Linked lists

We have already encountered linked lists before. We have used linked lists in the ADT stacks and queues. However, insertions and deletions follow specific rules in stacks and queues. For example, a new node can be added only at the back of a queue and deleted only from the front. Here, we study the details of a general linked list with reference to the following operations:

- Creating a list
- Printing a list
- Sorting a list
- Counting the elements
- Inserting an element
- Deleting an element

Each node of a list may have a native data type or a complicated structure. Here, we construct a list where each node represents a term of a real polynomial. Thus, each node consists of a real number representing the coefficient, an integer representing the exponent, and a self-referential pointer:

```
typedef struct
{
 double coeff;
 int expo;
} data;
typedef struct node
{
 data d;
 struct node *next;
} node;
typedef node *POLY;
```

Hence, if we write `POLY Head`, then `Head` holds the address of a node. In case of an empty list, we assign `NULL` to `Head`. The functions in a linked list may be written with recursion or without recursion. Here, we write the functions without recursion.

Creating a list: Suppose the file "poly.dat" contains the coefficients and the exponents of a polynomial. The value `-1` of the exponent implies the end of the polynomial. Execution of `cat poly.dat` produces the following output:

```
2.0 11
-4.2 0
-3.5 7
-1.0 2
0.7 21
8.0 5
1.0 3
0 -1
```

The following code creates a polynomial list. The function `pol_cr()` takes the file "poly.dat" as an argument and returns the head (address of the first node) of the list. The code is self-explanatory.

```
POLY pol_cr(char *filename)
{
  int e;
  double c;
  FILE *inpol;
  POLY p,q;            // p points to 1st node, q auxillary pointer
  inpol=fopen(filename,"r");
  if(inpol==NULL)
  {
   printf("Error in opening %s\n",filename);
   exit(1);
  }
    fscanf(inpol,"%lf%d",&c,&e); // read the 1st line
    if(e==-1)    // non-existent polynomial
    {
     return NULL;
    }
    p=(POLY)calloc(1,sizeof(node));      // create 1st node
    q=p;     // q for traversing
    q->d.coeff=c;
    q->d.expo=e;
    fscanf(inpol,"%lf%d",&c,&e);         // read the next line
    while(e !=-1)    // valid term
    {
     q->next=(POLY)calloc(1,sizeof(node)); // create next node
     q=q->next;
     q->d.coeff=c;
```

```
     q->d.expo=e;
     fscanf(inpol,"%lf%d",&c,&e); // read the next line
    }
        q->next=NULL;       // last node points to NULL
   return p;                // return the address of the 1st node
}
```

Printing a list: Once a list is created, we next print it. The function `prnt_pol()` takes the address of the first node and prints the polynomial. Again, the code is self-explanatory.

```
void prnt_pol(POLY p)   // p address of the 1st node
{
 if(p != NULL)     // polynomial contains at least one term
 {
  if(p->d.expo !=0)  // term is not constant
  {
   printf("%0.2lf x^%d",p->d.coeff,p->d.expo);
  }
  else               // constant term
  {
   printf("%0.2lf",p->d.coeff);
  }
 }
 else                   // zero polynomial
 {
  printf("Zero polynomial\n");
  return;
 }
 p=p->next;
 while(p !=NULL)     // more than one term
 {
  if(p->d.expo !=0)
  {
   printf("%+0.2lf x^%d",p->d.coeff,p->d.expo);
  }
  else
  {
   printf("%+0.2lf",p->d.coeff);
  }
  p=p->next;
```

```
 }
 printf("\n");
}
```

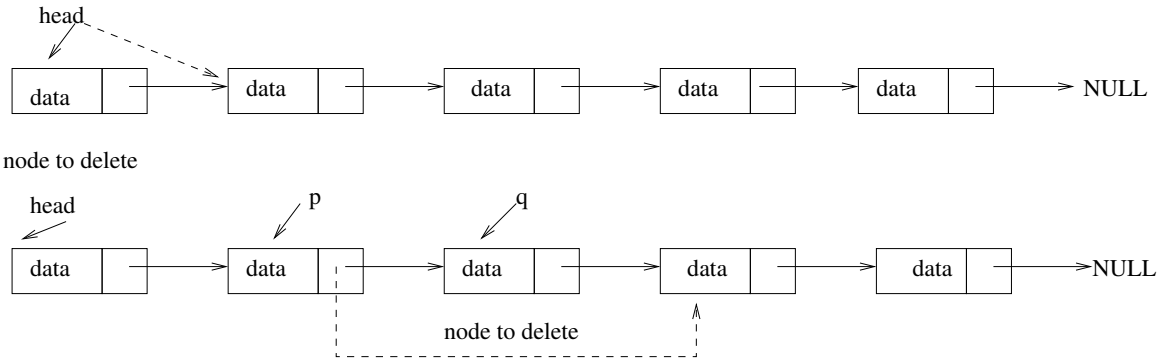
Sorting terms of polynomial: The terms of the polynomial in the file "poly.dat" are not in sorted order. The following code implements the sorting of the list using the bubble-sort technique.

```
void sort_pol(POLY p)
{
 double c;
 int e;
 POLY q;

 if(p==NULL)
 {
  return;
 }

 // follows bubble-sort used for the array earlier
 while(p->next !=NULL)        // at least two nodes
 {
  q=p->next;
  while(q!=NULL)
  {
   if(p->d.expo < q->d.expo)
  {
   e=p->d.expo;
   c=p->d.coeff;
   p->d.expo=q->d.expo;
   p->d.coeff=q->d.coeff;
   q->d.expo=e;
   q->d.coeff=c;
  }
  q=q->next;
  }
  p=p->next;
 }
}
```

Counting terms of polynomial: The function `count_pol()` returns the number of elements in the polynomial list. It returns zero if the list is empty.

```
int count_pol(POLY p)
{
int cnt=0;
while(p != NULL)
{
cnt++;
p=p->next;
}
return cnt;
}
```

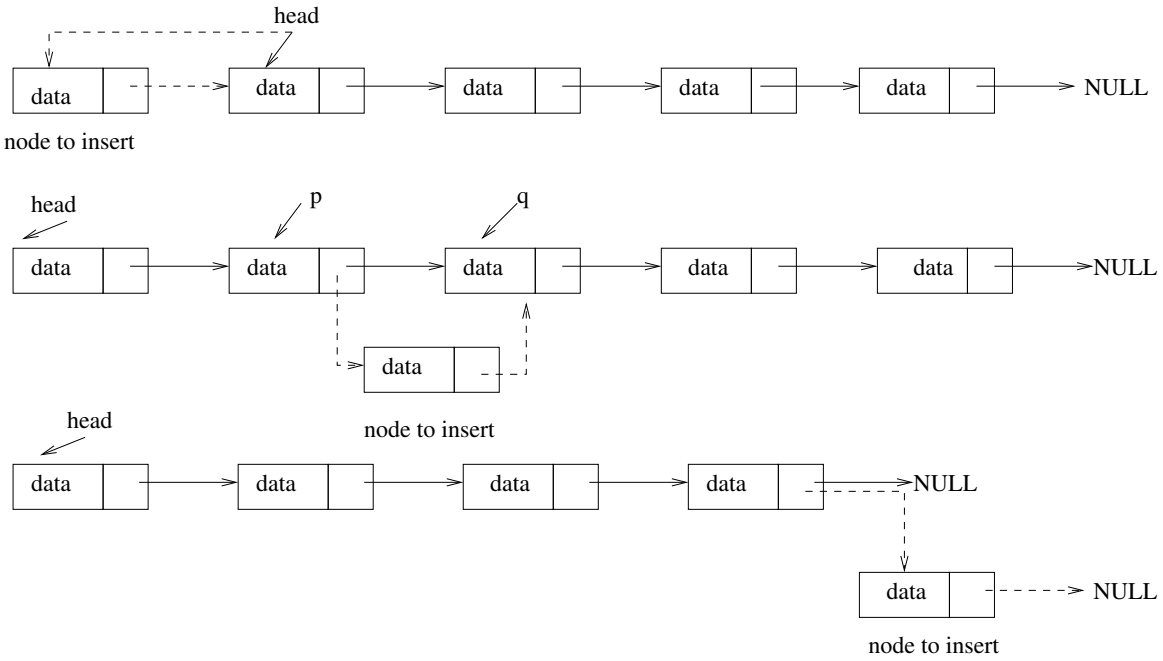
Deletion: Here, we read an exponent and delete the corresponding term. If a term with the given exponent does not exist, then it prints an appropriate message. The deletion operation varies depending on whether the deleted node is at the head of the list or not. A pictorial representation of the deletion operation is shown in the diagram.




```
POLY del_pol(POLY hd,int e)
{
 POLY p,q;

 if(hd->d.expo == e) // head node to be deleted
 {
  p=hd;
  hd=hd->next;
  free(p);
  return hd;
 }
p=hd;                // non-head node to be deleted
q=p->next;
```

```
while(q != NULL)
{

if(q->d.expo == e)
{
 p->next=q->next;
 free(q);
 return hd;
}
p=q;
q=p->next;
}
if(q==NULL)
{
printf("A term with the given exponent does not exist: no change\n");
}
return hd;
}
```

Insertion: First, we read a term of the polynomial consisting of a coefficient and an exponent. Then it is added at an appropriate place in the sorted polynomial. If a term with the given exponent exists, then it prints an appropriate message. The insertion operation varies depending on whether the inserted node is at the head, in the interior, or at the end of the list. A pictorial representation of the insertion operation is shown in the diagram.



```
POLY insrt_pol(POLY hd,double c,int e)
{
 POLY p,q;


 if(hd->d.expo < e)  // node to be inserted is head
 {
  p=(POLY)calloc(1,sizeof(node));
  p->d.coeff=c;
  p->d.expo=e;
  p->next=hd;
  hd=p;
  return hd;
 }


if(hd->d.expo == e)  // exponent of node and head same
{
 printf("A term with the given exponent already exists: no change\n");
 return hd;
}

p=hd;
q=p->next;

while(q != NULL)
{

if(q->d.expo < e)       // node to be inserted in the interior
{
 p->next=(POLY)calloc(1,sizeof(node));
 p=p->next;
 p->d.coeff=c;
 p->d.expo=e;
 p->next=q;
 return hd;
}

if(q->d.expo == e)
```

```
{
 printf("A term with the given degree already exists: no change\n");
 return hd;
}

p=q;
q=p->next;
}

 p->next=(POLY)calloc(1,sizeof(node)); // node to be inserted at the end
 p=p->next;
 p->d.coeff=c;
 p->d.expo=e;
 p->next=NULL;

 return hd;
}
```

Using all the functions discussed above, we write a program that deals with a polynomial list. The program is self-explanatory.

```
#include <stdio.h>
#include <stdlib.h>
typedef struct
{
 double coeff;
 int expo;
} data;
typedef struct node
{
 data d;
 struct node *next;
} node;
typedef node *POLY;
POLY pol_cr(char *);
void prnt_pol(POLY);
void sort_pol(POLY);
int count_pol(POLY);
POLY del_pol(POLY,int);
```

```
POLY insrt_pol(POLY,double,int);
int main()
{
 POLY head;
 int e,ch;
 double c;

 head=pol_cr("poly.dat"); // create a list for polynomial
 printf("The polynomial read is:\n");
 prnt_pol(head);          // print polynomial
 sort_pol(head);
 printf("The sorted polynomial is:\n");
 prnt_pol(head);
 printf("Enter 2 to insert, 3 to delete a term into the sorted polynomial:\n");
 printf("Enter 1 to print number of terms and 0 to finish:\n");
 scanf("%d",&ch);
 while(ch !=0)
 {
  if(ch==1)
  {
   printf("Number of terms:%d\n",count_pol(head));
  }
  else if(ch==2)
  {
   printf("Enter the coeff and exponet of the term to be inserted:");
   scanf("%lf%d",&c,&e);
   head=insrt_pol(head,c,e);
   printf("The polynomial after insertion is:\n");
   prnt_pol(head);
  }
  else if(ch==3)
  {
   printf("Enter the exponet of the term to be deleted:");
   scanf("%d",&e);
   head=del_pol(head,e);
   printf("The polynomial after deletion is:\n");
   prnt_pol(head);
  }
  else
```

```
  {
   printf("The polynomial remains unchanged\n");
   printf("Wrong choice:\n");
  }
 printf("Enter 2 to insert, 3 to delete a term into the sorted polynomial:\n");
 printf("Enter 1 to print number of terms and 0 to finish:\n");
 scanf("%d",&ch);
 }

 return 0;
}
POLY insrt_pol(POLY hd,double c,int e)
{
 POLY p,q;
 if(hd->d.expo < e)
 {
  p=(POLY)calloc(1,sizeof(node));
  p->d.coeff=c;
  p->d.expo=e;
  p->next=hd;
  hd=p;
  return hd;
 }
if(hd->d.expo == e)
{
 printf("A term with the given exponent already exists: no change\n");
 return hd;
}
p=hd;
q=p->next;
while(q != NULL)
{
if(q->d.expo < e)
{
 p->next=(POLY)calloc(1,sizeof(node));
 p=p->next;
 p->d.coeff=c;
 p->d.expo=e;
 p->next=q;
```

```
 return hd;
}
if(q->d.expo == e)
{
 printf("A term with the given degree already exists: no change\n");
 return hd;
}
p=q;
q=p->next;
}
 p->next=(POLY)calloc(1,sizeof(node));
 p=p->next;
 p->d.coeff=c;
 p->d.expo=e;
 p->next=NULL;
 return hd;
}
POLY del_pol(POLY hd,int e)
{
 POLY p,q;
 if(hd->d.expo == e)
 {
  p=hd;
  hd=hd->next;
  free(p);
  return hd;
 }
p=hd;
q=p->next;
while(q != NULL)
{
if(q->d.expo == e)
{
 p->next=q->next;
 free(q);
 return hd;
}
p=q;
q=p->next;
```

```
}
if(q==NULL)
{
printf("A term with the given exponent does not exist: no change\n");
}
return hd;
}
int count_pol(POLY p)
{
int cnt=0;
while(p != NULL)
{
cnt++;
p=p->next;
}
return cnt;
}
void sort_pol(POLY p)
{
 double c;
 int e;
 POLY q;
 if(p==NULL)
 {
  return;
 }
 // follows bubble-sort used for array earlier
 while(p->next !=NULL)        // at least two nodes
 {
  q=p->next;
  while(q!=NULL)
  {
   if(p->d.expo < q->d.expo)
  {
   e=p->d.expo;
   c=p->d.coeff;
   p->d.expo=q->d.expo;
   p->d.coeff=q->d.coeff;
   q->d.expo=e;
```

```
   q->d.coeff=c;
  }
  q=q->next;
  }
  p=p->next;
 }
}
void prnt_pol(POLY p)
{
 if(p != NULL)
 {
  if(p->d.expo !=0)
  {
   printf("%0.2lf x^%d",p->d.coeff,p->d.expo);
  }
  else
  {
   printf("%0.2lf",p->d.coeff);
  }
 }
 else
 {
  printf("Zero polynomial\n");
  return;
 }
 p=p->next;
 while(p !=NULL)
 {
  if(p->d.expo !=0)
  {
   printf("%+0.2lf x^%d",p->d.coeff,p->d.expo);
  }
  else
  {
   printf("%+0.2lf",p->d.coeff);
  }
  p=p->next;
 }
 printf("\n");
```

```
}

POLY pol_cr(char *filename)
{
  int e;
  double c;
  FILE *inpol;
  POLY p,q;            // p points to 1st node, q auxillary pointer
  inpol=fopen(filename,"r");
  if(inpol==NULL)
  {
   printf("Error in opening %s\n",filename);
   exit(1);
  }
    fscanf(inpol,"%lf%d",&c,&e); // read the 1st line
    if(e==-1)    // non-existent polynomial
    {
     return NULL;
    }
    p=(POLY)calloc(1,sizeof(node));      // create 1st node
    q=p;     // q for traversing
    q->d.coeff=c;
    q->d.expo=e;
    fscanf(inpol,"%lf%d",&c,&e);         // read the next line
    while(e !=-1)    // valid term
    {
     q->next=(POLY)calloc(1,sizeof(node));
     q=q->next;
     q->d.coeff=c;
      q->d.expo=e;
     fscanf(inpol,"%lf%d",&c,&e); // read the next line
    }
        q->next=NULL;       // last node points to NULL
   return p;
}
```

Listing 79: C Program "linkpoly.c"

Below is a sample input-output of the program.

```
The polynomial read is:
2.00 x^11-4.20-3.50 x^7-1.00 x^2+0.70 x^21+8.00 x^5+1.00 x^3
The sorted polynomial is:
0.70 x^21+2.00 x^11-3.50 x^7+8.00 x^5+1.00 x^3-1.00 x^2-4.20
Enter 2 to insert, 3 to delete a term into the sorted polynomial:
Enter 1 to print number of terms and 0 to finish:
1
Number of terms:7
Enter 2 to insert, 3 to delete a term into the sorted polynomial:
Enter 1 to print number of terms and 0 to finish:
2
Enter the coeff and exponet of the term to be inserted:4 18
The polynomial after insertion is:
0.70 x^21+4.00 x^18+2.00 x^11-3.50 x^7+8.00 x^5+1.00 x^3-1.00 x^2-4.20
Enter 2 to insert, 3 to delete a term into the sorted polynomial:
Enter 1 to print number of terms and 0 to finish:
3
Enter the exponet of the term to be deleted:5
The polynomial after deletion is:
0.70 x^21+4.00 x^18+2.00 x^11-3.50 x^7+1.00 x^3-1.00 x^2-4.20
Enter 2 to insert, 3 to delete a term into the sorted polynomial:
Enter 1 to print number of terms and 0 to finish:
3
Enter the exponet of the term to be deleted:6
A term with the given exponent does not exist: no change
The polynomial after deletion is:
0.70 x^21+4.00 x^18+2.00 x^11-3.50 x^7+1.00 x^3-1.00 x^2-4.20
Enter 2 to insert, 3 to delete a term into the sorted polynomial:
Enter 1 to print number of terms and 0 to finish:
0
```

Return to Lecture Topics