Stacks

This lesson is borrowed from the following:

Reference

CS 367 – Introduction to Data Structures

http://pages.cs.wisc.edu/~mattmcc/cs367/notes/Stacks.ppt

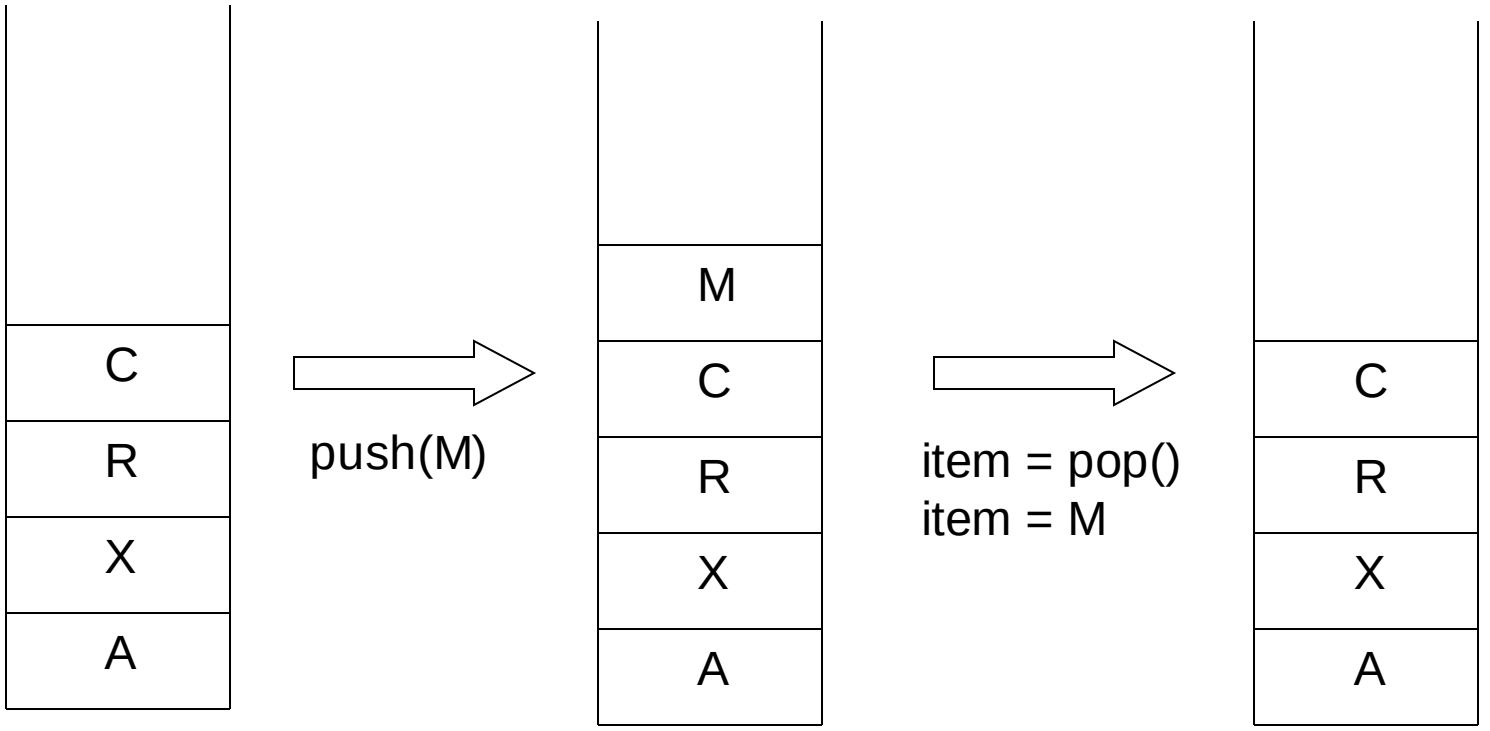
Stack

- A stack is a data structure that stores data in such a way that the last piece of data stored, is the first one retrieved
  - also called last-in, first-out
- Only access to the stack is the top element
  - consider trays in a cafeteria
    - to get the bottom tray out, you must first remove all of the elements above

Stack

- *Push*
  - the operation to place a new item at the top of the stack
- *Pop*
  - the operation to remove the next item from the top of the stack

Stack



Implementing a Stack

- At least three different ways to implement a stack
  - array
  - Vector (not covered in this lesson)
  - linked list
- Which method to use depends on the application
  - what advantages and disadvantages does each implementation have?

Implementing Stacks: Array

- Advantages
  - best performance
- Disadvantage
  - fixed size
- Basic implementation
  - initially empty array
  - field to record where the next data gets placed into
  - if array is full, push() returns false
    - otherwise adds it into the correct spot
  - if array is empty, pop() returns null
    - otherwise removes the next item in the stack

Stack Class (array based)

```
class StackArray {
    private Object[ ] stack;
    private int nextIn;
    public StackArray(int size) {
        stack = new Object[size];
        nextIn = 0;
    }
    public boolean push(Object data);
    public Object pop();
    public void clear();
    public boolean isEmpty();
    public boolean isFull();
}
```

push() Method (array based)

```
public boolean push(Object data) {
    if(nextIn == stack.length) { return false; }   // stack is full

    // add the element and then increment nextIn
    stack[nextIn] = data;
    nextIn++;
    return true;
}
```

pop() Method (array based)

```
public Object pop() {
    if(nextIn == 0) { return null; }   // stack is empty

    // decrement nextIn and return the data
     nextIn--;
    Object data = stack[nextIn];
    return data;
}
```

Notes on *push()* and *pop()*

- Other ways to do this even if using arrays
  - may want to keep a *size* variable that tracks how many items in the list
  - may want to keep a *maxSize* variable that stores the maximum number of elements the stack can hold (size of the array)
    - you would have to do this in a language like C++
  - could add things in the opposite direction
    - keep track of *nextOut* and decrement it on every push; increment it on every pop

Remaining Methods (array based)

```
public void clear() {
    nextIn = 0;
}

public boolean isEmpty() {
    return nextIn == 0;
}

public boolean isFull() {
    return nextIn == stack.length;
}
```

Additional Notes

- Notice that the array is considered empty if *nextIn* equals zero
  - doesn't matter if there is more data stored in the array – it will never be retrieved
    - *pop()* method will automatically return
- For a truly robust implementation
  - should set array elements equal to null if they are not being used
    - why? how?

Implementing a Stack: Linked List

- Advantages:
  - always constant time to push or pop an element
  - can grow to an infinite size
- Disadvantages
  - the common case is the slowest of all the implementations
  - can grow to an infinite size
- Basic implementation
  - list is initially empty
  - *push()* method adds a new item to the head of the list
  - *pop()* method removes the head of the list

Stack Class (list based)

```
class StackList {
    private LinkedList list;
    public StackList() { list = new LinkedList(); }
    public void push(Object data) { list.addHead(data); }
    public Object pop() { return list.deleteHead(); }
    public void clear() { list.clear(); }
    public boolean isEmpty() { return list.isEmpty(); }
}
```

Additional Notes

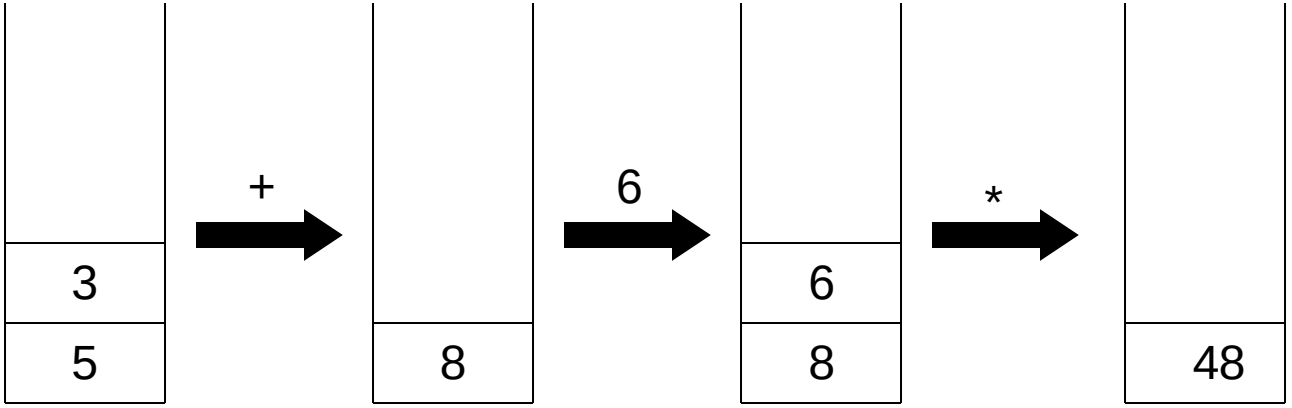
- It should appear obvious that linked lists are very well suited for stacks
  - *addHead()* and *deleteHead()* are basically the *push()* and *pop()* methods
- Our original list implementation did not have a *clear()* method
  - it's very simple to do
  - how would you do it?
- Again, no need for the *isFull()* method
  - list can grow to an infinite size

Stack Applications

- Stacks are a very common data structure
  - compilers
    - parsing data between delimiters (brackets)
  - operating systems
    - program stack
  - virtual machines
    - manipulating numbers
      - pop 2 numbers off stack, do work (such as add)
      - push result back on stack and repeat
  - artificial intelligence
    - finding a path

Reverse Polish Notation

- Way of inputting numbers to a calculator
  - (5 + 3) * 6 becomes 5 3 + 6 *
  - 5 + 3 * 6 becomes 5 3 6 * +
- We can use a stack to implement this
  - consider 5 3 + 6 *

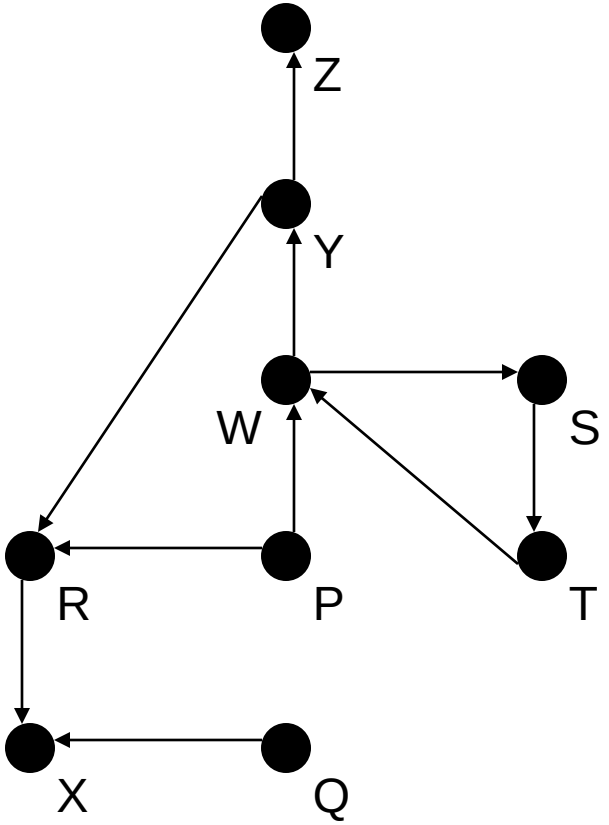


  - try doing 5 3 6 * +

```
public int rpn(String equation) {
    StackList stack = new StackList();
    StringTokenizer tok = new StringTokenizer(equation);
    while(tok.hasMoreTokens()) {
      String element = tok.nextToken();
      if(isOperator(element)) {
        char op = element.charAt(0);
        if(op == '=') {
          int result = ((Integer)stack.pop()).intValue();
          if(!stack.isEmpty() || tok.hasMoreTokens()) { return Integer.MAX_VALUE; } // error
          else { return result; }
        }
        else {
          Integer op1 = (Integer)stack.pop()
          Integer op2 = (Integer)stack.pop();
          if((op1 == null) || (op2 == null)) { return Integer.MAX_VALUE; }
          stack.push(doOperation(op, op1, op2));
        }
      }
      else {
        Integer operand = new Integer(Integer.parseInt(element));
        stack.push(operand);
      }
    }
    return Integer.MAX_VALUE;
}
```

Finding a Path

- Consider the following graph of flights

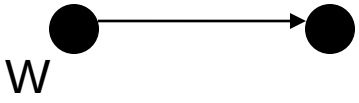


Key

: city (represented as C)

C_1 → C_2 : flight from city C_1 to city C_2

Example



flight goes from W to S

Finding a Path

- If it exists, we can find a path from any city C_1 to another city C_2 using a stack
  - place the starting city on the bottom of the stack
    - mark it as visited
    - pick any arbitrary arrow out of the city
      - city cannot be marked as visited
    - place that city on the stack
      - also mark it as visited
    - if that's the destination, we're done
    - otherwise, pick an arrow out of the city currently at
      - next city must not have been visited before
      - if there are no legitimate arrows out, pop it off the stack and go back to the previous city
    - repeat this process until the destination is found or all the cities have been visited

Example

- Want to go from P to Y
  - push P on the stack and mark it as visited
  - pick R as the next city to visit (random select)
    - push it on the stack and mark it as visited
  - pick X as the next city to visit (only choice)
    - push it on the stack and mark it as visited
  - no available arrows out of X – pop it
  - no more available arrows from R – pop it
  - pick W as next city to visit (only choice left)
    - push it on the stack and mark it as visited
  - pick Y as next city to visit (random select)
    - this is the destination – all done

Psuedo-Code for the Example

```
public boolean findPath(City origin, City destination) {
    StackArray stack = new Stack(numCities);
    clearAllCityMarks();
    stack.push(origin);
    origin.mark();
    while(!stack.isEmpty()) {
        City next = pickCity();
        if(next == destination) { return true; }
        if(next != null) { stack.push(next); }
        else { stack.pop(); }   // no valid arrows out of city
    }
    return false;
}
```