Application of linked lists

Stacks and Queues , Polynomial handling

Inserting an element in a sorted linked list.

Let the data be sorted and put in a singly linked linear list which is being pointed by address “head “.
Let the new data to be entered be “d”.
Use malloc get a node with address “pNew”.
Suppose we want to write a code to enter data “d” into the node and insert it into its proper place in the list.

```
typedef struct node {
     int data;
     struct node  *next;
   };

struct node* pNew = (struct node*)
(malloc(sizeof(struct node)));

pNew -> data = d;
pNew ->next = NULL;
pCur = head ;

/* check if data is smaller than smallest item on the list*/
if (pNew -> data  <   pCur -> data )
 {
      pNew ->next = pCur ;
      head = pNew;
  }

/* now examine the remaining list */
```

```
p = pCur -> next ;

while(p!=NULL ||p->data <  pNew->data )
 {
       pCur = pCur -> next ;
      p = p -> next ;
 }

pNew -> next = pCur -> next;
pCur -> next = pNew ;
```

Implementation of stacks and queues using linked lists.

Stacks and Queues are easier to implement using linked lists. There is no need to set a limit on the size of the stack. Let us develop push, pop and enqueue and dequeue functions for nodes containing integer data.

```
typedef struct node {
     int data;
     struct node  *next;
  };
```

The Push function places the data in a new node, attaches that node to the front of stacktop.

```
void push(struct node**stacktop,int d )
{
struct node* pNew = (struct node*)
(malloc(sizeof(struct node)));

       pNew-> data =  d ;
       pNew->next  =  *stacktop;
      *stacktop  =  pNew ;
}
```

Pop Function returns the data from top element, and frees that element from the stack.

```
int pop(struct node* *stacktop)
{
  struct node* temp;

 if(*stacktop== NULL)
   {
       printf(“\nstack empty\n”);
    }
  else
   {
      temp = *stacktop;
      d  =  temp->data;
      *stacktop  =  temp->next ;
      free (temp);
   }
 return d;
}
```

For Queue operations, we can maintain two pointers – qfront and qback as we had done for the case of array implementation of queues.

For the Enqueue operation, the data is first loaded on a new node.
If the queue is empty, then after insertion of the first node, both qfront and qback are made to point to this node, otherwise, the new node is simply appended and qback updated.

```
void enqueue(struct node**qfront,int d
, struct node**qback)
{

  struct node* pNew = (struct node*)
(malloc(sizeof(struct node)));

 pNew-> data =  d ;
 pNew->next  = NULL;

 if (*qfront ==NULL && *qback == NULL)
      {
        *qfront = pNew;
        *qback  = pNew;
      }
 else
      {
        *qback->next = pNew;
        *qback = pNew;
      }

 }
```

In Dequeue function, first of all check if at all there is any element .
If there is none, we would have *qfront as NULL, and so report queue to be empty,
otherwise return the data element, update the *qfront pointer and free the node. Special care has to be taken if it was the only node in the queue.

```
int dequeue(struct node**qfront, struct
node**qback)
{
  struct node* temp;

  if (*qfront== NULL)
     printf("\nqueue is empty\n");

  else
    {
        temp = *qfront;
        d = temp->data;
        *qfront = *qfront->next;
        if (*qfront == NULL)
             *qback = NULL;
        free (temp);


    }
  return d;
}
```

Representing a polynomial using a linked list

A polynomial can be represented in an array or in a linked list by simply storing the coefficient and exponent of each term.
However, for any polynomial operation , such as addition or multiplication of polynomials , you will find that the linked list representation is more easier to deal with.
First of all note that in a polynomial all the terms may not be present, especially if it is going to be a very high order polynomial. Consider

$5\ x^{12} + 2\ x^{9} + 4x^{7} + 6x^{5} + x^{2} + 12\ x$

Now this 12^{th} order polynomial does not have all the 13 terms (including the constant term).
It would be very easy to represent the polynomial using a linked list structure, where each node can hold information pertaining to a single term of the polynomial.

Each node will need to store
the variable x,
the exponent and
the coefficient for each term.

It often does not matter whether the polynomial is in x or y. This information may not be very crucial for the intended operations on the polynomial.

Thus we need to define a node structure to hold two integers , viz. exp and coff

Compare this representation with storing the same polynomial using an array structure.
In the array we have to have keep a slot for each exponent of x,
thus if we have a polynomial of order 50 but containing just 6 terms, then a large number of entries will be zero in the array.

You will also see that it would be also easy to manipulate a pair of polynomials if they are represented using linked lists.

Addition of two polynomials

Consider addition of the following polynomials

$5\ x^{12} + 2\ x^{9} + 4x^{7} + 6x^{6} + x^{3}$

$7\ x^{8} + 2\ x^{7} + 8x^{6} + 6x^{4} + 2x^{2} + 3\ x + 40$

The resulting polynomial is going to be

$5\ x^{12} + 2\ x^{9} + 7\ x^{8} + 6\ x^{7} + 14x^{6} + 6x^{4} + x^{3}$

$2x^2 + 3x + 40$

Now notice how the addition was carried out. Let us say the result of addition is going to be stored in a third list.
We started with the highest power in any polynomial.

If there was no item having same exponent , we simply appended the term to the new list, and continued with the process.

Wherever we found that the exponents were matching, we simply added the coefficients and then stored the term in the new list.

If one list gets exhausted earlier and the other list still contains some lower order terms, then simply append the remaining terms to the new list.

Now we are in a position to write our algorithm for adding two polynomials.

Let phead1 , phead2 and phead3 represent the pointers of the three lists under consideration.

Let each node contain two integers exp and coff .

Let us assume that the two linked lists already contain relevant data about the two polynomials.

Also assume that we have got a function **append** to insert a new node at the end of the given list.

```
p1 = phead1;
p2 = phead2;
```

Let us call malloc to create a new node p3 to build the third list

```
p3 = phead3;
```

/* now traverse the lists till one list gets exhausted */

```
while ((p1 != NULL) || (p2 != NULL))
{
```

/ * if the exponent of p1 is higher than that of p2 then the next term in final list is going to be the node of p1* /

```
while (p1 ->exp   >  p2 -> exp )
   {
        p3 -> exp  = p1 -> exp;
        p3 -> coff = p1 -> coff ;
        append (p3, phead3);
```

/* now move to the next term in list 1*/

```
        p1 = p1 -> next;
    }
```

/ * if p2 exponent turns out to be higher then make p3 same as p2 and append to final list * /

```
while (p1 ->exp   <  p2 -> exp )
    {
         p3 -> exp  = p2 -> exp;
         p3 -> coff = p2 -> coff ;
         append (p3, phead3);
         p2 = p2 -> next;
     }
```

/* now consider the possibility that both exponents are same , then we must add the coefficients to get the term for the final list */

```
while (p1 ->exp   =  p2 -> exp )
     {
       p3-> exp = p1-> exp;
       p3->coff = p1->coff + p2-> coff ;
       append (p3, phead3) ;
       p1 = p1->next ;
       p2 = p2->next ;
      }
}
```

/* now consider the possibility that list2 gets exhausted , and there are terms remaining only in list1. So all those terms have to be appended to end of list3. However, you do not have to do it term by term, as p1 is already pointing to remaining terms, so simply append the pointer p1 to phead3 */

```
if ( p1 != NULL)
```

```
        append (p1, phead3) ;
else
        append (p2, phead3);
```

Now, you can implement the algorithm in C, and maybe make it more efficient.