Module -3

Linear Data Structures and their Linked Storage Representation

Teaching Hours: 10

1. *Linked List*

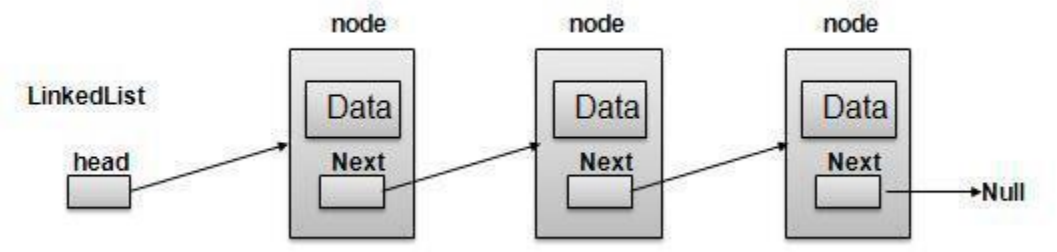
1.1. **Definition**

A linked-list is a sequence of data structures which are connected together via links.

Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list the second most used data structure after array. Following are important terms to understand the concepts of Linked List.

- **Link** − Each Link of a linked list can store a data called an element.
- **Next** − Each Link of a linked list contain a link to next link called Next.
- **LinkedList** − A LinkedList contains the connection link to the first Link called First.

1.2. **Representation**



As per above shown illustration, following are the important points to be considered.

- LinkedList contains an link element called first.
- Each Link carries a data field(s) and a Link Field called next.
- Each Link is linked with its next link using its next link.
- Last Link carries a Link as null to mark the end of the list.

1.3. **Operations**

Following are the basic operations supported by a list.

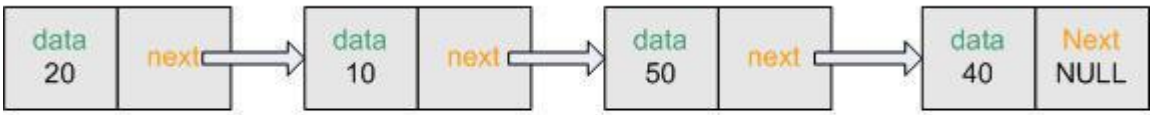
- **Insertion** − add an element at the beginning of the list.
- **Deletion** − delete an element at the beginning of the list.
- **Display** − displaying complete list.
- **Search** − search an element using given key.

- **Delete** − delete an element using given key.

2. Types:

2.1. **Singly Linked List**

Linked list is one of the fundamental data structures, and can be used to implement other data structures. In a linked list there are different numbers of nodes. Each node is consists of two fields. The first field holds the value or data and the second field holds the reference to the next node or null if the linked list is empty.



Linked list

Figure: Linked list

Pseudocode:

```
Linkedlist Node {
   data // The value or data stored in the node
   next // A reference to the next node, null for last node
 }
```

The singly-linked list is the easiest of the linked list, which has one link per node.

Pointer

To create linked list in C/C++ we must have a clear understanding about pointer. Now I will explain in brief what is pointer and how it works.

A pointer is a variable that contains the address of a variable. The question is why we need pointer? Or why it is so powerful? The answer is they have been part of the C/C++ language and so we have to use it. Using pointer we can pass argument to the functions. Generally we pass them by value as a copy. So we cannot change them. But if we pass argument using pointer, we can modify them. To understand about pointers, we must know how computer store variable and its value. Now, I will show it here in a very simple way.

Let us imagine that a computer memory is a long array and every array location has a distinct memory location.

```
int a = 50 // initialize variable a
```

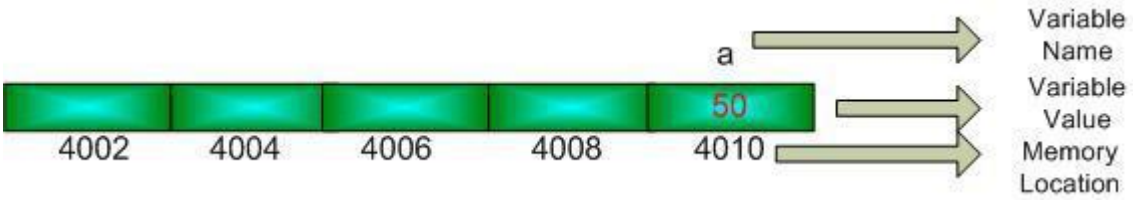


Figure: Variable value store inside an array

It is like a house which has an address and this house has only one room. So the full address is-

Name of the house: a

Name of the person/value who live here is: 50

House Number: 4010

If we want to change the person/value of this house, the conventional way is, type this code line

```
a = 100    // new initialization
```

But using pointer we can directly go to the memory location of 'a' and change the person/value of this house without disturbing _a‘. This is the main point about pointer.

Now the question is how we can use pointer. Type this code line:

```
int *b;    // declare pointer b
```

We transfer the memory location of a to b .

```
b = &a;    // the unary operator & gives the address of an object
```

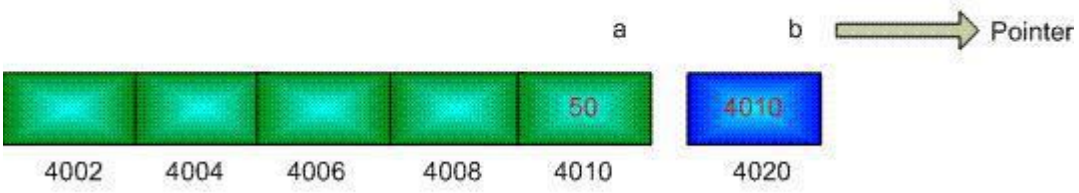


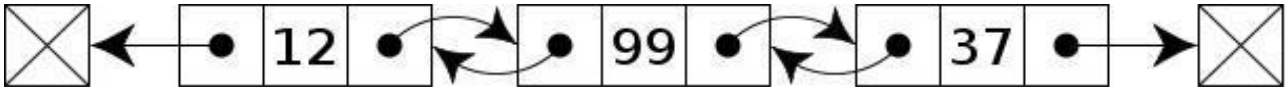
Figure: Integer pointer b store the address of the integer variable a

2.2. **Doubly Linked list**

A doubly linked list is a sequential data structure that consists of set of linked records called nodes.
In a doubly linked list, each node consists of:-

1. Two address fields, one pointing to next node, and other pointing to the previous pointer.
2. and one data part that is used to store the data of the node.

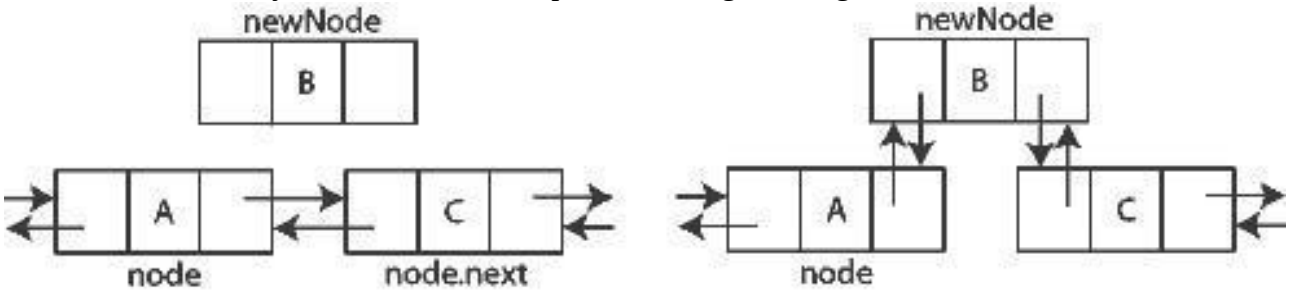
The following diagram shows the representation of doubly link-list:-



Here I have a complete C program for the implementation of doubly linked list.
I have performed the following operations on the doubly linked list:-

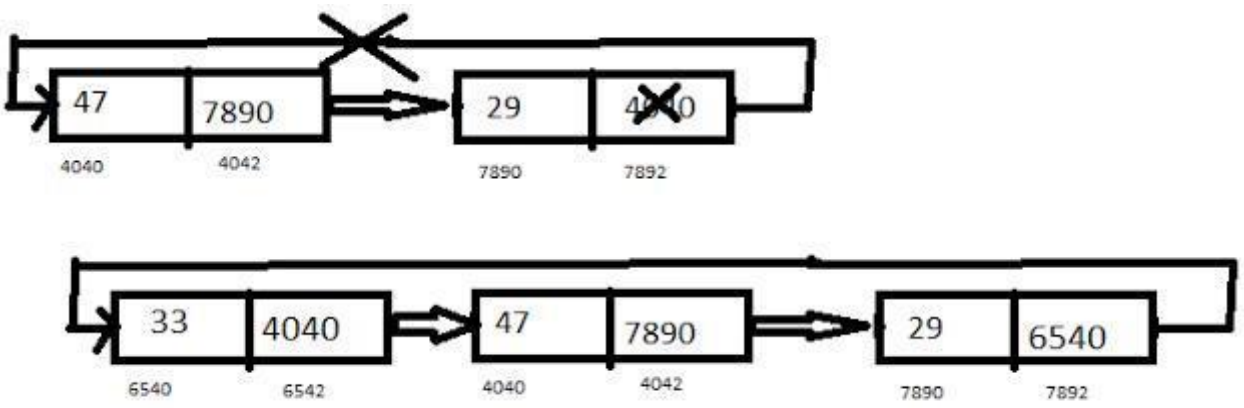
1. Insertion of node after a specific node
2. Insertion of node before a specific node
3. Forward printing
4. Reverse printing
5. Deletion of a specific node

Simple insertion in a doubly linked list can be explained through this figure:-

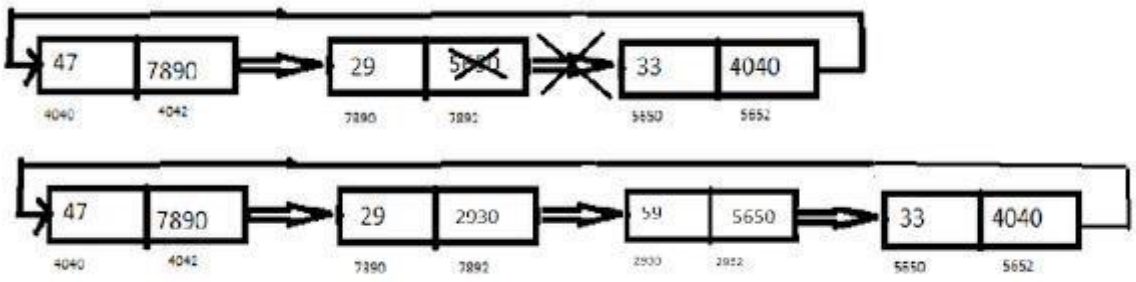


2.3. **Circular linked list**

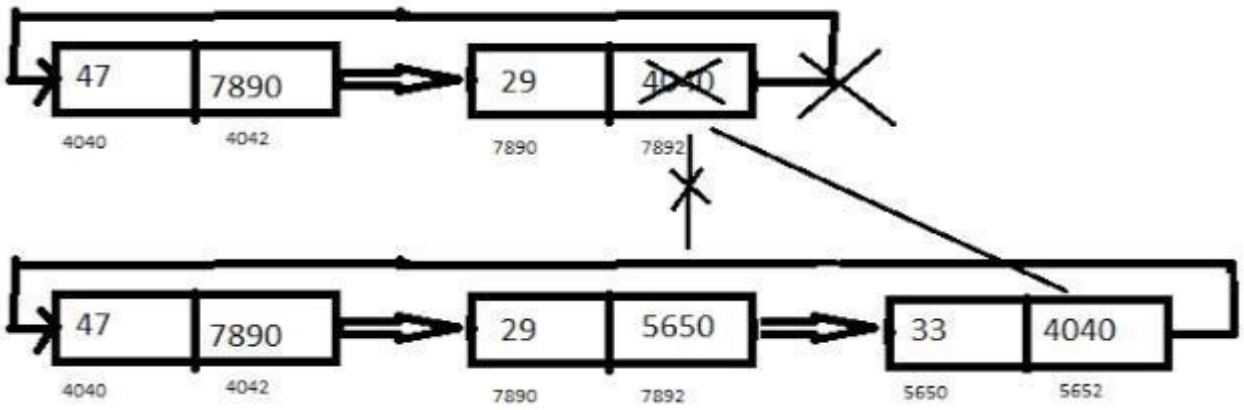
let us consider the code in insertbeg().we check that is linked list empty or not by checking value of Start!=NULL. (Note: Click on image for better view)



if start=null then the new created node is assign to Start else consider the code
temp=start;
while(temp->next!=start)
    temp=temp->next;
temp->next=nn;
 nn->next=start; start=nn;
assume the above image,we want to add 33 at the begg.so the temp pointer is traversed to the end of list,inserting the address of new node in temp->next and inserting address pointed by start in the new nodes next,make the start pointer to point new node.

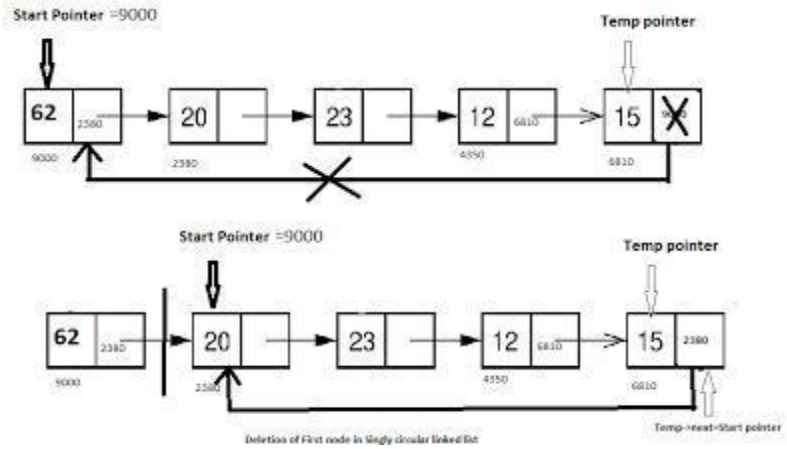


No Explanation is needed for inserting element at the mid,it remains the same as that of singly linked list

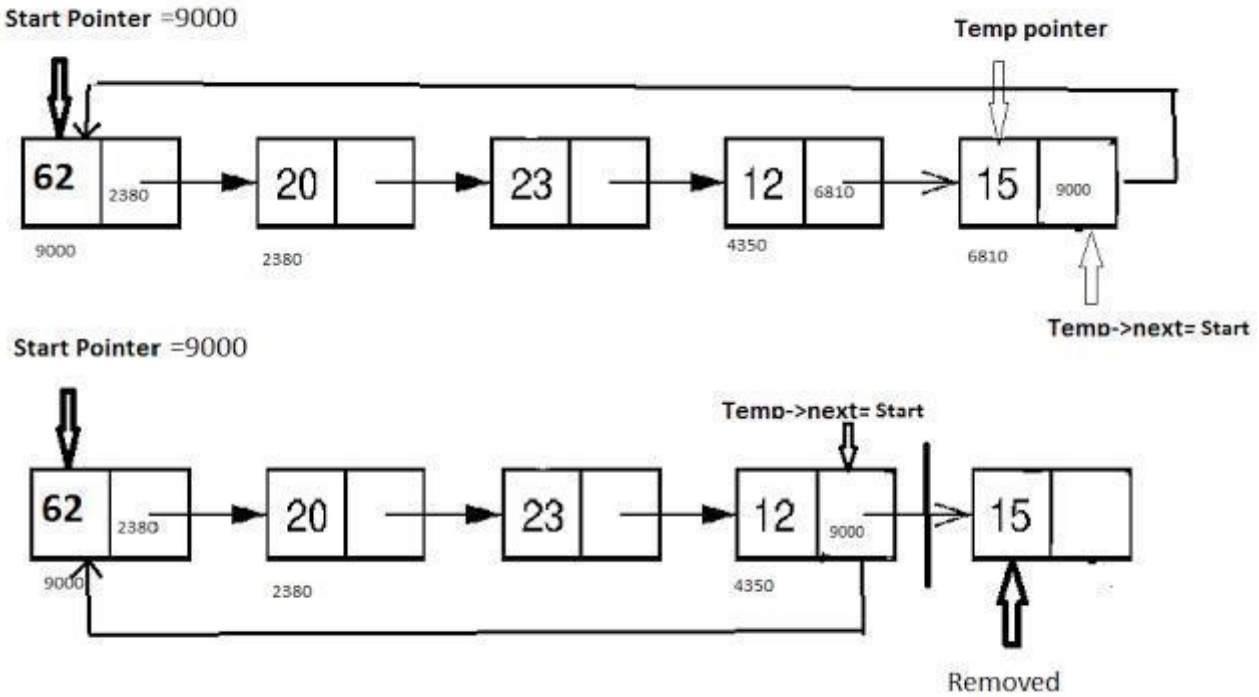


Inserting 33 at the end of list

Consider the case of **inserting element at the end of singly circular linked list.** Assume we wanted to add 33 at the end of list,we will traverse the temporary pointers till the temps next address is not start.So at this point we have last node pointed by temp.we will insert temp->next address in new nodes next and new nodes address in temp->next.



Deletion of First node.For deletion of first node again we have to traversed the temp pointer till the end of list so that address of start can be removed from temp->next and addrss of start's next is inserted in temp->next.Start pointer is made to point the address contained by start->next. Deletion of middle node is same as that of singly linked list



3. Linked *implementation*

3.1. **Stack**

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

```
struct Node
{
   int Data;
   struct Node *next;
}*top;

void popStack()
{
   struct Node *temp, *var=top;
   if(var==top)
   {
      top = top->next;
      free(var);
   }
   else
   printf("\nStack Empty");
}

void push(int value)
{
   struct Node *temp;
   temp=(struct Node *)malloc(sizeof(struct Node));
   temp->Data=value;
   if (top == NULL)
   {
       top=temp;
       top->next=NULL;
   }
   else
   {
      temp->next=top;
      top=temp;
   }
}

void display()
{
    struct Node *var=top;
    if(var!=NULL)
    {
        printf("\nElements are as:\n");
        while(var!=NULL)
        {
            printf("\t%d\n",var->Data);
            var=var->next;
        }
```

```
        printf("\n");
        }
        else
        printf("\nStack is Empty");
    }

    int main(int argc, char *argv[])
    {
        int i=0;
        top=NULL;
        printf(" \n1. Push to stack");
        printf(" \n2. Pop from Stack");
        printf(" \n3. Display data of Stack");
        printf(" \n4. Exit\n");
        while(1)
        {
            printf(" \nChoose Option: ");
            scanf("%d",&i);
            switch(i)
            {
                case 1:
                {
                int value;
                printf("\nEnter a valueber to push into Stack: ");
                scanf("%d",&value);
                push(value);
                display();
                break;
                }
                case 2:
                {
                popStack();
                display();
                break;
                }
                case 3:
                {
                display();
                break;
                }
                case 4:
                {
                struct Node *temp;
                while(top!=NULL)
                {
                    temp = top->next;
                    free(top);
```

```
                    top=temp;
                }
                exit(0);
                }
                default:
                {
                printf("\nwrong choice for operation");
                }
            }
        }
    }
```

3.2. **Queues**

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

struct node
{
   int info;
   struct node *ptr;
}*front,*rear,*temp,*front1;

int frontelement();
void enq(int data);
void deq();
void empty();
void display();
void create();
void queuesize();

int count = 0;

void main()
{
   int no, ch, e;

   printf("\n 1 - Enque");
   printf("\n 2 - Deque");
   printf("\n 3 - Front element");
   printf("\n 4 - Empty");
   printf("\n 5 - Exit");
   printf("\n 6 - Display");
   printf("\n 7 - Queue size");
   create();
   while (1)
   {
      printf("\n Enter choice : ");
      scanf("%d", &ch);
```

```
        switch (ch)
        {
        case 1:
           printf("Enter data : ");
           scanf("%d", &no);
           enq(no);
           break;
        case 2:
           deq();
           break;
        case 3:
           e = frontelement();
           if (e != 0)
              printf("Front element : %d", e);
           else
              printf("\n No front element in Queue as queue is
           empty"); break;
        case 4:
           empty();
           break;
        case 5:
           exit(0);
        case 6:
           display();
           break;
        case 7:
           queuesize();
           break;
        default:
           printf("Wrong choice, Please enter correct choice  ");
           break;
        }
    }
}

/* Create an empty queue */
void create()
{
   front = rear = NULL;
}

/* Returns queue size */
void queuesize()
{
   printf("\n Queue size : %d", count);
}
```

```
/* Enqueing the queue */
void enq(int data)
{
   if (rear == NULL)
   {
      rear = (struct node *)malloc(1*sizeof(struct node));
      rear->ptr = NULL;
      rear->info = data;
      front = rear;
   }
   else
   {
      temp=(struct node *)malloc(1*sizeof(struct node));
      rear->ptr = temp;
      temp->info = data;
      temp->ptr = NULL;

      rear = temp;
   }
   count++;
}

/* Displaying the queue elements */
void display()
{
   front1 = front;

   if ((front1 == NULL) && (rear == NULL))
   {
      printf("Queue is empty");
      return;
   }
   while (front1 != rear)
   {
      printf("%d ", front1->info);
      front1 = front1->ptr;
   }
   if (front1 == rear)
      printf("%d", front1->info);
}

/* Dequeing the queue */
void deq()
{
   front1 = front;

   if (front1 == NULL)
```

```
    {
        printf("\n Error: Trying to display elements from empty
        queue"); return;
    }
    else
        if (front1->ptr != NULL)
        {
            front1 = front1->ptr;
            printf("\n Dequed value : %d", front->info);
            free(front);
            front = front1;
        }
        else
        {
            printf("\n Dequed value : %d", front->info);
            free(front);
            front = NULL;
            rear = NULL;
        }
        count--;
}

/* Returns the front element of queue */
int frontelement()
{
    if ((front != NULL) && (rear != NULL))
        return(front->info);
    else
        return 0;
}

/* Display if queue is empty or not */
void empty()
{
     if ((front == NULL) && (rear == NULL))
        printf("\n Queue empty");
    else
       printf("Queue not empty");
}
```

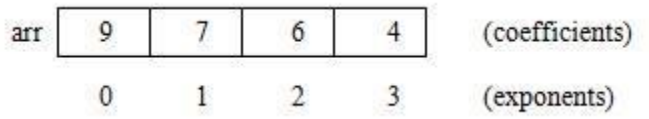
4. Applications of Linked lists

4.1. **Polynomial Manipulation**

A polynomial is an expression that contains more than two terms. A term is made up of coefficient and exponent. An example of polynomial is

$P(x) = 4x^3+6x^2+7x+9$

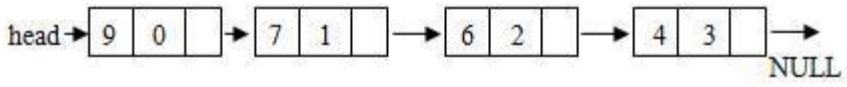
A polynomial thus may be represented using arrays or linked lists. Array representation assumes that the exponents of the given expression are arranged from 0 to the highest value (degree), which is represented by the subscript of the array beginning with 0. The coefficients of the respective exponent are placed at an appropriate index in the array. The array representation for the above polynomial expression is given below:



A polynomial may also be represented using a linked list. A structure may be defined such that it contains two parts- one is the coefficient and second is the corresponding exponent. The structure definition may be given as shown below:

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

Thus the above polynomial may be represented using linked list as shown below:



Addition of two Polynomials:

For adding two polynomials using arrays is straightforward method, since both the arrays may be added up element wise beginning from 0 to n-1, resulting in addition of two polynomials. Addition of two polynomials using linked list requires comparing the exponents, and wherever the exponents are found to be same, the coefficients are added up. For terms with different exponents, the complete term is simply added to the result thereby making it a part of addition result. The complete program to add two polynomials is given in subsequent section.

Multiplication of two 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 first polynomial 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. The _C' program for polynomial manipulation is given below:

Program for Polynomial representation, addition and multiplication

/*Polynomial- Representation, Addition, Multiplication*/

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

struct poly
{
int coeff;
int exp;
struct poly *next;
}*head1=NULL,*head2=NULL,*head3=NULL,*head4=NULL,*temp,*ptr;

void create();
void makenode(int,int);
struct poly *insertend(struct poly *);
void display(struct poly *);
struct poly *addtwopoly(struct poly *,struct poly *,struct poly *);
struct poly *subtwopoly(struct poly *,struct poly *,struct poly *);
struct poly *multwopoly(struct poly *,struct poly *,struct poly
*); struct poly *dispose(struct poly *); int search(struct poly
*,int);

void main()
{
int ch,coefficient,exponent;
int listno;
while(1)
{
clrscr();
printf(―ntMenu‖);
printf(―nt1. Create First Polynomial.‖);
printf(―nt2. Display First Polynomial.‖);
printf(―nt3. Create Second Polynomial.‖);
printf(―nt4. Display Second Polynomial.‖);
printf(―nt5. Add Two Polynomials.‖);
printf(―nt6. Display Result of Addition.‖);
printf(―nt7. Subtract Two Polynomials.‖);
printf(―nt8. Display Result of Subtraction.‖);
printf(―nt9. Multiply Two Polynomials.‖);
printf(―nt10. Display Result of Product.‖);
printf(―nt11. Dispose List.‖);
printf(―nt12. Exit‖);
printf(―nntEnter your choice?‖);
scanf(―%d‖,&ch);
switch(ch)
{
case 1:
printf(―nGenerating first polynomial:‖);
printf(―nEnter coefficient?‖);
```

```
scanf(—%d‖,&coefficient);
printf(—nEnter exponent?‖);
scanf(—%d‖,&exponent);
makenode(coefficient,exponent);
head1 = insertend(head1);
break;
case 2:
display(head1);
break;
case 3:
printf(—nGenerating second polynomial:‖);
printf(—nEnter coefficient?‖);
scanf(—%d‖,&coefficient);
printf(—nEnter exponent?‖);
scanf(—%d‖,&exponent);
makenode(coefficient,exponent);
head2 = insertend(head2);
break;
case 4:
display(head2);
break;
case 5:
printf(—nDisposing result list.‖);
head3=dispose(head3);
head3=addtwopoly(head1,head2,head3);
printf(—Addition successfully done!‖);
break;
case 6:
display(head3);
break;
case 7:
head3=dispose(head3);
head3=subtwopoly(head1,head2,head3);
printf(—Subtraction successfully done!‖);
getch();
break;
case 8:
display(head3);
break;
case 9:
head3=dispose(head3);
head4=dispose(head4);
head4=multwopoly(head1,head2,head3);
break;
case 10:
display(head4);
break;
```

```
case 11:
printf(—Enter list number to dispose(1 to 4)?‖);
scanf(—%d‖,&listno);
if(listno==1)
head1=dispose(head1);
else if(listno==2)
head2=dispose(head2);
else if(listno==3)
head3=dispose(head3);
else if(listno==4)
head4=dispose(head4);
else
printf(—Invalid number specified.‖);
break;
case 12:
exit(0);
default:
printf(—Invalid Choice!‖);
break;
}
}
}

void create()
{
ptr=(struct poly *)malloc(sizeof(struct poly));
if(ptr==NULL)
{
printf(—Memory Allocation Error!‖);
exit(1);
}
}

void makenode(int c,int e)
{
create();
ptr->coeff = c;
ptr->exp = e;
ptr->next = NULL;
}

struct poly *insertend(struct poly *head)
{
if(head==NULL)
head = ptr;
else
{
temp=head;
```

```
while(temp->next != NULL)
temp = temp->next;
temp->next = ptr;
}
return head;
}

void display(struct poly *head)
{
if(head==NULL)
printf(—List is empty!‖);
else
{
temp=head;
while(temp!=NULL)
{
printf(—(%d,%d)->‖,temp->coeff,temp->exp);
temp=temp->next;
}
printf(—bb —);
}
getch();
}

struct poly *addtwopoly(struct poly *h1,struct poly *h2,struct poly *h3)
{
/*
(5,3)->(6,1) + (7,3)->(9,2) = (12,3)->(6,1)->(9,2)
*/
struct poly *temp1,*temp2,*temp3;
temp1=h1;
temp2=h2;
while(temp1!=NULL || temp2!=NULL)
{
if(temp1->exp==temp2->exp)
{
makenode(temp1->coeff+temp2->coeff,temp1->exp);
h3=insertend(h3);
}
else
{
makenode(temp1->coeff,temp1->exp);
h3=insertend(h3);
makenode(temp2->coeff,temp2->exp);
h3=insertend(h3);
}
temp1=temp1->next;
```

```
temp2=temp2->next;
}
if(temp1==NULL && temp2!=NULL)
{
while(temp2!=NULL)
{
makenode(temp2->coeff,temp2->exp);
h3=insertend(h3);
temp2=temp2->next;
}
}
if(temp2==NULL && temp1!=NULL)
{
while(temp1!=NULL)
{
makenode(temp2->coeff,temp2->exp);
h3=insertend(h3);
temp1=temp1->next;
}
}
return h3;
}

struct poly *subtwopoly(struct poly *h1,struct poly *h2,struct poly *h3)
{
/*
(5,3)->(6,1) – (7,3)->(9,2) = (-2,3)+(6,1)-(9,2)
*/
struct poly *temp1,*temp2,*temp3;
temp1=h1;
temp2=h2;
while(temp1!=NULL || temp2!=NULL)
{
if(temp1->exp==temp2->exp)
{
makenode(temp1->coeff-temp2->coeff,temp1->exp);
h3=insertend(h3);
}
else
{
makenode(temp1->coeff,temp1->exp);
h3=insertend(h3);
makenode(-temp2->coeff,temp2->exp);
h3=insertend(h3);
}
temp1=temp1->next;
temp2=temp2->next;
```

```
}
if(temp1==NULL && temp2!=NULL)
{
while(temp2!=NULL)
{
makenode(temp2->coeff,temp2->exp);
h3=insertend(h3);
temp2=temp2->next;
}
}
if(temp2==NULL && temp1!=NULL)
{
while(temp1!=NULL)
{
makenode(-temp2->coeff,temp2->exp);
h3=insertend(h3);
temp1=temp1->next;
}
}
return h3;
}

struct poly *multwopoly(struct poly *h1,struct poly *h2,struct poly *h3)
{
/*
h1=(5,3)->(6,1) * h2=(7,3)->(9,2)
(5,3)->(7,3),(9,2) = (35,6),(45,5)
(6,1)->(7,3),(9,2) = (42,4),(54,3)
h3->(35,6)->(45,5)->(42,4)->(54,3)
(35,6)+(45,5)+(42,4)+(54,3)=Result
*/
int res=0;
struct poly *temp1,*temp2,*temp3;
printf("\nDisplaying First Polynomial:\n");
display(h1);
printf("\nDisplaying Second Polynomial:\n");
display(h2);

temp1=h1;
while(temp1!=NULL)
{
temp2=h2;
while(temp2!=NULL)
{
makenode(temp1->coeff*temp2->coeff,temp1->exp+temp2->exp);
h3=insertend(h3);
temp2=temp2->next;
```

```
}
temp1=temp1->next;
}

printf(—nDisplaying Initial Result of Product:nttǁ);
display(h3);
getch();

temp1=h3;
while(temp1!=NULL)
{temp2=temp1->next;
res=0;
while(temp2!=NULL)
{
if(temp1->exp==temp2->exp)
res += temp2->coeff;
temp2=temp2->next;
}
if(search(head4,temp1->exp)==1)
{
makenode(res+temp1->coeff,temp1->exp);
head4=insertend(head4);
}
temp1=temp1->next;
}
return head4;
}

int search(struct poly *h,int val)
{
struct poly *tmp;
tmp=h;
while(tmp!=NULL)
{if(tmp->exp==val)
return 0;
tmp=tmp->next;
}
return 1;
}

    struct poly *dispose(struct poly *list)
    {
    if(list==NULL)
    {
    printf(—List is already empty.ǁ);
    return list;
    }
    else
```

```
    {
    temp=list;
    while(list!=NULL)
    {
    free(temp);
    list=list->next;
    temp=list;
    }
    return list;
    }
    }
```

4.2. **Symbol table organizations**

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

struct hash *hashTable = NULL;
int eleCount = 0;

struct node {
   int key, age;
   char name[100];
   struct node *next;
};

struct hash {
   struct node *head;
   int count;
};

struct node * createNode(int key, char *name, int age) {
   struct node *newnode;
   newnode = (struct node *) malloc(sizeof(struct node));
   newnode->key = key;
   newnode->age = age;
   strcpy(newnode->name, name);
   newnode->next = NULL;
   return newnode;
}

void insertToHash(int key, char *name, int age) {
   int hashIndex = key % eleCount;
   struct node *newnode = createNode(key, name, age);
   /* head of list for the bucket with index "hashIndex" */
   if (!hashTable[hashIndex].head) {
```

```
        hashTable[hashIndex].head = newnode;
        hashTable[hashIndex].count = 1;
        return;
    }
    /* adding new node to the list */
    newnode->next = (hashTable[hashIndex].head);
    /*
     * update the head of the list and no of
     * nodes in the current bucket
     */
    hashTable[hashIndex].head = newnode;
    hashTable[hashIndex].count++;
    return;
}

void deleteFromHash(int key) {
    /* find the bucket using hash index */
    int hashIndex = key % eleCount, flag = 0;
    struct node *temp, *myNode;
    /* get the list head from current bucket */
    myNode = hashTable[hashIndex].head;
    if (!myNode) {
        printf("Given data is not present in hash Table!!\n");
        return;
    }
    temp = myNode;
    while (myNode != NULL) {
        /* delete the node with given key */
        if (myNode->key == key) {
            flag = 1;
            if (myNode == hashTable[hashIndex].head)
                hashTable[hashIndex].head = myNode->next;
            else
                temp->next = myNode->next;

            hashTable[hashIndex].count--;
            free(myNode);
            break;
        }
        temp = myNode;
        myNode = myNode->next;
    }
    if (flag)
        printf("Data deleted successfully from Hash
    Table\n"); else
        printf("Given data is not present in hash
    Table!!!!\n"); return;
```

```
}

void searchInHash(int key) {
   int hashIndex = key % eleCount, flag = 0;
   struct node *myNode;
   myNode = hashTable[hashIndex].head;
   if (!myNode) {
      printf("Search element unavailable in hash table\n");
      return;
   }
   while (myNode != NULL) {
      if (myNode->key == key) {
         printf("VoterID  : %d\n", myNode->key);
         printf("Name     : %s\n", myNode->name);
         printf("Age      : %d\n", myNode->age);
         flag = 1;
         break;
      }
      myNode = myNode->next;
   }
   if (!flag)
      printf("Search element unavailable in hash table\n");
   return;
}

void display() {
   struct node *myNode;
   int i;
   for (i = 0; i < eleCount; i++) {
      if (hashTable[i].count == 0)
         continue;
      myNode = hashTable[i].head;
      if (!myNode)
         continue;
      printf("\nData at index %d in Hash Table:\n", i);
      printf("VoterID     Name          Age   \n");
      printf("--------------------------------\n");
      while (myNode != NULL) {
         printf("%-12d", myNode->key);
         printf("%-15s", myNode->name);
         printf("%d\n", myNode->age);
         myNode = myNode->next;
      }
   }
   return;
}
```

```
int main() {
    int n, ch, key, age;
    char name[100];
    printf("Enter the number of elements:");
    scanf("%d", &n);
    eleCount = n;
    /* create hash table with "n" no of buckets */
    hashTable = (struct hash *) calloc(n, sizeof(struct
    hash)); while (1) {
        printf("\n1. Insertion\t2. Deletion\n");
        printf("3. Searching\t4. Display\n5. Exit\n");
        printf("Enter your choice:");
        scanf("%d", &ch);
        switch (ch) {
        case 1:
            printf("Enter the key value:");
            scanf("%d", &key);
            getchar();
            printf("Name:");
            fgets(name, 100, stdin);
            name[strlen(name) - 1] = '\0';
            printf("Age:");
            scanf("%d", &age);
            /*inserting new node to hash table */
            insertToHash(key, name, age);
            break;

        case 2:
            printf("Enter the key to perform deletion:");
            scanf("%d", &key);
            /* delete node with "key" from hash table */
            deleteFromHash(key);
            break;

        case 3:
            printf("Enter the key to search:");
            scanf("%d", &key);
            searchInHash(key);
            break;
        case 4:
            display();
            break;
        case 5:
            exit(0);
        default:
            printf("U have entered wrong option!!\n");
            break;
```

```
      }
   }
   return 0;
}
```

4.3. **Sparse matrix representation with multilinked data structure**

Linked List Representation Of Sparse Matrix
If most of the elements in a matrix have the value 0, then the matrix is called spare matrix.

Example For 3 X 3 Sparse Matrix:

```
| 1  0  0 |
| 0  0  0 |
| 0  4  0 |
```

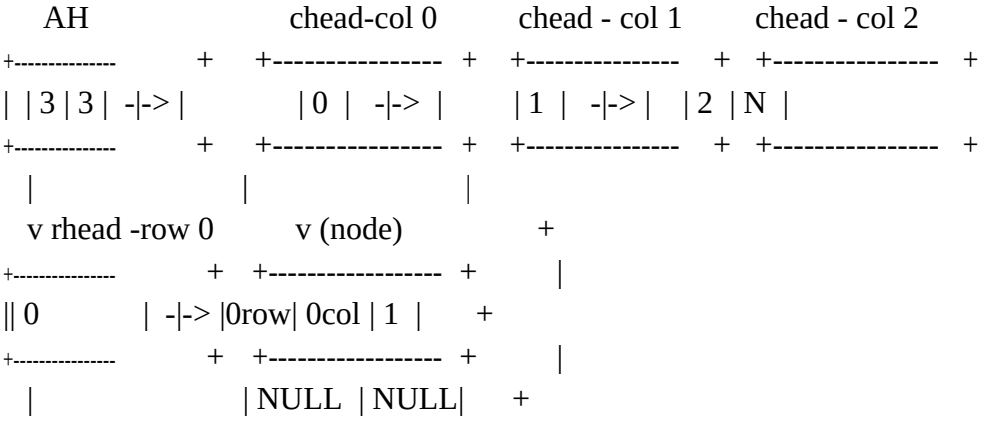
3-Tuple Representation Of Sparse Matrix Using Arrays:

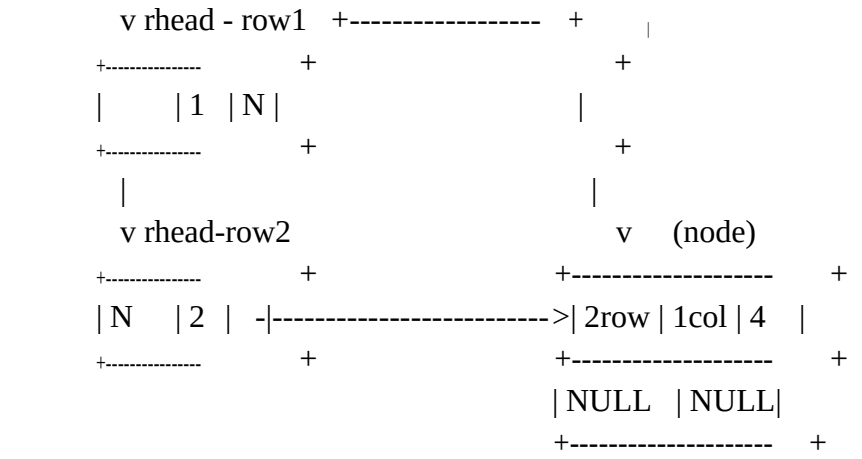
```
| 3  3  2 |
| 0  0  1 |
| 2  1  4 |
```

Elements in the first row represents the number of rows, columns and non-zero values in sparse matrix.
First Row - | 3 3 2 |
3 - rows
3 - columns
2 - non- zero values

Elements in the other rows gives information about the location and value of non-zero elements.
| 0 0 1| (Second Row) - represents value 1 at 0th Row, 0th column
| 2 1 4| (Third Row) - represents value 4 at 2nd Row, 1st column

3-Tuple Representation Of Sparse Matrix Using Linked List:
AH - Additional Header (sparseHead)





5. Programming Examples

5.1. **length of a list**

int getCount(head)
1) If head is NULL, return 0.
2) Else return 1 + getCount(head->next)

5.2. **Merging two lists**

```
struct node* SortedMerge(struct node* a, struct node* b)
{
  struct node* result = NULL;

  /* Base cases */
  if (a == NULL)
     return(b);
  else if (b==NULL)
     return(a);

  /* Pick either a or b, and recur */
  if (a->data <= b->data)
  {
     result = a;
     result->next = SortedMerge(a->next, b);
  }
  else
  {
     result = b;
     result->next = SortedMerge(a, b->next);
  }
  return(result);
}
```

5.3. **Removing duplicates**

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

```
/* A linked list node */
struct node
{
 int data;
 struct node *next;
};

/* Function to remove duplicates from a unsorted linked list */
void removeDuplicates(struct node *start) {

  struct node *ptr1, *ptr2, *dup;
  ptr1 = start;

  /* Pick elements one by one */
  while(ptr1 != NULL && ptr1->next != NULL)
  {
     ptr2 = ptr1;

     /* Compare the picked element with rest of the elements */
     while(ptr2->next != NULL)
     {
       /* If duplicate then delete it */
       if(ptr1->data == ptr2->next->data)
       {
          /* sequence of steps is important here */
          dup = ptr2->next;
          ptr2->next = ptr2->next->next;
          free(dup);
       }
       else /* This is tricky */
       {
          ptr2 = ptr2->next;
       }
     }
     ptr1 = ptr1->next;
  }
}

/* UTILITY FUNCTIONS */
/* Function to push a node */
void push(struct node** head_ref, int new_data);

/* Function to print nodes in a given linked list */
void printList(struct node *node);

/* Druver program to test above function */
int main()
```

```
{
  struct node *start = NULL;

  /* The constructed linked list is:
   10->12->11->11->12->11->10*/
  push(&start, 10);
  push(&start, 11);
  push(&start, 12);
  push(&start, 11);
  push(&start, 11);
  push(&start, 12);
  push(&start, 10);

  printf("\n Linked list before removing duplicates ");
  printList(start);

  removeDuplicates(start);

  printf("\n Linked list after removing duplicates ");
  printList(start);

  getchar();
}

/* Function to push a node */
void push(struct node** head_ref, int new_data)
{
  /* allocate node */
  struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

  /* put in the data */
  new_node->data  = new_data;

  /* link the old list off the new node */
  new_node->next = (*head_ref);

  /* move the head to point to the new node */
  (*head_ref)    = new_node;
}

/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
  while(node != NULL)
  {
    printf("%d ", node->data);
```

```
    node = node->next;
  }
}
```

5.4. **Reversing a list**

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

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Function to reverse the linked list */
static void reverse(struct node** head_ref)
{
    struct node* prev   = NULL;
    struct node* current = *head_ref;
    struct node* next;
    while (current != NULL)
    {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    *head_ref = prev;
}

/* Function to push a node */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

    /* put in the data */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}
```

```
/* Function to print linked list */
void printList(struct node *head)
{
   struct node *temp = head;
   while(temp != NULL)
   {
      printf("%d ", temp->data);
      temp = temp->next;
   }
}

/* Driver program to test above function*/
int main()
{
   /* Start with the empty list */
   struct node* head = NULL;

    push(&head, 20);
    push(&head, 4);
    push(&head, 15);
    push(&head, 85);

    printList(head);
    reverse(&head);
    printf("\n Reversed Linked list \n");
    printList(head);
    getchar();
}
```

5.5. **Union and intersection of two lists**

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

/* Link list node */
struct node
{
   int data;
   struct node* next;
};

/* A utility function to insert a node at the beginning of
   a linked list*/
void push(struct node** head_ref, int new_data);

/* A utility function to check if given data is present in a list
*/ bool isPresent(struct node *head, int data);
```

```
/* Function to get union of two linked lists head1 and head2 */
struct node *getUnion(struct node *head1, struct node *head2)
{
   struct node *result = NULL;
   struct node *t1 = head1, *t2 = head2;

   // Insert all elements of list1 to the result list
   while (t1 != NULL)
   {
      push(&result, t1-
      >data); t1 = t1->next;
   }

   // Insert those elements of list2 which are not
   // present in result list
   while (t2 != NULL)
   {
      if (!isPresent(result, t2->data))
         push(&result, t2->data);
      t2 = t2->next;
   }

   return result;
}

/* Function to get intersection of two linked lists
  head1 and head2 */
struct node *getIntersection(struct node *head1,
                             struct node *head2)
{
   struct node *result = NULL;
   struct node *t1 = head1;

   // Traverse list1 and search each element of it in
   // list2. If the element is present in list 2, then
   // insert the element to result
   while (t1 != NULL)
   {
      if (isPresent(head2, t1->data))
         push (&result, t1->data);
      t1 = t1->next;
   }

   return result;
}

/* A utility function to insert a node at the begining of a linked list*/
```

```
void push (struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

/* A utility function to print a linked list*/
void printList (struct node *node)
{
    while (node != NULL)
    {
        printf ("%d ", node->data);
        node = node->next;
    }
}

/* A utility function that returns true if data is
   present in linked list else return false */
bool isPresent (struct node *head, int data)
{
    struct node *t = head;
    while (t != NULL)
    {
        if (t->data == data)
            return 1;
        t = t->next;
    }
    return 0;
}

/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct node* head1 = NULL;
    struct node* head2 = NULL;
    struct node* intersecn = NULL;
```

```
    struct node* unin = NULL;

    /*create a linked lits 10->15->5->20 */
    push (&head1, 20);
    push (&head1, 4);
    push (&head1, 15);
    push (&head1, 10);

    /*create a linked lits 8->4->2->10 */
    push (&head2, 10);
    push (&head2, 2);
    push (&head2, 4);
    push (&head2, 8);

    intersecn = getIntersection (head1, head2);
    unin = getUnion (head1, head2);

    printf ("\n First list is \n");
    printList (head1);

    printf ("\n Second list is \n");
    printList (head2);

    printf ("\n Intersection list is \n");
    printList (intersecn);

    printf ("\n Union list is \n");
    printList (unin);

    return 0;
}
```