Stack and Queue Implementation Code

CSE 225 - Data Structures and Algorithms

Md. Mahfuzur Rahman
ECE Department
North South University

1 Stack Implementation

1.1 ItemType

```
//FILE "ItemType.h"
const int MAX_ITEMS = 5;
typedef int ItemType;
```

1.2 Static-Array Based Implementation

```
#include "ItemType.h"

//   The user of this file must provied a file "ItemType.h" that defines:
//       ItemType : the class definition of the objects on the stack.
//       MAX_ITEMS: the maximum number of items on the stack.

//   Class specification for Stack ADT in file Stack1.h


class FullStack
// Exception class thrown by Push when stack is full.
{};

class EmptyStack
// Exception class thrown by Pop and Top when stack is emtpy.
{};


class StackType
{
public:

   StackType();
   // Class constructor.
   bool IsFull() const;
   // Function: Determines whether the stack is full.
   // Pre:  Stack has been initialized.
   // Post: Function value = (stack is full)
   bool IsEmpty() const;
   // Function: Determines whether the stack is empty.
   // Pre:  Stack has been initialized.
   // Post: Function value = (stack is empty)
   void Push(ItemType item);
   // Function: Adds newItem to the top of the stack.
   // Pre:  Stack has been initialized.
   // Post: If (stack is full), FullStack exception is thrown;
```

```
    //      otherwise, newItem is at the top of the stack.
    void Pop();
    // Function: Removes top item from the stack.
    // Pre:  Stack has been initialized.
    // Post: If (stack is empty), EmptyStack exception is thrown;
    //      otherwise, top element has been removed from stack.
    ItemType Top();
    // Function: Returns a copy of top item on the stack.

private:
    int top;
    ItemType  items[MAX_ITEMS];
};
```

```
// File: StackType.cpp

#include "StackType.h"
#include <iostream>
StackType::StackType()
{
  top = -1;
}

bool StackType::IsEmpty() const
{
  return (top == -1);
}

bool StackType::IsFull() const
{
  return (top ==  MAX_ITEMS-1);
}

void StackType::Push(ItemType newItem)
{
  if( IsFull() )
    throw FullStack();
  top++;
  items[top] = newItem;
}

void StackType::Pop()
{
  if( IsEmpty() )
    throw EmptyStack();
  top--;
}

ItemType StackType::Top()
{
  if (IsEmpty())
    throw EmptyStack();
  return items[top];
}
```

1.3 Dynamic-Array Based Implementation

```
#include "StackType.h"
StackType::StackType(int max)
{
  maxStack = max;
  top = -1;
  items = new ItemType[maxStack];
}


StackType::StackType()
{
  maxStack = 500;
  top = -1;
  items = new ItemType[maxStack];
}


bool StackType::IsEmpty() const
{
  return (top == -1);
}


bool StackType::IsFull() const
{
  return (top == maxStack-1);
}


void StackType::Push(ItemType newItem)
{
  if (IsFull())
    throw FullStack();
  top++;
  items[top] = newItem;
}


void StackType::Pop()
{
  if( IsEmpty() )
    throw EmptyStack();
  top--;
}


ItemType StackType::Top()
{
  if (IsEmpty())
    throw EmptyStack();
  return items[top];
}


StackType::~StackType()
{
  delete [] items;
}
```

1.4 Linked-List Based Implementation

```
// Implementation file for linked StackType
#include "StackType.h"
#include <new>
struct NodeType
{
  ItemType info;
  NodeType* next;
};

void StackType::Push(ItemType newItem)
// Adds newItem to the top of the stack.
// Stack is bounded by size of memory.
// Pre:  Stack has been initialized.
// Post: If stack is full, FullStack exception is thrown;
//       else newItem is at the top of the stack.

{
  if (IsFull())
    throw FullStack();
  else
  {
    NodeType* location;
    location = new NodeType;
    location->info = newItem;
    location->next = topPtr;
    topPtr = location;
  }
}
void StackType::Pop()
// Removes top item from Stack and returns it in item.
// Pre:  Stack has been initialized.
// Post: If stack is empty, EmptyStack exception is thrown;
//       else top element has been removed.
{
  if (IsEmpty())
    throw EmptyStack();
  else
  {
    NodeType* tempPtr;
    tempPtr = topPtr;
    topPtr = topPtr->next;
    delete tempPtr;
  }
}

ItemType StackType::Top()
// Returns a copy of the top item in the stack.
// Pre:  Stack has been initialized.
// Post: If stack is empty, EmptyStack exception is thrown;
//       else a copy of the top element is returned.
{
  if (IsEmpty())
    throw EmptyStack();
  else
    return topPtr->info;
}

StackType::StackType()  // Class constructor.
{
  topPtr = NULL;
}
bool StackType::IsFull() const
// Returns true if there is no room for another ItemType
//  on the free store; false otherwise.
{
    NodeType* location;
  try
  {
    location = new NodeType;
    delete location;
    return false;
  }
  catch(std::bad_alloc exception)
  {
    return true;
  }
}

StackType::~StackType()
// Post: stack is empty; all items have been deallocated.
{
  NodeType* tempPtr;

  while (topPtr != NULL)
  {
    tempPtr = topPtr;
    topPtr = topPtr->next;
    delete tempPtr;
  }
}

bool StackType::IsEmpty() const
{
  return (topPtr == NULL);
}
```

2 Queue Implementation

2.1 Dynamic-Array Based Implementation

```
//FILE QueueType.h
class FullQueue
{};

class EmptyQueue
{};
typedef char ItemType;
class QueType
{
public:
    QueType();
    // Class constructor.
    // Because there is a default constructor, the precondition
    // that the queue has been initialized is omitted.
    QueType(int max);
    // Parameterized class constructor.
    ~QueType();
    // Class destructor.
    void MakeEmpty();
    // Function: Initializes the queue to an empty state.
    // Post: Queue is empty.
    bool IsEmpty() const;
    // Function: Determines whether the queue is empty.
    // Post: Function value = (queue is empty)
    bool IsFull() const;
    // Function: Determines whether the queue is full.
    // Post: Function value = (queue is full)
    void Enqueue(ItemType newItem);
    // Function: Adds newItem to the rear of the queue.
    // Post: If (queue is full) FullQueue exception is thrown
    //       else newItem is at rear of queue.
    void Dequeue(ItemType& item);
    // Function: Removes front item from the queue and returns it in item.
    // Post: If (queue is empty) EmptyQueue exception is thrown
    //       and item is undefined
    //       else front element has been removed from queue and
    //       item is a copy of removed element.
private:
    int front;
    int rear;
    ItemType* items;
    int maxQue;
};
```

```cpp
#include "QueType.h"

QueType::QueType(int max)
// Parameterized class constructor
// Post: maxQue, front, and rear have been initialized.
//       The array to hold the queue elements has been dynamically
//       allocated.
{
  maxQue = max + 1;
  front = maxQue - 1;
  rear = maxQue - 1;
  items = new ItemType[maxQue];
}
QueType::QueType()          // Default class constructor
// Post: maxQue, front, and rear have been initialized.
//       The array to hold the queue elements has been dynamically
//       allocated.
{
  maxQue = 501;
  front = maxQue - 1;
  rear = maxQue - 1;
  items = new ItemType[maxQue];
}
QueType::~QueType()         // Class destructor
{
  delete [] items;
}

void QueType::MakeEmpty()
// Post: front and rear have been reset to the empty state.
{
  front = maxQue - 1;
  rear = maxQue - 1;
}

bool QueType::IsEmpty() const
// Returns true if the queue is empty; false otherwise.
{
  return (rear == front);
}

bool QueType::IsFull() const
// Returns true if the queue is full; false otherwise.
{
  return ((rear + 1) % maxQue == front);
}

void QueType::Enqueue(ItemType newItem)
// Post: If (queue is not full) newItem is at the rear of the queue;
//       otherwise a FullQueue exception is thrown.
{
  if (IsFull())
    throw FullQueue();
  else
  {
    rear = (rear +1) % maxQue;
    items[rear] = newItem;
  }
}

void QueType::Dequeue(ItemType& item)
// Post: If (queue is not empty) the front of the queue has been
//       removed and a copy returned in item;
//       othersiwe a EmptyQueue exception has been thrown.
{
  if (IsEmpty())
    throw EmptyQueue();
  else
  {
    front = (front + 1) % maxQue;
    item = items[front];
  }
}
```

2.2 Linked-List Based Implementation

```
// Header file for Queue ADT
class FullQueue
{};

class EmptyQueue
{};
typedef char ItemType
struct NodeType;

class QueType
{
public:
  QueType();
  ~QueType();
  void MakeEmpty();
  void Enqueue(ItemType);
  void Dequeue(ItemType&);
  bool IsEmpty() const;
  bool IsFull() const;
private:
  NodeType* front;
  NodeType* rear;
};

#include <cstddef>                    // For NULL.
#include <new>                        // For bad_alloc.
struct NodeType
{
  ItemType info;
  NodeType* next;
};
```

```cpp

QueType::QueType()           // Class constructor.
// Post:  front and rear are set to NULL.
{
  front = NULL;
  rear = NULL;
}

void QueType::MakeEmpty()
// Post: Queue is empty; all elements have been deallocated.
{
  NodeType* tempPtr;

  while (front != NULL)
  {
    tempPtr = front;
    front = front->next;
    delete tempPtr;
  }
  rear = NULL;
}

// Class destructor.
QueType::~QueType()
{
  MakeEmpty();
}

bool QueType::IsFull() const
// Returns true if there is no room for another NodeType object
//  on the free store and false otherwise.
{
  NodeType* location;
  try
  {
    location = new NodeType;

    delete location;
    return false;
  }
  catch(std::bad_alloc exception)
  {
    return true;
  }
}

bool QueType::IsEmpty() const
// Returns true if there are no elements on the queue and false otherwise.
{
  return (front == NULL);
}

void QueType::Enqueue(ItemType newItem)
// Adds newItem to the rear of the queue.
// Pre:  Queue has been initialized.
// Post: If (queue is not full), newItem is at the rear of the queue;
//       otherwise, a FullQueue exception is thrown.

{
  if (IsFull())
    throw FullQueue();
  else
  {
    NodeType* newNode;

    newNode = new NodeType;
    newNode->info = newItem;
    newNode->next = NULL;
    if (rear == NULL)
      front = newNode;
    else
      rear->next = newNode;
    rear = newNode;
  }
}

void QueType::Dequeue(ItemType& item)
// Removes front item from the queue and returns it in item.
// Pre:  Queue has been initialized
// Post: If (queue is not empty), the front of the queue has been
//       removed and a copy returned in item;
//       otherwise, an EmptyQueue exception is thrown.

{
  if (IsEmpty())
    throw EmptyQueue();
  else
  {
    NodeType* tempPtr;

    tempPtr = front;
    item = front->info;
    front = front->next;
    if (front == NULL)
      rear = NULL;
    delete tempPtr;
  }
}
```