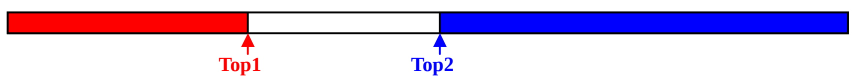
p.88   3.21

Write routines to implement two stacks using only one array. Your stack routines should not declare an overflow unless every slot in the array is used.

```
#ifndef   _Stack_h
#define   _Stack_h
    struct   StackRecord;
    typedef  struct    StackRecord   *Stack;
    int   IsEmpty( Stack   S,   int   Stacknum );
    int   IsFull( Stack   S );
    Stack   CreateStack( int   MaxElements );
    void   Push( ElementType   X,   Stack   S,   int   Stacknum );
    ElementType   Top( Stack   S,   int   Stacknum );
    void   Pop( Stack   S,   int   Stacknum );
#endif   /* _Stack_h */
#define   MinStackSize ( 5 )

struct   StackRecord   {
        int   Capacity;
        int   Top1;
        int   Top2;
        ElementType   *Array;
 }
/* The bottom of the first stack is at Array[0]                          */
/* The bottom of the second stack is at Array[ MaxElement ]              */
/* Top1 and Top2 point to the top elements of two stacks respectively  */
```




```
Stack CreateStack ( int   MaxElements )
{     Stack   S;
      if ( MaxElements < MinStackSize )    return Error("Stack size is too small");
      else   {
           S = malloc( sizeof ( struct   StackRecord ) );   /* S points to a stack */
           if ( S == Null )     FatalError("Out of space");
           /* create an array for stack elements */
           S->Array = malloc( sizeof ( ElementType ) * MaxElements );
           if ( S->Array == Null )     FatalError("Out of space");
           S->Capacity = MaxElements;
           S->Top1 = –1;     S->Top2 = MaxElements;    /* initialize 2 empty stacks */
           return ( S );
      }
}
```

```
int   IsEmpty ( Stack   S,  int   Stacknum )
{
      if ( Stacknum ==1)    /* check stack 1 */
           return   S->Top1 == –1;
      else   {
           if ( Stacknum == 2 )    /* check stack 2 */
               return   S->Top2 == S->Capacity;
           else
               return   Error("stack number error");
       }
}

int   IsFull( Stack   S )
{
       return   S->Top1+1 == S->Top2;
}

void   Push( ElementType   X,   Stack   S,   int   Stacknum )
{
        if ( IsFull( S ) )
             FatalError("Stack is full");
        if ( Stacknum == 1 )   /* push onto stack 1 */
            S->Array[++S->Top1] = X;
        else   {
            if ( Stacknum == 2 )   /* push onto stack 2 */
                 S->Array[++S->Top2] = X;
            else
                 Error("stack number error");
        }
}

ElementType   Top( Stack   S,   int   Stacknum )
{
       if ( IsEmpty( S, Stacknum ) )
             return   FatalError("Stack is empty");
       if ( Stacknum == 1 )   /* check stack 1 */
            return   S->Array[S->Top1];
       else   {
            if ( Stacknum == 2 )   /* check stack 2 */
                 return   S->Array[S->Top2];
            else
                 return   Error("stack number error");
       }
}
```

```
void   Pop( Stack   S,   int   Stacknum )
{
       if ( IsEmpty( S, Stacknum ) )
            FatalError("Stack is empty");
       if ( Stacknum == 1 )   /* pop stack 1 */
            S->Top1 – –;
       else   {
            if ( Stacknum == 2 )   /* pop stack 2 */
                 S->Top2 – –;
            else
                  Error("stack number error");
        }
}
```

p.88 3.26

A "deque" is a data structure consisting of a list of items, on which the following operations are possible:

    Push(X,D): Insert item X on the front end of deque D.
    Pop(D): Remove the front item from deque D and return it.
    Inject(X,D): Insert item X on the rear end of deque D.
    Eject(D): Remove the rear item from deque D and return it.

Write routines to support the deque that take O(1) time per operation.

```
#ifndef   _deque _h
#define   _deque _h
       struct   Node;
       typedef   struct   Node   *PtrToNode;
       struct   DequeRecord
       {
             PtrToNode   Front, Rear;
       };
       typedef   struct   DequeRecord   *Deque;
       void   Push( ElementType X, Deque D );
       ElementType   Pop( Deque D );
       void   Inject( ElementType X, Deque D );
       ElementType   Eject( Deque D );
#endif   /* _deque_h */
struct Node
{
      ElementType   Element;
      PtrToNode   Next, Last;
};
```

```
 /* Implement a deque using a doubly linked list with a header      */
 /* front and rear point to two ends of the deque respectively      */
 /* front always points to the header                                */
 /* deque is empty when front == rear                                */

void   Push( ElementType   X,   Deque   D )
{   /* Insert item X on the front end of deque D */
        PtrToNode   Newnode;

        Newnode = malloc( sizeof ( Node ) );   /* create a new node */
        if ( Newnode == Null )
             FatalError ("The memory is full");

        Newnode->Element = X;
        Newnode->Next = D->Front->Next;
        Newnode->Last = D->Front;   /* Newnode is all set now */

        if ( D->Front == D->Rear )   /* if deque was originally empty */
             D->Rear = Newnode;   /* reset rear */
        else   /* if deque was not empty */
             D->Front->Next->Last = Newnode;   /* reset original first element */
        D->Front->Next = Newnode;   /* reset the first element in any case */
}

ElementType   Pop( Deque   D )
{   /* Remove the front item from deque D and return it */
        PtrToNode   Temp;
        ElementType   Item;

        if   ( D->Front == D->Rear )
            return   Error(“Deque is empty”);   /* an element must be returned */
        else
        {
             Temp = D->Front->Next;   /* get the first element */
             D->Front->Next = Temp->Next;   /* reset the new first element */
             Temp->Next->Last = D->Front;   /* reset the new first element’s last link */
             if   ( Temp == D->Rear ) /* if there was only one element in deque */
                 D->Rear = D->Front; /* set deque to be empty */
             Item = Temp->Element;
             free ( Temp );
             return   Item;
         }
}
```

```
void   Inject ( ElementType   X,   Deque   D )
{    /*Insert item X on the rear end of deque D*/
        PtrToNode   Newnode;

        Newnode = malloc( sizeof ( Node ) );   /* create a new node */
        if ( Newnode == Null )
             FatalError ("The memory is full");

        Newnode->Element = X;
        Newnode->Next = Null;
        Newnode->Last = D->Rear;     /* Newnode is all set now */

        D->Rear->Next = Newnode;   /* reset original last node */
        D->Rear = Newnode;              /* reset last */
}

ElementType   Eject ( Deque   D )
{    /* Remove the rear item from deque D and return it */
        PtrToNode   Temp;
        ElementType   Item;

        if   ( D->Front == D->Rear )
            retrun   Error(“Empty Deque”);   /* an element must be returned */
        else
        {
             Temp = D->Rear;                /* get the first element */
             D->Rear = D->Rear->Last;      /* reset last */
             D->Rear->Next = Null;        /* reset the last element’s link */
             Item = Temp->Element;
             free ( Temp );
             return   Item;
         }
}
```

p.143 4.35

Write a routine to list out the nodes of a binary tree in "level-order". List the root, then nodes at depth 1, followed by nodes at depth 2, and so on. You must do this in linear time. Prove your time bound.

```
typedef struct TreeNode *PtrToNode;
typedef PtrToNode Tree;
struct TreeNode
{
        ElementType Element;
        Tree Left;
        Tree Right;
}

#define MaxElements 100 /* max queue size */


void Level_order ( Tree T )
{
   /* Level order binary tree traversal                              */
   /* 1) enqueue ( the root T )                                       */
   /* while (queue is not empty), do steps 2 and 3:                  */
   /* 2) list the node at the front of the queue and dequeue it      */
   /* 3) enqueue the node's left and right children                  */

      Queue Q;
      if ( !T ) return;      /*empty tree */
      Q = CreateQueue( MaxElements ); /* create an empty queue */
      Enqueue( T, Q ); /* enqueue the root */
      while ( ! IsEmpty( Q ) )
      {
          T = FrontAndDequeue ( Q );  /* get the front element and dequeue it */
          ListOut ( T );
          if ( T->Left )  /* if this node has a left child */
              Enqueue( T->Left, Q );
          if ( T->Right ) /* if this node has a right child */
              Enqueue( T->Right, Q );
       }
}
```

Since each node is visited exactly once, this routine runs in linear time.