EE 441 – CH4

STACKS AND QUEUES

Instructor: UĞUR HALICI

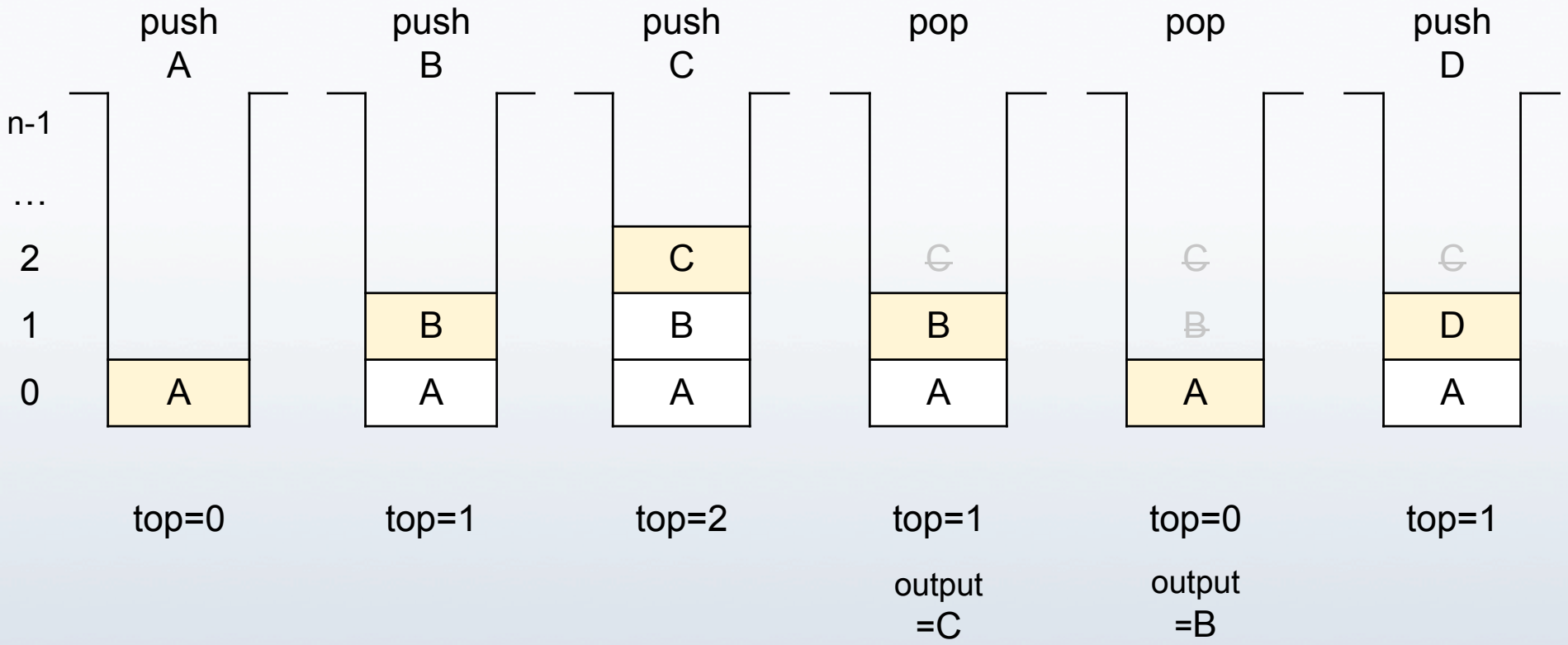
STACKS

A stack is a data structure consisting of a list of items and a pointer to the "top" item in the list.

Items can be inserted or removed from the list only at the top, i.e. the list is ordered in the sequence of entry of items,

Insertions and removals proceed in the "LIFO" last-in-first-out order.

initially stack empty, top=-1



```
# include <iostream.h>
# include <stdlib.h>
const int MaxStackSize=50;
template <class T>
Class Stack
{
 private:
   T stacklist[MaxStackSize],
     int top;
 public:
     Stack(void); // constructor to initialize top
 //modification operations
     void Push(const T& item);
     T Pop(void);
     void Clearstack(void);
//just copy top item without modifying stack contents
     T  Peek(void) const;
//check stack state
     int StackEmpty(void) const;
     int StackFull(void) const;
}
```

```
// Now implementation of Stack methods
template <class T>
Stack<T>::Stack(void):top(-1)
{}
//Stack Empty
template <class T>
int Stack<T>::StackEmpty(void) const
{
  return top==-1; //value is 1 if equal, 0 otherwise
}
// Stack Full
template <class T>
int Stack<T>::StackFull(void) const
{
 return top==(MaxstackSize-1);
}
```

```
//Push
template <class T>
void Stack<T>::Push(const T& item)
{
//can not push if stack has exceeded its limits
    if  StackFull( )
        {
       cerr<<"Stack overflow"<<endl;
       exit(1);
        }
// increment top ptr and copy item into list
    top++;
    stacklist[top]=item;
}
```

```
//pop
template <class T>
T Stack<T>::Pop(void)
    {
      T temp;
      // is stack empty nothing to pop
      if StackEmpty( )
     { cerr<<"Stack empty"<<endl;
          exit(1);
       }
    //record the top element
       temp=stacklist[top];
//decrement top and return the earlier top element
       top--;
       return temp;
  }
```

```
//Peek is the same as Pop, except top is not moved
template <class T>
T Stack<T>::Peek(void) const
    {// write Peek as exercise
    }
//Clear Stack
void Stack::ClearStack(void)
{
 top=-1;
}
```

Example:

Write a program that uses a stack to test a given character string and decides if the string is a palindrome (i.e. reads the same backwards and forwards, e.g. "kabak", " a man a plan a canal panama", etc.)

```
// Algorithm:
// first get rid of all blanks in the string, then push the whole string
// characterwise, into a stack, then pop out characters one by one,
// comparing with the characters of the original de-blanked string
// Assuming that the stack declaration and implementation  are in "stack.h"
# include “stack.h"
#include <iostream.h>
void Deblank(char *s, char *t)
{while (*s!=NULL)
   {if (*s!='  ')
        {*t=*s; t++;}
     s++;
    }
*t=NULL; //append NULL to newstring
}
```

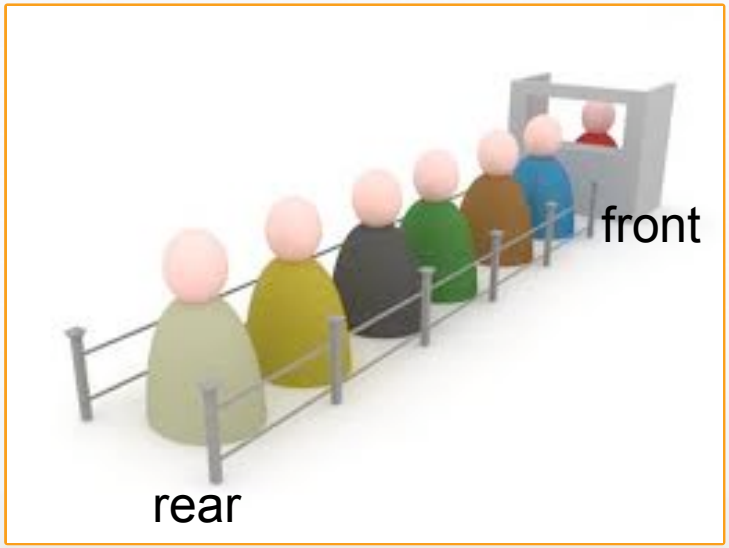
Example (continues)

```
void main ()
{const int True=1, False=0;
// create stack object to store string in reverse order.
Stack s<char>;
char palstring[80], deblankedstring[80], c;
int ispalindrome=True; //we'll stop if false
// get input
    cin.Getline(palstring,80,'\n');
//remove blanks
    Deblank(palstring,deblankedstring);
//push character onto stack
    int i=0; // string array index
    while(deblankedstring[i]!=NULL)
    {
      S.Push(deblankedstring[i]);
      i++;
  }
```

Example (continues)

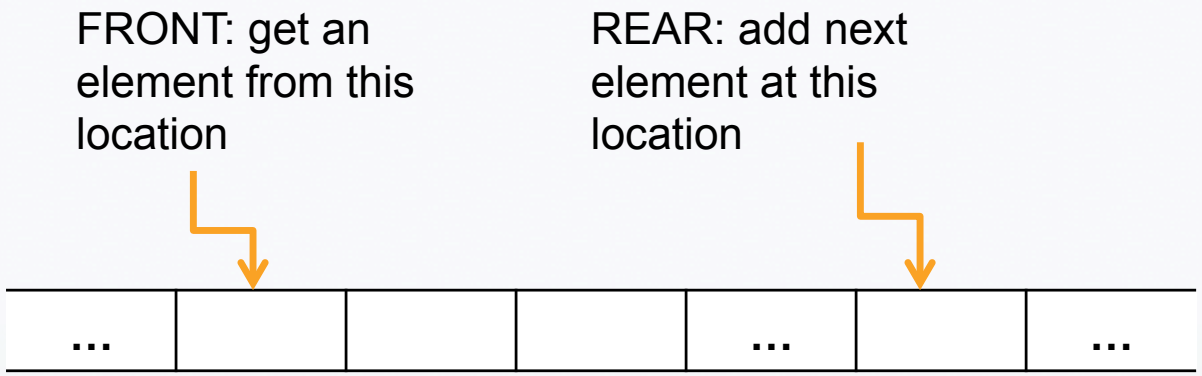
```
//now pop one-by-one comparing with original
    i=0;
    while (!s.StackEmpty())
{
    c=s.Pop();
    //get out of loop when first nonmatch
    if (c!=deblankedstring[i])
    {isPalindrome=False;
    break;
    }
// continue till end of string
    i++;
}
//operation finished. Printout result
if (ispalindrome)
    cout<<"\"<<palstring<<"\"<<"is a palindrome<<endl;
else
    cout<<"\"<<palstring<<"\"<<"is not a palindrome<<endl;
```

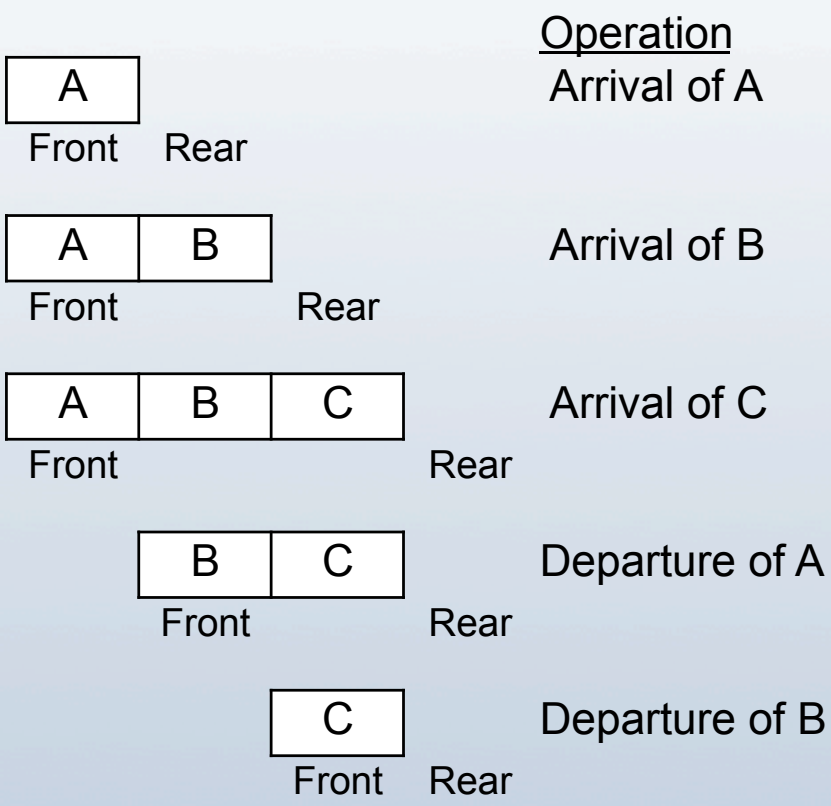
QUEUES



A queue is a data structure that stores elements in a list and permits data access only at the two ends of the list. An element is inserted at the rear of the list and deleted from the front of the list.

Elements are removed in the same order in which they are stored and hence a queue provides FIFO(first-in/first-out) or FCFC(first-come/ first-served) ordering


FRONT: get an element from this location
REAR: add next element at this location
...
...
...


Operation
A
Front
Rear
Arrival of A
A
B
Front
Rear
Arrival of B
A
B
C
Front
Rear
Arrival of C
B
C
Front
Rear
Departure of A
C
Front
Rear
Departure of B

Linear Queues

Consider the following operations performed on a queue of size four. The values of pointers front and rear, and the contents of the queue are shown below after each operation.

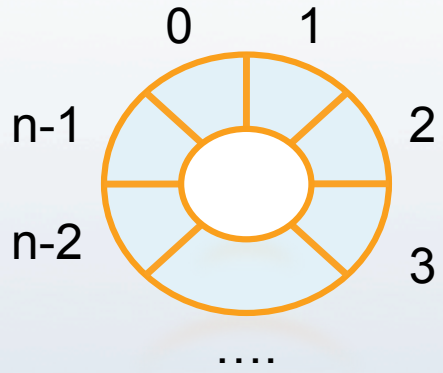
operation	front	rear	0	1	2	3	element	
	0	0						Queue is empty
1. insert 12	0	1	12				12	
2. insert 15	0	2	12	15			15	
3. insert 17	0	3	12	15	17		17	
4. delete	1	3		15	17		12	
5. delete	2	3			17		15	
6. insert 10	2	4			17	10	10	Queue is full
7. insert 5	2	4			17	10		Error

Although there is empty space, it can not be used, so a better representation is needed.

Circular Queues

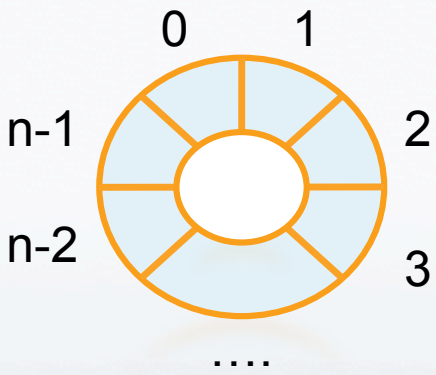
For a more efficient queue representation, consider the array to be circular:

Initially FRONT=REAR=COUNT=0,



There are n locations in the queue, 0,1,...(n-1), and count=0 corresponds to the case the queue is empty. Initially front=rear=0.

We insert a new element at the location pointed by rear, and then we update the rear pointer to point to the next available location. As we insert elements, rear becomes rear+1, however if we insert an element when rear=(n-1), rear becomes 0. The front pointer changes similarly.

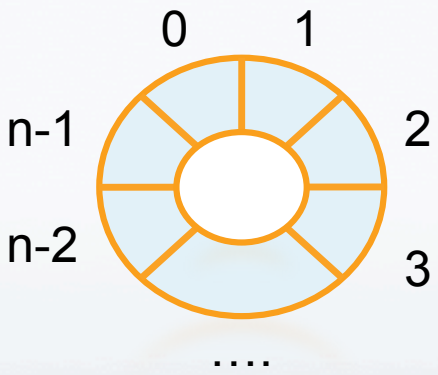


Now consider the case front=0 and rear=(n-1), count=(n-1), that is there are (n-1) elements in the queue.

If we insert another element, queue becomes full and rear=front=0 and count=n.

Therefore in both cases whether the queue is empty or full we have rear=front .

The cases the queue empty or full are discriminated by the value of count.



By using only front and rear we can not discrimnate empty and full cases.

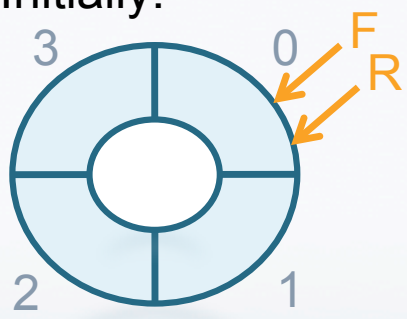
One way of overcoming this problem is using only (n-1) locations of the queue. Then, rear=front represents the case the queue is empty, and

((rear+1) mod n)=front corresponds to the case the queue is full.

Another method might be keeping a flag to indicate whether the queue may become full or empty when rear becomes equal to front, or to use count as explained before

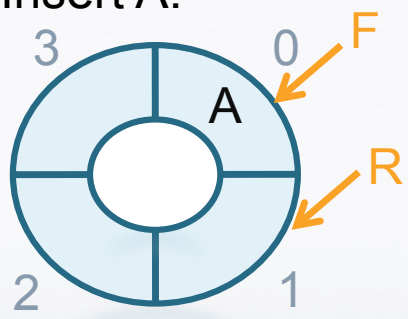
Example

Initially:



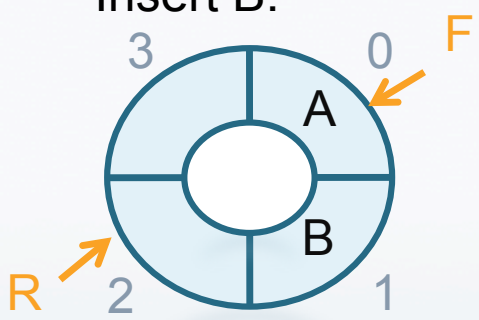
F=0, R=0, C=0

Insert A:



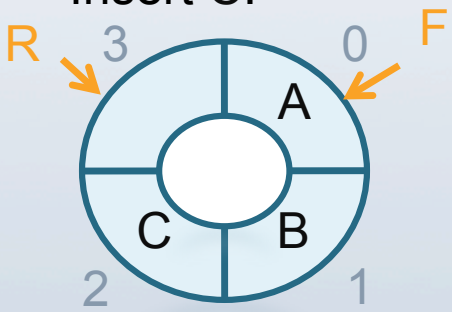
F=0, R=1, C=1

Insert B:



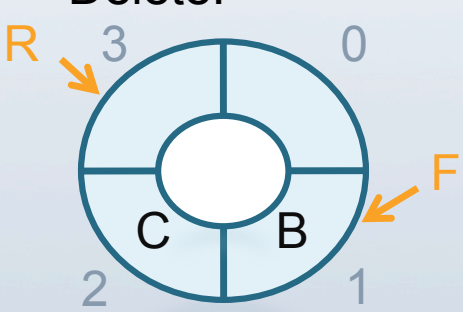
F=0, R=2, C=2

Insert C:



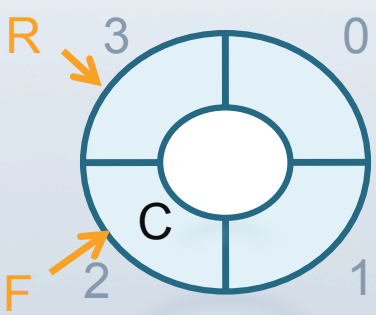
F=0, R=3, C=3

Delete:



F=1, R=3, C=2

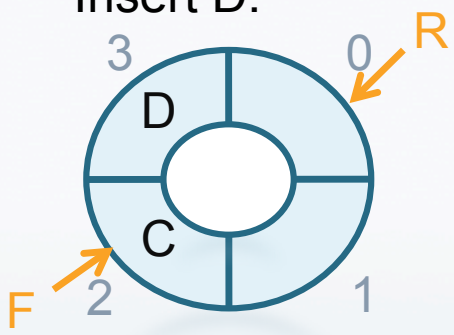
Delete:



F=2, R=3, C=1

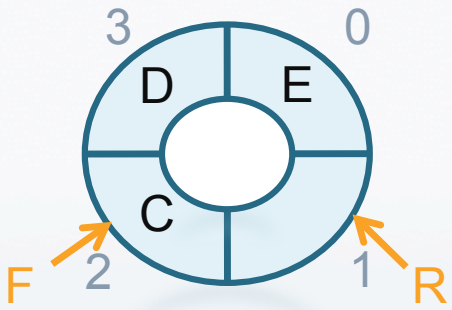
Example

Insert D:



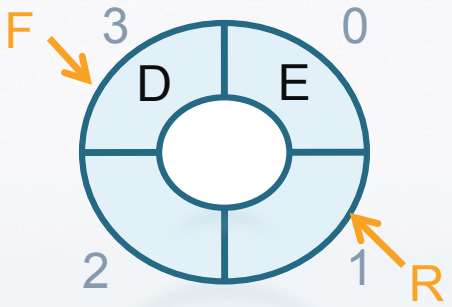
F=2, R=0, C=2

Insert E:



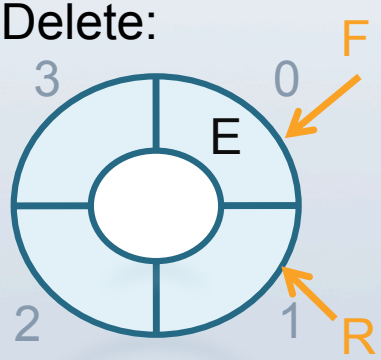
F=2, R=1, C=3

Delete:



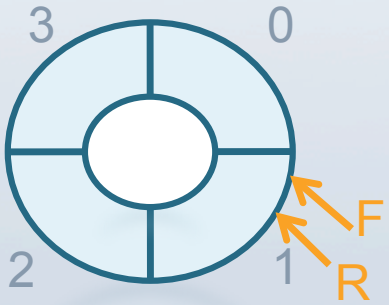
F=3, R=1, C=2

Delete:



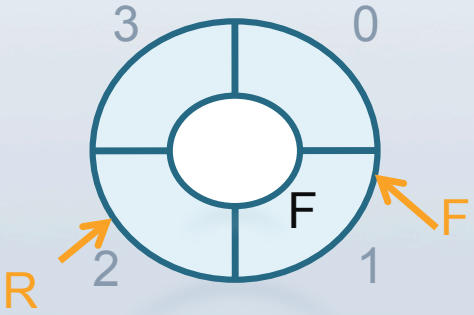
F=0, R=1, C=1

Delete:



F=1, R=1, C=0

Insert F:



F=1, R=2, C=1

Implementation of Queue in C++

Decleration:

```
#include <iostream.h>
#include <stdlib.h>
const int MaxQSize=50;
template <class DataType>

class Queue
{private:
 // queue array and its parameters
int front, rear, count
DataType qlist[MaxQSize];

public:
//constructor
Queue(void); // initialize data members

//queue modification operations
void Qinsert(const &Datatype item);
DataType Qdelete(void);
void ClearQueue(void);

// queue access
DataType QFront(void) const;

// queue test methods
int QLength(void) const;
int QEmpty(void) const;
int QFull(void) const;
};
```

Implementation:

```
// Queue constructor
//initialize queue front, rear, count
Queue::Queue(void): front(0), rear (0), count(0)
{ }


// queue test methods
int Queue::QLength(void) const
{return count};
int Queue::QEmpty(void) const
{return (count= =0)};
int Queue::QFull(void) const
{return (count= =MaxQSize) };
```

```
//Queue Modification Operations

//Qinsert: insert item into the queue
template <class DataType>
void Queue::Qinsert(const Datatype& item)
{
// terminate if queue is full
    if QFull( )
    {
     cerr<<"Queue overflow! <<endl;
     exit(1)
    }
//increment count, assign item to qlist and update rear
    count++;
    qlist[rear]=item;
    rear=(rear+1)% MaxQSize;
 }
```

```
//QDelete : delete element from the front of the queue and return its value
template <class DataType>
Datatype Queue<Datatype>::QDelete(void)
{
  DataType temp;
  // if qlist is empty, terminate the program
  if QEmpty( )
  {
     cerr<<"Deleting from an empty queue!"<<endl;
     exit(1);
  }
  //record value at the front of the queue
 temp=qlist[front];
//decrement count, advance front and return former front
 count --;
 front=(front+1) % MaxQsize;
 return temp;
}
```

```
// queue access
template <class DataType>
DataType Queue<DataType>::QFront(void) const
{return qlist[front]};
```