CSCI 2170 **Lecture Notes(14)**

ADT Queue -- First In First Out (FIFO)

Operations :

create an empty queue	
Destroy a queue	
Determine whether a queue is empty	bool IsEmpty()
Add a new item to the queue (**EnQueue**)	void QueueInsert(ItemType newItem, bool &success))
Retrieve the content of the item at the front of the queue	void GetQueueFront(ItemType & queueFront, bool &success)
Remove the item at the front of the queue (**DeQueue**)	void QueueDelete(bool &success)
Remove and retrieve the item at the front of the queue (**DeQueue**)	void QueueDelete(ItemType & queueFront, bool & success))

Examples of client programs using ADT queue:

(a) convert a sequence of characters entered from keyboard into the corresponding integer value

```
Queue Q;
Read character from std input stream      // cin.get(ch);
while (not end of line);                  // while (ch != '\n')
{
    add character to queue                // Q.QueueInsert(ch, success);
    read in the next character            // cin.get(ch);
}

Q.QueueDelete (ch, success);
n = 0;
while (success && isdigit(ch))
{
    n = 10 * n + atoi(ch);
    Q.QueueDelete (ch, success);
}
cout << n;
```

(b) recognizing palindrome (string of characters that reads the same from left to right as it does from right to left)

For example, abcba is a palindrome, bcbac is not.

```
bool   IsPalindrome(string str)
{
        Queue    Q;
        Stack     S;
        char      queueFront;
```

```
                for (int i=0; i< str.length(); i++)
                {
                        nextChar = str[i];
                        Q.QueueInsert(nextChar, success);
                        S.Push(nextChar, success);
                }

                while (!Q.IsEmpty())
                {
                        Q.QueueDelete(queueFront, success);
                        S.Pop(stackTop, success);
                        if (queueFront != stackTop)
                                return false;
                }

                return true;
        }
```

Array based implementation:
<version 1>   Circular array is used:
Initially, front = 0, back = MAXQUEUE-1; count = 0;

Enqueue:

```
 if (count <MAXQUEUE)
{
   back = (back+1)%MAXQUEUE;
   items[back] = newItem;
}
```

Dequeue:

```
if (count > 0)
   front = (front +1 )%MAXQUEUE;
```

➔ need to use “count” to detect “queueEmpty” and “queueFull” situations

<version 2> circular array with extra array element   ➔ no need for variable “count”
(MAXQUEUE elements in array “items”, only use MAXQUEUE-1 elements, the extra array element is sacrificed for efficiency)

Initially, front = 0 (array index before the first value in queue), back=0

Enqueue:

```
        if (front != (back+1)%MAXQUEUE)      // queue not full
        {
           back = (back +1 )%MAXQUEUE;
           item[back] = newItem;
```

```
      count++;
    }

Dequeue:
    if (front !=back)    // queue not empty
    {
        front = (front+1)%MAXQUEUE;
        count --;
    }
```

Queue Class (pointer based implementation)

```
typedef desired-type-of-queue-item queueItemType;

struct queueNode;  // defined in implementation file
typedef queueNode* ptrType;  // pointer to node

class queueClass
{
public:
// constructors and destructor:
   queueClass();                     // default constructor
   queueClass(const queueClass& Q);  // copy constructor
   ~queueClass();                    // destructor

// queue operations:
   bool QueueIsEmpty() const;
   // Determines whether a queue is empty.
   // Precondition: None.
   // Postcondition: Returns true if the queue is empty;
   // otherwise returns false.

   void QueueInsert(queueItemType NewItem, bool& Success);
   // Inserts an item at the back of a queue.
   // Precondition: NewItem is the item to be inserted.
   // Postcondition: If insertion was successful, NewItem
   // is at the back of the queue and Success is true;
   // otherwise Success is false.

   void QueueDelete(bool& Success);
   // Deletes the front of a queue.
   // Precondition: None.
   // Postcondition: If the queue was not empty, the item
   // that was added to the queue earliest is deleted and
   // Success is true. However, if the queue was empty,
   // deletion is impossible and Success is false.

   void QueueDelete(queueItemType& QueueFront,
                    bool& Success);
   // Retrieves and deletes the front of a queue.
   // Precondition: None.
   // Postcondition: If the queue was not empty, QueueFront
   // contains the item that was added to the queue
   // earliest, the item is deleted, and Success is true.
   // However, if the queue was empty, deletion is
   // impossible and Success is false.

   void GetQueueFront(queueItemType& QueueFront,
                      bool& Success) const;
   // Retrieves the item at the front of a queue.
   // Precondition: None.
   // Postcondition: If the queue was not empty, QueueFront
   // contains the item that was added to the queue earliest
   // and Success is true. However, if the queue was empty,
   // the operation fails, QueueFront is unchanged, and
   // Success is false. The queue is unchanged.

private:
   ptrType BackPtr;
};  // end class
// End of header file.

// *********************************************************
// Implementation file QueueP.cpp for the ADT queue.
// Pointer-based implementation.
// *********************************************************
#include "QueueP.h"  // header file
#include <stddef.h>  // for NULL

// The queue is implemented as a circular linked list
// with one external pointer to the back of the queue.
struct queueNode
{  queueItemType Item;
   ptrType       Next;
};  // end struct

queueClass::queueClass() : BackPtr(NULL)
{
}  // end default constructor

queueClass::queueClass(const queueClass& Q)
{
   if  (Q.QueueIsEmpty())
   {
       BackPtr = NULL;
   }
   else
   {
       ptrType curr = Q.BackPtr->next;
       do
       {
           QueueInsert(curr->item, Success);
           curr = curr->next;
       } while (curr != Q.BackPtr->next);
    }
}

queueClass::~queueClass()
{
   bool Success;
   while (!QueueIsEmpty())
      QueueDelete(Success);
   // Assertion: BackPtr == NULL
}  // end destructor

bool queueClass::QueueIsEmpty() const
{
   return bool(BackPtr == NULL);
}  // end QueueIsEmpty

void queueClass::QueueInsert(queueItemType NewItem,
                             bool& Success)
{
   // create a new node
   ptrType NewPtr = new queueNode;

   Success = bool(NewPtr != NULL);  // check allocation
   if (Success)
   {  // allocation successful; set data portion of new node
      NewPtr->Item = NewItem;

      // insert the new node
      if (QueueIsEmpty())
         // insertion into empty queue
         NewPtr->Next = NewPtr;

      else
      {  // insertion into nonempty queue
         NewPtr->Next = BackPtr->Next;
         BackPtr->Next = NewPtr;
      }  // end if

      BackPtr = NewPtr;  // new node is at back
   }  // end if
}  // end QueueInsert

void queueClass::QueueDelete(bool& Success)
{
   Success = bool(!QueueIsEmpty());

   if (Success)
   {  // queue is not empty; remove front
      ptrType FrontPtr = BackPtr->Next;
      if (FrontPtr == BackPtr)   // special case?
         BackPtr = NULL;         // yes, one node in queue
      else
         BackPtr->Next = FrontPtr->Next;

      FrontPtr->Next = NULL;  // defensive strategy
      delete FrontPtr;
   }  // end if
}  // end QueueDelete

void queueClass::QueueDelete(queueItemType& QueueFront,
                             bool& Success)
{
   Success = bool(!QueueIsEmpty());

   if (Success)
   {  // queue is not empty; retrieve front
      ptrType FrontPtr = BackPtr->Next;
      QueueFront = FrontPtr->Item;

      QueueDelete(Success);  // delete front
   }  // end if
}  // end QueueDelete

void queueClass::GetQueueFront(queueItemType& QueueFront,
                               bool& Success) const
{
   Success = bool(!QueueIsEmpty());

   if (Success)
   {  // queue is not empty; retrieve front
      ptrType FrontPtr = BackPtr->Next;
      QueueFront = FrontPtr->Item;
   }   // end if
}  // end GetQueueFront
// End of implementation file.
```

ADT Queue (array based implementation)

```
// ********************************************************
// Header file QueueA.h for the ADT queue.
// Array-based implementation.
// ********************************************************
const int MAX_QUEUE = maximum-size-of-queue;
typedef desired-type-of-queue-item queueItemType;

class queueClass
{
public:
// constructors and destructor:
   queueClass();  // default constructor
   // copy constructor and destructor are
   // supplied by the compiler

// queue operations:
   bool QueueIsEmpty() const;
   void QueueInsert(queueItemType NewItem, bool& Success);
   void QueueDelete(bool& Success);
   void QueueDelete(queueItemType& QueueFront,
                    bool& Success);
   void GetQueueFront(queueItemType& QueueFront,
                      bool& Success) const;

private:
   queueItemType Items[MAX_QUEUE];
   int           Front;
   int           Back;
   int           Count;
};  // end class
```

```
// End of header file.
// ********************************************************
// Implementation file QueueA.cpp for the ADT queue.
// Circular array-based implementation.
// The array has indexes to the front and back of the
// queue. A counter tracks the number of items currently
// in the queue.
// ********************************************************
#include "QueueA.h"  // header file

queueClass::queueClass():
             Front(0), Back(MAX_QUEUE-1), Count(0)
{
}  // end default constructor

bool queueClass::QueueIsEmpty() const
{
   return bool(Count == 0);
}  // end QueueIsEmpty

void queueClass::QueueInsert(queueItemType NewItem,
                             bool& Success)
{
   Success = bool(Count < MAX_QUEUE);

   if (Success)
   {  // queue is not full; insert item
      Back = (Back+1) % MAX_QUEUE;
      Items[Back] = NewItem;
      ++Count;
   }  // end if
}  // end QueueInsert

void queueClass::QueueDelete(bool& Success)
{
   Success = bool(!QueueIsEmpty());

   if (Success)
   {  // queue is not empty; remove front
      Front = (Front+1) % MAX_QUEUE;
      --Count;
   }  // end if
}  // end QueueDelete

void queueClass::QueueDelete(queueItemType& QueueFront,
                             bool& Success)
{
   Success = bool(!QueueIsEmpty());

   if (Success)
   {  // queue is not empty; retrieve and remove front
      QueueFront = Items[Front];
      Front = (Front+1) % MAX_QUEUE;
      --Count;
   }  // end if
}  // end QueueDelete
```

```
void queueClass::GetQueueFront(queueItemType& QueueFront,
                               bool& Success) const
{
   Success = bool(!QueueIsEmpty());

   if (Success)
      // queue is not empty; retrieve front
      QueueFront = Items[Front];
}  // end GetQueueFront
// End of implementation file
```