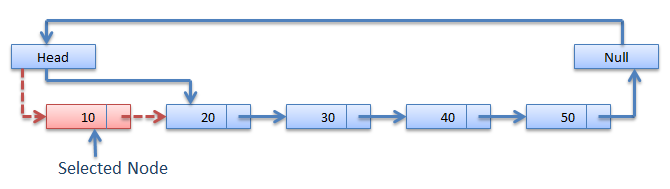
1. node\* AddEnd(node\* tail, int num){
2. node \*temp = (node\*)malloc(sizeof(node));
3. temp->data = num;
4. temp->next = temp;
5. if (tail == NULL)
6. return temp;
7. temp->next = tail->next;
8. tail->next = temp;
9. return temp;}

**Deletion In Circular linked list**

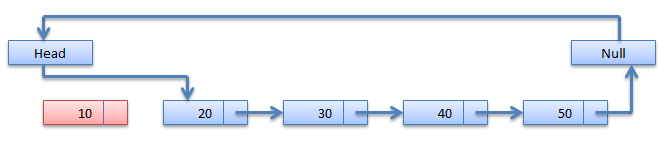
There are three situation for Deleting element in list.  
**1.**Deletion at beginning of the Circular linked list.  
**2.**Deletion at the middle of the Circular linked list.  
**3.**Deletion at the end of the Circular linked list.

**Deletion Procedure**

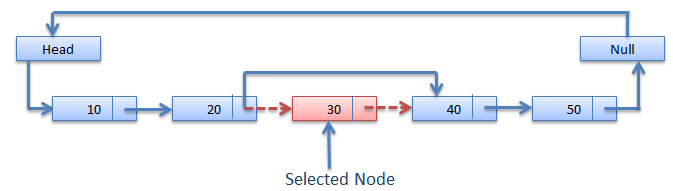
Deletion at beginning of the Circular linked list

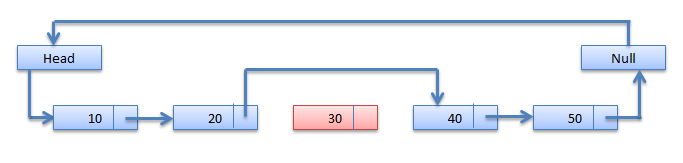


After Deletion

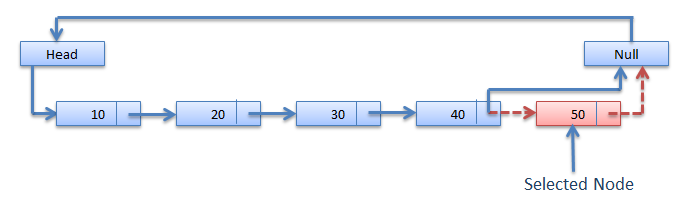


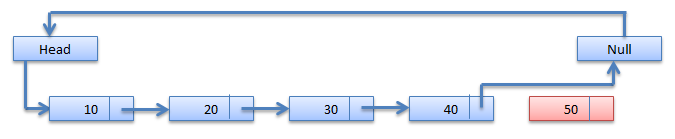
Deletion at the middle of the Circular linked list

  
After Deletion



Deletion at the end of the Circular linked list

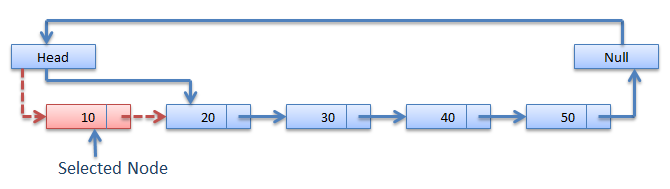
  
After Deletion

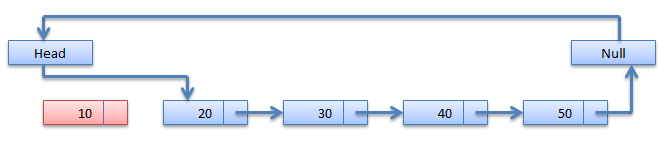


**Deletion at beginning in the Circular linked list**

**C function for Deletion at beginning in Circular linked list**

1. void delete\_first(struct link \*node)
2. {
3. node=start->next;
4. ptr=start;
5. if(i==0)
6. {
7. printf("\n List is empty");
8. exit(0);
9. }
10. ptr->next=node->next;
11. free(node);
12. i--;
13. }



****

**Deletion at given location in the Circular linked list**

C function for Deletion at location in Circular linked list

void delete\_Location(struct link \*node)

{

int node\_no=1,delete\_no,flag=0,count;

node=start->next;

ptr=start;

count=i;

printf("\n Enter position where you want to delete node:");

scanf("%d",&delete\_no);

while(count)

{

if(node\_no==delete\_no)

{

ptr->next=node->next;

free(node);

flag=1;

break;

}

else

{

ptr=ptr->next;

node=node->next;

}

node\_no++;

count--;

}

if(flag==0)

{

printf("\n Position not found");

}

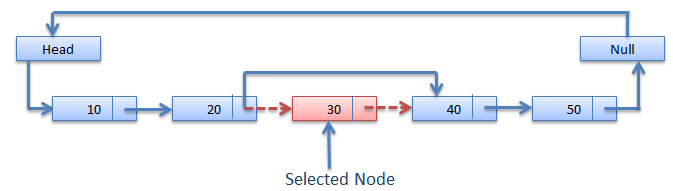
else

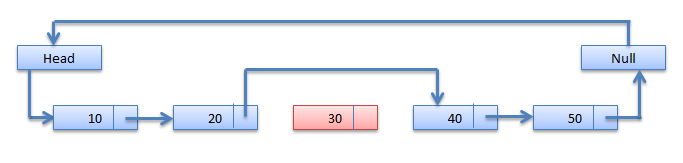
{

i--;

}

}



****

**Experiment No: 7**

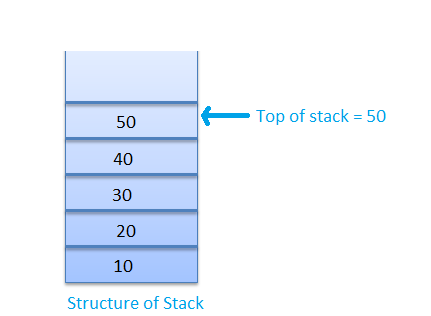
**Title:** Write a menu-based program to implement stack operations: PUSH, POP using array implementation of stack.

Stack

Stack is abstract data type and linear data structure.

Concept of Stack

A Stack is data structure in which addition of new element or deletion of existing element always takes place at a same end. This end is known as the top of the stack. That means that it is possible to remove elements from a stack in reverse order from the insertion of elements into the stack.  
  
One other way of describing the stack is as a last in, first out (LIFO) abstract data type and linear data structure. 



**Operations on Stack**

The stack is basically performed two operations PUSH and POP. Push and pop are the operations that are provided for insertion of an element into the stack and the removal of an element from the stack, respectively.  
PUSH:- PUSH operation performed for the adding item to the stack.  
POP:- POP operation performed for removing an item from a stack.

**Push Fuction**

1. void push()
2. {
3. int n;
4. printf("\n Enter item in stack");
5. scanf("%d",&n);
6. if(top==size-1)
7. {
8. printf("\nStack is Full");
9. }
10. else
11. {
12. top=top+1;
13. stack[top]=n;
14. }
15. }

**Pop Fuction**

1. void pop()
2. {
3. int item;
4. if(top==-1)
5. {
6. printf("\n Stack is empty");
7. }
8. else
9. {
10. item=stack[top];
11. printf("\n item popped is=%d", item);
12. top--;
13. }
14. }

**Experiment No: 8**

**Title:**Write a menu-based program using functions to implement stack operations: PUSH, POP using linked implementation of stack.

**Push Fuction**

void push()

{

int n;

struct node \*nw;

nw=(struct node\*)malloc(sizeof(struct node));

printf("\n Enter item in stack:");

scanf("%d",&n);

nw->item=n;

nw->next=0;

if(top==0)

{

top=nw;

}

else

{

nw->next=top;

top=nw;

}

}

**Pop Fuction**

void pop()

{

int item;

struct node \*ptr;

if(top==0)

{

printf("\n Stack is empty");

}

else

{

item=top->item;

ptr=top;

printf("\n item popped is=%d", item);

top=top->next;

free(ptr);

}

}

**Experiment No: 9**

**Title:** Write a program to convert infix expression into postfix expression and then to evaluate resultant postfix expression.

**Application of Stack**

1. Expression Evolution
2. Expression conversion
   1. [Infix to Postfix](http://scanftree.com/Data_Structure/infix-to-postfix)
   2. [Infix to Prefix](http://scanftree.com/Data_Structure/infix-to-prefix)
   3. [Postfix to Infix](http://scanftree.com/Data_Structure/postfix-to-infix)
   4. [Prefix to Infix](http://scanftree.com/Data_Structure/prefix-to-infix)
3. Parsing
4. Simulation of recursion
5. Fuction call

Expression Representation

|  |  |  |
| --- | --- | --- |
| **Infix** | **Prefix** | **Postfix** |
| a + b | + a b | a b + |
| a + b \* c | + a \* b c | a b c \* + |
| (a + b) \* (c - d) | \* + a b - c d | a b + c d - \* |

**Infix to Postfix Conversion**

Procedure for Postfix Conversion

|  |  |
| --- | --- |
| **1.** | Scan the Infix string from left to right. |
| **2.** | Initialize an empty stack. |
| **3.** | If the scanned character is an operand, add it to the Postfix string. |
| **4.** | If the scanned character is an operator and if the stack is empty push the character to stack. |
| **5.** | If the scanned character is an Operator and the stack is not empty, compare the precedence of the character with the element on top of the stack. |
| **6.** | If top Stack has higher precedence over the scanned character pop the stack else push the scanned character to stack. Repeat this step until the stack is not empty and top Stack has precedence over the character. |
| **7.** | Repeat 4 and 5 steps till all the characters are scanned. |
| **8.** | After all characters are scanned, we have to add any character that the stack may have to the Postfix string. |
| **9.** | If stack is not empty add top Stack to Postfix string and Pop the stack. |
| **10.** | Repeat this step as long as stack is not empty. |

**Algorithm for Postfix Conversion**

1. S:stack
2. while(more tokens)
3. x<=next token
4. if(x == operand)
5. print x
6. else
7. while(precedence(x)<=precedence(top(s)))
8. print(pop(s))
9. push(s,x)
10. while(! empty (s))
11. print(pop(s))

**Experiment No:10**

**Title:** Write a program to print Fibonacci series using Recursion.

**Recursion**

Recursion is a programming technique that allows the programmer to express operations in terms of themselves. Simply we can say that when a 'c' function calls itself.  
In other words recursion is thus the process of defining something in terms of itself.

**Recursive functions**

1. void recurse()
2. {
3. recurse(); /\* Function calls itself \*/
4. }
5. void main()
6. {
7. recurse(); /\* Sets off the recursion \*/
8. getch();
9. }

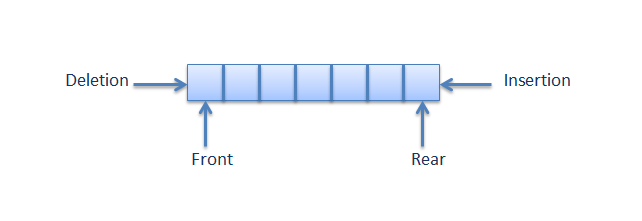
**Experiment No: 11**

**Title:** Write a menu-based program to implement linear queue operations: INSERTION, DELETION using array implementation of queue.

**Queues**

Queues is a kind of abstract data type where items are inserted one end (rear end) known as **enqueue** operation and deteted from the other end(front end) known as**dequeue** operation.

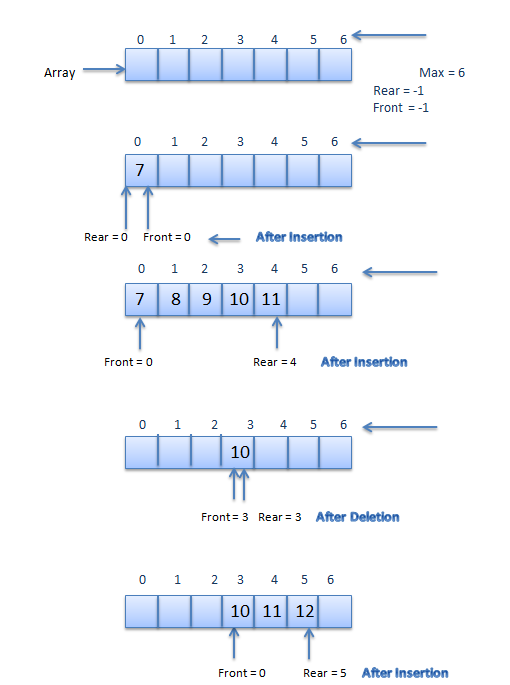
This makes the queue a **First-In-First-Out (FIFO)** data structure. The queue performs the function of a buffer.



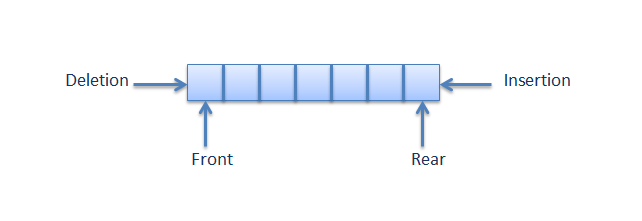
**Operation on Queues**

|  |  |
| --- | --- |
| **Operation** | **Description** |
| initialize() | Initializes a queue by adding the value of rear and font to -1. |
| enqueue() | Insert an element at the rear end of the queue. |
| dequeue() | Deletes the front element and return the same. |
| empty() | It returns true(1) if the queue is empty and return false(0) if the queue is not empty. |
| full() | It returns true(1) if the queue is full and return false(0) if the queue is not full. |

**Operation on queue**



Implementation of Queues

1. Implementation using Array (**Static Queue**)  
2. Implementation using Linked List (**Dynamic Queue**)  


**Structure of Queue**

1. #define max 5
2. struct queue
3. {
4. int data[max];
5. int front, rear ;
6. }

**Algorithm for Insertion in Queue**

1. Insert ( )
2. If (REAR == N) Then [Check for overflow]
3. Print: Overflow
4. Else
5. If (FRONT and REAR == 0) Then [Check if QUEUE is empty]
6. (a) Set FRONT = 1
7. (b) Set REAR = 1
8. Else
9. Set REAR = REAR + 1 [Increment REAR by 1]
10. [End of Step 4 If]
11. QUEUE[REAR] = ITEM
12. Print: ITEM inserted
13. [End of Step 1 If]
14. Exit

**Algorithm for Deletion in Queue**

1. Delete ( )
2. If (FRONT == 0) Then [Check for underflow]
3. Print: Underflow
4. Else
5. ITEM = QUEUE[FRONT]
6. If (FRONT == REAR) Then [Check if only one element is left]
7. (a) Set FRONT = 0
8. (b) Set REAR = 0
9. Else
10. Set FRONT = FRONT + 1 [Increment FRONT by 1]
11. [End of Step 5 If]
12. Print: ITEM deleted
13. [End of Step 1 If]
14. Exit

**Experiment No: 12**

**Title:** Write a menu-based program to implement linear queue operations: INSERTION, DELETION using linked list implementation of queue.

**Dynamic Queue**

Structure of Dynamic Queue.

1. struct link
2. {
3. int info;
4. struct link \*next;
5. }\*front,\*rear;

**void insert\_q(int no)**

{

  struct link \*new1;

  new1=(struct link\*)malloc(sizeof(struct link));

  new1->info=no;

  new1->next=NULL;

  if(rear==NULL||front==NULL)

  {

  front=new1;

  }

  else

  {

  rear->next=new1;

  }

  rear=new1;

 }

**int delete\_q()**

**{**

  struct link \*t;

  int no;

if(front==NULL||rear==NULL)

{

printf("\n queue is Under Flow");

getch();

return;

}

else

{

t=front;

no=t->info;

front=front->next;

free(t);

}

return(no);

}

**void display()**

{

struct link \*t;

t=front;

if(front==NULL||rear==NULL)

{

printf("\nQueue is empty");

getch();

exit(0);

}

while(t!=NULL)

{

printf("\n %d",t->info);

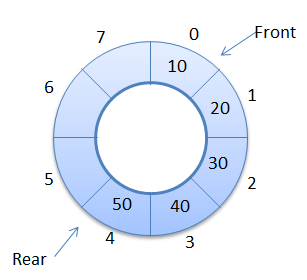
t=t->next;

}

**Experiment No: 13**

**Title:**Write a menu-based program to implement circular queue operations: INSERTION, DELETION.

**Circular Queue**

A circular queue is an abstract data type that contains a collection of data which allows addition of data at the end of the queue and removal of data at the beginning of the queue. Circular queues have a fixed size. Circular queue follows FIFO principle. Queue items are added at the rear end and the items are deleted at front end of the circular queue.  


**Algorithm for Insertion in a circular queue**

1. Insert CircularQueue ( )
2. If (FRONT == 1 and REAR == N) or (FRONT == REAR + 1) Then
3. Print: Overflow
4. Else
5. If (REAR == 0) Then [Check if QUEUE is empty]
6. (a) Set FRONT = 1
7. (b) Set REAR = 1
8. Else If (REAR == N) Then [If REAR reaches end if QUEUE]
9. Set REAR = 1
10. Else
11. Set REAR = REAR + 1 [Increment REAR by 1]
12. [End of Step 4 If]
13. Set QUEUE[REAR] = ITEM
14. Print: ITEM inserted
15. [End of Step 1 If]
16. Exit

**Algorithm for Deletion in a circular queue**

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

**Experiment No: 14**

**Title:**Write a program to traverse a binary tree using PRE-ORDER, IN-ORDER, POST-ORDER traversal techniques.

**Algorithm for Pre-order Traversal**

An algorithm for preorder traversal (V-L-R) is given below. It is provided with a pointer called Tree that points to the root of the binary tree.

Algorithm preorderTravel(Tree) {

if (Tree == NULL) return

else {

process DATA(Tree);

preorderTravel (leftChild (Tree));

preorderTravel (rightChild (Tree));

} }

**Algorithm for In-order Traversal**

An algorithm for preorder traversal (L-V-R) is given below. It is provided with a pointer called Tree that points to the root of the binary tree.

Algorithm inorderTravel(Tree) {

if (Tree == NULL) return

else {

inorderTravel (leftChild (Tree));

process DATA(Tree);

inorderTravel (rightChild (Tree));

} }

**Algorithm for Post-order Traversal**

An algorithm for postorder traversal (L-R-V) is given below. It is provided with a pointer called Tree that points to the root of the binary tree.

Algorithm postorderTravel(Tree) {

if (Tree == NULL) return

else {

postorderTravel (leftChild (Tree));

postorderTravel (rightChild (Tree));

process DATA(Tree);

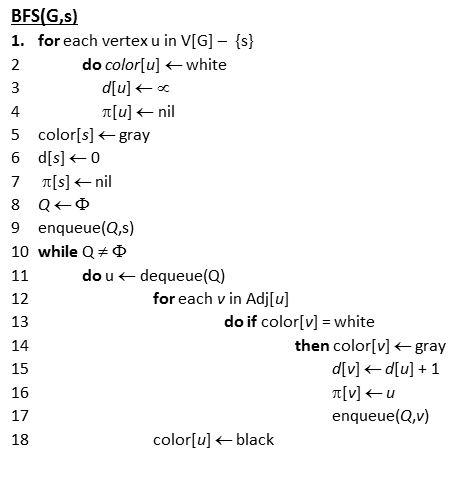
}

}

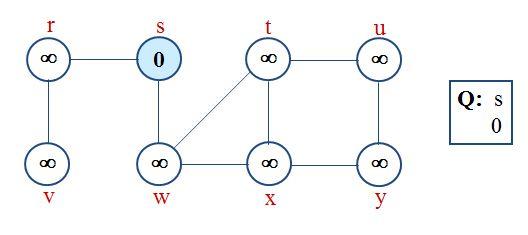
**Experiment No: 15**

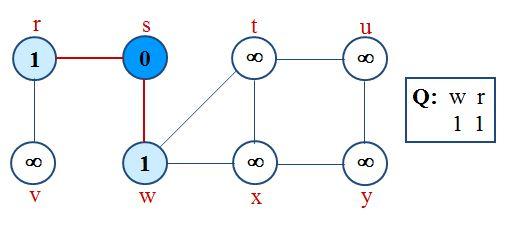
**Title:** Write a program to traverse a graph using breadth-first search (BFS), depth-first search (DFS).

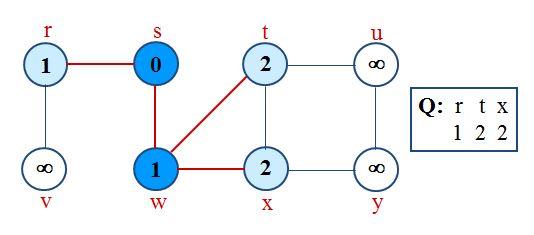
**Breadth-first search**

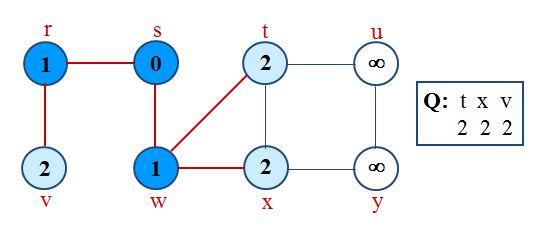


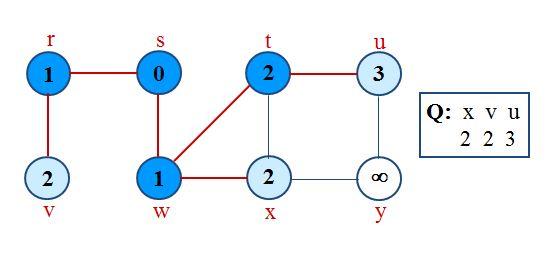
* Q: a queue of discovered vertices
* color[v]: color of v
* d[v]: distance from s to v
* π[u]: predecessor of v
* white: undiscovered
* gray: discovered
* black: finished

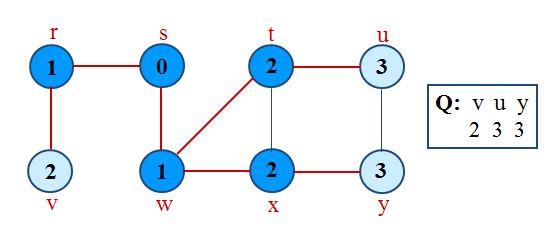


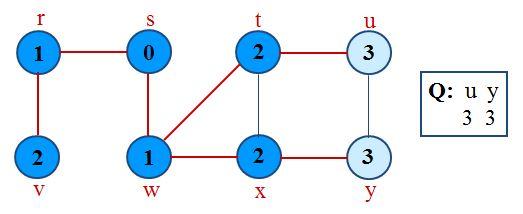


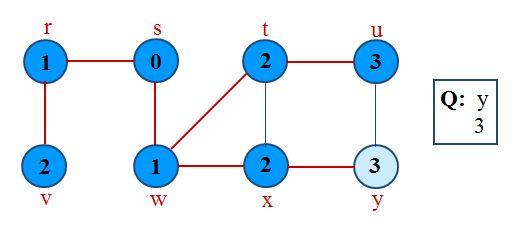


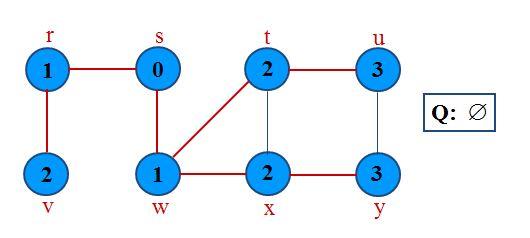


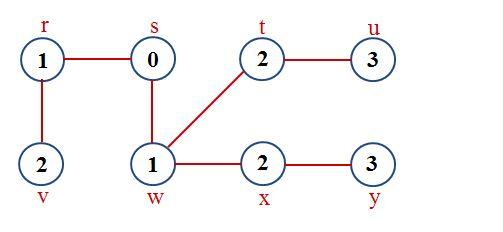












**Analysis of BFS**

Initialization takes O(V).  
Traversal Loop After initialization, each vertex is enqueued and dequeued at most once, and each operation takes O(1). So, total time for queuing is O(V).  
The adjacency list of each vertex is scanned at most once. The sum of lengths of all adjacency lists is ?(E).  
Summing up over all vertices => total running time of BFS is O(V+E), linear in the size of the adjacency list representation of graph.

**Experiment No: 19**

**Title:** Write a program to sort given set of numbers in ascending/descending order using insertion sort and also search a number using binary search.

**Insertion Sort**

Insertion sort is a simple sorting algorithm that builds the final sorted array one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort.

**Algorithm**

1. INSERTION-SORT(A)
2. for j = 2 to A.length
3. key = A[j]
4. // Insert A[j] into the sorted sequence A[1...j-1]
5. i = j - 1
6. while i > 0 and A[i] > key
7. A[i+1] = A[i]
8. i = i - 1
9. A[i+1] = key

**Experiment No: 20**

**Title:** Write a program to sort given set of numbers in ascending/descending order using Quick sort and selection sort. Also record the time taken by these two programs and compare them.

**Quick sort**

Quicksort is a simple sorting algorithm using the divide-and-conquer recursive procedure.It is the quickest comparison-based sorting algorithm in practice with an average running time of O(n log(n)).It is also known as partition-exchange sort.

**Algorithm**

1. QUICKSORT(A, p, r)
2. if p < r
3. q = PARTITION(A, p, r)
4. QUICKSORT(A, p, q - 1)
5. QUICKSORT(A, q + 1, r)
6. PARTITION(A, p, r)
7. x = A[r]
8. i = p - 1
9. for j = p to r - 1
10. if A[j] <= x
11. i = i + 1
12. exchange A[i] with A[j]
13. exchange A[i + 1] with A[r]
14. return i + 1

**Experiment No: 21**

**Title:** Write a program to sort given set of numbers in ascending/descending order using Merge sort.

**Merge sort**

Merge sort is comparison-based sorting algorithm. Merge sort is a stable sort, which means that the implementation preserves the input order of equal elements in the sorted output.

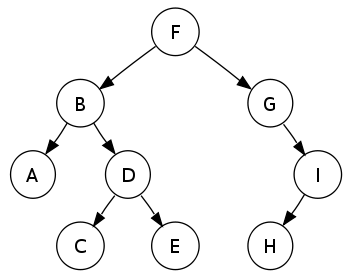
**Algorithm**

1. MERGE (A,p,q,r)
2. n1 = q + p + 1
3. n2 = r + q
4. let L[1. . n1 + 1] and R[1 . . n2 + 1] be new arrays
5. for i = 1 to n1
6. L[i] = A[p + i + 1]
7. for j = 1 to n2
8. R[j ] = A[q + j]
9. L[n1 + 1] = 9999
10. R[n2 + 1] = 9999
11. i = 1
12. j = 1
13. for k = p to r
14. if L[i] <= R[j ]
15. A[k] = L[i]
16. i = i + 1
17. else A[k] = R[j]
18. j = j + 1

**Experiment No: 22**

**Title:** Write a menu-based program to perform operations for a binary search tree (BST).

1. Search an element
2. Find minimum
3. Find maximum
4. Insertion
5. Deletion



|  |  |
| --- | --- |
| 1  2  3  4  5  6 | Preorder traversal sequence : F, B, A, D, C, E, G, I, H     (root, left, right)  Inorder traversal sequence  : A, B, C, D, E, F, G, H, I     (left, root, right)  Postorder traversal sequence: A, C, E, D, B, H, I, G, F     (left, right, root) |

# node \*get\_node() {

# node \*temp;

# temp = (node \*) malloc(sizeof(node));

# temp->lchild = NULL;

# temp->rchild = NULL;

# return temp;

# }

/\*

This function is for creating a binary search tree

\*/

void insert(node \*root, node \*new\_node) {

   if (new\_node->data < root->data) {

      if (root->lchild == NULL)

         root->lchild = new\_node;

      else

         insert(root->lchild, new\_node);

   }

   if (new\_node->data > root->data) {

      if (root->rchild == NULL)

         root->rchild = new\_node;

      else

         insert(root->rchild, new\_node);

   }

}

/\*

This function is for searching the node from

binary Search Tree

\*/

node \*search(node \*root, int key, node \*\*parent) {

   node \*temp;

   temp = root;

   while (temp != NULL) {

      if (temp->data == key) {

         printf("\nThe %d Element is Present", temp->data);

         return temp;

      }

      \*parent = temp;

      if (temp->data > key)

         temp = temp->lchild;

      else

         temp = temp->rchild;

   }

   return NULL;

}

/\*

This function displays the tree in inorder fashion

\*/

void inorder(node \*temp) {

   if (temp != NULL) {

      inorder(temp->lchild);

      printf("%d", temp->data);

      inorder(temp->rchild);

   }

}

/\*

This function displays the tree in preorder fashion

\*/

void preorder(node \*temp) {

   if (temp != NULL) {

      printf("%d", temp->data);

      preorder(temp->lchild);

      preorder(temp->rchild);

   }

}

/\*

This function displays the tree in postorder fashion

\*/

void postorder(node \*temp) {

   if (temp != NULL) {

      postorder(temp->lchild);

      postorder(temp->rchild);

      printf("%d", temp->data);

   }

}

# Explanation :

# get\_node() function will allocate memory dynamicallyand allocate one node.

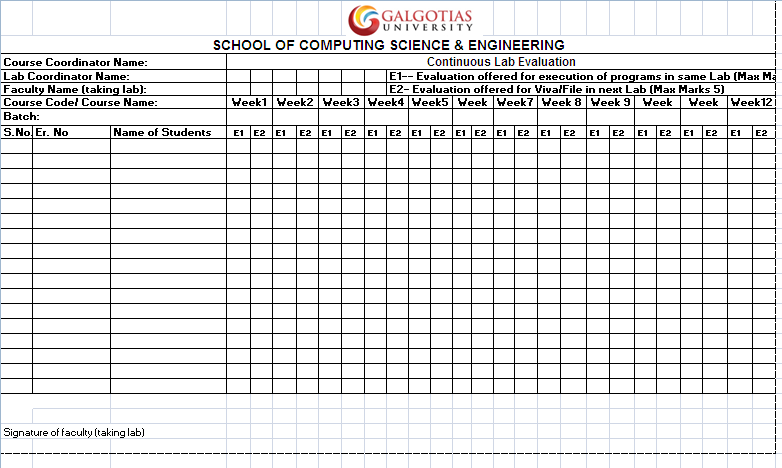
# if below condition is satisfied then we can say that we are going to create first node of the tree. (i.e Tree is empty and this created node is very first node)

|  |
| --- |
| if(root == NULL) |

# If condition does not satisfied then we can say that we have already node in a tree. (i.e this node which we have created is not a first node)

# GUIDELINES FOR CONTINUOUS ASSESSMENT





|  |
| --- |
| Internal Lab Assessment (End Semester) |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **S. No.** | **Enrol. No.** | **Name of the Student** | **Continuous assessment (30)** | **Lab Experiment & Demo (10)** | **Viva-Voce (10)** | **Total (50)** | **Marks (in words)** |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |