Revised: 1/3/2020

Stacks, Queues, and Deques

Michael C. Hackett

Computer Science Department

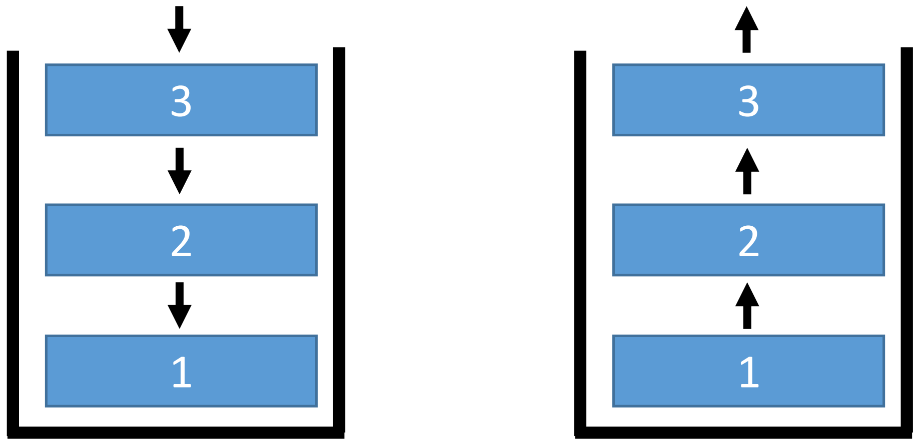
Community College of Philadelphia

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 Stack {
    private:
        int *a;     //The array
        int max;    //The capacity
        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 Stack {
    private:
        int *a;
        int max;
        int top;

    public:
        Stack(int sizeIn) {
            max = sizeIn;
            a = int[sizeIn];
            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.

```
void push(int newData) {
    if(top >= max-1) {
        __throw_overflow_error("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.

```
int pop() {
    if(top < 0) {
        __throw_underflow_error("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, but not remove it.

```
int peek(int newData) {
    if(top < 0) {
        __throw_underflow_error("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 (have to account for index 0)

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

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

Array-Based Stacks

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

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

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

List-Based Stacks

```
struct Node {
    int data;                       //Data stored in the node
    Node *next;                     //Pointer to the next node
};

class Stack {
    private:
        int count;                  //Keeps track of how many nodes are in the stack
        Node *head;                 //Pointer to the head/top of the stack

    public:
        StackList() {
            count = 0;              //Stack is empty to start
            head = NULL;
        }
};
```

List-Based Stacks

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

```
void push(int newData) {
    Node *temp = new Node;
    temp->data = newData;
    temp->next = head;
    head = temp;
    count++;
}
```

List-Based Stacks

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

```
int pop() {
    if(head == NULL) {
        __throw_underflow_error("Stack Underflow");
    }
    int tempData = head->data;
    Node *tempNode = head;
    head = head->next;
    free(tempNode);
    count--;
    return tempData;
}
```

List-Based Stacks

- Peeking at the top of the stack:

```
int peek(int newData) {
    if(head == NULL) {
        __throw_underflow_error("Stack Underflow");
    }
    return head->data;
}
```

List-Based Stacks

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

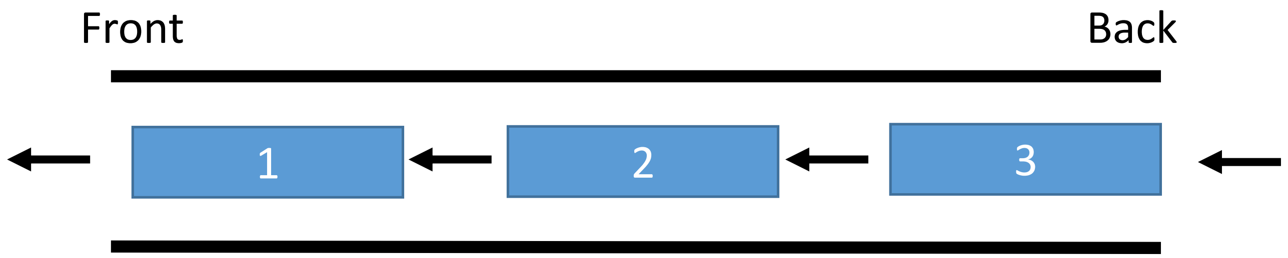
```
int size() {
    return count;
}

bool isEmpty() {
    return head == NULL ? 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.

- All operations are O(1)

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

Queues

```
struct Node {
    int data;                       //Data stored in the node
    Node *next;                     //Pointer to the next node
};

class Queue {
    private:
        int count;                  //Keeps track of how many nodes are in the queue
        Node *front;                //Pointer to the front/head of the queue
        Node *back;                 //Pointer to the back/tail of the queue

    public:
        Queue() {
            count = 0;              //Queue is empty to start
            front = NULL;
            back = NULL;
        }
};
```

Queues

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

```
void push(int newData) {
    Node *temp = new Node;
    temp->data = newData;
    temp->next = NULL;
    if(back == NULL) {
       front = temp;
    }
    else {
       back->next = temp;
    }
     back = temp;
    count++;
}
```

Queues

- Items are retrieved from the front of the queue.

```
int pop() {
    if(front == NULL) {
        __throw_underflow_error("Queue is empty");
    }
    int tempData = front->data;
    Node *tempNode = front->next;
    free(front);
    front = tempNode;
    count--;
    return tempData;
}
```

Queues

- Peeking at the front of the queue:

```
int peek() {
    if(front == NULL) {
        __throw_underflow_error("Queue is empty");
    }
    return front->data;
}
```

Queues

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

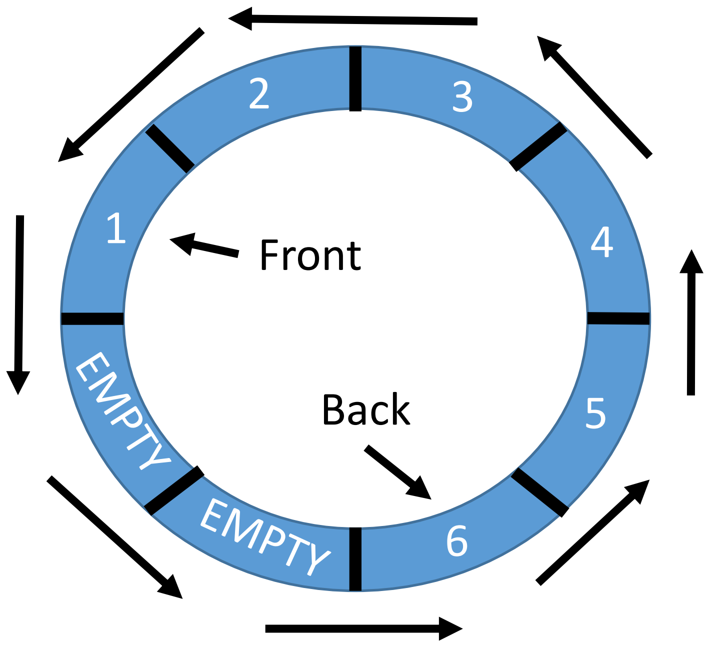
```
int size() {
    return count;
}

bool isEmpty() {
    return front == NULL ? 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 count;                  //Keeps track of how many items are in the queue
        int front;                  //Array index of the front
        int back;                   //Array index of the back
        int *a;                     //The array that will store the data
        int max;                    //The capacity of the array (and thus the queue)

    public:
        CQueue(int sizeIn) {
            count = 0;              //Queue is empty to start
            max = sizeIn;           //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 of 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

```
void push(int newData) {
    if(count == max) {                                  //Check if full
        __throw_overflow_error("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

```
int pop() {
    if(front == -1) {                                       //Check if empty
        __throw_underflow_error("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:

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

Circular Queues

- Getting the capacity and size of the queue:

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

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

Circular Queues

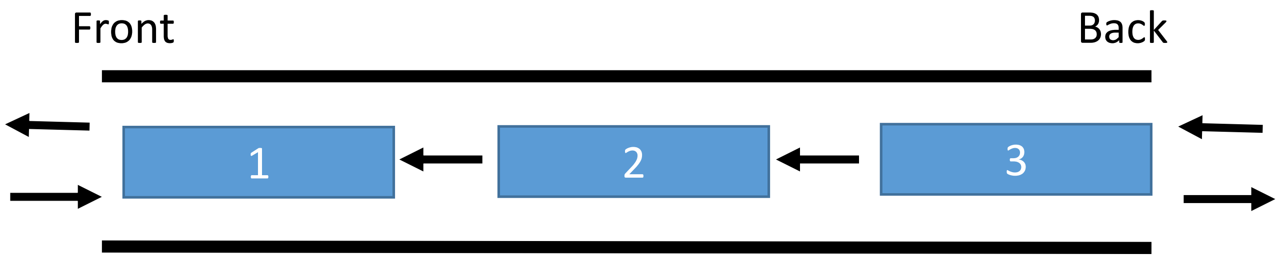
- If the queue is full or empty:

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

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

Deques

- A deque ("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

```
struct Node {
    int data;                           //Data stored in the node
    Node *next;                         //Pointer to the next node
    Node *previous;                     //Pointer to the previous node
};

class Deque {
    private:
        int count;                      //Keeps track of how many nodes are in the deque
        Node *front;                    //Pointer to the front/head of the deque
        Node *back;                     //Pointer to the back/tail of the deque

    public:
        Deque() {
            count = 0;                  //Deque is empty to start
            front = NULL;
            back = NULL;
        }
};
```

Deques

- Adding new items to the back of the queue.

```
void push_back(int newData) {
    Node *temp = new Node;
    temp->data = newData;
    temp->next = NULL;
    if(back == NULL) {            //Deque is empty
       front = temp;
       back = temp;
       temp->previous = NULL;
    }
    else {
       temp->previous = back;
       back->next = temp;
       back = temp;
    }
    count++;
}
```

Deques

- Adding new items to the front of the queue.

```
void push_front(int newData) {
    Node *temp = new Node;
    temp->data = newData;
    temp->previous = NULL;
    if(front == NULL) {            //Deque is empty
       front = temp;
       back = temp;
       temp->next = NULL;
    }
    else {
       temp->next = front;
       front->previous = temp;
       front = temp;
    }
    count++;
}
```

Deques

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

```
int pop_front() {
    if(front == NULL) {
        __throw_underflow_error("Deque is empty");
    }
    int tempData = front->data;
    Node *tempNode = front->next;
    free(front);
    if(tempNode != NULL) {
        tempNode->previous = NULL;
    }
    else {
        back = tempNode;
    }
    front = tempNode;
    count--;
    return tempData;
}
```

Deques

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

```
int pop_back() {
    if(back == NULL) {
        __throw_underflow_error("Deque is empty");
    }
    int tempData = back->data;
    Node *tempNode = back->previous;
    free(back);
    if(tempNode != NULL) {
        tempNode->next = NULL;
    }
    else {
        front = tempNode;
    }
    back = tempNode;
    count--;
    return tempData;
}
```

Deques

- Peeking at the front and back of the deque:

```
int peek_front() {
    if(front == NULL) {
        __throw_underflow_error("Deque is empty");
    }
    return front->data;
}

int peek_back() {
    if(back == NULL) {
        __throw_underflow_error("Deque is empty");
    }
    return back->data;
}
```

Deques

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

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

bool isEmpty() {
    return front == NULL ? true : false;
}
```

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