Assignment 3

1. Stack using an array:

Java

```
class Stack {
    int[] arr;
    int top;

    Stack(int size) {
        arr = new int[size];
        top = -1;
    }

    boolean isEmpty() {
        return top == -1;
    }

    boolean isFull() {
        return top == arr.length - 1;
    }

    void push(int x) {
        if (isFull()) {
            System.out.println("Stack Overflow");
            return;
        }
        arr[++top] = x;
    }

    int pop() {
        if (isEmpty()) {
            System.out.println("Stack Underflow");
            return -1;
        }
        return arr[top--];
    }
}
```

2. Balanced parentheses:

Java

```
boolean isBalanced(String str) {
    Stack<Character> stack = new Stack<>(str.length());
    for (int i = 0; i < str.length(); i++) {
        char c = str.charAt(i);
        if (c == '(' || c == '{' || c == '[') {
            stack.push(c);
        } else {
            if (stack.isEmpty() || !isMatchingPair(stack.pop(), c)) {
                return false;
            }
        }
    }
    return stack.isEmpty();
}

boolean isMatchingPair(char a, char b) {
    return (a == '(' && b == ')') || (a == '{' && b == '}') || (a == '[' && b == ']');
}
```

3. Reverse a string:

Java

```
String reverseString(String str) {
    Stack<Character> stack = new Stack<>(str.length());
    for (int i = 0; i < str.length(); i++) {
        stack.push(str.charAt(i));
    }
    StringBuilder reversed = new StringBuilder();
    while (!stack.isEmpty()) {
        reversed.append(stack.pop());
    }
    return reversed.toString();
}
```

4. Evaluate postfix expression:

Java

```
int evaluatePostfix(String expr) {
    Stack<Integer> stack = new Stack<>(expr.length());
    for (int i = 0; i < expr.length(); i++) {
```

```
        char c = expr.charAt(i);
        if (Character.isDigit(c)) {
            stack.push(Character.getNumericValue(c));
        } else {
            int op2 = stack.pop();
            int op1 = stack.pop();
            stack.push(performOperation(op1, op2, c));
        }
    }
    return stack.pop();
}

int performOperation(int op1, int op2, char operator) {
    switch (operator) {
        case '+':
            return op1 + op2;
        case '-':
            return op1 - op2;
        case '*':
            return op1 * op2;
        case '/':
            return op1 / op2;
        default:
            return    -1;
    }
}
```

5. Infix to postfix:

Java

```
String infixToPostfix(String expr) {
    Stack<Character> stack = new Stack<>(expr.length());
    StringBuilder postfix = new StringBuilder();
    for (int i = 0; i < expr.length(); i++) {
        char c = expr.charAt(i);
        if (Character.isDigit(c) || Character.isLetter(c)) {
            postfix.append(c);
        } else if (c == '(') {
            stack.push(c);
        } else if (c == ')') {
            while (!stack.isEmpty() && stack.peek() != '(') {
                postfix.append(stack.pop());
            }
            stack.pop();
        } else    {
            while (!stack.isEmpty() && precedence(c) <= precedence(stack.peek())) {
                postfix.append(stack.pop());
            }
            stack.push(c);
        }
    }
    while (!stack.isEmpty()) {
        postfix.append(stack.pop());
    }
    return postfix.toString();
}

int precedence(char c) {
    if (c == '+' || c == '-') {
        return 1;
    } else if (c == '*' || c == '/') {
        return 2;
    } else {
        return -1;
    }
}
```

6. Queue using an array:

Java

```
class Queue {
    int[] arr;
    int front, rear;

    Queue(int size) {
        arr = new int[size];
        front = -1;
        rear = -1;
    }

    boolean isEmpty() {
        return front == -1;
    }

    boolean isFull() {
        return (rear + 1) % arr.length == front;
```

```
    }

    void enqueue(int x) {
        if (isFull()) {
            System.out.println("Queue Overflow");
            return;
        }
        if (front == -1) {
            front = 0;
        }
        rear = (rear + 1) % arr.length;
        arr[rear] = x;
    }

    int dequeue() {
        if (isEmpty()) {
            System.out.println("Queue Underflow");
            return -1;
        }
        int x = arr[front];
        if (front == rear) {
            front = -1;
            rear = -1;
        } else {
            front = (front + 1) % arr.length;
        }
        return x;
    }
}
```

7. Circular Queue using an array:

Java

```
class CircularQueue {
    int[] arr;
    int front, rear;
    int size;

    CircularQueue(int size) {
        arr = new int[size];
        front = -1;
        rear = -1;
        this.size = size;
    }

    boolean isEmpty() {
        return front == -1;
    }

    boolean isFull() {
        return (rear + 1) % size == front;
    }

    void enqueue(int x) {
        if (isFull()) {
            System.out.println("Queue Overflow");
            return;
        }
        if (front == -1) {
            front = 0;
        }
        rear = (rear + 1) % size;
        arr[rear] = x;
    }

    int dequeue() {
        if (isEmpty()) {
            System.out.println("Queue Underflow");
            return -1;
        }
        int x = arr[front];
        if (front == rear) {
            front = -1;
            rear = -1;
        } else {
            front = (front + 1) % size;
        }
        return x;
    }
}
```

8. Queue using two Stacks:

Java

```
class QueueUsingStacks {
    Stack<Integer> stack1, stack2;
```

```
    QueueUsingStacks() {
        stack1 = new Stack<>();
        stack2 = new Stack<>();
    }

    void enqueue(int x) {
        stack1.push(x);
    }

    int dequeue() {
        if (stack2.isEmpty()) {
            if (stack1.isEmpty()) {
                return -1;
            }
            while (!stack1.isEmpty()) {
                stack2.push(stack1.pop());
            }
        }
        return stack2.pop();
    }
}
```

9. Min-Heap:

Java

```
class MinHeap {
    int[] arr;
    int size;

    MinHeap(int capacity) {
        arr = new int[capacity];
        size = 0;
    }

    int parent(int i) {
        return (i - 1) / 2;
    }

    int left(int i) {
        return 2 * i + 1;
    }

    int right(int i) {
        return 2 * i + 2;
    }

    void insertKey(int    k) {
        if (size == arr.length) {
            return;
        }
        size++;
        arr[size - 1] = k;
        int i = size - 1;
        while (i != 0 && arr[parent(i)] > arr[i]) {
            int temp = arr[parent(i)];
            arr[parent(i)] = arr[i];
            arr[i] = temp;
            i = parent(i);
        }
    }

    int extractMin() {
        if (size == 0) {
            return Integer.MAX_VALUE;
        }
        int root = arr[0];
        arr[0] = arr[size - 1];
        size--;
        heapify(0);
        return root;
    }

    void deleteKey(int i) {
        if (i < 0 || i >= size) {
            return;
        }
        arr[i] = Integer.MIN_VALUE;
        heapify(i);
        extractMin();
    }

    void heapify(int i) {
        int smallest = i;
```

```
        int l = left(i);
        int r = right(i);
        if (l < size && arr[l] < arr[smallest]) {
            smallest = l;
        }
        if (r < size && arr[r] < arr[smallest]) {
            smallest = r;
        }
        if (smallest != i)    {
            int temp = arr[smallest];
            arr[smallest] = arr[i];
            arr[i]    = temp;
            heapify(smallest);
        }
    }
}
```

10. Max-Heap:

Java

```
class MaxHeap {
    int[] arr;
    int size;

    MaxHeap(int capacity) {
        arr = new int[capacity];
        size = 0;
    }

    int parent(int i) {
        return (i - 1) / 2;
    }

    int left(int i) {
        return 2 * i + 1;
    }

    int right(int i) {
        return 2 * i + 2;
    }

    void insertKey(int    k) {
        if (size == arr.length) {
            return;
        }
        size++;
        arr[size - 1] = k;
        int i = size - 1;
        while (i != 0 && arr[parent(i)] < arr[i]) {
            int temp = arr[parent(i)];
            arr[parent(i)] = arr[i];
            arr[i] = temp;
            i = parent(i);
        }
    }

    int extractMax() {
        if (size == 0) {
            return Integer.MIN_VALUE;
        }
        int root = arr[0];
        arr[0] = arr[size - 1];
        size--;
        heapify(0);
        return root;
    }

    void deleteKey(int i) {
        if (i < 0 || i >= size) {
            return;
        }
        arr[i] = Integer.MIN_VALUE;
        heapify(i);
        extractMax();
    }

    void heapify(int i) {
        int largest = i;
        int l = left(i);
        int r = right(i);
        if (l < size && arr[l] > arr[largest]) {
            largest = l;
        }
        if (r < size && arr[r] > arr[largest]) {
            largest = r;
        }
```

```
        if (largest    != i) {
            int temp = arr[largest];
            arr[largest] = arr[i];
            arr[i] = temp;
            heapify(largest);
        }
    }
}
```

11. Heap Sort:

Java

```
void heapSort(int[] arr) {
    int n = arr.length;
    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(arr, n, i);
    }
    for (int i = n - 1; i >= 0; i--) {
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;
        heapify(arr, i, 0);
    }
}

void heapify(int[] arr, int n, int    i) {
    int largest = i;
    int l = 2 * i + 1;
    int r = 2 * i + 2;
    if (l < n && arr[l] > arr[largest]) {
        largest = l;
    }
    if (r < n && arr[r] > arr[largest]) {
        largest = r;
    }
    if (largest != i) {
        int temp = arr[largest];
        arr[largest] = arr[i];
        arr[i] = temp;
        heapify(arr, n, largest);
    }
}
```

12. Kth largest element in a stream of numbers:

Java

```
class KthLargest {
    PriorityQueue<Integer> pq;
    int k;

    KthLargest(int k, int[] nums) {
        this.k = k;
        pq = new PriorityQueue<>(k);
        for (int num : nums) {
            add(num);
        }
    }

    public int add(int val) {
        pq.offer(val);
        if (pq.size() > k) {
            pq.poll();
        }
        return pq.peek();
    }
}
```

13. Priority Queue using a heap:

Java

```
class PriorityQueue {
    PriorityQueue<Integer> pq;

    PriorityQueue() {
        pq = new PriorityQueue<>();
    }

    void enqueue(int x, int priority) {
        pq.offer(new Pair(x, priority));
    }

    int dequeue() {
        if (pq.isEmpty()) {
            return -1;
        }
```

```
        return pq.poll().getKey();
    }

    class Pair implements Comparable<Pair> {
        int key, priority;

        Pair(int key, int priority) {
            this.key = key;
            this.priority = priority;
        }

        public int getKey() {
            return key;
        }

        @Override
        public int compareTo(Pair other) {
            return other.priority - this.priority;
        }
    }
}
```

14. Stack with getMin() in constant time:

Java

```
class MinStack {
    Stack<Integer> stack;
    Stack<Integer> minStack;

    MinStack() {
        stack = new Stack<>();
        minStack = new Stack<>();
    }

    void push(int val) {
        stack.push(val);
        if (minStack.isEmpty() || val <= minStack.peek()) {
            minStack.push(val);
        }
    }

    void pop() {
        if (!stack.isEmpty()) {
            if (stack.peek().equals(minStack.peek())) {
                minStack.pop();
            }
            stack.pop();
        }
    }

    int top() {
        return stack.peek();
    }

    int getMin() {
        return minStack.peek();
    }
}
```

15. Circular Queue with fixed size:

Java

```
class FixedSizeCircularQueue {
    int[] arr;
    int front, rear, size;

    FixedSizeCircularQueue(int size) {
        arr = new int[size];
        front = -1;
        rear = -1;
        this.size = size;
    }

    boolean isEmpty() {
        return front == -1;
    }

    boolean isFull() {
        return (rear + 1) % size == front;
    }

    void enqueue(int x) {
        if (isFull()) {
            System.out.println("Queue Overflow");
```

```
            return;
        }
        if (front == -1) {
            front = 0;
        }
        rear = (rear + 1) % size;
        arr[rear] = x;
    }

    int dequeue() {
        if (isEmpty()) {
            System.out.println("Queue Underflow");
            return -1;
        }
        int x = arr[front];
        if (front == rear) {
            front = -1;
            rear = -1;
        } else {
            front = (front + 1) % size;
        }
        return x;
    }
}
```