CS200: Stacks

- Prichard Ch. 7

Linear, time-ordered structures

- Two data structures that reflect a ***temporal*** relationship
  - order of removal based on order of insertion
- We will consider:
  - “first come, first serve”
    - first in first out - FIFO (queue)
  - “take from the top of the pile”
    - last in first out - LIFO (stack)

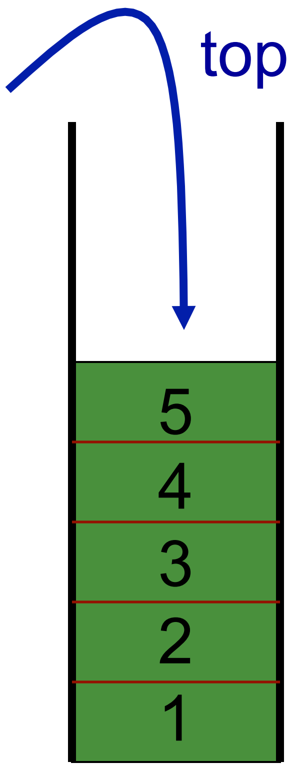
Stacks or queues?

What can we do with coin dispenser?

- "**push**" a coin into the dispenser.
- "**pop**" a coin from the dispenser.
- "**peek**" at the coin on top, but don't pop it.
- "**isEmpty**" check whether this dispenser is empty or not.

Stacks

- Last In First Out (LIFO) structure
  - A stack of dishes in a café
  - A stack of quarters in a coin dispenser
- Add/Remove done from **same end**: the top



Possible Stack Operations

- **`isEmpty()`**: determine whether stack is empty
- **`push()`**: add a new item to the stack
- **`pop()`**: remove the item added most recently
- **`peek()`**: retrieve, but don't remove, the item added most recently

- What would we call a collection of these ops?

Checking for balanced braces

- How can we use a stack to determine whether the braces in a string are **balanced**?

abc{defg**{**ijk**}** **{**l**{**mn**}** **}**op}qr

abc{def}}{ghij{kl}m

Can you define balanced?

Pseudocode

```
while ( not at the end of the string){
      if (the next character is a “{“){
            aStack.push(“{“)
      }
      else if (the character is a “}”) {
            if(aStack.isEmpty()) ERROR!!!
            else aStack.pop()
      }
}
if(!aStack.isEmpty()) ERROR!!!
```

question

- Could you use a single int to do the same job?

- How?

Try it on

abc{defg{ijk}{l{mn}}op}qr {st{uvw}xyz}

abc{def}}{ghij{kl}m

Expressions

- Types of Algebraic Expressions
  - Prefix
  - Postfix
  - Infix
- Prefix and postfix are easier to parse. No ambiguity. Infix requires extra rules: **precedence** and **associativity**. **What are these?**
- Postfix: operator applies to the operands that immediately precede it.
- Examples:
  1. - 5 * 4 3
  2. 5 - 4 * 3
  3. 5 4 3 * -

What type of expression is "5 4 3 – *"?

A. Prefix
B. Infix
C. Postfix
D. None of the above (i.e., illegal)

What is the infix form of “5 4 3 – *”?

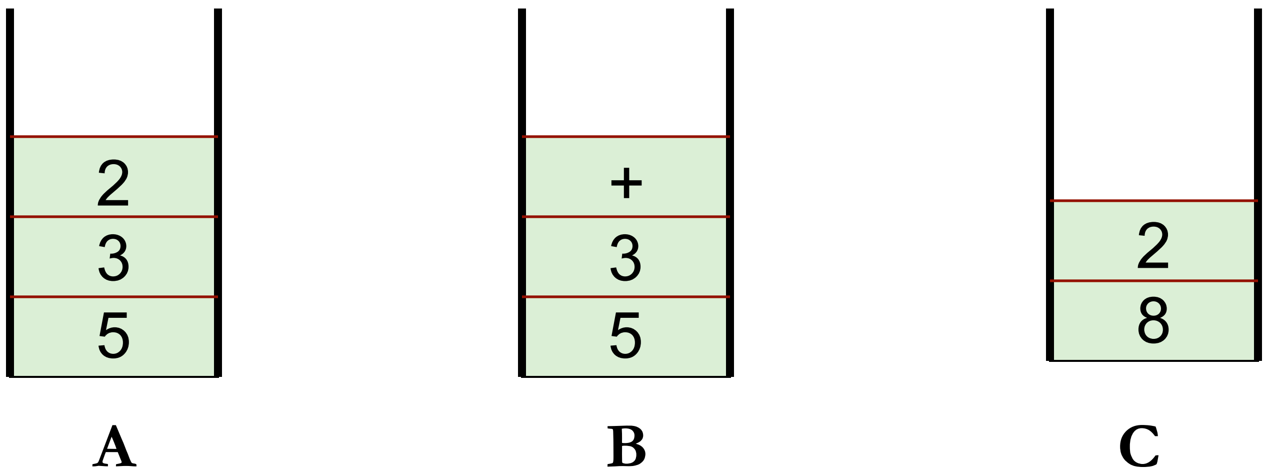
Evaluating a Postfix Expression

```
while there are input tokens left
    read the next token
    if the token is a value
        push it onto the stack.
    else
        //the token is a operator taking n arguments
        pop the top n values from the stack and perform the operation
        push the result on the stack
If there is only one value in the stack return it as the result
else
    throw an exception
```

Draw Stacks to evaluate “5 4 3 – *”?

Quick check

- If the input string is “5 3 + 2 *”, which of the following could be what the stack looks like when trying to parse it?



Stack Interface

push(StackItemType newItem)

- adds a new item to the top of the stack

StackItemType pop() throws StackException

- deletes the item at the top of the stack and returns it
- Exception when deletion fails

StackItemType peek() throws StackException

- returns the top item from the stack, but does not remove it
- Exception when retrieval fails

boolean isEmpty()

- returns true if stack empty, false otherwise

Preconditions? Postconditions?

Comparison of Implementations

- Options for Implementation:
  - **Array** based implementation
  - **Array List** based implementation
  - **Linked List** based implementation
- What are the advantages and disadvantages of each implementation?
- Let's look at a Linked List based implementation
- In P1 you program an Array List based implementation

Stacks and Recursion

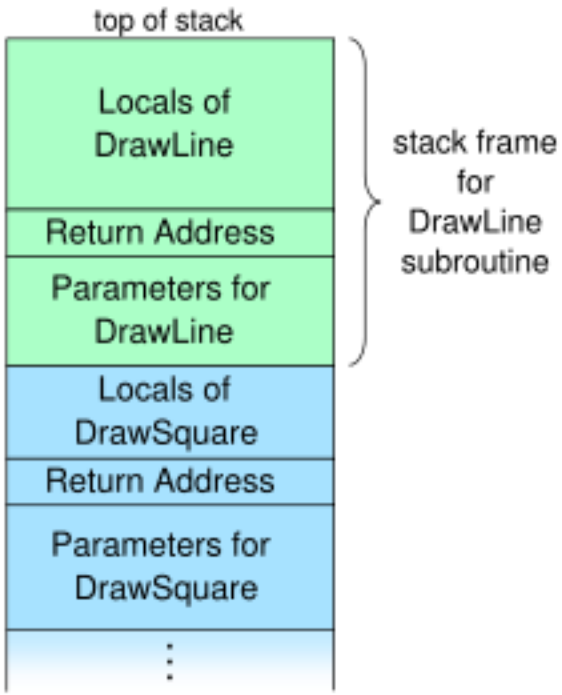
- Most implementations of recursion maintain a stack of activation records, called

  the Run Time Stack

- Activation records, or Stack Frames, contain parameters, local variables and return information of the method called
- The most recently executed activation record is stored at the top of the stack. So a call pushes a new activation record on the RT stack

Applications - the run-time stack

- Nested method calls tracked on call stack (aka run-time stack)
  - First method that returns is the last one invoked
- Element of call stack - activation record or stack frame
  - parameters
  - local variables
  - return address: pointer to next instruction to be executed in calling method



http://en.wikipedia.org/wiki/Image:Call_stack_layout.svg

Factorial example

```
int factorial(n){
  // pre n>=0
  // post return n!
  if(n==0) { r=1; return r;}
  else {r=n*factorial(n-1); return r;}
}
```

RTS factorial(3): wind phase

only active frame: top of the run time stack

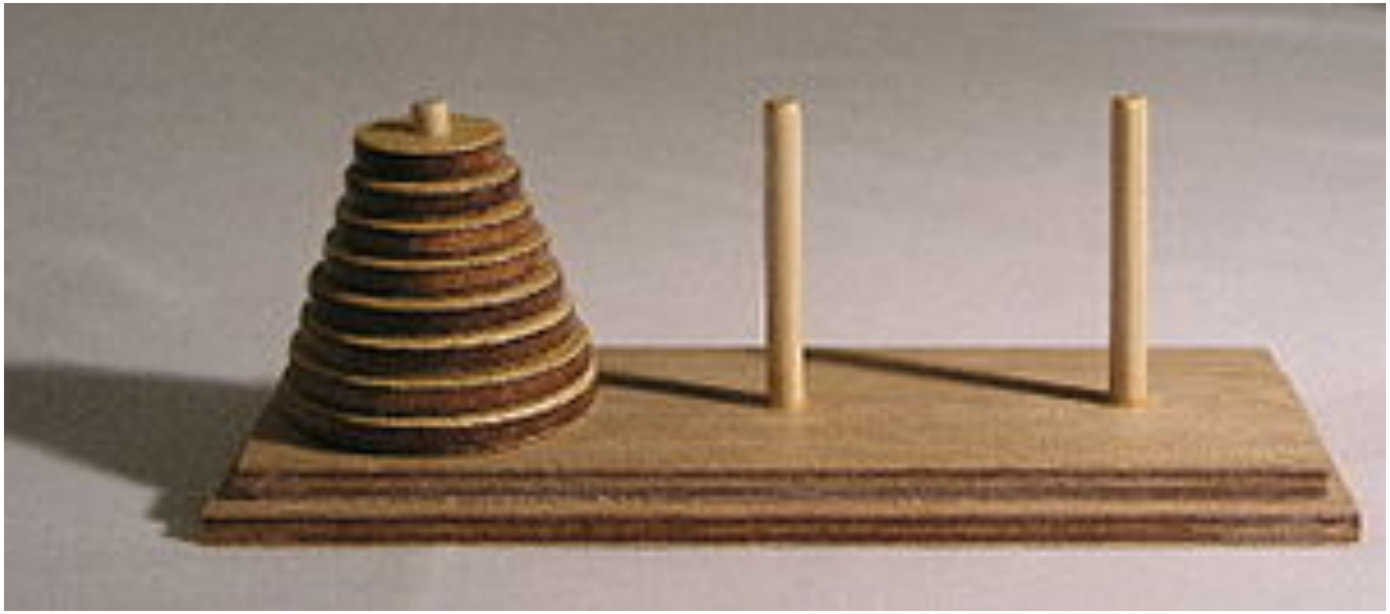
			n=0, r=1
		n=1, r=?	n=1, r=?
	n=2, r=?	n=2, r=?	n=2, r=?
n=3, r=?	n=3, r=?	n=3, r=?	n=3, r=?

RTS factorial(3): unwind phase

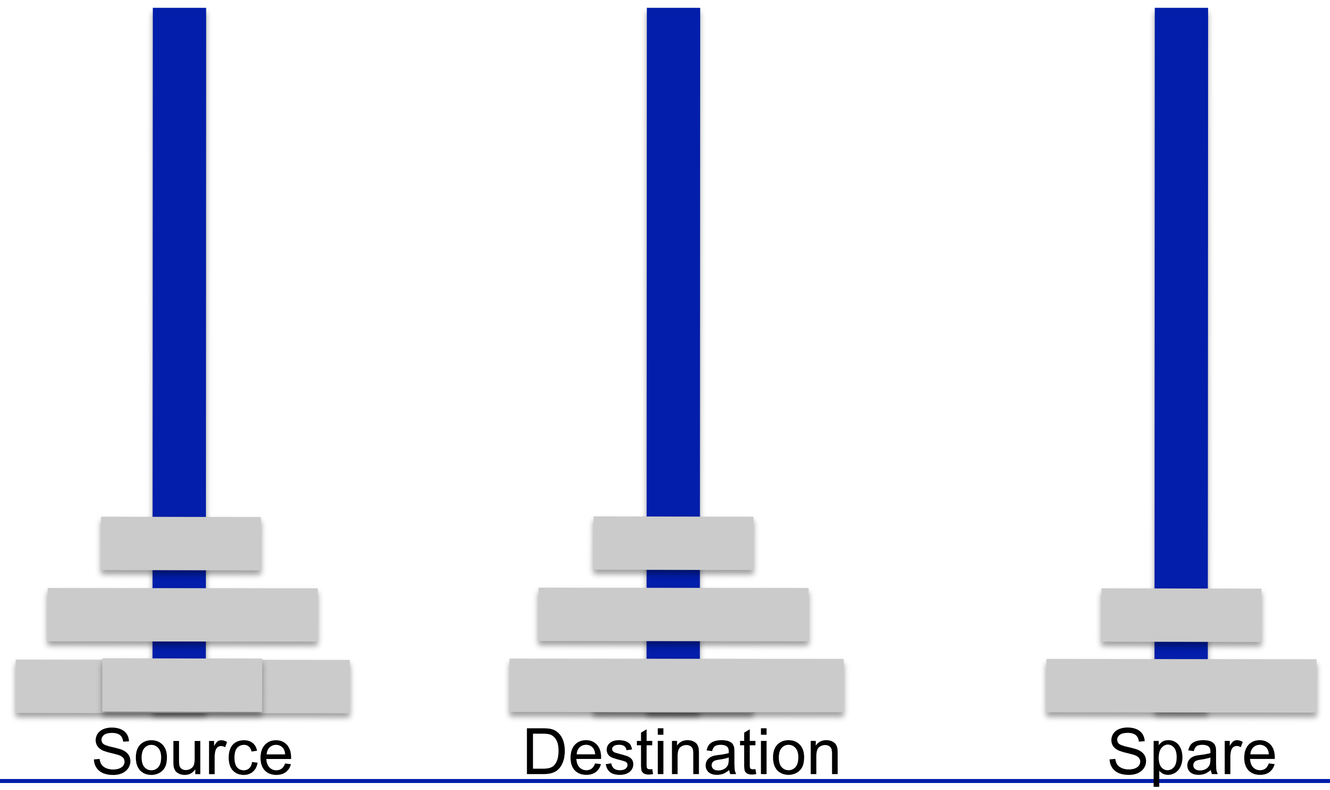
n=1, r=1			
n=2, r=?	n=2, r=2		
n=3, r=?	n=3, r=?	n=3, r=6	return 6

More complex example: The Towers of Hanoi

- **Move pile of disks from source to destination**
- **Only one** disk may be moved at a time.
- No disk may be placed on top of a smaller disk.



Moves in the Towers of Hanoi



Recursive Solution

```
// pegs are numbers, via is computed
// f: from: source peg, t: to: destination peg, v: via: intermediate peg
// state corresponds to return address, v is computed
public void hanoi(int n, int f, int t){
   if (n>0) {
      // state 0
      int v = 6 - f - t;
      hanoi(n-1,f, v);
      // state 1
      System.out.println("move disk " + n + " from " + f + " to " + t);
      hanoi(n-1,v,t);
      //  state 2
   }
}
```

Run time stack for hanoi(3,1,3)

```
if (n>0) {
    // state 0
    int v = 6 - f - t;
    hanoi(n-1,f, v);
    // state 1
    System.out.println("move disk " + n +
                        " from" + f + " to" + t);
    hanoi(n-1,v,t);
    // state 2
}
```

only active frame:
top of the run time stack

			0:n=0,f=1,t=2
		0:n=1,f=1,t=3	1:n=1,f=1,t=3
	0:n=2,f=1,t=2	1:n=2,f=1,t=2	1:n=2,f=1,t=2
0:n=3,f=1,t=3	1:n=3,f=1,t=3	1:n=3,f=1,t=3	1:n=3,f=1,t=3

Run time stack for hanoi(3,1,3)

```
if (n>0) {
    // state 0
    int v = 6 - f - t;
    hanoi(n-1,f, v);
    // state 1
    System.out.println("move disk " + n +
                          “ from" + f + " to" + t);
    hanoi(n-1,v,t);
    // state 2
  }
```

System.out:

“move disk 1 from 1 to 3”

“move disk 2 from 1 to 2”

etcetera

	0:n=0,f=2,t=3		
1:n=1,f=1,t=3	2:n=1,f=1,t=3	2:n=1,f=1,t=3	
1:n=2,f=1,t=2	1:n=2,f=1,t=2	1:n=2,f=1,t=2	1:n=2,f=1,t=2
1:n=3,f=1,t=3	1:n=3,f=1,t=3	1:n=3,f=1,t=3	1:n=3,f=1,t=3

Hanoi with explicit run time stack

- In Programming Assignment 1 you will create a Hanoi program with an explicit run time stack ***rts***.
- The main loop of the program is:

```
while(rts not empty){
  pop frame
  check frame state
  perform appropriate actions, including pushing frames
}
```

While loop:

Hanoi using an explicit Run Time Stack rts

Assume initially there is one Frame [state,n,from,to] on rts

Keep popping frames until rts is empty

When popping a frame:

if n == 0 do nothing (discard frame)

else if frame in state 0:

// do first call hanoi(n-1,from,via):

pushing [1,n,from,to] and push [0,n-1,from,via]

else if in state 1:

print disk n move

//do second call hanoi(0,n-1,via,to)

push [2,n,from,to] and push [0,n-1,via,to]

else (in state 2):

do nothing