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;
```

```
37    //     otherwise, newItem is at the top of the stack.
38    void Pop();
39    // Function: Removes top item from the stack.
40    // Pre:  Stack has been initialized.
41    // Post: If (stack is empty), EmptyStack exception is thrown;
42    //     otherwise, top element has been removed from stack.
43    ItemType Top();
44    // Function: Returns a copy of top item on the stack.
45
46 private:
47    int top;
48    ItemType  items[MAX_ITEMS];
49 };
```

```
1  // File: StackType.cpp
2
3  #include "StackType.h"
4  #include <iostream>
5  StackType::StackType()
6  {
7    top = -1;
8  }
9
10 bool StackType::IsEmpty() const
11 {
12   return (top == -1);
13 }
14
15 bool StackType::IsFull() const
16 {
17   return (top ==  MAX_ITEMS-1);
18 }
19
20 void StackType::Push(ItemType newItem)
21 {
22   if( IsFull() )
23     throw FullStack();
24   top++;
25   items[top] = newItem;
26 }
27
28 void StackType::Pop()
29 {
30   if( IsEmpty() )
31     throw EmptyStack();
32   top--;
33 }
34
35 ItemType StackType::Top()
36 {
37   if (IsEmpty())
38     throw EmptyStack();
39   return items[top];
40 }
```

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];
```

```
52 }
53
54
55 StackType::~StackType()
56 {
57   delete [] items;
58 }
```

1.4 Linked-List Based Implementation

```
1  // Implementation file for linked StackType
2  #include "StackType.h"
3  #include <new>
4  struct NodeType
5  {
6    ItemType info;
7    NodeType* next;
8  };
9
10 void StackType::Push(ItemType newItem)
11 // Adds newItem to the top of the stack.
12 // Stack is bounded by size of memory.
13 // Pre:  Stack has been initialized.
14 // Post: If stack is full, FullStack exception is thrown;
15 //       else newItem is at the top of the stack.
16
17 {
18   if (IsFull())
19     throw FullStack();
20   else
21   {
22     NodeType* location;
23     location = new NodeType;
24     location->info = newItem;
25     location->next = topPtr;
26     topPtr = location;
27   }
28 }
29 void StackType::Pop()
30 // Removes top item from Stack and returns it in item.
31 // Pre:  Stack has been initialized.
32 // Post: If stack is empty, EmptyStack exception is thrown;
33 //       else top element has been removed.
34 {
35   if (IsEmpty())
36     throw EmptyStack();
37   else
38   {
39     NodeType* tempPtr;
40     tempPtr = topPtr;
41     topPtr = topPtr->next;
42     delete tempPtr;
43   }
```

```
44 }
45
46 ItemType StackType::Top()
47 // Returns a copy of the top item in the stack.
48 // Pre:  Stack has been initialized.
49 // Post: If stack is empty, EmptyStack exception is thrown;
50 //       else a copy of the top element is returned.
51 {
52   if (IsEmpty())
53     throw EmptyStack();
54   else
55     return topPtr->info;
56 }
57
58 StackType::StackType()  // Class constructor.
59 {
60   topPtr = NULL;
61 }
62 bool StackType::IsFull() const
63 // Returns true if there is no room for another ItemType
64 //  on the free store; false otherwise.
65 {
66     NodeType* location;
67   try
68   {
69     location = new NodeType;
70     delete location;
71     return false;
72   }
73   catch(std::bad_alloc exception)
74   {
75     return true;
76   }
77 }
78
79 StackType::~StackType()
80 // Post: stack is empty; all items have been deallocated.
81 {
82   NodeType* tempPtr;
83
84   while (topPtr != NULL)
85   {
86     tempPtr = topPtr;
87     topPtr = topPtr->next;
88     delete tempPtr;
89   }
90 }
91
92 bool StackType::IsEmpty() const
93 {
94   return (topPtr == NULL);
95 }
```

2 Queue Implementation

2.1 Dynamic-Array Based Implementation

```
1  //FILE QueueType.h
2  class FullQueue
3  {};
4
5  class EmptyQueue
6  {};
7  typedef char ItemType;
8  class QueType
9  {
10 public:
11     QueType();
12     // Class constructor.
13     // Because there is a default constructor, the precondition
14     // that the queue has been initialized is omitted.
15     QueType(int max);
16     // Parameterized class constructor.
17     ~QueType();
18     // Class destructor.
19     void MakeEmpty();
20     // Function: Initializes the queue to an empty state.
21     // Post: Queue is empty.
22     bool IsEmpty() const;
23     // Function: Determines whether the queue is empty.
24     // Post: Function value = (queue is empty)
25     bool IsFull() const;
26     // Function: Determines whether the queue is full.
27     // Post: Function value = (queue is full)
28     void Enqueue(ItemType newItem);
29     // Function: Adds newItem to the rear of the queue.
30     // Post: If (queue is full) FullQueue exception is thrown
31     //       else newItem is at rear of queue.
32     void Dequeue(ItemType& item);
33     // Function: Removes front item from the queue and returns it in item.
34     // Post: If (queue is empty) EmptyQueue exception is thrown
35     //       and item is undefined
36     //       else front element has been removed from queue and
37     //       item is a copy of removed element.
38 private:
39     int front;
40     int rear;
41     ItemType* items;
42     int maxQue;
43 };
```

```
#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
```

```
55    {
56      rear = (rear +1) % maxQue;
57      items[rear] = newItem;
58    }
59  }
60
61  void QueType::Dequeue(ItemType& item)
62  // Post: If (queue is not empty) the front of the queue has been
63  //       removed and a copy returned in item;
64  //       othersiwe a EmptyQueue exception has been thrown.
65  {
66    if (IsEmpty())
67      throw EmptyQueue();
68    else
69    {
70      front = (front + 1) % maxQue;
71      item = items[front];
72    }
73  }
```

2.2 Linked-List Based Implementation

```
1  // Header file for Queue ADT
2  class FullQueue
3  {};
4
5  class EmptyQueue
6  {};
7  typedef char ItemType
8  struct NodeType;
9
10 class QueType
11 {
12 public:
13   QueType();
14   ~QueType();
15   void MakeEmpty();
16   void Enqueue(ItemType);
17   void Dequeue(ItemType&);
18   bool IsEmpty() const;
19   bool IsFull() const;
20 private:
21   NodeType* front;
22   NodeType* rear;
23 };
24
25 #include <cstddef>                  // For NULL.
26 #include <new>                      // For bad_alloc.
27 struct NodeType
28 {
29   ItemType info;
30   NodeType* next;
31 };
```

```
32
33 QueType::QueType()           // Class constructor.
34 // Post:  front and rear are set to NULL.
35 {
36   front = NULL;
37   rear = NULL;
38 }
39
40 void QueType::MakeEmpty()
41 // Post: Queue is empty; all elements have been deallocated.
42 {
43   NodeType* tempPtr;
44
45   while (front != NULL)
46   {
47     tempPtr = front;
48     front = front->next;
49     delete tempPtr;
50   }
51   rear = NULL;
52 }
53
54 // Class destructor.
55 QueType::~QueType()
56 {
57   MakeEmpty();
58 }
59
60 bool QueType::IsFull() const
61 // Returns true if there is no room for another NodeType object
62 //  on the free store and false otherwise.
63 {
64   NodeType* location;
65   try
66   {
67     location = new NodeType;
68
69     delete location;
70     return false;
71   }
72   catch(std::bad_alloc exception)
73   {
74     return true;
75   }
76 }
77
78 bool QueType::IsEmpty() const
79 // Returns true if there are no elements on the queue and false otherwise.
80 {
81   return (front == NULL);
82 }
83
84 void QueType::Enqueue(ItemType newItem)
85 // Adds newItem to the rear of the queue.
```

```
86 // Pre:  Queue has been initialized.
87 // Post: If (queue is not full), newItem is at the rear of the queue;
88 //       otherwise, a FullQueue exception is thrown.
89
90 {
91   if (IsFull())
92     throw FullQueue();
93   else
94   {
95     NodeType* newNode;
96
97     newNode = new NodeType;
98     newNode->info = newItem;
99     newNode->next = NULL;
100    if (rear == NULL)
101      front = newNode;
102    else
103      rear->next = newNode;
104    rear = newNode;
105  }
106 }
107
108 void QueType::Dequeue(ItemType& item)
109 // Removes front item from the queue and returns it in item.
110 // Pre:  Queue has been initialized
111 // Post: If (queue is not empty), the front of the queue has been
112 //       removed and a copy returned in item;
113 //       otherwise, an EmptyQueue exception is thrown.
114
115 {
116   if (IsEmpty())
117     throw EmptyQueue();
118   else
119   {
120     NodeType* tempPtr;
121
122     tempPtr = front;
123     item = front->info;
124     front = front->next;
125     if (front == NULL)
126       rear = NULL;
127     delete tempPtr;
128   }
129 }
```