CSE 12 – Basic Data Structures and Object-Oriented Design
Lecture 8

Greg Miranda & Paul Cao, Winter 2021

This lecture is being recorded

Announcements

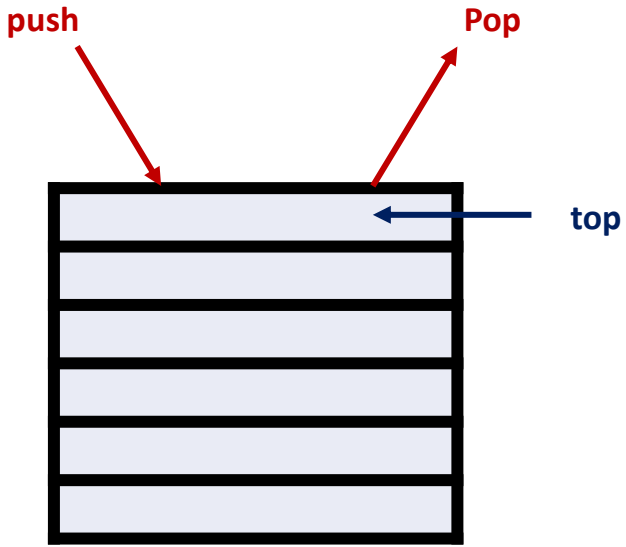
- Quiz 8 due Monday @ 8am
- Survey 3 due tonight @ 11:59pm
- PA3 due Wednesday @ 11:59pm
- Exam 1 next Friday
  - Released @ 8am on Friday
  - Closes @ 10am on Saturday
  - More details to be released on Piazza soon

Topics

Deque

- Stack
- Runtime analysis

Stack



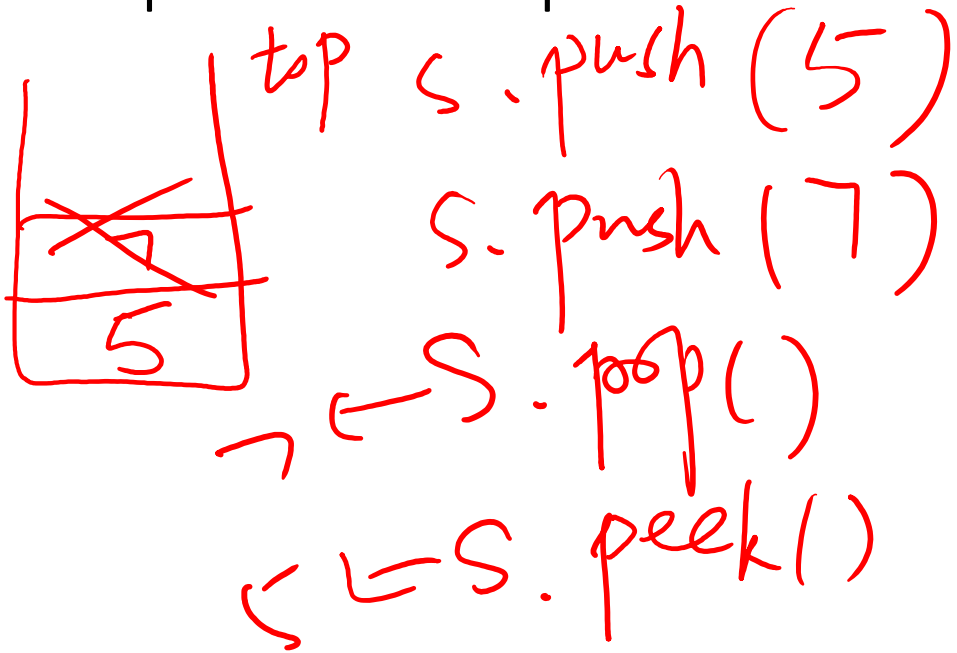
- A stack is a data Structure in which insertion and deletion take place at the same end.
- Stacks are known as LIFO (Last In, First Out) lists.
  - The last element inserted will be the first to be retrieved
  - We can only add/remove/examine the last element added (the "top").

Stack Operations

- Push
  - To insert an item from Top of stack is called push operation.
- POP
  - To put-off, get or remove some item from top of the stack is the pop operation.
- Peek
  - Looks at the object at the top of this stack without removing it from the stack.
- IsEmpty
  - Stack considered empty when there is no item on top. IsEmpty operation return true when there is no item in stack else false.
- IsFull
  - Stack considered full if no other element can be inserted on top of the stack.

https://docs.oracle.com/javase/10/docs/api/java/util/Stack.html

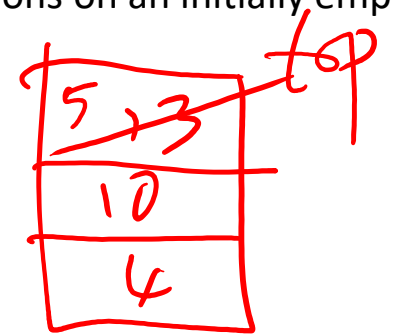
Basic Operation Explanation



Stack

- Consider doing the following operations on an initially empty stack, s:

```
s.push(4)
s.push(10)
s.push(13)
s.pop()
s.push(5)
```



What are the contents of the stack, from top (left) to bottom (right):

A. 4, 10, 13, 5

B. B. 5, 13, 10, 4

C. C. 5, 10, 4

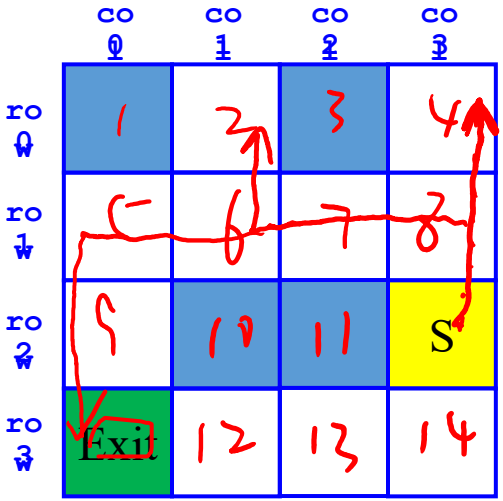
D. D. 5, 13, 10

E. E. other

Is a Stack an ADT or a data structure?

A. ADT
B. Data structure
C. I have no idea, what's the difference again??

Searching with a Stack



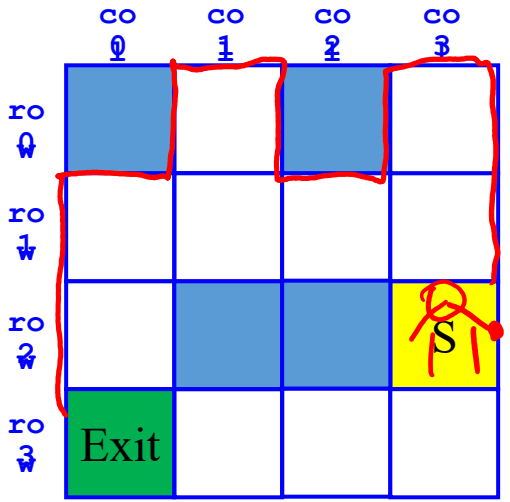
SearchForTheExit

- Initialize a **Stack** to hold Squares as we search
- Mark starting square as visited
- Put starting square on task list

- While **Stack** is not empty
  - Remove square sq from task list
  - Mark sq as visited
  - If sq is the Exit, we're done!
  - For each of square's unseen neighbors (S, W, N, E):
    - Set neighbor's previous to sq
    - Add neighbor to **Stack**

Label the cells in the order in which they are visited by the algorithm.
What is the final path to the goal?

```java
// Iterative DFS using stack
public  void dfsUsingStack(Node node){
   Stack<Node> stack=new  Stack<Node>();
   stack.add(node);
   while (!stack.isEmpty()){

      Node element=stack.pop();
      if(!element.visited){
                System.out.print(element.data + " ");
                element.visited = true;
      }
      List<Node> neighbours=element.getNeighbours();
      for (int i = 0; i < neighbours.size(); i++) {
                Node n=neighbours.get(i);
                if(n!=null && !n.visited){
                    stack.add(n);
                }
       }
   }
}
```

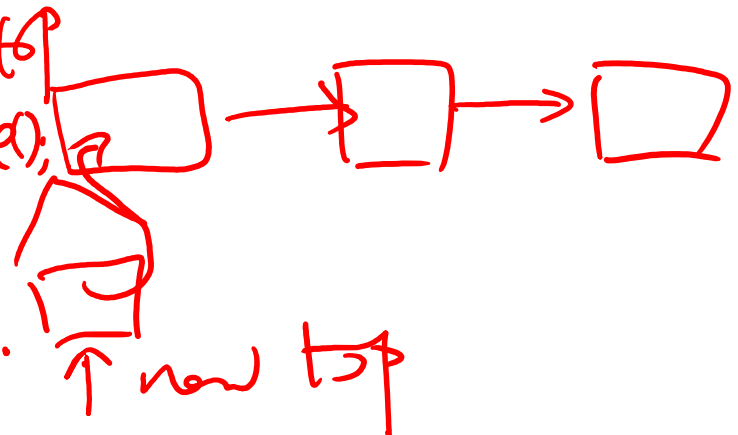


Stacks - Implementation

One option: Implement the methods in the ADT from scratch.

```
public class StackWorklist implements SearchWorklist{
    protected class StackNode {
        Square element;
        StackNode next;
    }

    private StackNode top;
    public void add( Square elem ) {
        StackNode ref = new StackNode();
        ref.element = elem;
        ref.next = top
        top = ref; // update
    }
}
```

linked list idea



Stacks - Implementation

adapter pattern

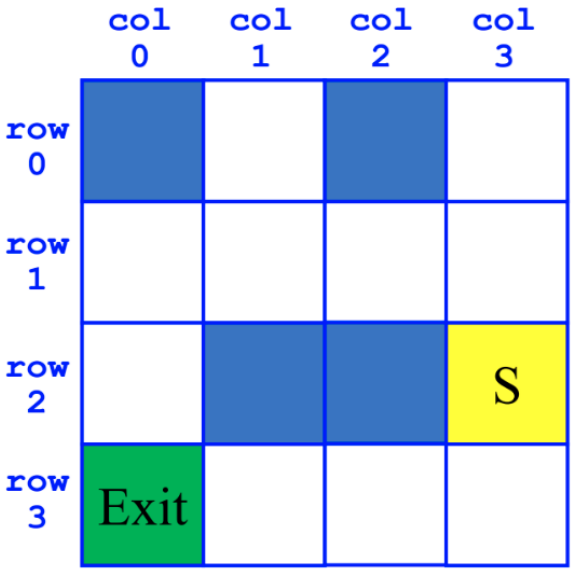
One option: Implement the methods in the ADT from scratch.

```
public class myStack<E> {
    protected class StackNode {
        E element;
        StackNode next;
    }

    private StackNode top;
    public boolean push( E elem ) {
        StackNode n = new StackNode();
        n.next = top;
        n.element = elem;
        top = n;
        return true;
    }
}
```

```
public class myList<E> {
    protected class ListNode {
        E element;
        ListNode next;
    }

    private ListNode head;
    public boolean addFirst( E elem ) {
        ListNode n = new ListNode();
        n.next = head;
        n.element = elem;
        head = n;
        return true;
    }
}
```

Why redo all this work??

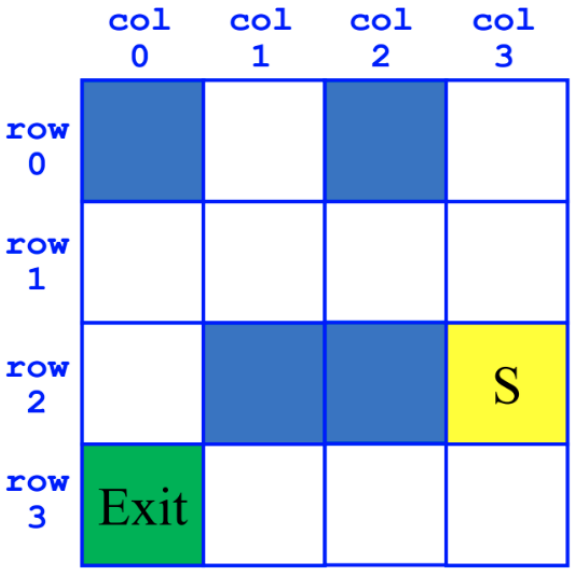
Breadth-First Search (BFS)



SearchForTheExit

Initialize a **Queue** to hold Squares as we search
Mark starting square as visited
Enqueue starting square on **Queue**
While **Queue** is not empty
    Dequeue square sq from **Queue**
    Mark sq as visited
    If sq is the Exit, we're done!
    For each of square's unvisited neighbors (S, W, N, E):
        Set neighbor's previous to sq
        Enqueue neighbor to **Queue**

Depth-First Search (DFS)



SearchForTheExit

Initialize a **Stack** to hold Squares as we search
Mark starting square as visited
Push starting square on **Stack**
While **Stack** is not empty
    Pop square sq from **Stack**
    Mark sq as visited
    If sq is the Exit, we're done!
    For each of square's unvisited neighbors (S, W, N, E):
        Set neighbor's previous to sq
        Push neighbor to **Stack**

How to compare ideas in computing

- Bench mark
  - code things up and run it
  - Eg. implement priority queue with
    - sorted array
    - sorted linked list
    - unsorted linked list
    - heap
- Pro and con of this approach?

Lists: One application

```
List<Songs> playlist
```

Lists: One application

`List<Songs> playlist`

Imagine that this list is implemented as an array and that there are currently 20 songs in my playlist. How many places do I have to look to determine whether “All About that Bass” is in my playlist?

A. 1
B. 3
C. 10
D. 20
E. Other

Lists: Running time

```
List<Songs> playlist
```

Imagine that this list is implemented as an array and that there are currently 20 songs in my playlist, but now you can't see what any of them are.
How many places do I have to look to determine whether "Riptide" is in my playlist in the BEST case?

A. 1
B. 10
C. 15
D. 20
E. Other

Lists: Running time

`List<Songs> playlist`

Imagine that this list is implemented as an array and that there are currently 20 songs in my playlist, but now you can't see what any of them are.
How many places do I have to look to determine whether "Riptide" is in my playlist in the ~~WORST~~ best case?

A. 1
B. 10
C. 15
D. 20
E. Other

Running time: What version of the problem are you analyzing

- One part of figuring out how long a program takes to run is figuring out how "lucky" you got in your input.
  - You might get lucky (**best case**), and require the least amount of time possible
  - You might get unlucky (**worst case**) and require the most amount of time possible
  - Or you might want to know "on average" (**average case**) if you are neither lucky or unlucky, how long does an algorithm take.

Almost always, what we care about is the WORST CASE or the AVERAGE CASE. Best case is usually not that interesting, unless we can prove it's slow!

In CSE 12 when we do analysis, we are doing **WORST CASE** analysis unless otherwise specified.

Optional Application of Stack

Expression Evaluation Using Stack

Given an algebraic expression, evaluate its value

12*(5 + 2) – 12/4 + 11%3

Assume: only has binary operator (+, - , *, /, %)

Input: an algebraic expression and order of precedence on operators

Output: the value of the expression

Infix vs Postfix

- If a mathematical expression is written as `operand1 operator operand2`, then it is in the **`infix`** format
  - 4 + 5

- If an expression is written as `operand1 operand2 operator`, then it is in the **`postfix`** format
  - 4 5 +

12*(5 + 2) – 12/4 + 11%3

infix format

Infix to Postfix algorithm

5+2*6

Initial expression: Q

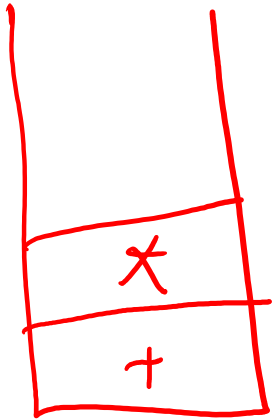
Final expression P

5 2 6 * +

Start with an empty stack. We scan Q from left to right.

A. While (we have not reached the end of Q)

1. If (**an operand is found**) Add it to P
2. If (**a left paren is found**) Push it onto the stack
3. If (**a right paren is found**)
   While (the stack is not empty AND no matching paren is on top)
   Pop the stack and add the popped value to P
   Pop the left parenthesis from the stack and discard it
4. If (**an operator is found**)
   If (the stack is empty or if the top element is a left paren)
   Push the operator onto the stack
   Else
   While (the stack is not empty AND the top of the stack is not a left paren
   AND precendece operator <= precedence of the top of the stack)
   Pop the stack and add the top value to P
   Push the latest operator onto the stack



B. While (the stack is not empty) Pop the stack and add the popped value to P

Start with an empty stack. We scan Q from left to right.

A. While (we have not reached the end of Q)

1. If (**an operand is found**) Add it to P
2. If (**a left paren is found**) Push it onto the stack
3. If (**a right paren is found**)
   - While (the stack is not empty AND no matching paren is on top)
     - Pop the stack and add the popped value to P
   - Pop the left parenthesis from the stack and discard it
4. If (**an operator is found**)
   - If (the stack is empty or if the top element is a left paren)
     - Push the operator onto the stack
   - Else
     - While (the stack is not empty AND the top of the stack is not a left paren
       AND precendece operator <= precedence of the top of the stack)
       - Pop the stack and add the top value to P
     - Push the latest operator onto the stack

B. While (the stack is not empty) Pop the stack and add the popped value to P

Convert **5 + 3 * (6 / 2 + 4)**

Start with an empty stack. We scan Q from left to right.

A. While (we have not reached the end of Q)

1. If (**an operand is found**) Add it to P
2. If (**a left paren is found**) Push it onto the stack
3. If (**a right paren is found**)
   - While (the stack is not empty AND no matching paren is on top)
     - Pop the stack and add the popped value to P
   - Pop the left parenthesis from the stack and discard it
4. If (**an operator is found**)
   - If (the stack is empty or if the top element is a left paren)
     - Push the operator onto the stack
   - Else
     - While (the stack is not empty AND the top of the stack is not a left paren AND precendece operator <= precedence of the top of the stack)
       - Pop the stack and add the top value to P
     - Push the latest operator onto the stack

B. While (the stack is not empty) Pop the stack and add the popped value to P

3+2+5

A. 3 2 5 + +

B. 3 2 + 5 +

C. 3 + 2 5 +

D. 5 2 3 + +

E. Something else

Start with an empty stack. We scan Q from left to right.

A. While (we have not reached the end of Q)

1. If (**an operand is found**) Add it to P
2. If (**a left paren is found**) Push it onto the stack
3. If (**a right paren is found**)
   - While (the stack is not empty AND no matching paren is on top)
     - Pop the stack and add the popped value to P
   - Pop the left parenthesis from the stack and discard it
4. If (**an operator is found**)
   - If (the stack is empty or if the top element is a left paren)
     - Push the operator onto the stack
   - Else
     - While (the stack is not empty AND the top of the stack is not a left paren AND precendece operator <= precedence of the top of the stack)
       - Pop the stack and add the top value to P
     - Push the latest operator onto the stack

B. While (the stack is not empty) Pop the stack and add the popped value to P

3+2+5*(4-12)+6%2

A. 3 2 + 5 4 12 - * + 6 2 % +

B. 3 2 + + 5 4 12 - * + 6 2 %

C. 3 2 5 + 4 12 - + 6 2 % +

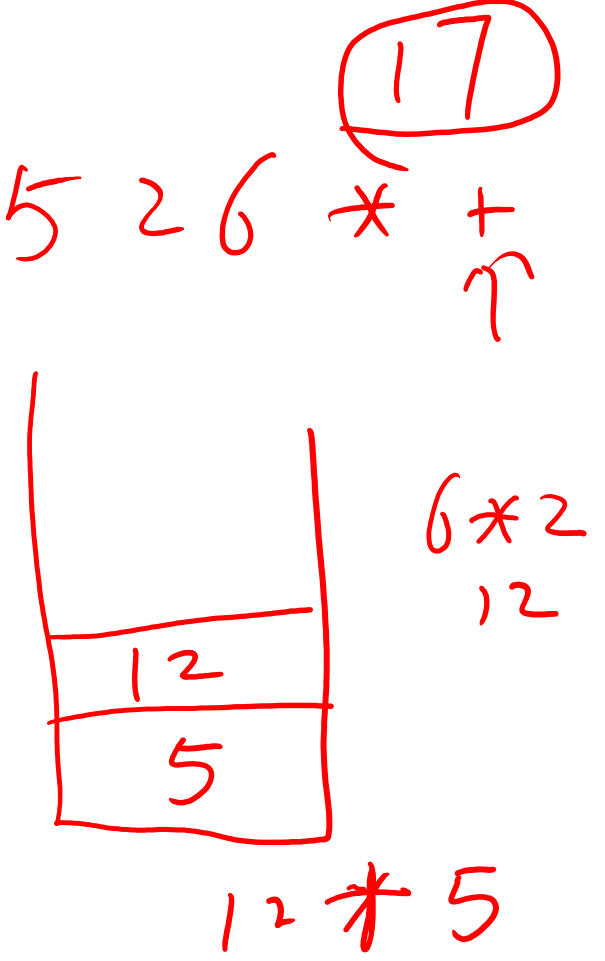
D. 3 2 5 + + 4 12 + - 6 2 %

E. Something else

Postfix to evaluation

Start with an empty stack. We scan P from left to right.

While (we have not reached the end of P)
    If an operand is found
        push it onto the stack
    If an operator is found
        Pop the stack and call the value A
        Pop the stack and call the value B
        Evaluate B op A using the operator just found.
        Push the resulting value onto the stack
Pop the stack (this is the final value)



Start with an empty stack. We scan P from left to right.

Evaluate **5 3 6 2 / 4 + * +**

While (we have not reached the end of P)
    If an operand is found
        push it onto the stack
    If an operator is found
        Pop the stack and call the value A
        Pop the stack and call the value B
        Evaluate B op A using the operator just found.
        Push the resulting value onto the stack
Pop the stack (this is the final value)