Stacks and Queues

Unit 6
Chapter 19.1-2,4-5

CS 2308
Spring 2019
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
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);

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

IntStack: A stack class

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

void IntStack::push(int num)
{
  assert (!isFull());
  top++;
  stackArray[top] = num;
}

assert will abort the program if its argument evaluates to false. It requires #include <cassert>

The driver programmer should never call push when the stack is full!
// Member function pop pops the value at the top of the stack off, and returns it as the result.

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

Stack Underflow: attempting to pop from an empty stack.

The driver programmer should never call pop when the stack is empty!

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

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

#include<iostream>
using namespace std;
#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;
    return 0;
}

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

- **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

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

To “wrap” the rear index back to the front of the array, you can use this code to increment rear during enqueue:

```c
if (rear == queueSize-1)
    rear = 0;
else
    rear = rear+1;
```

The following code is equivalent, but shorter (assuming 0 <= rear < queueSize):

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

Do the same for advancing the front index.

- When is it full?

q.enqueue(5);
q.enqueue(2);
q.enqueue(1);
  3 4 5 2 1 7 9 6

one element left:
  (rear+1)%queueSize==front

- When is it empty?

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

Note: dequeue increments front after the first one:

one element left:
  (rear+1)%queueSize==front

no elements left, front passes rear:
  (rear+1)%queueSize==front

It’s full:

It’s empty:
Implementing a Queue Class

- When is it full? \((\text{rear}+1)\%\text{queueSize}==\text{front}\)
- When is it empty? \((\text{rear}+1)\%\text{queueSize}==\text{front}\)
- 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

A static queue: enqueue/dequeue

```cpp
void IntQueue::enqueue(int num)
{
    assert(!isFull());
    rear = (rear + 1) % QUEUE_SIZE; //calc new position
    queueArray[rear] = num; //insert new item
    numItems++; //update count
}
```

```cpp
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: a queue class

```cpp
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();
};
```

IntQueue: test functions

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

```cpp
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?**

---

**A Dynamic Stack Class: Linked List implementation**

- **Push and pop from the head of the list:**

```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; }
};
```

---

**A Dynamic Stack Class: Linked List implementation**

- **Push and pop from the head of the list:**

```cpp
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:

```
struct Node {
    int data;
    Node* next;
};
```

```
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; }
        bool isEmpty() const { return front == NULL; }
};
```

A Dynamic Queue Class: Linked List implementation

- Enqueue at the rear, dequeue from the front:

```
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
    }
}
```

```
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;
}
```