Chapter 18

Stacks and Queues

Introduction to the Stack ADT

- A stack is a data structure that stores and retrieves items in a last-in-first-out (LIFO) manner
- It’s like the plates in a cafeteria

![Stack Diagram]
Applications of Stacks

- Computer systems use stacks during a program’s execution to store function return addresses, local variables, etc.
- The good calculators (like HP’s) use stacks for performing mathematical operations

Static and Dynamic Stacks

- **Static Stacks**
  - Fixed size
  - Can be implemented with an array
- **Dynamic Stacks**
  - Grow in size as needed
  - Can be implemented with a linked list
Stack Operations

- Operations:
  - push - causes a value to be stored in (pushed onto) the stack
  - pop - retrieves and removes (pops) a value from the stack

- Functions:
  - isFull – true if stack is currently full
  - isEmpty – true if stack currently contains no elements

The Push Operation

- Suppose we have an empty integer stack that is capable of holding a maximum of three values. With that stack we execute the following push operations:

  push(5);
  push(10);
  push(15);
The Push Operation

The state of the stack after each of the push operations:

- push(5);
- push(10);
- push(15);

The Pop Operation

Now, suppose we execute three consecutive pop operations on the same stack:

- pop;
- pop;
- pop;
Contents of Stack.h

```cpp
#ifndef STACK_H
#define STACK_H

template< class T >
class Stack
{
private:
    T *stackArray;
    int stackSize;
    int top;

public:
    Stack ( int ); // constructor
    ~Stack( ); // destructor
    void push ( T );
    T pop ( void ); // different from book
    bool isEmpty ( void );
    bool isFull ( void );
}; // stack.h
#endif
```

Contents of Stack.cpp

```cpp
template < class T >
Stack< T >::Stack(int size)
{
    stackArray = new T[size];
    stackSize = size;
    top = -1;
} // Stack::Stack

template <class T >
Stack< T >::~Stack()
{
    delete [] stackArray;
} // Stack::~Stack
```
template < class T >
void Stack< T >::push(T num)
{  if (isFull())
    cout << "The stack is full.\n";
else
    stackArray[++top] = num;
} // Stack::push

template < class T >
T Stack< T >::pop( void)
{  if (isEmpty())
    cout << "The stack is empty.\n";
else
    return stackArray[top--];
} // Stack::pop

template < class T >
bool Stack< T >::isFull( void )
{  return (top == stackSize - 1) ? true : false;
} // Stack::isFull

template < class T >
bool Stack< T >::isEmpty( void )
{  return (top == -1) ? true : false;
} // Stack::isEmpty
Using a Stack – Program

```cpp
#include <iostream>
#include "Stack.h"
using namespace std;

int main( void )
{   Stack stack(5);
    int catchVar;

    cout << "Pushing 5\n";
    stack.push(5);
    cout << "Pushing 10\n";
    stack.push(10);
    cout << "Pushing 15\n";
    stack.push(15);
    cout << "Pushing 20\n";
    stack.push(20);
    cout << "Pushing 25\n";
    stack.push(25);

    cout << "Popping...\n";
    stack.pop(catchVar);
    cout << catchVar << endl;
    stack.pop(catchVar);
    cout << catchVar << endl;
    stack.pop(catchVar);
    cout << catchVar << endl;
    stack.pop(catchVar);
    cout << catchVar << endl;
    return 0;
} // main
```
Using a Stack – Output

Pushing 5
Pushing 10
Pushing 15
Pushing 20
Pushing 25
Popping...
  25
  20
  15
  10
  5

Using a Stack – Explanation

- In the program, the constructor is called with the argument 5. This sets up the member variables as shown below. Since top is set to –1, the stack is empty

```
stackArray     [0]     top -1 stackSize 5
[1]
[2]
[3]
[4]
```
Using a Stack – Explanation

- Below shows the state of the member variables after the `push` function is called the first time (with 5 as its argument). The top of the stack is now at element 0.

![Stack Diagram 1]

- Below shows the state of the member variables after all five calls to the `push` function. Now the top of the stack is at element 4, and the stack is full.

![Stack Diagram 2]
Using a Stack – Explanation

Implementing Other Stack Operations

- The MathStack class demonstrates functions for stack-based arithmetic like some calculators use.
- The notation for doing stack-based arithmetic is called reversed Polish notation.
Dynamic Stacks

- A dynamic stack is built using a linked list instead of an array.
- A linked list-based stack offers two advantages over an array-based stack.
  - No need to specify the starting size of the stack. A dynamic stack simply starts as an empty linked list, and then expands by one node each time a value is pushed.
  - A dynamic stack is never full, as long as the system has enough free memory.

DynIntStack.h – Program

```cpp
#ifndef DYNINTSTACK_H
#define DYNINTSTACK_H

class DynIntStack {
private:
    struct StackNode {
        int value;
        StackNode *next;
    }; // StackNode
public:
    DynIntStack ( void ) { top = NULL; }
    ~DynIntStack( );
    void push ( int );
    void pop ( int &);
    bool isEmpty ( void );
}; // DynIntStack
#endif
```
#include <iostream>
#include "DynIntStack.h"
using namespace std;

DynIntStack::~DynIntStack()
{
    StackNode *nodePtr, *nextNode;
    for ( nodePtr = top; nodePtr; nodePtr = nextNode )
        { nextnode = nodePtr->next;
          delete nodePtr;
        } // for
} // DynIntStack::~DynIntStack

void DynIntStack::push ( int num )
{ stackNode *newNode = new Node( num );
    if ( isEmpty() )
        { top = newNode; newNode->next = NULL; } // if
    else
        { newNode->next = top; top = newNode; } // else
} // DynIntStack::push

void DynIntStack::pop ( int &num )
{ stackNode *temp;
    if ( isEmpty( ) )
        cout << "The stack is empty.\n";
    else
        { num = top->value;
          temp = top->next;
          delete top;
          top = temp;
        } // else
} // DynIntStack::pop

bool DynIntStack::isEmpty ( void )
{ return !top;
} // DynIntStack::isEmpty
The STL Stack Container

- The STL stack container may be implemented as a vector, a list, or a deque (which you will learn about later in this chapter)
  - Implements push, pop, and empty
  - Also implements size and top
- Because the stack container is used to adapt these other containers, it is often referred to as a container adapter

Here are examples of how to declare a stack of ints, implemented as a vector, a list, and a deque

```cpp
stack< int, vector<int> > iStack;  // Vector stack
stack< int, list<int> > iStack;    // List stack
stack< int > iStack;              // Default - deque stack
```

- The stack container’s member functions are listed and described in the book
### STL Stack – Program

```cpp
#include <iostream>
#include <vector>
#include <stack>
using namespace std;

int main ( void )
{
    int x;
    stack< int, vector<int> > iStack;
    for ( x = 2; x < 8; x += 2 )
    {
        cout << "Pushing " << x << endl;
        iStack.push(x);
    } // for

    cout << "The size of the stack is ";
    cout << iStack.size() << endl;
    for ( x = 2; x < 8; x += 2 )
    {
        cout << "Popping " << iStack.top() << endl;
        iStack.pop();
    } // for
} // main
```

### STL stack – Output

Pushing 2  
Pushing 4  
Pushing 6  
The size of the stack is 3  
Popping 6  
Popping 4  
Popping 2
Introduction to the Queue ADT

- Like a stack, a queue (pronounced "cue") is a data structure that holds a sequence of elements.
- A queue, however, provides access to its elements in first-in, first-out (FIFO) order.
- The elements in a queue are processed like customers standing in a grocery check-out line: the first customer in line is the first one served.

Example Applications of Queues

- In a multi-user system, a queue is used to hold print jobs submitted by users, while the printer services those jobs one at a time.
- Communications software also uses queues to hold information received over networks connections.
- Sometimes information is transmitted to a system faster than it can be processed, so it is placed in a queue when it is received.
Static and Dynamic Queues

- Just as stacks are implemented as arrays or linked lists, so are queues.
- Dynamic queues offer the same advantages over static queues that dynamic stacks offer over static stacks.

Queue Operations

- Think of queues as having a front and a rear. This is illustrated below.
Queue Operations

- The two primary queue operations are:
  - **enqueue** – insert an element at the rear of a queue
  - **dequeue** – remove an element from the front of a queue

Suppose we have an empty static integer queue that is capable of holding a maximum of three values.

With that queue we execute the following enqueue operations:

enqueue(3);
enqueue(6);
enqueue(9);
Queue Operations

- Below illustrates the state of the queue after each of the enqueue operations.

![Diagram of queue operations](image)

Queue Operations

- Now let's see how dequeue operations are performed. Below illustrates the state of the queue after each of three consecutive dequeue operations.

![Diagram of dequeue operations](image)
Queue Operations

- When the last dequeue operation is performed in the illustration, the queue is empty
  - An empty queue can be signified by setting both front and rear indices to –1
- We implement the queue as a circular array

Contents of IntQueue.h

```cpp
#ifndef INTQUEUE_H
#define INTQUEUE_H

class IntQueue
{
  private:
    int *queueArray;
    int queueSize;
    int front;
    int rear;
    int numItems;
  public:
    IntQueue ( int );
    ~IntQueue ( void );
    void enqueue ( int );
    int dequeue ( void );
    bool isEmpty ( void );
    bool isFull ( void );
    void clear ( void );
}; // IntQueue

#endif
```

# Contents of IntQueue.cpp

```cpp
#include <iostream>
#include "IntQueue.h"
using namespace std;

IntQueue::IntQueue( int s )
{ queueArray = new int[s];
  queueSize = s;
  front = -1;
  rear = -1;
  numItems = 0;
} // IntQueue::IntQueue

IntQueue::~IntQueue( void )
{ delete [] queueArray;
} // IntQueue::~IntQueue

void IntQueue::enqueue( int num )
{ if ( isFull() )
    cout << "The queue is full.\n";
  else
    { rear = (rear + 1) % queueSize;
      queueArray[rear] = num;
      numItems++;
    } // else
} // IntQueue::enqueue

int IntQueue::dequeue( void )
{ if ( isEmpty() )
    cout << "The queue is empty.\n";
  else
    { front = (front + 1) % queueSize;
      numItems--;
      return queueArray[front];
    } // else
} // IntQueue::dequeue
```
### Contents of IntQueue.cpp

```cpp
bool IntQueue::isEmpty ( void )
{ return numItems == 0;
} // IntQueue::isEmpty

bool IntQueue::isFull ( void )
{ return numItems < queueSize);
} // IntQueue::isFull

void IntQueue::clear ( void)
{ front = queueSize - 1;
  rear = queueSize - 1;
  numItems = 0;
} // IntQueue::clear
```

### Using Queues – Program

```cpp
#include <iostream>
#include "intqueue.h"
using namespace std;
int main ( void )
{
    int iQueue ( 5 )
    { ntQueue iQueue ( 5 );
        cout << "Enqueuing 5 items...\n";
        for ( int x = 0; x < 5; x++ )
            iQueue.enqueue(x);
        cout << "Now attempting to enqueue again...\n";
        iQueue.enqueue(5);
        cout << "The values in the queue were:\n";
        while ( !iQueue.isEmpty() )
        { int value;
            iQueue.dequeue( value );
            cout << value << endl;
        } // while
        return 0;
    } // main
```
Using Queues – Output

Enqueuing 5 items...
Now attempting to enqueue again...
The queue is full.
The values in the queue were:
0

Dynamic Queues

- A dynamic queue starts as an empty linked list
- With the first enqueue operation, a node is added, which is pointed to by front and rear pointers
- As each new item is added to the queue, a new node is added to the rear of the list, and the rear pointer is updated to point to the new node
- As each item is dequeued, the node pointed to by the front pointer is deleted, and front is made to point to the next node in the list
Dynamic Queues

- Below shows the structure of a dynamic queue

```
#include <iostream>

class Node
{
public:
    int value;
    Node *next;
};

class DynIntQueue
{
private:
    Node *front;
    Node *rear;
    int numItems;
public:
    DynIntQueue ( void );
    ~DynIntQueue ( void );
    void enqueue ( int );
    int dequeue ( void );
    bool isEmpty ( void );
    bool isFull ( void );
    void clear ( void );
};
```

Contents of DynIntQueue.h
# Contents of DynIntQueue.cpp

```cpp
#include <iostream>
#include "DynIntQueue.h"
using namespace std;

DynIntQueue::DynIntQueue( void )
{ front = NULL;
  rear = NULL;
  numItems = 0;
} // DynIntQueue::DynIntQueue

DynIntQueue::~DynIntQueue( void )
{ clear();
} // DynIntQueue::~DynIntQueue

void DynIntQueue::enqueue(int num)
{ QueueNode *newNode);
  newNode = new QueueNode;
  newNode->value = num;
  newNode->next = NULL;
  if ( isEmpty() )
  { front = newNode;
    rear = newNode;
  } // if
  else
  { rear->next = newNode;
    rear = newNode;
  } // else
  numItems++;
} // DynIntQueue::enqueue
```
Contents of DynIntQueue.cpp

```cpp
int DynIntQueue::dequeue( void )
{ QueueNode *temp;
  int num;
  if ( isEmpty( ) )
    cout << "The queue is empty.\n";
  else
  { num = front->value;
    temp = front->next;
    delete front;
    front = temp;
    numItems--;
    return num;
  } // else
} // DynIntQueue::dequeue
```

```cpp
bool DynIntQueue::isEmpty ( void )
{ return numItems == 0;
} // DynIntQueue::isEmpty
```

```cpp
void DynIntQueue::clear ( void )
{ int value;
  for ( ; !isEmpty( ); dequeue(value) );
} // DynIntQueue::clear
```
#include <iostream>
#include "dynintqueue.h"
using namespace std;

int main(void)
{
    DynIntQueue iQueue;

    cout << "Enqueuing 5 items...
";
    for ( int x = 0; x < 5; x++ )
    
        iQueue.enqueue(x);
    cout << "The values in the queue were:
";
    while ( !iQueue.isEmpty() )
    {
        int value;
        iQueue.dequeue(value);
        cout << value << endl;
    } // while

    return 0;
} // main
The STL deque and queue Containers

- A **deque** (pronounced "deck") is a double-ended queue
  - It similar to a vector, but allows efficient access to values at both the front and the rear
- The queue ADT is like the stack ADT; it is actually a container adapter.

The deque Container

- Programs that use the deque ADT must include the deque header file
- The `push_back`, `pop_front`, and `front` member functions are described in the book
#include <iostream>
#include <deque>
using namespace std;

int main ( void )
{
    int x;
    deque<int> iDeque;
    cout << "I will now enqueue items...\n";
    for ( x = 2; x < 8; x += 2 )
    {
        cout << "Pushing " << x << endl;
        iDeque.push_back( x );
    } // for
    cout << "I will now dequeue items...\n";
    for ( x = 2; x < 8; x += 2 )
    {
        cout << "Popping " << iDeque.front() << endl;
        iDeque.pop_front( );
    } // for
    return 0;
} // main

Using STL deques – Output

I will now enqueue items...
Pushing 2
Pushing 4
Pushing 6
I will now dequeue items...
Popping 2
Popping 4
Popping 6
The queue Container Adapter

- The queue container adapter can be built upon vectors, lists, or deques
- By default, it uses deque as its base

The queue Container Adapter

- The queue insertion and removal operations are the same as those supported by the stack ADT: push, pop, and top
- The queue version of push always inserts an element at the rear of the queue
- The queue version of pop always removes an element from the structure’s front.
- The top function returns the value of the element at the front of the queue
Using STL queue – Program

```cpp
#include <iostream>
#include <queue>
using namespace std;

int main ( void )
{
    int x;
    queue<int> iQueue;
    cout << "I will now enqueue items...
";
    for ( x = 2; x < 8; x += 2 )
    {
        cout << "Pushing " << x << endl;
        iQueue.push_back( x );
    } // for
    cout << "I will now dequeue items...
";
    for ( x = 2; x < 8; x += 2 )
    {
        cout << "Popping " << iQueue.front() << endl;
        iQueue.pop_front();
    } // for
    return 0;
} // main
```

Using STL queue – Output

I will now enqueue items...
Pushing 2
Pushing 4
Pushing 6
I will now dequeue items...
Popping 2
Popping 4
Popping 6