**SUBJECT:- DAA LAB**

**LAB NO: 1**

**REVIEW OF FUNDAMENTAL DATA STRUCTURES**

**Objectives:**

In this lab, student will be able to:

* recall the concepts learnt in Data Structures Lab
* implement basic data structures

**Description :** **Data Structures** specify the structure of data storage in a program. Various data structures namely **arrays, stacks, queues, linked lists, trees** are used for storing data. Each data structure is different from the other in its fundamental way of storing. In **arrays**, a contiguous piece of memory is allocated for storing data. **Static** and **Dynamic arrays** are two types of array which differ by the instance at which memory is allocated. In **static arrays**, memory is allocated at compile time of the program whereas in **Dynamic arrays** memory is allocated at run time. **Dynamic arrays** overcomes the problem of unnecessary wastage of memory space. **Stack** is a data structure in which the insertion to the **stack**(called as push) and removal from the **stack**(called as pop) operations are performed in Last In First Out(LIFO) order. LIFO specifies that the last item to be pushed is the first one to be popped. **Queue** is a data structure in which the insertion to the queue(enque) and removal of element from the queue(Dequeue) happen in the same order. This means it follows First In First Out(FIFO) order. Linked lists store data in non-linea manner. A node of a **linked list** is created at run time and is used to store data element. Nodes of a **linked list** will be allocated memory at run time and the nodes can be anywhere in memory. **Singly linked** list and **doubly linked** lists are two broad types of **linked lists**. **Single linked list** has a single pointer to the next node whereas doubly linked list has two pointers one to the left node and other to the right node. A special value NULL will be used to denote the non-existence of node. **Trees** are very useful in specific storage requirements of graphs, dictionaries etc. **Binary tree** is a special form of trees in which every node can have maximum two children.

1. **SOLVED EXERCISE:**

1) Write an algorithm and program to implement a doubly linked list which supports

the following operations

i. Create the list by adding each node at the front

ii. Insert a new node to the left of the node whose key value is read as an input.

iii. Delete all occurrences of a given key, if it is found. Otherwise, display

appropriate message

iv. Search a node based on its key value

v. Display the contents of the list

|  |
| --- |
| **Description :** Doubly linked list is a data structure in which the data elements are stored in nodes and the nodes are connected by two links. Out of two links one link points to the neighboring node in the left direction and the other link to the node in the right direction. Addresses of nodes will be used to represent the node. A special value NULL is used to represent the absence of a node. Creating the doubly linked list, insertion of an element to the left/right of any node, deletion of all nodes with specific node content and displaying all nodes are the operations commonly performed on a doubly linked list. Each of the operations consumes certain amount of time and memory. Hence they differ in time and space efficiency. |

|  |
| --- |
| **Algorithm :** Doubly Linked list  Define a structure to hold list node  Define two links within the node one for left link and the other for rlink  **CreateList(int val)**  begin  if head == NULL then  node = allocate memory for a Node  node ->llink = node->rlink = NULL  node->val = val  head=node  else  print “List is already created …”  end if  end  **insertIntoList(int before, int val)**  begin  node=head  while node->val != before  node = node->rlink  end while  if node != NULL then  newNode = allocate memory for a node  newNode->val = val  if node->llink != NULL then  node->llink->rlink = newNode  newNode->llink = node->llink  newNode->rlink = node  node->llink = newNode  else  newNode->rlink = node  node->llink = newNode  head = newNode  end if  else  print “Unable to insert, node with value “ val “not found”  return  end if  end  **deleteAll(int delVal)**  begin  node = head  while node != NULL  if node->val == delVal  if node->llink != NULL then  node -> llink ->rlink = node -> rlink  if node->rlink != NULL then  node->rlink->llink = node->llink  node = node->rlink  else  node->llink->rlink = NULL  node=NULL  end if  else  if node->rlink != NULL then  node ->rlink->llink = NULL  head = node->rlink  node = head  release memory for node  else  head = node = NULL  release memory for node  end if  end if  else  node = node->rlink  end if  end while  end  **searchNode(int searchVal)**  begin  node=head  while node != NULL  do  if node->val == searchVal then  print “Node is found with key “, searchVal  end if  node = node->rlink  end do  end  **displayAll()**  begin  node = head  while node != NULL  do  print “Node with val “, node->val  node = node->rlink  end do  end |

|  |
| --- |
| **Time Complexity :**  For creating list θ(1) is the time complexity    For Insertion, Search, Delete, Display All operations the complexity is θ(n) where n is the number of nodes |

|  |
| --- |
| **Program**  #include<stdio.h>  #include<stdlib.h>  struct node {  int val;  struct node \*llink,\*rlink;  };  typedef struct node \*NODE;  NODE head=NULL;  void CreateList(int val)  {  NODE nd;  if (head == NULL) {  nd = (NODE) malloc(sizeof(struct node));  nd->llink = nd->rlink = NULL;  nd->val = val;  head=nd;  }  else {  printf("List is already created ….\n");  }  }  void insertIntoList(int before, int val)  {  NODE nd, newnd;  nd=head;  while (nd != NULL && nd->val != before)  nd = nd->rlink;  if (nd != NULL) {  newnd = (NODE)malloc(sizeof(struct node));  newnd->llink = newnd->rlink = NULL;  newnd->val = val;  if (nd->llink != NULL) {  nd->llink->rlink = newnd;  newnd->llink = nd->llink;  newnd->rlink = nd;  nd->llink = newnd;  }  else {  newnd->rlink = nd;  nd->llink = newnd;  head=newnd;  }  }  else  printf( "Unable to insert, node with value %d not found", val);  }  void deleteAll(int delVal)  {  NODE nd,nxtNode;  nd = head;  while (nd != NULL) {  if (nd->val == delVal) {  if (nd->llink != NULL) {  nd-> llink ->rlink = nd -> rlink;  if (nd->rlink != NULL) {  nd->rlink->llink = nd->llink;  nxtNode = nd->rlink;  free(nd);  nd = nxtNode;  }  else {  nd->llink->rlink = NULL;  free(nd);  nd=NULL;  }    }  else {  if (nd->rlink != NULL) {  nd ->rlink->llink = NULL;  head = nd->rlink;  free(nd);  nd = head;    }  else {  free(nd);  head = nd = NULL;  }  }  }  else  nd = nd->rlink;  }  }  void searchNode(int searchVal) {  NODE nd;  nd=head;  while (nd != NULL) {  if (nd->val == searchVal)  printf( "Node is found with key %d\n", searchVal);  nd = nd->rlink;  }  }  void displayAll()  {  NODE nd;  nd = head;  while (nd != NULL) {  printf("Node with val %d\n", nd->val);  nd = nd->rlink;  }  }  void main() {  int choice, val,before;  do {  printf("1. Create List\n");  printf("2. Insert into List\n");  printf("3. Delete all by value\n");  printf("4. Search by value\n");  printf("5. Display all\n");  printf("6. Exit\n");  printf("Enter your choice :");  scanf("%d", &choice);  switch(choice) {  case 1: printf("Please enter the node value");  scanf("%d", &val);  CreateList(val);  break;  case 2: printf("Please enter the node value to insert ");  scanf("%d", &val);  printf("Please enter the node value before which new node has to be inserted ");  scanf("%d", &before);  insertIntoList(before, val);  break;  case 3:printf("Enter the node value to be deleted ");  scanf("%d", &val);  deleteAll(val);  break;  case 4:printf("Enter the node value to be searched ");  scanf("%d", &val);  searchNode(val);  break;  case 5:displayAll();  break;  case 6:  break;  default :printf("Invalid choice ");  break;  }  }while(choice != 6);  } |

|  |
| --- |
| **Sample Input and Output :**  1. Create List  2. Insert into List  3. Delete all by value  4. Search by value  5. Display all  6. Exit  Enter your choice :1  Please enter the node value5  1. Create List  2. Insert into List  3. Delete all by value  4. Search by value  5. Display all  6. Exit  Enter your choice :2  Please enter the node value to insert 3  Please enter the node value before which new node has to be inserted 5  1. Create List  2. Insert into List  3. Delete all by value  4. Search by value  5. Display all  6. Exit  Enter your choice :3  Enter the node value to be deleted 3  1. Create List  2. Insert into List  3. Delete all by value  4. Search by value  5. Display all  6. Exit  Enter your choice :4  Enter the node value to be searched 5  Node is found with key 5  1. Create List  2. Insert into List  3. Delete all by value  4. Search by value  5. Display all  6. Exit  Enter your choice :5  Node with val 5  1. Create List  2. Insert into List  3. Delete all by value  4. Search by value  5. Display all  6. Exit  Enter your choice :6 |

1. **LAB EXERCISES**

1). Write a program to construct a binary tree to support the following operations.

Assume no duplicate elements while constructing the tree.

i.  Given a key, perform a search in the binary tree. If the key is found then

display “key found” else inserts the key in the BST.

ii. Display the tree using inorder, preorder and post order traversal methods

2). Write a program to implement the following graph representations.

i. Adjacency list

ii. Adjacency matrix

1. **ADDITIONAL EXERCISES:**

1). Repeat problem given in solved exercise using singly linked list.

2). Write a program to implement Stack and Queue using circular doubly linked list.

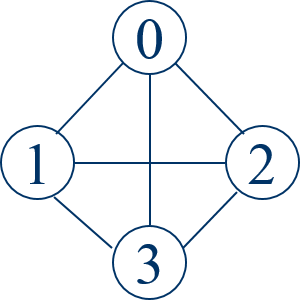
**Notes:-**

## Graph Representations

Graph is a data structure that consists of following two components:

1. A finite set of vertices also called as nodes.
2. A finite set of ordered pair of the form (u, v) called as edge. The pair is ordered because (u, v) is not same as (v, u) in case of a directed graph(di-graph). The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.

Following two are the most commonly used representations of a graph.

1. Adjacency Matrix
2. Adjacency List Adjacency Matrix:

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

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

**Adjacency List:**

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

