Big-O, Stacks, and Queues

Objectives

1. Undstand the utility of big-O notation
2. Compare ArrayList and LinkedList on the basis of big-O notation
3. Understand the distinction between stacks and queues.
4. Use the Queue interface and Stack and their common methods in Java.
5. Use a stack to build an RPN calculator.
6. Use a queue to simulate round robin process scheduling.

Algorithmic Complexity

1. **Definition:** A way of defining the number of "operations" and "amount of memory" an algorithm (method) will take depending on the size of the input (N).

Algorithmic Complexity

1. **Definition:** A way of defining the number of "operations" and "amount of memory" an algorithm (method) will take depending on the size of the input (N).
2. We consider two modes of complexity:
   - **Time Complexity:** The number of operations we can expect an operation to take, in the worst case.
   - **Space Complexity:** The amount of memory we can expect an operation to take, in the worst case.

Algorithmic Complexity

1. **Definition:** A way of defining the number of "operations" and "amount of memory" an algorithm (method) will take depending on the size of the input (N).
2. We consider two modes of complexity:
   - **Time Complexity:** The number of operations we can expect an operation to take, in the worst case.
   - **Space Complexity:** The amount of memory we can expect an operation to take, in the worst case.
3. Today we will focus on time complexity.

Algorithmic Complexity

1. **Definition:** A way of defining the number of “operations” and “amount of memory” an algorithm (method) will take depending on the size of the input (N).
2. We consider two modes of complexity:
   - **Time Complexity:** The number of operations we can expect an operation to take, in the worst case.
   - **Space Complexity:** The amount of memory we can expect an operation to take, in the worst case.
3. Today we will focus on time complexity.
4. Denoted as $O(\text{input space})$ to represent the upper bound (i.e., the worst case).

Analyzing Algorithms

```
public void printList(Integer item){
    System.out.println(item);
}
```

```
public void printList(List<Integer> lst){
    for(int i = 0; i < lst.size(); i++){
        System.out.println(lst.get(i));
    }
}
```

```
public void multiplyAndPrint(Integer item1, Integer item2){
    int result = item1 * item2;
    System.out.println(result);
}
```

What is the maximum number of operations each method will perform? What is the minimum?

An example of constant time

```java
public void multiplyAndPrint(Integer item1, Integer item2){
    int result = item1 * item2;
    System.out.println(result);
}
```

```java
public void pow(Integer item1, Integer item2){
    int result = item1 * item2;
    System.out.println(result);
}
```

```java
public void multiplyAndPrint(Integer item1, Integer item2){
    int result = item1 * item2;
    System.out.println(result);
}
```

All of the above are treated as $O(1)$ since they don't increase with input size.

Analyzing Algorithms

```java
public void printList(List<Integer> lst){
    for(int i = 0; i < lst.size(); i++){
        System.out.println(lst.get(i));
    }
}
```

```java
public void printList(List<Integer> lst){
    for(int i = 0; i < lst.size(); i++){
        for(int j = 0; j < lst.size(); j++){
            System.out.println(lst.get(i) * lst.get(j));
        }
    }
}
```

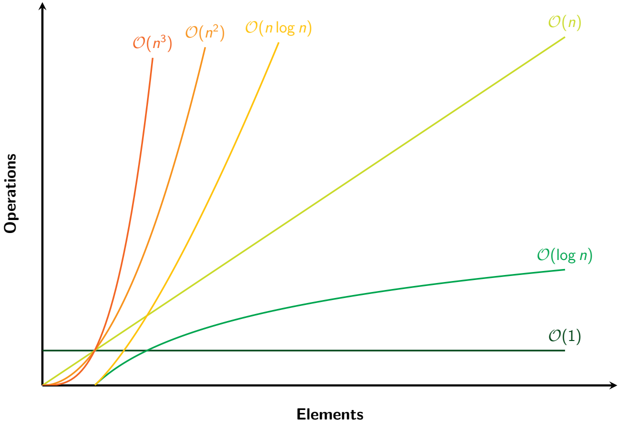
If one for loop (the first one) is $O(N)$, what is a nested for loop (the second one)?

Analyzing Algorithms

```java
public void printList(List<Integer> lst){
    for(int i = 0; i < lst.size(); i++){
        System.out.println(lst.get(i));
    }
    for(int i = 0; i < lst.size(); i++){
        System.out.println(lst.get(i));
    }
}
```

So is two for loops, unnested, $O(2N)$?.

Examples


$\mathcal{O}(n^3)$
$\mathcal{O}(n^2)$
$\mathcal{O}(n \log n)$
$\mathcal{O}(n)$
$\mathcal{O}(\log n)$
$\mathcal{O}(1)$
Operations
Elements

ArrayList vs LinkedList - Searching

```java
public void search(ListNode<E> head, E data){
    ListNode<E> tmp = head;
    while(tmp != null && !tmp.data.equals(data)){
        tmp = tmp.next;
    }
    return tmp;
}
```

1. **LinkedList:** O(N) since we have to traverse to find the item.
2. **ArrayList:** O(N) since we also have to traverse the item.

ArrayList vs LinkedList - Getting Item at Index

1. What is the complexity of getting an item at an arbitrary position in a LinkedList? And an ArrayList?

ArrayList vs LinkedList - Getting Item at Index

1. What is the complexity of getting an item at an arbitrary position in a LinkedList? And an ArrayList?
    1. **LinkedList:** O(N) since we have to traverse to find the item.
    2. **ArrayList:** O(1) since we can index.

ArrayList vs LinkedList - Getting Item at Index

1. What is the complexity of getting an item at an arbitrary position in a LinkedList? And an ArrayList?
    1. **LinkedList:** O(N) since we have to traverse to find the item.
    2. **ArrayList:** O(1) since we can index.
2. What would the complexity be for getFront() or getEnd() be for a LinkedList? How would this depend on whether we are keeping track of the tail?

ArrayList vs LinkedList - Getting Item at Index

1. What is the complexity of getting an item at an arbitrary position in a LinkedList? And an ArrayList?
    1. **LinkedList:** O(N) since we have to traverse to find the item.
    2. **ArrayList:** O(1) since we can index.
2. What would the complexity be for getFront() or getEnd() be for a LinkedList? How would this depend on whether we are keeping track of the tail?
    1. **LinkedList wo/ tail:** $O(1)$ to get head and $O(N)$ to get last node.
    2. **LinkedList w/ tail:** $O(1)$ for each.

ArrayList vs LinkedList - Insertion/Removal

1. Adding/removing in the front?

ArrayList vs LinkedList - Insertion/Removal

1. Adding/removing in the front?
    1. **LinkedList:** We track the head so we can just add on to the front, O(1).
    2. **ArrayList:** Finding the position is O(1) and copy/shift is O(N).

ArrayList vs LinkedList - Insertion/Removal

1. Adding/removing in the front?
    1. **LinkedList:** We track the head so we can just add on to the front, O(1).
    2. **ArrayList:** Finding the position is O(1) and copy/shift is O(N).
2. Adding/removing in the middle?

ArrayList vs LinkedList - Insertion/Removal

1. Adding/removing in the front?
    1. **LinkedList:** We track the head so we can just add on to the front, O(1).
    2. **ArrayList:** Finding the position is O(1) and copy/shift is O(N).
2. Adding/removing in the middle?
    1. **LinkedList:** Finding the position(s) is O(N).
    2. **ArrayList:** Finding the position is fast (O(1)) buth copy/shift makes this O(N).
    3. So both are O(N).

ArrayList vs LinkedList - Insertion/Removal

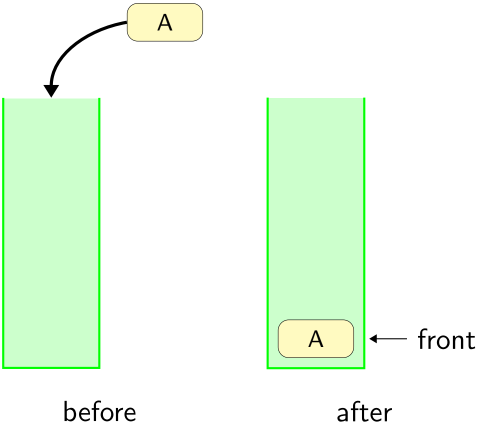
1. Adding/removing in the front?
    1. **LinkedList:** We track the head so we can just add on to the front, O(1).
    2. **ArrayList:** Finding the position is O(1) and copy/shift is O(N).
2. Adding/removing in the middle?
    1. **LinkedList:** Finding the position(s) is O(N).
    2. **ArrayList:** Finding the position is fast (O(1)) buth copy/shift makes this O(N).
    3. So both are O(N).
3. Adding/removing to the end?

ArrayList vs LinkedList - Insertion/Removal

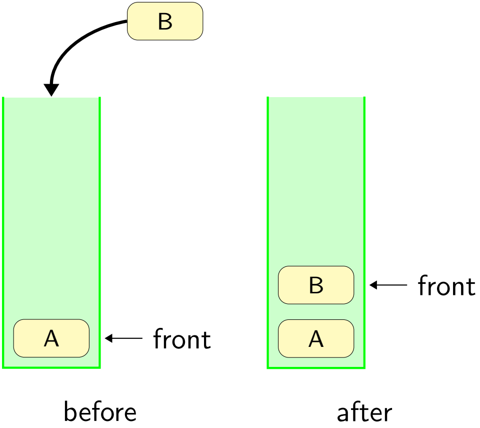
1. Adding/removing in the front?
    1. **LinkedList:** We track the head so we can just add on to the front, O(1).
    2. **ArrayList:** Finding the position is O(1) and copy/shift is O(N).
2. Adding/removing in the middle?
    1. **LinkedList:** Finding the position(s) is O(N).
    2. **ArrayList:** Finding the position is fast (O(1)) buth copy/shift makes this O(N).
    3. So both are O(N).
3. Adding/removing to the end?
    1. **LinkedList (wo/tail):** O(N) since we have to search for the end.
    2. **LinkedList (w/tail):** O(1) for adding since we track the tail.
    3. **ArrayList:** Finding the position is O(1) and since it's at the end

Stacks

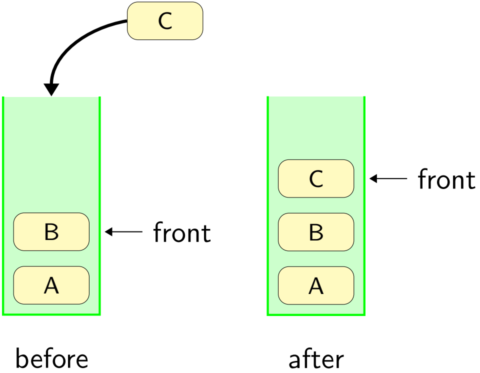
1. Last in, First out (LIFO) data structure.
2. Stack is a class Java
3. Uses the following operations:
    1. push to add to the top of the stack.
    2. pop to remove the element at the top of the stack.



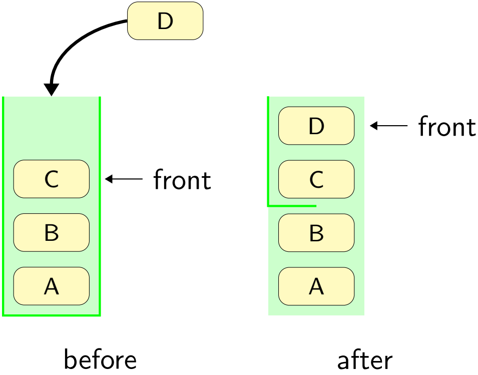
```
Stack<Integer> nums = new Stack<>();
nums.push("A")
```



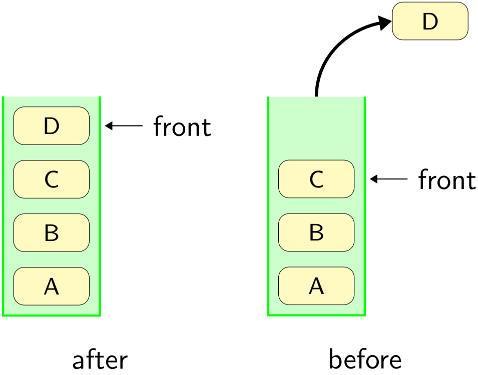
```
Stack<Integer> nums = new Stack<>();
nums.push("A")
nums.push("B")
```



```
Stack<Integer> nums = new Stack<>();
nums.push("A")
nums.push("B")
nums.push("C")
```



```
Stack<Integer> nums = new Stack<>();
nums.push("A")
nums.push("B")
nums.push("C")
nums.push("D")
```



```
Stack<Integer> nums = new Stack<>();
nums.push("A")
nums.push("B")
nums.push("C")
nums.push("D")
nums.pop()
```

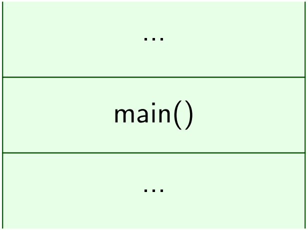
Q: Where is this used? A: Call Stacks

```java
public class PowerClass {

   public static int mult(int times, int val){
      int product = 0;
      for(int i = 0; i < times; i++){
         product += val;
      }
      return product;
   }

   public static int pow(int num, int raise){
      int total = 1;
      for(int i = 0; i < raise; i ++){
         total = mult(total, num);
      }
      return total;
   }

   public static void main(String[] args) {
      pow(2, 5);
   }
}
```


...

main()

...

Our program starts at main.

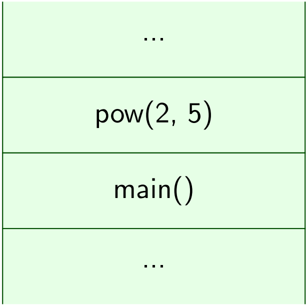
Q: Where is this used? A: Call Stacks

```java
public class PowerClass {

   public static int mult(int times, int val){
      int product = 0;
      for(int i = 0; i < times; i++){
         product += val;
      }
      return product;
   }

   public static int pow(int num, int raise){
      int total = 1;
      for(int i = 0; i < raise; i ++){
         total = mult(total, num);
      }
      return total;
   }

   public static void main(String[] args) {
      pow(2, 5);
   }
}
```


...

pow(2, 5)

main()

...

Our program starts at the pow method is then called an placed on the call stack.

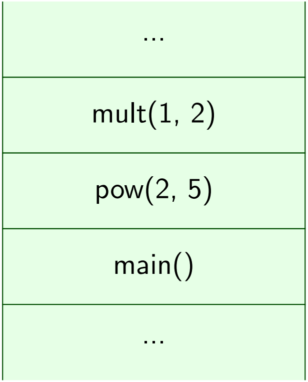
Q: Where is this used? A: Call Stacks

```
public class PowerClass {

   public static int mult(int times, int val){
      int product = 0;
      for(int i = 0; i < times; i++){
         product += val;
      }
      return product;
   }

   public static int pow(int num, int raise){
      int total = 1;
      for(int i = 0; i < raise; i ++){
         total = mult(total, num);
      }
      return total;
   }

   public static void main(String[] args) {
      pow(2, 5);
   }
}
```



...

mult(1, 2)

pow(2, 5)

main()

...

The pow method then calls the mutl method.

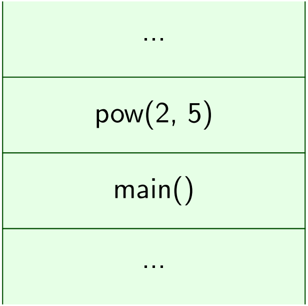
Q: Where is this used? A: Call Stacks

```
public class PowerClass {

   public static int mult(int times, int val){
      int product = 0;
      for(int i = 0; i < times; i++){
         product += val;
      }
      return product;
   }

   public static int pow(int num, int raise){
      int total = 1;
      for(int i = 0; i < raise; i ++){
         total = mult(total, num);
      }
      return total;
   }

   public static void main(String[] args) {
      pow(2, 5);
   }
}
```



...

pow(2, 5)

main()

...

Once the call to mult is completed it is “popped” off of the stack and we return to where we left off in pow.

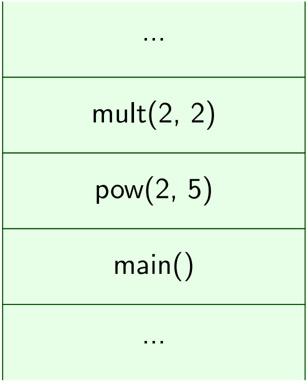
Q: Where is this used? A: Call Stacks

```
public class PowerClass {

   public static int mult(int times, int val){
      int product = 0;
      for(int i = 0; i < times; i++){
         product += val;
      }
      return product;
   }

   public static int pow(int num, int raise){
      int total = 1;
      for(int i = 0; i < raise; i ++){
         total = mult(total, num);
      }
      return total;
   }

   public static void main(String[] args) {
      pow(2, 5);
   }
}
```


...

mult(2, 2)

pow(2, 5)

main()

...

We then call the mult method again.

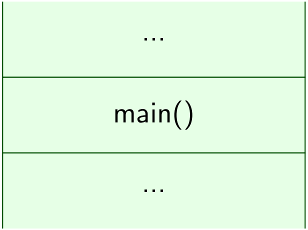
Q: Where is this used? A: Call Stacks

```
public class PowerClass {

  public static int mult(int times, int val){
    int product = 0;
    for(int i = 0; i < times; i++){
      product += val;
    }
    return product;
  }

  public static int pow(int num, int raise){
    int total = 1;
    for(int i = 0; i < raise; i ++){
      total = mult(total, num);
    }
    return total;
  }

  public static void main(String[] args) {
    pow(2, 5);
  }
}
```



We continue this until pow is complete and then pop it off the stack.

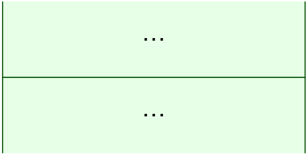
Q: Where is this used? A: Call Stacks

```
public class PowerClass {

   public static int mult(int times, int val){
      int product = 0;
      for(int i = 0; i < times; i++){
         product += val;
      }
      return product;
   }

   public static int pow(int num, int raise){
      int total = 1;
      for(int i = 0; i < raise; i ++){
         total = mult(total, num);
      }
      return total;
   }

   public static void main(String[] args) {
      pow(2, 5);
   }
}
```



main has finished so that is popped as well and the program terminates.

Worksheet: Stack Practice

Off to work on the worksheet to play with stacks.

RPN Calculator

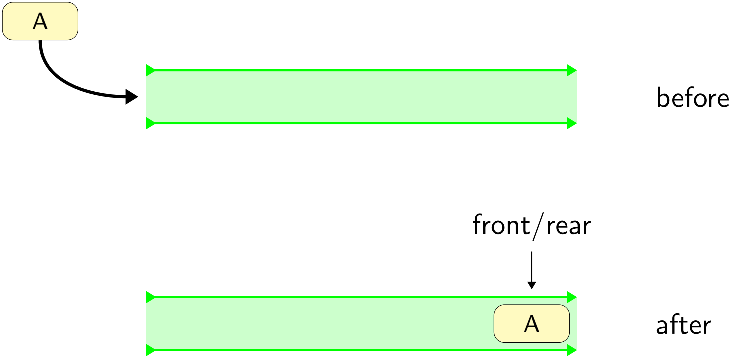
Look at each element in the list and, at each stage:

1. If an element is an operation (i.e., +/-):
    1. Pop two numbers from the stack
    2. Perform the operation
    3. Push the result onto the stack
2. Otherwise, it must be a string representation of a number so:
    1. Convert it to an ‘Integer‘
    2. Push it to the stack

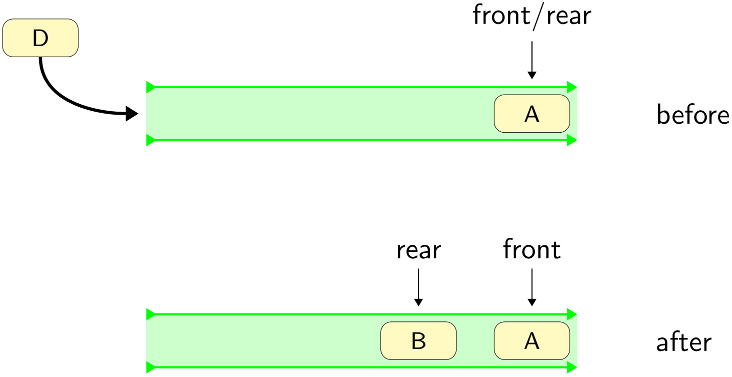
Worksheet: RPN Calculator

Off to work on the worksheet to implement the RPN calculator

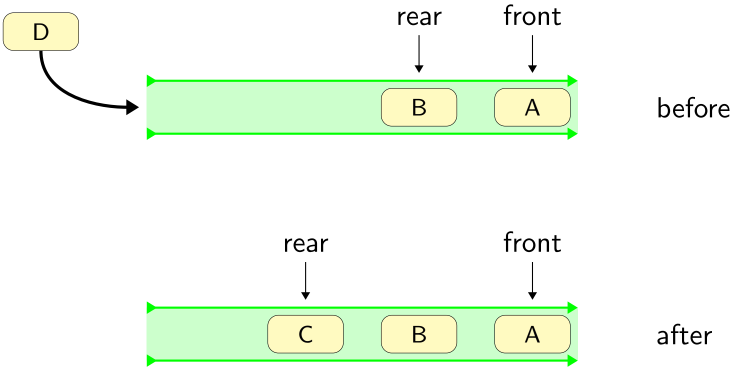
1. First in, First out (FIFO) data structure.
2. `Queue` is an interface in Java.
3. Uses the following operations:
    1. enqueue: to add to the end of a queue.
    2. dequeue: to remove the element at the front of the queue.



```
Queue<Integer> nums = new ArrayDequeue<>();
nums.offer("A")
```



```
Queue<Integer> nums = new ArrayDequeue<>();
nums.offer("A")
nums.offer("B")
```



D

rear

front

B

A

before

rear

front

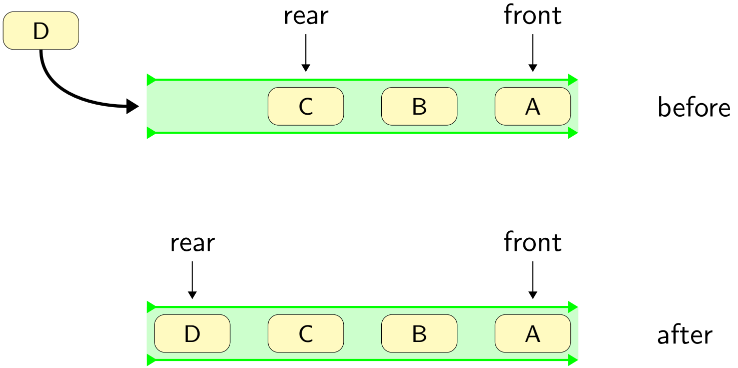
C

B

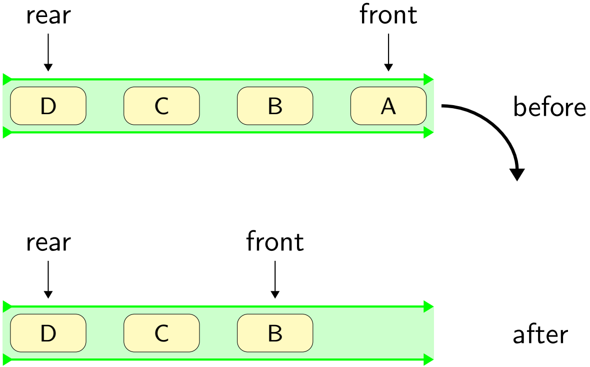
A

after

```
Queue<Integer> nums = new ArrayDequeue<>();
nums.offer("A")
nums.offer("B")
nums.offer("C")
```



```
Queue<Integer> nums = new ArrayDequeue<>();
nums.offer("A")
nums.offer("B")
nums.offer("C")
nums.offer("D")
```



```
Queue<Integer> nums = new ArrayDequeue<>();
nums.offer("A")
nums.offer("B")
nums.offer("C")
nums.offer("D")
nums.poll()
```

Worksheet: Queue Practice

Off to work on the worksheet to play with queues.

Process Scheduling

1. Dequeue a process
2. Check if the allowed processing time (quanta) is less than the time remaining to serve that proccess:
3. If it is:
    1. reduce the proc's remaining time by that quanta
    2. increment the total process time by the quanta
    3. increment that proc's contex switch count
    4. enqueue the process
    5. Print a message indicating the name of the process and it's quanta
4. Otherwise:
    1. increment the total processing time by the time remaining for that process
    2. print a message indicating the event name, the total time the process spent in the queue, and the number of time's it was switched out of context.

Worksheet: Round Robin Scheduler

Off to work on the worksheet to implement a round robin simulator.

Application: Card Game