so on. The terminating condition for outer loop is taken as `(end!=start->link)`, so the outer loop will terminate when end points to second node, i.e. the outer loop will work only till end reaches the third node.

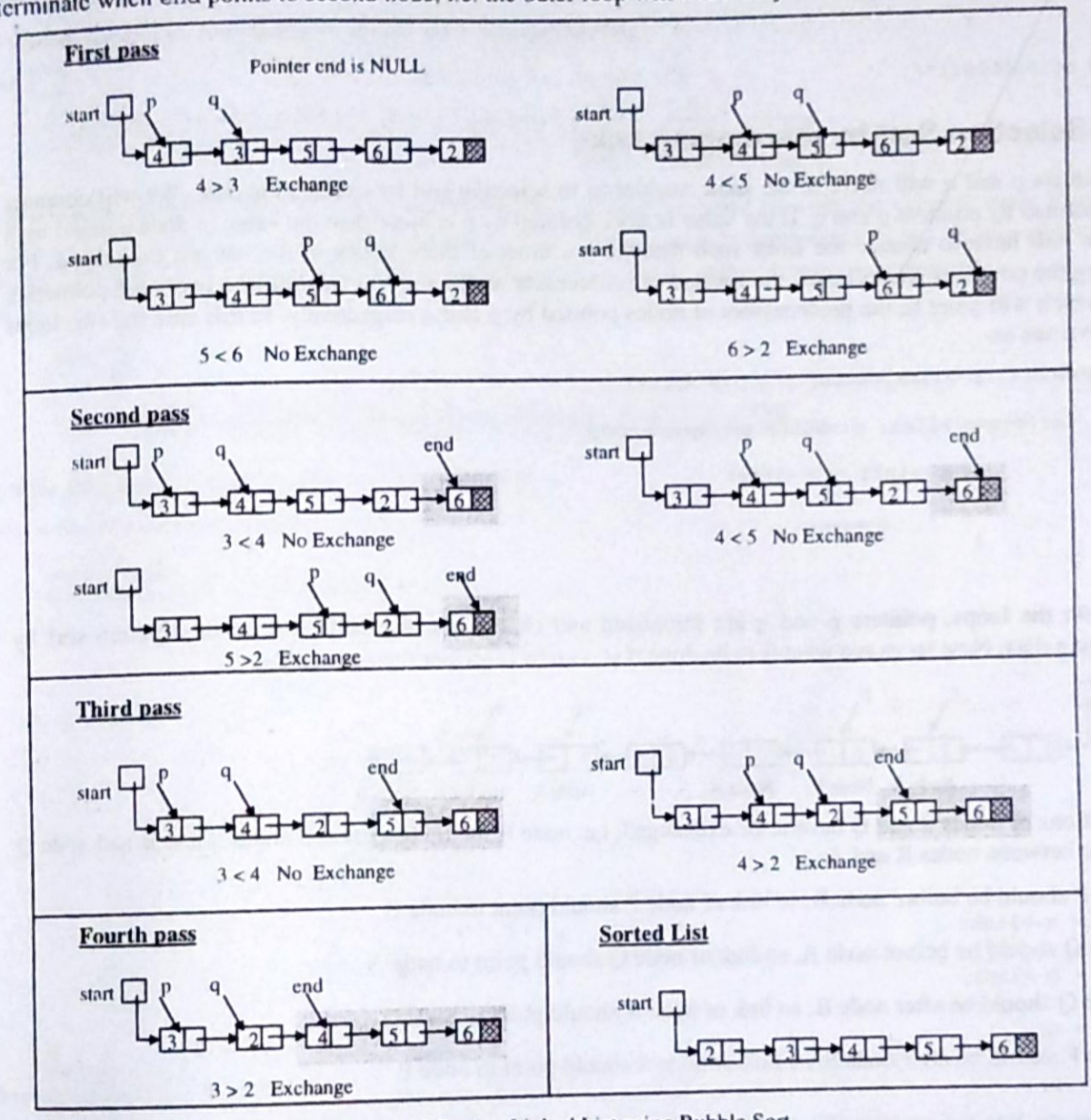


Figure 3.41 Sorting Linked List using Bubble Sort

The following function will sort a single linked list through bubble sort technique by exchanging data.

```
void bubble(struct node *start)
{
        struct node *end,*p,*q;
        int tmp;
        for(end=NULL; end!=start->link; end=q)
        {
                for(p=start; p->link!=end; p=p->link)
                {
                        q = p->link;
                        if(p->info > q->info)
                        {
                                tmp = p->info;
```

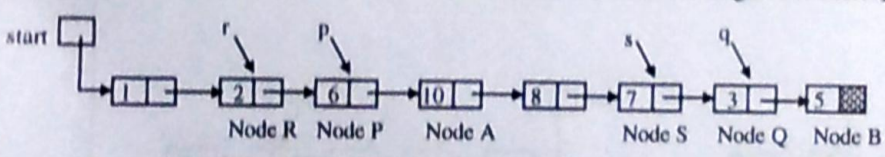
```
                        p->info = q->info;
                        q->info = tmp;
                    }
                }
            }
}/*End of bubble()*/
```

3.6.3 Selection Sort by rearranging links

The pointers p and q will move in the same manner as in selection sort by exchanging data. We will compare nodes pointed by pointers p and q. If the value in node pointed by p is more than the value in node pointed by q then we will have to change the links such that the positions of these nodes in the list are exchanged. For changing the positions we will need the address of predecessor nodes also. So we will take two more pointers r and s which will point to the predecessors of nodes pointed by p and q respectively. In this case the two loops can be written as-

```
for(r=p=start; p->link!=NULL; r=p,p=p->link)
{
        for(s=q=p->link; q!=NULL; s=q,q=q->link)
        {
                if(p->info > q->info)
                {
                        ..................
                }
        }
```

In both the loops, pointers p and q are initialized and changed in the same way as in selection sort by exchanging data. Now let us see what is to be done if p->info is greater then q->info.



The positions of nodes P and Q have to be exchanged, i.e. node P should be between nodes S and B and node Q should be between nodes R and A

(i) Node P should be before node B, so link of node P should point to node B

```
p->link = q->link;
```

(ii) Node Q should be before node A, so link of node Q should point to node A

```
q->link = p->link;
```

(iii) Node Q should be after node R, so link of node R should point to node Q

```
r->link = q;
```

(iv) Node P should be after node S, so link of node S should point to node P

```
s->link = p;
```

For writing the first two statements we will need a temporary pointer, since we are exchanging p->link and q->link.

```
tmp = p->link;
p->link = q->link;
q->link = tmp;
```

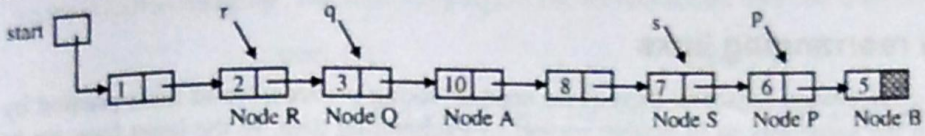
If p points to the first node, then r also points to the first node i.e. nodes R and P both are same, so in this case there is no need of writing the third statement (r->link = q;)
We need the third statement only if the pointer p is not equal to start. So it can be written as-

```
if(p!=start)
        r->link = q;
```

If start points to node P, then start needs to be updated and now it should point to node Q.

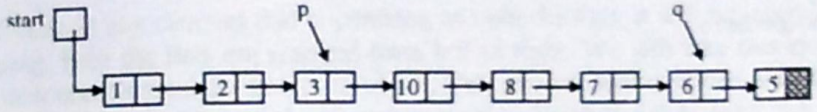
```
if(p==start)
        start = q;
```

After writing the above statements the linked list will look like this-



The positions of nodes P and Q have changed and this is what we want because the value in node P was more than value in node Q. Now we will bring the pointers p and q back to their positions to continue with our sorting process. For this we will exchange the pointers p and q with the help of a temporary pointer.

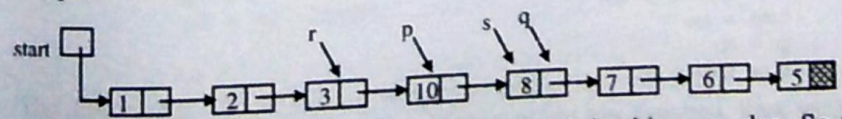
```
tmp = p;    p = q;   q = tmp;
```



Here is the function for selection sort by rearranging links.

```
struct node *selection_1(struct node *start)
{
        struct node *p,*q,*r,*s,*tmp;
        for(r=p=start; p->link!=NULL; r=p,p=p->link)
        {
                for(s=q=p->link; q!=NULL; s=q,q=q->link)
                {
                        if(p->info > q->info)
                        {
                                tmp = p->link;
                                p->link = q->link;
                                q->link = tmp;
                                if(p!=start)
                                r->link = q;
                                s->link = p;
                                if(p == start)
                                        start = q;
                                tmp = p;
                                p = q;
                                q = tmp;
                        }
                }
        }
        return start;
}/*End of selection_1()*/
```

In the previous figures, we have taken the case when p and q point to non adjacent nodes, and we have written our code according to this case only. Now let us see whether this code will work when p and q point to adjacent nodes. The pointers p and q will point to adjacent nodes only in the first iteration of inner loop. In that case s and q point to same node. The following figure shows this situation-



You can see that the code we have written will work in this case also. So there is no need to consider a separate case when nodes p and q are adjacent.

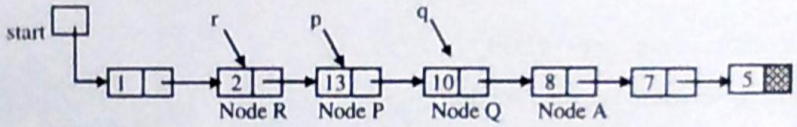
When we sort a linked list by exchanging data, links are not disturbed so `start` remains same. But wh sorting is done by rearranging links, the value of `start` might change so it is necessary to return the value o `start` at the end of the function.

3.6.4 Bubble sort by rearranging links

In bubble sort, since p and q are always adjacent, there is no need of taking predecessor of node pointed by Both the loops are written in the same way as in bubble sorting by exchanging data. In the inner loop we ha taken a pointer r that will point to the predecessor of node pointed by p.

```
for(end=NULL; end!=start->link; end=q)
{
        for(r=p=start; p->link!=end; r=p,p=p->link)
        {
                q = p->link;
                if(p->info > q->info )
                {
                        ..................
                }
        }
}
```

Now let us see what is to be done if `p->info` is greater than `q->info`.



Node P should be before node A, so link of node P should point to node A

```
p->link = q->link;
```

Node Q should be before node P, so link of node Q should point to node P

```
q->link = p;
```

Node Q should be after node R, so link of node R should point to node Q

```
r->link = q;
```

Here is the function for bubble sort by rearranging links.

```
struct node *bubble_l(struct node *start)
{
        struct node *end,*r,*p,*q,*tmp;
        for(end=NULL; end!=start->link; end=q)
        {
                for(r=p=start; p->link!=end; r=p,p=p->link)
                {
                        q = p->link;
                        if(p->info > q->info )
                        {
                                p->link = q->link;
                                q->link = p;
                                if(p!=start)
                                  r->link = q;
                                else
                                    start = q;
                                tmp = p;
                                p = q;
                                q = tmp;
                        }
                }
        }
        return start;
}/*End of bubble_l()*/
```

3.7 Merging

If there are two sorted linked lists, then the process of combining these sorted lists into another list of sorted order is called merging. The following figure shows two sorted lists and a third list obtained by merging them.

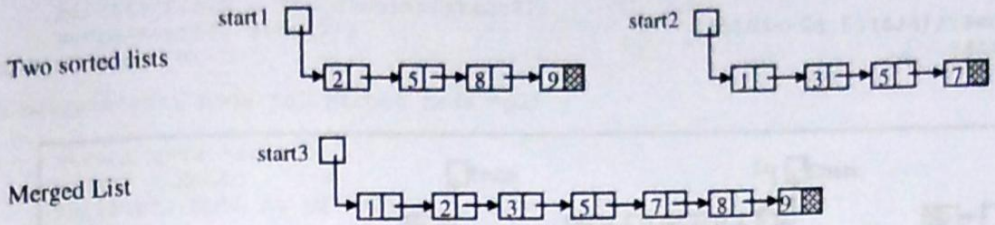


Figure 3.42

If there is any element that is common to both the lists, it will be inserted only once in the third list. For merging, both the lists are scanned from left to right. We will take one element from each list, compare them and then take the smaller one in third list. This process will continue until the elements of one list are finished. Then we will take the remaining elements of unfinished list in third list. The whole process for merging is shown in the figure 3.43. We've taken two pointers `p1` and `p2` that will point to the nodes that are being compared. There can be three cases while comparing `p1->info` and `p2->info`

1. If `(p1->info) < (p2->info)`
The new node that is added to the resultant list has info equal to `p1->info`. After this we will make `p1` point to the next node of first list.

2. If `(p2->info) < (p1->info)`
The new node that is added to the resultant list has info equal to `p2->info`. After this we will make `p2` point to the next node of second list.

3. If `(p1->info) == (p2->info)`
The new node that is added to the resultant list has info equal to `p1->info`(or `p2->info`). After this we will make `p1` and `p2` point to the next nodes of first list and second list respectively. The procedure of merging is shown in figure 3.43.

```
while(p1!=NULL && p2!=NULL)
{
      if(p1->info < p2->info)
      {
            start3 = insert(start3,p1->info);
            p1 = p1->link;
      }
      else if(p2->info < p1->info)
      {
            start3 = insert(start3,p2->info);
            p2 = p2->link;
      }
      else if(p1->info == p2->info)
      {
            start3 = insert(start3,p1->info);
            p1 = p1->link;
            p2 = p2->link;
      }
}
```

The above loop will terminate when any of the list will finish. Now we have to add the remaining nodes of the unfinished list to the resultant list. If second list has finished, then we will insert all the nodes of first list in the resultant list as-

```
while(p1!=NULL)
{
      start3 = insert(start3,p1->info);
```

```
        p1 = p1->link;
}
```

If first list has finished, then we will insert all the nodes of second list in the resultant list as-

```
while(p2!=NULL)
{
        start3 = insert(start3,p2->info);
        p2 = p2->link;
}
```

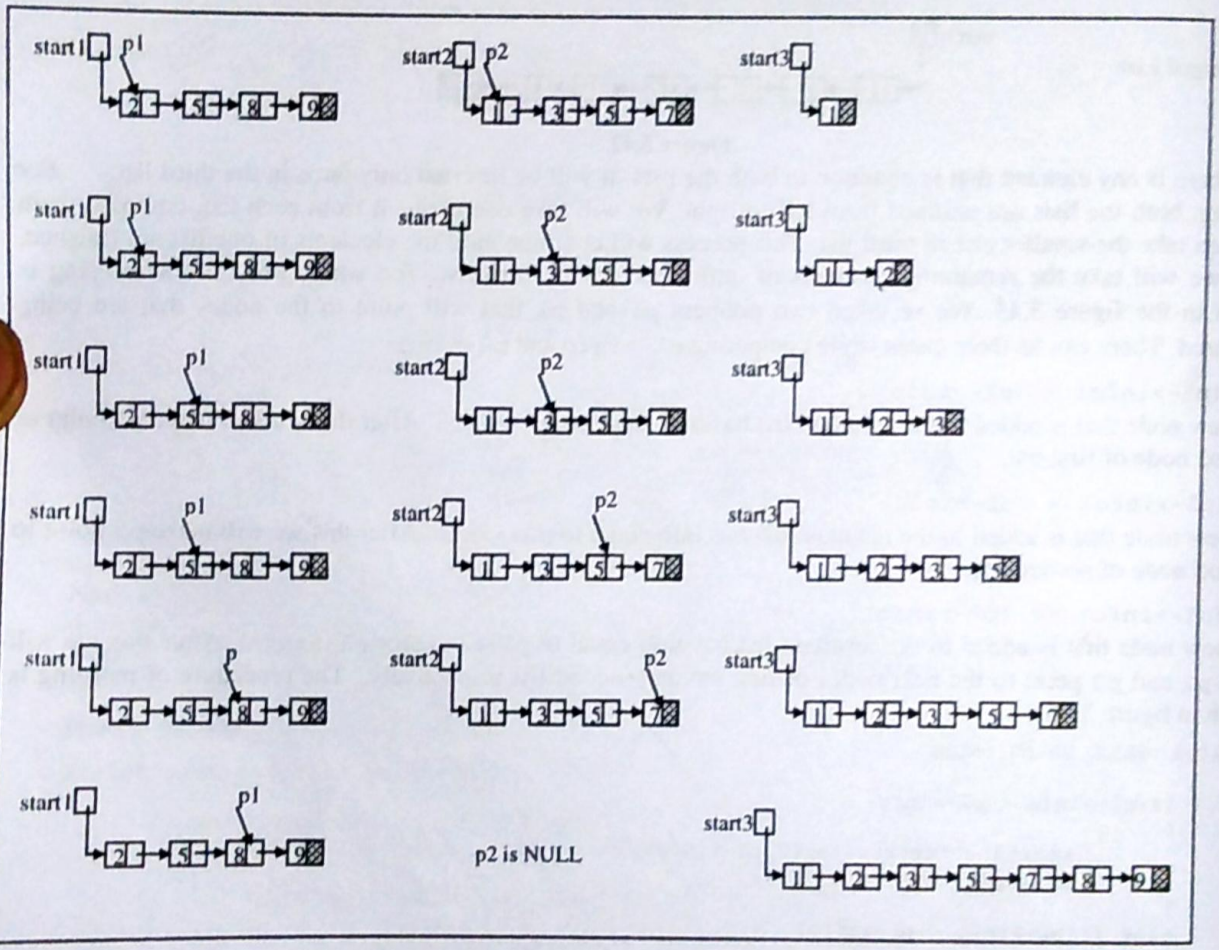


Figure 3.43 Merging two sorted linked lists

The program for merging is given next.

```
/*P3.7 Program of merging two sorted single linked lists*/
#include<stdio.h>
#include<stdlib.h>
struct node
{
        int info;
        struct node *link;
};
struct node *create(struct node *start);
struct node *insert_s(struct node *start,int data);
struct node *insert(struct node *start,int data);
void display(struct node *start);
void merge(struct node *p1,struct node *p2);
```

```
main()
{
        struct node *start1 = NULL,*start2 = NULL;
        start1 = create(start1);
        start2 = create(start2);
        printf("List1 : "); display(start1);
        printf("List2 : "); display(start2);
        merge(start1, start2);
}/*End of main()*/

void merge(struct node *p1,struct node *p2)
{
        struct node *start3;
        start3 = NULL;
        while(p1!=NULL && p2!=NULL)
        {
                if(p1->info < p2->info)
                {
                        start3 = insert(start3,p1->info);
                        p1 = p1->link;
                }
                else if(p2->info < p1->info)
                {
                        start3 = insert(start3,p2->info);
                        p2 = p2->link;
                }
                else if(p1->info==p2->info)
                {
                        start3 = insert(start3,p1->info);
                        p1 = p1->link;
                        p2 = p2->link;
                }
        }
        /*If second list has finished and elements left in first list*/
        while(p1!=NULL)
        {
                start3 = insert(start3,p1->info);
                p1 = p1->link;
        }
        /*If first list has finished and elements left in second list*/
        while(p2!=NULL)
        {
                start3 = insert(start3,p2->info);
                p2 = p2->link;
        }
        printf("Merged list is : ");
        display(start3);
}

struct node *create(struct node *start)
{
        int i,n,data;

        printf("Enter the number of nodes : ");
        scanf("%d",&n);
        start = NULL;
        for(i=1;i<=n;i++)
        {
                printf("Enter the element to be inserted : ");
                scanf("%d",&data);
                start = insert_s(start, data);
        }
        return start;
}/*End of create_slist()*/

struct node *insert_s(struct node *start,int data)
```

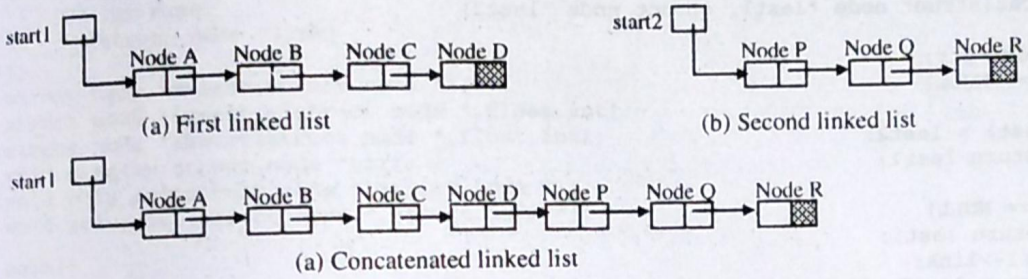
```
{
        struct node *p,*tmp;
        tmp = (struct node *)malloc(sizeof(struct node));
        tmp->info = data;
        /*list empty or data to be added in beginning */
        if(start == NULL || data<start->info)
        {
                tmp->link = start;
                start = tmp;
                return start;
        }
        else
        {
                p = start;
                while(p->link!=NULL && p->link->info < data)
                        p = p->link;
                tmp->link = p->link;
                p->link = tmp;
        }
        return start;
}/*End of insert_s()*/

struct node *insert(struct node *start,int data)
{
        struct node *p,*tmp;
        tmp = (struct node *)malloc(sizeof(struct node));
        tmp->info = data;
        if(start == NULL)  /*If list is empty*/
        {
                tmp->link = start;
                start = tmp;
                return start;
        }
        else    /*Insert at the end of the list*/
        {
                p = start;
                while(p->link!=NULL)
                        p = p->link;
                tmp->link = p->link;
                p->link = tmp;
        }
        return start;
}/*End of insert()*/

void display(struct node *start)
{
        struct node *p;
        if(start == NULL)
        {
                printf("List is empty\n");
                return;
        }
        p = start;
        while(p!=NULL)
        {
                printf("%d ",p->info);
                p = p->link;
        }
        printf("\n");
}/*End of display()*/
```

The function insert_s() is the same that we have made in sorted linked list and it inserts node ascending order. We have used this function to create the two sorted lists that are to be merged. The fu

`insert()` is a simple function that inserts nodes in a linked list at the end. We have used this function to insert nodes in the third list.

3.8 Concatenation

Suppose we have two single linked lists and we want to append one at the end of another. For this the link of last node of first list should point to the first node of the second list. Let us take two single linked lists and concatenate them.



(a) First linked list

(b) Second linked list

(a) Concatenated linked list

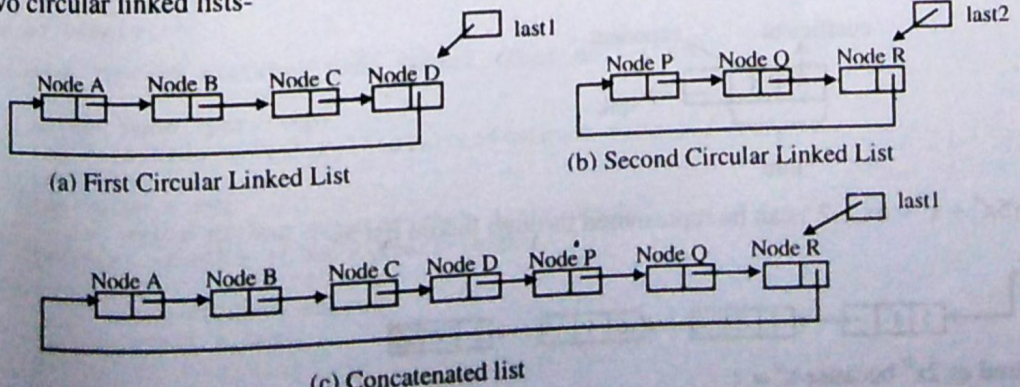
Figure 3.44

For concatenation, link of node D should point to node P. To get the address of node D, we have to traverse the first list till the end. Suppose `ptr` points to node D, then `ptr->link` should be made equal to `start2`.

```
ptr->link = start2;
```

```
struct node *concat(struct node *start1, struct node *start2)
{
        struct node *ptr;
        if(start1 == NULL)
        {
                start1 = start2;
                return start1;
        }
        if(start2 == NULL)
                return start1;
        ptr = start1;
        while(ptr->link != NULL)
                ptr = ptr->link;
        ptr->link = start2;
        return start1;
}
```

If we have to concatenate two circular linked lists, then there is no need to traverse any of the lists. Let us take two circular linked lists-



(a) First Circular Linked List

(b) Second Circular Linked List

(c) Concatenated list

Figure 3.45

Link of node D should point to node P

```
last1->link = last2->link;
```

We will lose the address of node A, so before writing this statement we should save `last1->link`.

```
ptr = last1->link;
```

Link of node R should point to node A

```
last2->link = ptr;
```

The pointer `last1` should point to node R

```
last1 = last2;
```

```
struct node *concat(struct node *last1, struct node *last2)
{
        struct node *ptr;
        if(last1 == NULL)
        {
                last1 = last2;
                return last1;
        }
        if(last2 == NULL)
                return last1;
        ptr = last1->link;
        last1->link = last2->link;
        last2->link = ptr;
        last1 = last2;
        return last1;
}
```

3.9 Polynomial arithmetic with linked list

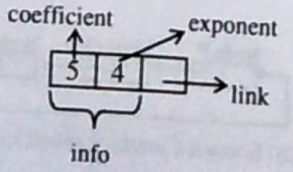
A useful application of linked list is the representation of polynomial expressions. Let us take a poly expression with single variable-

$$5x^4 + x^3 - 6x + 2$$

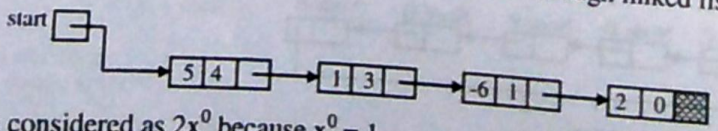
In each term we have a coefficient and an exponent. For example in the term $5x^4$, coefficient is exponent is 4. The whole polynomial can be represented through linked list where each node will rep term of the expression. The structure for each node will be-

```
struct node{
        float coefficient;
        int exponent;
        struct node *link;
    }
```

Here the info part of the node contains coefficient and exponent and the link part is same as before be used to point to the next node of the list. The node representing the term $5x^4$ can be represented as-



The polynomial $(5x^4 + x^3 - 6x + 2)$ can be represented through linked list as-



Here 2 is considered as $2x^0$ because $x^0 = 1$.

The arithmetic operations are easier if the terms are arranged in descending order of their exponents example it would be better if the polynomial expression $(5x + 6x^3 + x^2 - 9 + 2x^6)$ is stored as $(2x^6 + 6x^3 + x^2 + 5x - 9)$. So for representing the polynomial expression, we will use sorted linked list which would

descending order based on the exponent. An empty list will represent zero polynomial. The following program shows creation of polynomial linked lists and their addition and multiplication.

```
/*P3.10 Program of polynomial addition and multiplication using linked list*/
#include<stdio.h>
#include<stdlib.h>
struct node
{
        float coef;
        int expo;
        struct node *link;
};
struct node *create(struct node *);
struct node *insert_s(struct node *,float,int);
struct node *insert(struct node *,float,int);
void display(struct node *ptr);
void poly_add(struct node *,struct node *);
void poly_mult(struct node *,struct node *);

main()
{
        struct node *start1 = NULL,*start2 = NULL;
        printf("Enter polynomial 1 :\n");    start1 = create(start1);
        printf("Enter polynomial 2 :\n");    start2 = create(start2);
        printf("Polynomial 1 is :  ");  display(start1);
        printf("Polynomial 2 is :  ");  display(start2);
        poly_add(start1, start2);
        poly_mult(start1, start2);
}/*End of main()*/

struct node *create(struct node *start)
{
        int i,n,ex;
        float co;
        printf("Enter the number of terms : ");
        scanf("%d",&n);
        for(i=1;i<=n;i++)
        {
                printf("Enter coeficient for term %d : ",i);
                scanf("%f",&co);
                printf("Enter exponent for term %d : ", );
                scanf("%d",&ex);
                start = insert_s(start,co,ex);
        }
        return start;
}/*End of create()*/

struct node *insert_s(struct node *start,float co,int ex)
{
        struct node *ptr,*tmp;
        tmp = (struct node *)malloc(sizeof(struct node));
        tmp->coef = co;
        tmp->expo = ex;
        /*list empty or exp greater than first one*/
        if(start == NULL || ex > start->expo)
        {
                tmp->link = start;
                start = tmp;
        }
        else
        {
                ptr = start;
                while(ptr->link!=NULL && ptr->link->expo >= ex)
                        ptr = ptr->link;
```

```
                tmp->link = ptr->link;
                ptr->link = tmp;
        }
        return start;
}/*End of insert()*/

struct node *insert(struct node *start,float co,int ex)
{
        struct node *ptr,*tmp;
        tmp = (struct node *)malloc(sizeof(struct node));
        tmp->coef = co;
        tmp->expo = ex;
        if(start == NULL)       /*If list is empty*/
        {
                tmp->link = start;
                start = tmp;
        }
        else    /*Insert at the end of the list*/
        {
                ptr = start;
                while(ptr->link!=NULL)
                        ptr = ptr->link;
                tmp->link = ptr->link;
                ptr->link = tmp;
        }
        return start;
}/*End of insert()*/
void display(struct node *ptr)
{
        if(ptr == NULL)
        {
                printf("Zero polynomial\n");
                return;
        }
        while(ptr!=NULL)
        {
                printf("(%.1fx^%d)", ptr->coef,ptr->expo);
                ptr = ptr->link;
                if(ptr!=NULL)
                        printf(" + ");
                else
                        printf("\n");
        }
}/*End of display()*/

void poly_add(struct node *p1,struct node *p2)
{
        struct node *start3;
        start3 = NULL;
        while(p1!=NULL && p2!=NULL)
        {
                if(p1->expo > p2->expo)
                {
                        start3 = insert(start3,p1->coef,p1->expo);
                        p1 = p1->link;
                }
                else if(p2->expo > p1->expo)
                {
                        start3 = insert(start3,p2->coef,p2->expo);
                        p2 = p2->link;
                }
                else if(p1->expo == p2->expo)
                {
                        start3 = insert(start3,p1->coef+p2->coef,p1->expo);
```

```
                p1 = p1->link;
                p2 = p2->link;
            }
        }
        /*if poly2 has finished and elements left in poly1*/
        while(p1!=NULL)
        {
                start3 = insert(start3,p1->coef,p1->expo);
                p1 = p1->link;
        }
        /*if poly1 has finished and elements left in poly2*/
        while(p2!=NULL)
        {
                start3 = insert(start3,p2->coef,p2->expo);
                p2 = p2->link;
        }
        printf("Added polynomial is : ");
        display(start3);
}/*End of poly_add() */

void poly_mult(struct node *p1, struct node *p2)
{
        struct node *start3;
        struct node *p2_beg = p2;

        start3 = NULL;
        if(p1 == NULL || p2 == NULL)
        {
                printf("Multiplied polynomial is zero polynomial\n");
                return;
        }
        while(p1!=NULL)
        {
           p2 = p2_beg;
           while(p2!=NULL)
           {
             start3 = insert_s(start3,p1->coef*p2->coef,p1->expo+p2->expo);
             p2 = p2->link;
           }
           p1 = p1->link;
        }
        printf("Multiplied polynomial is : ");
        display(start3);
}/*End of poly_mult()*/
```

3.9.1 Creation of polynomial linked list

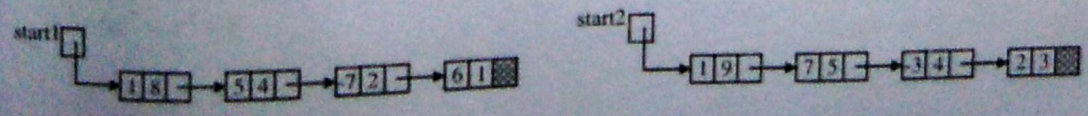
The function `create()` is very simple and calls another function `insert_s()` that inserts a node in polynomial linked list. The function `insert_s()` is similar to that of sorted linked lists. The only difference is that here our list is in descending order based on the exponent.

3.9.2 Addition of 2 polynomials

The procedure for addition of 2 polynomials represented by linked lists is somewhat similar to that of merging. Let us take two polynomial expression lists and make a third list by adding them.

Poly1- $x^8 + 5x^4 - 7x^2 + 6x$

Poly2- $x^9 + 7x^5 - 3x^4 + 2x^3$



The pointers p1 and p2 will point to the current nodes in the polynomials which will be added. The p of addition is shown in the figure 3.46. Both polynomials are traversed until one polynomial finishes. W have three cases-

1. If (p1->expo) > (p2->expo)
The new node that is added to the resultant list has coefficient equal to p1->coef and exponent equal to p1->expo. After this we will make p1 point to the next node of polynomial1.

2. If (p2->expo) > (p1->expo)
The new node that is added to the resultant list has coefficient equal to p2->coef and exponent equal to p2->expo. After this we will make p2 point to the next node of polynomial2.

3. If (p1->expo) == (p2->expo)
The new node that is added to the resultant list has coefficient equal to (p1->coef + p2->coef) and exp equal to p1->expo(or p2->expo). After this we will make p1 and p2 point to the next nodes of polyno and polynomial 2 respectively. The procedure of polynomial addition is shown in figure 3.46.

```
while(p1!=NULL && p2!=NULL)
{
        if(p1->expo > p2->expo)
        {
                p3_start = insert(p3_start,p1->coef,p1->expo);
                p1 = p1->link;
        }
        else if(p2->expo > p1->expo)
        {
                p3_start = insert(p3_start,p2->coef,p2->expo);
                p2 = p2->link;
        }
        else if(p1->expo == p2->expo)
        {
                p3_start = insert (p3_start,p1->coef+p2->coef,p1->expo);
                p1 = p1->link;
                p2 = p2->link;
        }
}
```

The above loop will terminate when any of the polynomial will finish. Now we have to add the remai nodes of the unfinished polynomial to the resultant list. If polynomial 2 has finished, then we will put all terms of polynomial 1 in the resultant list as-

```
while(p1!=NULL)
{
        p3_start = insert(p3_start,p1->coef,p1->expo);
        p1 = p1->link;
}
```

If polynomial 1 has finished, then we will put all the terms of polynomial 2 in the resultant list as-

```
while(p2!=NULL)
{
        p3_start = insert(p3_start,p2->coef,p2->expo);
        p2 = p2->link;
}
```

We can see the advantage of storing the terms in descending order of their exponents. If it was not so we would have to scan both the lists many times.

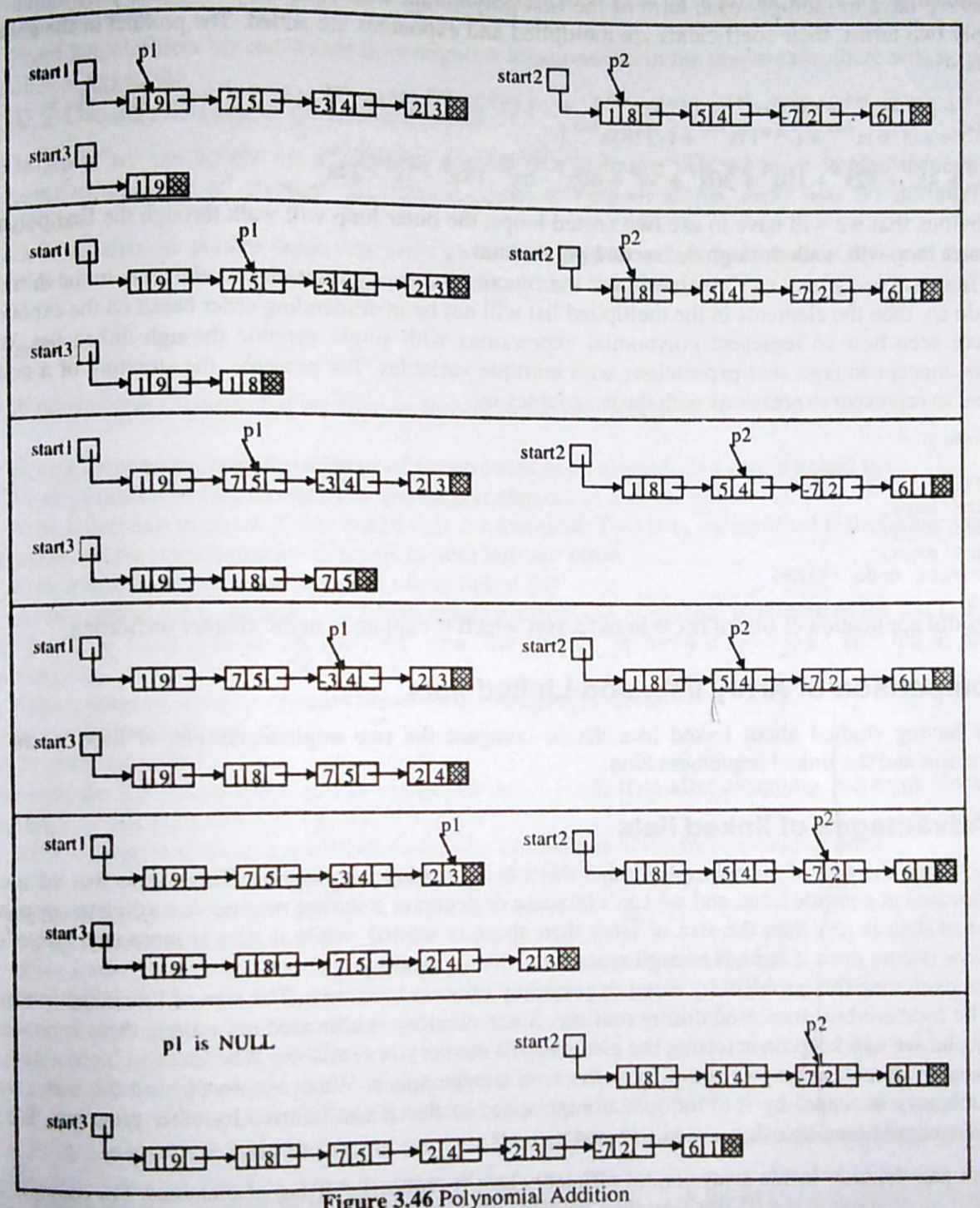


Figure 3.46 Polynomial Addition

3.9.3 Multiplication of 2 polynomials

Suppose we have to multiply the two polynomials given below-

$4x^3 + 5x^2 - 3x$

$2x^5 + 6x^4 + x^2 + 8$

For this we have to multiply each term of the first polynomial with each term of second polyno... we multiply two terms, their coefficients are multiplied and exponents are added. The product of the ... polynomials is-

$[\ (4*2)x^{3+5} + (4*6)x^{3+4} + (4*1)x^{3+2} + (4*8)x^{3+0}\ ] + [\ (5*2)x^{2+5}\ (5*6)x^{2+4} + (5*1)x^{2+2} + (5*8)x^{2+0}\ ] +$
$[\ (-3*2)x^{1+5} + (-3*6)x^{1+4} + (-3*1)x^{1+2} + (-3*8)x^{1+0}\ ]$

$8x^8 + 24x^7 + 4x^5 + 32x^3 + 10x^7 + 30x^6 + 5x^4 + 40x^2 - 6x^6 - 18x^5 - 3x^3 - 24x^1$

It is obvious that we will have to use two nested loops, the outer loop will walk through the first p... and the inner loop will walk through the second polynomial.

In the function `poly_mult()`, we have used the function `insert_s()` to insert elements in the r... we don't do so, then the elements in the multiplied list will not be in descending order based on the ex...

We have seen how to represent polynomial expressions with single variable through linked lis... extend this concept to represent expressions with multiple variables. For example, the structure of a... can be used to represent expressions with three variables is-

```
struct node
{
        float coef;
        int expx;
        int expy;
        int expz;
        struct node *link;
};
```

Another useful application of linked list is in radix sort which is explained in the chapter on Sorting.

3.10 Comparison of Array lists and Linked lists

Now after having studied about linked lists, let us compare the two implementations of lists i.e. t... implementation and the linked implementation.

3.10.1 Advantages of linked lists

(1) We know that the size of array is specified at the time of writing the program. This means that the ... space is allocated at compile time, and we can't increase or decrease it during runtime according to our ... the amount of data is less than the size of array then space is wasted, while if data is more than size ... then overflow occurs even if there is enough space available in memory.
Linked lists overcome this problem by using dynamically allocated memory. The size of linked list is ... and it can be increased or decreased during runtime. Since memory is allocated at runtime, there is no ... of memory and we can keep on inserting the elements till memory is available. Whenever we need a n... we dynamically allocate it i.e. we get it from the free storage space. When we don't need the node ... return the memory occupied by it to the free storage space so that it can be used by other programs. T... done these two operations by using `malloc()` and `free()`.

(2) Insertion and deletion inside arrays is not efficient since it requires shifting of elements. For examp... want to insert an element at the 0^{th} position then we have to shift all the elements of the array to the right... we want to delete the element present at the 0^{th} position then we have to shift all the elements to the left... are the worst cases of insertion and deletion and the efficiency is O(n). If the array consists of big record... this shifting is even more time consuming.
In linked lists, insertion and deletion requires only change in pointers. There is no physical movement ... only the links are altered.

(3) We know that in arrays all the elements are always stored in contiguous locations. Sometimes arrays ... be used because the amount of memory needed by them is not available in contiguous locations i.e. th...

memory required by the array is available but it is dispersed. So in this case we can't create an array in spite of available memory.
In linked list, elements are not stored in contiguous locations. So in the above case, there will be no problem in creation of linked list.

3.10.2 Disadvantages of linked lists

(1) In arrays we can access the n^{th} element directly, but in linked lists we have to pass through the first n-1 elements to reach the n^{th} element. So when it comes to random access, array lists are definitely better than linked lists.
(2) In linked lists the pointer fields take extra space that could have been used for storing some more data.
(3) Writing of programs for linked list is more difficult than that for arrays.

Exercise

In all the problems assume that we have an integer in the info part of nodes.

1. Write a function to count the number of occurrences of an element in a single linked list.
2. Write a function to find the smallest and largest element of a single linked list.
3. Write a function to check if two linked lists are identical. Two lists are identical if they have same number of elements and the corresponding elements in both lists are same
4. Write a function to create a copy of a single linked list.
5. Given a linked list L, write a function to create a single linked list that is reverse of the list L. For example if the list L is `1->2->3->4->5` then the new list should be `5->4->3->2->1`. The list L should remain unchanged.
6. Write a program to swap adjacent elements of a single linked list
(i) by exchanging info part
(ii) by rearranging links.
For example if a linked list is `1->2->3->4->5->6->7->8`, then after swapping adjacent elements it should become `2->1->4->3->6->5->8->7`.
7. Write a program to swap adjacent elements of a double linked list by rearranging links.
8. Write a program to swap the first and last elements of a single linked list
(i) by exchanging info part.
(ii) by rearranging links.
9. Write a function to move the largest element to the end of a single linked list.
10. Write a function to move the smallest element to the beginning of a single linked list.
11. Write a function for deleting all the nodes from a single linked list which have a value N.
12. Given a single linked list L1 which is sorted in ascending order, and another single linked list L2 which is not sorted, write a function to print the elements of second list according to the first list. For example if the first list is `1->2->5->7->8`, then the function should print the 1^{st}, 2^{nd}, 5^{th}, 7^{th}, 8^{th} elements of second list.
13. Write a program to remove first node of the list and insert it at the end, without changing info part of any node.
14. Write a program to remove the last node of the list and insert it in the beginning, without changing info part of any node.
15. Write a program to move a node n positions forward in a single linked list.
16. Write a function to delete a node from a single linked list. The only information we have is a pointer to the node that has to be deleted.
17. Write functions to insert a node just before and just after a node pointed to by a pointer p, without using the pointer start.

18. What is wrong in the following code that attempts to free all the nodes of a single linked list.

```
p=start;
while(p!=NULL)
{
        free(p);
        p=p->link;
}
```

Write a function `Destroy()` that frees all the nodes of a single linked list.

19. Write a function to remove duplicates from a sorted single linked list.

20. Write a function to remove duplicates from an unsorted single linked list.

21. Write a function to create a linked list that is intersection of two single linked lists, i.e. it contains elements which are common to both the lists.

22. Write a function to create a linked list that is union of two single linked lists, i.e. it contains all elements both lists and if an element is repeated in both lists, then it is included only once.

23. Given a list L1, delete all the nodes having negative numbers in info part and insert them into list L2, the nodes having positive numbers into list L3. No new nodes should be allocated.

24. Given a linked list L1, create two linked lists one having the even numbers of L1 and the other having odd numbers of L1. Don't change the list L1.

25. Write a function to delete alternate nodes(even numbered nodes) from a single linked list. For example list is `1->2->3->4->5->6->7` then the resulting list should be `1->3->5->7`.

26. Write a function to get the n^{th} node from the end of a single linked list, without counting the elements reversing the list.

27. Write a function to find out whether a single linked is NULL terminated or contains a cycle/loop. If it contains a cycle, find the length of the cycle and the length of the whole list. Find the node that causes the i.e. the node at which the cycle starts. This node is pointed by two different nodes of the list. Remove the from the list and make it NULL terminated.

28. Write a function to find out the middle node of a single linked list without counting all the elements list.

29. Write a function to split a single linked list into two halves.

30. Write a function to split a single linked list into two lists at a node containing the given information.

31. Write a function to split a single linked list into two lists such that the alternate nodes(even numbered go to a new list.

32. Write a function to combine the alternate nodes of two null terminated single linked lists. For example first list is `1->2->3->4` and the second list is `5->7->8->9` then after combining them the first list should `1->5->2->7->3->8->4->9` and second list should be empty. If both lists are not of the same length, the remaining nodes of the longer list are taken in the combined list. For example if the first list is `1->2->3->` the second list is `5->7` then the combined list should be `1->5->2->7->3->4`.

33. Suppose there are two null terminated single linked lists which merge at a given point and share nodes after that merge point (Y shaped lists). Write a function to find the merge point (intersection point).

34. Create a double linked list in which info part of each node contains a digit of a given number. The should be stored in reverse order, i.e. the least significant digit should be stored in the first node and the significant digit in the last node. If the number is 5468132 then the linked list should be `2->3->1->8->6->4->5`. Write a function to add two numbers represented by linked lists.

35. Modify the program in the previous problem so that now in each node of the list we can store 4 digits given number. For example if the number is 23156782913287 then the linked list would be `3287->8291->1567->23`.

36. Write a function to find whether a linked list is palindrome or not.

37. Construct a linked list in which each node has the following information about a student - rollno, marks in 3 subjects. Enter records of different students in list. Traverse this list and calculate the total percentage of each student. Count the number of students who scored passing marks(above 40 percent).

38. Modify the previous program so that now the names are inserted in alphabetical order in the list. Make program menu driven with the following menus-

(i) Create list (ii) Insert (iii) Delet
Delete menu should have the fa
deleted. Display record menu sh
should display the number of s
given the roll number.

(i) Create list (ii) Insert (iii) Delete (iv) Modify (v) Display record (vi) Display result
Delete menu should have the facility of entering name of a student and the record of that student should be deleted. Display record menu should ask for the roll no of a student and display all information. Display result should display the number of students who have passed. Modify menu has the facility of modifying a record given the roll number.