Q and A for Data Structure (CS206)

Sung-Eui Yoon
Dept. of Computer Science
KAIST
sungeui@kaist.edu
http://sgvr.kaist.ac.kr

May 1, 2019

Abstract

I have asked my class students to write down their questions related to the course materials. Some of them are asked repeatedly and I felt that providing my version of answers to those questions should be useful for students to get higher understanding on data structure. Note that these answers may not be correct and are not even designed to be correct answers. I wish that these answers serve as an initial starting point to those questions and help students to make more thought-provoking questions and embark deeper intellectual explorations on data structure and its related topics.

1 Introduction to Course and JAVA

Why are we using JAVA for the data structure?

We can use virtually any language for explaining concepts of data structure. Nonetheless, JAVA is chosen for the course and its text book for many reasons. Some of them are:

- JAVA is one of the most widely used languages in modern times. This is mainly because a program written by JAVA runs on Java Virtual Machine (JVM), which runs on most of platforms including your cellphone (Android).
- Many companies (e.g., Google) use JAVA and contribute a lot to improve the JAVA community. As a result, it is not a bad idea to learn JAVA.
- As far as I know, the ecosystem for JAVA is rich. As a result, you can grab many tools and achieve your goal in a productive manner.

What is the meaning of the parameter, String args [], in the main function?

args, the string array, contains input command strings typed in the console. Note that a lot of languages have been created much earlier and thus reflected the computing environment at that time. Early on many people used the console, simple text input/display computing env., not windows nor GUI, the common input/interaction scheme in these days.

With console, we typically pass parameters following the program name like:

```
$ ls -alF
drwxr-xr-x+ 1 sungeui mkpasswd 0 2 17:21 ./
drwxr-xr-x+ 1 sungeui mkpasswd 0 7 10:32 ../
-rwxr-xr-x 1 sungeui mkpasswd 2196 7 10:40 .log*
```

In this case we pass two strings of "ls" and "-alF" in the args so that the main function can process those information.

Can you summarize Java keywords that we use frequently in the class?

- **int**: 4 bytes integer
- **long**: 8 bytes integer: we can represent a big integer that can be represented within 8 bytes.
- **public**: Anyone can access
- **protected**: Only methods of the same package or of its subclass can access
- **private**: Only methods within the same class can access. The default modifier is friendly, which means that any class in the same package can access.
- **static**: The variable is associated with the class, not the instance of the class. This serves as a kind of global variables.
- **final**: A final variable plays as a constant value.
- **this**: **this** refers to the current instance of that class.

2 Array and Linked Lists

Can we have array with more than three dimensional spaces?

Sure. Array is represented linearly in a memory space. For example, 2 D array can be easily mapped by storing row-by-row in the memory space. In the same manner, we can map any high dimensional array into a linear space.

3 Recursion

Why are we using recursion, which seems to be complex?

There are many problems that can be naturally represented or solved in a recursive way. For example, many progression methods that we learned in high school are defined in a recursive way. For those problems, it is actually very easy to design recursive procedures.

Any recursive functions can be transformed into iterative forms?

Yes. As far as I know, there is even a theorem proving it. Intuitively, recursion is implemented in a computer using stacks and thus can be implemented iteratively by using stacks, which will be discussed later.

4 Analysis Tools

What are limitations of the random access machine (RAM) model?

The RAM model assumes that there are infinite amount of data and we can access any element in a constant amount of time. For data sets that can fit into main memory, the RAM model well approximates the case, and thus our analysis tools derived from the RAM model works well.

Nonetheless, there are large-scale data that cannot fit into main memory. In this case the data access time varies a lot depending on an actual location of the data, since some data are stored in main memory with a fast access time, and other data in disk with drastically slower access time. For this kind of cases we need to have a new computational model reflecting varying access time. Already a few well known models including external memory model have been proposed for this case.

5 Lists and Iterators

What happens to the deleted nodes in the list? If they just leave, it seems to waste its memory space.

JAVA does not provide explicit "delete" or memory release operations for such deleted operations. Nonetheless, JAVA virtual machine has a garbage collector that identify such deleted data and release automatically. This kind of approaches using the garbage collector is convenient for users, while it is less efficient than an explicit approach of releasing memory space.

6 Trees

In the class we have learned how we evaluate the tree representing an arithmetic equation. How can we then represent a tree given an arithmetic equation?

We can covert it to a tree by using a stack. I found the following example from a website called "StackOverFlow". This example is mainly about processing the equation using the stack, but we can use the same scheme for converting it to a tree.

parse	what to do?	Stack looks like
(	push it onto the stack	(
5	push 5	(, 5
+	push +	(, 5, +
2	push 2	(, 5, +, 2
)	evaluate until (	7
*	push *	7, *
7	push 7	+7, *, 7
eof	evaluate until top	49

In this example, once we evaluate (5+2) we insert its evaluation value to the stack. For tree construction, once we process the equation segment, we have a parent node which contains the operator (i.e. +) and whose child nodes have operands (i.e. 5 and 2). We then insert the parent node into the stack. We then push the next operator and operand (i.e. * and 7). Right after that we can pop out three elements and make them a sub-tree with them, as we

did before.

How can we delete a data from the tree? Do we have to scan all the nodes for locating and deleting a node?

When we do not have any information, we may have to search a particular node that you look for. In another direction, we can store a position (or a rank) of an item into its entry; we can such entries *location-aware entries*. Once we have such position, we can directly access the entry from the position. It is like saving a location of your car in the parking lot. This is covered in "Adaptable Priority Queues", which may or may not be covered in the class.

7 Priority Queues

In the class, we mainly used the binary tree for the heap. I think we can use a higher branching factor, say, three children for the tree. In that case, can we get a better time complexity for the down-heap operations?

We learn that the time complexity of the down-heap with the binary tree is $O(\log_2 n)$; when we consider the comparison operations with two children nodes, it could be $O(2\log_2 n)$.

Now, let's suppose that we have a ternary tree, each node of which can have up to three children. In this case, its tree depth is $O(\log_3 n)$. Nonetheless, when we perform down-heap operations, we can need to compare a node against its three children. As a result, its down-heap operation takes the following time complexity:

$$O(3\log_3 n) = O(3\log_2 n/\log_2 3) = O(\log_3 8\log_2 n). \tag{1}$$

Overall, its time complexity is still $O(\log n)$, while its constant factor $\log_3 8$ is slightly lower than 2.

Nonetheless, when we consider d-ary tree, its down-heap operation takes $O(d\log_d n) = O(d\log_2 n/\log_2 d)$, whose constant factor could be even higher than 2, when d becomes large, say, 5.

Advanced: When we sequentially add n data into an empty heap, what is its time complexity?

In that case, the tree depth increases from $\log 1$, $\log 2$ up to $\log n$. Its cost is $\log 1 + \log 2 + \cdots + \log n = \log n!$.

Its upper bound can be easily set by $O(n\log n)$. Surprisingly, its lower bound can be shown by $\Omega(n\log n)$, as the following:

$$\log n! \geq \log 1 + \log 2 + \cdots \log\frac{n}{2} + \log\frac{n}{2} + 1 \cdots + \log n \tag{2}$$

$$\geq \log\frac{n}{2} + \log\frac{n}{2} + 1 \cdots + \log n \text{ // we drop the first half of the right-side terms} \tag{3}$$

$$\geq \frac{n}{2}\log\frac{n}{2} \text{ // merge them into the smallest term} \tag{4}$$

From the equation, we can find proper constants and thus it is $\Theta(n\log n)$.

8 Search Trees

AVL trees and red-black trees has the same $O(n\log n)$ time complexity. In what cases do we use AVL or red-black trees?

AVL trees requires at most one level difference between the left and right subtrees. On the other hand, red-black trees requires less stringent constraint. As a result, red-black trees are more useful for cases where we have many

updates (i.e. adding or deleting), while AVL trees are more useful for rather static cases that do not have many update.

Why do we use red-black trees, which seem to be very complex?

Red-black trees guarantee $O(n \log n)$ worst time complexity for adding and deleting elements in the tree. This property is very useful for various applications that requires high performance. Furthermore, red-black tree is used as a building block in many other data structures for guaranteeing the worse case time complexity.