Contents

Introduction

Stack

One of the most used data structure, which is often used with various different algorithms, such as graph or tree traversals, can be used to implement recursive algorithms in an iterative manner. The common interface that a stack uses is as follows

- pop() - remove the element at the top of the stack
- push() - push a new element at the top of the stack
- peek() - peek the element at the top of the stack
- empty() - checks if the stack is empty, true / false

Note, that in the default implementation in java, when using most of those methods on an empty stack, they would throw StackEmtpyException, or something of that nature - pop, peek for example.

```
public static final class Stack<T> {

    // simple node to describe an entry in the stack, due to the nature of
       the
    // stack, it does not need to be doubly linked, the implementation is
       very
    // similar to a singly linked list
    private static final StackNode<T> {
        private T value;
        private StackNode<T> next;
    }

    // simply hold a reference to the head element, the head will move
       forward
    // when pushing, and move backward when popping or removing
    private StackNode<T> head;

    public void push(T value) {
        if (this.head == null) {
            // when the head of the stack if not initialized make sure to
               set it
            // and leave, that for the first time adding to the stack
            this.head = new StackNode<T>();
            this.head.value = value;
            this.head.next = null;
        } else {
            // on each new element, simply make the new element the head,
               and
            // correctly link the next element of the new element to the
               current
            // head.
            StackNode<T> newHead = new StackNode<>();
            newHead.value = value;
            newHead.next = this.head;
            this.head = newHead;
        }
    }

    public T pop() {
        if (this.head == null) {
            // in case the stack is empty, simply throw, which is what
               actually
            // happens in the default java stack implementation from the
               standard
            // library
            throw new StackEmtpyException();
        }

        // otherwise when we have at least one head element, extract it,
           the new
        // head becomes the immediate next node after the current head,
           and simply
        // return the value of the current head
        StackNode<T> current = this.head;
        this.head = current.next;
        return current.value;
    }

    public T peek() {
        if (this.head == null) {
            // in case the stack is empty, simply throw, which is what
               actually
            // happens in the default java stack implementation from the
               standard
            // library
            throw new StackEmtpyException();
        }

        // peeking, does not mutate the state of the stack, it simply
           returns
        // whatever is at the top, where the current head points at
        return this.head.value;
    }
    public boolean empty() {
        // simple check to verify the stack has any elements at all, when
           all
        // elements are removed, the head will eventually point at null,
           or when
        // the stack is not initialized in the first place
        return this.head != null;
    }
}
```

Min stack

One common problem that might come up is to keep track of the min element inserted in a stack, such that peeking at the min element in the stack is always constant

The solution here is to have two stacks one, which holds the min elements, in the same order in which they were inserted, the other stack is the actual stack holding the elements

```
// the min stack holds only the min elements, those elements are ordered
   in the
// same way they are inserted in the main stack, but only `min` elements
   are added
Stack<Integer> stackMin = new Stack<>();

// the main stack which holds all elements, including the min elements also
// contained in the min stack
Stack<Integer> stackBase = new Stack<>();

void push(Integer value) {
    // if the min stack is empty, default to the smallest valid value,
       otherwise get the head from the min stack
    Integer min = !this.stackMin.isEmpty() ? this.stackMin.peek() :
       Integer.MAX_VALUE;

    if (value < min) {
        // when the value is less than head of the min stack, then add
           that value to the min stack
        this.stackMin.push(value);
    }
    // normal pushing to the main stack
    this.stackBase.push(value);
}

Integer pop() {
    if (!this.stackMin.isEmpty()) {
        // get the top element from the min stack
        Integer min = this.stackMin.peek();

        if (min.equals(this.stackBase.peek())) {
            // the top element are removing is the same as the one from
               the min stack, therefore we have to remove it from the min
            // stack too, to keep them sync. After removing from the main
               stack and the min stack, the min stack would now contain
            // the next most minimal element
            this.stackMin.pop();
        }
    }
    // normal removal from the main stack
    return this.stackBase.pop();
}

Integer min() {
    if (!this.stackMin.isEmpty()) {
        // whatever is at the head here, since pushing and popping were
           kept in sync between the two stacks, we must find the min
        // element in the main stack be present at the head of the min
           stack
        return this.stackMin.peek();
    }
    return null;
}
```

The keynote here is to take a look at the inserting and removing, when a new element is added to the stack, it is checked against the head of the min stack, if it is smaller, it is added to the head of the min stack, when removing an element from the main stack we do the same, check the head of the min stack, if we are removing the most recent `min` element from the main stack, we do the same, and remove from the min stack too

Stack of stacks

Another problem is often revolving around storing a stack inside a stack, where each sub-stack has a max limit of elements it can hold, and when exceeded a new sub-stack is created and pushed on top of the main stack (which holds the sub-stacks)

```

// limit is a integral value which limits the number of items that can be
// inserted in each sub-stack, when exceeded a new stack is created on top
int limit;

// store a stack of stacks, each of the sub-stacks has at most `limit`
   number
// of elements inside of it
Stack<Stack<Integer>> stack = new Stack<>();

void push(Integer value) {
    if (this.stack.isEmpty() || this.stack.peek().size() >= this.limit) {
        // in case the main stack is empty or the current stack at the
           head has reached it's limits, add new stack to the total
        // stack of stacks
        this.stack.push(new Stack<>());
    }
    // peek the head stack and add new element
    stack.peek().push(value);
}
```

```java
Integer pop() {
    if (this.stack.isEmpty() || this.stack.peek().isEmpty()) {
        return null;
    }
    // get the stack at the top, and pop the value from it, which we will
       return
    Stack<Integer> top = stack.peek();
    Integer value = top.pop();

    if (top.isEmpty()) {
        // in case that top stack now became empty, we can remove it from
           the stack of stacks,
        this.stack.pop();
    }
    // return the top value from the last stack that was at the top of
       stack of stacks
    return value;
}

Integer peek() {
    if (this.stack.isEmpty() || this.stack.peek().isEmpty()) {
        return null;
    }
    // the top of the stack of stacks, would contain at this point a stack
       with at least one element, peek it
    return this.stack.peek().peek();
}
```

Sorting stack

Yet another very nice problem is how to keep a stack of elements always sorted, such that the elements are ordered in ascending or descending order. To achieve this when inserting a new element into the stack we first pop all elements which are smaller (or bigger, depending on the order we desire) we then insert the new element, and pop back all elements we removed.

```java
// the main stack which holds the sorted elements, such that at the head
   we have
// the smallest element, and at the bottom of the stack is the biggest
   element
Stack<Integer> sorted = new Stack<>();

// a temporary stack which we will fill up with smaller / bigger elements
   while
// inserting the new element
Stack<Integer> temp = new Stack<>();

void push(Integer value) {
    if (!sorted.isEmpty()) {
        // pull all elements from the actual stack, that are greater than
           the new element to be inserted, and put them in the
        // temporary one, once that is done, the temp stack will have all
           the elements that are bigger than the element we want to
        // insert
        while (!sorted.isEmpty() && sorted.peek() > value) {
            temp.push(sorted.pop());
        }

        // add the element, meaning that the head of the sorted stack
           before this push will only have elements smaller than value,
        // or equal, and after we do the push the value would be the new
           head of the stack
        sorted.push(value);

        // from the temporary stack, return back all elements, back to the
           sorted stack, the temporary stack will have the order
        // correctly preserved, the temporary stack would have the old
           elements from the sorted stack in a sorted order, therefore
        // the top of temporary would be whatever was last pulled from the
           sorted stack. Poping from temp and adding to the sroted
        while (!temp.isEmpty()) {
            sorted.push(temp.pop());
        }
    } else {
        // the sorted is empty, so just push the element, this case would
           happen only if we have nothing in the sorted stack, there
        // is nothing to sort
        sorted.push(value);
    }
}
```

Queue from stacks

Queue implementation from two stacks, is a very famous problem, which aims at producing a queue interface by only using two stacks to hold all elements. This is achieved by cleverly using two stacks in a way where we do not constantly move from one to the other on each operation, but instead batch the operations

Here is how this is achieved, we have two stacks, old and new, when we insert new elements we only push into the new elements stack, when we want to remove, we move all elements into the old stack, the old stack will have now the elements but in inverted order, perfect for queue `remove` which pulls from the head of the queue.

However once the old stack has elements pushed into it, we will not do the same element dumping into `old` again until the old elements stack becomes empty.

Why ? Well the old elements stack acts as a some sort of buffer, we know that when removing from a queue, we always remove from the head, if the old stack still has elements, then there are still elements to be removed, there is no reason to constantly keep dumping the new elements stack into the old one, on each new element insert, while we have `old` elements to `remove` from the queue

This is a sort of optimized solution, the obvious solution, which is to always generate the `queue order` by popping out all elements from the stack, getting the `head` and then poping back the elements into the stack, will work, however it is much more inefficient, than the solution above, where the temp stack is used as a buffer instead.

```java
// only inserted elements are added to this stack, however they are moved
   into
// the old stack, when the old stack becomes empty, from removals, this
   stack
// represents the `tail` of the queue
Stack<Integer> stackNew = new Stack<>();

// only removals remove from this stack, we bulk insert all elements from
   the
// new stack when this stack becomes empty, and then we only remove from it
// until it becomes empty again, represents the `head` of the queue
Stack<Integer> stackOld = new Stack<>();

Integer peek() {
    // first shift all new elements from the new stack to the old stack,
       only if the old stack was already empty, if not do nothing
    shift();
    return this.stackOld.peek();
}

Integer remove() {
    // first shift all new elements from the new stack to the old stack,
       only if the old stack was already empty, if not do nothing
    shift();
    return this.stackOld.pop();
}

void shift() {
    // basically the new elements stack is used as temporary storage which
       is moved to the old elements only when old elements
    // becomes empty, either by popping from old elements stack when doing
       remove() queue operation this is the meat of the
    // algorithm, we have two cases of how to update the two stacks, see
       below:
    // - when the old stack is empty, we move all elements from the new
       stack, and push them in the old elements stack, that would
    // cause the old elements stack to have the reverse order of elements
       as the ones in the new elements stack, due to the fact
    // that we pop from the new elements stack, and push into the old
       elements stack, at this point the old elements stack will
    // contain at it's head the first element ever pushed in new elements
    // - when the old stack is not empty, we keep the elements in new
       elements stack, we do not pop or update old stack elements,
    // because since we implement a queue, we can only remove from the
       tail, so there are elements to remove from old elements stack
    // still, so until old elements becomes empty, we can keep popping,
       when it becomes empty, the new batch of elements will come
    // from new elements stack
    if (this.stackOld.isEmpty()) {
        while (!this.stackNew.isEmpty()) {
            this.stackOld.push(this.stackNew.pop());
        }
    }
}
```

Composite stacks

Another interesting problem is how would we store multiple stacks using a simple dynamic array, one solution is to interleave the stacks, such that the structure in the array looks something like this

```
    a1 b1 c1 a2 b2 c2 a3 b3 c3 .... aN bN cN
```

We can see that the elements of the stack (in this case this array holds 3 stacks) are interleaved into each other.

```

// the number of `stacks` the dynamic array is allowed to hold
int stacks;

// the actual dynamic array holding the stacks' data
Integer[] stackArray;

// array of `stacks` size which holds the head `index` of each stack
Integer[] stackHeads;

create(int stacks) {
    // the number of stacks this array is supposed to hold
    this.stacks = stacks;
    // the array to hold the stacks, starts off initially able to hold at
       least 1 element for each of the stacks
    this.stackArray = new Integer[stacks];
    // holds the indicies for where each head, for each stack is at the
       moment, in the big static array
    this.stackHeads = new Integer[stacks];
    for (int i = 0; i < stacks; i++) {
        // initialize the heads to be negative, that would indicate that
           the stacks start off initialy empty
        this.stackHeads[i] = Integer.MIN_VALUE;
    }
}

boolean push(int stack, Integer value) {
    if (stack >= this.stacks) {
        return false;
    }

    int index = this.stackHeads[stack];

    if (index < 0) {
        // when the current index is negative, the stack was empty, the
           first element is simply at the index which corresponds to
        // the stack index itself
        this.stackHeads[stack] = stack;
    } else {
        // the head index was not negative, therefore we offset it forward
           by the total number of stacks the array holds
        this.stackHeads[stack] += this.stacks;
    }

    // update the index varialbe
    index = this.stackHeads[stack];

    // resize the static array in case the index overshoots the actual
       size of the array, the resize is done in chunks of of
    // elements equal to the number total stacks, i.e each resize would
       add atleast this.stacks number of elements, or in other
    // words, 1 element for each stack it can hold
    if (index >= this.stackArray.length) {
        this.stackArray = Arrays.copyOf(this.stackArray,
           this.stackArray.length + this.stacks);
    }

    // set the element value at the index, the index inside the stackHeads
       would not point at the head of the stack, for the current
    // stack
    this.stackArray[index] = value;
    return true;
}

Integer pop(int stack) {
    if (stack >= this.stacks) {
        return null;
    }

    // get the current head of the specific stack first, and verify that
       specific stack is valid
    int index = this.stackHeads[stack];

    if (index < 0) {
        // negative index would indicate that there is nothing to pop from
           this specific stack
        return null;
    }

    // negate the curent head, by the total number of stacks, the index
       can become 0, when removing the last element from the
    // specific stack, which is okay, since that is the condition we use
       to verify if the specific stack is empty
    this.stackHeads[stack] -= this.stacks;

    // return whatever value was at the old head index
    return this.stackArray[index];
}
```

Queue

One of the most used data structure, which is often used with various different algorithms, such as graphs or tree traversals. The common interface that a queue uses is as follows

- remove() - remove the element from the front of the queue
- insert() - push a new element to the end of the queue
- peek() - peek the element at the front of the queue
- empty() - checks if the queue is empty, true / false

Note, that in the default implementation in java, when using most of those methods on an empty queue, they would throw QueueEmtpyException, or something of that nature - peek, remove for example.

```
public static final class Queue<T> {

    // simple node to describe an entry in the queue, due to the nature of
       the
    // queue, it does not need to be doubly linked, the implementation is
       very
    // similar to a singly linked list
    private static final QueueNode<T> {
        private T value;
        private QueueNode<T> next;
    }

    // simply hold a reference to the head element, when removing we would
    // extract value from the head, and move the head forward, when
       inserting,
    // the head is used to find the end of the queu, to insert at
    private QueueNode<T> head;

    public void insert(T value) {
        if (this.head == null) {
            // when the head of the queue if not initialized make sure to
               set it
            // and leave, that for the first time adding to the queue
            this.head = new QueueNode<T>();
            this.head.value = value;
            this.head.next = null;
        } else {
            // traverse the queue to find where it ends, the loop below
               would
            // find the last valid node, or in other words the tail of the
               queue
            // note that we could maintain a tail node, but this is done
               for
            // simplicity, and ease of use
            QueueNode<T> tail = this.head;
            while (tail && tail.next != null) {
                tail = tail.next;
            }
            // on each new element, simply attach it to the found tail, the
            // queue will always have a tail as long as the head is
               initialized,
            // which we guarantee in the main if above
            QueueNode<T> nextNode = new QueueNode<>();
            nextNode.value = value;
            curr.next = nextNode;
        }
    }

    public T remove() {
        if (this.head == null) {
            // in case the queue is empty, simply throw, which is what
               actually
            // happens in the default java queue implementation from the
               standard
            // library
            throw new QueueEmtpyException();
        }

        // otherwise when we have at least one head element, extract it,
           the new
        // head becomes the immediate next node after the current head,
           and simply
        // return the value of the current head
        QueueNode<T> current = this.head;
        this.head = current.next;
        return current.value;
    }

    public T peek() {
        if (this.head == null) {
            // in case the queue is empty, simply throw, which is what
               actually
            // happens in the default java queue implementation from the
               standard
            // library
            throw new QueueEmtpyException();
        }

        // peeking, does not mutate the state of the queue, it simply
           returns
        // whatever is at the start, where the current head points at
        return this.head.value;
    }

    public boolean empty() {
        // simple check to verify the queue has any elements at all, when
           all
        // elements are removed, the head will eventually point at null,
           or when
        // the queue is not initialized in the first place
        return this.head != null;
    }
}
```