Revised: 6/9/2024

Lists II

Michael C. Hackett

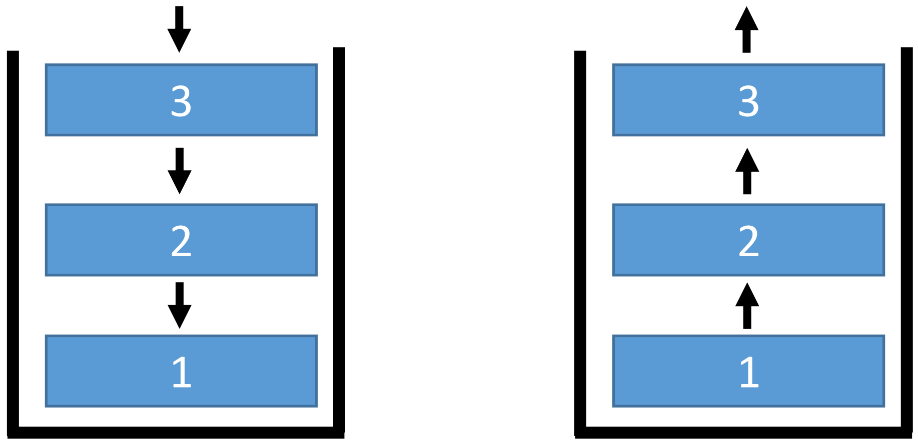
Assistant Professor, Computer Science

Lecture Topics

- Stacks
  - Array-Based Stacks
  - List-Based Stacks
- Queues
  - Circular Queues
  - Deques

Stacks

- A stack is a linear data structure that operates on the FILO principle.
  - **FILO** – First In Last Out
- Items are added ("pushed") onto the top of a stack
- Items are retrieved ("popped") from the top of a stack



Stacks

- Stacks can be created with an array or a singly linked list.
  - Arrays would give the stack an implicit size limit.
  - We’d have to explicitly set a size limit for a list-based stack.

Array-Based Stacks

- An array-based stack will need:
  - An array
  - An int that represents the array’s length (the stack’s capacity)
  - An int that keeps track of the index that is the top of the stack

```
class ArrayStack {
    private int[] a;        //The array
    private int max;        //The capacity
    private int top;        //The index that is the top of the stack
}
```

Array-Based Stacks

- The stack's constructor will:
  - Accept an int argument that sets the capacity
  - Set the max field with this value
  - Create an array of that length
  - Set top to -1 (signifying the stack is empty)

Array-Based Stacks

```
class ArrayStack {
    private int[] a;        //The array
    private int max;        //The capacity
    private int top;        //The index that is the top of the stack

    public ArrayStack(int sizeIn) {
        max = sizeIn;
        a = new int[max];
        top = -1;
    }
}
```

Array-Based Stacks

- New items are added to the stack starting at index 0 and working its way to the end of the array.

```
public void push(int newData) {
    if(top >= max-1) {
        throw new StackOverflowError("Stack Overflow");
    }
    a[++top] = newData;
}
```

Array-Based Stacks

- Items are retrieved from the stack starting at “top” and working its way to the beginning of the array.

```
public int pop() {
    if(top < 0) {
        throw new EmptyStackException("Stack Underflow");
    }
    return a[top--];
}
```

Array-Based Stacks

- The “pop” method just shown retrieves and removes the data on the top of the stack.
- It’s common to have a method that simply retrieve the data at the top (“peek”), but not remove it.

```
public int peek() {
    if(top < 0) {
        throw new EmptyStackException("Stack Underflow");
    }
    return a[top];
}
```

Array-Based Stacks

- Getting the capacity of the stack is as easy as returning the value of max
- Getting the size of the stack (how many things are in it) is as easy as returning top + 1 (need to account for index 0)

```
public int capacity() {
    return max;
}

public int size() {
    return top + 1;
}
```

Array-Based Stacks

- Simple logic can determine if a stack is full or empty.

```
public boolean isFull() {
    return top == max-1 ? true : false;
}

public boolean isEmpty() {
    return top < 0 ? true : false;
}
```

List-Based Stacks

```
class Node {
    int data;                          //Data stored in the node
    Node next;                         //Reference to the next node
}

class ListStack {
    private Node top;          //Reference to the top of the stack
    private int size;          //Keeps track of how many nodes are in the stack

    public Stack () {
        top = null;
        size = 0;
    }
}
```

List-Based Stacks

- New items are added to the top of the stack.

```
public void push(int newData) {
    Node temp = new Node();
    temp.data = newData;
    temp.next = top;
    top = temp;
    size++;
}
```

List-Based Stacks

- Items are retrieved from the top/head of the stack.

```
public int pop() {
    if(size == 0) {
        throw new EmptyStackException("Stack Underflow");
    }
    int data = top.data;
    top = top.next;
    size--;
    return data;
}
```

List-Based Stacks

- Peeking at the top of the stack:

```
public int peek() {
    if(size == 0) {
        throw new EmptyStackException("Stack Underflow");
    }
    return top.data;
}
```

List-Based Stacks

- Getting the size of the stack and if it is empty:

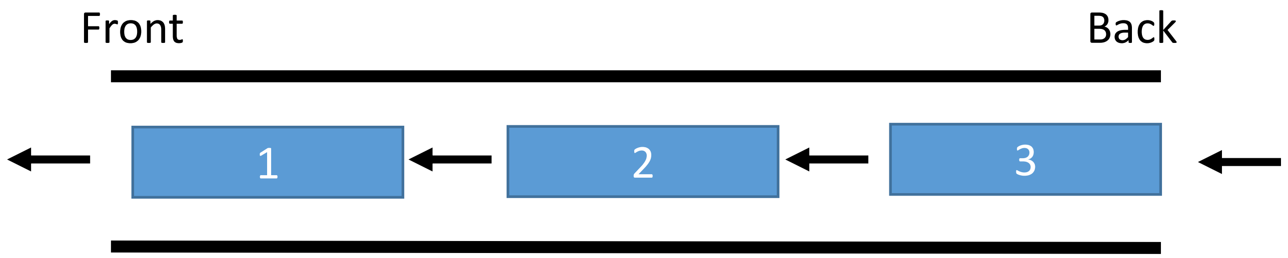
```
public int getSize() {
    return size;
}

public boolean isEmpty() {
    return size == 0 ? true : false;
}
```

- Not concerned with capacity or isFull because there isn't an explicit capacity.

Queues

- A queue is a linear data structure that operates on the FIFO principle.
  - **FIFO** – First In First Out
- Items are added (“pushed”) to the back of a queue
- Items are retrieved (“popped”) from the front of a queue



Queues

- Queues can be created with an array or a singly linked list.
  - Arrays would give the queue an implicit size limit.
  - We'd have to explicitly set a size limit for a list-based queue.

- We'll see an implementation of a normal queue with a linked list.
  - We'll use an array-based queue for something different

Queues

```
class Queue {

    class Node {
        int data;                   //Data stored in the node
        Node next;                  //Reference to the next node
    }

    private Node front;             //Reference to the front/head of the queue
    private Node back;              //Reference to the back/tail of the queue
    private int size;               //Keeps track of how many nodes are in the queue

    public Queue() {
        front = null;
        back = null;
        size = 0;
    }

}
```

Queues

- New items are added to the back of the queue.

```
public void push(int newData) {
    Node temp = new Node();
    temp.data = newData;
    temp.next = null;
    if(size == 0) {
       front = temp;
       back = temp;
    }
    else {
       back.next = temp;
       back = back.next;
    }
    size++;
}
```

Queues

- Items are retrieved from the front of the queue.

```
public int pop() {
    if(size == 0) {
        throw new RuntimeException("Queue is empty");
    }
    int data = front.data;
    front = front.next;
    size--;
    if(size == 0) {
        back = null;
    }
    return data;
}
```

Queues

- Peeking at the front of the queue:

```
public int peek() {
    if(size == 0) {
        throw new RuntimeException("Queue is empty");
    }
    return front.data;
}
```

Queues

- Getting the size of the queue and if it is empty:

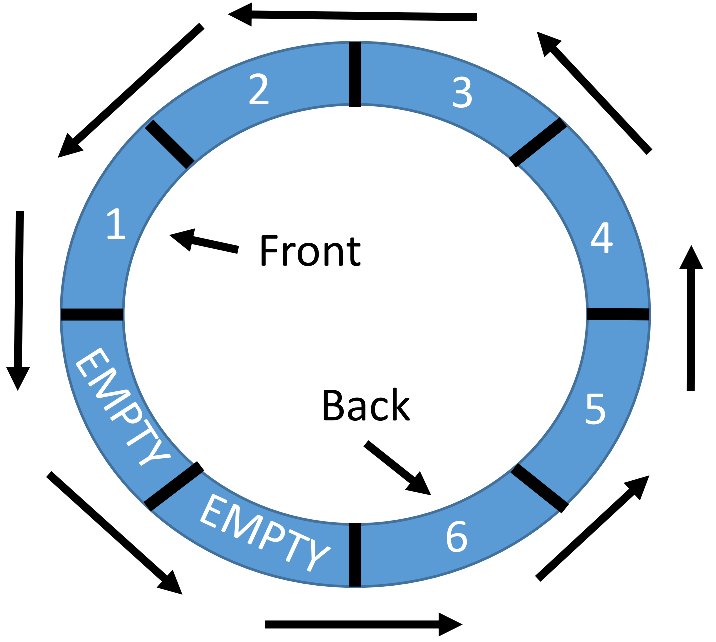
```
public int getSize() {
    return size;
}

public boolean isEmpty() {
    return size == 0 ? true : false;
}
```

- Not concerned with capacity or isFull because there isn't an explicit capacity.

Circular Queues

- A circular queue is a linear data structure that operates on the FIFO principle, but the end of the queue is linked to the beginning of the queue.
  - Sometimes called a *ring buffer*
  - “Front” and “Back” are relative



Circular Queues

```
class CQueue {

    private int front;              //Array index of the front
    private int back;               //Array index of the back
    private int[] a;                //The array that will store the data
    private int max;                //The capacity of the array (and thus the queue)
    private int count;              //Keeps track of how many items are in the queue

    public CQueue(int size) {
        count = 0;                  //Queue is empty to start
        max = size;                 //Set capacity
        a = new int[max];           //Create the array
        front = -1;                 //Indicates queue is empty
        back = -1;                  //Indicates queue is empty
    }

}
```

Circular Queues

- To add a new item, we need to check…
  - If the queue is full
  - If the queue is empty
  - If we are at the end of the array and need to loop back around to 0

- If all those conditions are false, we simply increment back by 1
- We place the new value at the index now assigned to “back”
- Increment count by one

Circular Queues

```
public void push(int newData) {
    if(count == max) {                                  //Check if full
        throw new RuntimeException("Queue is full");
    }
    else if(front == -1) {                              //Check if empty
        front = 0;
        back = 0;
    }
    else if(back == max-1) {                            //Check if it needs to loop around
        back = 0;
    }
    else {
        back++;                                         //Add one to back
    }
    a[back] = newData;
    count++;
}
```

Circular Queues

- To retrieve an item, we need to…
  - Get the data at the index assigned to “front”
  - Check if removing this item will make the queue empty
    - Set front and back to -1
  - Check if front was at the end of the array and needs to loop back around to 0
  - If both conditions are false, simply increment front by one

- Decrease the count
- Return the value/data

Circular Queues

```
public int pop() {
    if(front == -1) {                                       //Check if empty
        throw new RuntimeException("Queue is empty");
    }
    int temp = a[front];                                    //Get the data
    if(front == back) {                                     //Check if it is now empty
        front = -1;
        back = -1;
    }
    else if(front == max-1) {                               //Check if it needs to loop around
        front = 0;
    }
    else {
        front++;                                            //Add one to front
    }
    count--;
    return temp;
}
```

Circular Queues

- Peeking at the front of the queue:

```
public int peek() {
    if(front == -1) {
        throw new RuntimeException("Queue is empty");
    }
    return a[front];
}
```

Circular Queues

- Getting the capacity and size of the queue:

```
public int capacity() {
    return max;
}

public int size() {
    return count;
}
```

Circular Queues

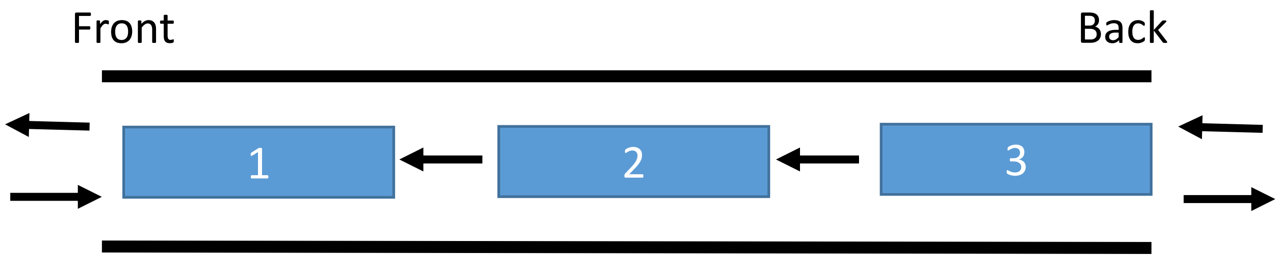
- If the queue is full or empty:

```
public boolean isFull() {
    return count == max ? true : false;
}

public boolean isEmpty() {
    return front == -1 ? true : false;
}
```

Deques

- A deque (pronounced *deck)* is a **d**ouble **e**nded **que**ue.
- Items can be added ("pushed") to the end *and beginning*
- Items can be retrieved ("popped") from the beginning *and end*
- Normally implemented using a doubly linked list



Deques

```
class Deque {
    private DLinkedList dequeList;        //Linked list used to store the deque's data

    public Deque() {
        dequeList = new DLinkedList();
    }
}
```

Deques

- Adding new items to the back of the queue.

```
public void pushBack(int newData) {
    dequeList.moveToEnd();     //Set current reference to the back
    dequeList.push(newData);   //Push the new data
}
```

- Adding new items to the front of the queue.

```
public void pushFront(int newData) {
    dequeList.moveToStart();     //Set current reference to the front
    dequeList.insert(newData);   //Insert the new data (to the front)
}
```

Deques

- Retrieving/Removing items from the back of the queue.

```
public int popBack() {
    if(dequeList.getLength() == 0) {
        throw new RuntimeException("Deque is empty.");       //Deque is empty
    }
    dequeList.moveToEnd();
    int data = dequeList.get();                               //Get the data at the back
    dequeList.remove();                                       //Remove it
    return data;                                              //Return the data
}
```

Deques

- Retrieving/Removing items from the front of the queue.

```
public int popFront() {
    if(dequeList.getLength() == 0) {
        throw new RuntimeException("Deque is empty.");       //Deque is empty
    }
    dequeList.moveToStart();
    int data = dequeList.get();                              //Get the data at the front
    dequeList.remove();                                      //Remove it
    return data;                                             //Return the data
}
```

Deques

- Peeking at the back and front of the deque:

```
public int peekBack() {
    if(dequeList.getLength() == 0) {
        throw new RuntimeException("Deque is empty.");       //Deque is empty
    }
    dequeList.moveToEnd();
    return dequeList.get();        //Return the data at the back of the deque
}


public int peekFront() {
    if(dequeList.getLength() == 0) {
        throw new RuntimeException("Deque is empty.");       //Deque is empty
    }
    dequeList.moveToStart();
    return dequeList.get();        //Get the data at the front of the deque
}
```

Deques

- Getting the size of the deque and if it is empty:

```
public int size() {
    return dequeList.getLength();
}

public boolean isEmpty() {
    return dequeList.getLength() == 0 ? true : false;
}
```

- Not concerned with capacity or isFull because there isn’t an explicit capacity.