stackar.h

```
typedef int ElementType;

#ifndef _Stack_h
#define _Stack_h

struct StackRecord;
typedef struct StackRecord *Stack;

int IsEmpty( Stack S );
int IsFull( Stack S );
Stack CreateStack( int MaxElements );
void DisposeStack( Stack S );
void MakeEmpty( Stack S );
void Push( ElementType X, Stack S );
ElementType Top( Stack S );
void Pop( Stack S );
ElementType TopAndPop( Stack S );

#endif  /* _Stack_h */
```

stackar.c

```
#include "stackar.h"
#include "fatal.h"
#include <stdlib.h>

#define EmptyTOS ( -1 )
#define MinStackSize ( 5 )

struct StackRecord
{
    int Capacity;
    int TopOfStack;
    ElementType *Array;
};
```

```
int IsEmpty( Stack S )
{
    return S->TopOfStack == EmptyTOS;
}

int IsFull( Stack S )
{
  return S->TopOfStack == S->Capacity - 1;
}

   Stack CreateStack( int MaxElements )
   {
     Stack S;

/* 1*/    if( MaxElements < MinStackSize )
/* 2*/        Error( "Stack size is too small"
);
/* 3*/    S = malloc( sizeof( struct
StackRecord ) );
/* 4*/    if( S == NULL )
/* 5*/        FatalError( "Out of space!!!" );

/* 6*/      S->Array = malloc( sizeof(
ElementType ) * MaxElements );
/* 7*/      if( S->Array == NULL )
/* 8*/         FatalError( "Out of space!!!" );
/* 9*/      S->Capacity = MaxElements;
/*10*/      MakeEmpty( S );

/*11*/      return S;
   }

   void MakeEmpty( Stack S )
   {
       S->TopOfStack = EmptyTOS;
   }
```

```
    void DisposeStack( Stack S )
    {
        if( S != NULL )
        {
            free( S->Array );
            free( S );
        }
    }

    void Push( ElementType X, Stack S )
    {
        if( IsFull( S ) )
            Error( "Full stack" );
        else
           S->Array[ ++S->TopOfStack ] = X;
    }

    ElementType Top( Stack S )
    {
        if( !IsEmpty( S ) )
          return S->Array[ S->TopOfStack ];
        Error( "Empty stack" );
        return 0; /* Return value used to avoid warning */
    }

    void Pop( Stack S )
    {
        if( IsEmpty( S ) )
            Error( "Empty stack" );
        else
            S->TopOfStack--;
    }
ElementType TopAndPop( Stack S )
{
     if( !IsEmpty( S ) )
     return S->Array[ S->TopOfStack-- ];
        Error( "Empty stack" );return 0;
}
```

stackli.h

```
typedef int ElementType;

#ifndef _Stack_h
#define _Stack_h

struct Node;
typedef struct Node *PtrToNode;
typedef PtrToNode Stack;

int IsEmpty( Stack S );
Stack CreateStack( void );
void DisposeStack( Stack S );
void MakeEmpty( Stack S );
void Push( ElementType X, Stack S );
ElementType Top( Stack S );
void Pop( Stack S );

#endif  /* _Stack_h */
```

stackli.c

```
#include "stackli.h"
#include "fatal.h"
#include <stdlib.h>

struct Node
{
    ElementType Element;
    PtrToNode   Next;
};

int IsEmpty( Stack S )
{
    return S->Next == NULL;
}
```

```
        Stack CreateStack( void )
        {
          Stack S;

          S = malloc( sizeof( struct Node ) );
          if( S == NULL )
              FatalError( "Out of space!!!" );
          S->Next = NULL;
          MakeEmpty( S );
          return S;
      }

void MakeEmpty( Stack S )
{
    if( S == NULL )
        Error( "Must use CreateStack first" );
    else
        while( !IsEmpty( S ) )
            Pop( S );
}

        void DisposeStack( Stack S )
        {
            MakeEmpty( S );
            free( S );
        }

void Push( ElementType X, Stack S )
{
   PtrToNode TmpCell;

    TmpCell = malloc( sizeof( struct Node ) );
    if( TmpCell == NULL )
        FatalError( "Out of space!!!" );
    else
    {
        TmpCell->Element = X;
```

```
        TmpCell->Next = S->Next;
        S->Next = TmpCell;
    }
}

    ElementType Top( Stack S )
    {
        if( !IsEmpty( S ) )
            return S->Next->Element;
        Error( "Empty stack" );
        return 0; /* Return value used to avoid warning */
    }

    void Pop( Stack S )
    {
        PtrToNode FirstCell;

        if( IsEmpty( S ) )
            Error( "Empty stack" );
        else
        {
            FirstCell = S->Next;
            S->Next = S->Next->Next;
            free( FirstCell );
        }
    }
```

queue.h

```
    typedef int ElementType;

    #ifndef _Queue_h
    #define _Queue_h

    struct QueueRecord;
    typedef struct QueueRecord *Queue;

    int IsEmpty( Queue Q );
```

```
int IsFull( Queue Q );
Queue CreateQueue( int MaxElements );
void DisposeQueue( Queue Q );
void MakeEmpty( Queue Q );
void Enqueue( ElementType X, Queue Q );
ElementType Front( Queue Q );
void Dequeue( Queue Q );
ElementType FrontAndDequeue( Queue Q );

#endif  /* _Queue_h */
```

queue.c

```
#include "queue.h"
#include "fatal.h"
#include <stdlib.h>

#define MinQueueSize ( 5 )

struct QueueRecord
{
    int Capacity;
    int Front;
    int Rear;
    int Size;
    ElementType *Array;
};

int IsEmpty( Queue Q )
{
    return Q->Size == 0;
}

int IsFull( Queue Q )
{
    return Q->Size == Q->Capacity;
}
```

```
        Queue CreateQueue( int MaxElements )
        {
            Queue Q;

/* 1*/      if( MaxElements < MinQueueSize )
/* 2*/          Error( "Queue size is too
small" );
/* 3*/      Q = malloc( sizeof( struct
QueueRecord ) );
/* 4*/      if( Q == NULL )
/* 5*/          FatalError( "Out of space!!!"
);
/* 6*/      Q->Array = malloc( sizeof(
ElementType ) * MaxElements );
/* 7*/      if( Q->Array == NULL )
/* 8*/          FatalError( "Out of space!!!"
);
/* 9*/      Q->Capacity = MaxElements;
/*10*/      MakeEmpty( Q );
/*11*/      return Q;
        }

        void MakeEmpty( Queue Q )
        {
            Q->Size = 0;
            Q->Front = 1;
            Q->Rear = 0;
        }

        void DisposeQueue( Queue Q )
        {
            if( Q != NULL )
            {
                free( Q->Array );
                free( Q );
            }
        }
```

```
static int Succ( int Value, Queue Q )
{
    if( ++Value == Q->Capacity )
        Value = 0;
    return Value;
}

void Enqueue( ElementType X, Queue Q )
{
    if( IsFull( Q ) )
        Error( "Full queue" );
    else
    {
        Q->Size++;
        Q->Rear = Succ( Q->Rear, Q );
        Q->Array[ Q->Rear ] = X;
    }
}

ElementType Front( Queue Q )
{
    if( !IsEmpty( Q ) )
        return Q->Array[ Q->Front ];
    Error( "Empty queue" );
    return 0; /* Return value used to avoid warning */
}

void Dequeue( Queue Q )
{
    if( IsEmpty( Q ) )
        Error( "Empty queue" );
    else
    {
        Q->Size--;
        Q->Front = Succ( Q->Front, Q );
    }
}
```

```
ElementType FrontAndDequeue( Queue Q )
{
    ElementType X = 0;

    if( IsEmpty( Q ) )
        Error( "Empty queue" );
    else
    {
        Q->Size--;
        X = Q->Array[ Q->Front ];
        Q->Front = Succ( Q->Front, Q );
    }
    return X;
}
```