Module 3

Linear data structures and linked storage representation

<u>Advantages of arrays</u>

Linear data structures such as stacks and queues can be represented and implemented using sequential allocation ie using arrays. Use of arrays have the following advantages

Data accessing is faster because we just need to specify the array name and the index of the element to be accssed.. ie arrays are simple to use

<u>Disadvantages</u>

The size of the array is fixed. This limits the user in determining the required storage well in advance. However memory requirement cannot be determined while writing the program . ie if the allocation is more and required is less then memory is wasted. Similarly if less memory is allocated and requirement is more then we cannot allocate memory during execution.

Also array items are stored sequentially. In such a case contiguous memory may not be available. Most of the times chunks of memory will be available but they cannot be used as they are not contiguous.

Finally insertion and deletion of items in a array is difficult.

Insertion of an element at the i^{th} position requires us to move all the elements from $i+1^{th}$ position

To the next position and only after this movement insertion can be performed. A similar movement is required for deletion also.

<u>Linked lists</u>

A linked list is a collection of zero or more nodes where each node has some information

data	link

Link will contain the address of the next node

data will contain the information.

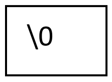
Hence if the address of a node is known we can easily access the subsequent nodes

Types of lists

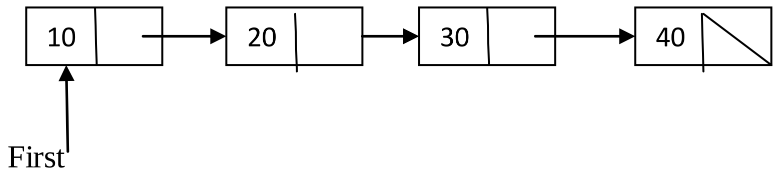
- Singly linked lists
- doubly linked lists
- circular linked lists
- circular doubly linked lists

Singly linked list

A singly linked list is a collection of zero or more nodes. Each node will have one info field and one link field. A chain is a singly list with zero or more nodes. A empty list is as shown



A list is represented as follows



Note:

List contains 4 nodes where each node contains a info and link fields

data filed contains the data

Link field contains the address of the next node

First is a variable containing the address of the first node of the list

Using first we can access any node in the list

The link field of the last node will have NULL

Representation

Nodes in the list are self referential structures.

A structure is a collection of one or more fields which are of the same type or different type. A self refrential structure is a structure which has at least one field which is a pointer to the same structure .

ex

struct Node

{

int data;

struct node * link;

};

data is a int containing integer data

Link is a pointer and should contain the address and normally it contains the address of the next node

Declaration of variable

We have the following declaration

struct Node

{

Int info;

Struct Node * link;

};

struct Node * first;

Creation of a empty list

Before creating a list we need to create a empty list first. This is achieved by assigning NULL to the variable first ie

first=NULL

creating a new node

A new node is created as follows

first=(struct Node *) malloc(sizeof(struct Node));

The above statement will allocate memory to the variable first

Using first we can access the contents of the node such as

First → data or first →link(for the next node)

To store the data. We use

first→data=10;

We need to store NULL as this is the first node

First→link=NULL;

Deleting a node

Any node which is not required is deleted using the free function

free(first);

When the above statement is executed the memory which was allocated to first using malloc is deallocated and returned to the Operating System. The variable first will have a invalid address and hence is called as dangling pointer.

<u>Operations on a singly linked list</u>

Inserting a node

Deleting a node

Search for a node

Display the list

```
#include<stdio.h>
#include<stdlib.h>

// Declaration of the node

struct Node
{
   int data;
   struct Node *link;
};
struct Node first =NULL;

//function to add a node at the end of the list

void InsertL( int ele)
{
   struct node *p,*q;
   p= (struct node *)malloc(sizeof(struct node));
   p →data=ele;
   for(q = first; q→ link != NULL; q=q→link)
                  ;
   q→link = p;
    q = p;
    q→link=NULL;
}


Void  InsertF( int ele )
{
   struct node *p;
   p =(struct Node *)malloc(sizeof(struct Node));
   p→data=ele;
   p→link =NULL;
   if (first = = NULL)
      first=p;
   else
   {
            p→link=first;
            first = p;
   }
}
```

```
//function to add a node after a given node
void addAfter(int ele, int loc)
{
   int i;
   struct node *p,*q,*prev = NULL;
   q = first;
   for(i=1; i<loc;i++)
   {
            prev = q;
            q = q →link;
   }
   p=(struct node *)malloc(sizeof(struct node));
   p→data= ele;
   prev→link =p;
   prev = p;
   prev →link= q;
}
```

Note: For all delete Function   refer Class notes

```
// function to display a list

void display(struct Node *first)
{  struct Node *p;
   p = first;
   if(p = =NULL)
   {
   Printf("List is Empty\n");
  return;
   }
   while(p !=NULL)
   {
            printf("%d ",p →data);
            p=p→link;
   }
}
```

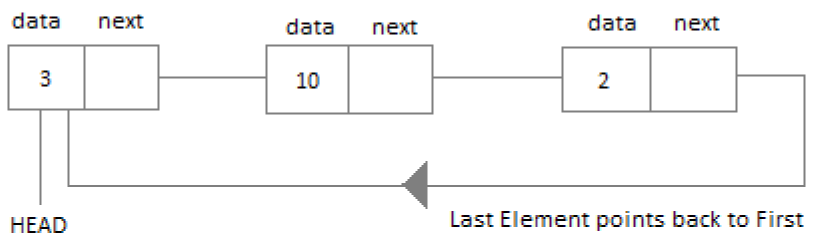
```
//function to count the number of nodes in a list
int count(struct Node *first)
{
   struct Node *p = first;
   int cnt=0;
   while(p!=NULL)
   {
            p=p→link;
            cnt++;
   }
   return cnt;
}
```

Function to search for a element in a list

```
struct Node * search(struct Node *first, int key)
{ struct Node *p
while (p != NULL)
{
  if (p → data = = key) /* successful search */
       return p;
 else
      p = p→ link;
}
return  NULL; /* unsuccessful search */
}
```

<u>Circular linked list</u>

Circular Linked List is little more complicated linked data structure. In the circular linked list we can insert elements anywhere in the list whereas in the array we cannot insert element anywhere in the list because it is in the contiguous memory. In the circular linked list the previous element stores the address of the next element and the last element stores the address of the starting element. The elements points to each other in a circular way which forms a circular chain. The circular linked list has a dynamic size which means the memory can be allocated when it is required.



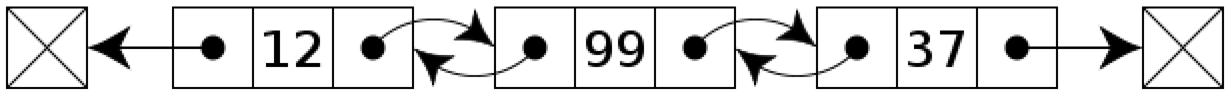
Application of Circular Linked List

- The real life application where the circular linked list is used is our Personal Computers, where multiple applications are running. All the running applications are kept in a circular linked list and the OS gives a fixed time slot to all for running. The Operating System keeps on iterating over the linked list until all the applications are completed.
- Another example can be Multiplayer games. All the Players are kept in a Circular Linked List and the pointer keeps on moving forward as a player's chance ends.
- Circular Linked List can also be used to create Circular Queue. In a Queue we have to keep two pointers, FRONT and REAR in memory all the time, where as in Circular Linked List, only one pointer is required.

Note: For Operations on circular Linked List refer class Notes

Doubly linked list

In computer science, a **doubly linked list** is a linked data structure that consists of a set of sequentially linked records called nodes. Each node contains two fields, called *links*, that are references to the previous and to the next node in the sequence of nodes. The beginning and ending nodes' **llink** and **rlink** links, respectively, point to some kind of terminator, typically a sentinel node or null, to facilitate traversal of the list. If there is only one sentinel node, then the list is circularly linked via the sentinel node. It can be conceptualized as two singly linked lists formed from the same data items, but in opposite sequential orders.



A doubly linked list whose nodes contain three fields: an integer value, the link to the next node, and the link to the previous node.

The two node links allow traversal of the list in either direction. While adding or removing a node in a doubly linked list requires changing more links than the same operations on a singly linked list, the operations are simpler and potentially more efficient (for nodes other than first nodes) because there is no need to keep track of the previous node during traversal or no need to traverse the list to find the previous node, so that its link can be modified.

Note: For Operations on doubly Linked List refer class Notes

Applications of linked lists

Polynomial addition

A polynomial is sum of terms where each term is of the form

$\mathbf{ax^e}$

Where a is the coefficient is the variable, e is the exponent

The largest exponent is also called as the degree.

Various operations can be performed on the polynomials

Iszero(poly)-Returns 0 if the polynomial does not exist

Coeff(poly,expo)-returns the coefficient

Leadexpo(poly)-returns the largest exponent

Attach- Attach a term<coeff,expo> onto the polynomial

Remove (poly,expo)-delete a term

Add(poly1,poly2)-adds two polynomials

General procedure to add two polynomials

Three cases exist

Case 0 :

Powers of the two terms to be added are equal

In such a case the coefficients of the two terms are added using

Sum=coeff(a,leadexpo(a)+coeff(b,leadexpo(b))

And then insert the resultant term on to the resulting polynomial using the attach function. We move onto the next term using the remove function.

Ie

A=remove(a,leadexpo(a));

B= remove (b,leadexpo(b));

Case 1:

Power of the first term is > power the second term

In such a case we insert the first term onto the resulting polynomial using

Attach(c,coeff(a,leadexpo(a))

And move onto the next term using remove function

A=remove(a,leadexpo(a)

Default case:

Here the power of the first term is < the power of the second term

In such a case we insert the second term onto the resulting polynomial using

Attach(c,coeff(b,leadexpo(b))

And move onto the next term using remove function

b=remove(b,leadexpo(b)

```
//9.c polynomial operations
#include<stdio.h>
#include<process.h>
#include<conio.h>
#include<math.h>

typedef struct node
{
int expo,coef;
struct node* next;
}node;
node * insert(node *,int,int);
node * create();
node * add(node *p1,node *p2);
int eval(node *p1);
```

```
void display(node *head);



node * insert(node *head,int coef1,int expo1)

{

node *p,*q;

p=(node *) malloc(sizeof(node));

p->expo=expo1;

p->coef=coef1;

p->next=NULL;


if(head==NULL)

{

head=p;

head->next=head;

return (head);

}

if(expo1>head->expo)

{

p->next=head->next;

head->next=p;

head=p;

return(head);

}
```

```
if(expo1==head->expo)

{

head->coef=head->coef+coef1;

return(head);

}

q=head;

while(q->next!=head && expo1>=q->next->expo)


//{//insert

q=q->next;

  if(p->expo==q->expo)

  q->coef=q->coef+coef1;

  else

  {

  p->next=q->next;

  q->next=p;

  }

  return (head);

  }

  //}//inserted


  node *create()

  {

  int n,i,expo1,coef1;
```

```
node *head =NULL;
printf("\n enter num of poly terms");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("\n enter coef and expo");
scanf("%d%d",&coef1,&expo1);
head=insert(head,coef1,expo1);
}
return (head);
}


node *add(node *p1,node *p2)
{
node *p;
node *head=NULL;
printf("\n addition of ploy");
p=p1->next;
do
{
head=insert(head,p->coef,p->expo);
p=p->next;
} while(p!=p1->next);
p=p2->next;
```

```
do
{
head=insert(head,p->coef,p->expo);
p=p->next;
}while(p!=p2->next);
return (head);
}
int eval(node *head)
{
node *p;
int x,ans=0;
printf("\n enter the value of x");
scanf("%d",&x);
p=head->next;
do
{
ans=ans+p->coef*pow(x,p->expo);
p=p->next;
} while(p!=head->next);
return (ans);
}

void display(node *head)
{
```

```
node *p,*q;
int n=0;
q=head->next;
p=head->next;
do
{
n++;
q=q->next;
} while(q!=head->next);
printf("\n poly is ");
do
{
if(n-1)
{
printf(" %dx^(%d) +  ",p->coef,p->expo);
p=p->next;
}
else
{
printf(" %dx^(%d)"  ,p->coef,p->expo);
p=p->next;
}
n--;
} while(p!=head->next);
```

```
}



void main()
{
node *p1,*p2,*p3;
int a,x,ch;
p1=p2=p3=NULL;
while(1)
{
printf("\n1.add");
printf("\n2.evaluate");
printf("\n 3.exit");
printf("enter ur choice");
scanf("%d",&ch);
switch(ch)
{
case 1:
p1=create();
display(p1);
p2=create();
display(p2);
p3=add(p1,p2);
display(p3);
break;


case 2: p1=create();

display(p1);

a=eval(p1);

printf("\n value of poly is  %d",a);

break;

case 3: exit(0);

break;

default: printf("\n invalid choice");

    break;

    }

    }

    }
```