المادة: هياكل بيانات

مدرس المادة : م . علاء عبدالحسين هاشم

2025-2024

1. Introduction to Data Structures

What is Data Structure?

Whenever we want to work with large amount of data, then organizing that data is very important. If that data is not organized effectively, it is very difficult to perform any task on that data. If it is organized effectively then any operation can be performed easily on that data.

A data structure can be defined as follows:

Data structure is a method of organizing large amount of data more efficiently so that any operation on that data becomes easy.

NOTE

☼ Every data structure is used to organize the large amount of data
☼ Every data structure follows a particular principle
☼ The operations in a data structure should not violate the basic principle of that data structure.

Based on the organizing method of a data structure, data structures are divided into two types.

1. Linear Data Structures
2. Non - Linear Data Structures

1. Linear Data Structures

If a data structure is organizing the data in ***sequential order***, then that data structure is called as **Linear Data Structure.**

Example

1. Arrays
2. Stack
3. Queue
4. List (Linked List)

2. Non - Linear Data Structures

If a data structure is organizing the data **in *random order***, then that data structure is called as **Non-Linear Data Structure.**

Example

1. Tree
2. Graph
3. Heaps, Etc.

2. STACK

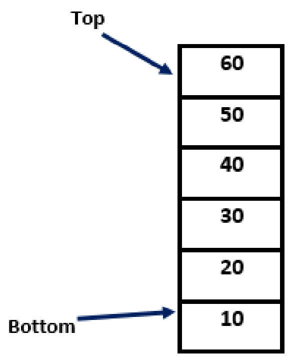
2.1 what is the stack?

Stack: is a linear list of homogeneous elements in which all the insertions and deletions are at one end called **top** for this reason stack is referred to as **Last- In- First-Out** (**LIFO**) lists. Given a stack **S=(10,20,30,40,50,60),** we say that **10** is a bottom element and the **60** is on top of the elements .

Example

Stack (1….6)

[10, 20, 30, 40 ,50 60]



2.2 Stack Representation using array

The simplest method to represent a stack is to use an array, one end of the array is the fixed bottom of the stack while other is the top of the stack. during the execution of the program we should keep tracks of the current position of the ***top*** of the stack.

***Datatype* stackName [size];**

int top=-1;

2.3 Stack's operations:

1. Stack Full: this operation is to check whether the stack is full or not, and it depends on the value of the **top**, this should be checked before the push operation.

```
if  top == size -1
then  stackfull
```

2. Stack Empty: this operation is to check whether the stack is empty or not, and it depends on the value of the **top**, this should be checked before the pop operation.

```
if  top == -1    then  stackempty
```

3. Push: It means insert new element to the stack, and before the insertion we should do the following:

- Check whether **stack** is **FULL**. (**top == size-1**)
- If it is **FULL**, then display **"Stack is FULL!!! Insertion is not possible!!!"** and terminate the function.
- If it is **NOT FULL**, then increment **top** value by one (**top++**) and set stack[top] to value (**stack[top] = value**).

4.Pop: It means delete element from the stack, and the element is always deleted from top position. before the deletion we should do the following:

- Check whether **stack** is **EMPTY**. (**top == -1**)
- If it is **EMPTY**, then display "**Stack is EMPTY!!! Deletion is not possible!!!"** and terminate the function.
- If it is **NOT EMPTY**, then delete **stack[top]** and decrement top value by one (**top--**).

2.4 Stack's algorithms:

1. Push(value) algorithm

```
void pushSt (stack[size], int &top)
{     int value;
      if (top==(size-1)  cout << "stack is full, insertion is not possible";
        else
           {
             cin>>value;
             ++top;
             stack[top]=value;
            }
}
```

2. POP(value) algorithm

```
void popSt (int stack[size],  int  &top)

{     int value;
      if (top == -1) cout<<"stack is empty, deletion is not possible ";
     else
     {
      value= stack[top];
      --top;
     }
 }
```

3. Print Stack algorithm

Display the contents of the stack.

```
 void printSt (int stack[size],  int  top)

{

int i;

for(i=top;i>=0;--i)

cout<<st[i]<<endl<<endl;

}
```

3. QUEUE

3.1 what is the Queue?

Queue is a linear data structure in which the insertion and deletion operations are performed at two different ends. The Insertion is performed at one end and deletion is performed at another end. The insertion operation is performed at a position which is known as '**rear**' and the deletion operation is performed at a position which is known as '**front**'. In queue data structure, the insertion and deletion operations are performed based on

FIFO (First- In- First- Out) principle.

Queue data structure can be defined as follows...

Queue is a linear data structure in which the operations are performed based on FIFO principle.

Example

Queue after inserting 20, 33, 54, 65 and 90.

3.2 Queue Representation using array

The Queue can be represented using one-dimension array, the two integer variables **'front'** and '**rear**' must be initialized both with **'-1'**, as below:

***Datatype* QueueName [size];**
int front=-1, rear=-1;

3.3 Queue's operations:

1. Inserting value into the Queue

It means inserting a new element into the queue. The new element is always inserted at **rear** position.

The following steps should be checked before insert an element into the queue.

1. Check whether queue is **FULL**. (**rear == SIZE-1**)

2 - If it is **FULL**, then display "**Queue is FULL!!! Insertion is not possible** " and terminate.

3 - If it is NOT FULL, then increment rear value by one (**rear++)** and set **queue[rear] = value**.

2. Deleting value from the Queue

It means deleting an element from the queue. The element is always deleted at **front** position. The following steps should be checked before delete an element from the queue.

1. Check whether queue is **EMPTY**. (**front == rear**)

2. If it is **EMPTY**, then display "**Queue is EMPTY!!! Deletion is not possible**" and terminate.

3. If it is **NOT EMPTY**, then increment the **front** value by one (**front ++**). Then display **queue[front]** as deleted element. Then check whether both front and rear are equal (**front == rear**), if it TRUE, then set both front and rear to '-1' (**front = rear = -1**).

2.3 Queue's algorithms:

1.Insertion(value) to queue algorithm

```
void insertQ(int q[size], int &f, int &r)
{
   int value;

   If (r==size -1)    cout<<" Queue is full !! Insertion is not possible \n";
    else
    {
       cin>>value;
       ++r;
       q[r]=value;
   }
  If (f == -1)
      f=0;
}
```

2. Deleting (value) from the Queue algorithm

```
void deletQ(int q[size], int &f, int &r)
{
      int value;

         if (f == -1)    cout<<"under flow !! Queue is empty deletion is not possible \n";
          else
             if (f == r)
              {
                 value=q[f];
                 f=-1
                 r=-1;
              }
         else
             {
               value=q[f];
               ++f;
              }
}
```

3. Prints() the elements of a Queue

```
void printq (int q[6], int f, int r)
{
     int i;
      if (r==-1)      cout<< "Queue is empty nothing to print!!" ;
      else
         for(i=f; i<=r; i++)
          cout <<q[i]<<" ";

}
```

4. Circular Queue

4.1 What is Circular Queue

In a normal Queue Data Structure, we can insert elements until queue becomes full. But once if queue becomes full, we cannot insert the next element until all the elements are deleted from the queue.
For example consider the queue below:
After inserting all the elements into the queue, Queue is full.

Now consider the following situation after deleting four elements from the queue, the queue is full even four elements are deleted.

This situation also says that Queue is Full and we can not insert the new element because, '**rear**' is still at last position. In above situation, even though we have empty positions in the queue we can not make use of them to insert new element. This is the major problem in normal queue data structure. To overcome this problem, we use circular queue data structure.

A Circular Queue can be defined as follows...

Circular Queue is a linear data structure in which the operations are performed based on **FIFO (First -In-First- Out)** principle and the last position is connected back to the first position to make a circle.

Graphical representation of a circular queue is as follows...

4.2 Implementation of Circular Queue

circular queue data structure is implementing by using one-dimension array, and define two integer variables **'front'** and '**rear**' and initialize both with **'-1'** as below:

Datatype QueueName [size];

(**int front = -1, rear = -1**)

4.2 CQueue's operations:

4.2.1. Inserting value into the CQueue

It means inserting a new element into the cqueue. The new element is always inserted at **rear** position. The following steps should be checked before insert an element into the cqueue.

1. Check whether cqueue is **FULL. ((rear == SIZE-1 && front == 0) || (front == rear+1)).**

2. If it is **FULL**, then print **"Queue is FULL!!! Insertion is not possible!!!"** and terminate.

3. If it is **NOT FULL**, then check if there is an empty place at the front of the cqueue (**rear == SIZE - 1 && front > 0) if it is TRUE, then set rear = 0,** set **CQueue[rear] = value.** (this is rotation state during the insertion).

4. Otherwise increment **rear** value by one **(rear++)**, set **CQueue[rear] = value**

5. Check if **'front == -1' if it is TRUE, then set front = 0**.

4.2.2 Insertion(value) to CQueue algorithm

```
void insertCQ (int CQ[size], int &F, int &R)
 {  int value;
    if((R==size -1 && F==0 || F== R+1)  cout<<"Queue is FULL!!! Insertion is not possible";
    else if (R== size -1 && F>0)
              {
                 cin>>value;
                 R=0;
                CQ[R]=value;
               }
           else
             {    cin>>value;
                   R++;
                   CQ[R]= value;
              }
           if(F==-1)
           F=0;
}
```

4.2.3 Deleting value from the CQueue

It means deleting an element from the cqueue. The element is always deleted at **front** position. The following steps should be checked before delete an element from the cqueue.

1. Check whether queue is **EMPTY**. **(front == -1 && rear == -1)**

2. If it is **EMPTY**, then display "**CQueue is EMPTY!!! Deletion is not possible**" and terminate.

3. If it is **NOT EMPTY**, then check whether both front and rear are equal (**front == rear**), if it **TRUE**, this means there is only one element in the queue so delete the element and set both front and rear to '-1' (**front = rear = -1**).

4. check if the **(front=size -1)** then delete the element and set the **front=0 (**this is rotation state during the deletion).

4.2.4 Deletion(value) from CQueue algorithm

```
void del(int CQ[6],int &F,int &R)
{  int value ;
   if(F==-1)  cout<<" CQueue is EMPTY!!! Deletion is not possible ";
   else  if  (F==R)
         {   value =CQ[F];
             R=F=-1;
          }
        else if (F==size-1  &&  F>R)
                {
                    value =CQ[F];
                    F=0;
                  }
              else
                  {
                      value =CQ[F];
                       F++;
                    }
                   }
```

4.2.5 Print () - prints the elements of a Circular Queue

```
void printCQ (int CQ[size] , int F , int R)
{     int i;
       if(R>=F)
         {
          for(i=F;i<=R;i++)
          cout<<CQ[i]<<" ";
          cout<<'\n';
        }
    else
        {
          for(i=F;i<=5;i++)
          cout<<CQ[i]<<" ";
          for(i=0;i<=R;i++)
          cout<<CQ[i]<<" ";
          cout<<'\n';
        }
}
```

5. Single Linked List

5.1 What is Linked List?

When we want to work with an unknown number of data values, we use a linked list data structure to organize that data. **The linked list** is a linear data structure that contains a sequence of elements such that each element links to its next element in the sequence. Each element in a linked list is called "**Node**".

What is Single Linked List?

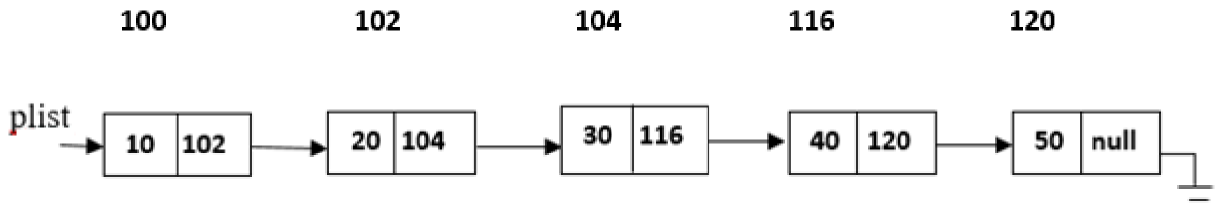
Single linked list is a sequence of elements in which every element has link to its next element in the sequence.

In any single linked list, the individual element is called as "**Node**". Every "**Node**" contains two fields, **data field**, and the **link field**. The data field is used to store actual **value** of the node and link field is used to store the **address** of **next** node in the sequence.
The graphical representation of a node in a single linked list is as follows...

Important Note:

- In a single linked list, the address of the first node is always stored in a reference node known as "**head**". (we always refer to it as "**plist**")
- Always next field (link part) of the last node must be **NULL**.

Example



5.2 Declaration of linked list

Here a declaration example of linked list:

declare the null value as below at the header of the program

```
#define null 0;
struct node
   {
         int info;
         struct node * next;

   }; typedef struct node* nodeptr;
```

5.3 Operations on Single Linked List

The following operations are performed on a Single Linked List:

- Creation
- Insertion
- Deletion
- Display

5.3.1 Create new linked list

Before we implement any operations, first we should create the first node in the list, as in the below:

```
void creatlist(nodeptr plist)
{   nodeptr   p,q;
    p=new  node;
    cin>>p->info;
    p->next=null;
    plist=p;
}
```

5.3.2 Insertion

In a single linked list, the insertion operation can be performed in three ways. as follows:

- Inserting at Beginning of the list
- Inserting at End of the list
- Inserting at Specific location in the list

1. Inserting at Beginning of the list

```
void insertBegin(nodeptr plist)

{
    nodeptr p;
    p=new node;
    cin>>p->info;
    p->next=plist;
    plist =p;
}
```

2. Inserting at End of the list

```
void insertend(nodeptr plist)
{
      nodeptr p,q;
       q=plist;
      while (q->next!= 0)
          q=q->next;
     p=new node;
     cin>>p->info;
     p->next=null;
     q->next=p;

 }
```

3. Inserting at Specific location in the list (Between two nodes)

```
void insertBetwen( nodeptr  plist )
{
         nodeptr p,a,b;
         int i,L;
        cout<<"enter the number of location=";
        cin>>L;
        p=new node;
        cin>>p->info;
        a=plist;
        for ( i=2; i<L; i++ )
           a=a->next;
        b=a->next;
        a->next=p;
        p->next=b;
 }
```

5.3.3Deletion

In single linked list, the deletion operation can be performed in three ways, as follows:

- Deleting from Beginning of the list
- Deleting from End of the list
- Deleting a Specific Node

1. Deleting from Beginning of the list

```
void deletBegin(nodeptr plist)
{
        nodeptr p;
        p=plist;
        plist=plist->next;
        free(p);
}
```

2.Deleting from End of the list

```
void deletEnd( nodeptr  plist )
{
       nodeptr q,p;
       p=plist;
       q=plist;
       while ( p->next != null )
         {    q=p;
              p=p->next;
        }
      free(p);
     q->next=null;
}
```

3.Deleting a Specific Node from the list

```
void deletBetwen  (nodeptr  plist )
{
       nodeptr p,b,a;
       int L,i;
       cout<<"enter the number of location= ";
       cin>>L;
       b=plist;
       for(i=2;i<L;i++)
       b=b->next;
       p=b->next;
       a=p->next;
       free(p);
       b->next=a;
 }
```

5.3.4 Displaying a Single Linked List

```
void DisplayList(nodeptr plist)
{       nodeptr q;
        If (plist == null)
         {
              cout<<("\n List is Empty \n");
         }
       else
       q=plist;
       while(q->next != 0)
         {
               cout<<q->info<<" " ;
               q=q->next;

         }
      cout<<q->info<<" ";
      cout<<'\n';
}
```