JEPPIAAR INSTITUTE OF TECHNOLOGY

"Self Belief | Self Discipline | Self Respect"

DEPARTMENT OF

COMPUTER SCIENCE AND ENGINEERING

LECTURE NOTES

CS8391 / DATA STRUCTURES

(2017 Regulation)

Year/Semester: II / III

Prepared by

Dr. K. Tamilarasi, Professor / Dept. of CSE.

CHAPTER ONE

LINEAR DATA STRUCTURES – LIST

Linear Data Structures describes about 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 operations (Insertion, Deletion, Merge, Traversal).

Data Structures

Data Structures is a **way of organizing the data** which also includes several operations to perform on that data.

Applications of Data Structures

- List: used in implementation of File allocation Table (FAT), Process Control Block (PCB) etc.
- Stack: It is always implicitly used in a program, whether or not we declare it explicitly, during the program execution.
- Queue: It is used to order the task as they arrive to get the attention of the CPU.
- Tree: Directory Structure is implemented using tree DS.
- Graph: used in Networking and mapping of tasks to processors in multiprocessor Platforms.

Data structures are widely applied in the areas like

- Compiler Design
- Operating System
- DBMS
- Artificial Intelligence
- Graphics
- Networking Etc.

Characteristics of Data Structures

- Represents logical relationship between various data elements.
- Helps in efficient manipulation of stored data.
- Allows the programs to process data easily / effectively.

Classification of Data Structures

- Primitive Data Structures: All the fundamental DS that can be directly manipulated by the machine level instructions.
- Non Primitive Data Structures: refer to all those that are derived from one or more primitive DS.
- Linear Data Structures: arranged in sequential form
- Non Linear Data Structures: not sequential order. Ex : data elements in a hierarchical fashion (Tree).

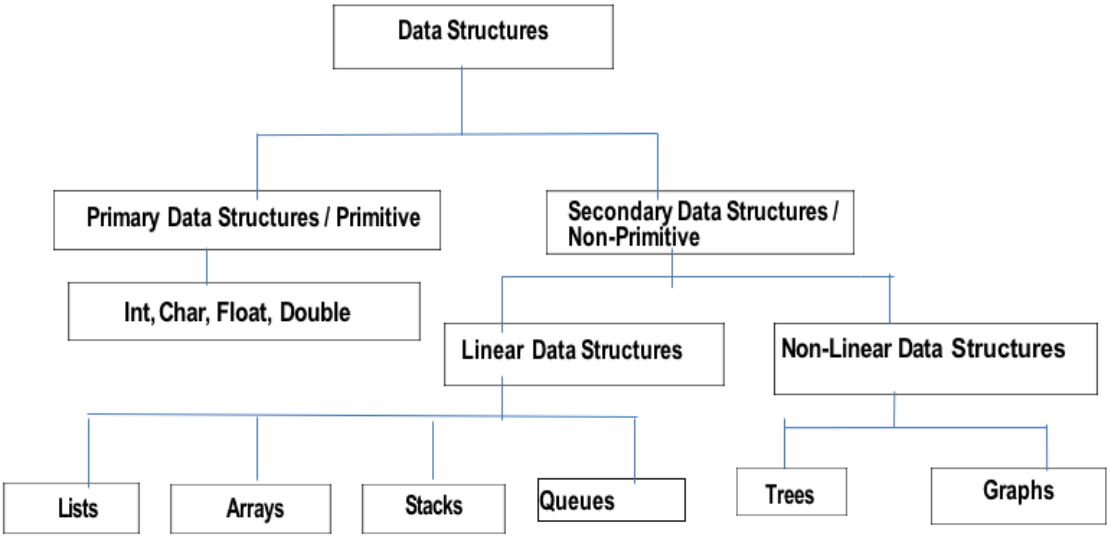


Fig 1.1 Classification of Data Structures

Abstract Data Types (ADTs)

ADT is defined as a mathematical model with a set of operations.

Basic Idea of ADT: The operations are written once in the program. Any other part of the program can access this operation by calling the appropriate function.

Examples

- Objects such as lists, sets, graph etc along with their operations are ADTs.
- Set of integers together with the operations Union, Intersection and difference form an example ADT.
- Similarly real Boolean have operations associated with them. So it is ADTs.

Advantages of ADT

1. Modularity
2. Reuse
3. Easy to understand
4. Implementation of ADTs can be changed anytime without changing the program that uses it.

The List ADT

List is an ordered set of elements. General Form of List ADT is

A1, A2, A3, ….AN , Where N is the size of the List.

A1 - First element of the list, A2 - Second element of the list, AN - Last element of the list

If the element at position i is Ai, Ai-1is the predecessor element, Ai+1 is the successor element.

Various operations performed on List

1. Insert (X, 3)- Insert the element X after the position 3.
2. Delete (X) - The element X is deleted
3. Find (X) - Returns the position of X.
4. Next (i) - Returns the position of its successor element i+1.
5. Previous (i) Returns the position of its predecessor i-1.

6. Print list - Contents of the list is displayed.
7. Makeempty- Makes the list empty.

Implementation of list ADT

1. Array based Implementation
2. Linked List based implementation

Array Implementation of list

Array is a collection of specific number of same type of data stored in consecutive memory locations. Array is a static data structure.

i.e., size of the array should be allocated in advance and the size is fixed.

- Insertion and Deletion operation are expensive as it requires more data movements. Worst case it requires O(N) Movements. Best case it requires O(1).

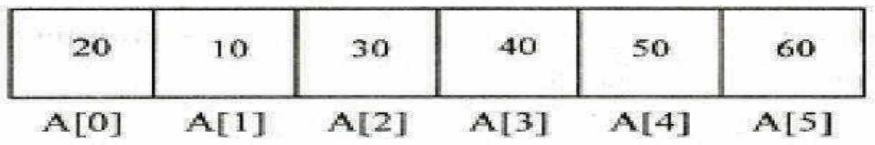


Fig 1.2 Array Implementation of List

The basic operations performed are

- Creation of List.
- Insertion / Deletion of data from the List
- Display all data in the List
- Searching for a data in the list

Creation of List: Create operation is used to create the list with "n" number of elements. If "n" exceeds the array's maxsize, then elements cannot be inserted into the list. Otherwise the array elements are stored in the consecutive array locations (i.e.) list [0], list [1] and so on.

Insertion / Deletion of data in the List: Insert / Delete operation is used to Insert/ Delete an element at particular position in the existing list. Inserting/ deleting the element in the last position of an array is easy.

But inserting/ Deleting the element at a particular position in an array is quite difficult since it involves all the subsequent elements to be shifted one position to the right / left.

Display/ Traversal all data in the List: Traversal is the process of visiting the elements in an array. Display( ) operation is used to display all the elements stored in the list from the index 0 to n-1.

Searching for a data in the list: Search( ) operation is used to determine whether a particular element is present in the list or not.

Procedure to create / insert / delete

Create	Insert	Delete

void Create()
{ int i;
printf("Enter number of elements");
scanf("%d",&n);
printf("Enter the array elements");
for(i=0; i<n; i++)
{scanf("%d", &list[i]); }
Display();
}
	void Insert()
	{ int i, data, pos;
	printf("Enter data to insert");
	scanf("%d", &data);
	printf("Enter the position to insert");
	scanf("%d", &pos);
	for(i = n-1; i >= pos-1 ; i--)
	{list[i+1] = list[i];}
	// **swap** //
	list[pos-1] = data;
	n= n+1; Display(); }
		void Delete( )
		{ int i,pos;
		printf("Enter the position to delete”);
		scanf("%d", &pos);
		printf("The data deleted is: %d", list[pos-1]);
		for(i=pos-1;i<n-1; i++)
		{list[i]=list[i+1]; }
		n=n-1;
		Display(); }

Procedure to display / search

Display	Search

void Display()
{
int i;
printf("Elements in the array");
for(i=0;i<n;i++)
printf("%d",list[i]);
}
	void Search()
	{ int search, i, count = 0;
	printf("Enter the element to be searched");
	scanf("%d", &search);
	for(i=0; i<n; i++)
	{ if(search == list[i])
	count++; }
	if(count==0)
	printf("Element not present ");
	else
	printf("Element present"); }

//Main Program //

```
#include<stdio.h>   #include<conio.h>
 #define maxsize 10
int list[maxsize],n;      //  Declaration of Array//
void Create();  void Insert(); void Delete(); void Display();  void Search();
 void main()
{ int choice;
 do
{printf("Array Implementation”);
printf("\t1.Create\n");printf("\t2.Insert\n");
printf("\t3.Delete\n");printf("\t4.Display\n");
printf("\t5.Search\n");  printf("\t6.Exit\n");
printf("\nEnter your choice:\t"); scanf("%d",&choice);
```

```
switch(choice)
{ case 1: Create();      break;
case 2: Insert();         break;
case 3: Delete();         break;
case 4: Display();        break;
case 5: Search();
          break;
default: printf("Enter option between 1 - 5"); break; }  }
while(choice<6);  }
```

Advantages of array implementation

1. The elements are faster to access
2. Searching an element is easier

Limitation of array implementation

1. An array uses consecutive memory locations.
2. The number of elements in the array is fixed.
3. Insertion and deletion operation in array are expensive.

Applications of arrays

Arrays are particularly used in programs that require storing large collection of similar type data elements.

Linked List based implementation

A Linked list is an ordered collection of elements. **Each element in the list is referred as a node**. Each node contains two fields namely,

1. **Data field-**The data field contains the actual data of the elements to be stored in the list.
2. **Next / Pointer field**- The next field contains the address of the next node in the list

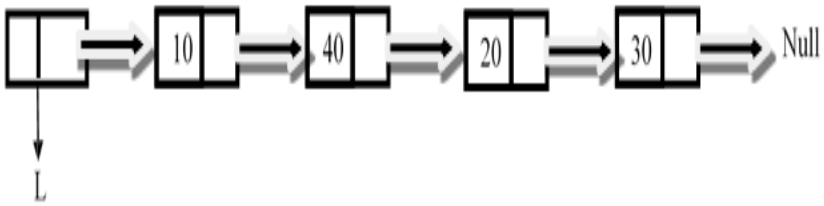


Fig 1.3 Linked Implementation of List

The nodes in the linked list are not necessarily adjacent in memory.

Advantages of Linked list

1. Insertion and deletion of elements can be done efficiently
2. It uses dynamic memory allocation
3. Memory utilization is efficient compared to arrays

Disadvantages of linked list

1. Linked list does not support random access
2. Extra Memory space is required to store next field
3. Searching takes time compared to arrays

Types of Linked List

1. Singly Linked List or One-Way List
2. Doubly Linked List or Two-Way Linked List
3. Circular Linked List

Differences between Array based and Linked List based Implementation of List

	Array	**Linked List**
Definition	Array is a collection of elements having same data type with common name	Linked list is an ordered collection of elements which are connected by links/pointers
Access	Elements can be accessed using index / subscript	Sequential access using pointers
Memory structure	Elements are stored in contiguous memory locations	Elements are stored at available memory space

Insertion & Deletion	Takes more time in array	Are fast and easy
Memory Allocation	Memory is allocated at compile time i.e static memory allocation	Memory is allocated at run time i.e dynamic memory allocation
Types	1D,2D and multi-dimensional	SLL, DLL and circular linked list
Dependency	Each element is independent	Each node is dependent on each other as address part contains address of next node in the list

Dynamic allocation

The process of allocating memory to the variables during execution of the program or at run time is known as dynamic memory allocation.

When we do not know memory requirements in advance, we can use Dynamic memory allocation.

C language has four library routines to support Dynamic allocation.

To use dynamic memory allocation functions, include the header file stdlib.h.

Memory allocation/de-allocation functions

Function	Task
malloc()	Allocates memory and returns a pointer to the first byte of allocated space
calloc()	Allocates space for an array of elements, initializes them to zero and returns a pointer to the memory
free()	Frees previously allocated memory
realloc()	Alters the size of previously allocated memory

Fig 1.4 memory allocation / De-allocation fucntions

malloc() and calloc()

malloc()	calloc()
Syntax: ptr =(cast-type*) malloc (byte-size);	Syntax:ptr=(cast-type*) calloc (n,elem_size);
Returns a pointer type of void after memory allocation, which can be casted to any form. However, if the space is insufficient, allocation fails and returns a NULL pointer	The malloc() function allocates a single block of memory. Whereas, calloc() allocates multiple blocks of memory and initializes them to zero.
Example : ptr = (int*) malloc(100 * sizeof(int));	Example: ptr = (float*) calloc(25, sizeof(float));
Considering the size of int is 4 bytes, this statement allocates 400 bytes of memory. And, the pointer ptr holds the address of the first byte in the allocated memory.	This statement allocates contiguous space in memory for 25 elements each with the size of float

free() and realloc()

free()	realloc()
Syntax: free(ptr);	Syntax : ptr = realloc(ptr,newsize);
the memory is returned back to the free list within the heap.	Relloactes the previous memory

	ptr = (int*) malloc(100 * sizeof(int)); // reserved 400 bytes
	ptr = realloc(ptr, 200 * sizeof(int)); // reserved 800 bytes.

Example Program : for sum of n numbers using malloc

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

```
int main()
{ int n, i, *ptr;
   printf("Enter number of elements: ");
   scanf("%d", &n);
   ptr = (int*) malloc(n * sizeof(int));
   if(ptr == NULL)
   { printf("Error! memory not allocated.");  }
   else { printf("Memory allocated");  }
   free(ptr);     return 0;
}
```

Singly Linked List

A singly linked list is a linked list in which each node contains **data field and only one link field** pointing to the next node in the list.

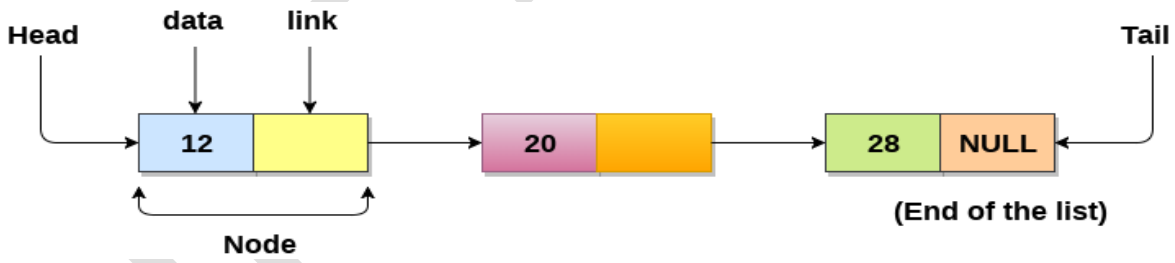


Fig 1.5 Singly Linked List

Head node which always points to the first node. Head node is also called as **sentinel node**.

Basic operations on a singly linked list are

1. Insert – Inserts a new node in the list.
2. Delete – Deletes any node from the list.

3. Find – Finds the position (address) of any node in the list.
4. FindPrevious - Finds the position (address) of the previous node.
5. FindNext - Finds the position(address) of the next node in the list.
6. Display - displays the data in the list
7. Search - finds whether an element is present in the list or not

Insertion: The insertion into a singly linked list can be performed in 3 ways.

1. **Insertion at beginning:** It involves **inserting any element at the front** of the list.

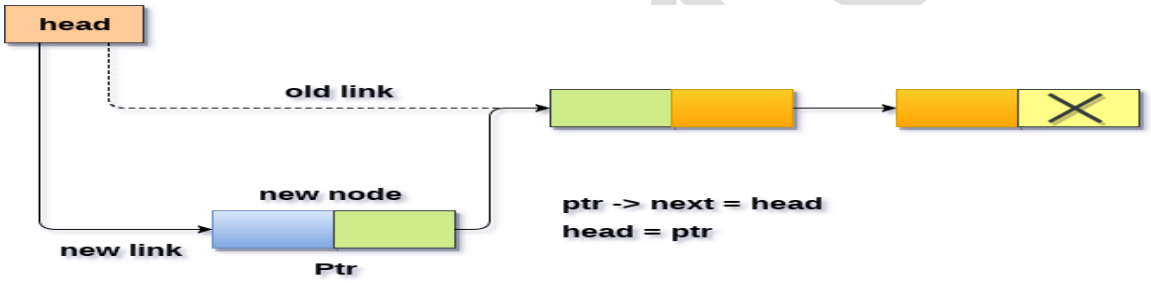


Fig 1.6 Inserting a new element into a Singly Linked List at beginning

2. **Insertion at end of the list:** It involves insertion at the last of the linked list. It can be done in 2 ways
   a. The node is being **added to an empty list** (only one node in the list)
   b. The node can be inserted as the **last one**.

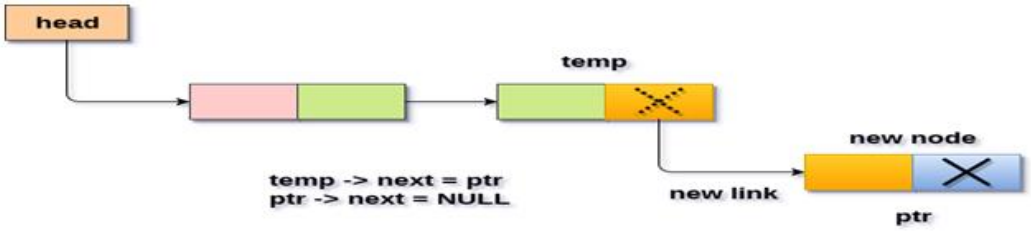


Fig 1.7 Inserting a new element into a Singly Linked List at end of the list

3. **Insertion at a given position** (after specified node): It involves insertion after the specified node of the linked list.

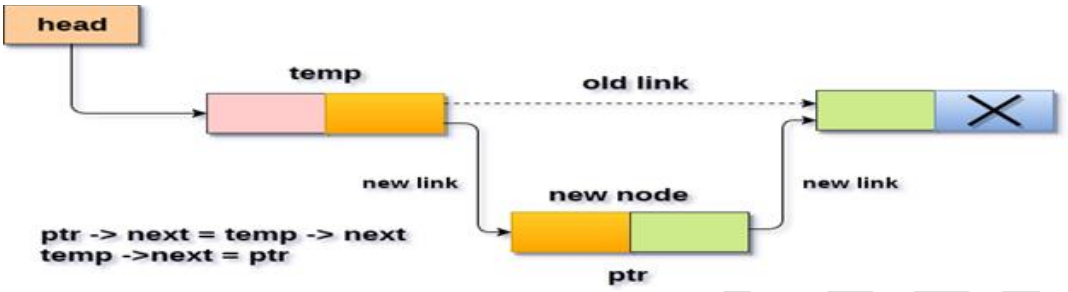


Fig 1.8 Inserting a new element at a given position

Procedure for Inserting a new element into a singly linked list

Insertion at beginning	**Insertion at end of the list**	**Insertion at a given position**

void beginsert()
{struct node *ptr;
int item;
ptr = (struct node *) malloc(sizeof(struct node));
if(ptr == NULL)
{printf("overflow"); }
else
{printf("Enter value");
scanf("%d",&item);
ptr→data = item;
ptr→next = head;
head = ptr;
printf("Node inserted");
} }
	void lastinsert()
	{struct node *ptr, *temp;
	int item;
	ptr = (struct node*) malloc(sizeof(struct node));
	if(ptr == NULL)
	{printf("overflow"); }
	else
	{printf("Enter value");
	scanf("%d",&item);
	ptr→data = item;
	if(head == NULL)
	{**ptr → next = NULL;**
	head = ptr;
	printf("Node inserted");}
		void randominsert()
		{int i,loc,item;
		struct node *ptr, *temp;
		ptr = (struct node *) malloc (sizeof(struct node));
		if(ptr == NULL)
		{printf("overflow"); }
		else
		{printf("Enter value");
		scanf("%d",&item);
		ptr→data = item;
		printf("Enter the location after which you want to insert ");
		scanf("%d",&loc);

	else
	{temp = head;
	while (temp → next != NULL)
	{temp = temp → next; }
	temp→next = ptr;
	ptr→next = NULL;
	printf("Node inserted");}
	}
	}
		temp=head;
		for(i=0;i<loc;i++)
		{ temp = temp→next;
		if(temp == NULL)
		{printf("can't insert");
		return; }
		}
		ptr →next = temp→next;
		temp →next = ptr;
		printf("Node inserted");
		}
		}

Deletion: The deletion into a singly linked list can be performed in 3 ways.

1. **Deletion at beginning: deleting an element at the front** of the list.

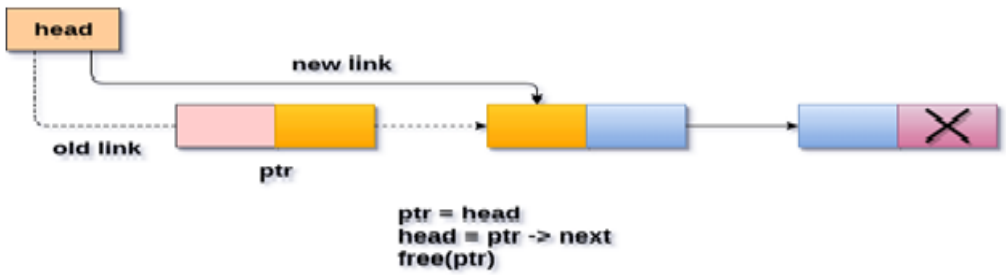


Fig 1.9 Deleting an element at the beginning

2. **Deletion at end of the list:** It can be done in 2 ways
   a. There is **only one node in the list** and that needs to be deleted.
   b. There are **more than one node in the list** and the last node of the list will be deleted.

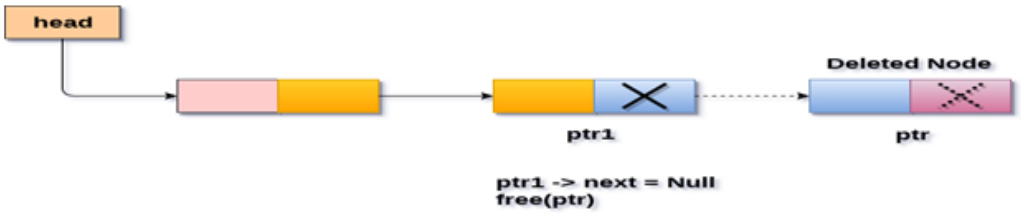


Fig 1.10 Deleting a node at end of the list

3. **Deletion at a given position** (after specified node): It involves deletion after the specified node of the linked list.

Fig 1.11 Deleting a node at a given position

Procedure for Deleting a node into a singly linked list

Deletion at beginning	**Deletion at end of the list**	**Deletion at a given position**

void begin_delete()
{ struct node *ptr;
if(head == NULL)
{ printf("List is empty");
}
else
{**ptr = head;**
head = ptr→next;
	void last_delete()
	{ struct node *ptr,*ptr1;
	if(head == NULL)
	{printf("list is empty"); }
	else if(head → next == NULL)
	{ head = NULL;
		void random_delete()
		{struct node *ptr,*ptr1;
		int loc,i;
		printf("Enter the location of the node after which you want to perform deletion");
		scanf("%d",&loc);
		ptr=head;

free(ptr);
printf("Node deleted from the begining ...");
}
}
	free(head); printf("Only node of the list deleted ..."); }
	else
	{ptr = head;
	while(ptr→next != NULL)
	{ptr1 = ptr;
	ptr = ptr →next; }
	ptr1→next = NULL;
	free(ptr);
	printf("Deleted Node from the last ...");
	}
	}
		for(i=0;i<loc;i++)
		{ptr1 = ptr;
		ptr = ptr→next;
		if(ptr == NULL)
		{printf("Can't delete");
		return; }
		}
		ptr1 →next = →next;
		free(ptr); printf("Deleted node %d ", loc+1);
		}

Traversing / Display in singly linked list

Traversing means visiting each node of the list once in order to perform some operation on that. Display the content of linked list.

Search in singly linked list

Comparing each node data with the item to be searched and return the location of the item in the list if the item found else return null.

Procedure for Display and Search

Display	Search
void display() { struct node *ptr; ptr = head;	void search() { struct node *ptr; int item,i=0,flag; ptr = head;

if(ptr == NULL)
{ printf("Empty List "); }
else
{ printf("printing values");
while (ptr!=NULL)
{printf("%d",ptr→data);
ptr = ptr -> next; }
}
}
	if(ptr == NULL)
	{printf("Empty List"); }
	else
	{printf("Enter item to search");
	scanf("%d",&item);
	while (ptr!=NULL)
	{if(ptr→data == item)
	{printf("item found at location %d ",i+1);
	flag=0; } else { flag=1; }
	i++; ptr = ptr -> next; }
	if(flag==1)
	{ printf("Item not found\n"); } } }

Singly Linked List main program

```
#include<stdio.h>
#include<stdlib.h>
struct node
{ int data;
 struct node *next;
 };  struct node *head;
void beginsert (); void lastinsert (); void randominsert();
void begin_delete(); void last_delete(); void random_delete();
void display();  void search();
void main ()
{ int choice =0;
  while(choice != 9)
```

```
{ printf("Main Menu");
  printf("Choose one option from the following list ...");

printf("1.Insert in beginning 2.Insert at last 3.Insert at any random location 4.Delete from Beginning 5.Delete from last 6.Delete node after specified location 7.Search for an element 8.Show 9.Exit");
printf("Enter your choice?");    scanf("%d", &choice);
switch(choice)
{  case 1:   beginsert(); break;
   case 2:  lastinsert(); break;
   case 3:  randominsert();   break;
   case 4:  begin_delete();  break;
   case 5:  last_delete();    break;
   case 6:  random_delete(); break;
   case 7:  search();   break;
   case 8:  display(); break;
   case 9:   exit(0);
             break;
   default:  printf("Please enter valid choice..");   }
   }       }
```

Complexity

	SLL Time Complexity				Space Complexity
	Access	Search	Insertion	Deletion	
Average	O(n)	O(n)	O(n)	O(n)	
Worst	O(n)	O(n)	O(1)	O(1)	O(n)

Fig 1.12 Complexity of SLL Operation

Advantages of SLL

1. The elements can be accessed using the next link
2. Occupies less memory than DLL as it has only one next field.

Disadvantages of SLL

1. Traversal in the backwards is not possible
2. Less efficient to for insertion and deletion.

Doubly Linked List

A doubly linked list is a linked list in which each node has **three fields namely** Data, Next, Prev. The prev part of the first node and the next part of the last node will always contain null indicating end in each direction.

Fig 1.13 Doubly Linked List

Basic operations on a doubly linked list are

1. Insert – Inserts a new node in the list.
2. Delete – Deletes any node from the list.

3. Find – Finds the position (address) of any node in the list.
4. Display - displays the data in the list
5. Search - finds whether an element is present in the list or not

Insertion: The insertion into a doubly linked list can be performed in 3 ways.

1. **Insertion at beginning:** It involves **inserting any element at the front** of the list.

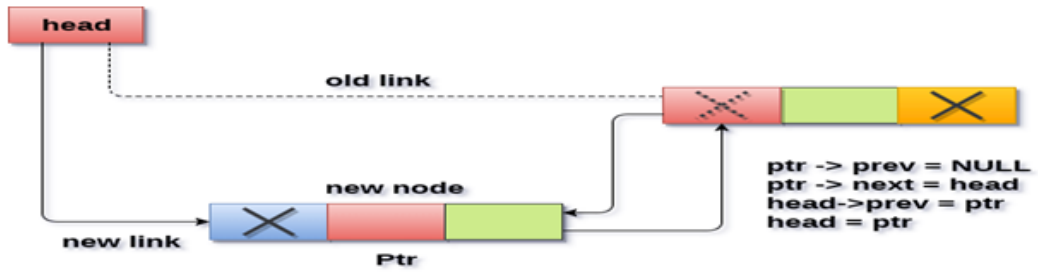


Fig 1.14 Inserting a new element into a doubly Linked List at beginning

2. **Insertion at end of the list:** It involves insertion at the last of the linked list. It can be done in 2 ways
   c. The node is being **added to an empty list** (only one node in the list)
   d. The node can be inserted as the **last one**.

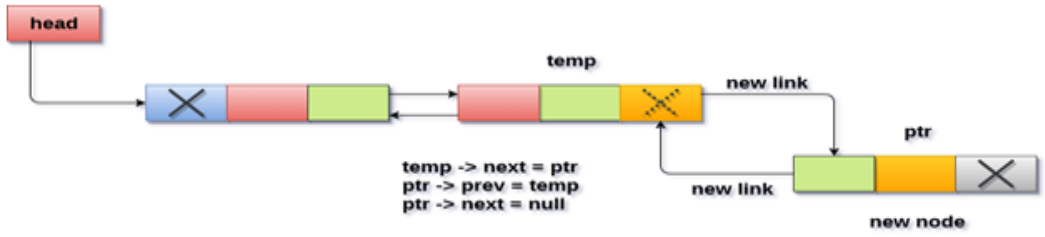


Fig 1.15 Inserting a new element into a doubly Linked List at end of the list

3. **Insertion at a given position** (after specified node): It involves insertion after the specified node of the linked list.

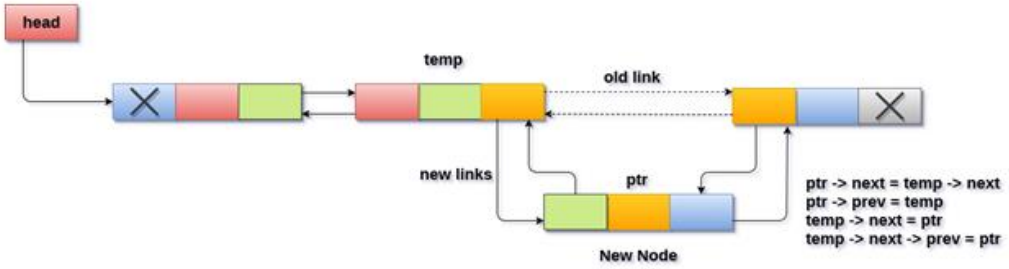


Fig 1.16 Inserting a new element at a given position

Procedure for Inserting a new element into a doubly linked list

Insertion at beginning	**Insertion at end of the list**	**Insertion at a given position**

void insertion_beginning()
{struct node *ptr;
int item;
ptr = (struct node *) malloc(sizeof(struct node));
if(ptr == NULL)
{printf("Overflow"); }
else
{ printf("Enter value");
scanf("%d",&item);
if(head==NULL)
{ ptr→next = NULL;
ptr→prev=NULL;
ptr→data=item;
	void insertion_last()
	{ struct node *ptr,*temp;
	int item;
	ptr = (struct node *) malloc(sizeof(struct node));
	if(ptr == NULL)
	{printf("Overflow"); }
	else
	{ printf("Enter value");
	scanf("%d",&item);
	ptr->data=item;
	if(head == NULL)
	{ ptr→next = NULL;
	ptr→prev = NULL;
	head = ptr; }
		void insertion_specified()
		{struct node *ptr, *temp;
		int item,loc,i;
		ptr = (struct node *) malloc(sizeof(struct node));
		if(ptr == NULL)
		{printf("Overflow");
		}
		else
		{ temp=head;
		printf("Enter location");
		scanf("%d",&loc);
		for(i=0;i<loc;i++)

head=ptr; }
else
{ ptr→data=item;
ptr→prev=NULL;
ptr→next = head;
head→prev=ptr;
head=ptr; }
printf("Node inserted"); }
}
	else
	{ temp = head;
	while(temp→next!=NULL)
	{temp = temp->next; }
	temp→next = ptr;
	ptr →prev=temp;
	ptr→next = NULL; }
	}
	printf("node inserted");
	}
		{temp = temp→next;
		if(temp == NULL)
		{printf("There are less than %d elements", loc); return; }
		}
		printf("Enter value");
		scanf("%d",&item);
		ptr→data = item;
		ptr→next=temp→next;
		ptr→prev = temp;
		temp→next = ptr;
		temp→next→prev=ptr;
		printf("node inserted");
		} }

Deletion: The deletion into a doubly linked list can be performed in 3 ways.

1. **Deletion at beginning: deleting an element at the front** of the list.

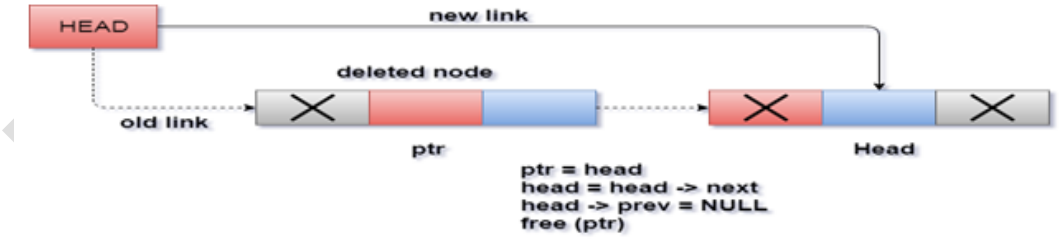


Fig 1.17 Deleting an element at the beginning

2. **Deletion at end of the list:** It can be done in 2 ways

   c. There is **only one node in the list** and that needs to be deleted.

   d. There are **more than one node in the list** and the last node of the list will be deleted.

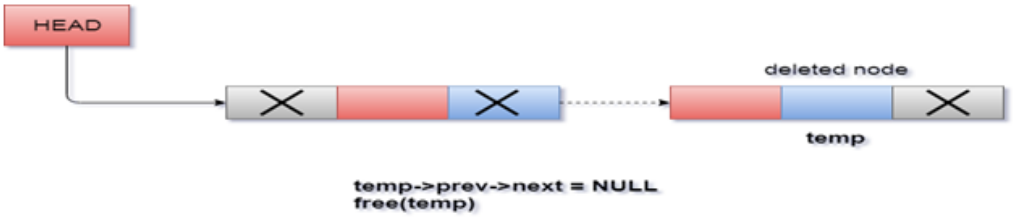


Fig 1.18 Deleting a node at end of the list

3. **Deletion at a given position** (after specified node): It involves deletion after the specified node of the linked list.

Fig 1.19 Deleting a node at a given position

Procedure for Deleting a node into a doubly linked list

Deletion at beginning	**Deletion at end of the list**	**Deletion at a given position**

void deletion_beginning()
{ struct node *ptr;
if(head == NULL)
{printf("Underflow"); }
else if(head→next == NULL)
{ head = NULL;
free(head);
printf("node deleted");
	void deletion_last()
	{ struct node *ptr;
	if(head == NULL)
	{printf("Underflow"); }
	else if(head→next == NULL)
	{ head = NULL;
	free(head);
	printf("node deleted");
		void deletion_specified()
		{struct node *ptr, *temp;
		int val;
		printf("Enter the data after which the node is to be deleted : ");
		scanf("%d", &val);
		ptr = head;

}
else
{ ptr = head;
head = head → next;
head → prev = NULL;
free(ptr);
printf("node deleted");
}
}
	}
	else
	{ ptr = head;
	if(ptr→next != NULL)
	{ ptr = ptr → next; } ptr → prev → next = NULL;
	free(ptr);
	printf("node deleted");
	}
	}
		while(ptr → data!= val) ptr = ptr→next;
		if(ptr -> next == NULL)
		{printf("Can't delete");}
		else if(ptr → next → next == NULL)
		{ptr →next = NULL; }
		else
		{temp = ptr → next;
		ptr → next = temp → next;
		temp → next → prev = ptr;
		free(temp);
		printf("node deleted");
		} }

Traversing / Display in doubly linked list

Traversing means visiting each node of the list once in order to perform some operation on that. Display the content of linked list.

Search in doubly linked list

Comparing each node data with the item to be searched and return the location of the item in the list if the item found else return null.

Procedure for Display and Search

Display	**Search**
void display() { struct node *ptr;	void search () { struct node *ptr; int item,i=0,flag;

```
printf("printing values");
ptr = head;
 while(ptr != NULL)
{printf("%d", ptr->data);
  ptr=ptr->next;    }
}
```

```
ptr = head;
if(ptr == NULL)
{ printf("\nEmpty List\n");      }
else
{printf("Enter item to search?");
scanf("%d",&item);
while (ptr!=NULL)
{ if(ptr->data == item)
{printf("item found at location %d ",i+1);
 flag=0;     break;       }
else    {    flag=1;        }
i++;   ptr = ptr -> next;   }
if(flag==1)
{ printf("Item not found");    }
  }     }
```

Doubly Linked List main program

```
#include<stdio.h>     #include<stdlib.h>
struct node
{  struct node *prev;
   struct node *next;
   int data;   };   struct node *head;
void insertion_beginning();  void insertion_last();
 void insertion_specified();  void deletion_beginning();
void deletion_last();  void deletion_specified();
void display();   void search();
```

```
void main ()
{  int choice =0;
   while(choice != 9)
   {  printf("Main Menu");
      printf("\nChoose one option from the following list ...\n");
printf("1.Insert in beginning 2.Insert at last 3.Insert at any random location 4.Delete from Beginning 5.Delete from last 6.Delete the node after the given data 7.Search 8.Show 9.Exit");
printf("Enter your choice?");
scanf("%d", &choice);
switch(choice)
{  case 1: insertion_beginning();          break;
   case 2: insertion_last();              break;
   case 3:   insertion_specified();  break;
   case 4:   deletion_beginning();   break;
   case 5:  deletion_last();       break;
   case 6:  deletion_specified();   break;
   case 7:  search();     break;
   case 8:  display(); break;
   case 9:  exit(0);  break;
   default: printf("Please enter valid choice..");          }
   }   }
```

Advantages of DLL

The DLL has two pointer fields. One field is prev link field and another is next link field. Because of these two pointer fields we can access any node efficiently whereas in SLL

only one link field is there which stores next node which makes accessing of any node difficult.

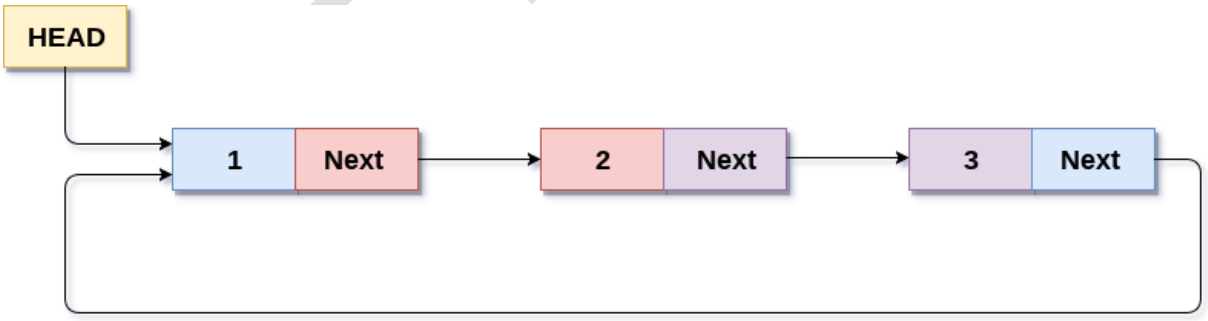
Disadvantages of DLL

The DLL has two pointer fields. One field is prev link field and another is next link field. Because of these two pointer fields, more memory space is used by DLL compared to SLL

Circular Singly Linked List

- In a circular singly linked list, the last node of the list contains a pointer to the first node of the list.
- We traverse a circular singly linked list until we reach the same node where we started.
- The circular singly liked list has no beginning and no ending.
- There is no null value present in the next part of any of the nodes.

The following image shows a circular singly linked list.



Circular Singly Linked List

C Program to perform all operation on Circular Singly Linked List

```
#include<stdio.h>
#include<stdlib.h>
struct node
{   int data;
   struct node *next;
};
struct node *head;

void beginsert ();   void lastinsert ();  void randominsert();
void begin_delete();  void last_delete();  void random_delete();
void display();  void search();
void main ()
{      int choice =0;
   while(choice != 7)
   {   printf("\nChoose one option from the following list ...\n");
      printf("\n1.Insert in begining\n2.Insert at last\n3.Delete from Beginning\n4.Delete
from last\n5.Search for an element\n6.Show\n7.Exit\n");
       printf("\nEnter your choice?\n");
       scanf("\n%d",&choice);
       switch(choice)
       {    case 1:
```

```
            beginsert();
            break;
            case 2:    lastinsert();        break;
            case 3:   begin_delete();     break;
            case 4:   last_delete();        break;
            case 5:  search();               break;
            case 6:    display();           break;
            case 7:    exit(0);             break;
            default:
            printf("Please enter valid choice..");
        }     }
}
void beginsert()
{
   struct node *ptr,*temp;
   int item;
   ptr = (struct node *)malloc(sizeof(struct node));
   if(ptr == NULL)
   {         printf("\nOVERFLOW");       }
   else
   {         printf("\nEnter the node data?");
```

```
        scanf("%d",&item);
        ptr -> data = item;
        if(head == NULL)
        {   head = ptr;
            ptr -> next = head;          }
        else
        {     temp = head;
            while(temp->next != head)
                temp = temp->next;
            ptr->next = head;
            temp -> next = ptr;
            head = ptr;          }
        printf("\nnode inserted\n");
    }
}
void lastinsert()
{
    struct node *ptr,*temp;
    int item;
    ptr = (struct node *)malloc(sizeof(struct node));
    if(ptr == NULL)
```

```
    {       printf("\nOVERFLOW\n");      }
    else
    {
       printf("\nEnter Data?");
       scanf("%d",&item);
       ptr->data = item;
       if(head == NULL)
       {    head = ptr;              ptr -> next = head;          }
       else
       {
          temp = head;
          while(temp -> next != head)
          {                temp = temp -> next;             }
          temp -> next = ptr;
          ptr -> next = head;
       }
       printf("\nnode inserted\n");
    }
}
```

```
void begin_delete()
{
   struct node *ptr;
   if(head == NULL)
   {          printf("\nUNDERFLOW");        }
   else if(head->next == head)
   {  head = NULL;
      free(head);
      printf("\nnode deleted\n");       }

   else
   {   ptr = head;
      while(ptr -> next != head)
          ptr = ptr -> next;
      ptr->next = head->next;
      free(head);
      head = ptr->next;
      printf("\nnode deleted\n");
     }
}
```

```
void last_delete()
{
   struct node *ptr, *preptr;
   if(head==NULL)
   {          printf("\nUNDERFLOW");      }
   else if (head ->next == head)
   {  head = NULL;
      free(head);
      printf("\nnode deleted\n");
     }
   else
   { ptr = head;
      while(ptr ->next != head)
      {  preptr=ptr;
         ptr = ptr->next;          }
      preptr->next = ptr -> next;
      free(ptr);
      printf("\nnode deleted\n");
   }
}
```

```
void search()
{
   struct node *ptr;
   int item,i=0,flag=1;
   ptr = head;
   if(ptr == NULL)
   {        printf("\nEmpty List\n");      }
   else
   {
      printf("\nEnter item which you want to search?\n");
      scanf("%d",&item);
      if(head ->data == item)
      {        printf("item found at location %d",i+1);
      flag=0;         }
      else
      {
        while (ptr->next != head)
      {    if(ptr->data == item)
          {
             printf("item found at location %d ",i+1);
             flag=0;                break;            }
```

```
        else
        {               flag=1;          }
        i++;
        ptr = ptr -> next;
      }         }
      if(flag != 0)
      {           printf("Item not found\n");        }
   }
}

void display()
{   struct node *ptr;
   ptr=head;
   if(head == NULL)
   {        printf("\nnothing to print");     }
   else
   {        printf("\n printing values ... \n");
              while(ptr -> next != head)
      {       printf("%d\n", ptr -> data);            ptr = ptr -> next;        }
      printf("%d\n", ptr -> data);
   }             }
```

Advantages of Circular linked List

- It allows to traverse the list starting at any point. It allows quick access to the first and last records.
- Circularly doubly linked list allows to traverse the list in either direction.

Applications of lists:

1. Polynomial ADT
2. Radix sort
3. Multilist

Polynomial Manipulation – All operations (Insertion, Deletion, Merge, Traversal).

Polynomial Implementation:

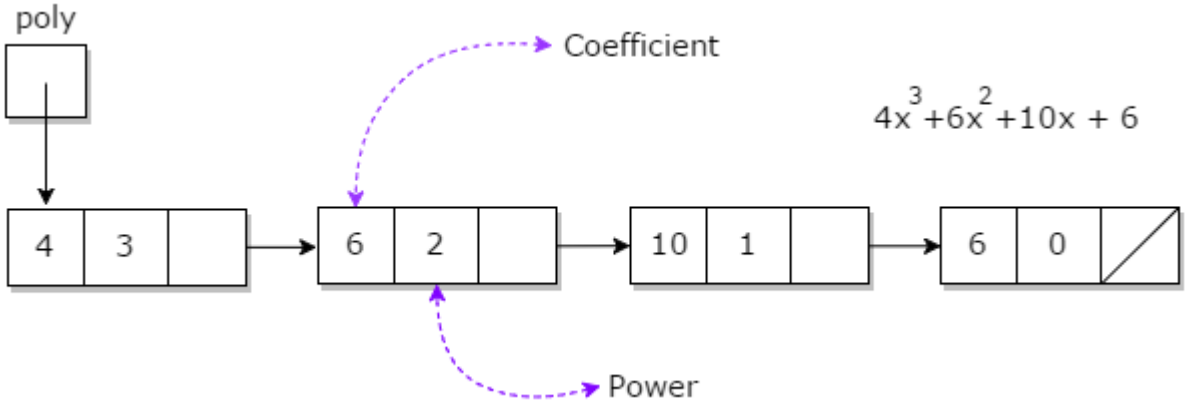
A polynomial p(x) is the expression in variable x which is in the form ($ax^n + bx^{n-1} + \ldots + jx + k$), where a, b, c …., k fall in the category of real numbers and 'n' is non negative integer, which is called the degree of polynomial.

- i.e A polynomial is an expression that contains more than two terms.
    - one is the coefficient
    - other is the exponent

$P(x) = 4x^3+6x^2+7x+9$, here 4,6,7, and 9 are coefficients and 3, 2, 1 is its exponential value

Polynomial can be represented in the various ways. These are:

- By the use of arrays
- By the use of Linked List



- Polynomial manipulations such as insertion, deletion, Merge, Traversal , addition, subtraction & differentiation etc.. can be performed using linked list / array.

Procedure for Polynomial Addition subtraction & differentiation using linked list:

Declaration of Linked list implementation of Polynomial:

```
struct poly{
{ int coeff;
int power;
struct poly *next;
} *list1, *list2, *list3;
```

Creation of Polynomial:

```
poly create(poly *head, poly *newnode)
{ poly*ptr;
```

```
if(head==NULL)
{ head=newnode;
return(head); }
else
{ ptr=head;
while(ptr-> next!=NULL)
ptr=ptr->next;
ptr->next=newnode; }
return(head);
}
```

Addition of Polynomials:

To add two polynomials we need to scan them once. If we find terms with the same exponent in the two polynomials, then we add the coefficients; otherwise, we copy the term of larger exponent into the sum and go on. When we reach at the end of one of the polynomial, then remaining part of the other is copied into the sum.

To add two polynomials follow the following steps:

- **Read two polynomials.**
- **Add them.**
- **Display the resultant polynomial.**

Addition of Polynomials:

```
void add()
{
poly *ptr1, *ptr2, *newnode;
ptr1=list1;
ptr2=list2;
while(ptr1!=NULL && ptr2!= NULL)
{ newnode=malloc(sizeof(struct poly));
if(ptr1->power==ptr2->power)
{ newnode->coeff = ptr1->coeff + ptr2->coeff;
newnode->power=ptr1->power;
newnode->next=NULL;
list3=create(list3,newnode);
ptr1=ptr->next;
ptr2=ptr2->next;
}
else
{ if(ptr1->power > ptr2->power)
{ newnode->coeff = ptr1->coeff
newnode->power=ptr1->power;
newnode->next=NULL;
```

```
list3=create(list3,newnode);
ptr1=ptr1->next;        }
else
{ newnode->coeff = ptr2->coeff
newnode->power=ptr2->power;
newnode->next=NULL;
list3=create(list3,newnode);
ptr2=ptr2->next; }
}
}
```

SUBTRACTION OF POLYNOMIALS:

(add this statement in the above program)

```
newnode->coeff = ptr1->coeff - ptr2->coeff
```

MULTIPLICATION OF POLYNOMIALS:

- Multiplication of two polynomials however requires manipulation of each node such that the exponents are added up and the coefficients are multiplied.
- After each term of firstpolynomial is operated upon with each term of the second polynomial,
- Then the result has to be added up by comparing the exponents and adding the coefficients for similar exponents and including terms as such with dissimilar exponents in the result

```
void Mul()
{
poly *ptr1, *ptr2, *newnode;
ptr1=list1;
ptr2=list2;
if(ptr1 == NULL && ptr2 == NULL)
return;
if(ptr1 == NULL) // I polynomial does not exist
list3 = list2;
elsif(ptr2 ==NULL) // II polynomial does not exist
list3 =list1;
else // Both polynomial exist
{ if(ptr1!=NULL && ptr2!= NULL)
{ while(ptr1!=NULL)
{ newnode=malloc(sizeof(struct poly));
while(ptr2!=NULL)
{ newnode->coeff = ptr1->coeff * ptr2 ->coeff;
newnode->power=ptr1->power + ptr2->power;
list3=create(list3,newnode);
ptr2=ptr2->next;
}
```

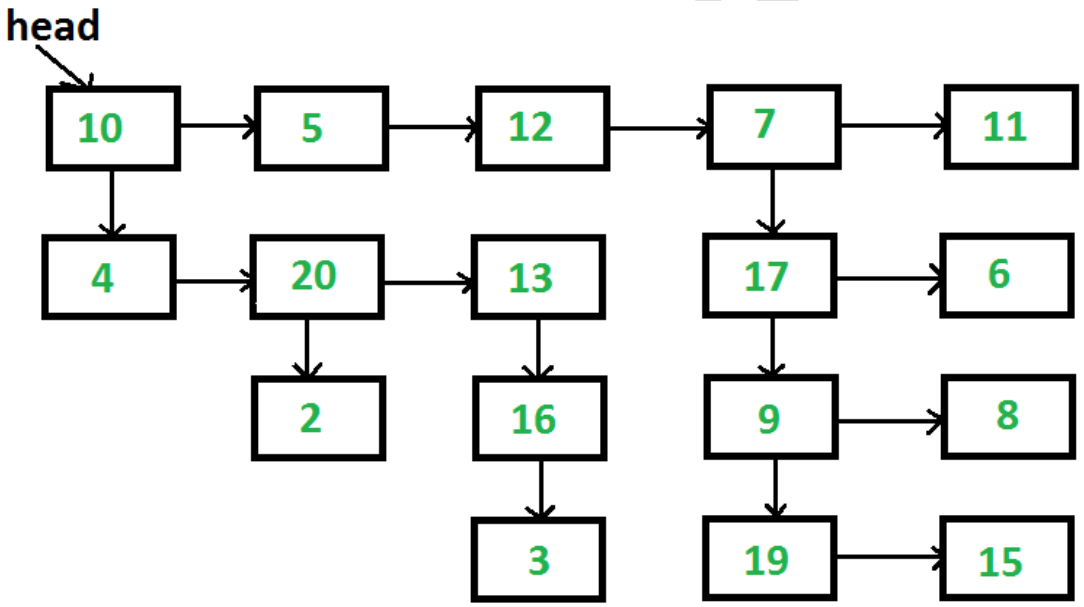
```
ptr2=list2;
ptr1=ptr1->next;
} }
} }
```

Multilists:

- In a general multi-linked list each node can have any number of pointers to other nodes, and there may or may not be inverses for each pointer.
- Multi-lists are essentially the technique of embedding multiple lists into a single data structure.
- A multi-list has more than one next pointer, like a doubly linked list, but the pointers create separate lists
- i. e Given a linked list where in addition to the next pointer, each node has a child pointer, which may or may not point to a separate list.
- These child lists may have one or more children of their own, and so on, to produce a multilevel data structure, as shown in below figure.
- You are given the head of the first level of the list.
- We can Flatten the list so that all the nodes appear in a single-level linked list.
- You need to flatten the list in way that all nodes at first level should come first, then nodes of second level, and so on.

Each node is a C struct with the following definition.

```
struct List
{
   int data;
   struct List *next;
   struct List *child;
};
```



The above list can be converted to 10->5->12->7->11->4->20->13->17->6->2->16->9->8->3->19->15