Stacks and Queues

Sujit Chakrabarti

1 Abstract Data Types and Data Structures

In one of our earlier modules we had discussed about two data structures: *array lists* and *linked lists*. We had how both are implementations of the same abstract data type *list*. We had talked about the various capabilities and properties of lists, and how they could be used in solving practical software development problems (recall *Institute Management System*). We had discussed how, despite being functionally nearly identical to each other, array lists and linked lists are really quite different from each other. In particular, when you take into account, their performance aspects, we observe that each of them fairs better than the other in certain scenarios, and worse in certain others. For example, getting a value by its index is typically much faster in an array list. However, adding/removing an element at an arbitrary location may be faster in a linked list.

Through the *Algorithms* course, now we have gathered enough conceptual ammunition so that we can launch into a more detailed discussion on more data structures and understand their capabilities and performance metrics.

In the upcoming several videos we will get introduced to a number of other data structures.

2 Data Structures

Algorithms are computational procedures which deal with data. They create them, modify them and refer to them time to time. In order to deliver good results, your algorithms need the program data to be arranged in a way suitable for the purpose. A well arranged data will make it easy to implement algorithms – and software systems in general – that not only are fast and efficient, but are also flexible while being robust and secure. Hence, how the data is arranged within a program is a crucial question that must be answered at a very early stage of design of a program.

The subject of *data structures* is the study of some of these 'arrangements' of data which have been found useful by software developers in a variety of scenarios. While studying a data structure, our aim should be to understand:

1. What are their capabilities?
2. In what scenarios can they be used and how?
3. What are various implementations and how do they compare in terms of performance (space/time)?

3 Abstract Data Types

I have been using two terms – *abstract data types* and *data structures* – apparently synonymously. While they are indeed somewhat similar, they aren't exactly the same. And while in general it's OK to use them a bit interchangeably, it's important that we know the difference.

Abstract data type is the definition of the interface (i.e. the properties and capabilities) of a data arrangement. Data structure is a specific implementation. For example, list is an ADT (where the elements are linearly sequence, each associated with an index or position represented by a non-negative integer); linked list is an implementation. In other words, there may be multiple data structures which implement the same ADT. Having said that, we repeat that we use these two terms as near synonyms. We hope that the context will make the distinction clear. If not, we will clarify the difference.

With that, let's talk of data structures, beginning with *stacks*.

4 Stacks

Consider a stack of clothes – or plates – or some such things. If all clothes in that stack are identical, and you were to pick one from it, from which portion of it would you pick it? From the top? From the bottom? Or from somewhere in between? Of course, any sane person would pick it up from the top. Trying to pull out a piece of clothing from anywhere else in the stack would be possible, but would lead to difficulties, and untoward results. In the same manner, if you were to add a clothe to this stack, where would you prefer to add it? Again, at the the top, of course. And if that's the way things are added to and removed from this stack, what can be said about the relation between the order of insertion of things into and their removal from the stack?

It turns out that the relation is quite simple: what goes in last is the first to come out. This property of the above structure is called *last in first out* (or LIFO) order. To give another example slightly closer to computation, think about a busy day at office which full of interruptions. Each task takes a fairly long time to complete. In between there are interruptions. And if there are interruptions, they must immediately be handled. The task you were doing in the beginning, say T_1, has to be paused to attend to the interrupt say T_2, and should be resumed when T_2 is completely handled. To make things more interesting, the task T_3 that interrupts the first task may in turn get interrupted by some other task T_3, which further could get interrupted by something else.

To keep track of your task, you would probably like to maintain a to-do list. But this to-do list is a bit different from the usual to-do lists (where completed tasks get added at the bottom, and typically, the one at the top is the first get checked off, i.e. completed). Here, the task at the bottom of this list is the one you are currently at. It was also the last one to be added, and it will be the first one to be checked off. Once a task gets done, it gets checked off, and the task immediately above it in the list (which it had interrupted) now becomes the current task. Again, the LIFO principle is followed in the scenario.

In real life, and in computing, there happen umpteen instances wherein things have to inserted into *something* and removed from it in a LIFO order. These somethings – at least in computer science – are called stacks. A stack is therefore anything:

1. into which you insert (*push*) and remove (*pop*) things from one end of it.

2. that follows LIFO rule.

4.1 Applications

1. Think about a Internet browser and how it implements the functionality of the famous 'Back' button.

2. How text editors implement the *undo/redo* feature.

3. How a program runtime manages function calls.

4. Compilers – programs which take your program and turns it into a program in some other language, like machine language or byte code – do a very important thing called *parsing*. Parsing, in simple language,

means reading. A compiler really uses parsing to figure out if a program it's compiling is really well-formed or not. For a program to successfully compile, it's necessary but not sufficient that it should be well-formed. If a program is not well-formed, i.e. if it's ill-formed, the compiler will typically detect that, and will flag a syntax error. Parsing is a mind-blowingly complex process. But, worry not. We don't go there. What's interesting to us is that it uses stacks, among other things, a lot for parsing, and for many other things that it does.

4.2 Questions

1. Draw the snapshots of the program stack when computing recursive function `fibonacci(n)` that implements the `n`th Fibonacci number given by the following function:

```
fibonacci(n) =  1 if n = 1 or 0
                fibonacci(n - 1) + fibonacci(n - 2) otherwise
```

5 Detailed Application – Balanced Parentheses

5.1 Single Type Parentheses

Let's pick out a tiny portion of the parsing problem to understand where stacks come into picture in compiling. Let's just remove everything else from our language, leaving out only parentheses. One of the things that must happen for a program to be well-formed is that all its parentheses should match, i.e. if there's an open parenthesis anywhere in the program, there must also be a corresponding closing parenthesis. Obviously, the closing parenthesis must follow, and not precede, its corresponding open paren. For example, `"(())"` and `"()()"` are well-formed strings, but `")("` is not. So, having an equal number of open and close parens is necessary but not sufficient for a string to well-formed.

How do you find out if a string of parentheses is well-formed? For this simple language, an equally simple approach works.

(**code:** `code/Paren1.java`)

(TODO: Example Run.)

The same could be implemented using a stack.

(**code:** `code/Paren2.java`)

Although, this example doesn't necessarily need a stack to be used, it helps us get introduced to what operations can typically be done with a

Algorithm 1 Matching Parentheses using *count*

1. Initialise variable *count* to 0.
2. Scan the string from left to right.
3. As you go symbol by symbol, whenever you meet an open paren, increase *count* by 1.
4. If you meet a close paren, decrease *count* by 1. However, if *count* is already 0, it means error; so return false.
5. If you reach the end of string, and *count* = 0, it means that the string is well-formed; so return true. Otherwise, return false.

Algorithm 2 Matching Parentheses using stack

1. Initialise stack S to be empty.
2. Scan the string from left to right.
3. As you go symbol by symbol, whenever you meet an open paren, push `'('` into S.
4. If you meet a close paren, pop a `'('` from S. However, if S is already empty, it means error; so return false.
5. If you reach the end of string, and S, it means that the string is well-formed; so return true. Otherwise, return false.

stack. Here, we do a *push*, when we see an open paren; and we do a *pop* when we see a close paren. Moreover, we check if the stack is empty or not at two places: i.e. everytime we see a close paren, and at the very end when the entire string has been seen.

(TODO: Example Run.)

5.2 Multiple Type Parentheses

Algorithm 1 works well when there's only one variety of parentheses. Let's enrich our language to have two types of brackets: parentheses (`'('`) and braces (`'{'`). Can we modify the approach to deal with this variant? Having one count will clearly not work. This algorithm will accept strings "(}" as well formed as it's not even equipped to deal with the two types of brackets separately. An obviously extension, therefore, would be to have two counts: *count*1 for parentheses and *count*2 for braces. It will correctly accept all well-formed strings. It will correctly reject some of the ill-formed strings. Unfortunately, it will fail to detect strings where both the parentheses and braces are individually matched, but their relative positions are messed up, e.g. `"({)}"`. We have to conclude that the count based approach is too weak to work when there are multiple types of brackets to deal with.

This variant of the balanced parenthesis problem does indeed bring out the power of a stack based approach. Algorithm 3 modifies algorithm 2 to deal with the multiple bracket problem.

(**code:** `code/Paren3.java`)

(TODO: Example Run.)

5.3 Double-Ended Queue

5.4 Detailed Application – Palindrome

(**code:** `code/Palindrome.java`)

5.5 Programming Problems

1. In the `MyStack` class, replace the underlying `list.util.LinkedList` class with your own `LinkedList`.

2. Implement a function `reverseArray` that reverses an array. For example, `reverseArray({1,2,3})` should give `{3,2,1}`.

3. Improve the stack based parenthesis matcher to handle HTML tags (some of them, e.g. `body`, `title` etc.).

Algorithm 3 Matching Parentheses using stack

1. Initialise stack S to be empty.
2. Scan the string from left to right.
3. As you go symbol by symbol, whenever you meet an open paren, push `'('` into S. If see an open brace push `'{'` into S.
4. If you meet a close paren, pop a symbol from S. If it's not an open paren (e.g. if it's an open brace), return false. Also, if S is already empty, it means error; so return false.
5. If you meet a close brace, pop a symbol from S. If it's not an open brace (e.g. if it's an open paren), return false. Also, if S is already empty, it means error; so return false.
6. If you reach the end of string, and S, it means that the string is well-formed; so return true. Otherwise, return false.

4. Implement a Deque using:
   (a) Circular array
   (b) Doubly-linked list

6 Queues