UNIT I LINEAR DATA STRUCTURES – LIST

Abstract Data Types (ADTs) – List ADT – array-based implementation – linked list implementation – – singly linked lists- circularly linked lists- doubly-linked lists – applications of lists –Polynomial Manipulation – All operation (Insertion, Deletion, Merge, Traversal).

1.1 INTRODUCTION TO DATA STRUCTURES

Data structures

Data Structures is a means of storing a collection of data. It is the specification of the elements of the structure, the relationship between them and the operations that may be performed upon them.

1.1.1 Types of Data Structures

Data structures can be classified based on the organization and the operations defined on it.

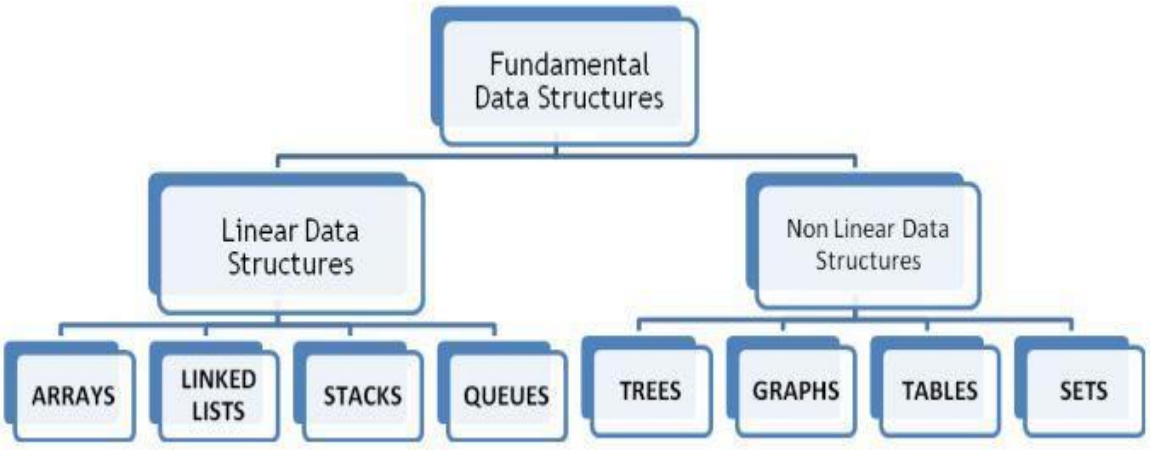


Fig: 1.1 Types of Data Structures

Linear and non-linear structure: Simple data structures can be combined in various ways to form more complex structure. There are two kinds of complex data structure. They are linear & non-linear, depending on the complexity of the logical relationship they represent

♣ **Linear data structure**: Stacks, queues, linear linked list, arrays

♣ **Non- linear data structure**: Tree and graph tables, sets.

Linear Data Structures

All the elements form a sequence or maintain a linear ordering. Data‟s are organized in a sequential manner.

Non Linear Data Structures

If all the elements are organized in a distributed manner then it was termed as Non-linear data structures

1.2 ABSTRACT DATA TYPE

- **Abstract data type** (**ADT**) is a specification of a set of data and the set of operations that can be performed on the data.
- Objects such as lists, sets, and graphs, along with their operations, can be viewed as abstract data types, just as integers, real, and Booleans are data types.
- Set of operations --- each operation does a specific task.
- ADT is a small function, a subprogram, which cannot be implemented as such, as it contains only the essential things and lacks the rest.

1.2.1 Operations on ADT:

1. Insertion at first, last and middle
2. Deletion at first, last and middle
3. Modifying the list,
4. Reversing the list,
5. Merging the list

Uses of ADT: -

1. It helps to efficiently develop well designed program
2. Facilitates the decomposition of the complex task of developing a software system into a number of simpler subtasks
3. Helps to reduce the number of things the programmer has to keep in mind at any time
4. Breaking down a complex task into a number of earlier subtasks also simplifies testing and debugging

1.3 THE LIST ADT

- List is the collection of related data items.
- Eg: Name list (collection of names)

List of ADT's:

1. Insertion at first,middle,last
2. Deletion at first,middle,last
3. Searching
4. Reversing
5. Traversing

Implementation of LIST ADT:

1) Array Implementation
2) Linked List Implementation(using Pointers)

1.4 ARRAY IMPLEMENTATION

Arrays

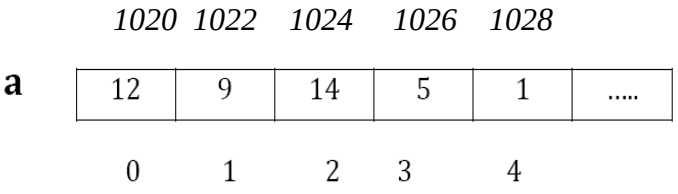
- It is the collection of related data items, stored in a continuous way.
- The very common linear structure is array. Since arrays are usually easy to traverse, search and sort, they are frequently used to store relatively permanent collections of data.
- An array is a list of a finite number n of *homogeneous* data elements (i.e., data elements of the same type) such that:
  a) The elements of the array are referenced respectively by an *index* consisting of n consecutive numbers.
  b) The elements of the array are stored respectively in successive memory locations.

ARRAY ADTS:

1. Creation:

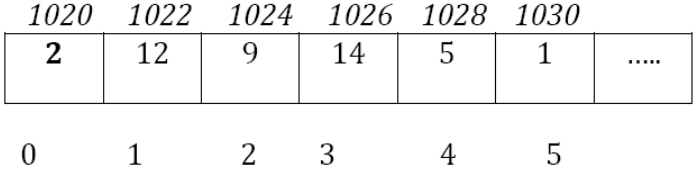
```
Void create(int a[20], int n)
{
int i;
for(i=0;i<n;i++)
scanf("%d",&a[i]);
}
```

Example: Let n=5



2. Insertion at first:

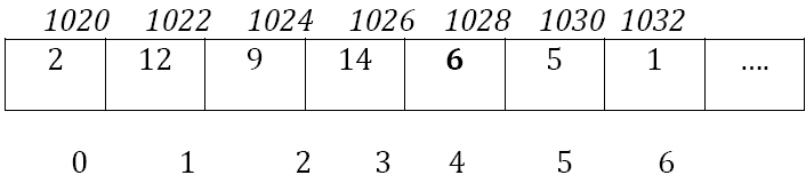
```
Void ins_first(int a[ ],int x,int n)
{
int i;
for(i=n-1;i>=0;i--)
a[i+1]=a[i];
a[0]=x;
}
```



Let the no to be inserted be **2**

3. Insertion at middle:

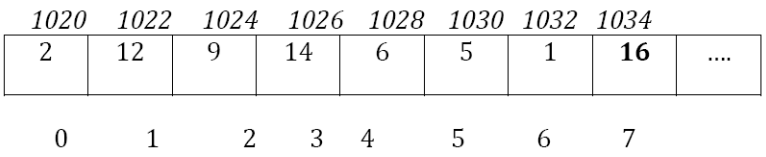
```
Void ins_middle(int a[ ], int num,int n,int pos)
{
int i;
for(i=n-1;i>=pos;i--)
a[i+1]=a[i];
a[pos]=num;
}
```



Let the no to be inserted be **6** at pos **4**

4. Insertion at LAST:

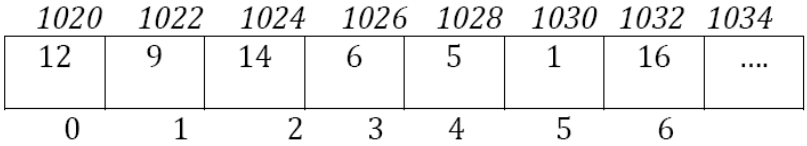
```
Void ins_end(int a[ ],int num,int n)
{
a[n]=x;
n=n+1;
}
```



Let the no to be inserted at last be 1**6**

5. Deletion at first:

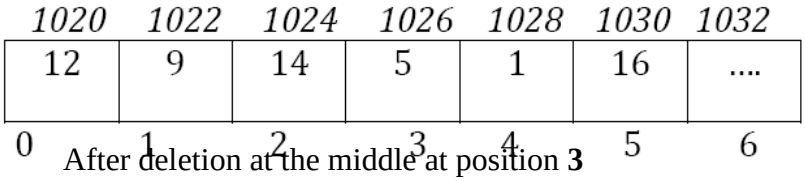
```
Void del_first(int a[ ],int n)
{
int i; n=n-1;
for(i=0;i<n;i++)
a[i]=a[i+1];

}
```



After deletion at the first the array looks like

6. Deletion at middle:

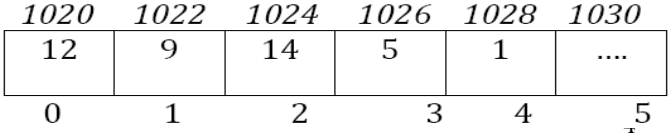
```
Void del_middle(int a[ ],int pos,int n)
{
int i; n=n-1;
for(i=pos;i<n;i++)
a[i]=a[i+1];
}
```



After deletion at the middle at position **3**

7. Deletion at end:

```
Void del_end(int a[ ],int n)
{
```



```
a[n-1]=0;
n=n-1;
}
```

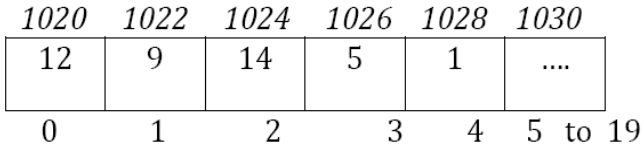
8. Searching:

```
Void search(int a[ ],int x,int n)
{
int i,flag=0;
for(i=0;i<n;i++)
{
if(a[i]==x)
flag=1;
} if(flag==1)
printf("Number found");
else
printf("Number not found");
}
```

If X=3 then output is **Number not found.**
If X=1 then output is **Number found**

9. To check if the list is empty or not:

```
Void empty(int a[ 20],int pos)
{
if(pos== -1)
printf("List is empty");
else
printf("List is filled");
}
```

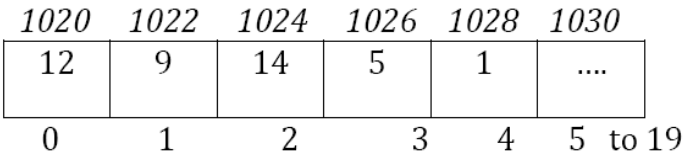


The index value of **n=20 here**.
BUT 5 index are filled in **array a.**

So the list is not empty bcoz it has 5 values filled it it.

10. To check if the list is full:

```
Void list_full(int a[20 ],int pos,int n)
{
if(pos==(n-1))
printf("List is full");
else
printf("List is not full");
}
```



The index value of **n=20 here**.
ONLY 5 index are filled in **array a.**

So the list is not full bcoz it has only 5 values filled in it & has 15 empty spaces.
So the output is **List is not full.**

1.5 THE LINKED LIST IMPLEMENTATION OF LIST ADT

Linked List:

It is a collection of nodes, each node should have the respective data field and a respective link fields (next address).

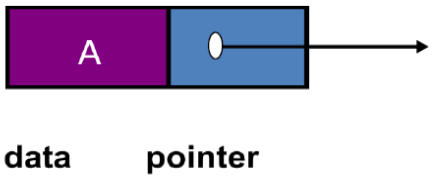


Fig: 1.2 Linked Lists Representation

Each node contains at least

- A piece of data (any type)
- Pointer to the next node in the list
- **Head**: pointer to the first node
- The last node points to **NULL**

Types of Linked List:

1. Singly Linked List
2. Doubly Linked List
3. Circular Singly Linked List
4. Circular Doubly Linked List

1.6 SINGLY LINKED LIST

Singly Linked Lists are a type of data structure. It is a type of list. In a singly linked list each node in the list stores the contents of the node and a pointer or reference to the next node in the list. It does not store any pointer or reference to the previous node. It is called a singly linked list because each node only has a single link to another node.

To store a single linked list, you only need to store a reference or pointer to the first node in that list. The last node points to null to indicate that it is the last node.

Each cell is called a **node** of a singly-linked list. First node is called **head** and it's a dedicated node.

By knowing it, we can access every other node in the list. last node, called **tail**, is also stored in order to speed up add operation.

Representation:

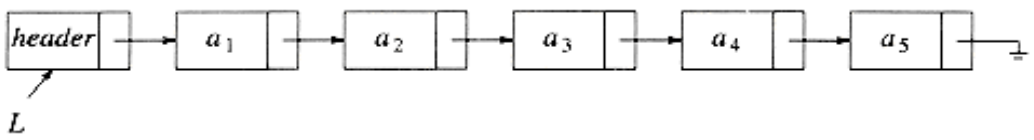


Fig: 1.3 Singly Linked Lists Representation

A singly linked list is one, which has only two fields for each node, i.e. a data field and a link field.

Deletion from a linked list

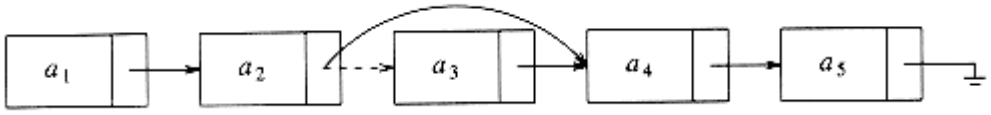


Fig: 1.4 Singly Linked Lists Representation-Deletion

Insertion into a linked list

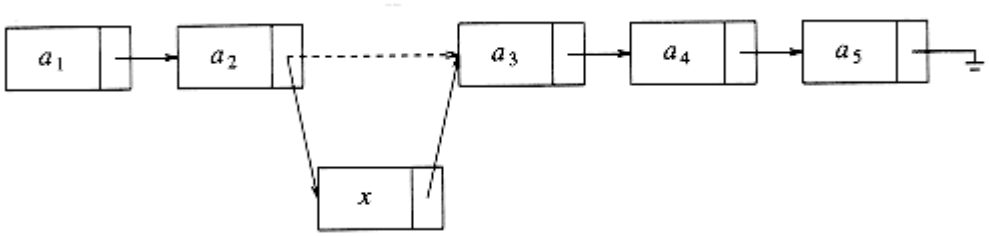


Fig: 1.5 Singly Linked Lists Representation-Insertion

Possible ADT's:

- Creation.
- Insertion at First, Middle, Last.
- Deletion at First, Middle,Last.
- Searching.
- Isempty.

Basic Steps for creating a linked list:

1. Allocate the memory
2. Assign the data
3. Assign the proper link

4. In order to allocate the memory for node, use the function malloc( )
5. In order to free the memory, use the function free( )

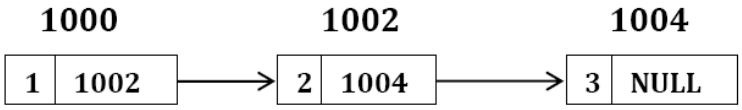
The structure definition of Singly Linked List

```
Struct node
{
int data;
struct node *next;
} *head;
```

The Possible ADTS Of Singly Linked List

1. Creation:

```
Void create(struct node *head,int num)
{
struct node *temp;
head= malloc(sizeof(struct node));
head --> data=num;
head --> next=null;
temp=head;
scanf("%d",&num);
do
{
Temp --> next=malloc(sizeof(struct node));
Temp= temp --> next;
Temp --> data=num;
scanf("%d",&num);
}while(num ! =0)
Temp --> next= null;
}
```

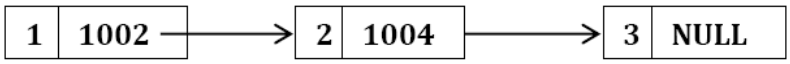


Let the list created be

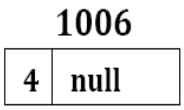
2. Insertion at first:

```
void ins_first(struct node *head,int num)
{
Struct node *temp;
temp= malloc(sizeof(struct node));
temp --> data= num;
temp --> next= head;
head= temp;
}
```

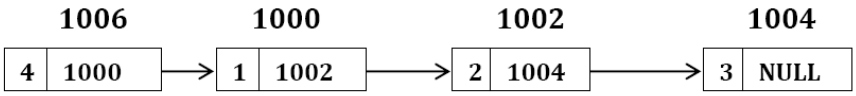
Before Insertion:



Node to be inserted:



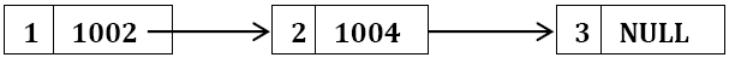
After Insertion:



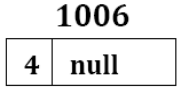
3. <u>Insertion at last:</u>

```
Void ins_last(struct node *head, int num)
{
Struct node *temp, *t;
temp=head;
while(temp-->next != NULL)
{
temp= temp-->next;
}
t=malloc(sizeof(struct node));
t-->data=num;
t-->next=NULL;
temp-->next=t;
}
```

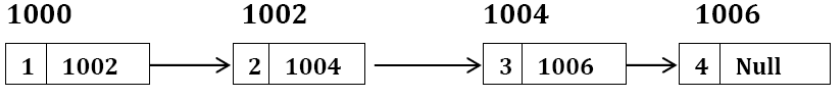
Before Insertion:



Node to be inserted:



After Insertion:

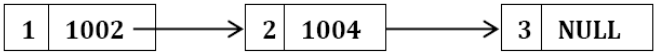


5. <u>Insertion at middle</u>:

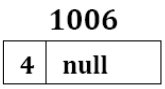
```
Void ins_middle(struct node *head,int num, int num1)
{
Struct node *temp, *ptr;
```

```
temp=malloc(sizeof(struct node));
temp-->data=num;
ptr=head;
while(ptr-->data != num1)
{
ptr=ptr-->next;
}
temp-->next=ptr-->next;
ptr-->next=temp;
}
```

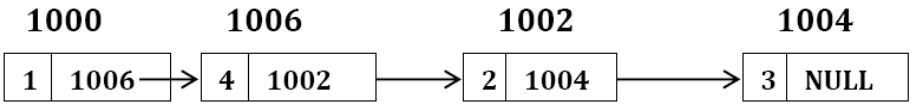
Before Insertion:



Node to be inserted at Position 2:

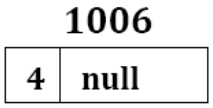


After Insertion:



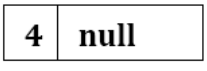
5. <u>Is Empty:</u>

```
Void isempty(struct node *head)
{
if(head == NULL)
Printf("The List is empty");
Else
Printf("The List is not empty");
}
```



6. <u>No. of nodes in the list:</u>

```
Int noofnodes(struct node *head)
{
int count=0;
struct node *temp;
temp=head;
while(temp-->next != NULL)
{
count ++;
```



Output: 1

```
temp=temp-->next;
}
count=count+1;
return count;
}
```

7. Search:

```
Void search(struct node *head,int num)
{
Struct node *temp;
temp=head;
int flag=0;
while(temp != NULL)
{
If(temp-->data==num)
{
Printf("Number found");
Flag=1;
break;
}
Else
temp=temp-->next;
}
If(flag==0)
{
Printf("Number not found");
}
```

4	null

No to be searched : 6
Output: Number not found

8. Deletion at first:

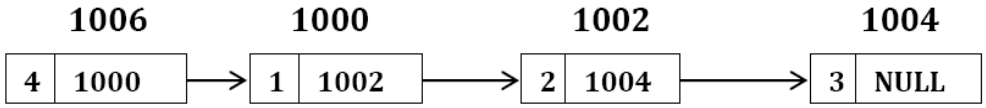
```
Void del_first(struct node *head)
{
Struct node *ptr;
Ptr=head;
Head=head-->next;
Free(ptr);
}
```

Node to be deleted:

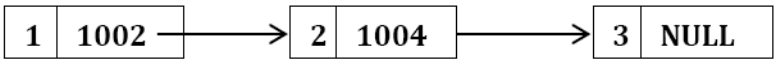
1006

4	null

Before deletion:



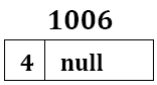
After deletion:



9. <u>Deletion at last:</u>

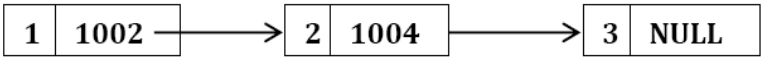
```
Void del_last(struct node *head)
{
Struct node *temp,*ptr;
Temp= head;
While(temp-->next != NULL)
{ Ptr=temp;
Temp=temp-->next;
}
Ptr-->next=NULL;
Free(temp);
}
```

Node to be deleted:



Before Deletion:

After Deletion

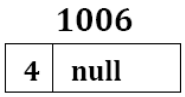


10. <u>Deletion at middle:</u>

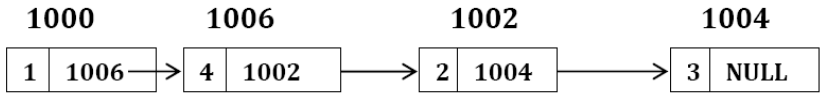
```
Void del_middle(struct node *head,int num)
{
Struct node *temp,*p,temp=head;
While(temp-->data != num)
{
P=temp;
```

```
Temp=temp-->next;
}
p-->next=temp-->next;
free(temp);
}
```

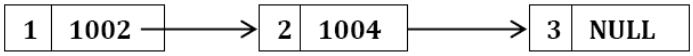
Node to be deleted:



Before Deletion:



After Deletion:



Advantages

1) Singly linked list can store data in non-contiguous locations. Thus there is no need of compaction of memory, when some large related data is to be stored into the memory.

2) Insertion and deletion of values is easier as compared to array, as no shifting of values is involved.

Disadvantages

1) Nodes can only be accessed sequentially. That means, we cannot jump to a particular node directly.

2) Because of the above disadvantage, binary search algorithm cannot be implemented on the singly linked list.

3) There is no way to go back from one node to previous one. Only forward traversal is possible.

1.7 DOUBLY LINKED LIST

A **doubly-linked list** is a linked data structure that consists of a set of data records, each having two special ***link*** fields that contain references to the previous and to the next record in the sequence. It can be viewed as two singly-linked lists formed from the same data items, in two opposite 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 links allow walking along the list in either direction with equal ease. Compared to a singly-linked list, modifying a doubly-linked list usually requires changing more pointers, but is simpler because there is no need to keep track of the address of the previous node.

In simpler terms, Doubly linked list

- Pointers exist between adjacent nodes in both directions.
- The list can be traversed either forward or backward.
- Usually two pointers are maintained to keep track of the list, *head* and *tail*.

Representation of a doubly linked list

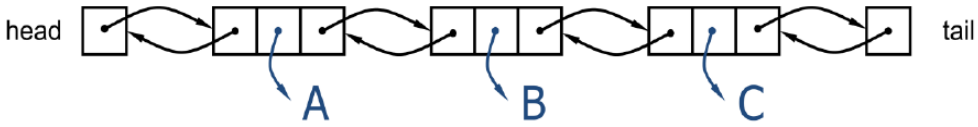
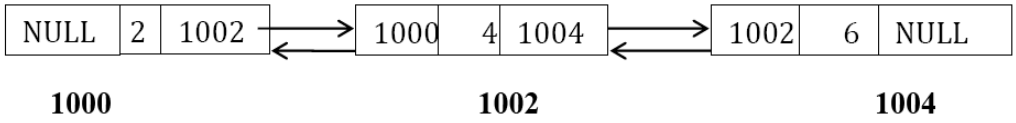


Fig: 1.6 Doubly Linked Lists Representation

Example



Insertion of Node in a Double Linked List

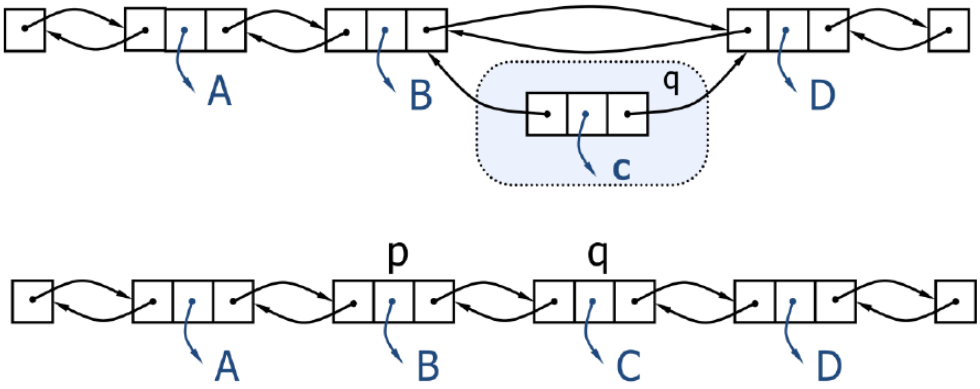


Fig: 1.7 Doubly Linked Lists Representation-Insertion

Deletion of Node in a Double Linked List

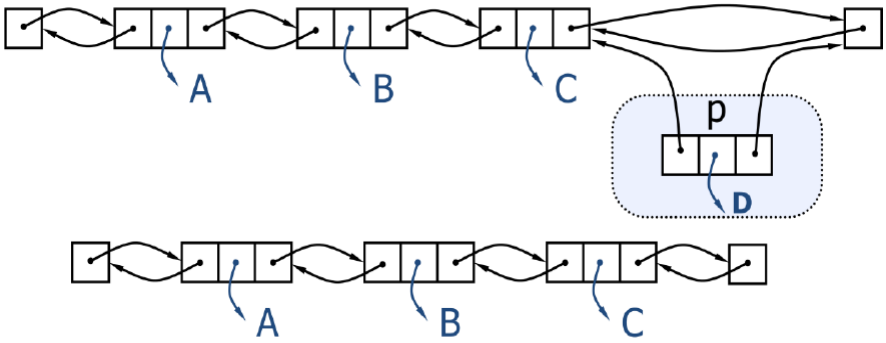


Fig: 1.8 Doubly Linked Lists Representation-Deletion

Structure Declaration :

```
Struct dnode
{
int data;
struct dnode *prev, *next;
} *head;
```

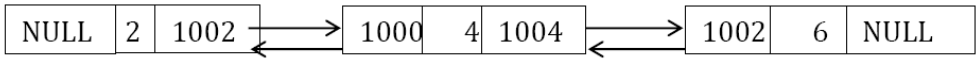
1. <u>Creation:</u>

```
Void create(struct dnode *head, int num)
{
Struct dnode *temp, *ptr;
Head=malloc(sizeof(struct dnode));
Head-->prev=NULL;
Head-->data=num;
Head-->next=NULL;
Temp=head;
Do
{
Temp-->next=malloc(sizeof(struct dnode));
Printf("Enter a number");
Scanf("%d",&num);
Ptr=temp;
Temp=temp-->next;
Temp-->data=num;
Temp-->prev=ptr;
} while(num != 0);
Temp -->next =NULL;
}
```

2. Insertion at first:

```
Void ins_first(struct dnode *head, int num)
{
Struct dnode *temp;
Temp=malloc(sizeof(struct dnode));
Temp-->data=num;
Temp-->prev=NULL;
Head-->prev=temp;
Temp-->next=head;
Head=temp;
}
```

Before Insertion

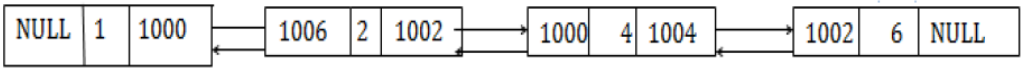


1000 **1002** **1004**

Node to be Inserted

1006

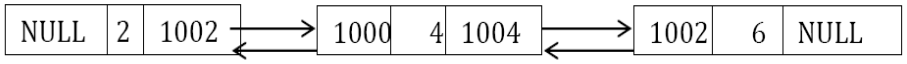
After Insertion



1006 **1000** **1002** **1004**

3. Insertion at last:

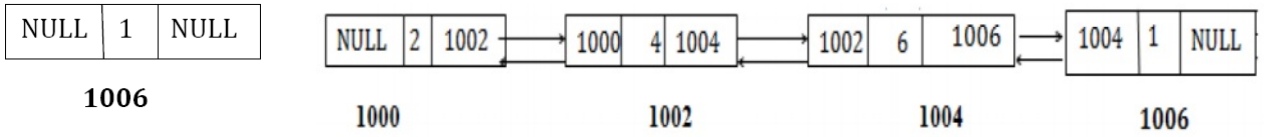
```
Void ins_last(struct dnode *head, int num)
{
Struct dnode *temp, *ptr;
Temp=head;
While(temp-->next != NULL)
{
Temp=temp-->next;
}
Ptr=malloc(sizeof(struct dnode));
Ptr-->data=num;
Ptr-->next=NULL;
Ptr-->prev=temp;
Temp-->next=ptr;
```



1000 **1002** **1004**

```
}
```

Node to be inserted **After Insertion**

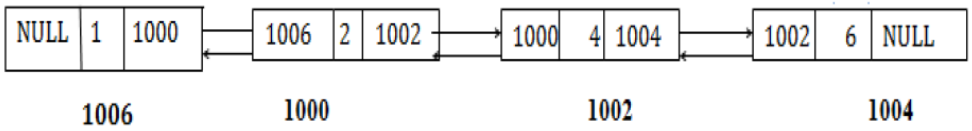


4. Insertion at middle:

```
Void ins_mid(struct dnode *head, int num, int num1)
{
Struct dnode *temp, *ptr, *t;
Temp=malloc(sizeof(struct dnode));
Temp-->data=num;
Ptr=head;
While(ptr-->data != num1)
{
Ptr=ptr-->next;
T=ptr-->next;
Ptr-->next=temp;
Temp-->prev=ptr;
Temp-->next=t;
t-->prev=temp;
}
```

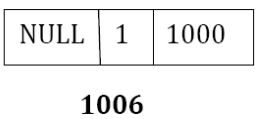
5. Deletion at first:

```
Void del_first(struct dnode *head)
{
Struct dnode *temp;
Temp=head;
Head=head-->next;
Head-->prev=NULL;
Free(temp);
}
```

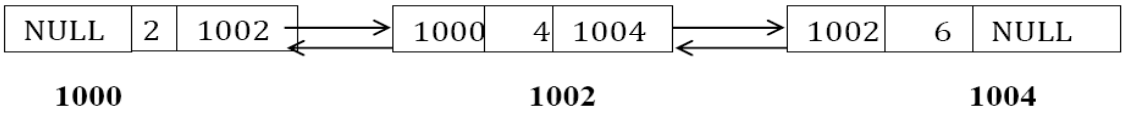


Node to be deleted

Before deletion



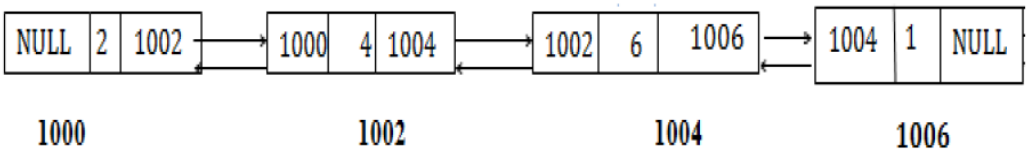
After Deletion



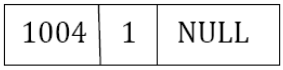
6. Deletion at last :

```
Void del_last(struct dnode *head)
{
Struct dnode *temp, *ptr;
Temp=head;
While(temp-->next != NULL)
{ Ptr=temp;
Temp=temp-->next;
}
Ptr-->next=NULL;
Free(temp);
}
```

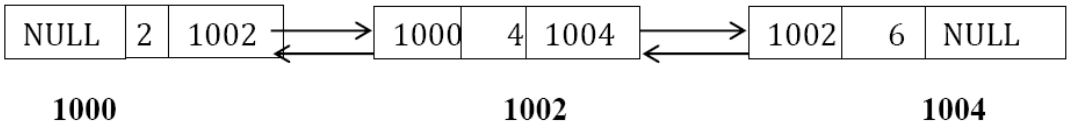
Before deletion



Node to be deleted



After Deletion

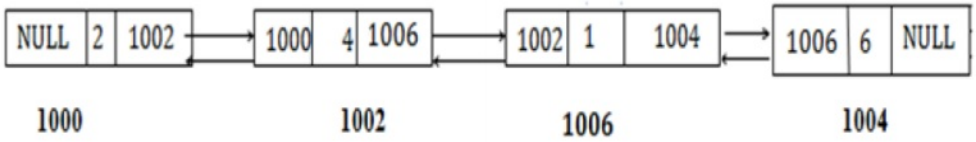


7. Deletion at middle:

```
Void del_mid(struct dnode *head, int num)
{
Struct dnode *temp;
Temp=head;
```

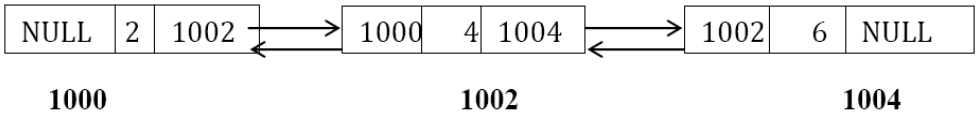
```
While(temp-->data != num)
{
Temp=temp-->next;
}
Temp-->prev-->next=temp-->next;
Temp-->next-->prev=temp-->prev;
Free(temp);
}
```

Before deletion



Node to be deleted

After Deletion



Applications.

1. Applications that have an MRU list (a linked list of file names)
2. The cache in your browser that allows you to hit the BACK button (a linked list of URLs)
3. Undo functionality in Photoshop or Word (a linked list of state)
4. A stack, hash table, and binary tree can be implemented using a doubly linked list.
5. A great way to represent a deck of cards in a game

Advantages

- The primary advantage of a doubly linked list is that given a node in the list, one can navigate easily in either direction.
- This can be very useful, for example, if the list is storing strings, where the strings are lines in a text file (e.g., a text editor).
- One might store the ``current line" that the user is on with a pointer to the appropriate node; if the user moves the cursor to the next or previous line, a single pointer operation can restore the current line to its proper value.
- Or, if the user moves back 10 lines, for example, one can perform 10 pointer operations (follow the chain) to get to the right line.
- For either of these operations, if the list is singly linked, one must start at the head of the list and traverse until the proper point is reached. This can be very inefficient for large lists.

Disadvantages

- each node requires an extra pointer, requiring more space
- The insertion or deletion of a node takes a bit longer (more pointer operations).

1.8 CIRCULAR SINGLY LINKED LIST

Singly Linked List has a major drawback. From a specified node, it is not possible to reach any of the preceding nodes in the list. To overcome the drawback, a small change is made to the SLL so that the next field of the last node is pointing to the first node rather than NULL. Such a linked list is called a circular linked list.

- Because it is a circular linked list, it is possible to reach any node in the list from a particular node.
- There is no natural first node or last node because by virtue of the list is circular.
- Therefore, one convention is to let the external pointer of the circular linked list, tail, point to the last node and to allow the following node to be the first node.
- If the tail pointer refers to NULL, means the circular linked list is empty.

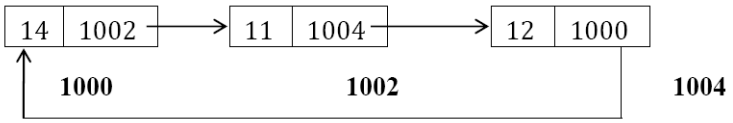


Fig: 1.9 Circular Singly Linked Lists Representation

Structure Declaration:

```
Struct node
{
int data;
struct node *next;
} *head;
```

1. **<u>Creation of a circular singly linked list</u> :**

```
Void create(struct node *head, int num)
{
Struct node *temp;
head-->data=num;
head-->next=NULL;
printf("Enter next data");
scanf("%d",&num);
```

```
temp=head;
while(num != 0)
{
temp-->next=malloc(sizeof(struct node));
temp=temp-->next;
temp-->data=num;
scanf("%d",num);
}
temp-->next=head;
}
```

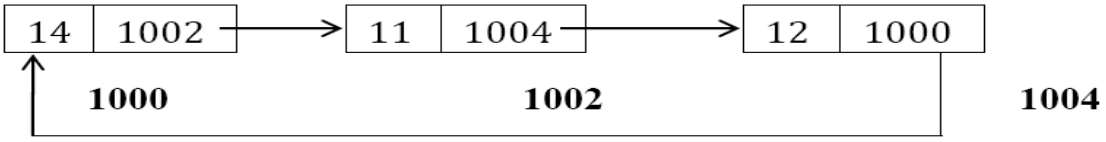
2. Insertion:

```
Void insert(struct node *head, int num, int n)
{
Struct node *temp, *ptr;
temp=malloc(sizeof(struct node));
temp-->data=num;
ptr=head;
while(ptr-->data != n)
{
Ptr=ptr-->next;
}
temp-->next=ptr-->next;
ptr-->next=temp;
}
```

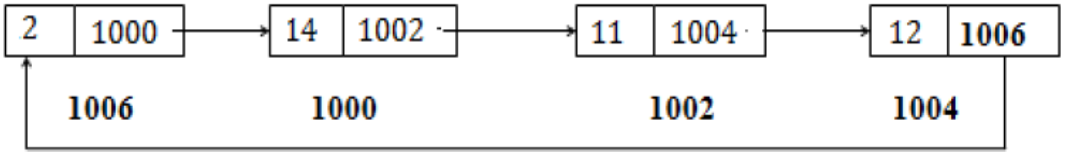
3. Insert before head:

```
Void ins(struct node *head, int num)
{
Struct node *temp, *ptr;
temp=malloc(sizeof(struct node));
temp-->data=num;
ptr=head;
while(ptr-->next != head)
{
ptr=ptr-->next;
}
ptr-->next=temp;
temp-->next=head;
}
```

Before Insertion



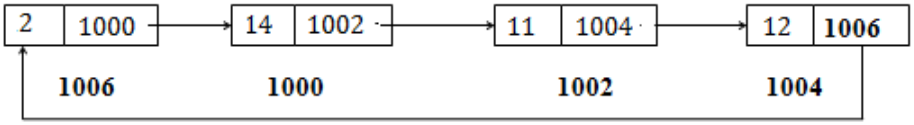
After Insertion



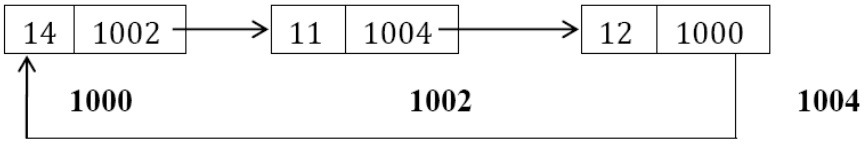
4. Deletion:

```
Void del(struct node *head, int num)
{
Struct node *temp, *ptr;
temp=head;
while(temp-->data != num)
{ ptr=temp;
temp=temp-->next;
}
ptr-->next=temp-->next;
free(temp);
}
```

Before Deletion at First



After Deletion at First



1.9 CIRCULAR DOUBLY LINKED LIST

A Circular Doubly Linked List (CDL) is a doubly linked list with first node linked to last node and vice-versa.

- The „ prev ‟ link of first node contains the address of last node and „ next ‟ link of last node contains the address of first node.
- Traversal through Circular Singly Linked List is possible only in one direction.
- The main advantage of Circular Doubly Linked List (CDL) is that, a node can be inserted into list without searching the complete list for finding the address of previous node.
- We can also traverse through CDL in both directions, from first node to last node and vice-versa.

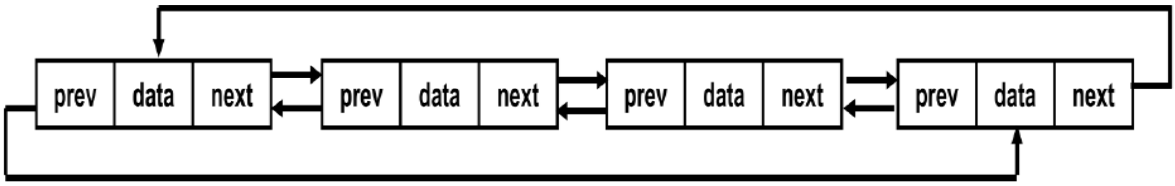
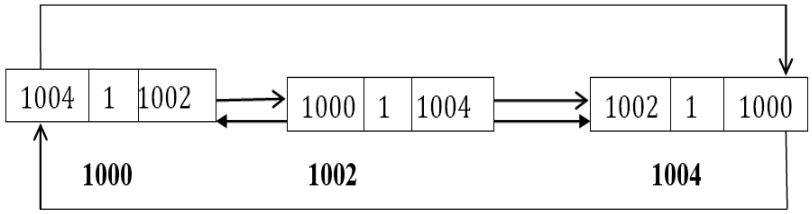


Fig: 1.10 Circular Doubly Linked Lists Representation

Structure Declaration:

```
Struct dnode
{
int data;
struct dnode *next;
struct dnode *prev;
} *head;
```



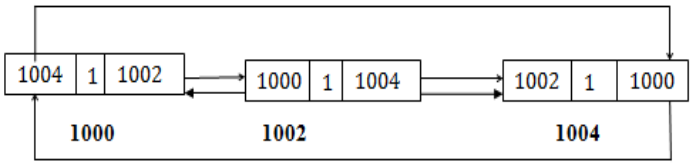
1. Creation of a circular doubly linked list :

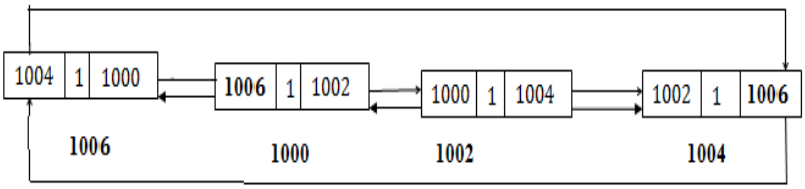
```
Void create(struct dnode *head, int n)
{
int n;
struct dnode *temp, *t;
printf("Enter number for the list");
scanf("%d",&num);
temp=malloc(sizeof(struct dnode));
temp-->data=num;
temp-->next=NULL;
```

```
temp-->prev=NULL;
head=temp;
while((n-1) != 0)
{
Printf("Enter the number");
Scanf("%d",&num);
t=temp;
temp-->next=malloc(sizeof(struct dnode));
temp=temp-->next;
temp-->data=num;
temp-->prev=t;
}
temp-->next=head;
head-->prev=temp;
}
```

2. Insertion :

```
Void ins(struct dnode *head, int num)
{
Struct dnode *temp,*new;
temp=head;
while(temp-->next != head)
{
temp=temp-->next;
}
new=malloc(sizeof(struct dnode));
new-->data=num;
temp-->next=new;
new-->prev=temp;
head-->prev=new;
}
```





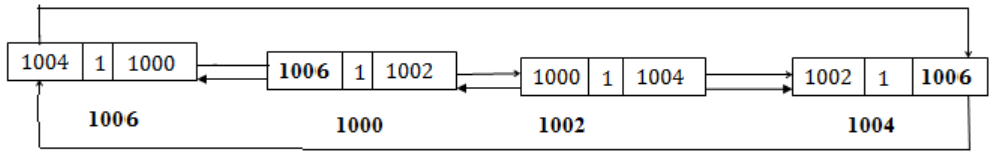
After Insertion At First

3. Deletion :

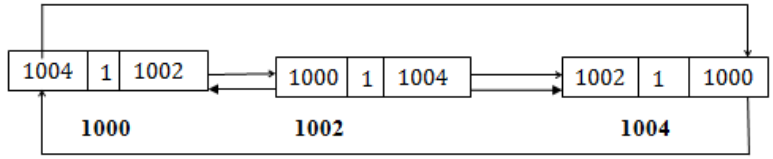
```
Void del(struct dnode *head, int num)
{
Struct dnode *temp;
temp=head;
```

```
while(temp-->data != num)
{
temp=temp-->next;
}
temp-->prev-->next=temp-->next;
temp-->next-->prev=temp-->prev;
free(temp);
}
```

Before Deletion At First



After Deletion At First



1.10 APPLICATIONS OF LISTS:

The Linked list is a data structure which makes use of dynamic memory. Hence it is possible to handle the list of any desired length using the linked list.

Various application of linked list are:

- The linked list is used for performing polynomial operations such as addition,multiplication evaluation and so on.
- The linked list is used for handling the Set Operations.
- The stack data structure can be implemented using linked List
- The Queue data structurs can be implemented using linked List

1.11 POLYNOMIAL MANIPULATION (INSERTION, DELETION,MERGE,TRAVERSAL)

- It is an application of linked list. A polynomial is a sum of terms where each term has a variable, coefficient and exponents.
- One can also perform various operations such as addition, multiplication, subtraction, division on these polynomials.

Representation of array polynomials:

- The index of an array will act as the exponent and the coefficient can be stored at the particular index and the constant value should be placed at the zeroth position.

Typedef struct
{
Int coeffarray[max];
Int highpower;
} *polynomial;

```
Void addpolynomial(const polynomial poly1, const polynomial poly2,polynomial polysum)
{
int i;
zeropolynomial(polysum);
polynum-->highpower=max(poly1-->highpower, poly2-->highpower);
for(i=polysum-->highpower; i>=0; i-- )
polysum-->coeffarray[i]=poly1-->coeffarray[i]+poly2-->coeffarray[i];
}

Void zeropolynomial(polynomial poly)
{
int i;
for(i=0; i<=maxdegree; i++)
poly-->coeffarray[i]=0;
poly-->highpower=0;
}
```

Representation of polynomial using linked list :

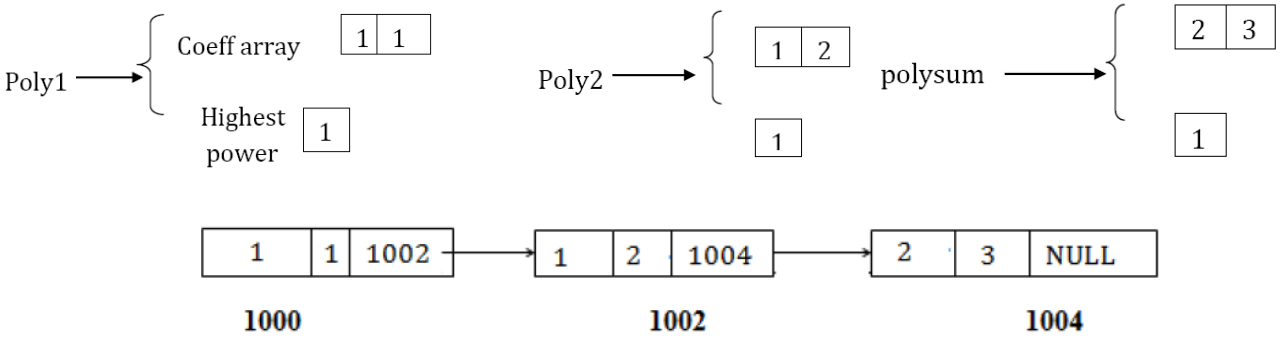


Fig: 1.11 Representation of polynomial

```
Struct node
{
int coefficient; int
exponent; struct
node *next;
} *head;

Void polyadd(struct node *head1, struct node *head2, struct node *head3)
{
int x;
head3=malloc(sizeof(struct node));
if(head1==head2==NULL)
printf("List is NULL");
else
x=max(head1-->exponent, head2-->exponent);
while(x>=0)
{
If(head1-->exponent==head2-->exponent)
{
head3-->coefficient=head1-->coefficient + head2-->coefficient;
head3-->exponent=head1-->exponent;
head3-->next=malloc(sizeof(struct node));
head1=head1-->next;
head2=head2-->next;
}
Else if(head1-->exponent > head2-->exponent)
{
head3-->coefficient=head1-->coefficient;
head3-->exponent=head1-->exponent;
head3-->next=malloc(sizeof(struct node));
}
Else if(head1-->exponent < head2-->exponent)
{
Head3-->coefficient=head2-->coefficient;
Head3-->exponent=head2-->exponent;
Head3-->next=malloc(sizeof(struct node));
}
Head3=head3-->next;
x--;
}
```

```
Struct polynode
{
int coeff;
int exp;
struct polynode *link;
} *x,*y,*z,*s;

Void polyadd(struct polynode *x, struct polynode *y, struct polynode *s)
{
Struct polynode *z;
If(x==NULL && y=NULL)
return;
while(x!=NULL && y!=NULL)
{
If(s==NULL)
{
s=malloc(sizeof(struct polynode));
z=s;
}
Else
{
z-->link=malloc(sizeof(struct polynode));
z=z-->link;
}
If(x-->exp < y-->exp)
{
z-->coeff = y-->coeff;
z-->exp=y-->exp;
y=y-->link;
}
Else
{
If(x-->exp > y-->exp)
{
z-->coeff=x-->coeff;
z-->exp=x-->exp;
x=x-->link;
}
Else
{
```

```
If(x-->exp==y-->exp)
{
z-->coeff=x-->coeff + y-->coeff;
z-->exp=x-->exp;
x=x-->link;
y=x-->link;
}
}
}
while(x!=NULL)
{
If(s==NULL)
{
s=malloc(sizeof(struct polynode));
z=s;
}
Else
{
z-->link=malloc(sizeof(struct polynode));
z=z-->link;
}
z-->coeff=x-->coeff;
z-->exp=x-->exp;
x=x-->link;
}
z-->link=NULL;
} // End of routine
```

PART-A

ABSTRACT DATA TYPES (ADTS) – LIST ADT

1. Define data structure. [L1]
2. Classify data structure [L1]
3. What are Abstract Data Types? **[Nov/Dec-2014, May/June-2015]** [L1]
4. List the ADT operations. [L1]
5. What is the advantage of an ADT?**[May/June-2015]** [L1]
6. Write the ADT for Set. [L2]
7. What is static and dynamic memory management? [L1]
8. Should arrays or linked lists be used for the following types of applications: - Justify your answer.

a. Many search operations in sorted list
b. Many search operation in unsorted list. **[May/June-2015]** [L3]

9. What are the differences between linear and non-linear data structure. [L1]

ARRAY-BASED IMPLEMENTATION – LINKED LIST IMPLEMENTATION

10. Define list. How it is implemented? [L1]
11. What is linked list? Give its types. [L1]
12. What is the need for the header? [L1]
13. Define singly linear linked list. [L1]
14. List the operations of linked list. [L1]
15. What are the advantages of linked list implementation of list? List the limitation in array based implementation of list ADT. [L3]
16. Compare linked list over arrays.**[Nov/Dec 2018]** [L2]
17. What is the advantage of linked list over arrays? [L2]

CIRCULARLY LINKED LISTS- DOUBLY-LINKED LISTS

18. Define singly circular linked list. [L1]
19. What is circular linked list?**[Nov/Dec-2014]** [L1]
20. Define doubly linear linked list. [L1]
21. Define doubly circular linked list. [L1]

APPLICATIONS OF LISTS – POLYNOMIAL MANIPULATION

22. What are the applications of linked list? **[May/June-2015]** [L1]
23. Define polynomial. How it is represented using linked list? [L2]

PART B

ABSTRACT DATA TYPES – LIST ADT – ARRAY-BASED IMPLEMENTATION

1. Write about abstract data type.[**Pg.No:2**] [L1]
2. Explain the array implementation of list ADT with routine and example. [**N/D 2018**] [L4]

(or)

Consider an array A [1, 2….n]. Given a position, write an algorithm to insert an element in the array. If the position is empty, the element is inserted easily. If the position is already occupied, the element should be inserted with minimum number of shifts. (Note: The elements can be shifted to left or right to make minimum number of moves). **[May/June 2014]**

LINKED LIST IMPLEMENTATION – SINGLY LINKED LISTS

3. Explain the linked list implementation of list ADT with routine and example. (or) Explain in detail about single linked list with routine and example. [**Pg.No:6**] [L1]

DOUBLY-LINKED LISTS

4. Define doubly linked list. Write the routine for its operations. (or) [L1]
Write an algorithm to perform insertion and deletion on a doubly linked list. Give the relevant coding in C. (or) **[May/June 2014]**
Describe the creation of double linked list and appending the list. Give the relevant coding in C. [**Pg.No:13**] **[Nov/Dec 2014]**

CIRCULARLY LINKED LISTS

5. Define circular linked list. Write the routine for its operations. **[May/June 2015]** [L1]
6. Define circular double linked list. Write the routine for its operations. [**N/D 2018**] [L1]

APPLICATIONS OF LISTS –

7. Explain Polynomial manipulation (or) [L1]
Write about the applications of list [**N/D 2018**]

PART -A

ABSTRACT DATA TYPES (ADTS) – LIST ADT

1. Define data structure.

The data structure can be defined as the collection of elements and all the possible operations which are required for those set of elements.

A data structure is a set of domains D, a set of Function F and set of axioms A. This triple (D, F, A) denotes the data structure d.

Or

The way of storing and organizing data in memory is known as data structure. They provide an orderly way to store and retrieve the elements in an efficient way.

2. Classify data structure

- Simple data structure or Primitive data structure
  - Example: int, char, float
- Compound data structure or Non primitive data structure

- Linear data structure
  - Example: list, stack, queue
- Non-Linear structure
  - Example: Trees, Graphs

3. What are Abstract Data Types?

The abstract data type is a triple of D-set of Domains, F- set of Functions, A- Axioms in which only ***what is to be done is mentioned but how is to be done is not mentioned***. In short, all the implementation details are hidden.

ADT = Type + Function name + Behavior of each function.

4. List the ADT operations.

b. Create - This operation creates the database.
c. Display - This operation is for displaying all the elements of the data structure.
d. Insertion - By this operation the element can be inserted at any desired position.
e. Deletion - By this operation any desired element can be deleted from the data structure.
f. Modification - this operation modifies the desired element‟s value by any other desired new value.

5. What is the advantage of an ADT?

- It can be reused in future programs.
- It reduces coding efforts.
- It ensured a robust data structure.
- Debugging is easier.
- Implementing of ADTs can be changed without requiring changes to the program that uses the ADTs.

6. Write the ADT for Set.

AbstractDataType SET
Instance: Set is a collection of integer type of elements.
Preconditions: None.
Operations:
1. Store ( ): This operation is for storing the integer element in a set.
2. Retrieve ( ): This operation is for retrieving the desired element from the given set.
3. Display ( ): This operation is for displaying the contents of set.

7. What is static and dynamic memory management?

The static memory management means allocating or de-allocating of memory at compilation time. The dynamic memory management means allocating or de-allocating of memory at running time (after compilation).

8. Should arrays or linked lists be used for the following types of applications: - Justify your answer?

a. Many search operations in sorted list

b. Many search operation in unsorted list.

Many search operations in sorted list

In this case, using arrays will save time because there will be fewer comparisons. Since the list is sorted the key value can be compared with the middle element of the array and if the key < array element, then left part of array should be searched; if key > array, right part of array should be searched. Best way is to implement a binary search algorithm.

Many search operation in unsorted list.

In this case array should be used because the elements of an array will be stored in consecutive memory locations whereas in the linked list the elements can be stored in any location and each node has to hold the address of the next element.

9. What are the differences between linear and non-linear data structure.

S.No	Linear Data Structure	Non-Linear Data Structure
1	It is collection of nodes which are logically adjacent in which logical adjacency is maintained by pointers.	Non-linear data structure can be constructed as a collection of randomly distributed set of data items joined together by using a special pointer.
2	They are constructed as a continuous arrangement of data element in the memory. The relationship of adjacency is maintained between the data elements.	In non-linear data structure the relationship of adjacency is not maintained between the data items.
3	Example: List, stack queue	Example: Tree, graph

ARRAY-BASED IMPLEMENTATION – LINKED LIST IMPLEMENTATION

10. Define list. How it is implemented?

- List is a collection of elements in sequential order.

- In memory we can store the list in two ways; one way is we can store the elements in sequential memory locations. This is known as arrays and the other way is we can use pointer or links to associate the elements sequentially.

Implementation ways

- Array based implementation
- Linked list based implementation

11. What is linked list? Give its types.

A linked list is a set of nodes where each node has two fields „data and a „link. Where „data filed stores the actual piece of information and „link field is used to point to next node. Basically link filed is nothing but the address node.

Linked list is a kind of series of data structures, which are not necessarily adjacent in memory. Each structure contains the element and a pointer to a record containing its successor.

Structure :

Data	Link

Types:

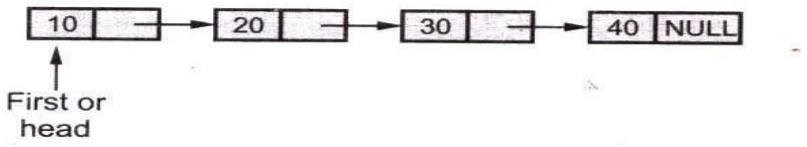
1. Singly linked list
2. Doubly linked list
3. Singly circularly linked list
4. Doubly circularly linked list

12. What is the need for the header?

Header of the linked list is the first element in the list and it stores the number of elements in the list. It points to the first data element of the list.

13. Define singly linear linked list.

This list consists of only one link, to point to next node or element. This is also called linear list because the last element points to nothing it is linear in nature. The last field of last node is NULL which means that there is no further list. The very first node is called head or first.



14. List the operations of linked list.

- Creation
- Display
- Insertion

- Deletion
- Searching

15. What are the advantages of linked list implementation of list? List the limitation in array based implementation of list ADT.

- Linked list facilities dynamic memory management by allowing elements to be added or deleted at any time during program execution.
- It ensures the efficient utilization of memory space as only that much amount of memory space is reserved as is required for storing the list of elements.
- It is easy to insert or delete elements in a linked list, unlike arrays, which require shuffling of other elements with each insert and delete operation.

Limitation of array implementation

- Insertion and deletion operation are expensive as it requires more data movement.
- Find and print list operation takes constant time.
- Even if the array is dynamically allocated, an estimate of the maximum size of the list is required which considerably wastes the memory space.

16. Compare linked list over arrays.

S.No	Linked list	Arrays
1.	Te linked list is a collection of nodes and each node is having one data filed and next link field.	The array is a collection of similar types of data elements. In array the data is always stored at some index of the array.
2.	Any element can be accessed by sequential access only.	Any element can be accessed randomly i.e. with the help of index of the array.
3.	Physically the data can be deleted.	Only logical deletion of the data is possible.
4.	Insertions and deletion of data is easy.	Insertions and deletion of data is difficult.
5.	Memory allocation is dynamic. Hence developer can allocate as well as de-allocate the memory and so no wastage of memory is there.	The memory allocation is static. Hence once the fixed amount of sixe is declared then that much memory is allocated. Therefore there is a chance of either memory wastage or memory shortage,

17. What is the advantage of linked list over arrays?

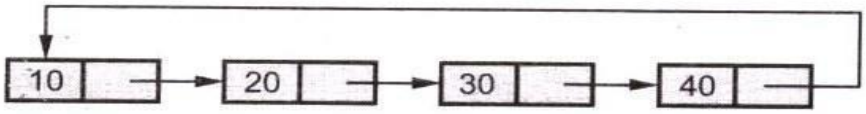
- The linked list makes use of the dynamic memory allocation. Hence the user can allocate or de-allocate the memory as per his requirements.

- On the other hand, the array makes use of the static memory location. Hence there are chances of wastage of the memory or shortage of memory.

CIRCULARLY LINKED LISTS- DOUBLY-LINKED LISTS

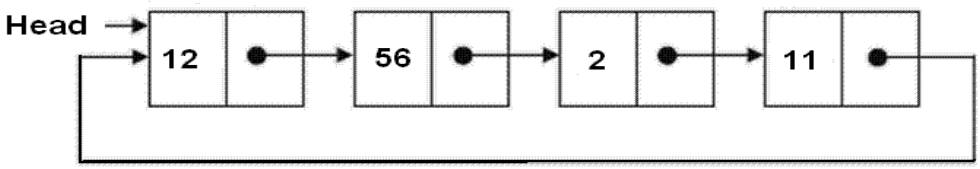
18. Define singly circular linked list.

In the singly circular linked list only one link is used to point to next element. The last node‟s link filed points to the first or head node.



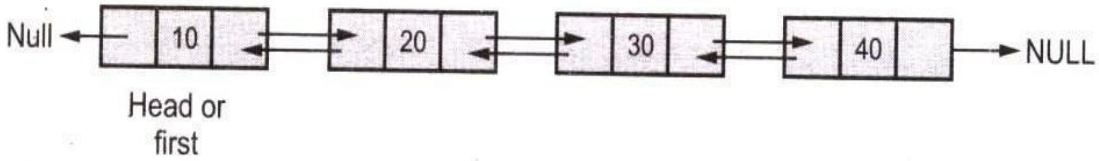
19. What is circular linked list?

The circular linked list (CLL) is similar to singly linked list except that the last node‟s next pointer points to first node. The list will be accessed like a chain. Circular linked list can be used to help the traverse the same list again and again if needed.



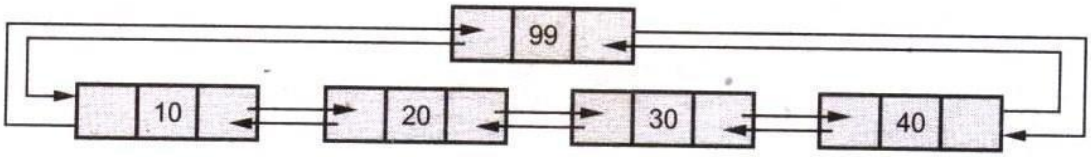
20. Define doubly linear linked list.

In this linked list each node has two pointers previous and next pointers. The previous pointer points to previous node and next pointer points to next node. Only in case of head node the previous pointer is obviously NULL and last node‟s next pointers points to NULL. This list is a linear one.



21. Define doubly circular linked list.

In circular doubly linked list the previous pointer of first node and the next pointer of last nodes are pointed to head node. Head node is a special node which may have any dummy data or it may have some useful information. Such as total number of nodes in the list which may be used to simplify the algorithms carrying various operations on the list.



APPLICATIONS OF LISTS – POLYNOMIAL MANIPULATION

22. What are the applications of linked list?

- It is used for performing polynomial operations such as addition, multiplication evaluation and so on.
- It is used for handling the set operations.
- The stack data structure can be implemented using linked list.
- The queue data structure can be implemented using linked list.

23. Define polynomial. How it is represented using linked list?

A polynomial is homogeneous ordered list of pairs <exponent, coefficient>, where each coefficient is unique.

Example:

3x2+5x+7

Linked list representation

The main fields of polynomial are coefficient and exponent, in linked list it will have one more filed called „link‟ field to point to next term in the polynomial. If there are „n‟ terms in the polynomial then „'n‟ such nodes have to be created.

The polynomial equation can be represented with linked list as follows:

Coefficient	Exponent	Next node link

```
struct polynomial
{
  int coefficient;
  int exponent;
  struct polynomial *next;
}
```