Computer Science 331

Stacks[1]

As part of the SAGES Teaching Scholar Program

Parthasarathi Das

Department of Computer Science
University of Calgary

Lecture #13

[1]Adapted from Dr. Michael Jacobson's lecture slides.

Outline

1. Learning Outcomes
2. Definition
3. Applications
   - Parenthesis Matching
4. Implementation
   - Array-Based Implementation
   - Linked List-Based Implementation
5. Additional Information

Learning Outcomes

By the end of today's session, you will be able to -

- understand what stacks are, their various types and some applications of stacks.
- implement stacks using arrays and linked lists
- apply this ADT suitably to solve problems

Definition of a Stack ADT

A *stack* is a collection of objects that can be accessed in "last-in, first-out"(LIFO) order: The only visible element is the (remaining) one that was most recently added.

It is easy to implement such a simple data structure extremely efficiently — and it can be used to solve several interesting problems.

Indeed, a *stack* is used to execute recursive programs — making this one of the more widely used data structures (even though you generally don't notice it!)

Stack ADT

Stack Interface:

```
public interface Stack<T> {
  public push(T x);
  public T peek();
  public T pop();
  public boolean isEmpty();
}
```

Stack Invariant:

- The object that is visible at the top of the stack is the object that has most recently been pushed onto it (and not yet removed).

A Stack Interface: Methods

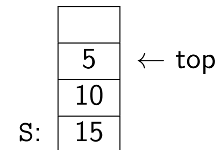
1. `void push(T obj)`:
   - *Precondition:*
     a) Interface invariant.
   - *Postcondition:*
     a) The input object has been pushed onto the stack (which is otherwise unchanged).
2. `T peek()` (called `top` in the textbook):
   - *Precondition:*
     a) Interface Invariant.
     b) The stack is not empty.
   - *Postcondition:*
     a) Value returned is the object on the top of the stack.
     b) The stack has not been changed.
   - *Exception:* An `EmptyStackException` is thrown if the stack is empty when this method is called.

A Stack Interface: Methods

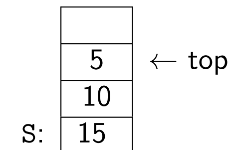
3. `T pop()`:
   - *Precondition:*
     a) Interface Invariant.
     b) The stack is not empty.
   - *Postcondition:*
     a) Value returned is the object on the top of the stack
     b) This top element has been removed from the stack
   - *Exception:* An `EmptyStackException` is thrown if the stack is empty when this method is called
4. `boolean isEmpty()`:
   - *Precondition:*
     a) Interface Invariant.
   - *Postcondition:*
     a) The stack has not been changed.
     b) Value returned is `true` if the stack is empty and `false` otherwise.

Example

Initial stack

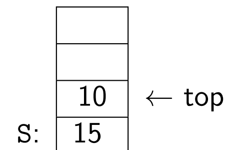


1) `S.peek()`



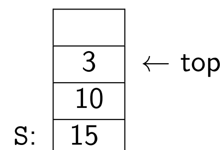
Output: 5

2) `S.pop()`



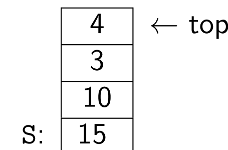
Output: 5

3) `S.push(3)`



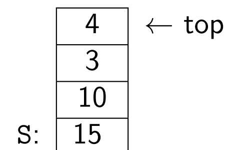
Output: no output

4) `S.push(4)`



Output: no output

5) `S.peek()`



Output: 4

Problem: Parenthesis Matching

Consider an expression, given as a string of text, that might include various kinds of brackets.

How can we confirm that the brackets in the expression are properly matched? Eg. $[(3 \times 4) + (2 - (3 + 6))]$

Solution : Using a stack (provable by induction on the length of the expression):

- Begin with an empty bounded stack (whose capacity is greater than or equal to the length of the given expression)
- Sweep over the expression, moving from left to right
- Push a left bracket onto the stack whenever one is found
- Try to pop a left bracket off the stack every time a right bracket is seen, checking that these two brackets have the same type
- Ignore non-bracket symbols

Solution Using a Stack (continued)

Then parentheses are matched if and only if:

- Stack is never empty when we want to pop a left bracket off it,and
- Compared left and right brackets always *do* have the same type, and
- The stack is empty after the last symbol in the expression has been processed.

Exercise: Trace execution of this algorithm on the preceding example.

Two possibilities

Dynamic array implementation:

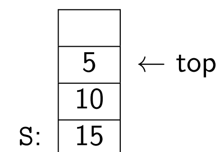
- Stack's contents stored in cells $0, \ldots, top - 1$; top element in $top - 1$.
- Can use a static array if size of stack is bounded.

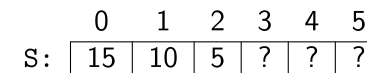
Linked implementation:

- Identify top of stack with the head of a singly-linked list
- Works well because stack operations only require access to the top of the stack, and linked list operations with the head are especially efficient.

Implementation Using an Array

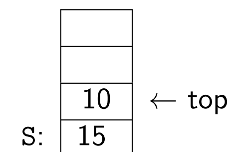
Initial Stack

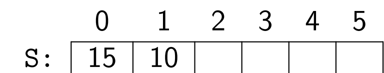




top = 2

Effect of `S.pop()`

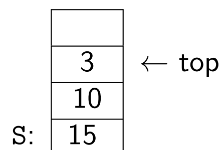


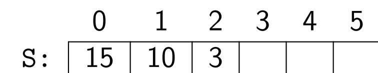


top = 1

Implementation Using an Array

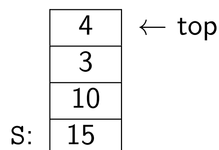
Effect of `S.push(3)`

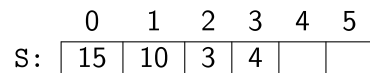




top = 2

Effect of `S.push(4)`





top = 3

Implementation of Stack Operations

```
public class ArrayStack<T> implements Stack<T> {
  private T[] stack;
  private int top;

  public ArrayStack(){
   top = -1;
   stack = (T[]) new Object[6]; }

  public boolean isEmpty(){
   return (top == -1); }

  public int size(){
   return top+1; }

  public void push(T x){
   ++top;
   stack[top] = x; }
```


Implementation of Stack Operations

```
  public T peek() {
    if (isEmpty())
     throw new EmptyStackException();
    return stack[top];}

  public T pop() {
    if (isEmpty())
     throw new EmptyStackException();
    T e = stack[top];
    stack[top] = null;
    --top;
    return e; }
}
```


Cost of Operations

All operations cost $\Theta(1)$ (constant time, independent of stack size).
Problem: What should we do if the stack size exceeds the array size?

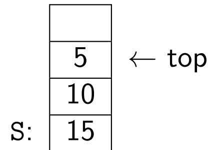
- Modify *push()* to reallocate a larger stack (or use a dynamic array)

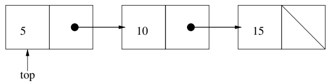
```
public void push(T x) {
  ++top;
  if (top == stack.length) {
    T [] stackNew = (T[]) new Object[2*stack.length];
    System.arraycopy(stackNew,0,stack,0,stack.length);
    stack = stackNew;
  }
  stack[top] = x;
}
```

Revised cost: $\Theta(n)$ in the worst case, $\Theta(1)$ amortized cost

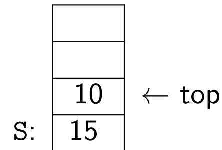
Implementation Using a Linked List

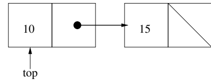
Initial Stack





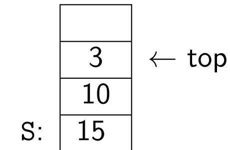
Effect of `S.pop()`

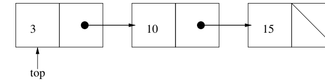




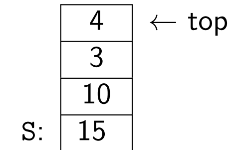
Implementation Using a Linked List

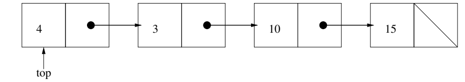
Effect of `S.push(3)`





Effect of `S.push(4)`





Implementation of Stack Operations

```
public class LinkedListStack<T> implements Stack<T> {
  private class StackNode<T> {
    private T value;
    private StackNode<T> next;

    private StackNode(T x, StackNode<T> n)
      { value = x; next = n; }
  }

  private StackNode<T> top;
  private int size;

  public LinkedListStack()
    { size = 0; top = (StackNode<T>) null; }
  public boolean isEmpty()  { return (size == 0); }
  public int size()  { return size; }
```


Implementation of Stack Operations (cont.)

```
public void push(T x)  {
  ++size; top = new StackNode<T>(x,top);
}

public T peek() {
  if (isEmpty())  throw new EmptyStackException();
  return top.value;
}

public T pop() {
  if (isEmpty())  throw new EmptyStackException();
  T x = top.value; top = top.next; --size; return x;
}
```

Cost of stack operations: $\Theta(1)$ (independent of stack size)

Variation: Bounded Stacks

Size-Bounded Stacks — Similar to stacks (as defined above) with the following exception:

- Stacks are created to have a maximum `capacity` (possibly user-defined — so that two constructors are needed)
- If the `capacity` would be exceeded when a new element is added to the top of the stack then `push` throws a `StackOverflowException` and leaves the stack unchanged
- A *static array* whose `length` is the stack's `capacity` can be used to implement a size-bounded stack, extremely simply and efficiently

Most "hardware" and physical stacks are bounded stacks.

Stacks in Java and the Textbook

Implementation in Java 8:

- Java 8 includes a `Stack` class as an extension of the `Vector` class (a dynamic array).
  Unfortunately, this implementation is somewhat problematic (Stack inheirit's Vector's methods, too!)

Introduction to Algorithms

- by Cormen, Lieserson, Rivest, and Stein
- Section 10.1

Data Structures: Abstraction and Design Using Java

- by Elliot B. Koffman and Paul A. T. Wolfgang
- Chapter 3