Stacks and Queues

Unit 6
Chapter 19.1-2, 4-5

CS 2308
Fall 2018

Jill Seaman

Abstract Data Type

- A data type for which:
  - only the properties of the data and the operations to be performed on the data are specific,
  - how the data will be represented or how the operations will be implemented is unspecified.
- An ADT may be implemented using various specific data types or data structures, in many ways and in many programming languages.
- Examples:
  - NumberList (implemented using linked list or array)
  - string class (not sure how it's implemented)

19.1 Introduction to the Stack

- Stack: an abstract data type that holds a collection of elements of the same type.
  - The elements are accessed according to LIFO order: last in, first out
  - No random access to other elements
- Examples:
  - plates or trays in a cafeteria
  - bangles . . .

Stack Operations

- Operations:
  - push: add a value onto the top of the stack
    - make sure it's not full first.
  - pop: remove a value from the top of the stack
    - make sure it's not empty first.
  - isFull: true if the stack is currently full, i.e., has no more space to hold additional elements
  - isEmpty: true if the stack currently contains no elements
Implementing a Stack Class

- Array implementations:
  - fixed size (static) arrays: size doesn’t change
  - dynamic arrays: can resize as needed in push

- Linked List
  - grow and shrink in size as needed

```cpp
int item;
stack.push(2);
stack.push(3);
stack.push(5);
item = stack.pop(); // item is 5
item = stack.pop(); // item is 3
stack.push(10);
```

IntStack: A stack class

```cpp
class IntStack
{
private:
    static const int STACK_SIZE = 100; // The stack size
    int stackArray[STACK_SIZE]; // The stack array
    int top; // Index to the top of the stack

public:
    // Constructor
    IntStack() { top = -1; } // empty stack

    // Stack operations
    void push(int);
    int pop();
    bool isFull() const;
    bool isEmpty() const;
};
```

IntStack: push

```cpp
//@ Member function push pushes the argument onto *the stack.
void IntStack::push(int num)
{
    assert (!isFull());
    top++;
    stackArray[top] = num;
}
```

Stack Overflow: attempting to push onto a full stack.

The driver should ensure that the assert condition is always true before push is called.
IntStack: pop

int IntStack::pop()
{
    assert (!isEmpty());
    int num = stackArray[top];
    top--;
    return num;
}

Stack Underflow: attempting to pop from an empty stack.

The driver should ensure that the assert condition is always true before pop is called.

IntStack: test functions

bool IntStack::isFull() const
{
    return (top == STACK_SIZE - 1);
}

bool IntStack::isEmpty() const
{
    return (top == -1);
}

IntStack: driver

#include<iostream>
#include "IntStack.h"

int main()
{
    // set up the stack
    IntStack stack;
    stack.push(2);
    stack.push(3);
    stack.push(5);
    int x;
    x = stack.pop();
    x = stack.pop();
    stack.push(10);
    cout << x << endl;
    cout << x << endl;
}

What is output?

What is left on the stack when the driver is done?

19.4 Introduction to the Queue

- **Queue**: an abstract data type that holds a collection of elements of the same type.
  - The elements are accessed according to FIFO order: first in, first out
  - No random access to other elements

- **Examples**:
  - people in line at a theatre box office
  - print jobs sent to a (shared) printer
Queue Operations

- **enqueue**: add a value onto the rear of the queue (the end of the line)
  - make sure it’s not full first.
- **dequeue**: remove a value from the front of the queue (the front of the line) “Next!”
  - make sure it’s not empty first.
- **isFull**: true if the queue is currently full, i.e., has no more space to hold additional elements
- **isEmpty**: true if the queue currently contains no elements

Queue illustrated

Implementing a Queue Class

Same as for Stacks:
- **Array implementations**:
  - fixed size (static) arrays: size doesn’t change
  - dynamic arrays: can resize as needed in enqueue
- **Linked List**
  - grow and shrink in size as needed

Implementing a Queue class

issues using a fixed length array

- The previous illustration assumed we were using an array to implement the queue
- When an item was dequeued, the items were NOT shifted up to fill the slot vacated by dequeued item
  - why not?
- Instead, both front and rear indices move through the array.
Implementing a Queue Class

- When front and rear indices move in the array:
  - problem: rear hits end of array quickly
  - solution: “circular array”: wrap index around to front of array

```
q.enqueue(3):
| 3 | 4 | 7 | 9 | 6 |
```

```
front  rear
```

```
q.enqueue(4):
| 3 | 4 | 7 | 9 | 6 |
```

```
rear  front
```

```
q.enqueue(5):
| 3 | 4 | 7 | 9 | 6 |
```

```
rear  front
```

```
q.enqueue(2):
| 3 | 4 | 7 | 9 | 6 |
```

```
rear  front
```

```
q.enqueue(1):
| 3 | 4 | 5 | 2 | 1 |
```

```
rear  front
```

```
q.enqueue(5):
| 3 | 4 | 5 | 2 | 1 | 7 | 9 | 6 |
```

```
rear  front
```

```
q.enqueue(2):
| 3 | 4 | 5 | 2 | 1 | 7 | 9 | 6 |
```

```
rear  front
```

```
q.enqueue(1):
| 3 | 4 | 5 | 2 | 1 | 7 | 9 | 6 |
```

```
rear  front
```

```
rear = (rear + 1) % queueSize;
```

```
rear = (rear == queueSize-1) ? 0 : rear + 1;
```

**When is it full?**

```
3 4 5 2 1 rear front
```

```
rear + 1) % queueSize == front
```

```
rear + 1) % queueSize == front
```

**When is it empty?**

```
int x;
for (int i=0; i<queueSize;i++)
    x = q.dequeue();
```

```
rear = (rear + 1) % queueSize;
```

```
rear = (rear == queueSize-1) ? 0 : rear + 1;
```

```
one element left:
```

```
no elements left, front passes rear:
```

```
It’s full:
(rear+1)%queueSize==front
```

```
It’s empty:
(rear+1)%queueSize==front
```
Implementing a Queue Class

- When is it full? (rear+1)%queueSize==front
- When is it empty? (rear+1)%queueSize==front
- How do we define isFull and isEmpty?
  - Use a counter variable, numItems, to keep track of the total number of items in the queue.
- enqueue: numItems++
- dequeue: numItems--
- isEmpty is true when numItems == 0
- isFull is true when numItems == queueSize

IntQueue: a queue class

class IntQueue
{
private:
  static const int QUEUE_SIZE = 100;  //The queue size
  int queueArray[QUEUE_SIZE];  // The queue array
  int front;  // Subscript of the front elem
  int rear;  // Subscript of the rear elem
  int numItems;  // Number of items in the queue

public:
  // Constructor
  IntQueue() { front = 0;  rear = -1;  numItems = 0;  }  
  // Queue operations
  void enqueue(int);  
  int dequeue();  
  bool isEmpty();  
  bool isFull();  
};

A static queue: enqueue/dequeue

```cpp
//*****************************************************
// Enqueue inserts a value at the rear of the queue. *
//*****************************************************
void IntQueue::enqueue(int num)
{
  assert(!isFull());
  rear = (rear + 1) % QUEUE_SIZE;  //calc new position
  queueArray[rear] = num;  //insert new item
  numItems++;  //update count
}
//*****************************************************
// Dequeue removes the value at the front of the queue and returns the value. *
//*****************************************************
int IntQueue::dequeue()
{
  assert(!isEmpty());
  int result = queueArray[front];  //retrieve front item
  front = (front + 1) % QUEUE_SIZE;  //calc new position
  numItems--;  //update count
  return result;
}
```

IntQueue: test functions

```cpp
//*****************************************************
// isEmpty returns true if the queue is empty, otherwise false. *
//*****************************************************
bool IntQueue::isEmpty()
{
  return (numItems == 0);
}
//*****************************************************
// isFull returns true if the queue is full, otherwise false. *
//*****************************************************
bool IntQueue::isFull()
{
  return (numItems == QUEUE_SIZE);
}
```
# IntQueue: driver

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

#include "IntQueue.h"

int main() {
    // set up the queue
    IntQueue q;
    int item;
    q.enqueue(2);
    q.enqueue(3);
    q.enqueue(5);
    item = q.dequeue();
    item = q.dequeue();
    q.enqueue(10);
    cout << item << endl;
}
```

What is output?

What is left on the queue when the driver is done?

---

## 19.2 A Dynamic Stack Class:

Linked List implementation

```cpp
class DynIntStack {
private:
    struct Node {
        int data;
        Node* next;
    };  
    Node* head; // ptr to top

public:
    // Constructor
    DynIntStack()  {  head = NULL; }  // empty stack

    // Stack operations
    void push(int);
    int pop();
    bool isFull() const  { return false; }
    bool isEmpty() const { return head == NULL; }
};
```

head points to top element.
add and remove at front of list

---

### Push and pop from the head of the list:

```cpp
//******************************************************
// Member function push pushes the argument onto *
// the stack.                                       *
//******************************************************
void DynIntStack::push(int num) {
    assert(!isFull());

    Node *temp = new Node;   //allocate new node
    temp->data = num;
    temp->next = head;       //insert at head of list
    head = temp;
}
```

---

### Push and pop from the head of the list:

```cpp
//******************************************************
// Member function pop pops the value at the top *
// of the stack off, and returns it.                 *
//******************************************************
int DynIntStack::pop() {
    assert(!isEmpty());

    int result = head->data;  //retrieve front item
    Node * temp = head;
    head = head->next;        //head points to second item
    delete temp;              //deallocate front item
    return result;
}
```
19.5 A Dynamic Queue Class: Linked List implementation

- Use pointers `front` and `rear` to point to first and last elements of the list:

![Diagram showing pointers front and rear]

```cpp
class DynIntQueue
{
private:
    struct Node {
        int data;
        Node* next;
    }
    Node* front; // ptr to first
    Node* rear;  // ptr to last
public:
    // Constructor
    DynIntQueue() { front = NULL; rear = NULL;  }
    // Queue operations
    void enqueue(int);
    int dequeue();
    bool isFull() const  { return false; } // empty
    bool isEmpty() const { return front == NULL; }
};
```

A Dynamic Queue Class: Linked List implementation

- Enqueue at the rear, dequeue from the front:

```cpp
//****************************************************
// Enqueue inserts a value at the rear of the queue. *
//****************************************************
void DynIntQueue::enqueue(int num)
{
    assert(!isFull());
    Node *temp=new Node;     //allocate new node
    temp->data = num;
    temp->next = NULL;
    if (isEmpty())
        front = rear = temp;  //set front AND rear to node
    else {
        rear->next = temp;    //append to rear of list
        rear = temp;          //reset rear
    }
}
```

```cpp
//****************************************************
// Dequeue removes the value at the front of the queue and returns the value. *
//****************************************************
int DynIntQueue::dequeue()
{
    assert(!isEmpty());
    int value = front->data;     //retrieve front item
    Node *temp = front;
    front = front->next;         //front points to 2nd item
    delete temp;                 //deallocate removed item
    return value;
}
```