Linked Lists – III

Stacks and Queues
Representing Polynomials
Circularly linked lists

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 or the queue. A linked list can serve as a stack, if the pointer to the first node is referred to as *stacktop*. To implement a push function, do malloc and put the new data in node *pNew*. Now add this node at the front of the list, and rename it as *stacktop.* Let us develop this function using a double pointer. (You can also do it using a single pointer and returning *stacktop*). In the main program you would first need to define a stack variable of the type struct node.

```
struct node* stack1;
int num;
stack1 = (struct node*) (malloc(sizeof(struct node)));
stack1 = NULL;
. . . . . .
. . . . . .

push( &stack1, num);
```

The Push function places the data num 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 ;
}
```

Similarly you can develop the pop function. Remember that you need to print an error message if the stack is empty. You should also free the first node of the stack and return it to memory.

Queues

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

For the enqueue operation, a new node is requested using malloc and the given data is stored in its data field.
If the queue is empty and a node is being added to it, both qfront and qRear are made to point to this node. If the queue is not empty, the new node is simply appended to the end of the list and qRear updated.

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

  struct node* pNew;
  pNew = (struct node*)(malloc(sizeof(struct node)));
  pNew-> data =  d ;
  pNew->next  = NULL;

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

 }
```

For the dequeue function, first of all check if at all there is any element in the queue. If there is none, we would have *qfront as NULL, and so report the 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. Both pointers would have to be set to NULL.

Application of Linked Lists:

Representing a polynomial

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 <u>just two integers</u> , viz. exp and coff, and the link to the next node address.

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, starting with the highest and going down to the constant term. You can see that it is not an efficient representation. If we have a polynomial of order 50, with fewer terms such as $x^{50} + 12\ x^{30} + 4x^{7} + 6x^{3} + x$ 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

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

B: $7\ x^8 + 2\ x^7 + 8x^6 + 6x^4 + 2x^2 + 3\ x + 40$

The resulting polynomial is going to be

C: $5\ x^{12} + 2\ x^9 + 7\ x^8 + 6\ x^7 + 14x^6 + 6x^4 + x^3 + 2x^2 + 3\ x + 40$

The addition can be carried out by starting with the highest powered terms in both the polynomials. If the exponents do not match , add the higher powered node to the final list. In this example, first node of A has higher power compared to first node of B. so the first node of A, $5\ x^{12}$ would be added to list C. Then move to the next node of A, and compare it with the first node of B.

Wherever the exponents match, we simply add the coefficients and then store the term in C.

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.

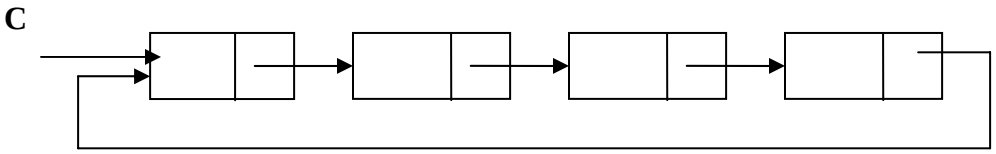
Circular lists

In some situations, a circular list is needed in which nodes form a ring. Here each node has a successor. An example of such a situation is when several processes are using the same computer resource for the same amount of time, and we have to assure that no process accesses the resource before all other processes did. *A process is may be a program in execution.* The processes are put on a circular list, and control passes to next one when any of the nodes (processes) is serviced.

A circular list is accessible through one of its nodes, usually called *the tail*. Note that unlike standard linked lists, there is no node with NULL pointer in a circularly linked list.

Counting nodes in a circular list:

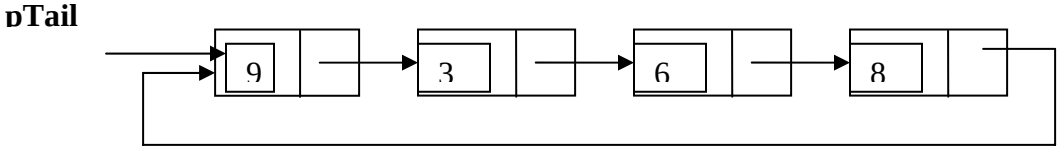
The following list is accessible through the node marked C. To count the nodes, the list has to be traversed from node marked C, with the help of a dummy node pCur. Counting can stop when pCur reaches the starting node C again. If the list is empty, C will be null, and in that case set count = 0. Otherwise, set the current pointer to the first node, and keep on counting till the current pointer reaches the starting node.




```
if(C == NULL) count = 0;
else {
     pCur = C;
     count = 1;
     while( pCur->next !=  C)
       {
          count++;
          pCur = pCur->next;
       }
     }
```

Printing the contents of a circular list:

We assume here that the list is being accessed by its last node (tail). Since all the nodes are arranged in a circular fashion, the first node of the list will be the node next to the tail node. Let us we want to print the contents of the nodes starting with the first node. Print its contents, move to the next node and continue printing till you reach the first node again. Thus a list of items 3, 6, 8, 9 would be represented by the following :




```
pCur = pTail->next;
do
{
     printf(" %d ",pCur->data);
     pCur= pCur->next;
}
while( pCur != pTail->next)
```

3 6 8 9

Inserting a node at the end of a (circular) list:

Let us add a node containing data d, at the end of a list (circular list) pointed to by pTail. The new node will be placed just after the tail node (which is the last node of the list), which means it will have to be inserted in between the tail node and the first node.

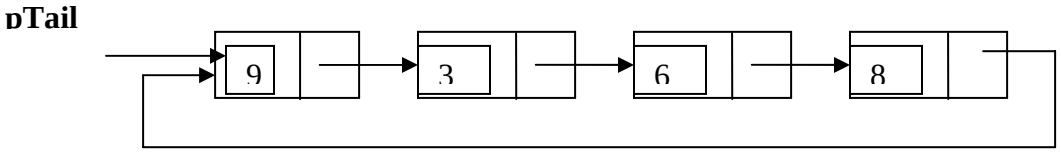
```
struct node* pNew = (struct node*)
(malloc(sizeof(struct node)));
pNew ->data = d;
pNew->next = pNew;
```

```
if (pTail==NULL)
     pTail = pNew;

else{
     pNew->next = pTail->next;
     pTail->next = pNew;
     pTail = pNew;
    }
```

What is the complexity? It is O(1). Compare this with the complexity of adding a node at the end of a standard linear linked list which is O(n), because in that case you have to traverse the complete list to reach the last node (with NULL value in next field).

Deleting the first node in a circular list:

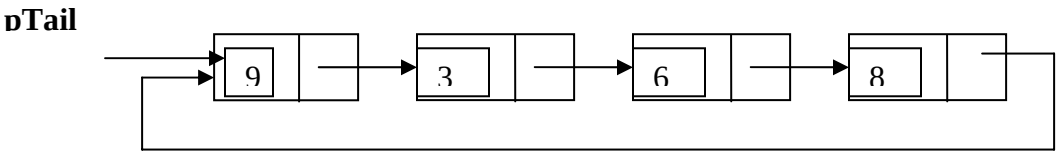


The first node can be deleted by simply replacing the next field of tail node with the next fieild of the first node.

```
temp = pTail->next;
pTail->next = temp->next;
free(temp);
```

Inserting a node at front of a circular list:

Let us add a node containing data d, at front of a circular list pointed to by pTail. The first node will be the node next to the tail node. The new node will have to be inserted just before the first node, i.e. in between the tail node and the first node. Get a new node from memory , put the data d in that node, and make it point to itself. If pTail is empty, this node will be the only node in the circular list. If it is not the first node, then its next part should contain the address which pTail was pointing to. Finally, its address must be stored in the next part of pTail.

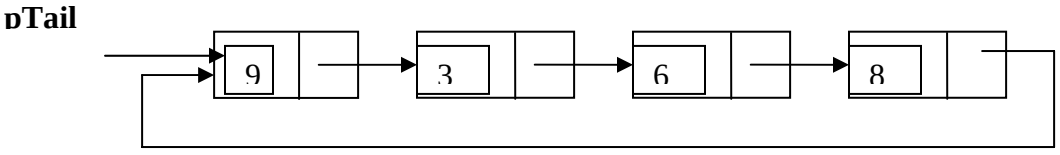



```
struct node* pNew = (struct node*) (malloc(sizeof(struct
node)));
     pNew ->data = d;
     pNew->next = pNew;
     if (pTail==NULL)
          pTail = pNew;
     else{
          pNew->next = pTail->next;
          pTail->next = pNew;
     }
```

What is the complexity of this operation? Since the complete list is not being traversed, it is O(1).

Deleting the last node in a circular list:

Deletion of last node is a more complicated case. The list has to be traversed to reach the last but one node. This has to be named as the tail node, and its next field has to point to the first node. Consider the following list. To delete the last node 9, the list has to be traversed till you reach 8. The next field of 8 has to be changed to point to 3, and this node must be renamed pTail.



This is left as an exercise for you. Write the code and work out its complexity.

Applications of circular list:

You have already seen an application of circular list in managing the computing resources of a computer. You can also use circular lists for implementing stacks and queues. There is an interesting problem from history known as the Josephus problem which can be solved using a circular linked list.

The Josephus Problem is a classic problem in combinatorics. According to legend, Flavius Josephus, a famous Jewish historian, survived the Jewish-Roman war due to his quick thinking and mathematical talent. 39 Jewish rebels, including Josephus, were trapped by the Romans. His companions preferred suicide to surrender and devised the following scheme: all the rebels would stand in a circle and, proceeding clockwise, they would kill every seventh standing rebel until only one was left (who was then expected to kill himself).

Josephus did not find suicide appealing and carefully calculated at which position he must stand in order to be the last one left alive. He therefore survived the vicious circle. You may just like to see what would have happened if he was standing at the 17^{th} position.

If you find that tedious to work out by hand try the following simpler problem:

Given a number *n*, the ordering of the men in the circle, and the man from whom the count begins, to determine the order in which the men are eliminated from the circle and which man escapes.

For example, suppose that *n* equals 3 and there are five men named A, B, C, D, and E. We count three men, starting at A, so that C is eliminated first. We then begin at D and count D, E, and back to A, so that A is eliminated next. Then we count B, D, and E (C has already been eliminated) and finally B, D, and B, so that D is the man who escapes.