Stacks

- A Stack is a linear data structure that follows the **LIFO (Last-In-First-Out)** principle.
- Stack has one end
    - Whenever an element is added in the stack, it is added on the top of the stack.
    - Whenever an element is removed from the stack, it is removed from the top of the stack.
- A stack can be implemented by means of Array and Linked List.
    - You can implement stack can either be a fixed size (use Array) or Stack can either be a dynamic size (Linked List).
- Some Basic Operations of Stacks:
    - push():
        - When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.
        - Some implementing classes may throw an IllegalStateException if the capacity limit is exceeded or a NullPointerException if the element being inserted is null.
        - To avoid the IllegalStateException, first test with isfull().
            - isFull(): It determines whether the stack is full or not.
        - LinkedList has no capacity limit and allows for null entries, though as we will see you should be very careful when adding null.
    - pop():
        - When we delete an element from the stack, the operation is known as a pop.
        - It will remove and return the element at the top of the stack.

        - If the stack is empty means that no element exists in the stack, this state is known as an underflow state.

        - Some implementing classes may throw a NoSuchElementException if the deque is empty. To avoid the exception, first test with isEmpty().

            - isEmpty(): It determines whether the stack is empty or not.

    - peek():

        - It will retrieve the element at the top of the stack, but does not remove.

        - If the stack is empty, it returns null. Since null may be a valid entry, this leads to ambiguity. Therefore a better test for an empty stack is isEmpty().

    - When the stack is implemented using a linked list, both push and pop operations have 0(1) time complexity. If it is implemented using an array, both push and pop operations have still 0(1) time complexity.

IMPLEMENT A STACK WITH MAX API

Design a stack that includes a max operation, in addition to push and pop. The max method should return the maximum value stored in the stack.

Approach:

- Use additional storage to track the maximum value.

- Suppose we use a single auxiliary variable, M, to record the element that is maximum in the stack. Updating M on pushes is easy: M = max(M,e), where e is the element being pushed.

- However, updating M on pop is very time consuming. If M is the element being popped, we have no way of knowing what the maximum remaining element is, and are forced to consider all the remaining elements.

- To overcome this problem, for each entry in the stack, we cache the maximum stored at or below that entry. Now when we pop, we evict the corresponding cached value.

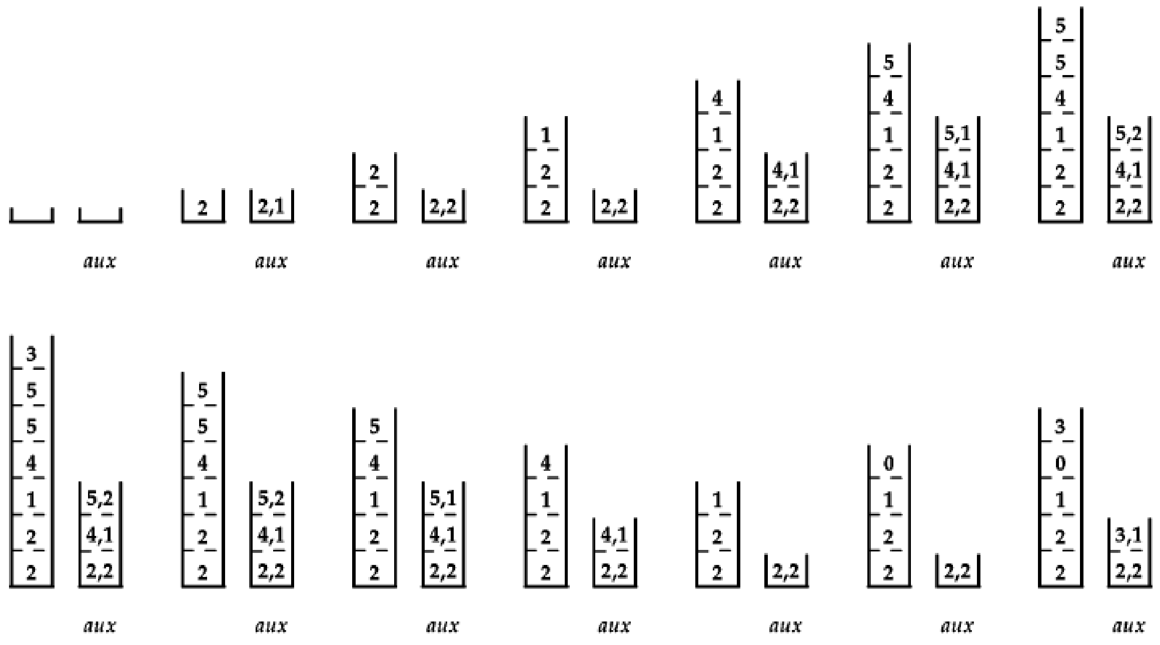


Figure 9.2: The primary and auxiliary stacks for the following operations: push 2, push 2, push 1, push 4, push 5, push 5, push 3, pop, pop, pop, pop, push 0, push 3. Both stacks are initially empty, and their progression is shown from left-to-right, then top-to-bottom. The top of the auxiliary stack holds the maximum element in the stack, and the number of times that element occurs in the stack. The auxiliary stack is denoted by *aux*.

Program:

```
import java.util.Arrays;
import java.util.Deque;
import java.util.LinkedList;

public class StackWithMax {

	// @include
	private static class ElementWithCachedMax {

		public Integer element;
		public Integer max;

		public ElementWithCachedMax(Integer element, Integer max) {

			this.element = element;
			this.max = max;
		}
	}
```

```
public static class Stack {

        // Stores (element, cached maximum) pair.
        private Deque<ElementWithCachedMax> elementWithCachedMax
                                                        = new LinkedList<>();

        public boolean empty() { return elementWithCachedMax.isEmpty(); }

        public Integer max() {

                if (empty()) {

                        throw new IllegalStateException("max(): empty stack");
                }
                return elementWithCachedMax.peek().max;
        }

        public Integer pop() {

                if (empty()) {

                        throw new IllegalStateException("pop(): empty stack");
                }

                return elementWithCachedMax.removeFirst().element;
        }

        public void push(Integer x) {

                elementWithCachedMax.addFirst (new
                        ElementWithCachedMax(x, Math.max(x, empty() ? x : max())));
        }
}
// @exclude

public static void check(boolean condition, String msg) {

        if (!condition) {

                System.err.println(msg);
                System.exit(-1);
        }
}

public static void missedMaxException() {
        System.err.println("Should have seen an exception, max() on empty stack!");
        System.exit(-1);
}

public static void missedPopException() {

        System.err.println("Should have seen an exception, pop() on empty stack!");
        System.exit(-1);
}

public static void main(String[] args) {

        Stack s = new Stack();
        s.push(1);
        s.push(2);
        check(s.max() == 2, "failed max() call with stack created by push 1, push 2");
        System.out.println(s.max()); // 2
        System.out.println(s.pop()); // 2
        check(s.max() == 1, "failed max() call with stack created by push 1, push 2,
                                                                            pop");
        System.out.println(s.max()); // 1
        s.push(3);
        s.push(2);
        check(s.max() == 3, "failed max() call with stack created by push 1, push 2, pop,
                                                                    push 3, push 2");
        System.out.println(s.max()); // 3
        s.pop();
        check(s.max() == 3, "failed max() call with stack created by push 1, push 2, pop,
                                                               push 3, push 2, pop");
        System.out.println(s.max()); // 3


        s.pop();
        check(s.max() == 1,  "failed max() call with stack created by push 1, push 2, pop,
                                                          push 3, push 2, pop, pop");
        System.out.println(s.max()); // 1
        s.pop();

        try {

                s.max();
                missedMaxException();

        } catch (RuntimeException e) {
                System.out.println("Got expected exception calling max() on an empty
                                                        stack:" + e.getMessage());
            }

            try {

                s.pop();
                missedPopException();

            } catch (RuntimeException e) {

                System.out.println("Got expected exception calling pop() on an empty
                                                        stack:" + e.getMessage());
            }
        }
}
```

Output:

```
2
2
1
3
3
1
Got expected exception calling max() on an empty stack:max(): empty stack
Got expected exception calling pop() on an empty stack:pop(): empty stack
```

Time and Space Complexity:

- The time complexity for each specified method is still *0(1).*
- In the best-case, the additional space complexity is less, *0(1)*, which occurs when the number of distinct keys is small, or the maximum changes infrequently.
- The worst-case additional space complexity is *0(n),* which occurs when each key pushed is greater than all keys in the primary stack.

EVALUATE Reverse Polish Notation (RPN) EXPRESSIONS

- Reverse Polish Notation is postfix notation which in terms of mathematical notion signify the operators following operands.

Write a program that takes an arithmetical expression in RPN and returns the number that the expression evaluates to.

Example:

- Let an arithmetical expression in RPN is inputted as string, i.e., “3,4,+,2,*,1,+”
- After evaluation of RPN arithmetical expression, the output is 15

Approach:

The basic approach for the problem is using the **stack.**

- Accessing all elements in the array, if the element is not matching with the special character (‘+’, ‘-‘,’*’, ‘/’) then push the element to the stack.
- Then whenever the special character is found then pop the first two-element from the stack and perform the action and then push the element to stack again.
- Repeat the above two process to all elements in the array.
- At last pop the element from the stack and print the Result.

Program:

```
import java.util.Deque;
import java.util.LinkedList;
import java.util.Scanner;

public class EvaluateRPNExpressionsProg1 {

        public static void main(String[] args) {

                Scanner sc = new Scanner(System.in);
                System.out.print("Enter an arithmetical expression in RPN, each character
                                                        separated by comma (,) : ");
                String str = sc.nextLine();

                int result = eval(str);

                System.out.print("The evaluation result of RPN arithmetical expression : " +
                                                                                result);
        }

        public static int eval(String RPNExpression) {

                Deque<Integer> intermediateResults = new LinkedList<>();

                String delimiter = ",";
                String[] symbols = RPNExpression.split(delimiter);
        for (String token : symbols) {

            if (token.length() == 1 && "/*+-".contains(token)){

                final int y = intermediateResults.removeFirst();
                final int x = intermediateResults.removeFirst();

                switch (token.charAt(0)){

                    case '+':

                        intermediateResults.addFirst(x + y);
                        break ;

                    case '-':

                        intermediateResults.addFirst(x - y);
                        break ;

                    case '*':

                        intermediateResults.addFirst(x * y);
                        break ;

                    case '/':

                        intermediateResults.addFirst(x / y);
                        break ;

                    default:

                        throw new IllegalArgumentException("Malformed
                                                 RPN at :" + token);
                }

            } else { // token is a number.

                intermediateResults.addFirst(Integer.parseInt(token));

            }
        }

        return intermediateResults.removeFirst();
    }
}
```

Output:

```
Enter an arithmetical expression in RPN, each character separated by comma (,) : 3,4,+,2,*,1,+
The evaluation result of RPN arithmetical expression : 15
```

Time and Space Complexity:

- Since we perform *0(1)* computation per character of the string, the time complexity is *0(n),* where *n* is the length of the string. The space complexity is O(1).

Queues

- A Queue is a linear structure that follows the First In First Out (FIFO) principle, , i.e., the data item stored first will be accessed first.

- A queue is open at both its ends :

    - One end is always used to insert data (enqueue) and the other is used to remove data (dequeue).

- A queue supports two basic operations:

    - **Enqueue:** Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition.

    - **Dequeue:** Removes an item from the queue. If the queue is empty, then it is said to be an Underflow condition. If the queue is empty, dequeue typically returns null or throws an exception.

    - Elements are added (enqueued) and removed (dequeued) in first-in, first-out order.

- The most recently inserted element is referred to as the tail or back element, and the item that was inserted least recently is referred to as the head or front element.

- A Queue can be implemented by means of Array and Linked List.

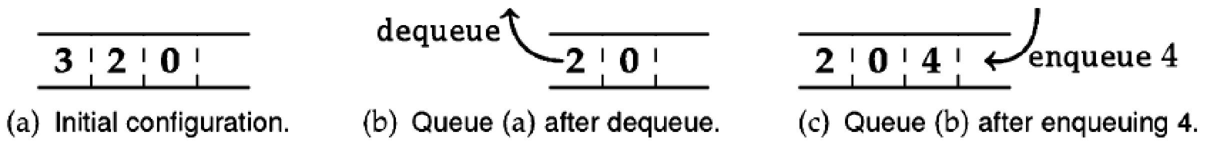

3 2 0
dequeue
2 0
2 0 4
enqueue 4
(a) Initial configuration.
(b) Queue (a) after dequeue.
(c) Queue (b) after enqueuing 4.

Figure 9.4: Examples of enqueuing and dequeuing.

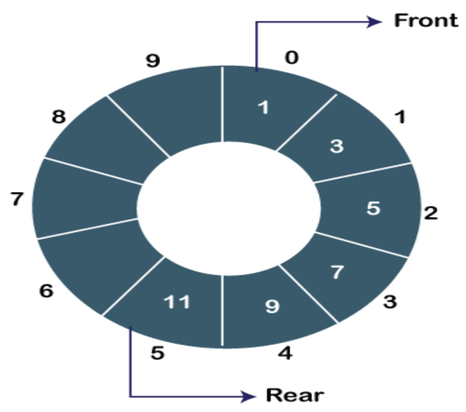
- The preferred way to manipulate queues is via the Deque interface.

- The LinkedList class is a doubly linked list that implements this interface, and provides efficient (0(1) time) queue (and stack) functionality.

- The key queue-related methods in the Deque interface are addLast(3.14), removeFirst(), getFirst().
    - addLast (e):
        - addLast (e) enqueues an element. Some classes implementing Deque have capacity limits and/or preclude null from being enqueued, but this is not the case for LinkedList.
    - removeFirst ()
        - removeFirst() retrieves and removes the first element of this deque, throwing NoSuchElementException if the deque is empty.
    - getFirst()
        - getFirst() retrieves, but does not remove, the first element of this deque, throwing NoSuchElementException if the deque is empty.
- **The time complexity**
    - The time complexity of enqueue and dequeue are O(1).
    - The time complexity of finding the maximum is 0(n), where n is the number of entries.

IMPLEMENT A CIRCULAR QUEUE

- A queue can be implemented using an array and two additional fields, the beginning/front and the end/rear indices.
- In Circular Queue, all the nodes are represented as circular. It is similar to the linear Queue except that the last element of the queue is connected to the first element.
- The circular queue can be represented as:



Implement a queue API using an array for storing elements. Your API should include a constructor function, which takes as argument the initial capacity of the queue, enqueue and dequeue functions, and a function which returns the number of elements stored. Implement dynamic resizing to support storing an arbitrarily large number of elements.

Program:

```
import java.util.Arrays;
import java.util.Collections;
import java.util.NoSuchElementException;

public class CircularQueue {

        // @include
        public static class Queue {

                private int head = 0, tail = 0, numQueueElements = 0;
                private static final int SCALE_FACTOR = 2;
                private Integer[] entries;

                public Queue(int capacity) { entries = new Integer[capacity]; }

                public void enqueue(Integer x) {

                        if (numQueueElements == entries.length) { // Need to resize.

                                // Makes the queue elements appear consecutively.
                                Collections.rotate(Arrays.asList(entries), -head);
                                // Resets head and tail.
                                head = 0;
                                tail = numQueueElements;
                                entries = Arrays.copyOf(entries, numQueueElements *
                                                                                SCALE_FACTOR);
                        }
```

```
            entries[tail] = x;
            tail = (tail + 1) % entries.length;
            ++numQueueElements;
        }

        public Integer dequeue() {

            if (numQueueElements != 0) {
            --numQueueElements;
            Integer ret = entries[head];
            head = (head + 1) % entries.length;
            return ret;
            }
            throw new NoSuchElementException("Dequeue called on an empty
                                                       queue.");
        }

        public int size() { return numQueueElements; }

    }  // @exclude


    private static void assertDequeue(Queue q, Integer t) {

        Integer dequeue = q.dequeue();
        assert(t.equals(dequeue));
    }

    public static void main(String[] args) {

        Queue q = new Queue(5);
        q.enqueue(1);
        q.enqueue(2);
        q.enqueue(3);

        int totalele = q.size();
        System.out.println("The total elements in Circular queue: " + totalele);

        q.enqueue(4);
        q.enqueue(5);
        q.enqueue(6);
        int totalele1= q.size();
        System.out.println("The total elements in Circular queue: " + totalele1);
    }
}
```

Output:

```
The total elements in Circular queue: 3
The total elements in Circular queue: 6
```

Time and Space complexity:

- The time complexity of dequeue is *0(1)*, and the amortized time complexity of enqueue is *0(1).*

IMPLEMENT A QUEUE USING STACKS

- Queue is a linear data structure that follows FIFO (First In First Out) principle in which insertion is performed from the rear end and the deletion is done from the front end.

- Stack is a linear data structure that follows LIFO (Last In First Out) principle in which both insertion and deletion are performed from the top of the stack.

Write a program to implement a Queue using Stacks.

Approach:

- A queue can be implemented using two stacks.

- Use the first stack for enqueue and the second for dequeue.

Program:

```
import java.util.Deque;
import java.util.LinkedList;
import java.util.NoSuchElementException;

public class ImplementQueueUsingStacks {

        public static class Queue {

                private Deque <Integer> enq = new LinkedList<>();
                private Deque <Integer> deq = new LinkedList<>();

                public void enqueue(Integer x) { enq.addFirst(x); }

                public Integer dequeue() {

                        if (deq.isEmpty()){

                                // Transfers the elements from enq to deq.
                while (!enq.isEmpty()){

                    deq.addFirst(enq.removeFirst());
                }
            }

            if (!deq.isEmpty()){

                return deq.removeFirst();
            }

            throw new NoSuchElementException("Cannot pop empty queue");
        }
    }

    // Driver code
    public static void main(String[] args) {

        Queue q = new Queue();

        q.enqueue(1);
        q.enqueue(2);
        q.enqueue(3);

        System.out.println(q.dequeue());
        System.out.println(q.dequeue());
        System.out.println(q.dequeue());
        //System.out.println(q.dequeue());
    }
}
```

Output:

```
1
2
3
```

Time and Space complexity:

- This approach takes *0(m)* time for *m* operations, which can be seen from the fact that each element is pushed no more than twice and popped no more than twice.

IMPLEMENT A QUEUE WITH MAX API

Implement a queue with enqueue, dequeue, and max operations. The max operation returns the maximum element currently stored in the queue.

Approach:

- Maintain two queues one for entries, i.e., originalQueue and another for max candidate, i.e., max_queue
- In the max_queue, head will be pointing to the current max element.
- When an element is enqueued to the originalQueue
    - Check the tail of max_queue, if tail is greater than element, then add the element at tail.
    - If tail is less than element, then remove from tail iteratively until max_queue is empty or tail element is greater than or equal to the element, then add the element at tail.
- When an element is dequeued from originalQueue
    - Check the element is same as max_queue's head or not, if it is same then remove the max_queue's head, otherwise the max_queue remains unchanged
- Note: max_queue must contain elements in decreasing order.

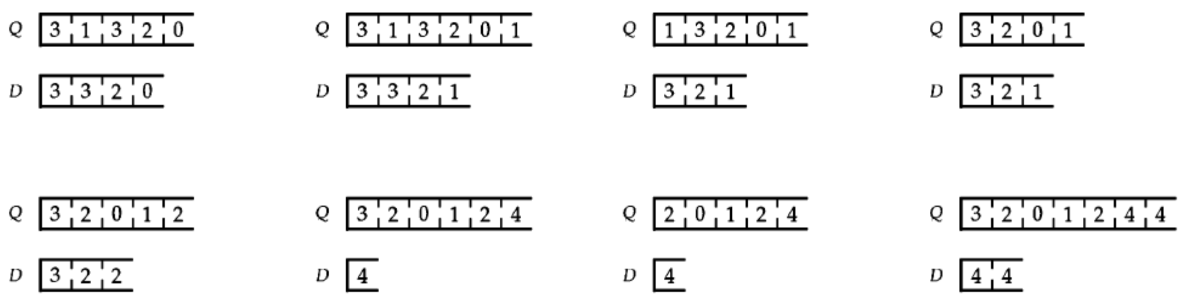


Figure 9.5: The queue with max for the following operations: enqueue 1, dequeue, dequeue, enqueue 2, enqueue 4, dequeue, enqueue 4. The queue initially contains $3, 1, 3, 2$, and 0 in that order. The deque D corresponding to queue Q is immediately below Q. The progression is shown from left-to-right, then top-to-bottom. The head of each queue and deque is on the left. Observe how the head of the deque holds the maximum element in the queue.

Program:

```
import java.util.Deque;
import java.util.LinkedList;
import java.util.NoSuchElementException;
import java.util.Queue;

public class QueueWithMax <T extends Comparable<T>> {

	//Maintain two queues one for entries and another for max candidate
	private Queue<T> entries = new LinkedList<>();
	private Deque<T> candidateForMax = new LinkedList<>();

	/* While inserting new element in the queue, check if the newly added
	* element is bigger than the last element on the candidateForMax if
	* yes remove it all small elements are removed then add the new
	* element to candidateForMax queue
	*/

	public void enqueue(T x){

		entries.add(x);

		while (!candidateForMax.isEmpty()){

			if(candidateForMax.getLast().compareTo(x) >= 0) {

				break;
			}
			candidateForMax.removeLast();
		}
		candidateForMax.addLast(x);
	}


	// Remove a entry from entries queue and check if its same as that of
	// the max candidate if yes remove that entry from max candidate as well
	public T dequeue(){

		if(!entries.isEmpty()){

			T result  = entries.remove();

			if(result.equals(candidateForMax.getFirst())){

					candidateForMax.removeFirst();
```

```
            }
            return result;
        }
        throw new NoSuchElementException("Called dequeue on empty row");
    }

    public T max(){

        if(!candidateForMax.isEmpty()) {

            return candidateForMax.getFirst();
        }
        throw new NoSuchElementException("Called dequeue on empty row");
    }

    //Display the original Queue elements
    public void displayEntries() {

        System.out.print("The Queue elements are: ");

        for(T Qelement : entries) {

            System.out.print(Qelement + " ");
        }
        System.out.println(" ");
    }


    //Display the MaxQueue candidate for Max elements
    public void displaycandidateForMax() {

        System.out.print("The candidate for Max elements are: ");

        for(T maxelement : candidateForMax) {

            System.out.print(maxelement + " ");
        }
        System.out.println(" ");
    }


    // Driver code
    public static void main(String[] args) {

        QueueWithMax q = new QueueWithMax();
```

```
                //The queue initially contains 3,1,3,2, and 0 in that order.
                q.enqueue(3);
                q.enqueue(1);
                q.enqueue(3);
                q.enqueue(2);
                q.enqueue(0);

                //Display the original Queue elements
                q.displayEntries();
                //Display the MaxQueue candidate for Max elements
                q.displaycandidateForMax();
                System.out.println("The max element in queue is " + q.max());

                q.enqueue(1);
                q.dequeue();
                q.dequeue();
                q.enqueue(2);
                q.enqueue(4);
                q.dequeue();
                q.enqueue(4);
                //Display the original Queue elements
                q.displayEntries();
                //Display the MaxQueue candidate for Max elements
                q.displaycandidateForMax();
                System.out.println("The max element in queue is " + q.max());
        }
}
```

Output:

```
The Queue elements are: 3 1 3 2 0
The candidate for Max elements are: 3 3 2 0
The max element in queue is 3
The Queue elements are: 2 0 1 2 4 4
The candidate for Max elements are: 4 4
The max element in queue is 4
```

Time and Space Complexity:

- Each dequeue operation has time *0(1)* complexity. A single enqueue operation may entail many ejections from the max_queue. However, the amortized time complexity of *n* enqueues and dequeues is *0(n),* since an element can be added and removed from the max_queue no more than once. The max operation is *0(1)* since it consists of returning the element at the head of the max_queue.