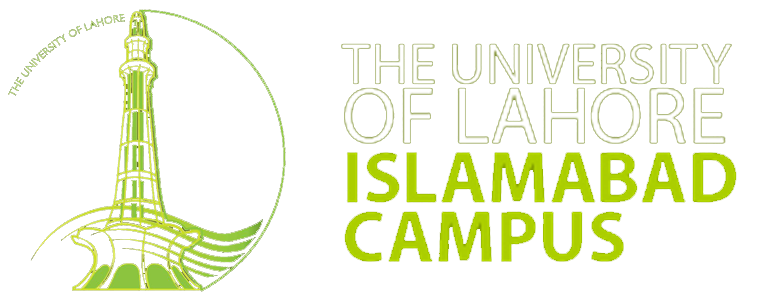
****

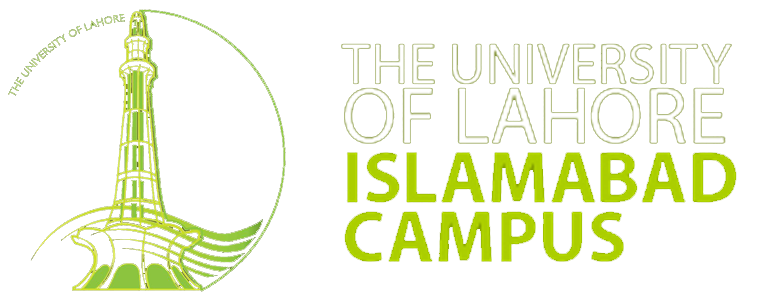
**School of Information Technology (SIT)**

**The University of Lahore, Islamabad Campus**

**Lab Manual**

**Data Structures and Algorithms**

**CS-2112**

****

**The University of Lahore, Islamabad Campus**

**School of Information Technology (SIT)**

**LIST OF EXPERIMENTS**

[**Experiment 01**](#_Toc150874301): Introduction to Arrays and its operation

[**Experiment**](#_Toc150874301) **02**: Queue with Array implementation

[**Experiment**](#_Toc150874301) **03**: Stack with Array implementation

[**Experiment**](#_Toc150874301) **04**: Link list-Basic Insertion and Traversal

[**Experiment**](#_Toc150874301) **05**: Link list-Basic Deletion at desired position

[**Experiment**](#_Toc150874301) **06**: Circular Link list

[**Experiment**](#_Toc150874301) **07**: Doubly Link list

[**Experiment**](#_Toc150874301) **08**: Basics of Graph Data Structure

[**Experiment**](#_Toc150874301) **09**: Directed and Undirected Graph Implementation

[**Experiment**](#_Toc150874301) **10**: Prim’s Algorithm

[**Experiment**](#_Toc150874301) **11 & 12**: In order, preorder and postorder Traversal

[**Experiment**](#_Toc150874301) **13 & 14**: AVL Tree Data Structure

Experiment # 01

**Introduction to Arrays and its operation**

**Objectives:-**

The objectives of this lab session are to understand the basic and various operations on arrays in C++.

**Software Tools:-**

1. **Code Blocks with GCC compiler**

**Theory:-**

We have already studied array in our computer programming course. We would be using the knowledge we learned there to implement different operation on arrays.

**Traversing Linear Arrays:-**

Let A be the collection of data elements stored in the memory of the computer. Suppose we want to print the contents of each element of A or suppose we want to count the number of elements of A with a given property. This can be accomplished by traversing A that is by accessing and Processing each element of A exactly once.

The following algorithm traverses a linear array. The simplicity of the algorithm comes from the fact that LA is a linear structure. Other linear structures such as linked list can also be easily traversed. On the other hand the traversal of non-linear structures such as trees and graphs is considerably more complicated.

**Algorithm:-**

(Traversing a Linear Array) Here LA is a linear Array with lower Bound LB and upper Bound UB. This algorithm traverses LA.

Applying an operation PROCESS to each element of LA.

*[Initialize Counter] Set X=LB.*

1. *Repeat Step 3 and 4 while K<=UB.*

*[Visit element] Apply PROCESS to LA[X].*

*[Increase Counter] Set X=X+1.*

*[End of Step 2 Loop]*

*5. Exit.*

**Inserting and Deleting:-**

Let A be a collection of data elements in the memory of computer. “Inserting” refers to the operation of adding another element to the collection A and “deleting” refers to the operation of removing one of the elements from A. Here we discuss the inserting and deleting when A is a linear array.

Inserting an element at the “end” of the linear array can be easily done provided the memory space allocated for the array is large enough to accommodate the additional element. On the other hand suppose we need to insert an element in the middle of the array. Then on average half of the elements must be moved downward to the new location to accommodate the new element and keep the order of other elements.

Similarly deleting the element at the “end” of an array presents no difficulties but deleting the element somewhere in the middle of the array would require that each subsequent element be moved one location upward in order to fill up the array.

**Algorithm of Insertion operation:-**

(Inserting into Linear Array) INSERT (LA, N, K, ITEM)

Here LA is a linear array with N elements and K is a positive integer such that K≤N. This algorithm inserts an element ITEM into the Kth position in LA.

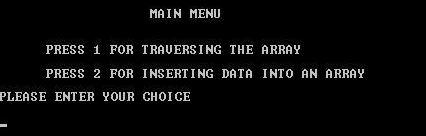
1. *[Initialize Counter] Set J=N.*
2. *Repeat Step 3 and 4 while J≥K.*
3. *[Move Jth element downward] Set LA [J+1] =LA[J].*
4. *[Decrease Counter] Set J=J-1.*

*End of Step 2 Loop.*

1. *[Insert element] Set LA[K]=ITEM.*
2. *[Reset N] Set N=N+1.*
3. *Exit.*

**Lab Task:-**

Write a C++ program to implement all the above described algorithms and display the following menu and ask the user for the desired operation.



The program should have the option for reusing it after you have completed the desired task.

## Conclusion

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

---------------------------------------------

**(Concerned Teacher/Lab Engineer)**

**Experiment # 02**

**Queue with Array implementation**

**Objective:-**

The objective of this session is to understand the various operations on queues using array structure in C++.

**Software Tools:-**

To achieve the goals of objectives, I use Code Blocks with GCC compiler.

**Theory:-**

**Queue using Array:-**

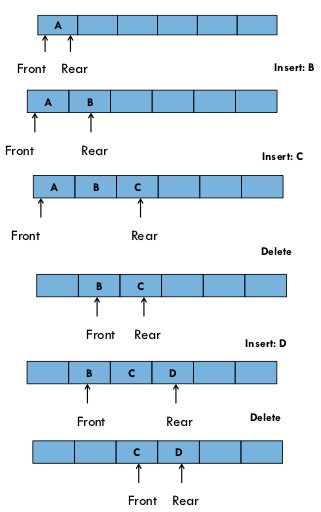
This manual discuss an important data structure, called a queue. The idea of a queue in computer science is the same as the idea of the queues to which you are accustomed in everyday life. There are queues of customers in a bank or in a grocery store and queues of cars waiting to pass through a tollbooth. Similarly, because a computer can send a print request faster than a printer can print, a queue of documents is often waiting to be printed at a printer. The general rule to process elements in a queue is that the customer at the front of the queue is served next and that when a new customer arrives, he or she stands at the end of the queue. That is, a queue is a First In First Out data structure.

A queue is a set of elements of the same type in which the elements are added at one end, called the back or rear, and deleted from the other end, called the front. For example, consider a line of customers in a bank, wherein the customers are waiting to withdraw/deposit money or to conduct some other business. Each new customer gets in the line at the rear. Whenever a teller is ready for a new customer, the customer at the front of the line is served.

The rear of the queue is accessed whenever a new element is added to the queue, and the front of the queue is accessed whenever an element is deleted from the queue. As in a stack, the middle elements of the queue are inaccessible, even if the queue elements are stored in an array.

Queue: A data structure in which the elements are added at one end, called the rear, and deleted from the other end, called the front; a First-In-First-Out (FIFO) data structure.

Queues may be represented in the computer in various ways, usually by means at one way list or linear arrays. Unless otherwise stated or implied each of our queues will be maintained by a linear array QUEUE and two pointer variable FRONT containing the location of the front element of the queue and REAR containing the location of the rear element of the queue. The condition FRONT = NULL will indicate that the queue is empty.



Whenever an element is deleted from the queue the value of FRONT is increased by one. This can be implemented by the assignment.

*FRONT = FRONT + 1*

Similarly whenever an element is added to the queue the value of REAR is increased by one. This can be implemented by the assignment.

*REAR = REAR + 1*

This means that after N insertions the rear element of the queue will occupy QUEUE [N] or in other words eventually the queue will occupy the last part of the array. This occurs even though the queue itself may not contain many elements.

Suppose we want to insert an element ITEM into a queue at the time the queue does occupy the last part of the array i.e. when REAR = N. One way is to do this simply move the entire queue to the beginning of the array changing FRONT and REAR accordingly, and then inserting ITEM as above. This procedure may by very expensive. The procedure we adopt is to assume that the array QUEUE is circular that is that QUEUE [1] comes after QUEUE [N] in the array. With this assumption we insert ITEM into the queue by assigning ITEM to QUEUE [1]. Specifically instead of increasing REAR to N+1 we reset REAR=1 and then assign

*QUEUE[REAR] = ITEM*

Similarly if FRONT=N and an element is deleted then we reset FRONT=1 instead of increasing

*FRONT to N+1.*

Suppose that our queue only contains one element i.e. suppose that

*FRONT = REAR ≠ NULL*

And suppose that the element is deleted. Then we assign *FRONT = NULL* and *REAR = NULL* to indicate that the queue is empty.

**Algorithm for insertion into the Queue:-**

**QINSERT(QUEUE, N ,FRONT,REAR, ITEM)**

**This procedure inserts an element ITEM into a queue.**

**1. [Queue already filled?]**

**If FRONT = 1 and REAR = N or if FRONT =REAR+1 then**

**Write OVERFLOW and Return.**

**2. [Find new value of REAR]**

**If FRONT = NULL then [Queue initially empty] Set FRONT =1 and REAR = 1.**

**else if REAR = N then**

**Set REAR = 1.**

**else**

**Set REAR = REAR + 1. [End of if Structure]**

**3. Set QUEUE[REAR] = ITEM [This inserts new element]**

**4. Return.**

**Algorithm for Deletion from Queues:-**

**QDELETE(QUEUE , N ,FRONT REAR, ITEM)**

**This procedure deletes an element from a queue and assigns it to the variable ITEM.**

**1. [Queue already empty?]**

**If FRONT = NULL then Write Underflow and Return.**

**2. Set ITEM = QUEUE[FRONT]**

**3. [Find new value of FRONT]**

**If FRONT = REAR then [Queue has only one element to start]**

**Set FRONT = NULL and REAR = NULL**

**else if FRONT = N then**

**Set FRONT = 1**

**else**

**Set FRONT = FRONT + 1 [End of If Structure]**

**4. Return.**

**Lab Task:-**

**Write a C++ code to perform insertion and deletion in queue using arrays applying the algorithms given in the manual. Create a menu shown below.**



## Conclusion

---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

----------------------------------------------

**(Concerned Teacher/Lab Engineer)**

**Experiment # 03**

**Stack with Array implementation**

**Objective:-**

The objective of this session is to understand the various operations on stack using arrays structure in C++.

**Software Tools:-**

1. Code Blocks with GCC compiler.

**Theory:-**

Stacks are the most important in data structures. The notation of a stack in computer science is the same as the notion of the Stack to which you are accustomed in everyday life. For example, a recursion program on which function call itself, but what happen when a function which is calling itself call another function. Such as a function ‘A’ call function ‘B’ as a recursion. So the firstly function ‘B’ is call in ‘A’ and then function ‘A’ is work. So this is a **Stack**. This is a Stack is **First in Last Out** data structure.

**Insertions in Stack:**

In Stacks, we know the array work, sometimes we need to modified it or add some element in it. For that purpose we use insertion scheme. By the use of this scheme we insert any element in Stacks using array. In Stack we maintain only one node which is called **TOP**. And **Push** terminology is used as insertions.

**Deletion in Stack:**

In the deletion process, the element of the Stack is deleted on the same node which is called **TOP**. In stacks, it’s just deleting the index of the TOP element which is added at last. In Stacks **Pop** terminology is used as deletion.

**Display of Stack:**

In displaying section, the elements of Stacks are been display by using loops and variables as a reverse order. Such that, last element is display at on first and first element enters display at on last.

**Algorithm for top of stack varying method**

*1. Declare and initialize necessary variables, eg top = -1, MAXSIZE etc.  
2. For push operation, If top = MAXSIZE - 1  
print "stack overflow"  
else  
top = top + 1;  
 Read item from user  
stack[top] = item  
3. For next push operation, goto step 2.  
4. For pop operation,  
If top = -1  
 print "Stack underflow"  
Else  
 item = stack[top]  
 top = top - 1  
 Display item  
5. For next pop operation, goto step 4.  
6. Stop*

**Lab Task:-**

1. Insertion in stack
2. Deletion in stack
3. Display the stack

## Conclusion

---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

----------------------------------------------

**(Concerned Teacher/Lab Engineer)**

**Experiment # 04**

**Link-list Basic Insertion and Traversal**

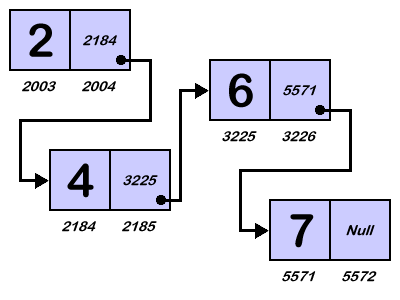
**Objective:-**

The objective of this session is to understand the various operations on linked list in C++.

**Software Tools:-**

1. Code Blocks with GCC compiler.

**Theory:-**



You have already seen how data is organized and processed sequentially using an array, called a sequential list. You have performed several operations on sequential lists, such as sorting, inserting, deleting, and searching. You also found that if data is not sorted, searching for an item in the list can be very time consuming, especially with large lists. Once the data is sorted, you can use a binary search and improve the search algorithm. However, in this case, insertion and deletion become time consuming, especially with large lists because these operations require data movement. Also, because the array size must be fixed during execution, new items can be added only if there is room. Thus, there are limitations when you organize data in an array.

This session helps you to overcome some of these problems. Last session showed how memory (variables) can be dynamically allocated and deallocated using pointers. This session uses pointers to organize and process data in lists, called linked lists. Recall that when data is stored in an array, memory for the components of the array is contiguous—that is, the blocks are allocated one after the other. However, as we will see, the components (called nodes) of a linked list need not be contiguous.

**LINKED LIST:-**

A linked list is a collection of components, called nodes. Every node (except the last node) contains the address of the next node. Thus, every node in a linked list has two components: one to store the relevant information (that is, data) and one to store the address, called the link, of the next node in the list. The address of the first node in the list is stored in a separate location, called the head or first. Figure 1 is a pictorial representation of a node.

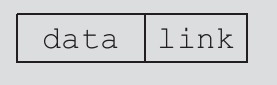


Figure 1

**Linked list:** A list of items, called **nodes**, in which the order of the nodes is determined by the address, called the **link**, stored in each node.

The list in Figure 2 is an example of a linked list.

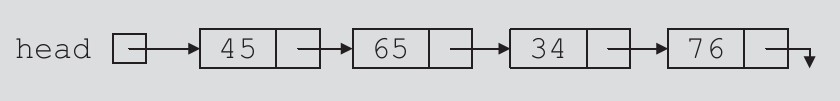


Figure 2

The arrow in each node indicates that the address of the node to which it is pointing is stored in that node. The down arrow in the last node indicates that this link field is **NULL**.

For a better understanding of this notation, suppose that the first node is at memory location

1200, and the second node is at memory location 1575, see Figure 3.

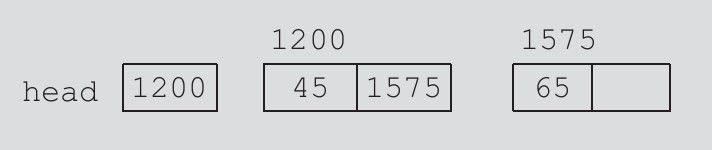


Figure 3

The value of the head is 1200, the data part of the first node is 45, and the link component of the first node contains *1575*, the address of the second node. We will use the arrow notation whenever we draw the figure of a linked list.

For simplicity and for the ease of understanding and clarity, Figures 3 through 5 use decimal integers as the values of memory addresses. However, in computer memory the memory addresses are in binary.

Because each node of a linked list has two components, we need to declare each node as a **class** or **struct**. The data type of each node depends on the specific application—that is, what kind of data is being processed. However, the link component of each node is a pointer. The data type of this pointer variable is the node type itself. For the previous linked list, the definition of the node is as follows. (Suppose that the data type is **int**.)

*struct nodeType*

*{*

*int info;*

*nodeType\* link;*

*};*

The variable declaration is as follows:

***nodeType\* head;***

**Linked Lists: Some Properties**

To better understand the concept of a linked list and a node, some important properties of linked lists are described next.

Consider the linked list in Figure 4.

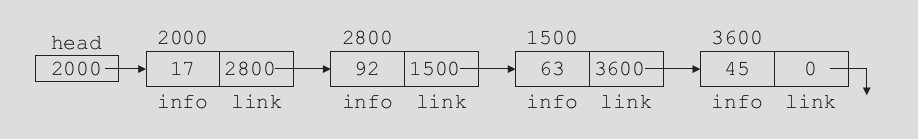
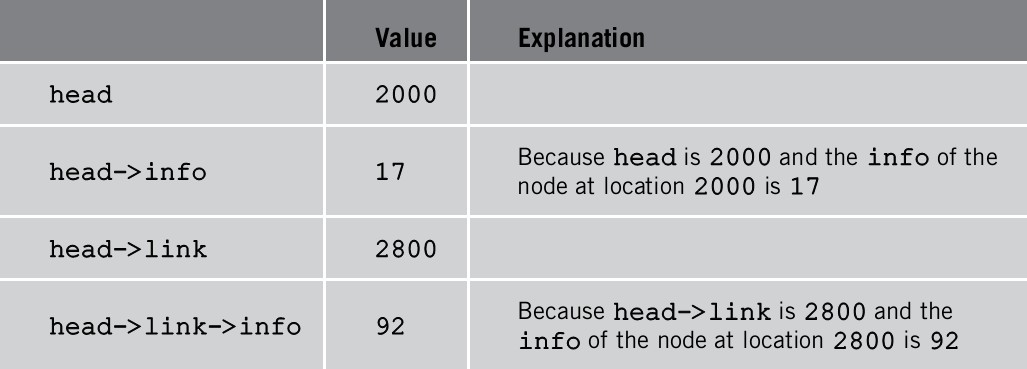


Figure 4

This linked list has four nodes. The address of the first node is stored in the pointer head. Each node has two components: info, to store the info, and link, to store the address of the next node. For simplicity, we assume that info is of type int. Suppose that the first node is at location 2000, the second node is at location 2800, the third node is at location 1500, and the fourth node is at location 3600. Table 1 shows the values of head and some other nodes in the list shown in Figure 4.

Table 1: Values of head and some of the nodes of the linked list in Figure 4



Suppose that current is a pointer of the same type as the pointer head. Then the statement

*current = head;*

Copies the value of head into current. Now consider the following statement:

*current = current->link;*

This statement copies the value of ***curren****t->****link****,* which is **2800**, into ***current****.* Therefore, after this statement executes, ***current*** points to the second node in the list. (When working with linked lists, we typically use these types of statements to advance a pointer to the next node in the list.) See Figure 5.

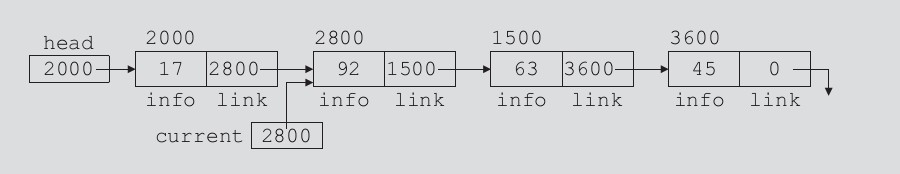
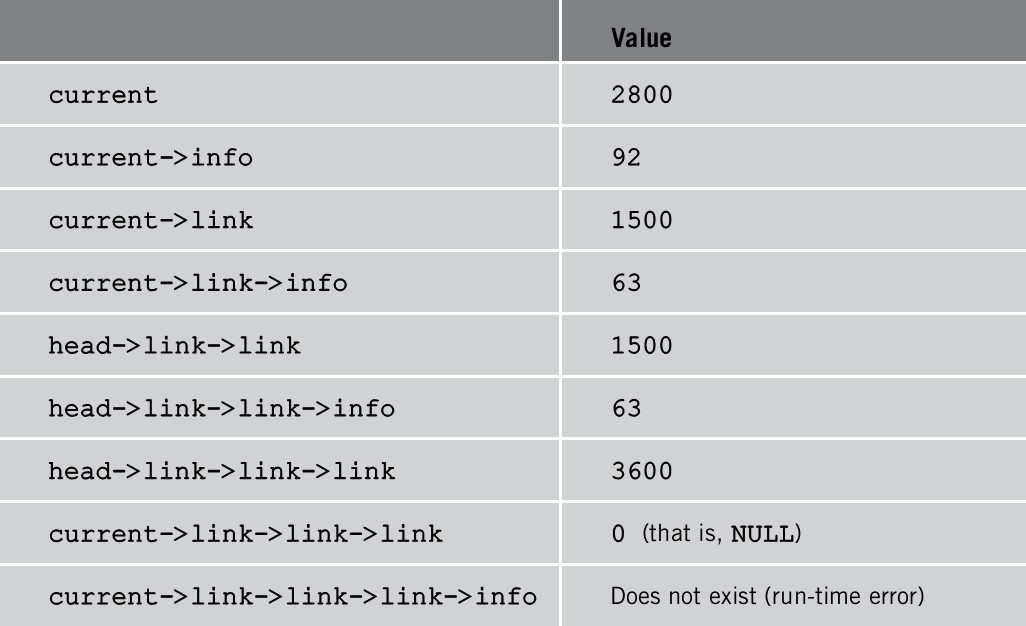


Figure 5:List after the statement current = current->link; executes

Table 2 shows the values of current, head, and some other nodes in Figure 5.

Table 2: Values of current, head, and some of the nodes of the linked list in Figure5



From now on, when working with linked lists, we will use only the arrow notation.

**TRAVERSING A LINKED LIST:-**

The basic operations of a linked list are as follows: Search the list to determine whether a particular item is in the list, insert an item in the list, and delete an item from the list. These operations require the list to be traversed. That is, given a pointer to the first node of the list, we must step through the nodes of the list.

**Algorithm:-**

**(Traversing a Linked List) Let LIST be a linked list in memory. This algorithm traverses LIST applying an operation PROCESS to each element of LIST. The variable PTR points to the node currently being processed.**

**1. Set PTR=START [Initialize pointer PTR]**

**2. Repeat Step 3 and 4 while PTR ≠ NULL.**

**3. Apply PROCESS TO INFO[PTR].**

**4. Set PTR=LINK[PTR] [PTR now points to the next node] [End of Step 2 Loop]**

**5. Exit.**

Suppose that the pointer **head** points to the first node in the list, and the link of the last node is **NULL**. We cannot use the pointer head to traverse the list because if we use the **head** to traverse the list, we would lose the nodes of the list. This problem occurs because the links are in only one direction. The pointer **head** contains the address of the first node, the first node contains the address of the second node, the second node contains the address of the third node, and so on. If we move **head** to the second node, the first node is lost (unless we save a pointer to this node). If we keep advancing **head** to the next node, we will lose all the nodes of the list (unless we save a pointer to each node before advancing **head**, which is impractical because it would require additional computer time and memory space to maintain the list).

Therefore, we always want **head** to point to the first node. It now follows that we must traverse the list using another pointer of the same type. Suppose that **current** is a pointer of the same type as **head**. The following code traverses the list:

*current =head;*

*while(current!=NULL)*

*{*

*cout<<current->info<<"";*

*current=current->link;*

*}*

**Lab Task:-**

**Write a C++ code using functions for the following operations.**

**1. Creating a linked List.**

**2. Traversing a Linked List.**

**Create a complete menu for the above options and also create option for reusing it.**

## Conclusion

---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

----------------------------------------------

**(Concerned Teacher/Lab Engineer)**

**Experiment # 05**

**Link list-Basic Deletion at desired position**

**Objective:-**

The objective of this session is to insertion, traversal and deletion at desired position in link list using C++.

**Software Tools:-**

1. Code Blocks with GCC compiler.

**Theory:-**

This section discusses how to insert an item into, and delete an item from, a linked list. Consider the following definition of a node. (For simplicity, we assume that the info type is int.

*struct nodeType*

*{*

*int info nodeType\* link;*

*};*

We will use the following variable *nodeType \*head, \*p, \*q, \*newNode;* **INSERTION:-**

Algorithms which insert nodes into the linked list come up in various situations. We discuss three of them here. The first one inserts a node at the beginning of the list, the second one inserts a node after a node with a given location, and the third one inserts a node into the sorted list.

**Inserting at the Beginning of the List:-**

Suppose our linked list is not necessarily sorted and there is no reason to insert a new node in any special place in the list. Then the easiest place to insert the node is at the beginning of the list. An algorithm that does so follows.

**Algorithm:-**

**INSFIRST(INFO,LINK,START,AVAIL,ITEM)**

**This algorithm inserts ITEM as the first node in the list.**

**1. [OVERFLOW?] If AVAIL=NULL then write OVERFLOW and Exit.**

**2. [Remove first node from AVAIL list.] Set NEW=AVAIL and AVAIL=LINK[AVAIL]**

**3. Set INFO[NEW]=ITEM [Copies new data into new node]**

**4. Set LINK[NEW]=START [New node now points to the original first node]**

**5. Set START=NEW [Change START so it points to the new node ]**

**6. Exit.**

**Inserting after a Given Node:- Algorithm:- INSLOC(INFO,;INK,START,AVAIL,LOC,ITEM)**

**This algorithm inserts ITEM so that item follows the node with location LOC or inserts ITEM as a first node when LOC=NULL.**

**1. [OVERFLOW?] If AVAIL=NULL then write OVERFLOW and Exit.**

**2. [Remove First Node from the AVAIL list].**

**Set NEW=AVAIL and AVAIL=LINK[AVAIL].**

**3. Set INFO[NEW] =ITEM. [Copies new data into new node.]**

**4. If LOC=NULL then [Inserts as first node] Set LINK[NEW]=START and START=NEW.**

**else [Insert after node with location LOC]**

**Set LINK[NEW]=LINK[LOC] and LINK[LOC]=NEW. [End of If Structure]**

**5. Exit.**

**Inserting into a Sorted Linked List:-**

Suppose ITEM is to be inserted into a sorted linked list. Then ITEM must be inserted between nodes A and B so that

*INFO(A) < ITEM < INFO(B)*

The following is the procedure which finds the location LOC of node A that is which finds the location LOC of the last node in the list whose value is less than ITEM. Traverse the list using pointer variables PTR and comparing ITEM with INFO[PTR] at every node. While traversing keep track of the location of the preceding node by using a pointer variable SAVE. Thus SAVE and PTR are updated by assignments.

**Algorithm:-**

**FINDA(INFO, LINK START, ITEM,LOC)**

**This procedure finds the location LOC of the last node in a sorted list such that INFO[LOC] < ITEM or set LOC = NULL.**

**1. [List Empty?] If START = NULL then set LOC= NULL and**

**Return.**

**2. [Special Case?] If ITEM<INFO[START] then Set LOC =NULL and**

**Return.**

**3. Set SAVE= START and PTR = LINK[START] [Initialize**

**Pointers]**

**4. Repeat Step 5 and 6 while PTR ≠ NULL.**

**5. If ITEM<INFO[PTR] then**

**Set LOC =SAVE and Return. [End of If Structure]**

**6. Set SAVE =PTR and PTR= LINK[PTR] [Update Pointers] [End of Step 4 Loop]**

**7. Set LOC=SAVE.**

**8. Exit.**

**Now we have all the components to present an algorithm which inserts ITEM into a linked list. The simplicity of the algorithm comes from using the previous two procedures.**

**Algorithm:-**

**INSERT(INFO, LINK , START, AVAIL, ITEM)**

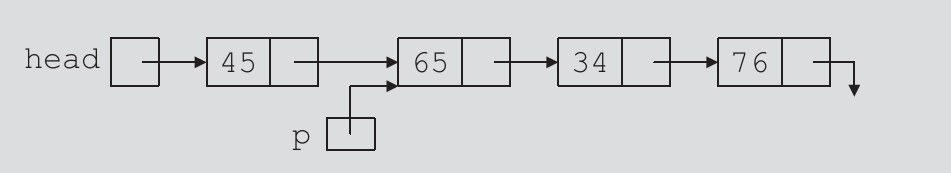
**This algorithm inserts ITEM into a sorted linked list.**

**1. Call FINDA(INFO, LINK, START, AVAIL , ITEM)**

**2. Call INSLOC(INFO, LINK, START, AVAIL, LOC, ITEM).**

**3. Exit.**

Consider the linked list shown in Figure 6.

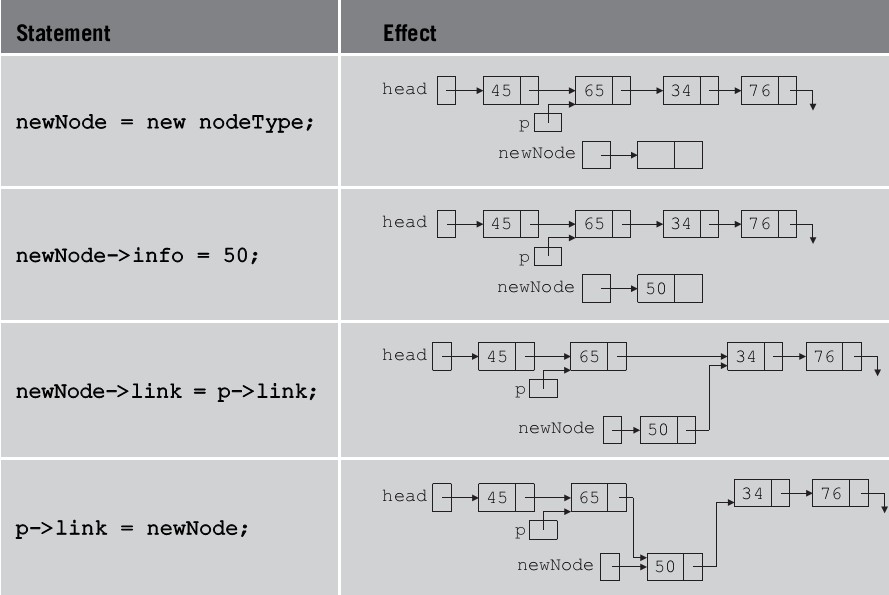


Suppose that **p** points to the node with **info 65**, and a new node with **info 50** is to be created and inserted after **p**. Consider the following statements:

**newNode=new nodeType; //create new Node newNode->info=50; //store 50 in the newnode newNode->link=p->link;**

**p->link=newNode;**

Table 3 shows the effect of these statements. Table 3: Inserting a node in a linked list



Note that the sequence of statements to insert the node, that is,

**newNode->link = p->link;**

**p->link = newNode;**

is very important because to insert **newNode** in the list we use only one pointer, **p**, to adjust the links of the nodes of the linked list. Suppose that we reverse the sequence of the statements and execute the statements in the following order:

**p->link = newNode;**

**newNode->link = p->link;**

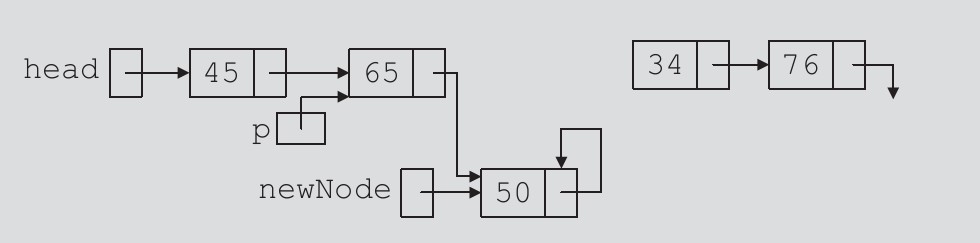


Figure: List after the execution of the statement **p->link = newNode;** followed by the execution of the statement **newNode->link = p->link;**

From Figure, it is clear that **newNode** points back to itself and the remainder of the list is lost.

**Lab Task:-**

**Write a C++ code using functions for the following operations.**

**1. Creating a linked List.**

**2. Traversing a Linked List.**

**3. Inserting the node at the start of the list.**

**4. Inserting a node after a given node.**

**5. Inserting a node in a sorted list.**

**Create a complete menu for the above options and also create option for reusing it.**

## Conclusion

---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

----------------------------------------------

**(Concerned Teacher/Lab Engineer)**

**Experiment # 06**

**Circular Link list**

**Objective:-**

The objective of this session is to understand the various operations on circular link list using C++.

**Software Tools:-**

1. Code Blocks with GCC compiler.

**Theory:-**

A circular list is one in which the next field of the last node points to the first node of the list or the next field of the last node contains the address of the first node of the list.

Looking at the figure one would say that the starting node is one the counter is at the last node is one that contain 30. Now what do we say if we represent the same list as shown on figure certainly, everyone would have a different answer

30

20

10

Representing a circular linked list

10

30

20

Another representing a circular linked list

To avoid these confusions, it is always a good practice to associate a front and rear pointer to pointer to the first node the last node as shown in figure. This acts as a reference and help us in carrying out the operation on the list.

Front Rear

30

20

10

Adding reference pointer to a circular lined list

**Creating circular linked list**

The structure of a circularly lined list is the same way as for a linear linked list

struct node

{

Intvalue ;

Struct node \*next;

};

Node node1;

Node1\*front,\* rear;

In order to create a list say of two node, we first create and allocate memory for the first node in the following manner

Node1 \*firstnode;

Now we should associate the front and the rear pointer to it and let front and rear pointer point to it. This is done by writing the following lines of code

Front=firstnode and rear=firstnode

Now we should insert same data in value field of the node just created through following lines of code

Firstnode->value=val1;

Having creating he first node, we create and insert the 2nd node after 1st node and then insert data into the value field

Node1 \*secondnode;

Secondnode =node1;

Secondnode->value=val2;

Now the rear pointer which is now pointer to the first node should point to 2nd node that is

Rear->next=firstnode;

**Operation on Circular Linked List**

Common operation performed on circular linked list are

* Inserting a node at the Front of the list
* Inserting a node at the End of the list
* Deleting a node from the Front of a list

**Inserting a Node at the Front of the List**

We first check if the list is empty. If the list exists, now create a new node that has to be the node in the list. Inserting data in its value field. Let its next field pointer to where front position and the front pointer point to the new node. This is illustrate in figure

Front Rear

20

40

30

Original list before inserting any node

Temp

Rear

Front

40

30

20

10

Inserting a node at the start of the list

**ALGORITHM**

*Insert front (node \*front, item)*

*Declear the pointers*

*Node1 \*temp, \*rear;*

*Check whether the list is empty or not*

*If (front==null)*

*Cout<<”The list is empty”<<endl;*

*Let temp=front*

*Let rear=temp*

*Allocate memory for node to be inserted and assign data in the value field of its rear node to be inserted*

*Temp-> value=data*

*For non-empty list*

*Temp->next=front*

*Front=temp*

*Rear->next=front*

**Inserting a Node at the End of the List**

As usual, we check whether the list is empty or not, if empty, then front and rear has to be updated to the newly created node that is pointed by the pointer temp. Insert data in the value field of the struct node. Otherwise the operation is similar to the previous once. If the list is not empty, the next pointer of the node pointer by rear point where temp points. Again let the rear pointer of the node pointer by temp pointer, where front pointer and advance

The rear pointer. This is illustrate by the following figure.

Front Rear

10

30

20

Original list before inserting any node

Temp

Rear

Front

40

30

20

10

Inserting a node at the End of the list

**ALGORITHIM**

*Insertatend (node \*front, item)*

*Declare the necessary pointers*

*Node \*temp, \*rear*

*Allocate memory for node to be the inserted and assign data to the value field of the node to be inserted*

*Temp->value=item;*

*Check whether it is an empty list*

*If (front==null)*

*Cout<<”The list is empty”<<endl;*

*Let temp=front*

*Let rear=temp*

*For non-empty list*

*Rear->next=temp*

*Temp->next=front*

*Rear=rear->next*

**Deletion of Node from the Start of the List:**

As it is evident from figures, when a node is deleted either from the list we have to free the memory space occupied by that node. In order to free the node a pointer pointing to this free node. For that we use the temp pointer and we assumed pointer temp to point where front pointer. Now the memory occupied by the node is released by writing free (temp). Front is now moved to point to the next node and the pointer where front is pointing now.

FrontRear

30

20

10

Original list before deleting a node from list

Temp

10 Null

Front Rear

30

20

The original list after deleting

**ALGORITHIM**

*Deleting node (node1 \*front, item)*

*Declare the pointers*

*Node1 \*temp, \*rear*

*Check if list is empty*

*If(front==null)*

*Cout<<”The list is empty”<<endl;*

*If the list is not empty*

*Let temp=front*

*Let front=front->next*

*Let rear->next=front*

*Cout<<”the element is deleted”<<endl;*

*Free the memory occupied by node 1*

*Free(temp)*

**Lab Task:-**

* Create the circular link list using classes.

## Conclusion

---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

----------------------------------------------

**(Concerned Teacher/Lab Engineer)**

**Experiment # 07**

**Double Link list**

**Objective:-**

The objective of this session is to understand the various operations on double link list using C++.

**Software Tools:-**

1. Code Blocks with GCC compiler.

**Theory:-**

Doubly link list help in traversing the list in both the forward and reverse direction. Every node has two links the forward link and backward list. Carrying out any operation on doubly linked list is quite similar to what we have been doing on a singly link list. The difference is that for any insertion or deletion of node, we need to adjust or set 4 pointer instead of two pointer.

*The structure of doubly link list is declare as*

Struct node

{

Int value;

Struct node \*next;

Struct node\* prev;

}

Type def struct node node1;

The major operation on a doubly link list that we discuss here are:

* Inserting a node at the front of a doubly linked list
* Deleting a node from the front of a doubly linked list
* Inserting a node at the rare of a doubly linked list
* Deleting a node from the rare of a doubly linked list

**Inserting a Node at The Front Of A Doubly Linked List:**

To start with this operation, we allocate memory for node to be insert at the front of the doubly linked list. Front and rare pointers are assigned for easier navigation of the list. Noe assigned a pointer say “temp” to it and store some data in value field. Let the next pointer of this node point, where front initially points. Let’s the previous pointer of this node pointed by front point to the next pointer of the node pointed by the pointer temp. Now, the previous pointer of this first node contain 100 should point to the next pointer of the node to be inserted. This is illustrated in figure.

100 NULL

200

NULL 100

FRONT fig (a) REAR

**ALGORITHM:**

***Insertatfront (node1 \*front)***

***If (front==null)***

***Cout<<”This is empty”<<endl;***

***Declaer the other two pointer to be used***

***Node 1 \*temp,\*rare;***

***Allocate memory for the new node to be inserted and assign data to it***

***Temp= (node1\*) malloc (sizeoff (node1)); OR Temp= Struct node;***

***Temp->value=data;***

***If it is an empty list***

***Temp=front;***

***Let temp=rare;***

***Let temp->next=null;***

***For non-empty list***

***Let temp->next =front;***

***Let front->prev=temp->next***

***Set temp-<prev=null***

***Let front=temp;***

***end***

**Inserting Node at The End Of A Doubly Linked List.**

300 NULL

200

NULL 100

Fig (a)

FRONT REAR

TEMP

400 NULL

300

200

NULL 100

FRONT

400 NULL

REAR

Fig (b)

We start with allocating memory for the node to be inserted in a given linked list. We also assign a pointer say “temp” to it and data in value field. Now, we also need a pointer that traversed the list. Let this pointer be “listptr”. Now in order to check for the last node, the node after which we have to insert the new node, we traversed with listptr until listptr-> next ==null. Now we stop with listptr at 2nd last node. Inittily, we will point listptr->next to where temp points to the next pointer of the node at the end of a list. Now, the previous pointer of temp should point to the next pointer of the node containing vale 300. The various pointer adjustments have been illustrated in figure

300

200

100

FRONT

400

TEMP

**ALGORITHM:**

*Check if the list is empty*

*If front ==null*

*Cout<<”The list is empty”<<endl;*

*Let temp =front*

*Let listptr=front*

*Declare pointer from traversing the list*

*Node 1 \*temp,\*listptr*

*Allocate memory for the node pointer by temp*

*Temp= struct node*

*Assign data and set next field*

*Temp->value=data*

*Temp->next=null*

*For non-empty list*

*While (listptr-> next ->next! =null)*

*Listptrr=listptr->next*

*Listptr->next->nextshould point to temp*

*Listptr->next->next=temp*

*Point previous field of temp where listptr->next is pointing*

*Temp->prev=list->next;*

**Deleting a Node from the Front of a List:**

When we have to delete a node from a doubly linked list, we use a extra pointer that points to the node to be deleted. This pointer is also later used to free the memory occupied by this node. We should advance the front pointer to point to the 2nd node now pointed by the front pointer should also be set to null this node become the first node of the list after the deletion operation. This is illustrate in figure

**TEMP**

300

200

NULL 100

FRONT

Fig (a)

400 NULL

400 NULL

300

NULL 200

FRONT REAL

**ALGORITHM**

*Declare the pointer*

*Node \*temp;*

*Check if the list is empty*

*If (front==null)*

*Cout<<”the list id empty”<<endl;*

*For non-empty list*

*Set temp t where front pointer*

*Temp=front;*

*Advance front to point to 2nd node*

*Front=front->next;*

*Set previous field of 2nd node to null*

*Front->previous =null;*

*Now free the memory occupied by node pointed by temp*

*Free (temp);*

*End*

**Deleting A Node From The END Of A List:**

To delete any node from a double link list, we have to associate another pointer with the list. This pointer temp as shown in fig helps as freeing the memory occupied by the node that has been deleted. We traverse the list with temp pointer but at first we check if the list is empty. In such case, we cannot delete any node.

300

200

NULL 100

FRONT

Fig (a) **Tem**

400 NULL

300

200

NULL 100

Fig (b) REAR

**ALOGORITHIM**

*Int another pointer in the first node*

*Node \*temp,\*rear;*

*Check if the list is empty*

*If (front==null)*

*cout<<”The list is empty”<<endl;*

*if (front!==null)*

*temp=front;*

*temp=temp->next*

*until temp->next->nextis null*

*pointer rear to 2nd last node*

*rear ->temp;*

*set the next field of 2nd last node to null*

*rear->next=null;*

*free the memory*

*free(temp)*

*end;*

**Lab Task:-**

* Create the doubly link list.

## Conclusion

---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

----------------------------------------------

**(Concerned Teacher/Lab Engineer)**

**Experiment # 08**

**Basics of Graph Data Structure**

**Objective:-**

The objective of this session is to understand the Basic concepts of Graph.

**Software Tools:-**

1. Code Blocks with GCC compiler.

**Theory:-**

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as **vertices,** and the links that connect the vertices are called **edges**.

Formally, a graph is a pair of sets **(V, E),** where **V** is the set of vertices and **E** is the set of edges, connecting the pairs of vertices. Take a look at the following graph −

In the above graph,

V = {a, b, c, d, e}

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

**Graph Data Structure**

Mathematical graphs can be represented in data-structure. We can represent a graph using an array of vertices and a two dimensional array of edges. Before we proceed further, let's familiarize ourselves with some important terms

* **Vertex** :

Each node of the graph is represented as a vertex. In example given below, labeled circle represents vertices. So A to G are vertices. We can represent them using an array as shown in image below. Here A can be identified by index 0. B can be identified using index 1 and so on.

* **Edge**:

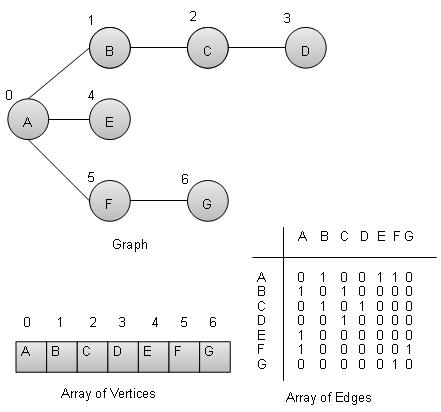
Edge represents a path between two vertices or a line between two vertices. In example given below, lines from A to B, B to C and so on represents edges. We can use a two dimensional array to represent array as shown in image below. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.

* **Adjacency**:

Two node or vertices are adjacent if they are connected to each other through an edge. In example given below, B is adjacent to A, C is adjacent to B and so on.

* **Path**:

Path represents a sequence of edges between two vertices. In example given below, ABCD represents a path from A to D.



**Basic Operations:-**

Following are basic primary operations of a Graph which are following.

* **Add Vertex** − add a vertex to a graph.
* **Add Edge** − add an edge between two vertices of a graph.
* **Display Vertex** − display a vertex of a graph.

**Add Vertex Operation:-**

//add vertex to the vertex list

Void addVertex (char label){

struct vertex\* vertex =(struct vertex\*) malloc (sizeof(struct vertex));

vertex->label = label;

vertex->visited =false;

lstVertices[vertexCount++]= vertex;

}

**Add Edge Operation:-**

//add edge to edge array

voidaddEdge(intstart,intend){

adjMatrix[start][end]=1;

adjMatrix[end][start]=1;

}

**Display Edge Operation:-**

//display the vertex

voiddisplayVertex(intvertexIndex){

printf("%c ",lstVertices[vertexIndex]->label);

}

## Conclusion

---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

----------------------------------------------

**(Concerned Teacher/Lab Engineer)**

**Experiment # 09**

**Directed and Undirected graph Implementation**

**Objective:-**

The objective of this session is to learn how to implement the Directed and Undirected Graph.

**Software Tools:-**

1. Code Blocks with GCC compiler.

**Theory:-**

Graphs are widely-used structure in computer science and different computer applications. We don't say data structure here and see the difference. Graphs mean to store and analyze metadata, the connections, which present in data. For instance, consider cities in your country. Road network, which connects them, can be represented as a graph and then analyzed. We can examine, if one city can be reached from another one or find the shortest route between two cities.

First of all, we introduce some definitions on graphs. Next, we are going to show, how graphs are represented inside of a computer. Then you can turn to basic graph algorithms.

There are two important sets of objects, which specify graph and its structure. First set is **V,** which is called **vertex-set**. In the example with road network cities are vertices. Each vertex can be drawn as a circle with vertex's number inside.

|  |  |
| --- | --- |
| http://www.algolist.net/img/graphs/graph-def-1.png |  |
| Vertices |  |

Next important set is **E,** which is called **edge-set. E** is a subset of **V x V**. Simply speaking, each edge connects two vertices, including a case, when a vertex is connected to itself (such an edge is called a loop).All graphs are divided into two big groups: directed and undirected graphs. The difference is that edges in directed graphs, called arcs, have a direction. These kinds of graphs have much in common with each other, but significant differences are also present. We will accentuate which kind of graphs is considered in the particular algorithm description. Edge can be drawn as a line. If a graph is directed, each line has an arrow.

|  |  |
| --- | --- |
| http://www.algolist.net/img/graphs/graph-def-2.png | http://www.algolist.net/img/graphs/graph-def-3.png |
| undirected graph | directed graph |

Now, we present some basic graph definitions.

* Sequence of vertices, such that there is an edge from each vertex to the next in sequence, is called **path**. First vertex in the path is called the start vertex; the last vertex in the path is called the end vertex. If start and end vertices are the same, path is called **cycle**. Path is called simple, if it includes every vertex only once. Cycle is called simple, if it includes every vertex, except start (end) one, only once. Let's see examples of path and cycle.

|  |  |
| --- | --- |
| http://www.algolist.net/img/graphs/graph-def-5.png | http://www.algolist.net/img/graphs/graph-def-6.png |
| path (simple) | cycle (simple) |

The last definition we give here is a weighted graph. Graph is called weighted, if every edge is associated with a real number, called edge weight. For instance, in the road network example, weight of each road may be its length or minimal time needed to drive along.

|  |  |
| --- | --- |
| http://www.algolist.net/img/graphs/graph-def-7.png |  |
| weighted graph |  |

# Undirected Graphs Representation:-

There are several possible ways to represent a graph inside the computer. We will discuss two of them: **adjacency matrix** and **adjacency list**.

## Adjacency matrix:-

Each cell **aij**of an adjacency matrix contains **0**, if there is an edge between i-th and j-th vertices, and **1**otherwise. Before discussing the advantages and disadvantages of this kind of representation, let us see an example.

|  |  |
| --- | --- |
| Graph sample | Adjacency matrix for the graph |
| Graph | Adjacency matrix |

|  |  |
| --- | --- |
| Edge (2, 5) | Cells for edge (2, 5) |
| Edge (2, 5) | Cells for the edge (2, 5) |

|  |  |
| --- | --- |
| Graph sample | Adjacency matrix for the graph |
| Edge (1, 3) | Cells for the edge (1, 3) |

The graph presented by example is undirected. It means that its adjacency matrix is symmetric. Indeed, in undirected graph, if there is an edge (2, 5) then there is also an edge (5, 2). This is also the reason, why there are two cells for every edge in the sample. Loops, if they are allowed in a graph, correspond to the diagonal elements of an adjacency matrix.

**Advantages.** Adjacency matrix is very convenient to work with. Add (remove) an edge can be done in O(1) time, the same time is required to check, if there is an edge between two vertices. Also it is very simple to program and in all our graph tutorials we are going to work with this kind of representation.

**Disadvantages.**

* Adjacency matrix consumes huge amount of memory for storing big graphs. All graphs can be divided into two categories, sparse and dense graphs. Sparse ones contain not much edges (number of edges is much less, that square of number of vertices, |E| << |V|2). On the other hand, dense graphs contain number of edges comparable with square of number of vertices. Adjacency matrix is optimal for dense graphs, but for sparse ones it is superfluous.
* Next drawback of the adjacency matrix is that in many algorithms you need to know the edges, adjacent to the current vertex. To draw out such an information from the adjacency matrix you have to scan over the corresponding row, which results in O(|V|) complexity. For the algorithms like DFS or based on it, use of the adjacency matrix results in overall complexity of O(|V|2), while it can be reduced to O(|V| + |E|), when using adjacency list.
* The last disadvantage, we want to draw you attention to, is that adjacency matrix requires huge efforts for adding/removing a vertex. In case, a graph is used for analysis only, it is not necessary, but if you want to construct fully dynamic structure, using of adjacency matrix make it quite slow for big graphs.

To sum up, adjacency matrix is a good solution for dense graphs, which implies having constant number of vertices.

## Adjacency list

This kind of the graph representation is one of the alternatives to adjacency matrix. It requires less amount of memory and, in particular situations even can outperform adjacency matrix. For every vertex adjacency list stores a list of vertices, which are adjacent to current one. Let us see an example.

|  |  |
| --- | --- |
| Graph sample | Adjacency list for the graph |
| Graph | Adjacency list |

|  |  |
| --- | --- |
| Graph sample | Adjacency list for the graph |
| Vertices, adjacent to {2} | Row in the adjacency list |

**Advantages.** Adjacent list allows us to store graph in more compact form, than adjacency matrix, but the difference decreasing as a graph becomes denser. Next advantage is that adjacent list allows to get the list ofadjacent vertices in O(1) time, which is a big advantage for some algorithms.

**Disadvantages.**

* Adding/removing an edge to/from adjacent list is not so easy as for adjacency matrix. It requires, on the average, O(|E| / |V|) time, which may result in cubical complexity for dense graphs to add all edges.
* Check, if there is an edge between two vertices can be done in O(|E| / |V|) when list of adjacent vertices is unordered or O(log2(|E| / |V|)) when it is sorted. This operation stays quite cheap.
* Adjacent list doesn't allow us to make an efficient implementation, if dynamically change of vertices number is required. Adding new vertex can be done in O(V), but removal results in O(E) complexity.

To sum up, adjacency list is a good solution for sparse graphs and lets us changing number of vertices more efficiently, than if using an adjacent matrix. But still there are better solutions to store fully dynamic graphs.

**Lab Task:-**

* Implement the directional graph with and without weight.
* Implement the bidirectional graph with and without weight.

## Conclusion

---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

----------------------------------------------

**(Concerned Teacher/Lab Engineer)**

**Experiment # 10**

**Prim’s Algorithm**

**Objective:-**

The objective of this session is to learn how to implement Prim’s Algorithm.

**Software Tools:-**

1. Code Blocks with GCC compiler.

**Theory:-**

Prim's algorithm is a [greedy algorithm](https://en.wikipedia.org/wiki/Greedy_algorithm) that finds a [minimum spanning tree](https://en.wikipedia.org/wiki/Minimum_spanning_tree) for a [weighted](https://en.wikipedia.org/wiki/Weighted_graph) [undirected graph](https://en.wikipedia.org/wiki/Undirected_graph). This means it finds a subset of the [edges](https://en.wikipedia.org/wiki/Edge_(graph_theory)) that forms a [tree](https://en.wikipedia.org/wiki/Tree_(graph_theory)) that includes every [vertex](https://en.wikipedia.org/wiki/Vertex_(graph_theory)), where the total weight of all the [edges](https://en.wikipedia.org/wiki/Graph_theory) in the tree is minimized. The algorithm operates by building this tree one vertex at a time, from an arbitrary starting vertex, at each step adding the cheapest possible connection from the tree to another vertex.

**Procedure:**

*#include <iostream>*

*#include <conio.h>*

*using namespace std;*

*int main()*

*{*

*int b[4][4];*

*int visited[4];*

*int min=999;*

*int u=0;*

*int v=0;*

*int total=0;*

*for (inti = 0; i< 4; i++)*

*{*

*visited[i]= 0;//nothing is visited*

*cout<<"enter values for "<<(i+1)<<" row"<<endl;*

*for (int j = 0; j < 4; j++)*

*{*

*cin>>b[i][j];*

*if(b[i][j]==0)*

*b[i][j]=999;//replacing it by infinity*

*}//end of internal loop*

*}//end of external for loop*

*visited[0]=1;//mark first node as visited*

*//start of algorithm*

*for(int counter=0;counter < 3;counter++)*

*{*

*min = 999;*

*for (inti = 0; i< 4; i++)*

*{*

*if (visited[i] == 1)//if the node is visited*

*{*

*for (int j = 0; j < 4; j++)*

*{*

*if (visited[j] != 1)//if the jth node is not visited*

*{*

*if (min>b[i][j])*

*{*

*min = b[i][j];*

*u = i;*

*v = j;*

*}//end of internal if*

*}//end of external if*

*}*

*}*

*}*

*visited[v]=1;//we marked this node as visited*

*total+=min;*

*cout<<"Edge found: "<<u<<"->"<<v<<" weight:"<<min<<endl;*

*}*

*cout<<"Total weight:"<<total<<endl;*

*getch();*

*}*

**Lab Task: -**

1. Travelling sales man problem case study.

## Conclusion

---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

----------------------------------------------

**(Concerned Teacher/Lab Engineer)**

**Experiment # 11& 12**

**In order, preorder and post order Traversal**

**Objective:-**

To understand the **Inorder**, **preorder** and **post** **order** Traversal.

**Software Tools:-**

1. Code Blocks with GCC compiler.

**Theory:-**

**1**-**Inorder.** The ordering is: the left subtree, the current node, the right subtree.

**2-Preorder.** The ordering is: the current node, the left subtree, the right subtree.

**3**-**Post order.** The ordering is: the left subtree, the right subtree, the current node.

**Procedure:**

**In order Traversal:**

Algorithm In order (tree)

1. Traverse the left subtree, i.e., call **inorder** (left-subtree)

2. Visit the root.

3. Traverse the right subtree, i.e., call **inorder** (right-subtree)

Uses of in order  
In case of binary search trees (BST), in order traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of in order traversal where Inorder traversal s reversed, can be used.

**Preorder Traversal:**

Algorithm Preorder(tree)

1. Visit the root.

2. Traverse the left subtree, i.e., call Preorder(left-subtree)

3. Traverse the right subtree, i.e., call Preorder(right-subtree)

Uses of Preorder  
Preorder traversal is used to create a copy of the tree. Preorder traversal is also used to get prefix expression on of an expression tree.

**Post order Traversal:**

Algorithm Post order (tree)

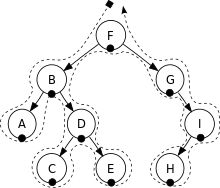
1. Traverse the left subtree, i.e., call Post order(left-subtree)

2. Traverse the right subtree, i.e., call Post order(right-subtree)

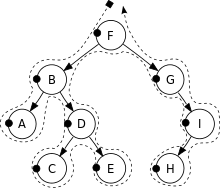
3. Visit the root.

Uses of Post order  
Post order traversal is used to delete the tree. Postorder traversal is also useful to get the postfix expression of an expression tree.

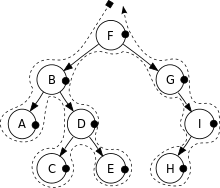
**EXAMPLE:-**



In-order: A, B, C, D, E, F, G, H, I.



Pre-order: F, B, A, D, C, E, G, I, H.



Post-order: A, C, E, D, B, H, I, G, F

**Lab Task: -**

1.Write the code for Depth first search and Breadth first search.

## Conclusion

---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

----------------------------------------------

**(Concerned Teacher/Lab Engineer)**

**Experiment # 13 & 14**

**AVL Tree Data structure**

**Objective:-**

The objective of this session is to understand the concepts of AVL Trees and its balance factor .

**Software Tools:-**

1. Code Blocks with GCC compiler.

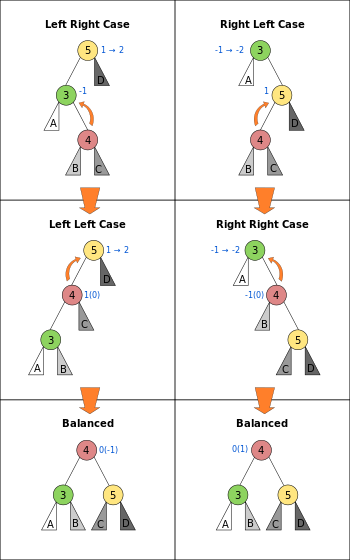
**Theory:-**

In [computer science](https://en.wikipedia.org/wiki/Computer_science), an AVL tree is a [self-balancing binary search tree](https://en.wikipedia.org/wiki/Self-balancing_binary_search_tree). It was the first such [data structure](https://en.wikipedia.org/wiki/Data_structure) to be invented. In an AVL tree, the [heights](https://en.wikipedia.org/wiki/Tree_height) of the two [child](https://en.wikipedia.org/wiki/Child_node) subtrees of any node differ by at most one; if at any time they differ by more than one, rebalancing is done to restore this property. Lookup, insertion, and deletion all take [O](https://en.wikipedia.org/wiki/Big_O_notation)(log *n*) time in both the average and worst cases, where *n* is the number of nodes in the tree prior to the operation. Insertions and deletions may require the tree to be rebalanced by one or more [tree rotations](https://en.wikipedia.org/wiki/Tree_rotation).

### Insertion:-

After inserting a node, it is necessary to check each node starting with the subject node and advancing up to the root for consistency with the [invariants](https://en.wikipedia.org/wiki/Invariant_(computer_science)) of AVL trees: this is called "retracing". This is achieved by considering the **balance factor** of each node, which is defined as the difference in heights of the left and right subtrees.

Thus the balance factor of any node of an AVL tree is in the integer range [-1,+1]. This *balance factor is stored in the node*, but may have to be corrected after an insertion or a deletion, which is also done during retracing. Since with a single insertion the height of an AVL subtree cannot increase by more than one, the temporarily recomputed balance factor of a node after an insertion will be in the range [−2,+2]. For each node checked, if the recomputed balance factor remains in the range [−1,+1] no rotations are necessary. However, if the recomputed balance factor becomes less than −1 or greater than +1, the subtree rooted at this node is unbalanced, and a rotation is needed.

[](https://en.wikipedia.org/wiki/File:AVL_Tree_Rebalancing.svg)

**Description of the Rotations:-**

Let us first assume the balance factor of a node P is 2 (as opposed to the other possible unbalanced value −2). This case is depicted in the left column of the illustration with P:=5. We then look at the left subtree (the higher one), N its root. If this subtree does not lean to the right - i.e. N has balance factor 1 (or, when deletion also 0) - we can rotate the tree (rooted by 5) to the right to get a balanced tree. This is labelled as the "Left Left Case" in the illustration with N:=4. If the subtree does lean to the right - i.e. N:=3 has balance factor −1 - we first rotate the subtree to the left and end up the previous case. This second case is labelled as "Left Right Case" in the illustration.

If the balance factor of the node P is −2 (this case is depicted in the right column of the illustration P:=3) we can mirror the above algorithm. I.e. if the root N of the (higher) right subtree has balance factor −1 (or, when deletion also 0) we can rotate the whole tree to the left to get a balanced tree. This is labelled as the "Right Right Case" in the illustration with N:=4. If the root N:=5 of the right subtree has balance factor 1 ("Right Left Case") we can rotate the subtree to the right to end up in the "Right Right Case".

**Lab Task:-**

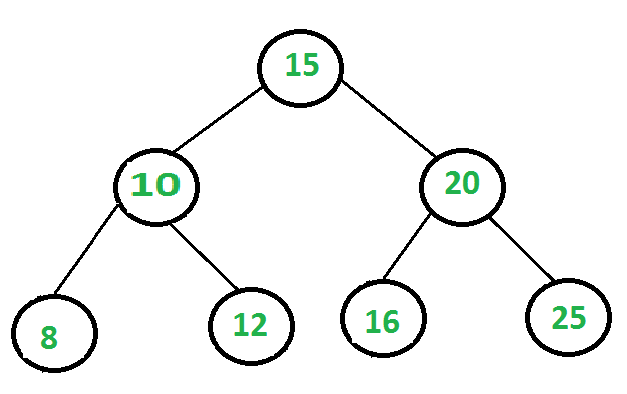
# Task 1

Create the AVL tree by inserting the nodes given bellow:

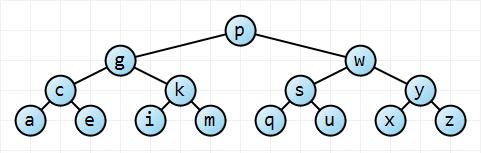
1. 43, 31, 29, 32, 25, 21, 26, 51, 59, 101, 99
2. 5, 4, 6, 2, 9, 7, 11, 14, 17, 18, 81, 16, 19

# Task 2

For the following AVL trees, insert or delete the asked nodes.



|  |  |
| --- | --- |
| Insert 6 | Insert 5 |
|  |  |
| Insert 27 | Insert 31 |
|  |  |
| Delete 6 | Delete 5 |
|  |  |
| Delete 27 | Delete 31 |
|  |  |



|  |  |
| --- | --- |
| Delete p | Delete w |
|  |  |
| Delete y | Delete z |
|  |  |

# Task 3

Write the pseudo-code of insertion of node in AVL

## Conclusion

---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

----------------------------------------------

**(Concerned Teacher/Lab Engineer)**