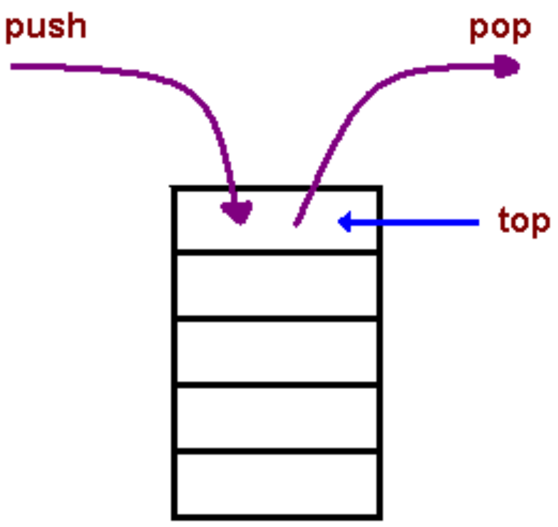
Stacks

Table of contents

Introduction

A Stack is a restricted ordered sequence in which we can only add to and remove from one end — the **top** of the stack. Imagine stacking a set of books on top of each other — you can **push** a new book on **top** of the stack, and you can **pop** the book that is currently on the top of the stack. You are not, strictly speaking, allowed to add to the middle of the stack, nor are you allowed to remove a book from the middle of the stack. The only book that can be taken out of the stack is the **most recently added** one; a stack is thus a "last in, first out" (LIFO) data structure.

We use stacks everyday — from finding our way out of a maze, to evaluating postfix expressions, "undoing" the edits in a word-processor, and to implementing recursion in programming language runtime environments.



Three basic stack operations are:

- **push(obj)**: adds `obj` to the top of the stack ("overflow" error if the stack has fixed capacity, and is full)
- **pop**: removes and returns the item from the top of the stack ("underflow" error if the stack is empty)
- **peek**: returns the item that is on the top of the stack, but does not remove it ("underflow" error if the stack is empty)

The following shows the various operations on a stack.

```
Java statement                   resulting stack
------------------------------------------------
Stack s = new Stack();
                                  -----  (empty stack)



S.push(7);
                                  | 7 |   <-- top
                                  -----

S.push(2);
                                  | 2 |   <-- top
                                  | 7 |
                                  -----

S.push(73);
                                  |73 |   <-- top
                                  | 2 |
                                  | 7 |
                                  -----

S.pop();
                                  | 2 |   <-- top
                                  | 7 |
                                  -----
S.pop();
                                  | 7 |   <-- top
                                  -----
S.pop();
                                  -----   (empty stack)

S.pop();
                                  ERROR "stack underflow"
```

Back to Table of contents

Stack applications

- **Reverse**: The simplest application of a stack is to reverse a word. You push a given word to stack – letter by letter – and then pop letters from the stack. Here's the trivial algorithm to print a word in reverse:

```
begin with an empty stack and an input stream.
while there is more characters to read, do:
   read the next input character;
   push it onto the stack;
end while;
while the stack is not empty, do:
   c = pop the stack;
   print c;
end while;
```

- **Undo**: Another application is an "undo" mechanism in text editors; this operation is accomplished by keeping all text changes in a stack. Popping the stack is equivalent to "undoing" the last action. A very similar one is going back pages in a browser using the *back* button; this is accomplished by keeping the pages visited in a stack. Clicking on the *back* button is equivalent to going back to the most-recently visited page prior to the current one.
- **Expression evaluation**: When an arithmetic expression is presented in the *postfix* form, you can use a stack to evaluate it to get the final value. For example: the expression `3 + 5 * 9` (which is in the usual *infix* form) can be written as `3 5 9 * +` in the *postfix*. More interestingly, postfix form removes all parentheses and thus all implicit precedence rules; for example, the infix expression `((3 + 2) * 4) / (5 - 1)` is written as the postfix `3 2 + 4 * 5 1 - /`. You can now design a calculator for expressions in postfix form using a stack.

  The algorithm may be written as the following:

```
begin with an empty stack and an input stream (for the expression).
while there is more input to read, do:
   read the next input symbol;
   if it's an operand,
      then push it onto the stack;
   if it's an operator
      then pop two operands from the stack;
           perform the operation on the operands;
           push the result;
end while;
// the answer of the expression is waiting for you in the stack:
pop the answer;
```

  Let's apply this algorithm to to evaluate the postfix expression `3 2 + 4 * 5 1 - /` using a stack.

```
 Stack    Expression

|   |
 ---        3 2 + 4 * 5 1 - /
(empty)

| 3 |       2 + 4 * 5 1 - /
 ---


| 2 |
| 3 |       + 4 * 5 1 - /
 ---

| 5 |       4 * 5 1 - /
 ---


| 4 |
| 5 |       * 5 1 - /
 ---

| 20|       5 1 - /
 ---


| 5 |
| 20|       1 - /
 ---
```

```
| 1 |
| 5 |
| 20|       - /
 ---

| 4 |
| 20|       /
 ---

| 5 |      (finished. the result is on top of the stack)
 ---
```

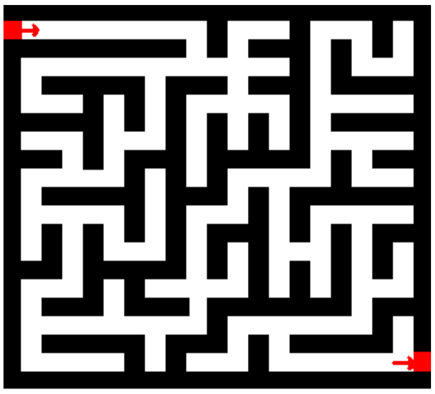
- **Parentheses matching**: We often have a expressions involving "()[]{}" that requires that the different types parentheses are *balanced*. For example, the following are properly balanced:
    - `(a (b + c) + d)`
    - `[ (a b) (c d) ]`
    - `( [a {x y} b] )`

  But the following are not:
    - `(a (b + c) + d`
    - `[ (a b] (c d) )`
    - `( [a {x y) b] )`

  The algorithm may be written as the following:

```
begin with an empty stack and an input stream (for the expression).
while there is more input to read, do:
   read the next input character;
   if it's an opening parenthesis/brace/bracket ("(" or "{" or "[")
      then push it onto the stack;
   if it's a closing parenthesis/brace/bracket (")" or "}" or "]")
      then pop the opening symbol from stack;
           compare the closing with opening symbol;
           if it matches
              then continue with next input character;
           if it does not match
              then return false;
end while;
// all matched, so return true
return true;
```

- **Backtracking**: This is a process when you need to access the most recent data element in a series of elements. Think of a labyrinth or maze - how do you find a way from an entrance to an exit?



Once you reach a dead end, you must backtrack. But backtrack to where? to the previous choice point. Therefore, at each choice point you store on a stack all possible choices. Then backtracking simply means popping a next choice from the stack.

- **Language processing**:
  - space for parameters and local variables is created internally using a stack (*activation records*).
  - compiler's syntax check for matching braces is implemented by using stack.
  - support for recursion

Back to Table of contents

Stack implementation

In the standard library of classes, the data type stack is an *adapter* class, meaning that a stack is built on top of other data structures. The underlying structure for a stack could be an array, a vector, an ArrayList, a linked list, or any other sequence (`Collection` class in JDK). Regardless of the type of the underlying data structure, a `Stack` must implement the same functionality. This is achieved by providing an interface:

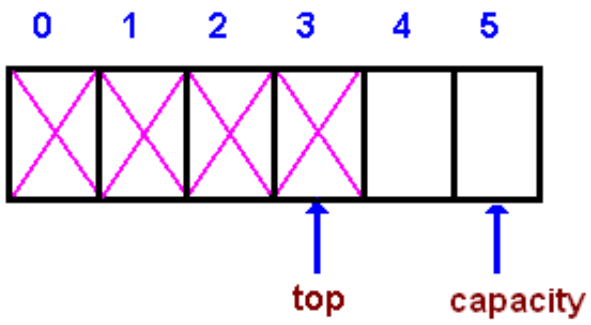
```
public interface Stack {
    // The number of items on the stack
    int size();
    // Returns true if the stack is empty
    boolean isEmpty();
    // Pushes the new item on the stack, throwing the
    // StackOverflowException if the stack is at maximum capacity. It
    // does not throw an exception for an "unbounded" stack, which
    // dynamically adjusts capacity as needed.
    void push(Object o) throws StackOverflowException;
    // Pops the item on the top of the stack, throwing the
    // StackUnderflowException if the stack is empty.
    Object pop() throws StackUnderflowException;
    // Peeks at the item on the top of the stack, throwing
    // StackUnderflowException if the stack is empty.
    Object peek() throws StackUnderflowException;
    // Returns a textual representation of items on the stack, in the
    // format "[ x y z ]", where x and z are items on top and bottom
    // of the stack respectively.
    String toString();
    // Returns an array with items on the stack, with the item on top
    // of the stack in the first slot, and bottom in the last slot.
    Object[] toArray();
    // Searches for the given item on the stack, returning the
    // offset from top of the stack if item is found, or -1 otherwise.
    int search(Object o);
}
```

One requirement of a `Stack` implementation is that the `push` and `pop` operations run in *constant time*, that is, the time taken for stack operation is independent of how big or small the stack is.

Back to Table of contents

Array based implementation

In an array-based implementation we add the new item being pushed at the end of the array, and consequently pop the item from the end of the array as well. This way, there is no need to shift the array elements. The top of the stack is always the last used slot in the array, so we can use `size-1` to refer to the top of the stack element instead of having to keep a separate field. The *top* of the stack is not defined for an empty stack.



In a *bounded* or fixed-size stack abstraction, the capacity stays unchanged, therefore when *top* reaches *capacity*, the stack object throws an exception.

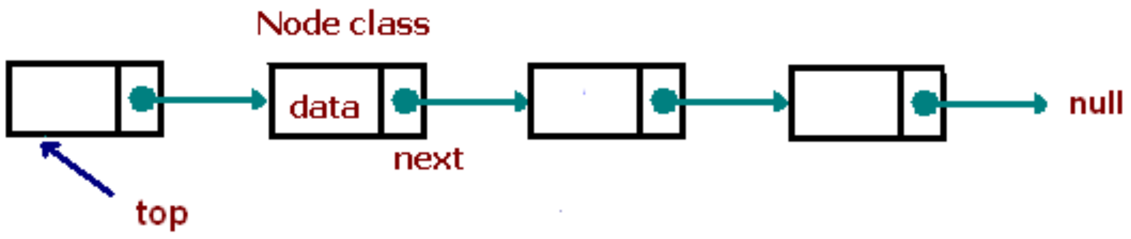
In an *unbounded* or dynamic stack abstraction when *top* reaches *capacity*, we double up the stack size. The following shows a partial array-based implementation of an *unbounded* stack.

```
public class ArrayStack implements Stack {
    private Object[] data;      // The array container
    private int      size;      // The number of items in the stack

    // Default initial capacity, which may then dynamically change
    private static final int DEF_INIT_CAPACITY = 100;

    public ArrayStack() {
        data = new Object[DEF_INIT_CAPACITY];
        size = 0;
    }
}
```

Back to Table of contents

Linked list based implementation

Linked List-based implementation provides the best (from the efficiency point of view) dynamic stack implementation.



The following shows a partial head-referenced singly-linked based implementation of an *unbounded* stack. In an singly-linked list-based implementation we add the new item being pushed at the beginning of the array (why?), and consequently pop the item from the beginning of the list as well.

```
public class ListStack implements Stack {
    private Node head;          // Reference to the top of the stack
    private int  size;          // The number of items in the stack

    public ListStack() {
        head = null;
        size = 0;
    }
}
```

Back to Table of contents