THE CONCEPT OF THE LINKED LIST

Introduction

When dealing with many problems we need a dynamic list, dynamic in the sense that the size requirement need not be known at compile time. Thus, the list may grow or shrink during runtime. A *linked list* is a data structure that is used to model such a dynamic list of data items, so the study of the linked lists as one of the data structures is important.

Concept

An array is represented in memory using sequential mapping, which has the property that elements are fixed distance apart. But this has the following disadvantage: It makes insertion or deletion at any arbitrary position in an array a costly operation, because this involves the movement of some of the existing elements.

When we want to represent several lists by using arrays of varying size, either we have to represent each list using a separate array of maximum size or we have to represent each of the lists using one single array. The first one will lead to wastage of storage, and the second will involve a lot of data movement.

So we have to use an alternative representation to overcome these disadvantages. One alternative is a linked representation. In a linked representation, it is not necessary that the elements be at a fixed distance apart. Instead, we can place elements anywhere in memory, but to make it a part of the same list, an element is required to be linked with a previous element of the list. This can be done by storing the address of the next element in the previous element itself. This requires that every element be capable of holding the data as well as the address of the next element. Thus every element must be a structure with a minimum of two fields, one for holding the data value, which we call a data field, and the other for holding the address of the next element, which we call link field.

Therefore, a linked list is a list of elements in which the elements of the list can be placed anywhere in memory, and these elements are linked with each other using an explicit link field, that is, by storing the address of the next element in the link field of the previous element.

Program

Here is a program for building and printing the elements of the linked list:

```
# include <stdio.h>
# include <stdlib.h>
struct node
{
int data;
struct node *link;
};
struct node *insert(struct node *p, int n)
```

```
   {
   struct node *temp;
   /* if the existing list is empty then insert a new node as the
starting node */
   if(p==NULL)
   {
      p=(struct node *)malloc(sizeof(struct node)); /* creates new node
data value passes
   as parameter */

         if(p==NULL)
         {
      printf("Error\n");
            exit(0);
         }
         p-> data = n;
         p-> link = p; /* makes the pointer pointing to itself because it
is a circular list*/
      }
      else
      {
      temp = p;
   /* traverses the existing list to get the pointer to the last node of
   it */
      while (temp-> link != p)
         temp = temp-> link;
            temp-> link = (struct node *)malloc(sizeof(struct node)); /*
   creates new node using
             data value passes as
               parameter and puts its
             address in the link field
             of last node of the
             existing list*/
           if(temp -> link == NULL)
           {
         printf("Error\n");
              exit(0);
           }
           temp = temp-> link;
           temp-> data = n;
           temp-> link = p;
          }
          return (p);
   }
   void printlist ( struct node *p )
   {
   struct node *temp;
    temp = p;
   printf("The data values in the list are\n");
      if(p!= NULL)
      {
      do
            {
            printf("%d\t",temp->data);
            temp=temp->link;
            } while (temp!= p);
      }
```

```
    else
       printf("The list is empty\n");
}

void main()
{
    int n;
    int x;
    struct node *start = NULL ;
    printf("Enter the nodes to be created \n");
    scanf("%d",&n);
    while ( n -- > 0 )
    {
printf( "Enter the data values to be placed in a node\n");
       scanf("%d",&x);
       start = insert ( start, x );
    }
    printf("The created list is\n");
    printlist ( start );
}
```

Explanation

1. This program uses a strategy of inserting a node in an existing list to get the list created. An `insert` function is used for this.
2. The `insert` function takes a pointer to an existing list as the first parameter, and a data value with which the new node is to be created as a second parameter, creates a new node by using the data value, appends it to the end of the list, and returns a pointer to the first node of the list.
3. Initially the list is empty, so the pointer to the starting node is `NULL`. Therefore, when `insert` is called first time, the new node created by the `insert` becomes the start node.
4. Subsequently, the insert traverses the list to get the pointer to the last node of the existing list, and puts the address of the newly created node in the link field of the last node, thereby appending the new node to the existing list.
5. The main function reads the value of the number of nodes in the list. Calls iterate that many times by going in a `while` loop to create the links with the specified number of nodes.

INSERTING A NODE BY USING RECURSIVE PROGRAMS

Introduction

A linked list is a recursive data structure. A *recursive data structure* is a data structure that has the same form regardless of the size of the data. You can easily write recursive programs for such data structures.

Program

```
# include <stdio.h>
# include <stdlib.h>
struct node
{
int data;
struct node *link;
};
struct node *insert(struct node *p, int n)
{
   struct node *temp;
   if(p==NULL)
   {
      p=(struct node *)malloc(sizeof(struct node));
      if(p==NULL)
      {
   printf("Error\n");
         exit(0);
   }
   p-> data = n;
   p-> link = NULL;
}
else
      p->link = insert(p->link,n);/* the while loop replaced by
recursive call */
   return (p);
}
void printlist ( struct node *p )
{
      printf("The data values in the list are\n");
      while (p!= NULL)
      {
   printf("%d\t",p-> data);
         p = p-> link;
      }
}
void main()
{
      int n;
      int x;
      struct node *start = NULL ;
      printf("Enter the nodes to be created \n");
      scanf("%d",&n);
      while ( n- > 0 )
      {
   printf( "Enter the data values to be placed in a node\n");
         scanf("%d",&x);
         start = insert ( start, x );
      }
      printf("The created list is\n");
      printlist ( start );
}
```

SORTING AND REVERSING A LINKED LIST

Introduction

To sort a linked list, first we traverse the list searching for the node with a minimum data value. Then we remove that node and append it to another list which is initially empty. We repeat this process with the remaining list until the list becomes empty, and at the end, we return a pointer to the beginning of the list to which all the nodes are moved, as shown in Figure 20.1.

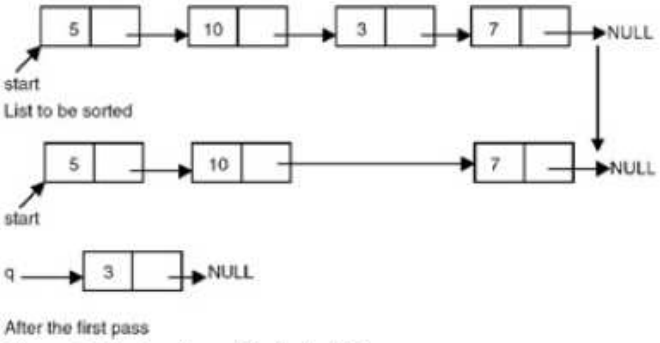


Figure 20.1: Sorting of a linked list.

To reverse a list, we maintain a pointer each to the previous and the next node, then we make the link field of the current node point to the previous, make the previous equal to the current, and the current equal to the next, as shown in Figure 20.2.

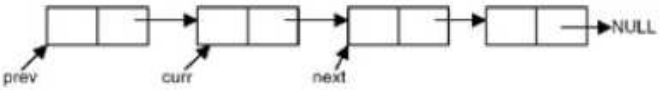


Figure 20.2: A linked list showing the previous, current, and next nodes at some point during reversal process.

Therefore, the code needed to reverse the list is

```
Prev = NULL;
While (curr != NULL)
{
   Next = curr->link;
   Curr -> link = prev;
   Prev = curr;
   Curr = next;
}
```

Program

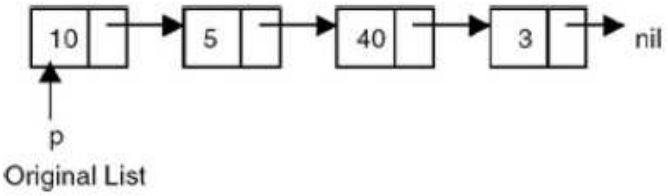
```
# include <stdio.h>
# include <stdlib.h>
struct node
{
   int data;
   struct node *link;
};
struct node *insert(struct node *p, int n)
{
   struct node *temp;
   if(p==NULL)
   {
      p=(struct node *)malloc(sizeof(struct node));
      if(p==NULL)
      {
            printf("Error\n");
         exit(0);
      }
      p-> data = n;
      p-> link = NULL;
   }
   else
   {
      temp = p;
      while (temp-> link!= NULL)
      temp = temp-> link;
      temp-> link = (struct node *)malloc(sizeof(struct node));
      if(temp -> link == NULL)
      {
            printf("Error\n");
         exit(0);
      }
       temp = temp-> link;
       temp-> data = n;
       temp-> link = null;
      }
      return(p);
}

void printlist ( struct node *p )
{
      printf("The data values in the list are\n");
      while (p!= NULL)
      {
            printf("%d\t",p-> data);
         p = p-> link;
      }
}

/* a function to sort reverse list */
struct node *reverse(struct node *p)
{
   struct node *prev, *curr;
   prev = NULL;
   curr = p;
```

```
   while (curr != NULL)
   {
      p = p-> link;
      curr-> link = prev;
      prev = curr;
      curr = p;
   }
   return(prev);
}
/* a function to sort a list */
struct node *sortlist(struct node *p)
{
   struct node *temp1,*temp2,*min,*prev,*q;
   q = NULL;
   while(p != NULL)
   {
         prev = NULL;
      min = temp1 = p;
      temp2 = p -> link;
      while ( temp2 != NULL )
      {
               if(min -> data > temp2 -> data)
         {
                     min = temp2;
                     prev = temp1;
         }
         temp1 = temp2;
         temp2 = temp2-> link;
      }
      if(prev == NULL)
            p = min -> link;
      else
            prev -> link = min -> link;
      min -> link = NULL;
      if( q == NULL)
      q = min; /* moves the node with lowest data value in the list
pointed to by p to the list
   pointed to by q as a first node*/
         else
         {
            temp1 = q;
            /* traverses the list pointed to by q to get pointer to its
last node */
            while( temp1 -> link != NULL)
                        temp1 = temp1 -> link;
            temp1 -> link = min; /* moves the node with lowest data value
in the list pointed to
   by p to the list pointed to by q at the end of list pointed by
   q*/
      }
   }
   return (q);
}

void main()
{
      int n;
```

```
        int x;
        struct node *start = NULL ;
        printf("Enter the nodes to be created \n");
        scanf("%d",&n);
        while ( n- > 0 )
        {
                printf( "Enter the data values to be placed in a
node\n");
           scanf("%d",&x);
           start = insert ( start,x);
        }
        printf("The created list is\n");
        printlist ( start );
        start = sortlist(start);
        printf("The sorted list is\n");
        printlist ( start );
        start = reverse(start);
        printf("The reversed list is\n");
        printlist ( start );
}
```

Explanation

The working of the sorting function on an example list is shown in Figure 20.3.



Original List

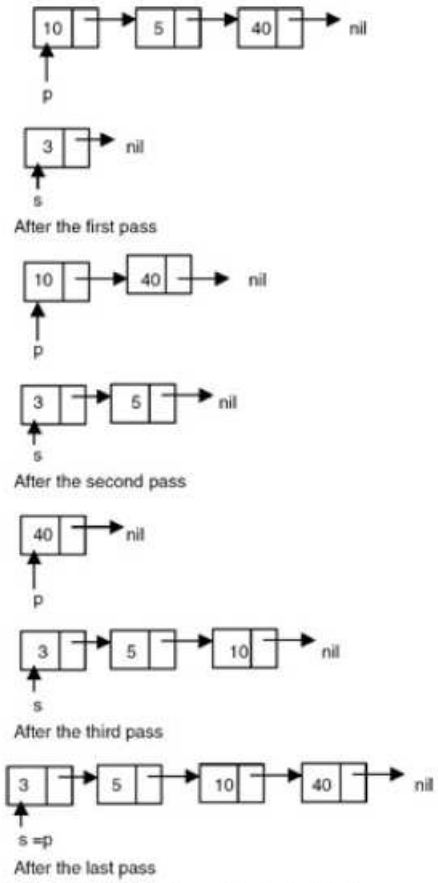


Figure 20.3: Sorting of a linked list.

The working of a reverse function is shown in Figure 20.4.

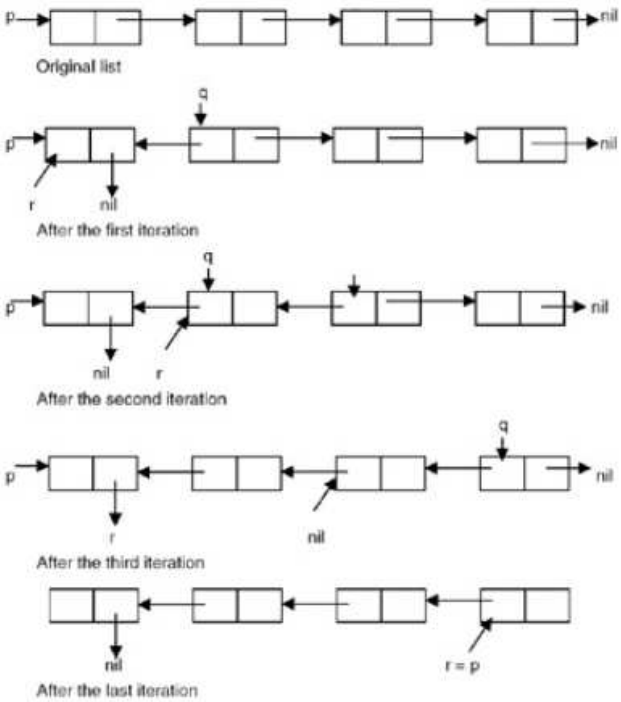


Figure 20.4: Reversal of a list.

DELETING THE SPECIFIED NODE IN A SINGLY LINKED LIST

Introduction

To delete a node, first we determine the node number to be deleted (this is based on the assumption that the nodes of the list are numbered serially from 1 to n). The list is then traversed to get a pointer to the node whose number is given, as well as a pointer to a node that appears before the node to be deleted. Then the link field of the node that appears before the node to be deleted is made to point to the node that appears after the node to be deleted, and the node to be deleted is freed. Figures 20.5 and 20.6 show the list before and after deletion, respectively.

Program

```
# include <stdio.h>
# include <stdlib.h>
struct node *delet ( struct node *, int );
int length ( struct node * );
struct node
{
   int data;
   struct node *link;
};
struct node *insert(struct node *p, int n)
{
```

```
    struct node *temp;
    if(p==NULL)
    {
       p=(struct node *)malloc(sizeof(struct node));
       if(p==NULL)
       {
             printf("Error\n");
          exit(0);
       }
       p-> data = n;
       p-> link = NULL;
    }
    else
    {
       temp = p;
       while (temp-> link != NULL)
        temp = temp-> link;
        temp-> link = (struct node *)malloc(sizeof(struct node));
       if(temp -> link == NULL)
       {
             printf("Error\n");
          exit(0);
       }
       temp = temp-> link;
       temp-> data = n;
       temp-> link = NULL;
       }
       return (p);
}

void printlist ( struct node *p )
{
       printf("The data values in the list are\n");
       while (p!= NULL)
       {
             printf("%d\t",p-> data);
          p = p-> link;
       }
}

void main()
{
   int n;
   int x;
   struct node *start = NULL;
   printf("Enter the nodes to be created \n");
   scanf("%d",&n);
   while ( n- > 0 )
   {
       printf( "Enter the data values to be placed in a node\n");
      scanf("%d",&x);
      start = insert ( start, x );
   }
   printf(" The list before deletion id\n");
   printlist ( start );
   printf("% \n Enter the node no \n");
   scanf ( " %d",&n);
```

```
   start = delet (start , n );
   printf(" The list after deletion is\n");
   printlist ( start );
}

   /* a function to delete the specified node*/
struct node *delet ( struct node *p, int node_no )
{

   struct node *prev, *curr ;
   int i;

   if (p == NULL )
   {
      printf("There is no node to be deleted \n");
   }
   else
   {
      if ( node_no > length (p))
      {
            printf("Error\n");
      }
      else
      {
            prev = NULL;
            curr = p;
            i = 1 ;
            while ( i < node_no )
            {
                  prev = curr;
                  curr = curr-> link;
                  i = i+1;
            }
            if ( prev == NULL )
            {
                  p = curr -> link;
                  free ( curr );
            }
            else
            {
               prev -> link = curr -> link ;
               free ( curr );
            }      }
   }
   return(p);
}
/* a function to compute the length of a linked list */
int length ( struct node *p )
{
   int count = 0 ;
   while ( p != NULL )
   {
      count++;
      p = p->link;
   }
   return ( count ) ;
}
```

Explanation

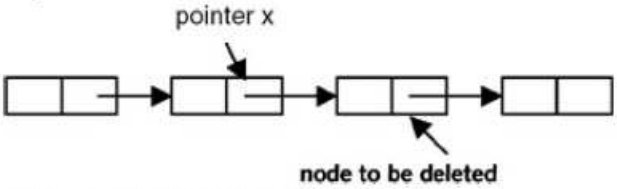


Figure 20.5: Before deletion.

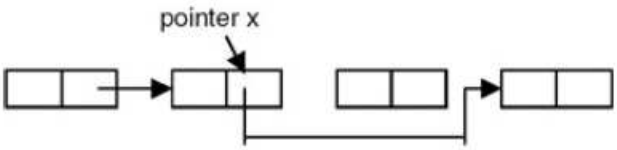


Figure 20.6: After deletion.

INSERTING A NODE AFTER THE SPECIFIED NODE IN A SINGLY LINKED LIST

Introduction

To insert a new node after the specified node, first we get the number of the node in an existing list after which the new node is to be inserted. This is based on the assumption that the nodes of the list are numbered serially from 1 to n. The list is then traversed to get a pointer to the node, whose number is given. If this pointer is x, then the link field of the new node is made to point to the node pointed to by x, and the link field of the node pointed to by x is made to point to the new node. Figures 20.7 and 20.8 show the list before and after the insertion of the node, respectively.

Program

```
   # include <stdio.h>
   # include <stdlib.h>
   int length ( struct node * );
   struct node
   {
      int data;
      struct node *link;
   };

   /* a function which appends a new node to an existing list used for
building a list */
   struct node *insert(struct node *p, int n)
   {
      struct node *temp;
      if(p==NULL)
      {
          p=(struct node *)malloc(sizeof(struct node));
         if(p==NULL)
         {
                 printf("Error\n");
```

```
            exit(0);
         }
         p-> data = n;
         p-> link = NULL;
      }
      else
      {
         temp = p;
         while (temp-> link != NULL)
                temp = temp-> link;
         temp-> link = (struct node *)malloc(sizeof(struct node));
        if(temp -> link == NULL)
         {
                 printf("Error\n");
            exit(0);
         }
         temp = temp-> link;
         temp-> data = n;
         temp-> link= NULL;
      }
      return (p);
   }
   /* a function which inserts a newly created node after the specified
node */
   struct node * newinsert ( struct node *p, int node_no, int value )
   {
      struct node *temp, * temp1;
      int i;
      if ( node_no <= 0 || node_no > length (p))
      {
              printf("Error! the specified node does not exist\n");
         exit(0);
      }
      if ( node_no == 0)
      {
         temp = ( struct node * )malloc ( sizeof ( struct node ));
         if ( temp == NULL )
         {
                 printf( " Cannot allocate \n");
                 exit (0);
         }
         temp -> data = value;
         temp -> link = p;
         p = temp ;
   }
   else
   {
       temp = p ;
      i = 1;
      while ( i < node_no )
      {
            i = i+1;
            temp = temp-> link ;
      }
      temp1 = ( struct node * )malloc ( sizeof(struct node));
      if ( temp == NULL )
      {
```

```
            printf ("Cannot allocate \n");
            exit(0)
        }
        temp1 -> data = value ;
        temp1 -> link = temp -> link;
        temp -> link = temp1;
    }
    return (p);
    }

    void printlist ( struct node *p )
    {
        printf("The data values in the list are\n");
        while (p!= NULL)
        {
            printf("%d\t",p-> data);
            p = p-> link;
        }
    }
    void main ()
    {
        int n;
        int x;
        struct node *start = NULL;
        printf("Enter the nodes to be created \n");
        scanf("%d",&n);
        while ( n- > 0 )
        {
            printf( "Enter the data values to be placed in a node\n");
            scanf("%d",&x);
            start = insert ( start, x );
        }
        printf(" The list before deletion is\n");
        printlist ( start );
        printf(" \n Enter the node no after which the insertion is to be
done\n");
        scanf ( " %d",&n);
        printf("Enter the value of the node\n");
        scanf("%d",&x);
        start = newinsert(start,n,x);
        printf("The list after insertion is \n");
        printlist(start);
    }
```

Explanation

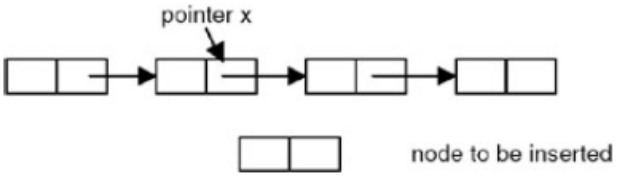


Figure 20.7: Before insertion.

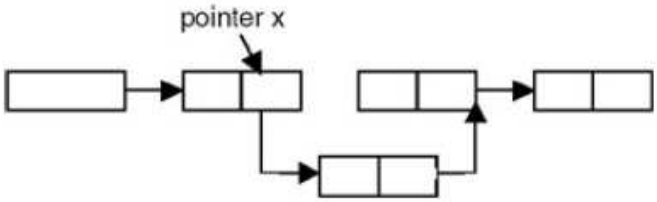


Figure 20.8: After insertion.

INSERTING A NEW NODE IN A SORTED LIST

Introduction

To insert a new node into an already sorted list, we compare the data value of the node to be inserted with the data values of the nodes in the list starting from the first node. This is continued until we get a pointer to the node that appears immediately before the node in the list whose data value is greater than the data value of the node to be inserted.

Program

Here is a complete program to insert an element in a sorted list of elements using the linked list representation so that after insertion, it will remain a sorted list.

```
# include <stdio.h>
# include <stdlib.h>
struct node
{
int data;
struct node *link;
};
struct node *insert(struct node *, int);
struct node *sinsert(struct node*, int );
void printlist ( struct node * );
struct node *sortlist(struct node *);

struct node *insert(struct node *p, int n)
{
   struct node *temp;
   if(p==NULL)
   {
      p=(struct node *)malloc(sizeof(struct node));
      if(p==NULL)
      {
   printf("Error\n");
            exit(0);
      }
      p-> data = n;
      p-> link = NULL;
   }
   else
   {
      temp = p;
```

```
       while (temp-> link!= NULL)
      temp = temp-> link;
       temp-> link = (struct node *)malloc(sizeof(struct node));
       if(temp -> link == NULL)
       {
   printf("Error\n");
          exit(0);
        }
      temp = temp-> link;
        temp-> data = n;
        temp-> link = NULL;
        }
       return (p);
}

void printlist ( struct node *p )
{
       printf("The data values in the list are\n");
       while (p!= NULL)
       {
   printf("%d\t",p-> data);
         p = p-> link;
       }
   }

/* a function to sort a list */
struct node *sortlist(struct node *p)
{
   struct node *temp1,*temp2,*min,*prev,*q;
   q = NULL;
   while(p != NULL)
   {
   prev = NULL;
      min = temp1 = p;
      temp2 = p -> link;
      while ( temp2 != NULL )
      {
         if(min -> data > temp2 -> data)
              {
         min = temp2;
                 prev = temp1;
               }
               temp1 = temp2;
               temp2 = temp2-> link;
            }
      if(prev == NULL)
            p = min -> link;
            else
            prev -> link = min -> link;
            min -> link = NULL;
            if( q == NULL)
            q = min; /* moves the node with lowest data value in the list
pointed to by p to the list
   pointed to by q as a first node*/
            else
            {
               temp1 = q;
```

```
               /* traverses the list pointed to by q to get pointer to its
last node */
               while( temp1 -> link != NULL)
               temp1 = temp1 -> link;
               temp1 -> link = min; /* moves the node with lowest data value
in the list pointed to
   by p to the list pointed to by q at the end of list pointed by
   q*/
         }
      }
      return (q);
   }

   /* a function to insert a node with data value n in a sorted list
pointed to by p*/
   struct node *sinsert(struct node *p, int n)
   {
      struct node *curr, *prev;
      curr =p;
      prev = NULL;
     while(curr ->data < n)
           {
                       prev = curr;
                 curr = curr->link;
           }
   if ( prev == NULL) /* the element is to be inserted at the start of
the list because
                    it is less than the data value of the first node*/
       {
            curr = (struct node *) malloc(sizeof(struct node));
            if( curr == NULL)
              {
                printf("error cannot allocate\n");
                exit(0);
              }
           curr->data = n;
           curr->link = p;
           p = curr;
       }
       else
        {
            curr->data = n;
            curr->link = prev->link;
            prev->link = curr;
        }
      return(p);
      }

      void main()
      {
         int n;
         int x;
         struct node *start = NULL ;
         printf("Enter the nodes to be created \n");
         scanf("%d",&n);
         while ( n-- > 0 )
         {
```

```
    printf( "Enter the data values to be placed in a node\n");
          scanf("%d",&x);
          start = insert ( start,x);
       }
       printf("The created list is\n");
       printlist ( start );
       start = sortlist(start);
       printf("The sorted list is\n");
       printlist ( start );
       printf("Enter the value to be inserted\n");
       scanf("%d",&n);
       start = sinsert(start,n);
       printf("The list after insertion is\n");
       printlist ( start );
}
```

Explanation

1. If this pointer is `prev`, then `prev` is checked for a `NULL` value.
2. If `prev` is `NULL`, then the new node is created and inserted as the first node in the list.
3. When `prev` is not `NULL`, then a new node is created and inserted after the node pointed by `prev`, as shown in Figure 20.9.

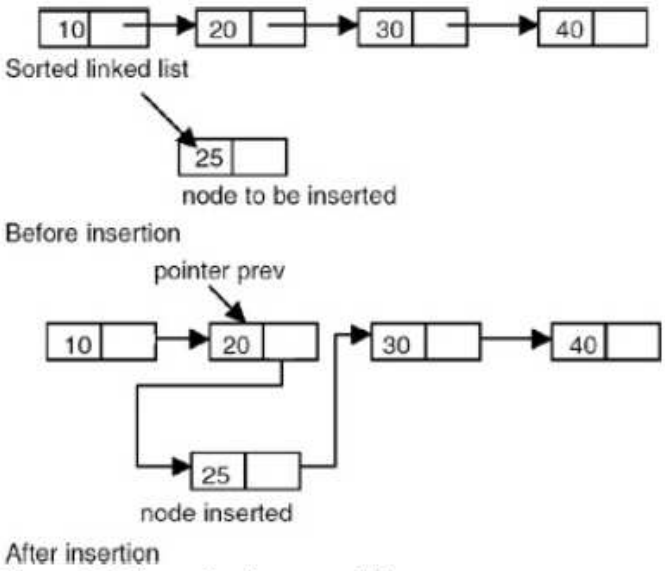


Figure 20.9: Insertion in a sorted list.

COUNTING THE NUMBER OF NODES OF A LINKED LIST

Introduction

Counting the number of nodes of a singly linked list requires maintaining a counter that is initialized to 0 and incremented by 1 each time a node is encountered in the process of traversing a list from the start.

Here is a complete program that counts the number of nodes in a singly linked chain p, where p is a pointer to the first node in the list.

Program

```
# include <stdio.h>
# include <stdlib.h>
struct node
{
int data;
struct node *link;
};
struct node *insert(struct node *, int);
int nodecount(struct node*);
void printlist ( struct node * );

struct node *insert(struct node *p, int n)
{
   struct node *temp;
   if(p==NULL)
   {
      p=(struct node *)malloc(sizeof(struct node));
      if(p==NULL)
      {
   printf("Error\n");
          exit(0);
      }
      p-> data = n;
      p-> link = NULL;
   }
   else
   {
      temp = p;
      while (temp-> link!= NULL)
     temp = temp-> link;
      temp-> link = (struct node *)malloc(sizeof(struct node));
      if(temp -> link == NULL)
      {
   printf("Error\n");
        exit(0);
      }
     temp = temp-> link;
      temp-> data = n;
      temp-> link = NULL;
```

```
      }
      return (p);
}

void printlist ( struct node *p )
{
      printf("The data values in the list are\n");
      while (p!= NULL)
      {
   printf("%d\t",p-> data);
         p = p-> link;
      }
}

/* A function to count the number of nodes in a singly linked list */
int nodecount (struct node *p )
{
   int count=0;
   while (p != NULL)
   {
      count ++;
      p = p->link;
   }
      return(count);
}

void main()
{
   int n;
   int x;
   struct node *start = NULL ;
   printf("Enter the nodes to be created \n");
   scanf("%d",&n);
   while ( n-- > 0 )
   {
printf( "Enter the data values to be placed in a node\n");
      scanf("%d",&x);
      start = insert ( start,x);
   }
   printf("The created list is\n");
   printlist ( start );
   n = nodecount(start);
   printf("The number of nodes in a list are: %d\n",n);

}
```

MERGING OF TWO SORTED LISTS

Introduction

Merging of two sorted lists involves traversing the given lists and comparing the data values stored in the nodes in the process of traversing.

If p and q are the pointers to the sorted lists to be merged, then we compare the data value stored in the first node of the list pointed to by p with the data value stored in the first node of the list pointed to by q. And, if the data value in the first node of the list pointed to by p is less than the data value in the first node of the list pointed to by q, make the first node of the resultant/merged list to be the first node of the list pointed to by p, and advance the pointer p to make it point to the next node in the same list.

If the data value in the first node of the list pointed to by p is greater than the data value in the first node of the list pointed to by q, make the first node of the resultant/merged list to be the first node of the list pointed to by q, and advance the pointer q to make it point to the next node in the same list.

Repeat this procedure until either p or q becomes NULL. When one of the two lists becomes empty, append the remaining nodes in the non-empty list to the resultant list.

Program

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

struct node *merge (struct node *, struct node *);
struct node *insert(struct node *p, int n)
{
   struct node *temp;
   if(p==NULL)
   {
      p=(struct node *)malloc(sizeof(struct node));
      if(p==NULL)
      {
   printf("Error\n");
          exit(0);
      }
      p-> data = n;
      p-> link = NULL;
   }
   else
   {
      temp = p;
      while (temp-> link!= NULL)
```

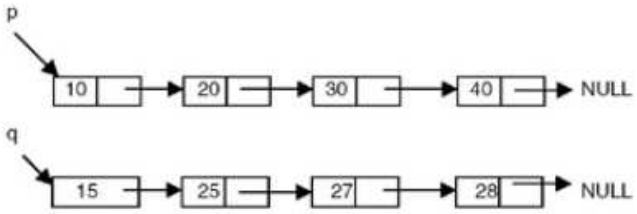
```
      temp = temp-> link;
      temp-> link = (struct node *)malloc(sizeof(struct node));
      if(temp -> link == NULL)
      {
   printf("Error\n");
          exit(0);
      }
     temp = temp-> link;
      temp-> data = n;
      temp-> link = NULL;
      }
      return (p);
   }

   void printlist ( struct node *p )
   {
      printf("The data values in the list are\n");
      while (p!= NULL)
         {
      printf("%d\t",p-> data);
           p = p-> link;
         }
   }
   /* a function to sort a list */
struct node *sortlist(struct node *p)
{
    struct node *temp1,*temp2,*min,*prev,*q;
   q = NULL;
   while(p != NULL)
   {
   prev = NULL;
      min = temp1 = p;
      temp2 = p -> link;
      while ( temp2 != NULL )
      {
   if(min -> data > temp2 -> data)
        {
   min = temp2;
          prev = temp1;
        }
        temp1 = temp2;
        temp2 = temp2-> link;
      }
   if(prev == NULL)
        p = min -> link;
        else
        prev -> link = min -> link;
        min -> link = NULL;
        if( q == NULL)
        q = min; /* moves the node with lowest data value in the list
pointed to by p to the list
   pointed to by q as a first node*/
        else
        {
           temp1 = q;
           /* traverses the list pointed to by q to get pointer to its
last node */
```

```
            while( temp1 -> link != NULL)
            temp1 = temp1 -> link;
            temp1 -> link = min; /* moves the node with lowest data value
in the list pointed to
   by p to the list pointed to by q at the end of list pointed by
   q*/
         }
      }
      return (q);
   }

   void main()
   {
      int n;
      int x;
      struct node *start1 = NULL ;
      struct node *start2 = NULL;
      struct node *start3 = NULL;
      /* The following code creates and sorts the first list */
      printf("Enter the number of nodes in the first list \n");
      scanf("%d",&n);
      while ( n-- > 0 )
      {
   printf( "Enter the data value to be placed in a node\n");
         scanf("%d",&x);
         start1 = insert ( start1,x);
      }
      printf("The first list is\n");
      printlist ( start1);
      start1 = sortlist(start1);
      printf("The sorted list1 is\n");
      printlist ( start1 );
      /* the following creates and sorts the second list*/
      printf("Enter the number of nodes in the second list \n");
      scanf("%d",&n);
      while ( n-- > 0 )
      {
         printf( "Enter the data value to be placed in a node\n");
         scanf("%d",&x);
         start2 = insert ( start2,x);
      }
      printf("The second list is\n");
      printlist ( start2);
      start2 = sortlist(start2);
      printf("The sorted list2 is\n");
      printlist ( start2 );
      start3 = merge(start1,start2);
      printf("The merged list is\n");
      printlist ( start3);
   }

   /* A function to merge two sorted lists */
   struct node *merge (struct node *p, struct node *q)
   {
      struct node *r=NULL,*temp;
      if (p == NULL)
           r = q;
```

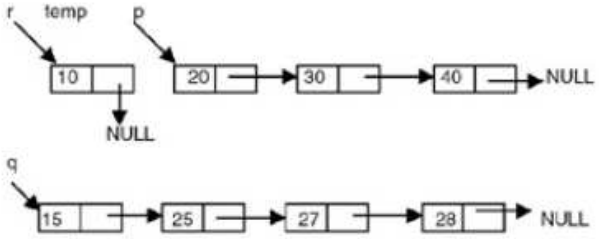
```
   else
   if(q == NULL)
        r = p;
   else
      {
         if (p->data < q->data )
         {
r = p;
temp = p;
p = p->link;
temp->link = NULL;
          }
          else
          {
r = q;
temp =q;
q =q->link;
temp->link = NULL;
          }
          while((p!= NULL) && (q != NULL))
          {
              if (p->data < q->data)
      {
      temp->link =p;
      p = p->link;
      temp =temp->link;
      temp->link =NULL;
      }
                  else
      {
         temp->link =q;
         q = q->link;
         temp =temp->link;
         temp->link =NULL;
         }
   }
      if (p!= NULL)
              temp->link = p;
      if (q != NULL)
              temp->link = q;
     }
return( r) ;
}
```

Explanation

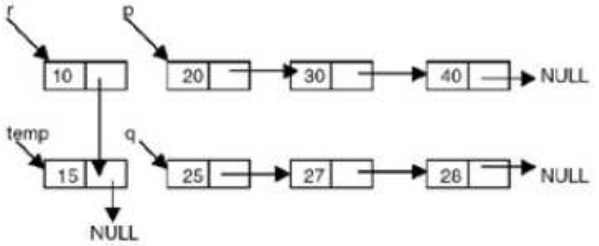
If the following lists are given as input, then what would be the output of the program after each pass? This is shown here:



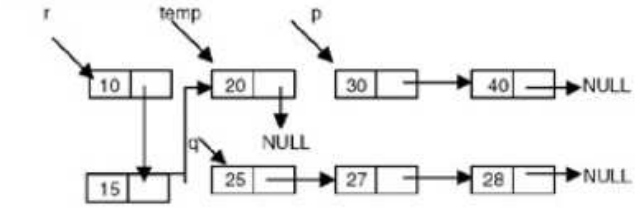
Two sorted lists before merging



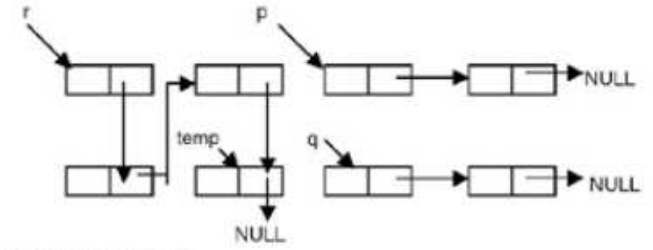
After the first pass



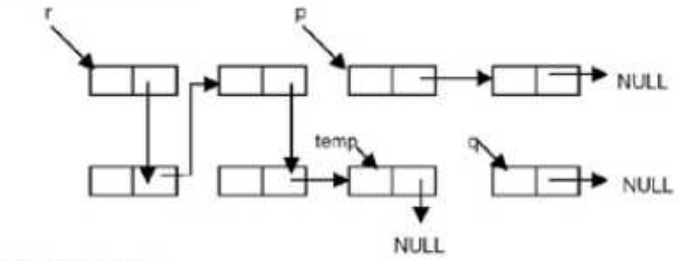
After the second pass



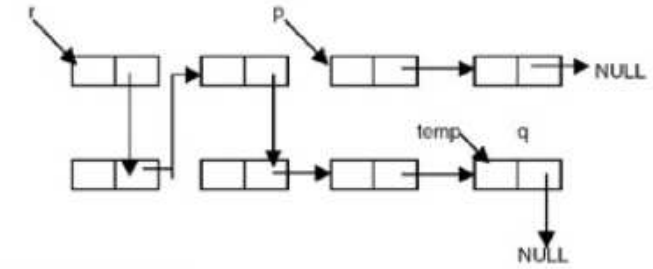
After the third pass



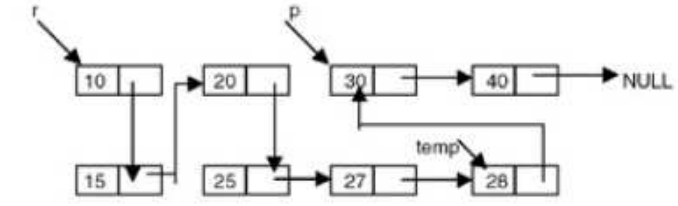
After the fourth pass



After the fifth pass



After the sixth pass



Final merged list

CIRCULAR LINKED LISTS

Introduction

A circular list is a list in which the link field of the last node is made to point to the start/first node of the list, as shown in Figure 20.14.

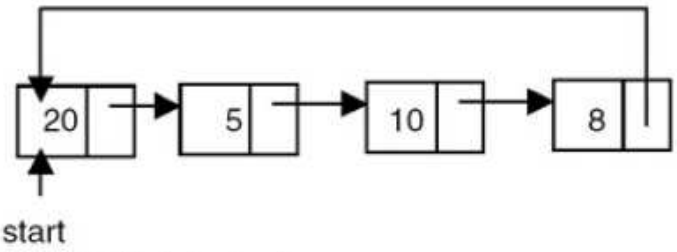


Figure 20.14: A circular list.

In the case of circular lists, the empty list also should be circular. So to represent a circular list that is empty, it is required to use a header node or a head-node whose data field contents are irrelevant, as shown in Figure 20.15.

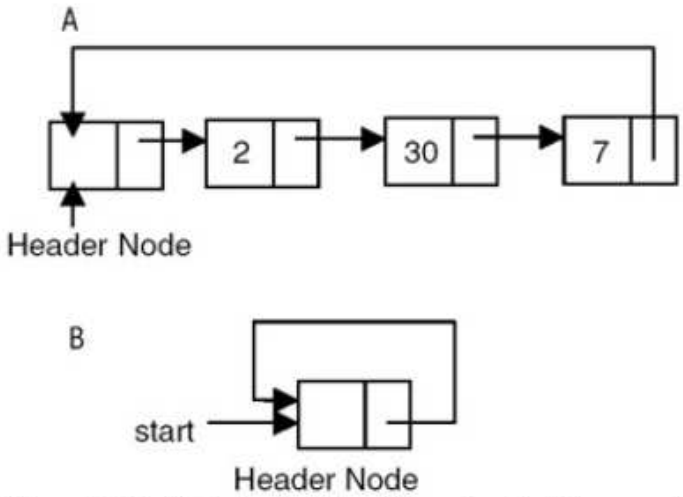


Figure 20.15: (A) A circular list with head node, (B) an empty circular list.

Program

Here is a program for building and printing the elements of the circular linked list.

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

```
struct node *insert(struct node *p, int n)
{
   struct node *temp;
   /* if the existing list is empty then insert a new node as the
starting node */
       if(p==NULL)
       {
         p=(struct node *)malloc(sizeof(struct node)); /* creates new
node data value passes
   as parameter */
        if(p==NULL)
        {
                 printf("Error\n");
                 exit(0);
         }
         p-> data = n;
          p-> link = p; /* makes the pointer pointing to itself because it
is a circular list*/
      }
      else
      {
         temp = p;
   /* traverses the existing list to get the pointer to the last node of
it */
         while (temp-> link != p)
                temp = temp-> link;
         temp-> link = (struct node *)malloc(sizeof(struct node)); /*
creates new node using
         data value passes as
         parameter and puts its
         address in the link field
         of last node of the
         existing list*/
         if(temp -> link == NULL)
         {
                  printf("Error\n");
         }
            exit(0);
            temp = temp-> link;
            temp-> data = n;
            temp-> link = p;
         }
         return (p);
   }
   void printlist ( struct node *p )
   {
      struct node *temp;
      temp = p;
      printf("The data values in the list are\n");
      if(p!= NULL)
      {
      do
      {
                printf(%d\t",temp->data);
                temp=temp->link;
      } while (temp!= p)
   }
```

```
   else
      printf("The list is empty\n");
   }

   void main()
   {
      int n;
      int x;
      struct node *start = NULL ;
      printf("Enter the nodes to be created \n");
      scanf("%d",&n);
      while ( n- > 0 )
      {
               printf( "Enter the data values to be placed in a
node\n");
         scanf("%d",&x);
         start = insert ( start, x );
      }
      printf("The created list is\n");
      printlist ( start );
   }
```

DOUBLY LINKED LISTS

Introduction

The following are problems with singly linked lists:

1. A singly linked list allows traversal of the list in only one direction.
2. Deleting a node from a list requires keeping track of the previous node, that is, the node whose link points to the node to be deleted.
3. If the link in any node gets corrupted, the remaining nodes of the list become unusable.

These problems of singly linked lists can be overcome by adding one more link to each node, which points to the previous node. When such a link is added to every node of a list, the corresponding linked list is called a doubly linked list. Therefore, a doubly linked list is a linked list in which every node contains two links, called left link and right link, respectively. The left link of the node points to the previous node, whereas the right points to the next node. Like a singly linked list, a doubly linked list can also be a chain or it may be circular with or without a header node. If it is a chain, the left link of the first node and the right link of the last node will be NULL, as shown in Figure 20.18.

Figure 20.18: A doubly linked list maintained as chain.

If it is a circular list without a header node, the right link of the last node points to the first node. The left link of the first node points to the last node, as shown in Figure 20.19.

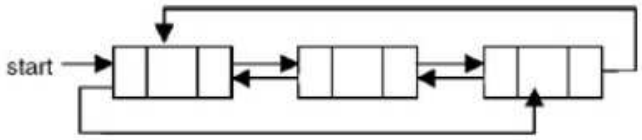


Figure 20.19: A doubly linked list maintained as a circular list.

If it is a circular list with a header node, the left link of the first node and the right link of the last node point to the header node. The right link of the header node points to the first node and the left link of the header node points to the last node of the list, as shown in Figure 20.20.

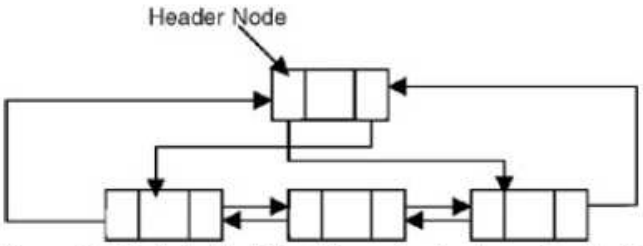


Figure 20.20: A doubly linked list maintained as a circular list with a header node.

Therefore, the following representation is required to be used for the nodes of a doubly linked list.

```
struct dnode
   {
      int data;
      struct dnode *left,*right;
   };
```

Program

A program for building and printing the elements of a doubly linked list follows:

```
   # include <stdio.h>
   # include <stdlib.h>
   struct dnode
   {
   int data;
   struct dnode *left, *right;
   };
   struct dnode *insert(struct dnode *p, struct dnode **q, int n)
   {
   struct dnode *temp;
   /* if the existing list is empty then insert a new node as the
starting node */
   if(p==NULL)
     {
         p=(struct dnode *)malloc(sizeof(struct dnode)); /* creates new
node data value
      passed as parameter */
```

```
         if(p==NULL)
         {
      printf("Error\n");
            exit(0);
         }
         p->data = n;
         p-> left = p->right =NULL;
         *q =p;
      }
      else
      {
         temp = (struct dnode *)malloc(sizeof(struct dnode)); /* creates
new node using
         data value passed as
           parameter and puts its
         address in the temp
   */
       if(temp == NULL)
       {
     printf("Error\n");
           exit(0);
       }
       temp->data = n;
       temp->left = (*q);
       temp->right = NULL;
       (*q)->right = temp;
       (*q) = temp;
    }
      return (p);
   }
   void printfor( struct dnode *p )
   {
      printf("The data values in the list in the forward order are:\n");
      while (p!= NULL)
      {
         printf("%d\t",p-> data);
         p = p->right;
      }
   }
   void printrev( struct dnode *p )
   {
   printf("The data values in the list in the reverse order are:\n");
   while (p!= NULL)
      {
         printf("%d\t",p->data);
         p = p->left;
      }
   }
   void main()
   {
         int n;
         int x;
         struct dnode *start = NULL ;
         struct dnode *end = NULL;
         printf("Enter the nodes to be created \n");
         scanf("%d",&n);
         while ( n-- > 0 )
```

```
        {
      printf( "Enter the data values to be placed in a node\n");
           scanf("%d",&x);
           start = insert ( start, &end,x );
        }
        printf("The created list is\n");
        printfor ( start );
        printrev(end);
   }
```

INSERTION OF A NODE IN A DOUBLY LINKED LIST

Introduction

The following program inserts the data in a doubly linked list.

Program

```
   # include <stdio.h>
   # include <stdlib.h>
   struct dnode
   {
   int data;
   struct node *left, *right;
   };
   struct dnode *insert(struct dnode *p, struct dnode **q, int n)
   {
   struct dnode *temp;
   /* if the existing list is empty then insert a new node as the
starting node */
   if(p==NULL)
     {
        p=(struct dnode *)malloc(sizeof(struct dnode)); /* creates new
node data value
      passed as parameter */

        if(p==NULL)
        {
      printf("Error\n");
            exit(0);
        }
        p-> data = n;
        p-> left = p->right =NULL;
        *q =p
      }
      else
      {
        temp = (struct dnode *)malloc(sizeof(struct dnode)); /* creates
new node using
        data value passed as
        parameter and puts its
       address in the temp
   */
      if(temp == NULL)
      {
```

```
   printf("Error\n");
          exit(0);
      }
      temp-> data = n;
      temp->left = (*q);
      temp->right = NULL;
      (*q) = temp;
    }
      return (p);
   }
   void printfor( struct dnode *p )
   {
      printf("The data values in the list in the forward order are:\n");
      while (p!= NULL)
         {
            printf("%d\t",p-> data);
            p = p-> right;
         }
   }
   /* A function to count the number of nodes in a doubly linked list */
   int nodecount (struct dnode *p )
   {
      int count=0;
      while (p != NULL)
      {
         count ++;
         p = p->right;
      }
         return(count);
   }

   /* a function which inserts a newly created node after the specified
node in a doubly
   linked list */
struct node * newinsert ( struct dnode *p, int node_no, int value )
      {
      struct dnode *temp, * temp1;
      int i;
      if ( node_no <= 0 || node_no > nodecount (p))
      {
      printf("Error! the specified node does not exist\n");
      exit(0);
      }
      if ( node_no == 0)
      {
      temp = ( struct dnode * )malloc ( sizeof ( struct dnode ));
      if ( temp == NULL )
      {
      printf( " Cannot allocate \n");
      exit (0);
      }
      temp -> data = value;
      temp -> right = p;
      temp->left = NULL
      p = temp ;
      }
      else
```

```
        {
        temp = p ;
        i = 1;
        while ( i < node_no )
        {
        i = i+1;
        temp = temp-> right ;
        }
        temp1 = ( struct dnode * )malloc ( sizeof(struct dnode));
        if ( temp == NULL )
        {
        printf("Cannot allocate \n");
        exit(0);
        }
        temp1 -> data = value ;
        temp1 -> right = temp -> right;
        temp1 -> left = temp;
        temp1->right->left = temp1;
        temp1->left->right = temp1
        }
        return (p);
    }
    void main()
    {
          int n;
          int x;
          struct dnode *start = NULL ;
          struct dnode *end = NULL;
          printf("Enter the nodes to be created \n");
          scanf("%d",&n);
          while ( n- > 0 )
          {
       printf( "Enter the data values to be placed in a node\n");
             scanf("%d",&x);
             start = insert ( start, &end,x );
          }
          printf("The created list is\n");
          printfor ( start );
          printf("enter the node number after which the new node is to be
inserted\n");
          scanf("%d",&n);
          printf("enter the data value to be placed in the new node\n");
          scanf("%d",&x);
          start=newinsert(start,n,x);
          printfor(start);
    }
```

DELETING A NODE FROM A DOUBLY LINKED LIST

Introduction

The following program deletes a specific node from the linked list.

Program

```
   # include <stdio.h>
   # include <stdlib.h>
   struct dnode
   {
   int data;
   struct dnode *left, *right;
   };
   struct dnode *insert(struct dnode *p, struct dnode **q, int n)
   {
   struct dnode *temp;
   /* if the existing list is empty then insert a new node as the
starting node */
    if(p==NULL)
      {
         p=(struct dnode *)malloc(sizeof(struct dnode)); /* creates new node
data value
      passed as parameter */

         if(p==NULL)
         {
      printf("Error\n");
            exit(0);
         }
         p-> data = n;
         p-> left = p->right =NULL;
         *q =p;
      }
      else
      {
         temp = (struct dnode *)malloc(sizeof(struct dnode)); /* creates
new node using
         data value passed as
           parameter and puts its
         address in the temp
   */
        if(temp == NULL)
        {
    printf("Error\n");
         exit(0);
      }
      temp-> data = n;
      temp->left = (*q);
      temp->right = NULL;
      (*q)->right = temp;
```

```
      (*q) = temp;
    }
      return (p);
   }
   void printfor( struct dnode *p )
   {
      printf("The data values in the list in the forward order are:\n");
      while (p!= NULL)
         {
            printf("%d\t",p-> data);
            p = p-> right;
         }
   }
   /* A function to count the number of nodes in a doubly linked list */
   int nodecount (struct dnode *p )
   {
      int count=0;
      while (p != NULL)
      {
         count ++;
         p = p->right;
      }
        return(count);
   }

   /* a function which inserts a newly created node after the specified
node in a doubly
          linked list */
   struct dnode * delete( struct dnode *p, int node_no, int *val)
   {
   struct dnode *temp ,*prev=NULL;
   int i;
   if ( node_no <= 0 || node_no > nodecount (p))
   {
   printf("Error! the specified node does not exist\n");
   exit(0);
   }
   if ( node_no == 0)
   {
    temp = p;
    p = temp->right;
    p->left = NULL;
    *val = temp->data;
    return(p);
   }
   else
   {
   temp = p ;
    i = 1;
   while ( i < node_no )
   {
   i = i+1;
    prev = temp;
   temp = temp-> right ;
   }
    prev->right = temp->right;
    if(temp->right != NULL)
```

```
      temp->right->left = prev;
     *val = temp->data;
     free(temp);
    }
       return (p);
    }

    void main()
    {
          int n;
          int x;
          struct dnode *start = NULL ;
          struct dnode *end = NULL;
          printf("Enter the nodes to be created \n");
          scanf("%d",&n);
          while ( n-- > 0 )
          {
       printf( "Enter the data values to be placed in a node\n");
            scanf("%d",&x);
            start = insert ( start, &end,x );
          }
          printf("The created list is\n");
          printfor ( start );
          printf("enter the number of the node which is to be deleted\n");
          scanf("%d",&n);
          start=delete(start,n,&x);
          printf("The data value of the node deleted from list is :
%d\n",x);
          printf("The list after deletion of the specified node is :\n");
          printfor(start);
    }
```