Algorithms and Datastructures (HS2024)
Week 6

Rui Zhang

October 28, 2024

Contents

1 Revision Theory

1.1 Dictionaries / Maps

Dictionaries are datastructures that contain key-value pairs. The keys have to be unique but the values do not. For every key, the dictionary will give you the value corresponding to that key.

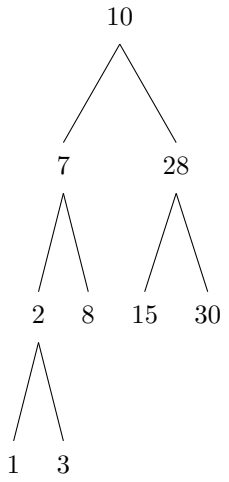
The most important part of this structure are the keys, which is why we will dumb this datastructure down to a collection W of keys. We have the functions *search(x, W)*, *insert(x, W)* and *delete(x, W)*. Search returns the position in memory where x is stored if x is in W. Insert adds key x to the collection W of keys if x is not already in W. Delete removes the key x and its corresponding value from the datastructure if x is present in W.

1.2 Binary Search Tree

1.2.1 Motivation

To implement a dictionary with "good" runtime, we will first try to use so called binary search trees, where the key-value pairs are inserted as nodes into the binary search tree. You have seen a similar principle with heaps. However, this time, we have a different structure.

Consider the following tree:



This is a binary tree, because the key of the right child is always larger than the key of the parent and the key of the left child is always less than the key of the parent. This is the main property of binary search trees: they are trees with a maximum of two children per parent and the keys are "sorted" like above.

Now, using this structure, we may implement search, insert and delete as algorithms very similar to binary search (hence the name). The exact details of

the implementation is left as an exercise to the reader (but of course, I will be ready to help and answer any questions).

Note that the delete operation has a few quirks however, since if we are to delete an element which has children and multiple descendants, then we will have to find an element to replace this deleted element with in order to preserve the structure.

The exact idea here is to replace the deleted element with the next largest element in the binary search tree. To do this, we traverse the tree by going to its right child first, then finding the left-most descendant of this right child. Note that this left-most descendant l cannot have a left child and thus we may use this node l as replacement and replace l with its only possible right child (if it even had a right child).

1.3 AVL Trees

Binary Search Trees are not as good as you think they might be. In the average case, the depth of the tree will indeed be $O(n \log n)$. However, assume we have a empty tree and now we call the following functions in order:

$$insert(1), insert(2), insert(3), insert(4)$$

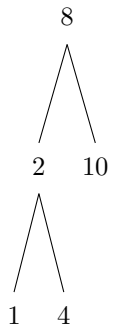
Notice that this will result in the following "tree":

Now, we have a very degenerate case of a binary search tree and the depth of this tree is $4 = n = O(n)$.

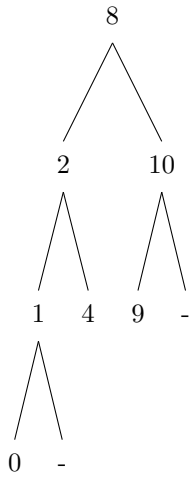
1.3.1 Definition & Examples

To improve this aspect, the lecturers introduced AVL trees in class. AVL Tress are just binary search trees, but they make sure that the binary tree is balanced. The invariant for AVL trees is that for each node, its descendant trees have

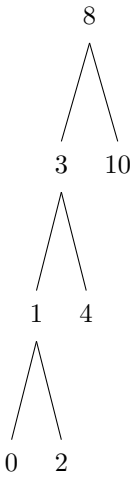
to have heights that do not differ by more than one unit. This assures the "balanced" part. Let us consider a few examples:



Here, the height of the left descendant tree of 8 is 2 and the height of the right descendant tree of 8 is 1, so this is fine. Consider now the follwing example, where "-" stands for a non-existent child (my latex skills are garbage so I did not know how to represent this better).



This is still a valid AVL tree despite having quite a skewed shape (if you ignore the "null"s), since the left subtree of 8 is of height 3 and the height of the right subtree of 8 is 2. The height of the left subtree of 2 is 2 and the height of the right subtree is 1. and - yeah you get what I mean.



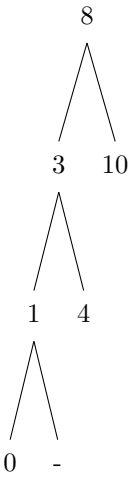
This however, is an example of a tree that is even more skewed and that violates the AVL-condition. Notice that the left subtree of 8 is of height 3 and the right subtree is of height 1.

1.3.2 Rotations & Restoring an AVL Tree

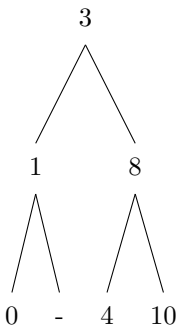
In the lecture, you have seen a really theoretical explanation of how to restore an AVL tree that I will try to dumb down here. However, after reading this document (and hopefully understanding / learning something), I would recommend you go back to the official lecture notes draw actual examples for each individual case your professor went through. This will help you in understanding the material on a much deeper level.

The idea is to add an algorithm to the "insert" operation such that it checks each ancestor of the newly inserted node for the AVL-condition. If we do not have an AVL-condition-violation, then this corresponds to the cases 2A and 2B in the lecture notes. If at any point we have a violation of the AVL-condition, we need to get to work (case 2C).

Single Rotations:
The first case of an AVL-violation is just similar to the example from above:



Here, we just inserted the 0 and now the AVL-Condition is violated. If you compare this example with the lecture document, you can see that we have the case "2Ca", where u is 8 and v is 3. The idea is to "rotate along v" and pass the right child of v (if it exists) to u:



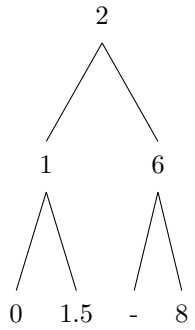
Try to visualize this rotation in your head and understand the analogy to "rotating". If you have any questions or need a more in depth explanation you can reach me any time.

Double Rotations:
This is case 2Cb. Let us look at the following tree:

Here, $u = 6, v = 1, w = 2$ In this case, the problem lies in 4. However, we cannot do the simple rotation we did before. Try it out, it will result in a tree which has the same bad structure as this tree has, but just on the right side instead of on the left side.

Instead, we will do a double-rotation, which consists of rotation in the opposite (left) direction along $v = 1$ first, analogous to the way we did in the single rotation:

And now we rotate in the forward (right) direction:



I encourage you to try out even more examples or do some exam exercises to get used to these rotations. To check if your solutions are correct, you can check out the AVL Tree visualization by the University of San Francisco which you can find linked on the website under this week's box.

1.4 Dynamic Programming

Instead of explaining all examples of DP you have gone through in the lecture, I will go into more detail on one specific example and explain how a DP Exercise should be solved in an exam. However, I heavily encourage you to revise the DP algorithms you have learned in class often to have a good basis for inspiration for DP exercises in the exam.

In this document, we will be looking closer at Minimum Edit Distance. In this problem, we are given two strings A (of size n) and B (of size m) (saved as arrays of characters) and would like to find the minumum number of operations to turn A into B. Here, we consider inserting a character, deleting a character and changing a character to be valid operations.

DP Algorithms are basically recursive algorithms with a datastructure saving the result of previous calculations to avoid unnecessary recursions. Addtionally, instead of actually writing a recursive function, we write the code "bottom-up" (i.e. as multiple nested loops) to avoid the runtime-overhead generated from recursive programming (resulting from some technical details we will not look into closer here).

When presented with an exercise like this in an exam, whether that be the theory or the CodeExpert exam, it is often necessary but also incredibly helpful to establish the following questions:

1. Definition of the DP-Table (Meaning of each entry, how many dimensions)?

2. Computation of an entry (How do we calculate one entry based off of previous entries we have already calculated? And which cases / entries are not dependent on other entries at all and may be filled in at the start?) (base cases are usually something like on the top most / left most / diagonal entries)

3. Calculation Order: In which order do we have to calculate the entries so that dependencies are always fulfilled?

4. Solution Extraction: How do you get the solution from the table? (often something like looping over the table (a constant number of times) or reading one single entry)

5. Runtime of your algorithm? (usually similar to the size of the DP-Table)

We will go through the Minumum Edit Distance now by answering each of these questions. You are encouraged to stop before a section to try to find the answer yourself by thinking back to the lecture from last week.

1.4.1 Definition of the DP-Table

We have two strings. Maybe we can let a DP entry $DP[i][j]$ be the minimum number of edit operations to convert the substring $A[1, ..., i]$ into $B[1, ..., j]$. Then, we could extract the solution by just returning the value of $DP[n][m]$. Furthermore, it may be a bit intuitive (or not, that is also totally normal), that we could calculate one entry (the min. number of edit operations to convert $A[1, ..., i]$ into $B[1, ..., j]$) by considering the min. number of edit operations to convert $A[1, ..., i-1]$ into $B[1, ..., j]$ or $A[1, ..., i]$ into $B[1, ..., j-1]$ (DP entries close to $DP[i][j]$). But in any case, let us just try out this approach and see where it leads us.

In this case, we would have a dimension of something like $n \times m$ (plus minus 1).

1.4.2 Calculation of a DP-Entry

How would we calculate one entry from others? Well, let us consider DP[i][j] (the min. number of edit operations to convert $A[1, ..., i]$ into $B[1, ..., j]$). One way of getting **a** (not the minimum) number of edit operations to convert $A[1, ..., i]$ into $B[1, ..., j]$ is by looking at the number of operations we need to convert $A[1, ..., i-1]$ into $B[1, ..., j]$ ($DP[i-1][j]$). The reason for this is that once we have converted $A[1, ..., i-1]$ into $B[1, ..., j]$, we can just delete $A[i]$ and then we would have converted $A[1, ..., i]$ into $B[1, ..., j]$ (i.e. $DP[i][j] = DP[i-1][j] + \mathbf{1}$ for this case.)

Another way is to consider converting $A[1, ..., i]$ into $B[1, ..., j-1]$. Then, we would only need to add / insert $B[j]$ afterwards, then we would have converted $A[1, ..., i]$ into $B[1, ..., j]$ (i.e. $DP[i][j] = DP[i][j-1] + \mathbf{1}$ for this case)

The last way to do this is changing a character: If we had already converted $A[1, .., i-1]$ into $B[1, ..., j-1]$, then we could convert $A[1, ..., i]$ into $B[1, ..., j]$ by changing $A[i]$ into $B[j]$. And if $A[i] = B[j]$, then we would not even need another operation! ($DP[i][j] = DP[i-1][j-1] + (1 \text{ or } 0)$ depending on if $A[i]$ or $B[j]$ are equal).

And there are no other possible operations other than inserting, deleting and changing. So there should not be another way to calculate $DP[i][j]$. Since we

want the minumum of all numbers of edit operations, we will take the minimum of the three options.

So which are the so called base cases then? The ones that we may fill in immediatly at the start without the need of other information except for the definition of the problem? In this case, consider how many operations it would take to convert an empty string into another string of length i. It would take i edit operations at least (i insertions), right? So we may extend the DP-Table by another row and another column to account for empty strings as well and define $DP[0][j]$ to be the number of operations going from an empty string to $B[1, ..., j]$ (j operations at least) and $DP[i][0]$ to be the number of operations going from $A[1, ..., i]$ to an empty string (i operations at least - i deletions). That means that $DP[i][0] = i$ and $DP[0][j] = j$ are the base cases.

1.4.3 Calculation Order?

Now, it always helps to draw the table. Consider some entry that is not a base case. Which other entries does it look at to calculate its own value. Well, it will be the entry left of it, the entry above it and the entry to the top left corner of it. Thus, if we calculate the entries from left to right, from top to bottom, we should never have any problems. (exercise: translate this order into i and j).

1.4.4 Extraction of Solution

Which entry, according to our definition, would be exactly the answer to the problem we have? Try it yourself, otherwise look at the lecture notes from your professor.

1.5 Runtime

We calculate each entry once. And we have $O(nm)$ entries, so that is a runtime of $O(nm)$.

1.5.1 General Tipps and Advice

What I wanted to illustrate in this process is that oftentimes, it is also about trial and error, starting at the definition of the DP-Table, but in an educated sense: We try out something that may seem intuitive "But in any case, let us try out this approach". If this attempt goes south (which in this case it did not), then I advise you to formulate an exact reasoning as a full sentence in your brain as to why this DP-entry Definition is not optimal / not correct. Based on this sentence, you will oftentimes be able to find a better way to define the DP-Table entry.

2 Exercises

2.1 Exercise Sheet 6 - Priority List

Again, top is most important, bottom is least important.

1. 5.1), 5.4) Exercises where you have to apply algorithms to datastructures such as heaps are common in exams
2. 5.3) Still a bonus exercise
3. 5.5) Being able to implement datastructures is an essential skill not just for A&D
4. 5.2) this exercise deepens your understanding of the proof pattern used for the the limits of comparison based sorting and searching.

2.2 Exercise Sheet 4 - Feedback

1. Please be careful of the induction hypothesis. In exercise 4.3, it is incredibly important that $k < n$! This is the main pitfall of the induction hypothesis step and many of you fell for it!
2. The initialization of new arrays of size n is in $O(n)$. This ruined the demanded runtime of $O(\log n)$ for some of you. Be careful in the future.

3 Supplementary Exercises

3.1 Trees

Draw a Venn-Diagram illustrating the relationship between the structures tree, binary tree, heaps, min-heaps, max-heaps, AVL-Trees and binary search trees. For example, a heap is a tree, thus, the circle containing heaps would be strictly contained within the circle which represents the set of all tress.

3.2 AVL-Tree Rotations

You have seen examples of rotations and double rotations towards the right in these notes and in the lecture document. Now, write down an analogous example for a rotation and a double rotation towards the left.

3.3 Dynamic Programmming

The next two exercises concern DP. Before tackling them, first remind yourself of which steps / questions should be addressed in every DP Exercise.

3.4 Longest Common Continuous Subsequence

Given two strings $A[1, ..., n]$ and $B[1, .., m]$, find the length of the longest common **continuous** subsequence between A and B. You may take inspiration from the Longest Common Subsequence Example from the lecture.

3.5 Coins

Given a set of coin-values C and a number n representing a value you want to accumulate using only coins with values in C, what is the minimum number of coins you have to use to get this value C.