Stack

Stack & Queue

- Two Different Data Structure
- Stack: LIFO (Last in first out)
- Queue: FIFO (First in first out)

When to use Queue?

- BFS (Breadth-first Search)
- Priority Queue (Heap)
- Multi Task Queue

When to use Stack?

- Invoke a function
- Recursion (This is a special case of the first)
- DFS (Depth-first Search) (This is a special case of the second)

Stack in Operating System

- Java
- C++

Java Stack & Heap

- Only Primitive type is stored in Stack
- Primitive type: boolean, char, short, int, long, byte, float, double
- Objects will be created in Heap and the reference will be stored in Stack
- For Boolean, Character, Short, Integer, Long, , ByteFloat and Double. They are not primitive types. They are Objects
- The data in Heap can be garbage collected
- Constant Pool and String Pool

Java Stack & Heap

- Stack
  - stack memory is used only by one thread of execution
  - stack memory cannot be accessed by other threads
  - stack memory is short-lived
  - When stack is full, it throws **StackOverFlowError**
- Heap
  - heap memory is used by all the parts of the application
  - objects stored in the heap are gloabally accessible
  - heap memory lives from the start till the end of application execution
  - When it is full, it throws **OutOfMemoryError**

C++ Stack & Heap

- The memory a program uses is typically divided into a few different areas, called segments:
  - code segment
  - bss segment
  - data segment
  - heap
  - call stack
- For heap, you can use malloc and calloc or new to dynamically get space to store data
- Remember to use free or delete the get the space back

C++ Stack & Heap

- Stack
  - very fast access
  - don't have to explicitly de-allocate variables
  - space is managed efficiently by CPU, memory will not become fragmented local variables only
  - limit on stack size (OS-dependent)
  - variables cannot be resized
- Heap
  - no limit on memory size but (relatively) slower access
  - no guaranteed efficient use of space, memory may become fragmented over time as blocks of memory are allocated, then freed
  - you must manage memory (you're in charge of allocating and freeing variables)
  - variables can be resized using realloc()

Stack

Java

```
Stack<String> stack = new Stack<String>();
```

C++

```
stack <string> cards;
```

Basic Opertaion

Java

- size
- pop
- push
- peek

C++

- size
- pop
- push
- top

Practice 1
Valid Parentheses

Valid Parentheses

We need to use the stack to store the information: Which left parentheses has not been matched yet.

Stack can help store every unused parentheses, no matter when the parentheses appears

```
public boolean isValid(String s) {
        Stack<Character> pair = new Stack<Character>();
        int n = s.length();
        if(n == 0) return true;
        for(int i = 0; i < n; i ++) {
                if(s.charAt(i) == '(' || s.charAt(i) == '[' || s.charAt(i) == '{')
                        pair.push(s.charAt(i));
                }
                else if(s.charAt(i) == ')') {
                        if(pair.isEmpty()) return false;
                        if(pair.pop() != '(') return false;
                }
                else if(s.charAt(i) == ']') {
                        if(pair.isEmpty()) return false;
                        if(pair.pop() != '[') return false;
                }
                else if(s.charAt(i) == '}') {
                        if(pair.isEmpty()) return false;
                        if(pair.pop() != '{') return false;
                }
        }
        if(pair.isEmpty()) return true;
        return false;
}
```

Practice 2

a stack with max()

Instead normal stack operation, this stack will have a max() function that could return the current max value of the stack

a stack with max()

One stack cannot tell the max in O(1)

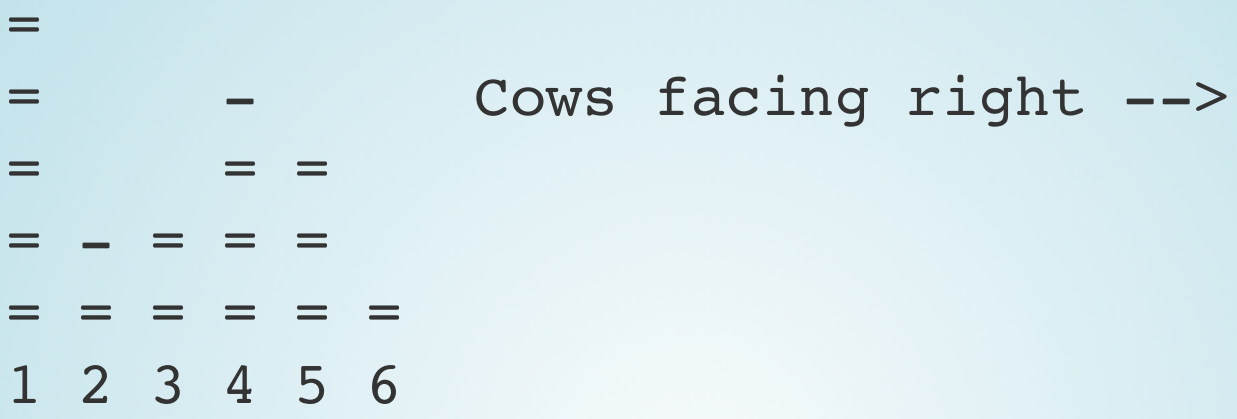
Need another stack to store the max

time <=> space

```
class MaxStack {
  Stack element;
  Stack maxValue;
  public MaxStack(int inCapacity) {
    element = new Stack(inCapacity);
    maxValue = new Stack(inCapacity);
  }
  public int size() {
    return element.size();
  }
  public void push(int value) {
    int currentMax;
    if(element.size() == 0) {
      currentMax = Integer.MIN_VALUE;
    } else {
      currentMax = maxValue.peek();
    }
    element.push(value);
    if(currentMax > value) {
      maxValue.push(currentMax);
    }
    else maxValue.push(value);
  }
  public int pop() {
    maxValue.pop();
    return element.pop();
  }
  public int max() {
    return maxValue.peek();
  }
}
```

Practice 3

Bad Hair Day



All the cows are facing right, and they can only see other cows in front of it and shorter than itself. It can be blocked by other taller cows.

Example: 6 2 3 1 7 4 Output: 5

SUM(each cow can see how many other cows?) =
SUM(each cow can be seen by how many other cows)

How to use Stack?

SUM(each cow can see how many other cows?) =
SUM(each cow can be seen by how many other cows)

How could Stack help record this?

Only allow cows whose height is lower
than you into the stack

How to use Stack?

Example: 6 2 3 1 7 4 Result: 5

Stack: 6	Add 6	
Stack: 6 2	Add 2	
Stack: 6 3	Pop 2 Add 3	+1
Stack: 6 3 1	Add 1	
Stack: 7	Pop 6 3 1 Add 7	+3
Stack: 7 4	Add 4	
Stack: 7	Pop 4	+1
Stack: empty	Pop 7	

```java
public static int seeCows(int[] cow) {
  int length = cow.length;
  int totalNum = 0;
  Stack<Integer> cowStack = new Stack<Integer>();
  for(int i = 0; i < length; i ++){
    if(cowStack.size() == 0 || cowStack.peek() > cow[i]) {
      cowStack.push(cow[i]);
    }
    else {
      while(cowStack.size() > 0 && cowStack.peek() <= cow[i]) {
        cowStack.pop();
        totalNum += cowStack.size();
      }
      cowStack.push(cow[i]);
    }
  }
  while(cowStack.size() > 0) {
    cowStack.pop();
    totalNum += cowStack.size();
  }
  return totalNum;
}
```

Practice 4

Remove Duplicate Letters

Remove Duplicate Letters

Given a string which contains only lowercase letters, remove duplicate letters so that every letter appear once and only once. You must make sure your result is the smallest in lexicographical order among all possible results.

Example:

Given "bcabc"

Return "abc"

Given "cbacdcbc"

Return "acdb"

Remove Duplicate Letters

recursion:

Every time choose the smallest one in the last position.

The letter is chosen either because that is the smallest letter or because there is no letter available after this one

Remove Duplicate Letters

```
public String removeDuplicateLetters(String s) {
    if (s == null ||s.length() == 0)
        return s;

    int[] cnt = new int[26];
    for (int i = 0; i < s.length(); i++) {
        cnt[s.charAt(i) - 'a']++;
    }

    int pos = 0;
    for (int i = 0; i < s.length(); i++) {
        if (s.charAt(i) < s.charAt(pos))
            pos = i;

        if (--cnt[s.charAt(i) - 'a'] == 0)
            break;
    }
    return s.charAt(pos) +
        removeDuplicateLetters(s.substring(pos + 1).replaceAll("" + s.charAt(pos), ""));
}
```

Remove Duplicate Letters

Consider a Stack:

What stack can help us?

Record the current shown letter

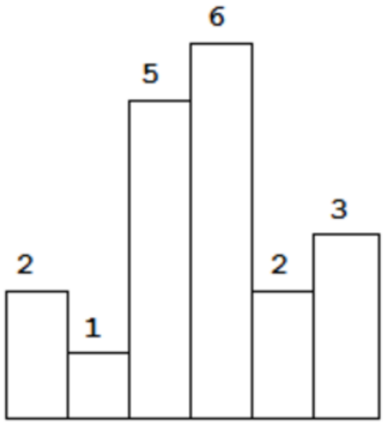
Pop element from the stack if:

1. this one is larger than the one just comes
2. this one still has some remaining can be used later

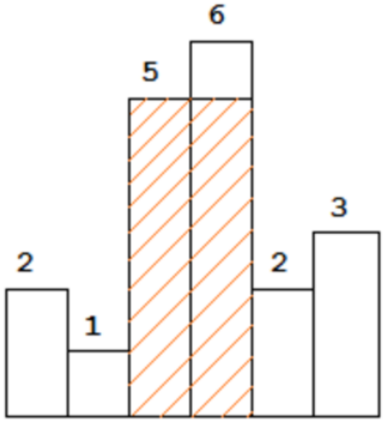
Remove Duplicate Letters

```
public String removeDuplicateLetters(String s) {
    int[] freqs = new int[256];
    for (int i = 0; i < s.length(); i++) {
        freqs[s.charAt(i)]++;
    }
    boolean[] visited = new boolean[256];
    Stack<Character> stack = new Stack<>();

    for (int i = 0; i < s.length(); i++) {
        char c = s.charAt(i);
        freqs[c]--;
        if (visited[c]) continue;

        while (!stack.isEmpty() && stack.peek() > c && freqs[stack.peek()] > 0) {
            visited[stack.pop()] = false;
        }
        stack.push(c);
        visited[c] = true;
    }

    StringBuilder sb = new StringBuilder();
    while(!stack.isEmpty()) {
        sb.append(stack.pop());
    }
    return sb.reverse().toString();
}
```

Practice 5
Largest Rectangle in Histogram



Above is a histogram where width of each bar is 1, given height = `[2,1,5,6,2,3]`.



The largest rectangle is shown in the shaded area, which has area = `10` unit.

Largest Rectangle in Histogram

Similar thought: How to use stack to find the right border?

Only put rectangle higher than peek into stack

Example: 2 1 5 6 2 3

The element before it is the left border

The element pushes it out is the right border

Largest Rectangle in Histogram

Example: 2 1 5 6 2 3

Stack: 2	Add 2	
Stack: 1	Pop 2 Add 1	2*1
Stack: 1 5	Add 5	
Stack: 1 5 6	Add 6	
Stack: 1 2	Pop 6 5 Add 2	6*1, 5*2
Stack: 1 2 3	Add 3	
Stack: 1 2	Pop 3	3*1
Stack: 1	Pop 2	2*4
Stack: Empty	Pop 1	1*6

Largest Rectangle in Histogram

Example: 2 1 5 6 2 3

How do we know the width?

Instead of directly putting height into stack, we put the index. Anyway height can be accessed by the index easily. And using index, we can know the width

Largest Rectangle in Histogram

Example: 2 1 5 6 2 3

Stack: 0	Add 0	
Stack: 1	Pop 0 Add 1	2*1
Stack: 1 2	Add 5	
Stack: 1 2 3	Add 6	
Stack: 1 4	Pop 6 5 Add 2	6*1, 5*2
Stack: 1 4 5	Add 3	
Stack: 1 4	Pop 3	3*1
Stack: 1	Pop 2	2*4
Stack: empty	Pop 1	1*6

```java
public int largestRectangleArea(int[] height) {
    int area = 0;
    java.util.Stack<Integer> stack = new java.util.Stack<Integer>();
    for (int i = 0; i < height.length; i++) {
        if (stack.empty() || height[stack.peek()] < height[i]) {
            stack.push(i);
        }
        else {
            int start = stack.pop();
            int width = stack.empty() ? i : i - stack.peek() - 1;
            area = Math.max(area, height[start] * width);
            i--;
        }
    }
    while (!stack.empty()) {
        int start = stack.pop();
        int width = stack.empty() ? height.length : height.length - stack.peek() - 1;
        area = Math.max(area, height[start] * width);
    }
    return area;
}
```

Homework

Simplify Path

Implementing Stack using Queues

Trapping Rain Water

Next Greater Element I