Stacks and Queues

Week 9
Gaddis: Chapter 18

CS 5301
Fall 2017
Jill Seaman

Introduction to the Stack

• **Stack**: a data structure 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 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 (and return) the 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

Stack illustrated

```java
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
- Templates
  - any of the above can be implemented using templates

A static stack class

```cpp
class IntStack {
private:
    const static int STACKSIZE = 100; // The stack size
    int stackArray[STACKSIZE];  // 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;
};
```

A static stack class: push & pop

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

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

A static stack class: functions

```cpp
bool IntStack::isFull() const {
    return (top == STACKSIZE - 1);
}

bool IntStack::isEmpty() const {
    return (top == -1);
}
```
A Dynamic Stack Class:  
Linked List implementation

- Push and pop from the head of the list:

```cpp
void DynIntStack::push(int num) {
    assert(!isFull());
    Node *temp=new Node;
    temp->data = num;
    //insert at head of list
    temp->next = head;
    head = temp;
}
```

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

    Node* head;  
```

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;
    Node * temp = head;
    head = head->next;
    delete temp;
    return result;
}
```

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

    Node* head;  
```

Introduction to the Queue

- **Queue**: a data structure 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
  - restocking perishable inventory

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: Array

- 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

```
int item;
q.enqueue(2);
q.enqueue(3);
q.enqueue(5);
item = q.dequeue(); //item is 2
item = q.dequeue(); //item is 3
q.enqueue(10);
```

Implementing a Queue: Array

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

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

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

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

- Do the same for advancing the front index.

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

- Templates
  - any of the above can be implemented using templates
Implementing a Queue: Array

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

![Queue full diagram]

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

![Queue empty diagram]

- Don’t use rear and front to determine if the queue is full or empty!!

A static queue class

```cpp
class IntQueue {
private:
    const static int QUEUESIZE = 100; // capacity of queue
    int queueArray[QUEUESIZE]; // The queue array
    int front; // Subscript of the queue front
    int rear; // Subscript of the queue rear
    int numItems; // Number of items in the queue
public:
    // Constructor
    IntQueue() { front = 0; rear = -1; numItems = 0; } // Constructor

    // Queue operations
    void enqueue(int); // Enqueue inserts a value at the rear of the queue.
    int dequeue(); // Dequeue removes the value at the front of the queue and returns the value.
    bool isEmpty() const { // isEmpty returns true if the queue is empty.
        return (numItems == 0); } // isEmpty returns true if the queue is empty.
    bool isFull() const { // isFull returns true if the queue is full.
        return (numItems == QUEUESIZE); } // isFull returns true if the queue is full.

    // Functions
    void enqueue(int num) { assert(!isFull());
        rear = (rear + 1) % QUEUESIZE;
        queueArray[rear] = num;
        numItems++;
    }

    int dequeue() { assert(!isEmpty());
        int result = queueArray[front];
        front = (front + 1) % QUEUESIZE;
        numItems--;
        return result;
    }
};
```
A Dynamic Queue Class: Linked List implementation

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

```
  front   rear
  +   +   +   NULL
```

A Dynamic Queue Class: Linked List implementation

- Enqueue at the rear of the list, dequeue from the front:

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

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

Node* front; // ptr to first
Node* rear;  // ptr to last

A Dynamic Queue Class: Linked List implementation

- Enqueue at the rear of the list, dequeue from the front:

```
//****************************************************
// Dequeue removes the value at the front of the queue and returns the value. *
//****************************************************
int DynIntQueue::dequeue() {
    assert(!isEmpty());
    int value = front->data;
    // remove the first node (front)
    Node *temp = front;
    front = front->next;
    delete temp;
    if (front==NULL) rear = NULL;
    return value;
}
```

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

Node* front; // ptr to first
Node* rear;  // ptr to last

A Dynamic Queue Class: Linked List implementation

- Sample Problem

- See Gaddis 7.12 (or google) for a description of the STL vector type.

- Implement the following `IntStack` using a vector (instead of an array) to store the elements.

```
class IntStack {
    private:
        struct Node {
            int data;
            Node* next;
        }; // ptr to first
    vector<int> stack; // store elements in a vector
    public:
        // Constructor
        IntStack() { } // stack vector is already empty
        // Stack operations
        void push(int);
        int pop();
        bool isFull() const;
        bool isEmpty() const;
};
```