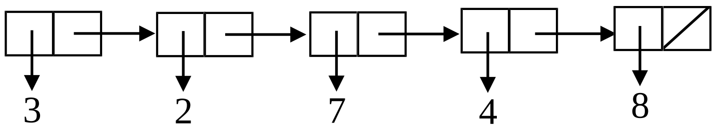
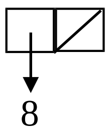
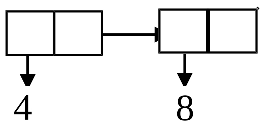
Student ID: 1422087
1. We want to put the numbers 8, 4, 7, 2, 3 into a stack, that means we need to insert them in reverse order – 3, 2, 7, 4, 8 (that's because of the First-In-Last-Out model of the stack). At the end we should have the following stack:



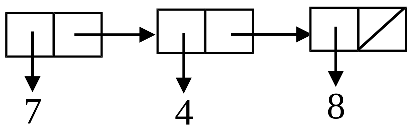
In Thethe beginning we have an EmptyStack. Then we use the constructor `push(8, EmptyStack)` to generate the following:



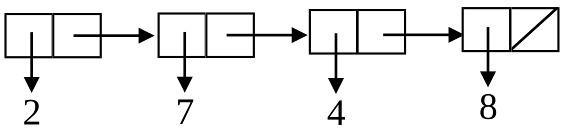
Then we do `push(4, push(8, EmptyStack)`:



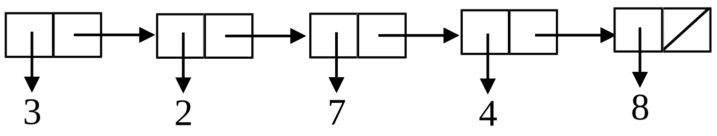
Then `push(7, push(4, push(8, EmptyStack)))`:



Then `push(2, push(7, push(4, push(8, EmptyStack)))`:

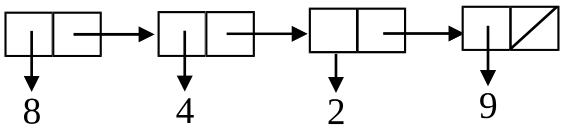


Then `push(3, push(2, push(7, push(4, push(8, EmptyStack))))):`

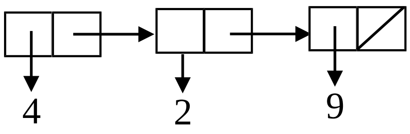


Finally, we have reached the desired stack.

2. If we add the numbers 9, 2, 4 and 8 in an empty stack, then eventually we will have the following stack:

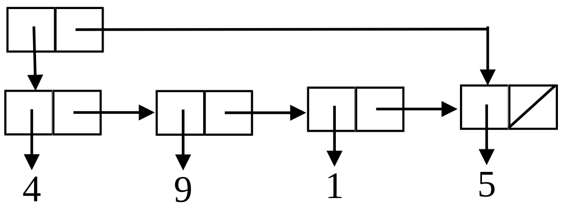


If we call the `top` operation on the stack, we will get the number 8. If we call the `pop` operation on the original stack we will get:

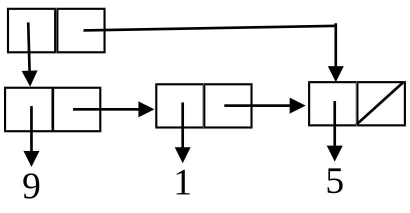


If we call the pop operation twice on the original stack and then call the `top` function, we will get the number 2 (we would have removed the numbers 8 and 4). If we call the function `pop` four times on the original stack, then we will get `EmptyStack` (we would have removed all the 4 numbers).

3. If we insert the numbers 4, 9, 1 and 5 one at a time in that order in a queue, then we will get the following queue:



Then, if we call the operation `top` on the queue, then we will get the number 4 (since it's the first element in the queue). If we call `pop` on the original queue we will get this:



If we apply the `pop` operation twice on the original queue and then call `top`, we will get the number 1 (we have removed the first two numbers from the queue and therefore 1 becomes the first element of the queue).

4. If we want to get the second last element of a list, then the function should look something like this:

```
secondLast(L) {
     if (isEmpty(L))
          error('Error: empty list in procedure last.')
     else if (isEmpty(rest(rest(L))))
          return rest(first(L))
     else
          return secondLast(rest(L))
}
```

Since we go through the function until there are exactly 2 items then the time complexity for the function is the `(length of the list – 1)`. That means the complexity is linear.

5. The code for the function should look something like this:

```
equalList(L1,L2) {
     if (isEmpty(L1) && isEmpty(L2)) {
          return true
     }
     else if (!isEmpty(L1) && isEmpty(L2)) {
          return false
     }
     else if (isEmpty(L1) && !isEmpty(L2)) {
          return false
     }
     else {
          if (first(L1) == first(L2)) equalList(rest(L1),rest(L2))
          else return false
     }
}
```
The `!isEmpty(L1)` means that the list L1 is not empty (same is for L2 too). == means equality. The complexity of this function is linear, since the execution time depends directly from the number of items in the smaller list (if they're not equal).

6. A function `member(x, S1)` should look something like this:

```
member(x,S1) {
     if (isEmpty(S1)) {
          return false
     }
     else if (first(S1)==x) {
          return true
     }
     else member(x,rest(S1))
}
```

For subset(S1, S2) we have:

```
subset(S1,S2) {
     if (isEmpty(S1)) {
          return true
     }
     else if (member(first(S1),S2)) {
          subset(rest(S1),S2)
     }
     else return false
}
```

And equalset(S1, S2) can be written like this:

```
equalset(S1, S2) {
     return (subset(S1, S2)) && (subset(S2, S1))
}
```

The function returns true only if both `subset(S1, S2)` and `subset(S2, S1)` are true. A similar solution using if-statements could look something like this:

```
equalset(S1,S2) {
     if (subset(S1,S2) {
          if(subset(S2,S1))
               return true
          else return false
     }
}
```