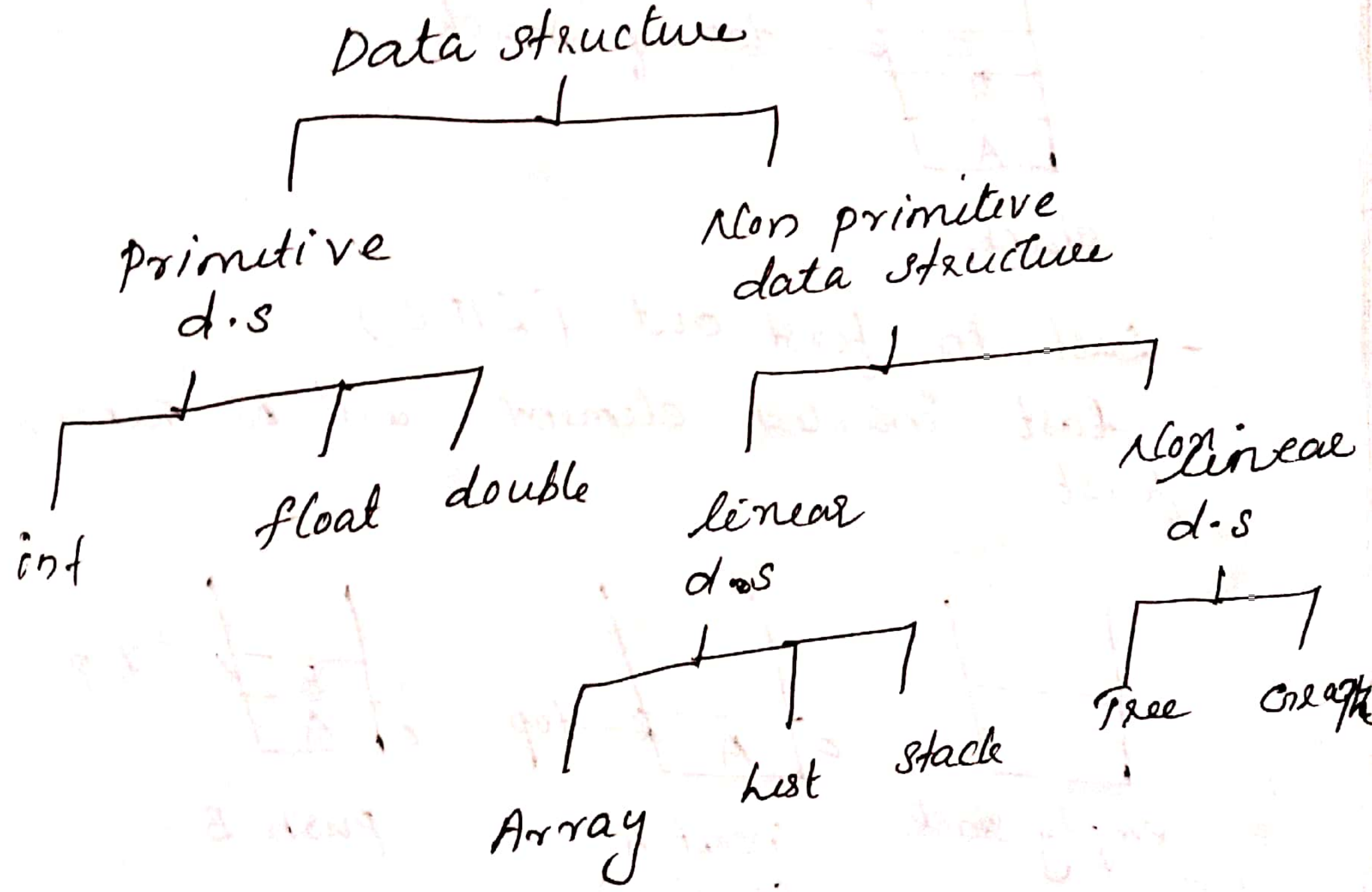
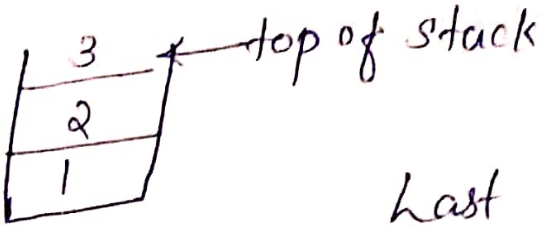
Data Structures Introduction

Data structure → way of organizing data in memory and their relationships



Stack

linear data structure for storing data.
based on LIFO principle (Last in first out)



Last inserted element will be deleted first

It has one end called top of stack

top of stack points to the last element of stack.

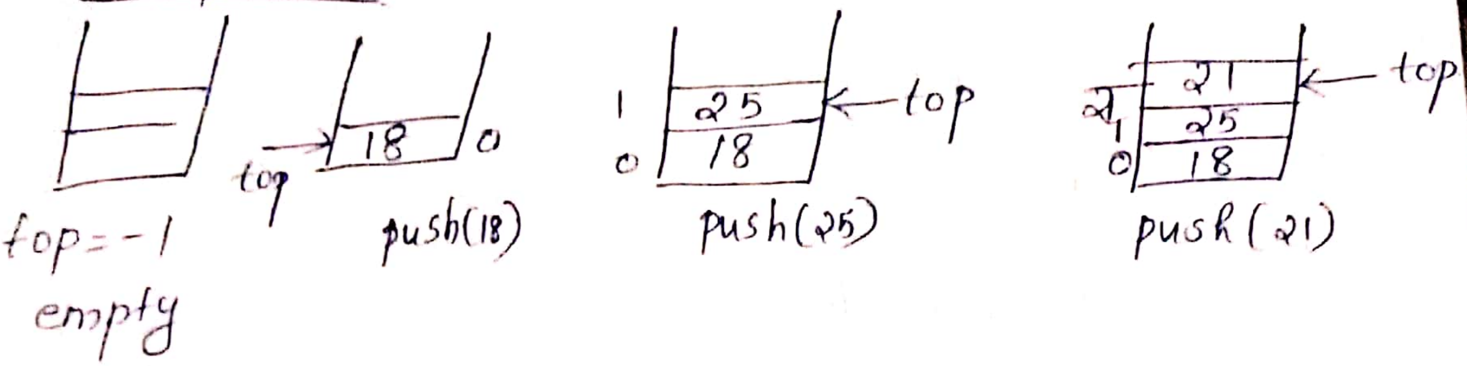
Two operations
1) push (insert an element into stack)
2) pop (delete an element into stack)

insertion and deletion done through one end - top

Implementation
- using array
- using linked list

Array Implementation

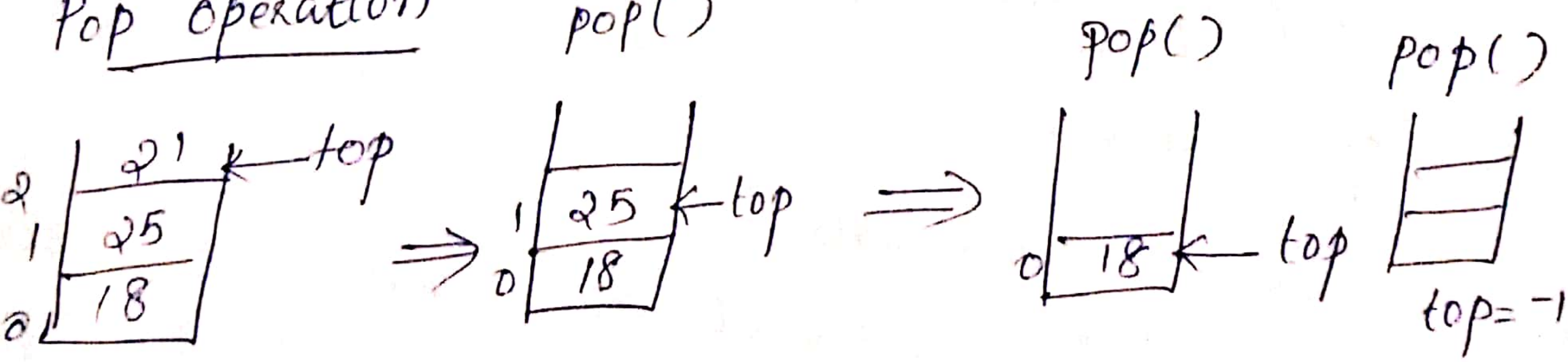
Push operation



Algm

if (top = ~~max~~ max − 1)
    Print Stack is ~~empty~~ full
    exit
else
    top = top + 1
    a[top] = item

Pop operation



Algm

if (top = -1)
    print stack is empty
else
    item = a[top]
    top = top - 1

Stack application

Real life

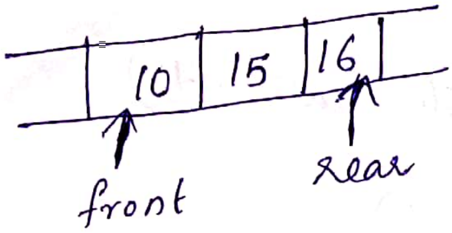
pile of books
plate trays

Computer Science

program execution stack
Evaluating expression.

Queue

- linear data structure for storing data.
- based on FIFO principle (First in First out)
  ↓
  first inserted element will be deleted first



Queue has two ends

front end and rear end

front end points to the first element of queue.

rear end points to the last element of queue.

insertion and deletion

insertion done through ~~front~~ rear end and deletion done through ~~rear~~ front end.

Implementation

↳ using array

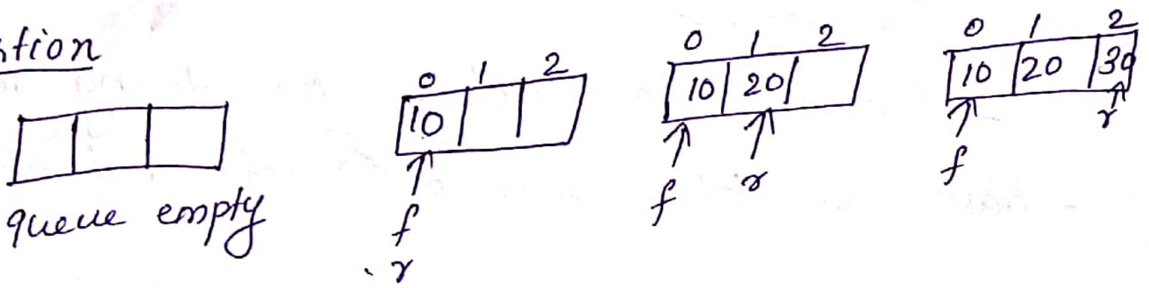
→ using linked list

Array Implementation

~~Before Inserte~~

Array Implementation

Insertion

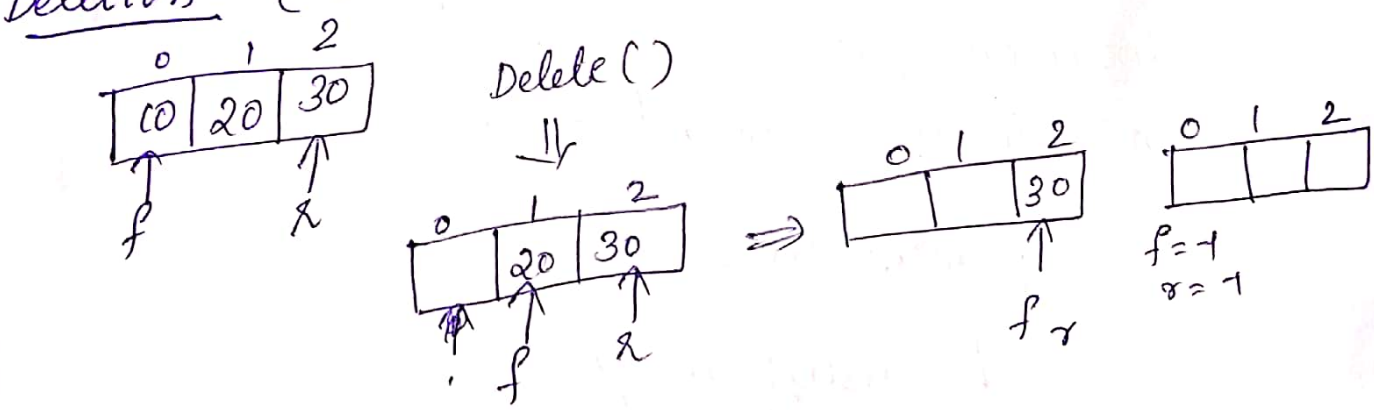


f = -1 queue empty
r = -1

Algm (done through rear end)

```
if (rear = max-1)
    print queue is ~~empty~~ full
    exit
else
  { rear = rear + 1
    a[rear] = item
  }
if (front = -1)
    front = 0.
```

Deletion (done through front end)



Algm

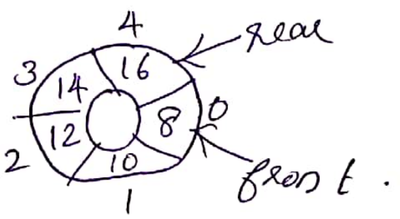
```
if f = -1
    print queue is empty
    exit
else
  { item = a[front]
```

```
if (front = rear)
    front = rear = -1
else
front = front + 1
}
```

Circular queue

Disadvantage of queue.

If the rear reaches the end of the queue, nomore items can be inserted although the items from the front of the queue have been deleted and there is a space in queue. To solve this problem we use circular queue

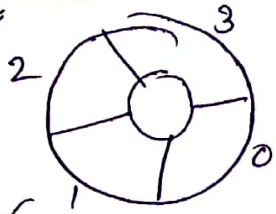


Circular Queue insertion.

Algm

```
if (front = -1 && rear = -1)
{
  front = rear = 0;
  a[rear] = item
}
else if ((rear+1 % N) == front)
{
  print queue is full
}
else
{ rear = (rear + 1) % N
  a [rear] = item
}
```

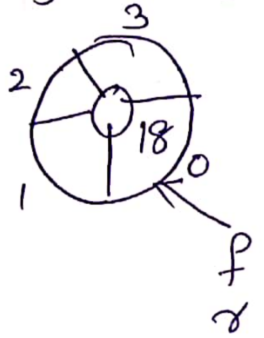
I



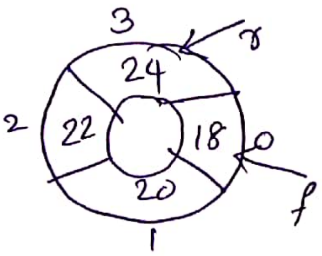
$f = -1$ } queue empty
$r = -1$

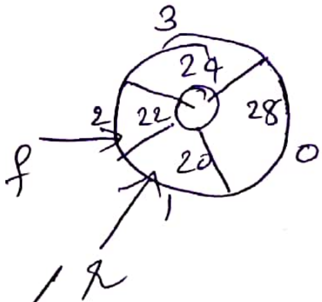
so $r = \cancel{r+1}$ 0
$f = 0$.

$a[r] = item$



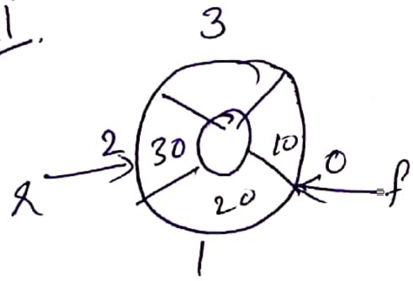
II

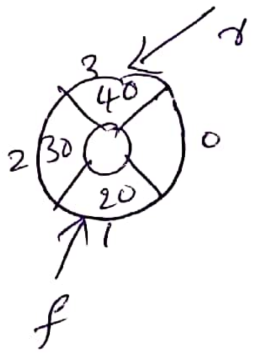




queue full.

III.





$rear = (r+1) \% N$

$r = (2+1) \% 4$

$3 \% 4 = \underline{\underline{3}}$

$r = (r+1) \% N$

$r = (3+1) \% 4$

$4 \% 4 = \underline{\underline{0}}$

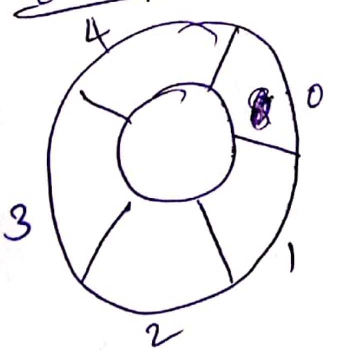
Circular queue deletion

Algorithm

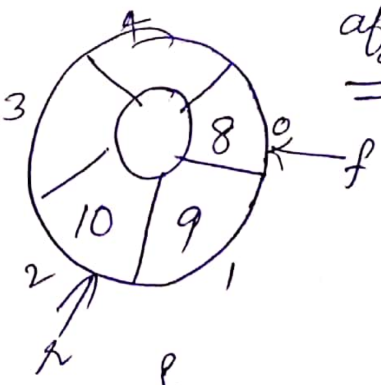
```
If (front = -1)
    print queue is empty
  exit

else
{
    Item = a[front]
        If (front == rear)
            front = rear = -1
        else
            front = (front + 1) % N
}
```

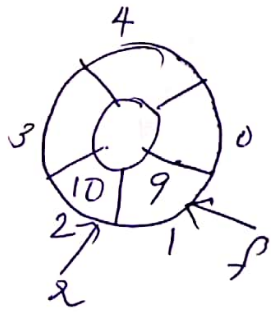
Example



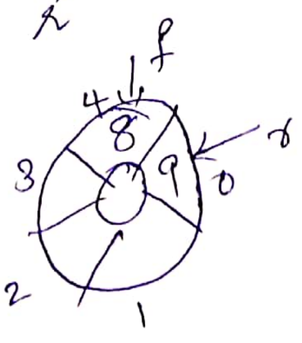
$f = -1$
$r = -1$

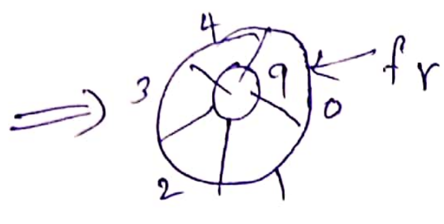


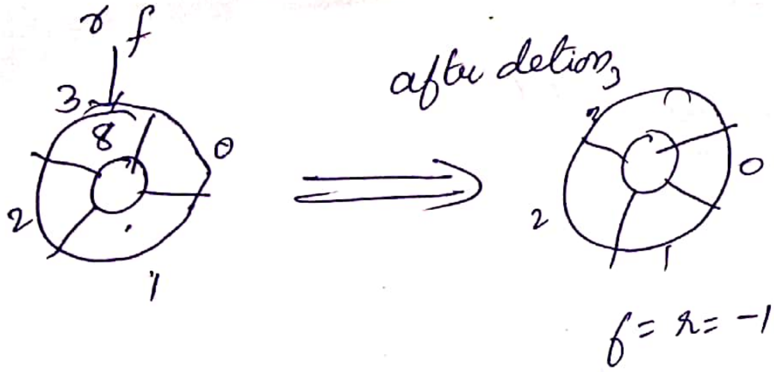
after deletion



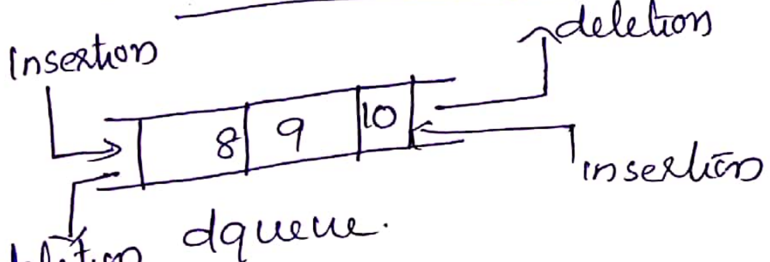
$f = (f+1) \% N$
$= (0+1) \% \cancel{5} = 1$







Double Ended Queue



dqueue.

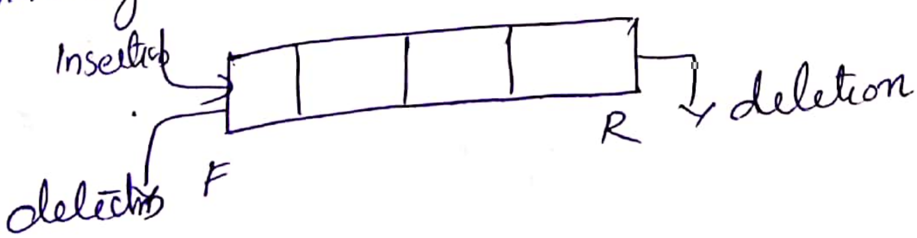
In double ended queue, insertion and deletion takes place in both ends (means front & rear)

Two types of dequeue are

1. Input restricted dequeue
2. output restricted dqueue.

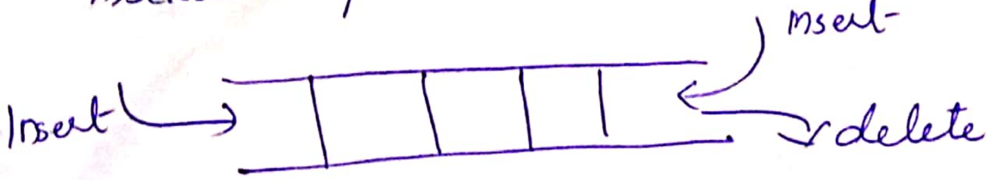
Input restricted dequeue

Insertion done through one end and deletion through both ends.



Output restricted Dqueue

Elements can be removed at one end only.
insertion possible through both ends.

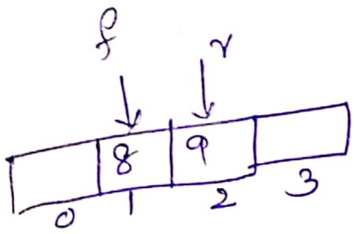


operations in Dequeue

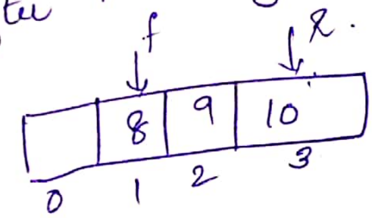
1. Insert element at rear end
2. Insert element at front end
3. Delete element at rear end
4. Delete element at front end

Insert rear (same as queue)

```
if (rear = max -1)
   print queue is full
   exit
else
{
   rear = rear +1
   a[rear] = item
   if (front = -1)
      froont = 0
}
```

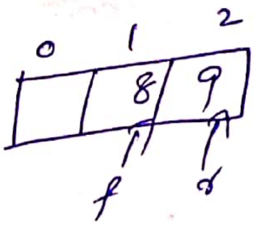


After inserting 10



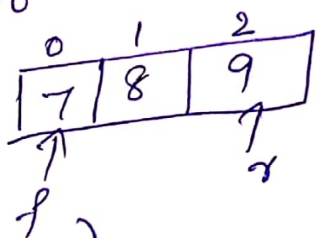
Insertion at front

If (front <= 0)
print (Cannot add item at front end
exit



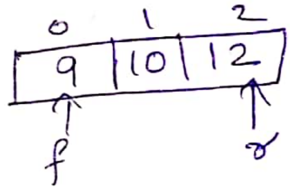
else
front = front - 1
a[front] = item

After inserting 7



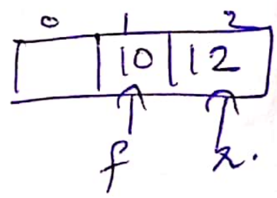
Deletion from front (Same as queue)

If (front == -1)
print queue is empty
exit



else
{
item = a[front]
If (front = rear)
front = rear = -1
else
front = front + 1
}

After deletion



Complexity of an Algorithm

* Algorithm is a ~~step~~ finite step by step instructions for solving a particular problem.

Complexity of Algorithm (Evaluation of algorithms)

It is the function which gives the running time or space in terms of input size.

Two types of complexity

1. Time complexity
2. Space complexity

Evaluating algorithms

Algorithms can be evaluated based on their performance.

Performance evaluation can be divided into

1. Performance Analysis
2. Performance Measurement

Performance Analysis

1) Time complexity
2) Space complexity

Space complexity

of an algm is the amount of memory it need to run to completion

Space needed is sum of two components

1) fixed part
2) variable part

Fixed part

independent of input and output characteristics

It includes

- instruction space
- space for constants
- Space for variable.

Variable part

- space needed by referenced variable
- Recursion stack space etc

$S(P) = c + S_p$

p is an algm

c - fixed space

S_p is the variable space.

eg. - ① Algm . Sum (a, n)

```
{
    s = 0
    for i = 0 to n-1
        s = s + a[i]
    return s;
```

mn = [illegible]

space for n - 1
Space for i - 1
Space for s - 1
space for array (a) of size n is n
So total space needed is $\underline{n+3}$

```
void main()
eg:- { int x, y, z, sum
Printf ("Enter 3 nos");
Scanf ("%d %d %d", &x, &y, &z);
Sum = x+y+z;
printf ("The sum = %d", sum);
}
```

space for variables x, y, z, & sum is 1

So Total space needed is 1+1+1+1 = 4

```
eg:- Algm sum (a,n)
{
  s:=0
  for (i:=0 to n-1)
    for (j:=0 to m-1)
      s:= s+a[i][j]
  return s;
}
```

— space for n → 1
s → 1
i → 1
j → 1
m → 1
array a[i][j] → nm

Total = $\underline{nm+5}$

Time complexity

- how much time needed for the completion of a pgm.

$$T(P) = C + T_p$$

C – compile time
T_p – Run time.

For calculating time complexity we use a method called frequency count i.e counting the no of steps.

for eg:- Comments – 0 steps
Assignment statement – 1 step
Conditional statement – 1 step
Loop condition for n nos – (n+1) steps
Body of loop – n steps.

frequency count

eg:- sum = 0 ⟶ 1

for (i=1; i<n; i++) ⟶ n+1

{

sum = sum + a[i] ⟶ n

}

2n+2

~~Frequency~~

Time complexity is 2n+2

2. Algm Sum (a[ ], n, m)

{

for i=1 to n do

{ for j=1 to m do

{ s = s + a[i][j]

}

} return s;

}

Scanned with CamScanner

frequency count

Algm sum (a[ ], n, m) —

{ —

for i=1 to n do ——— n+1

for j=1 to m do ——— n(m+1)

s = s + a[i][j] ——— nm

return s; ——— 1

}

~~2nm~~

$= \underline{\underline{2nm + 2n + 2}}$

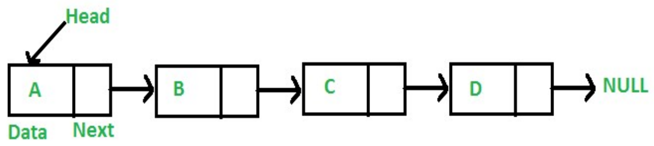
Scanned with CamScanner

Module II

Abstract and Concrete Data Structures- Basic data structures – vectors and arrays. Applications, Linked lists:- singly linked list, doubly linked list, Circular linked list, operations on linked list, linked list with header nodes, applications of linked list: polynomials

Linked List

Like arrays, Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at continuous location; the elements are linked using pointers.



In linked list the element is represented by **nodes**. Each node contains a data field and a link to the next node in the list.

Why Linked List?

Arrays can be used to store linear data of similar types, but arrays have following limitations.

1) The size of the arrays is fixed:

2) Inserting a new element in an array of elements is expensive, because we have to shift the elements

For example, id[] = [1000, 1010, 1050, 2000, 2040].

And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 .

Deletion is also expensive .For example, to delete 1010 in id[], everything after 1010 has to be moved.

Advantages over arrays

1) Dynamic size

2) Ease of insertion/deletion

Drawbacks of linked list:

1) Random access is not allowed. We have to access elements sequentially starting from the first node.

2) Extra memory space for a pointer is required with each element of the list.

3) Not cache friendly. Since array elements are contiguous locations, there is locality of reference which is not there in case of linked lists.

Application of linked list

- Polynomial addition
- Memory allocation
- Used to implement stack and queues
- Used to implement graphs
- Implementing hash tables

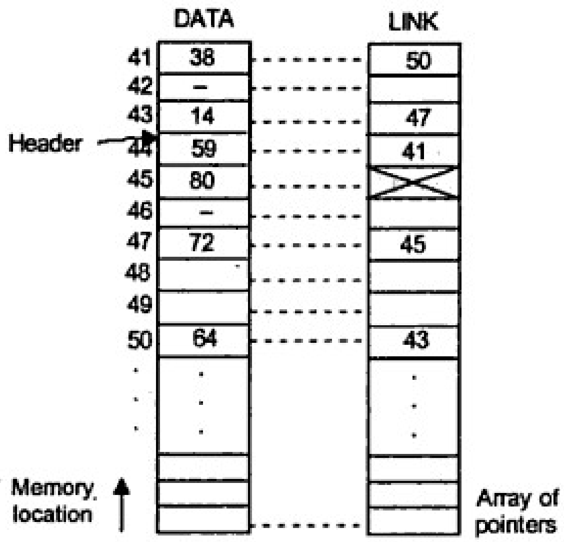
Representation of linked list in memory

1. **Static representation using array**

2. **Dynamic representation using free pool of storage**

Static representation

Here we use two arrays. One for storing data and another for links



Static representation of a single linked list using arrays.

Dynamic representation

The efficient way of representing a linked list is using free pool of storage. In this method, there is a *memory bank* (which is nothing but a collection of free memory spaces), and a *memory manager* (a program, in fact). During the creation of linked list, whenever a node is required the request is placed to the memory manager; memory manager will then search the memory bank for the block requested and if found grants a desired block to the caller. Again, there is also another program called *garbage collector*, it plays whenever a node is no more in use; it returns the unused node to the memory bank. It may be noted that memory bank is also a list of memory space that is available to a programmer. Such a memory management is known as *dynamic* memory management. Dynamic representation of linked list uses the dynamic memory management policy.

<u>Representation of a node in linked list</u>

A linked list is represented by a pointer to the first node of the linked list. The first node is called head. If the linked list is empty, then value of head is NULL.

Each node in a list consists of at least two parts:

1) data

2) Pointer to the next node

In C, we can represent a node using structures. Below is an example of a linked list node with an integer data.

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

Link is a pointer which points to struct node

Types of Linked List

Following are the various types of linked list.

- **Single Linked List** − The list can be traversed forward direction only.
- **Doubly Linked List** -Pointers exist between adjacent nodes in both directions.
    - The list can be traversed either forward or backward.
- **Circular Linked List** − The pointer from the last element in the list points back to the first element.
- **Circular doubly linked list** – both features of doubly and circular list

Singly linked list

In singly linked list is an ordered collection of finite , homogeneous data elements called nodes where the linear order is maintained by means of links or pointers.

Here, N1, N2, . . ., N6 are the constituent nodes in the list. HEADER is an empty node (having data content NULL) and only used to store a pointer to the first node N1. Thus, if one knows the address of the HEADER node from the link field of this node, next node can be traced and so on. This means that starting from the first node one can reach to the last node whose link field does not contain any address rather a null value. Note that in a single linked list one can move from left to right only; this is why a single linked list is also alternatively termed as *one way* list.

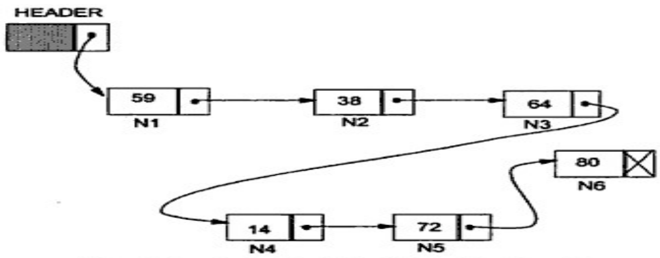


Fig. 3.2 A single linked list with 6 nodes.

Creation of a node

p= (struct node *) malloc (size of struct node));

p->data=8;

p->link=NULL;

Algorithm for inserting a node to the List

Insert node at beginning

Algorithm Insert begin

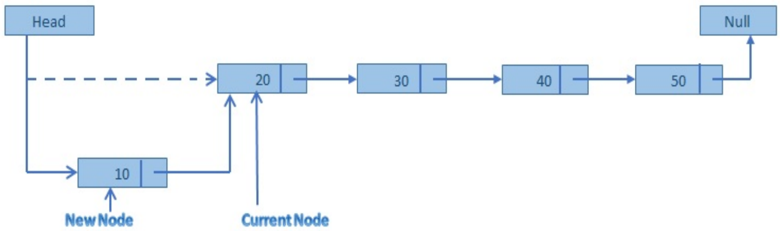
// initially set start as NULL

    Read the item
    p= (struct node *) malloc (size of struct node));
    p->data= item;
    p-> link=NULL;
    if(start=NULL)
        start=p

else

p->link=start

start=p



Insert node at last

Algorithm Insert end

Read the item

p= (struct node *) malloc (size of struct node));

p->data= item

p-> link=NULL

if(start=NULL)

start=p

else

temp=start

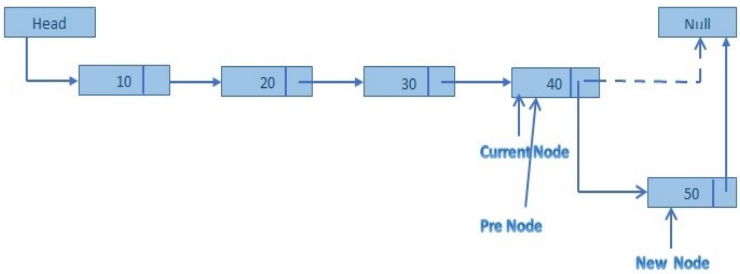
while(temp->link!=NULL)

{

temp=temp->link

}

temp->link=p;



Insert a node at middle

Algorithm Insert middle

Read the item and position

p= (struct node *) malloc (size of struct node));

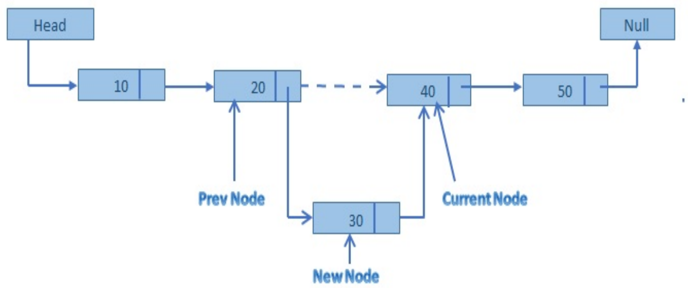
p->data= item

p-> link=NULL

temp=start

while(i<position-1)

{
            temp=temp->link
            i++
}
p->link=temp->link
temp->link=p



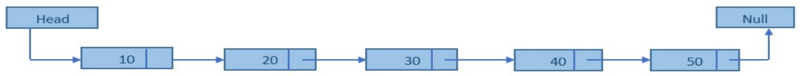
Deletion of a node from the List

There are three situations for Deleting element in list.

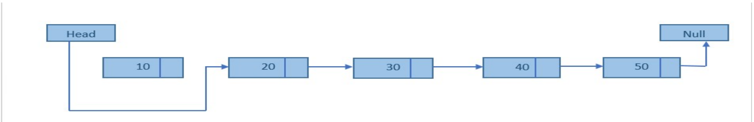
1. Deletion at beginning of the list.
2. Deletion at the middle of the list.
3. Deletion at the end of the list.

Deletion at beginning of the list.

Before Deletion



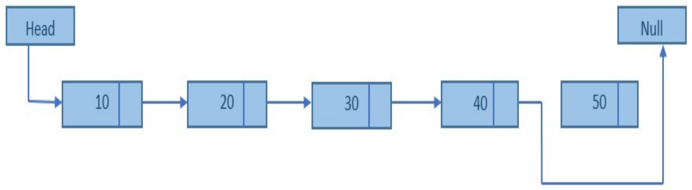
After Deletion



Algorithm Delete at begin
if(start==NULL)
print linked list is empty
Exit
Else
            temp=start
            start=start->link
delete(temp)

Deletion at the end of the list.

After Deletion



Algorithm Delete at end

if(start==NULL)

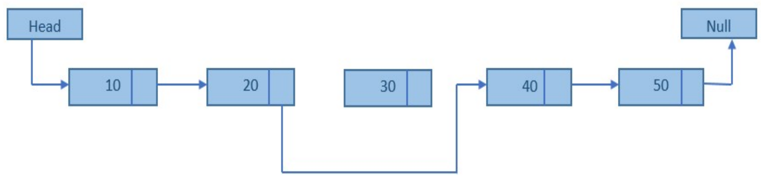
print linked list is empty

Exit

Else

```
        temp=start
        while(temp->link!=NULL)
        {
                temp1=temp
                temp=temp->link
        }
delete (temp)
temp1->link=NULL
```

Deletion at the middle of the list.



Algorithm Delete at middle

```
if(start==NULL)
        print linked list is empty
Exit
else
{
        temp=start
        while(i<pos-1)
        {
                temp1=temp
                temp=temp->link
```

```
            i++
        }
        temp1->link=temp->link
        delete (temp)
}
```

Deletion of a node based on their data

Assume x be the data to be deleted

```
  temp =start
  if (temp->link=NULL)
  delete temp
  else
   {
     while (temp-> data!=x)
   {
           temp1= temp
           temp=temp->link
     }
   temp1->link= temp->link
   }
```

Traversing a list

Algorithm for traversing

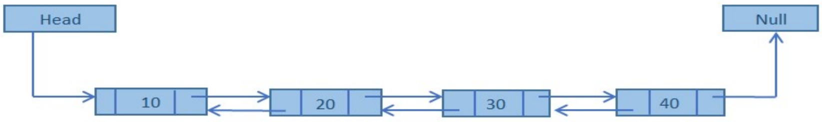
```
temp=start
While(temp!=NULL)
{
print temp->data
temp=temp->link
}
```

Doubly Linked list

In this type of liked list each node holds two-pointer field. Pointers exist between adjacent nodes in both directions. The list can be traversed either forward or backward.

- Doubly Linked List are more convenient than Singly Linked List since we maintain links for bi-directional traversing We can traverse in both directions and display the contents in the whole List.

Each Node contains two fields, called Links, that are references to the previous and to the Next Node in the sequence The previous link of the first node and the next link of the last node points to NULL.



Representation of a node in doubly linked list

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

Doubly Linked list Insertion

Three ways

-Insertion at beginning

- Insertion at the end of the list

- Insertion at anywhere in the list

Insertion at beginning

Algorithm_ Insert begin

```
        // node creation
        p=  (struct node *) malloc (size of struct node));
        p->data= item
        p-> prev=NULL
        p->next= NULL
if(start=NULL)
      start=p
else
        p->next=start
         start->prev=p
         start=p
```

Insert node at last

Algorithm Insert end

```
p=  (struct node *) malloc (size of struct node));
p->data= item
p-> prev=NULL
p->next=NULL
if(start=NULL)
        start=p
```

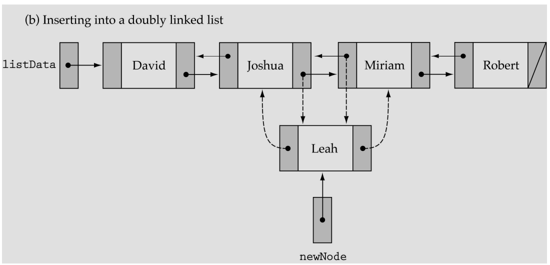
```
else
        temp=start
        while(temp->next!=NULL)
        {
                temp=temp->next
        }
        temp->next=p;
        p->prev=temp
```

Insert a node at middle

Algorithm Insert middle

Read the item and position

```
p=  (struct node *) malloc (size of struct node));
p->data= item
p-> next=NULL
p->prev=NULL
temp=start
while(i<position)
{
        temp=temp->next
        i++
}
temp1=temp->prev
p->prev=temp1
temp1->next=p
p->next=temp
temp->prev=p
```



<u>Deletion of a node from the Doubly Linked List</u>

There are three situations for Deleting element in list.

1. Deletion at beginning of the list.
2. Deletion at the middle of the list.
3. Deletion at the end of the list.

Deletion at beginning of the list.

```
Algorithm Delete at begin ()
if(start==NULL)
print linked list is empty
Exit
else
        temp=start
        start=start->next
        start->prev=NULL
        delete(temp)
```

<u>Deletion at the end of the list.</u>

```
Algorithm Delete at end ()
if(start==NULL)
print linked list is empty
Exit
else
        temp=start
        while(temp->next!=NULL)
        {
                temp=temp->next
        }
        temp1=temp->prev
        temp1->next=NULL
        delete (temp)
```

<u>Deletion at the middle of the list.</u>

```
if(start==NULL)
        print linked list is empty
Exit
else
{Read the position pos
        temp=start
        while(i<pos)
        {
                temp=temp->next
                i++
        }
```

```
        temp1=temp->prev
        temp2=temp->next
        delete (temp)
        temp1->next=temp2
        temp2->prev=temp1
}
```

Traversing a list

Algorithm for traversing

```
temp=start
While(temp!=NULL)
{
print temp->data
temp=temp->next
}
```

Circular linked list

A linked list where the last node points the header node is called *circular* linked list. Figure 3.8 shows a pictorial representation of a circular linked list.

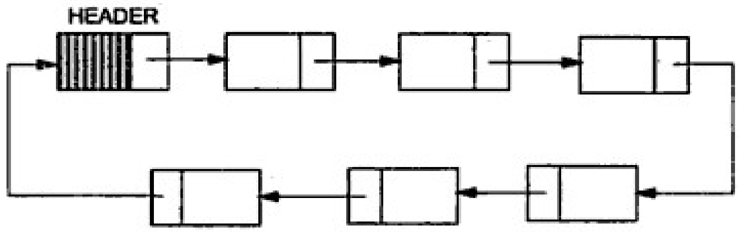


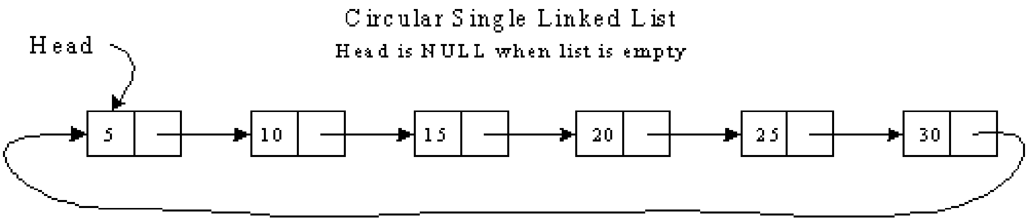
Fig. 3.8 A circular linked list.

Advantages of Circular Linked Lists:

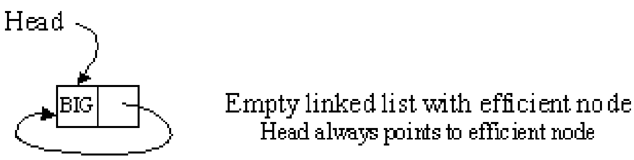
1) Any node can be a starting point.

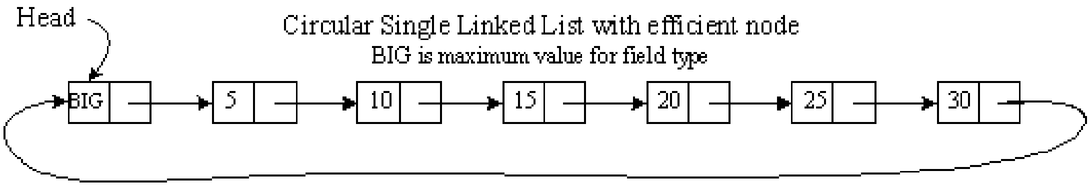
2) Useful for implementation of queue.

3) Circular lists are useful in applications to repeatedly go around the list



Circular linked list





Doubly linked list

Linked Implementation of stack

The operation of adding an element to the front of a linked list is quite similar to that of pushing an element on to a stack.

A stack can be accessed only through its top element, and a list can be accessed only from the pointer to its first element.

Similarly, removing the first element from a linked list is analogous to popping from a stack.

A linked-list is somewhat of a dynamic array that grows and shrinks as values are added to it and removed from it respectively.

Rather than being stored in a continuous block of memory, the values in the dynamic array are linked together with pointers.

Each element of a linked list is a structure that contains a value and a link to its neighbor.

The link is basically a pointer to another structure that contains a value and another pointer to another structure, and so on.

If an external pointer *p* points to such a linked list, the operation *push(p, t)* may be implemented by

```
f=getnode();
info(f)=t;
next(f)=p;
p = f;
```

The operation t = pop(p) removes the first node from a nonempty list and signals underflow if the list is empty

```
if(empty(p))
    {
printf('stackunderflow');
exit(1);
 }
 else{
f=p;
p=next(f);
  t=info(f);
  freenode(f);
      }
```

The *getnode* operation may be regarded as a machine that manufactures nodes. Initially there exist a finite pool of empty nodes and it is impossible to use more than that number at a given instant . If it is desired to use more than that number over a given period of time, some nodes must be reused. The function of *freenode* is to make a node that is no longer being used in its current context available for reuse in a different context.

The list of available nodes is called the ***available list.*** When the available list is empty that is all nodes are currently in use and it is impossible to allocate any more, overflow occurs.

Linked Implementation of Queue

Queues can be implemented as linked lists. Linked list implementations of queues often require two pointers or references to links at the beginning and end of the list.

Using a pair of pointers or references opens the code up to a variety of bugs especially when the last item on the queue is dequeued or when the first item is enqueued.

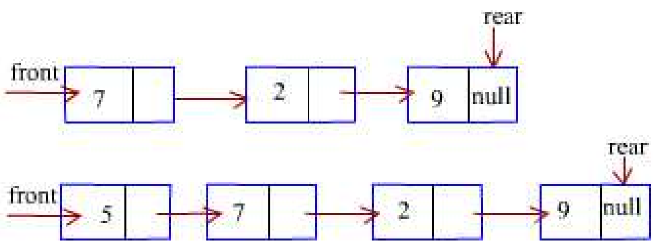
In a circular linked list representation of queues, ordinary 'for loops' and 'do while loops' do not suffice to traverse a loop because the link that starts the traversal is also the link that terminates the traversal.

The empty queue has no links and this is not a circularly linked list. This is also a problem for the two pointers or references approach.

If one link in the circularly linked queue is kept empty then traversal is simplified. The one empty link simplifies traversal since the traversal starts on the first link and ends on the empty one.

Because there will always be at least one link on the queue (the empty one) the queue will always be a circularly linked list and no bugs will arise from the queue being intermittently circular.

Let a pointer to the first element of a list represent the*front* of the queue. Another pointer to the last element of the list represents the *rear* of the queue as shown in fig. illustrates the same queue after a new item has been inserted



Under the list representation, a queue *q* consists of a list and two pointers, *q.front* and *q.rear*.

The operations are insertion and deletion. Special attention is required when the last element is removed from a queue.

In that case ,*q.rear* must also be set to *null*, Since in an empty queue both *r.front* and *q.rear* must be *null*.

The pseudo code for deletion is below:

```
if(empty(q))
{
printf("QueueisUnderflow");
exit(1);
 }
 f=q.front;
 t=info(f);
 q.front=next(f);
   if(q.front==null)
 q.rear=null;
 freenode(f);
return(t);
```

The operation insert algorithm is implemented

```
 f=getnode();
 info(f)=x;
 next(f)=null;
      if(q.rear==null)
       q.front=f;
      else
    next(q.rear)=f;
     q.rear = f;
```

Circular Linked Lists

In linear linked lists if a list is traversed (all the elements visited) an external pointer to the list must be preserved in order to be able to reference the list again.

Circular linked lists can be used to help the traverse the same list again and again if needed. A circular list is very similar to the linear list where in the circular list the pointer of the last node points not NULL but the first node.

In a circular linked list there are two methods to know if a node is the first node or not.

Either a external pointer, list , points the first node or A header node is placed as the first node of the circular list.

The header node can be separated from the others by either heaving a sentinel value as the info part or having a dedicated flag variable to specify if the node is a header node or not .

The structure definition of the circular linked lists and the linear linked list is the same:

```
struct node{
int info;
struct node *next;
};
typedef struct node *NODEPTR ;
```

The delete after and insert after functions of the linear lists and the circular lists are almost the same.

The delete after function : delafter( )

```
void delafter( NODEPTR p, int *px)
{
NODEPTR q;
if((p == NULL) || (p == p->next)){ /*the empty list
contains a single node and may be pointing itself*/
printf("void deletionn");
exit(1);
}
q = p->next;
*px = q->info; /*the data of the deleted node*/
p->next = q->next;
freenode(q);
```

}
The insertafter function : insafter( )
```
void insafter( NODEPTR p, int x)
{
NODEPTR q ;
if(p == NULL){
printf("void insertionn");
exit(1);
}
q = getnode();
q->info = x; /*the data of the inserted node*/
q->next = p->next;
p->next = q;
}
```

<u>Doubly Linked list</u>

It is a way of going *<u>both</u>* directions in a linked list, *<u>forward</u>* and *<u>reverse</u>*.

Many applications require a quick access to the *<u>predecessor</u>* node of some node in list

info: the user's data

next, back: the address of the next and previous node in the list

A doubly linked list provides a natural implementation of the List ADT

Nodes implement Position and store:

element

link to the previous node

link to the next node

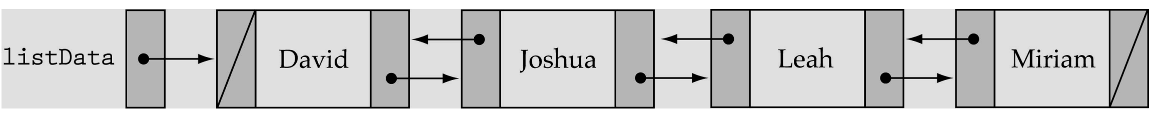
Special trailer and header nodes

To simplify programming, two special nodes have been added at both ends of the doubly-linked list.

Head and tail are dummy nodes, also called sentinels, do not store any data elements.

Head: header sentinel has a ***null-prev*** reference (link).

Tail: trailer sentinel has a ***null-next*** reference (link).



- We no longer need to use *prevLocation* (we can get the predecessor of a node using its *back* member)

Inserting into doubly linked list

1. AddFirst Algorithm

To add a new node as the first of a list:

Algorithm addFirst()

new(T)

T.data ← y

T.next ← head.next

T.prev ← head

head.next.prev ← T {Order is important}

head.next ← T

Size++

2. AddLast Algorithm

To add a new node as the last of list:

Algorithm addLast()

new(T)

T.data ← y

T.next ← tail

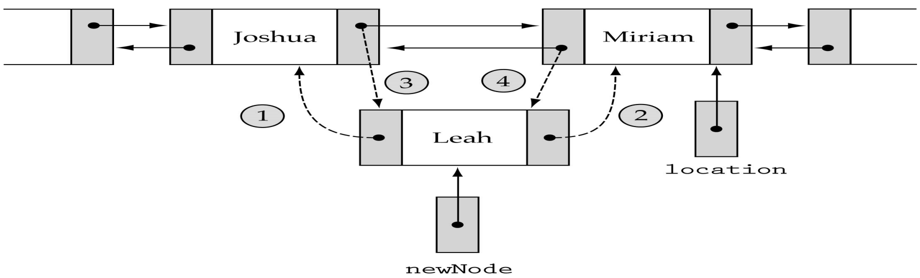
T.prev ← tail.prev

tail.prev.next ← T {Order is important}

tail.prev ← T

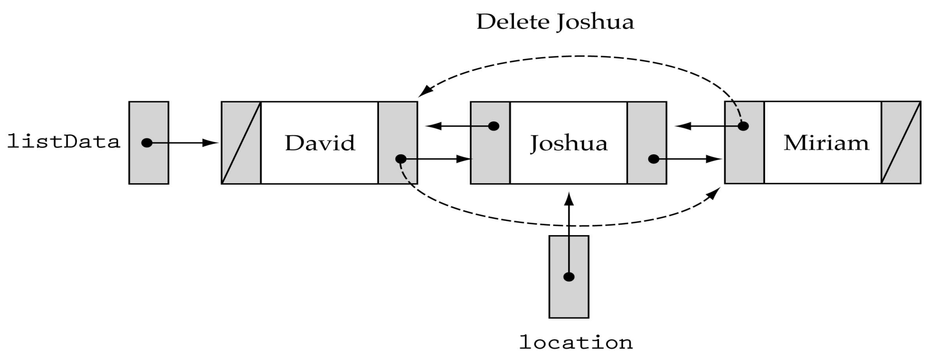
Size++

This Algorithm is valid also in case of empty list.



1. newNode->back = location->back;
2. newNode->next = location
3. location->back->next=newNode;
4. location->back = newNode;

Deleting from a doubly linked list



Algorithm removeLast()

If size = 0 then output “error”

else { T ← tail.prev

y ← T.data

T.prev.next ← tail

tail.prev ← T.prev

delete(T) {garbage collector}

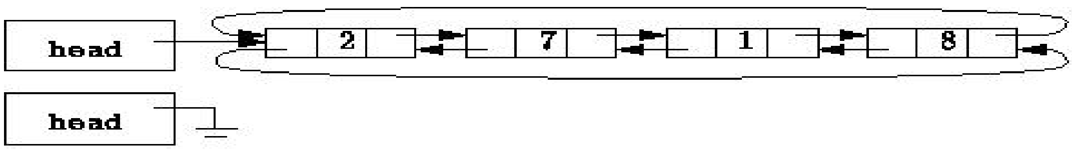
size--

return y

}

Circular Doubly Linked List

- Add a head node at the left and/or right ends
- In a non-empty circular doubly linked list:
  - LeftEnd->left is a pointer to the right-most node (i.e., it equals RightEnd)
  - RightEnd->right is a pointer to the left-most node (i.e., it equals LeftEnd)

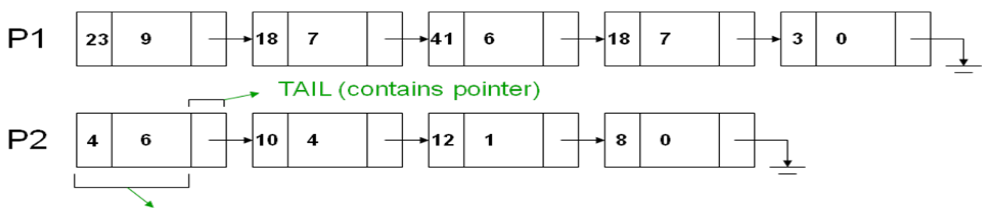


Polynomial Addition using Linked list

Example

$p1(x) = 23x^9 + 18x^7 + 41x^6 + 163x^4 + 3$

$p2(x) = 4x^6 + 10x^4 + 12x + 8$



- Advantages of using a Linked list:

  save space (don’t have to worry about sparse polynomials) and easy to maintain don’t need to allocate list size and can declare nodes (terms) only as needed

- Disadvantages of using a Linked list :

    can’t go backwards through the list

    can’t jump to the beginning of the list from the end.

Adding polynomials using a Linked list representation: (storing the result in p3)

To do this, we have to break the process down to cases:

Case 1: exponent of p1 > exponent of p2

Copy node of p1 to end of p3.[go to next node]

Case 2: exponent of p1 < exponent of p2

Copy node of p2 to end of p3.[go to next node]

Case 3: exponent of p1 = exponent of p2

Create a new node in p3 with the same exponent and with the sum of the coefficients of p1 and p2.

Arrays and polynomials

Polynomials like 5x4+2x3+7x2+10x-8 can be maintained using an array

To achieve each element of the array should have two values coefficient and exponent.

Two polynomial operations are performed

Polynomial addition and polynomial multiplication

Addition of two polynomials

Here if the exponents of two terms being compared are then equal then their coefficients are added and the result is stored in the 3rd polynomials

If the exponents of two terms are not equal then the term with bigger exponent is added to th third polynomial

If the term with an exponent is present in only 1 of the 2 polynomials then that term is added as it is to the 3rd polynomial.

Ex: 1st polynomial is $2^{x6}+3^{x5}+5^{x2}$

2nd polynomial is $1^{x6}+5^{x2}+1x+2$

Resultant polynomial is $3^{x6}+3^{x5}+10^{x2}+1x+2$

Multiplication of 2 polynomials:

Here each term of the coefficient of the 2 nd polynomial is multiplied with each term of the coefficient of the 1st polynomial.

Each term exponent of the 2 nd polynomial is added to the each tem of the 1st polynomial.

Adding the all terms and these equations placed to the resultant polynomial.

Ex: 1st polynomial is $1^{x4}+2^{x3}+2^{x2}+2x$

2nd polynomial is $2^{x3}+3^{x2}+4x$

Resultant polynomial is $2^{x7}+7^{x6}+14^{x5}+18^{x4}+14^{x3}+8^{x2}$

```
#include <iostream.h>
#include <conio.h>
class poly
{ int *coeff;
int dmax;
```

```
public:
void create(int);
void accept();
void display();
poly operator +(poly);
poly operator *(poly);
};
void poly::create( int m)
{ dmax=m;
coeff=new int[dmax+1];
for(int i=0;i<=dmax;i++)
coeff[i]=0;
}

void poly::accept()
{
for(int i=0;i<=dmax;i++)
{
cout<<"Enter the co-efficient at degree "<<i<<":";
cin>>coeff[i];
}
}

void poly::display()
{
cout<<endl<<"The polynomial is :";
for(int i=0;i<=dmax;i++)
{
if(coeff[i]!=0)
cout<< coeff[i] << "X^"<< i << " + ";
}
cout<<endl;
}

poly poly::operator *(poly p)
{
int s=dmax+p.dmax;
poly temp;
```

```
temp.create(s);
for(int i=0;i<=dmax;i++)
{
for(int j=0;j<=p.dmax;j++)
temp.coeff[i+j]+=(coeff[i] * p.coeff[j]);
}
return temp;
}

poly poly::operator +(poly p)
{
poly temp;
int small, large, flag;
if(dmax>p.dmax)
{
large=dmax;
small=p.dmax;
flag=1;
}
else
{
large=p.dmax;
small=dmax;
flag=0;
}
 temp.create(large);
for(int i=0;i<=small;i++)
{
temp.coeff[i]=coeff[i]+p.coeff[i];
}
for(i=small+1;i<=large;i++)
{
if(flag==1)
temp.coeff[i]=coeff[i];
else
temp.coeff[i]=p.coeff[i];
}
return temp;
```

```
}

void main()
{
clrscr();
poly p1,p2,p3;
cout<<"Enter the order of your first polynomial :";
int deg;
cin>>deg;
p1.create(deg);
p1.accept();
p1.display();

cout<<"Enter the order of your second polynomial :";
cin>>deg;
p2.create(deg);
p2.accept();
p2.display();

p3=p1+p2;
cout<<"Resultant polynomial after adding two polynomials"<<endl;
p3.display();
 p3=p1*p2;
cout<<"Resultant polynomial after multiplying two polynomials"<<endl;
p3.display();
getch();
}
```

Multiplication of two polynomials

f(x) and g(x) are two polynomials. In order to solve f(x)*g(x):

- Represent two polynomials in two linked lists (l1 for f(x) and l2 for g(x))
- For each item i in l1, multiply i with l2 and store the result in a new list. Add all the new lists together.
- To multiply an item i with a list l. You need to multiply i with each item in l and store the results in a new list.

Multiply an item i (exp1, coff1)with an item j(exp2, coff2), you get an item k (exp1+exp2, coff1*coff2)

```
public Polynomial multiply(Polynomial p) {
 Node temp1 = poly;
        Node temp2 = p.poly;
        Node front = null;
        Node last = null;

        while(temp1!=null)
```

```
    {
      if(temp2==null)
      {
        temp2 = p.poly;
      }
      while(temp2!=null)
      {

Nodeptr= new Node((temp1.term.coeff*temp2.term.coeff),(temp1.term.degree+temp2.term.degree), null);

        if(last!=null)
        {
          last.next = ptr;
        }
        else{
          front = ptr;
        }
        last = ptr;
        temp2=temp2.next;
      }
      temp1 = temp1.next;

    }
    Polynomial productPoly = new Polynomial();
    productPoly.poly = front;
    return productPoly;
  }
```