**STACK ABSTRACT DATA TYPE**

There are four methods of stack:

1. push
2. pop
3. peek
4. empty

Stack’s storage policy is LIFO.

Specifications of the Stack Abstract Data Type

Only the top element of a stack is visible; therefore the number of operations performed by a stack are few

We need the ability to:

* test for an empty space (empty)
* inspect the top element (peek)
* retrieve the top element (pop)
* put a new element on the stack (push)

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application

Description automatically generated

**Stack Applications**

Balancad Parantheses

Balanced parantheses means each parantheses is closed properly

When analyzing arithmetic expressions, it is important to determine whether an expression is balanced with respect to parantheses

( a + b \* ( c / ( d - e ) ) ) + ( d / e )

The problem is further complicated if braces or brackets are used in conjunction with parantheses

The solution is to use stacks

When we see a opening parantheses, we push that into a stack.

When we see a closing parantheses, we pop the last from the stack.

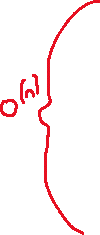
When we pop, we get last entered parantheses. Then we have to check whether opening and closing paratheses match.

Graphical user interface, application

Description automatically generated

Text

Description automatically generated



Character can be operator and operand but we just skip them.

🡪 We understood problem

🡪 We designed an algorithm

🡪 We checked if algorithm is correct

* We are sure that while loop proceeds each element correctly and we produce correct result

🡪 Analyze algorithm ( O(n) )

* First 3 operations are constant time.
* Inside the loop
  + 5, 6, 7, 8, 9, 10, 11, 12 🡪 constant
* Loop repeated size of the string (n) times 🡪 O(n)
  + worst case 1, best case n times
* 13 🡪 constant

When we talk about the space complexity, we talk about these 3:

1. Input space complexity
   1. Input is string and its complexity is θ(n)
2. Processing time space complexity
   1. During processing the string, how much space this algorithm uses?

We keep just 1 index value (at line 3), 1 boolean balanced value (at line 2), 1 character (at line 5), and 1 stack (at line 1). Memory of stack grows during processes. Maximum size of the stack (all of the string is open parantheses) can grow as big as size of the string θ(n), minimum size of the stack can grow just 1 (1 open 1 closed sequentially).

* 1. Sbest(n) = θ(1)

Sworst(n) = θ(n)

S(n) = O(n)

1. Output space complexity
   1. Output is boolean and output complexity is θ(1)

Sometimes we don’t care input and output space complexity since they are related to callee. They don’t change with the algorithm design.

Text

Description automatically generated

Text

Description automatically generated

🡪 Testing

* Provide a variety of input expressions displaying the result true or false
  + Try several levels of nested parantheses
  + Try nested parantheses where corresponding parantheses are not of the same type
  + Try unbalanced parantheses
  + No parantheses at all
* PITFALL : attempting to pop an empty stack will throw an EmptyStackException. You can guard against this by either testing for an empty stack or catching the exception

Text

Description automatically generated

**Implementing a Stack as an Extension of Vector**

The Java API includes a Stack class as part of the package java.util:

public class Stack<E> extends Vector<E>

The class is called Stack but it is not actually a stack since you can use other methods that shouldn’t be provided by a stack. So this is not a good way to implement a stack actually. You shouldn’t use stack provided by Java. You can add anywhere in Stack, that’s not what we want.

Diagram

Description automatically generated

The Vector class implements a growable array of objects

Elements of a Vector can be accessed using an integer index and the size can grow or shrink as needed to accommodate the insertion and removal of elements.

We can use Vector’s add method to implement push:

A picture containing text

Description automatically generated

pop can be coded as:

Graphical user interface, text, application

Description automatically generated

push 🡪 if there is enough space constant time, we can say it is amortized constant time

pop 🡪 constant time

Because a Stack is a Vector, all of Vector operations can be applied to a Stack (such as searches and access by index)

But, since only the top element of a stack should be accessible, this violates the principle of information hiding

**Implementing a Stack with a List Component**

As an alternative to a stack as an extension of Vector, we can write a class, ListStack, that has a List component (in the example below, theData)

We can use either the ArrayList, Vector, or the LinkedList classes, as all implement the List interface. The push method, for example, can be coded as:

public E push(E obj) {

theData.add(obj);

return obj;

}

So we restricted the access to add method etc.. theData is List data field for our stack.

Now push is again a constant time.

* If theData is ArrayList, it is amortized constant time
* If theData is LinkedList, it is constant time (LinkedList’s implemented as double linked list in Java)

pop can be done in constant time too.

A class which adapts methods of another class by giving different names to essentially the same methods (push instead of add) is called an adapter class

Writing methods in this way is called method delegation

Text, letter

Description automatically generated

Text

Description automatically generated

e.g., pop operation is delegated to remove operation of the ArrayList.

Running time is constant again.

Implementing a Stack Using an Array

Text

Description automatically generated

Diagram

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Implementing a Stack as a Linked Data Structure

Diagram

Description automatically generated

Top is the head.

Again this needs constant time.

Comparison of Stack Implementations

* Extending a Vector (as is done by Java) is a poor choice for stack implementation, since all Vector methods are accessible
* The easiest implementation uses a List component (ArrayList is simplest) for storing data
  + An underlying array requires reallocation of space when the array becomes full, and
  + an underlying linked data structure requires allocating storage for links
  + As all insertions and deletions occur at one end, they are constant time, O(1), regardless of the type of implementation used

peek:

* for ArrayList 🡪 θ(1)
* for LinkedList 🡪 θ(1)

ArrayList’s or LinkedList’s peek methods, which one is faster? They both are constant time, how come one of them become faster? 100 is constant, 5 is constant. So one of them may become faster, since they are asymptotic.

In LinkedList, you have to perform some reference assignments and garbage collecter gets rid of that memory.

In ArrayList, you access by using index easily.

So ArrayList is faster since there is no garbage collection thing going on.

push:

If you don’t need to reallocate, ArrayList is faster for same reason. In LinkedList running time is a little bit slower but always same. In ArrayList running time is kinda fast-fast-fast-slow... because of reallocation. You should choose depending on your requirements.

**Additional Stack Applications**

Postfix and infix notation

* Expressions normally are written in infix form, but it is easier to evaluate an expression in postfix form since there is no need to group sub-expressions in parantheses or worry about operator precedence

Table

Description automatically generated

Read left to right. When you see an operator, apply that operator to last 2 numbers.

Evaluating Postfix Expressions

Write a class that evaluates a postfix expression

Use the space character as a delimeter between tokens

Graphical user interface

Description automatically generated with medium confidence

Text

Description automatically generated

After all the operations if there are more than 1 operand in the stack, there should be a mistake. So if there is no mistake, then there is 1 operand in the stack and that is the result.

1 🡪 constant time

while loop 🡪 θ(n)

There is no early out in the while loop

3, 4, 5, 6, 7, 8, 9, 10, 11 🡪 constant time

9 may not be constant. For example if we have 2^5, this may not be constant. For the time being, we assume that there are only basic operations (+, -, \*, /).

How can you evaluate a^n?

You can calculate this with a loop from 1 to n by incrementing one by one and multiplying a with the result. So running time of this is θ(n).

You can also calculate it with logn time.

Testing: write a driver which

* creates a PostfixEvaluater object
* reads one or more expressions and report the result
* catches PostfixEvaluator.SyntaxErrorException
* exercises each path by using each operator
* exercises each path through the method by trying different orderings and multiple occurrences of operators
* tests for syntax errors:
  + an operator without any operands
  + a single operand
  + an extra operand
  + an extra operator
  + a variable name
  + the empty string

Converting from Infix to Postfix

Assume:

* expressions consist of only spaces, operands, and operators
* space is a delimeter character
* all operands that are identifiers begin with a letter or underscore
* all operands that are numbers begin with a digit

Table

Description automatically generated

A picture containing text

Description automatically generated

Table

Description automatically generated

For the \*, if - has higher precedence we would do same thing to - as we did to /.

If top of the stack has same or higher precedence operator than the current operator in the string, we add that to postfix. Since we are going left to right, left operator has higher precedence. That’s why if operators’ precedence is same, we add operator in the stack to the postfix.

Graphical user interface, text, table

Description automatically generated with medium confidence

Text

Description automatically generated

Text, letter

Description automatically generated

All algorithm is linear time.

Testing:

* Use enough test expressions to satisfy yourself that the conversions are correct for properly formed input expressions
* Use a driver to catch InfixToPostfix.SyntaxErrorException

Converting Expressions with Parantheses

The ability to convert expressions with parantheses is an important (and necessary) addition

Modify processOperator to push each opening paranthesis onto the stack as soon as it is scanned

When a closing paranthesis is encountered, pop off operators until the opening paranthesis is encountered