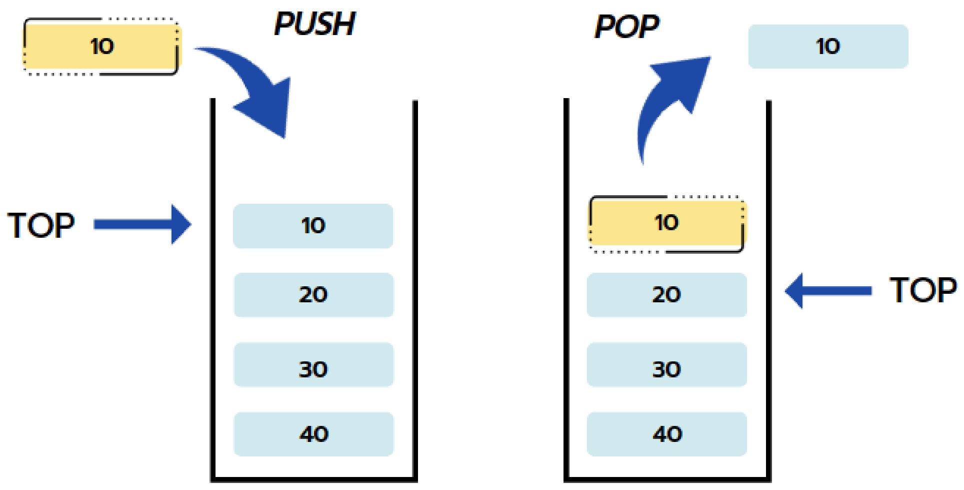
Chapter 6 - Stack

Introduction

There are some problems in which we want to store information, and we want to access the more recently stored information first. A data structure that allows us to access information in this manner is called a stack. It is so named because it resembles a stack of dishes. When you want a dish, you take it off the stack of dishes. When you clean a dish, you put it back on top of the stack of dishes. The same holds with a stack of data. When you store a new data item, you store it on top of the stack. When you want to retrieve a data item, you retrieve the top item from the stack. Note that this data item will be the one most recently stored on the stack.

Thus, 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.



STACK DATA STRUCTURE

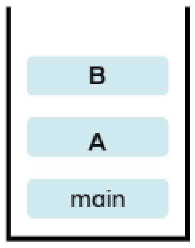
Three basic stack operations are:

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)

Stack Applications

A couple of examples where you want to access information in this manner in the real world are as follows:

Call Stacks: Whenever your C program calls a function, the C compiler creates a so-called *stack record* that contains important information about that function, such as storage for its parameters, its return value, and its local variables. The C compiler pushes this stack record onto something called a *call stack*. The call stack is organized so that the stack record of the most recently called function is on top of the stack, and the stack record of the least recently called function, which in the case of C is main, is at the bottom of the stack. For example, if the main calls A, which in turn calls B, then the call stack would be:



By convention stacks are shown growing upwards. While a function is executing, its stack record will be on top of the stack, and hence its information will be readily available. In the above case, B is currently executing and so we can readily access its parameters and local variables. If B calls a new function C, then a stack record for C is created and pushed onto the call stack. C now has access to its local variables and parameters. When C returns, we want B to resume executing, and thus we want access to B's stack record. By simply popping C's stack record off the call stack, we again have access to B's stack record. Hence a stack is the perfect data structure for implementing a call stack, because we always want access to the most recently called function, and when it returns, we want access to the function that called it.

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: In many editors, it is common to want to know which left parenthesis (or brace or bracket, etc) will be matched when we type a right parenthesis. Clearly we want the most recent left parenthesis. Therefore, the editor can keep track of parentheses by pushing them onto a stack, and then popping them off as each right parenthesis is encountered. We often have expressions involving "()[]{}" that require that the different types of 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

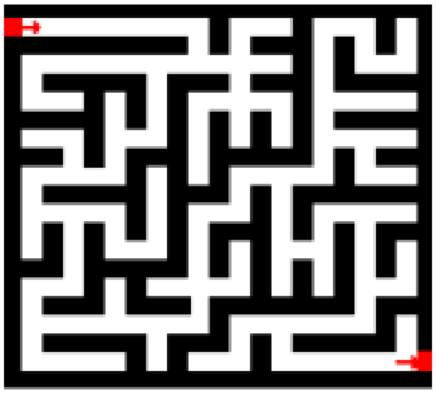
[(a b] (c d))

([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.

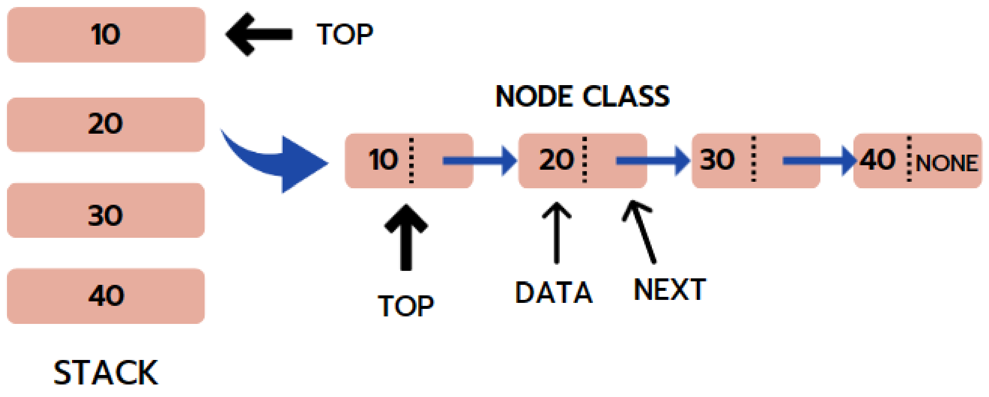
Language Processing:
- space for parameters and local variables is created internally using a stack (*activation records*).
- The compiler's syntax check for matching braces is implemented by using stack.
- support for recursion

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. Regardless of the type of the underlying data structure, a ***Stack*** must implement the same functionality. 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.

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 a 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.

First, we create a class node. This is our Linked List node class which will have data in it and a node pointer to store the address of the next node element.

```python
class Node:
  def __init__(self,element,next):
    self.element = element
    self.next = next
```

Then, we define the Stack class inside which we have the push(), pop() and peek() methods.

Push Operation

Adding a new node in the Stack is termed a push operation. Push operation on stack implementation using linked-list involves several steps:

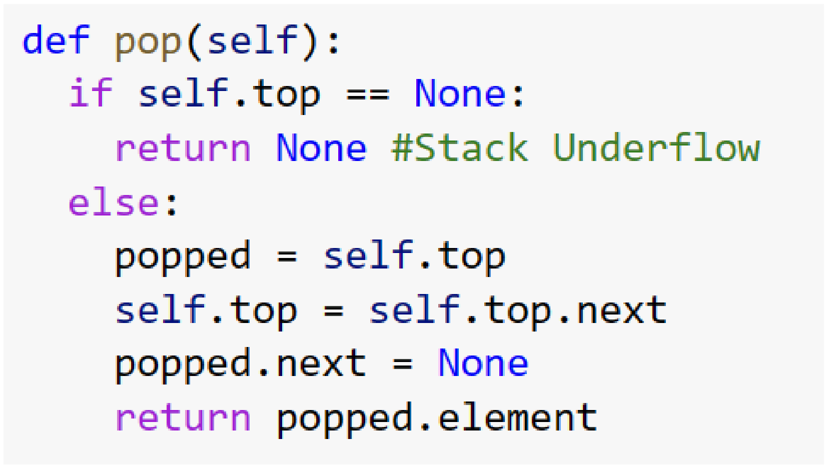
- Create a node first and allocate memory to it.
- If the stack is empty, then the node is pushed as the first node of the linked list and defined as top. This operation assigns a value to the data part of the node and gives None to the address part of the node.
- If some nodes are already there in the linked list, then we have to add a new node at the beginning of the linked list so that we do not violate Stack's property. For this, assign the previous top to the next address field of the new node and the new node will be the starting node or the current top of the list.

```python
class Stack:
  def __init__(self):
    self.top = None

  def push(self,elem):
    if self.top == None: #Stack is empty
      self.top = Node(elem,None)
    else:
      newNode = Node(elem,None)
      newNode.next = self.top
      self.top = newNode
```

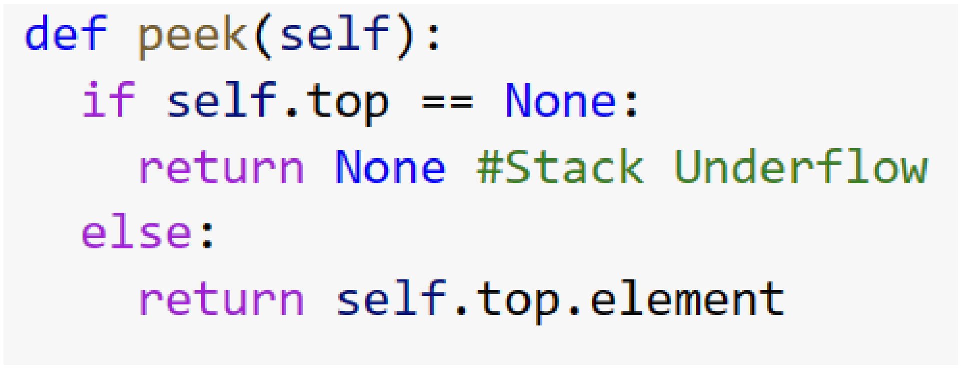
Pop Operation:
Deleting a node from the Stack is known as a pop operation. Pop operation involves the following steps:

- In Stack, the node is removed from the top of the linked list. Therefore, we must delete the value stored in the head/top pointer, and the node must get free. The following link node then becomes the head/top node.
- An underflow condition will occur when we try to pop a node when the Stack is empty.



```
def pop(self):
  if self.top == None:
    return None #Stack Underflow
  else:
    popped = self.top
    self.top = self.top.next
    popped.next = None
    return popped.element
```

Peek Operation:
In the peek operation, we simply return the data stored in the head/top of the Linked List based Stack. An underflow condition will occur when we try to peek at a value when the Stack is empty.



```
def peek(self):
  if self.top == None:
    return None #Stack Underflow
  else:
    return self.top.element
```