CSE 230: Data Structures

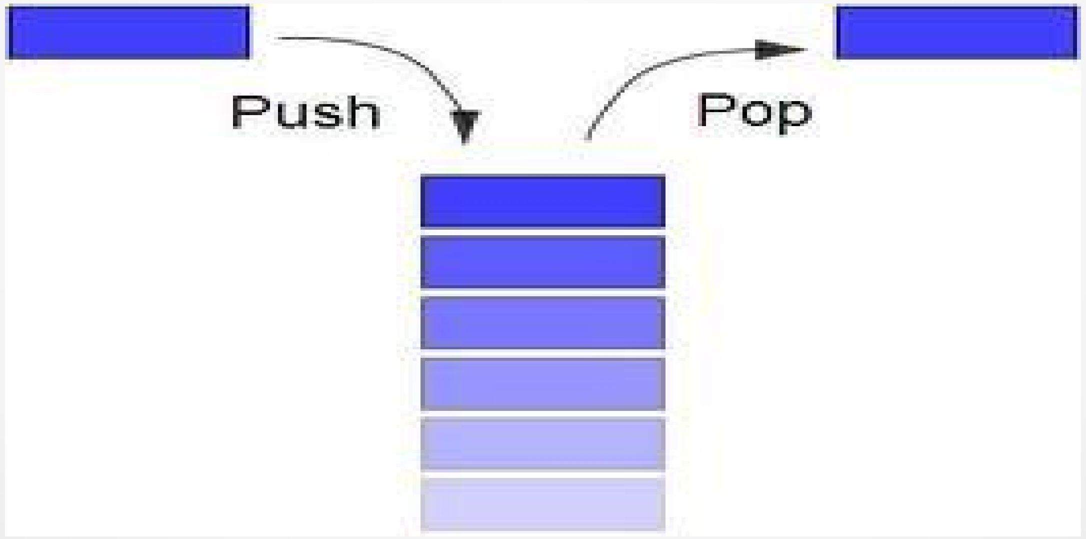
Lecture 4: Stacks

Dr. Vidhya Balasubramanian

Stacks in Nature

Stacks: An Overview

- It is a last-in-first-out abstract data type
  - Remembers the order in which data was entered
  - Linear Data Structure
- First proposed in 1946 by Alan M Turing



http://upload.wikimedia.org/wikipedia/commons/thumb/2/29/Data_stack.svg/200px-Data_stack.svg.png

Stack ADT: Main Operations

- push(o)
  - Push object o onto the stack
  - Input: object; Output: None
- pop()
  - Remove the last element that was pushed
  - Input: none; Output: object

Other Stack Operations

- size()
  - Returns the number of objects in the stack
- isEmpty()
  - Returns a Boolean indicating if a stack is empty
- top()
  - Return top object of the stack without removing it
  - Input: None; Output: Object

Stack Exceptions

- Some operations may cause an error causing an exceptions
- Exceptions in the Stack ADT
  - StackEmptyException
    - pop() and top() cannot be performed if the stack is empty
  - StackFullException
    - Occurs when the stack has a maximum size limit
    - push(o) cannot occur when the stack is full

Stack Example

Operation	Output	Stack Contents
push(5)	-	(5)
push(7)	-	(5,7)
push(3)	-	(5,7,3)
pop()	3	(5,7)
size()	2	(5,7)
push(4)	-	(5,7,4)
pop()	4	(5,7)
pop()	7	(5)
size()	1	(5)
pop()	5	()
pop()	“error”	()

Stack ADT in Java

```
public interface Stack<E>

    int size(): //returns number of objects in the stack

    boolean isEmpty() //returns true if stack is empty, false
    otherwise

    E top();

        //returns top object in stack, throws exception if stack
        empty

    void push(E element): //inserts object at top of stack

    E pop();

        //throws(StackEmptyException)

}
```

Exercise

- Describe the output of the following series of stack operations
  - push(8), push(3), pop(), push(2), push(5), pop(), pop(), push(9), push(1), pop(), push(7), top(), push(6), pop(), pop(), push(4), pop(), pop()
- Sedgewick, Exercise 4.6
  - A letter means push and an asterisk means pop in the following sequence. Give the sequence of values returned by the pop operations when this sequence of operations is performed on an initially empty LIFO stack.
    - E A S * Y * Q U E * * * S T * * * I O * N * * *

Exercise

- Assume that x, y, z are integer variables and that s is a stack of integers, state the output of the program fragment.

  -
    ```
    X = 3;; y = 5; z = 2;
    s.makeEmpty( );
    s.push(x); s.push(4);
    s.pop();
    s.push(y); s.push(3); s.push(z);
    s.pop();
    s.push(2); s.push(x);
    while(! s.isEmpty( ))
    {  x = s.pop();
       System.out.println(x);
    }
    ```

Exercise

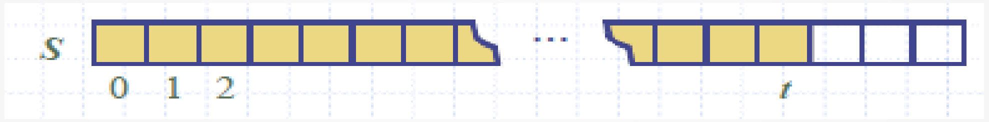
- Suppose an initially empty stack S has executed a total of 25 push operations, 12 top operations, and 10 pop operations, 3 of which raised Empty errors that were caught and ignored. What is the current size of S?
- Reverse the order of elements on stack S using
  - Two additional stacks
  - One additional stack and some additional non-array variables

Implementation of Stacks

- Considerations
  - Should the stack be statically or dynamically allocated
  - How is the top of the stack tracked
    - Use index to keep track of the top
    - Use pointers to reference the top
  - How are the bounds of the stacks tracked
  - When should the stack elements be constructed and deconstructed
    - Construct all elements at once, and destroy elements when stack is destroyed
    - Construct each element with push, and destroy when pop is called

Array based implementation

- A stack may be implemented by using a simple array
  - An N-element array
    - Stack is limited by the size of the array
  - Integer t that denotes the index of the top element



- Strategy
  - Elements are added left to right
  - Variable t keeps track of the topmost element
    - Initially t set to -1

Stack ADT Functions

```
• Algorithm size()
      return $t$+1
• Algorithm isEmpty()
      return ($t$<0)
• Algorithm top()
      if isEmpty() then
          throw a StackEmptyException
      return $S[t]$
```

Stack ADT Functions

```
• Algorithm push(o)
    if size() = N then
        throw a StackFullException
    t ← t+1
    S[t] ← o
• Algorithm pop()
    if isEmpty() then
        throw a StackEmptyException
    o ← S[t]
    t ← t-1
    return o
```

Complexity Analysis

- Time Complexity
  - size – $O(1)$
  - isEmpty – $O(1)$
  - top – $O(1)$
  - push – $O(1)$
  - pop – $O(1)$
- Space Complexity
  - $O(N)$

Pros and Cons of Array based Implementation

- Advantages
  - Simple
  - Efficient
  - Widely used
- Issues
  - Assumes a fixed upper bound on the size of the stack
  - Application will crash when this bound is exceeded
  - Useful when we have an estimate on the maximum size requirement of the stack

Amortization

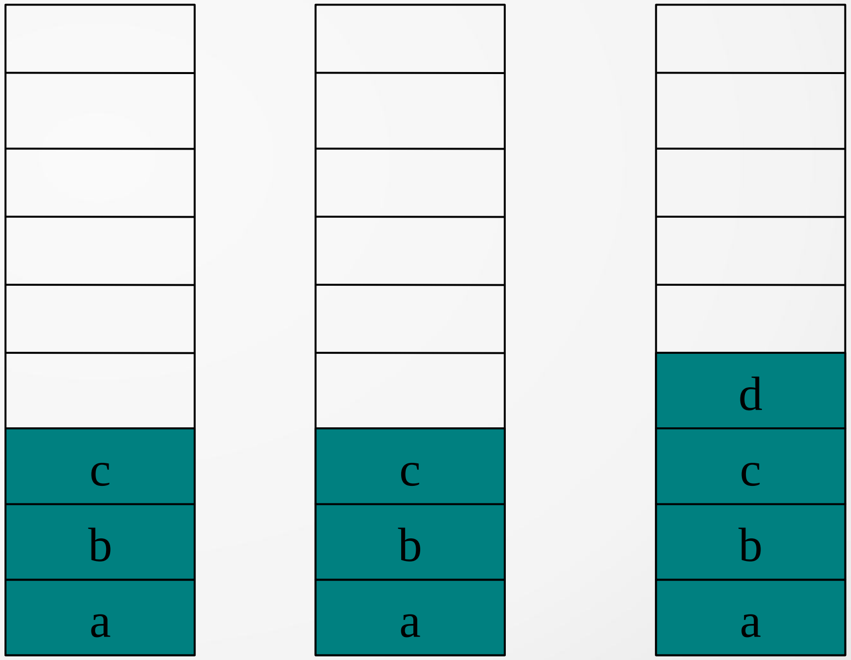
- Time required to perform a sequence of data structure operations is averaged over all the operations
  - Guarantees average performance of each operation in the worst case
  - Considers interactions between operations by studying running time of series of operations
- Techniques
  - Aggregate Method
  - Accounting Method
  - Potential Method

Aggregate Method

- Amortized running time =

$$\frac{(worstcaserunningtimeofaseriesofoperations)}{(totalnumberofoperations\ (n))}$$

- Stack of size n
  - Operations
    - Push – O(1)
    - Pop – O(1)
    - Multipop(S,K) – O(n)
  - Cost of Multipop
    - Min(s,k)



Aggregate Method

- Consider n operations
  - Push, pop, multipop
  - Multipop cost – O(n)
  - Total cost O(n^2)
    - O(n) multipop of O(n) each
- Using aggregate analysis
  - Cost of n operations is atmost O(n)
  - Each item can be popped atmost once it is pushed
  - Total number of pops that can be called on a non-empty stack is atmost n
  - n operations take O(n) time

Amortized cost per operation is O(n)/n = O(1)

Growable Array-based Stack

- When stack is full
  - during push instead of a StackFullException replace array with a larger one
- Method
  - **Algorithm** push(*o*)

```
if size() = $N$ then
    $A \leftarrow$ new array of size …
for $i \leftarrow 0$ to $t$ do
    $A[i] \leftarrow S[i]$
$S \leftarrow A$
$t \leftarrow t+1$
$S[t] \leftarrow o$
```

Increasing Array Size

- Size of the new array
  - Increasing strategy
    - Increase size by constant c
  - Doubling strategy
    - Double the size
- Comparison
  - Use Amortization Analysis
    - Analyze total time $t(n)$ needed to perform a series of push operations
    - Assume stack is empty and represented with array of size 1
    - Calculate amortized time of push = $t(n)/n$

Incremental Strategy Analysis

- We replace the array k times where $k = n/c$
- The total time T(n) of a series of n push operations is proportional to
  - $n + c + 2c + 3c + 4c + \ldots + kc =$
  - $n + c(1 + 2 + 3 + \ldots + k) =$
  - $n + ck(k + 1)/2$
- Since c is a constant, T(n) is $O(n + k^2)$, i.e.,
  - $O(n^2)$
- The amortized time of a push operation is O(n)

Doubling Strategy Analysis

- We replace the array $k = \log_2 n$ times
- The total time $T(n)$ of a series of n push operations is proportional to
  - $n + 1 + 2 + 4 + 8 + \ldots + 2^k =$
  - $n + 2^{k+1} - 1 = 2n - 1$
- $T(n)$ is $O(n)$
- The amortized time of a push operation is
  - $O(1)$