



Kristianstad
University
Sweden

Kristianstad University
SE-291 88 Kristianstad
+46 44-250 30 00
www.hkr.se

DA256D, 7,5 credits
Semester Year e.g. Autumn Semester 2021
Faculty of Natural Science

Seminar 3 – Hashing and trees

Sandra Kaljula

Author

Sandra Kaljula

Title

Hashing and trees – Seminar 3

Supervisor

Kamilla Klonowska

Examiner

Kamilla Klonowska

Abstract

Data structures like hash tables and trees have different operations and time complexities for each of them. In this report different approaches to these data structures and their operations are compared.

Keywords

Algorithms, data structures, big-O, trees, priority queue, hashing.

Content

Introduction	4
Method	5
Results	6
Task 0	6
Task 1	8
a)	8
b)	8
c).....	9
d)	10
e).....	11
Task 2	14
<i>Chaining hash table</i>	14
<i>Linear probing hash table</i>	14
<i>Quadratic probing hash table</i>	14
<i>Double hashing table</i>	15
<i>Double hashing table size</i>	15
<i>Other rehashing functions</i>	15
Task 3	16
Task 4	17
<i>Insertion</i>	18
<i>Deletion</i>	20
Conclusion	23
References	24
The code(also mentioned in each java file):	24
Appendix A	26

Introduction

This report is about solving the seminar 3 tasks and some reflections on them.

Limitations:

The algorithms are not guaranteed to be the most efficient ones out there, since they were the first working ones I came up with or were found on the internet. This is due to the time limit for this exercise. The exercises are also coded and benchmarked in java, which might bring different running times compared to other programming languages.

Method

First, I chose to use java to program my algorithms in, because I have lately used both python and Java and I would want to be better at this language.

Second, a file structure was created with all the files needed for execution. It is very important to have everything organized, since there were over 10 files in total.

Third, the information needed on various kinds of hashing and trees was gathered.

Fourth, I moved into writing the algorithms for the different tasks. It was decided to borrow some of the algorithms from the internet or the course literature and altered them to understand them. The references are found in the references part of the report as well as in the java files.

Then finally at the end of task 1 and 4 the tests were run automatically on that task on a computer with AMD Ryzen 3 1300X Quad-Core Processor 3.80 GHz. The results were drawn in excel as graphs and tables which are attached with this report.

Results

Task 0

Reflection on Fast and Compact Hash Tables for Integer Keys.

A hash table is a data structure that can store keys and their corresponding values. With the key the data(values) can be retrieved very fast. It is usually used with string keys. However, in this paper we are introduced to another approach which is using integer keys, which apparently is much faster.

The hash table's functionality is as follows. The methods for this data structure are insert, delete and search. When first creating a key pair, the keys are distributed amongst several slots by using a hash function. The distribution must even work when a collision happens. Collision happens when there are more than one keys with the same key value.

One way to deal with collisions is linked lists, which create a chaining hash table(standard chain hash table). Meaning that there are pointers to the next element in the elements.

But this is not the only way to solve this problem with cheap collision costs. All the pointers and string hashing can be avoided altogether with having keys as integer values. Cuckoo hashing uses two hash tables and two different hash functions. This prevents collisions because when a collision is raised the next hash function and table will be used to solve it. This will be repeated until there are no more collisions. The cons of this can be useless pointers in the memory as well as lots of memory allocation overhead.

There are even more approaches to hashing - array hash table, standard-chain hash table, clustered-chain hash table and bucketized cuckoo hashing.

Judging by the graphs presented in the article bucketized cuckoo was the worst so far, unless under extremely hard load. The most efficient option was the array hash.

In conclusion it depends on the situation, some of the data structures are better in some cases and some in others it is up to the programmer to choose the right approach[1].

Task 1

The inputs for these subtasks originate from Seminar 1's tasks. In the following figure 1 there are the outputs for algorithm 1 and algorithm 2 min heaps, that will be discussed further in this report.

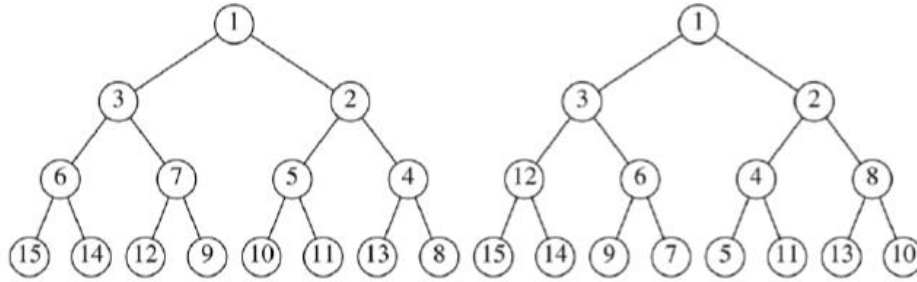


Figure 1 Output of algorithm 1 - task 1a (left tree) and algorithm 2 - task 1b (right tree).

a) Algorithm 1 is about inserting all the values one by one into an empty binary heap. The runtimes in Table 1 were tracked over five cycles and in nanoseconds. The values for input were from ten to one million different inputs. This algorithm should have the time complexity of $O(N \log N)$.

b) Algorithm 2's running times are also displayed in table, which were also traced the same way. But algorithm 2 is much different since its input was an array which was then copied into a heap and then heapified. This algorithm should have the time complexity of $O(N)$.

Input size /Algorithm	10	100	1000	10000	100000	1000000
Algorithm 1	181740	159240	88860	577440	5705520	13652360
Algorithm 2	183750	23520	136980	502980	1430940	8842800

Table 1 Average running times of both algorithms for different input sizes in nanoseconds over five cycles.

c) The results of the four traversing strategies applied on:

Algorithm 1's heap:

in-order: 15 6 14 3 12 7 9 1 10 5 11 2 13 4 8

pre-order: 1 3 6 15 14 7 12 9 2 5 10 11 4 13 8

post-order: 15 14 6 12 9 7 3 10 11 5 13 8 4 2 1

level-order: 1 3 2 6 7 5 4 15 14 12 9 10 11 13 8

Algorithm 2's heap:

in-order: 15 12 14 3 9 6 7 1 5 4 11 2 13 8 10

pre-order: 1 3 12 15 14 6 9 7 2 4 5 11 8 13 10

post-order: 15 14 12 9 7 6 3 5 11 4 13 10 8 2 1

level-order: 1 3 2 12 6 4 8 15 14 9 7 5 11 13 10

d) Finally, a comparison between the running times of building algorithms 1 and 2. See figure 2. Algorithm 2 seems to be faster for smaller inputs as well as bigger inputs as it should be for $O(N)$ complexity. However, algorithm 1 and 2 are really close to each other until about one thousand inputs. This could mean that algorithm 2 better for use if the number of inputs is unknown. But there is a spot at one thousand inputs where the algorithm 1 does a bit better than algorithm 2 but the difference is not that important since it is very slight.

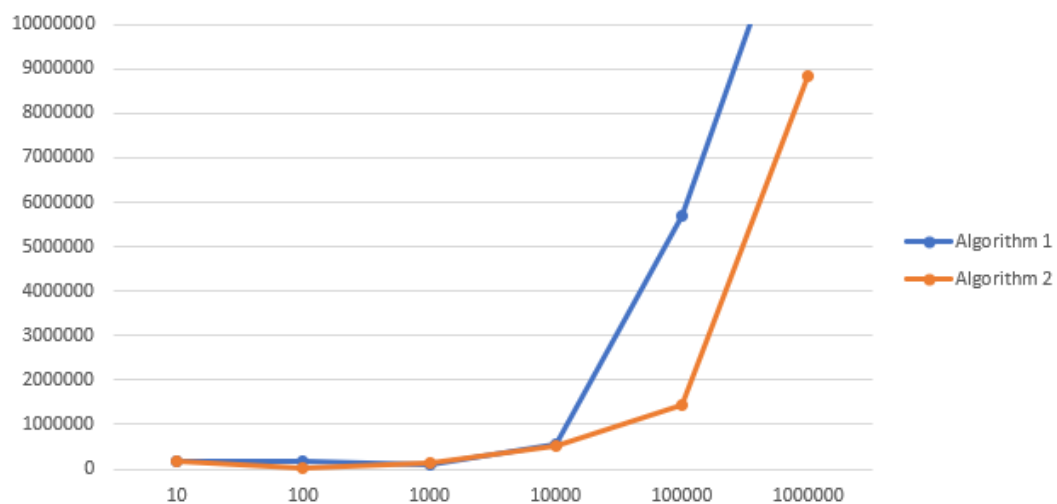


Figure 2 Average running times of both algorithms for different input sizes over five cycles.

In heap or priority queue the time it takes to search in the worst case is $O(N)$, while the best case is $O(1)$. The best case happens when the node that is searched after is the root node. Insertion is $O(\log N)$ in the worst case and can often be $O(1)$ since it might already be in its correct place. Deletion of a node always takes $O(\log N)$, because it must check with the rest of the heap that it's all ordered correctly.

e) For this task's tables see appendix A. Figure 3 shows the average running times of inserting and deleting for both algorithms with inputs from ten to one million in nanoseconds.

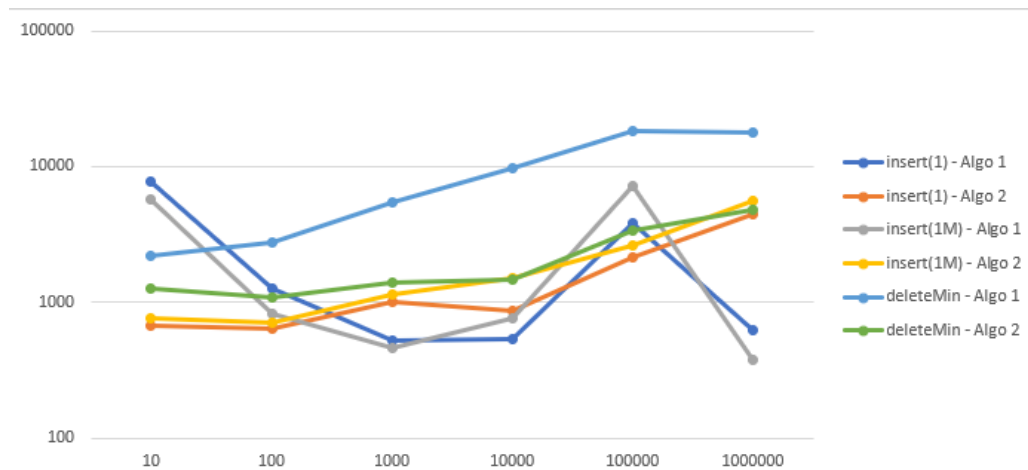


Figure 3 Average running times of inserting and deleting for both algorithms with different input sizes over five cycles

When only looking at inserting the very small number that in this case was 1, we see that it really depends on the values that are in the heap. See figure 4. But the values used for the same input sizes are the same so we can still look at the differences between the dots. The start and end seem to have a drastic difference whilst everything in the middle does not. The inconsistent line for algorithm 1 could be explained with something happening in the computer that messes up the results. Otherwise, it looks like that the first algorithm takes less time to insert a new small value that possible replaces the root and moves the rest of the heap around accordingly.

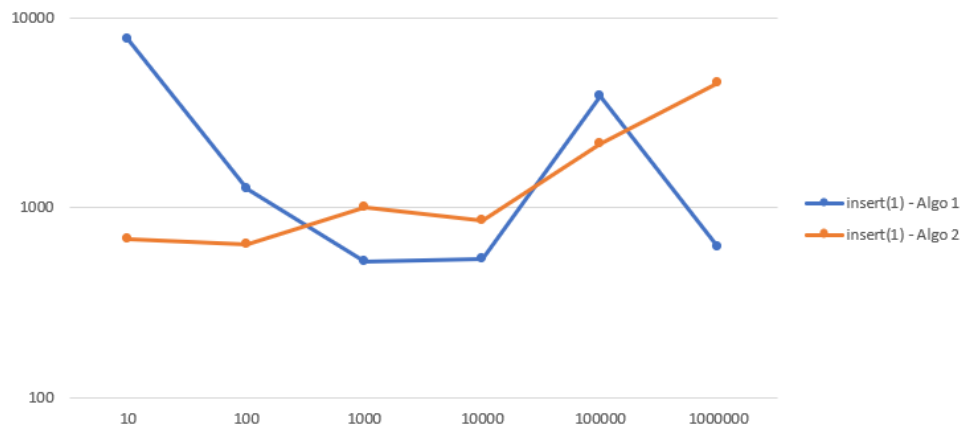


Figure 4 Running times for both algorithms inserting a small value.

For the large input the results for inserting are very similar to inserting the small value. See figure 5. It seems to take slightly less time to add a large number compared to the small one, but the difference is near nonexistent.

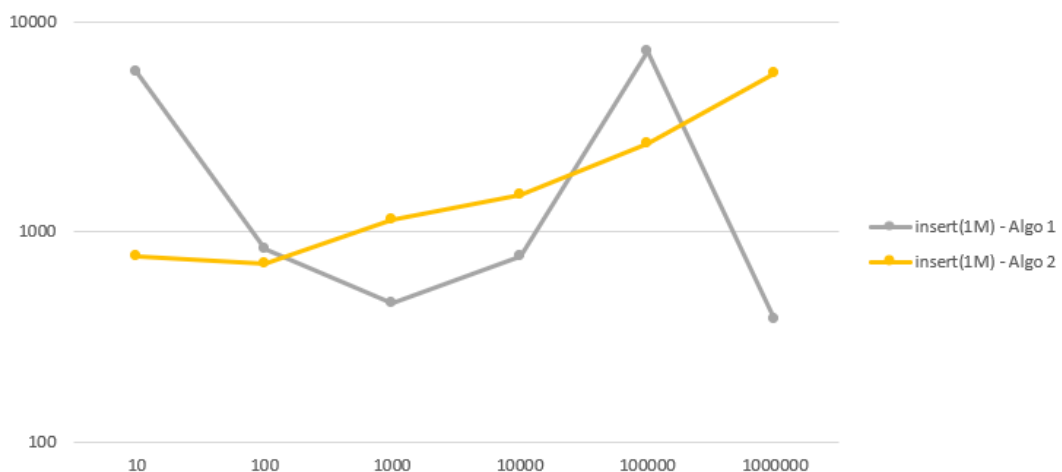


Figure 5 Running times for both algorithms inserting a large value.

For deleting the minimum value or in this case the root the graphs look as following. See figure 6. The graphs seem more evened out now and the rapid rise does not seem to be a problem any longer. It takes less time for the algorithm 2 to remove its minimum element than it takes algorithm 1.

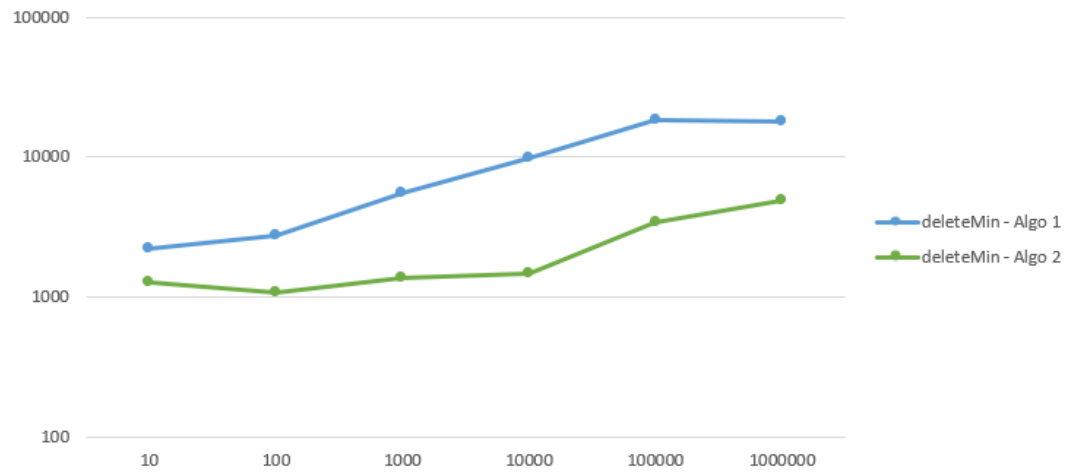


Figure 6 Running times for both algorithms deleting the minimum element.

In conclusion the deletion seems to be the most expensive operation judging by figure 3 where all the operations average running times are gathered.

Task 2

Collisions in hash tables happen when there is already a key with an existing value. There are different approaches to solving collisions. In this task we will get into the most popular approaches.

Chaining hash table

One of them being a separate chaining hash table, which handles collisions by creating a linked list to the record, which keys match. If there is already an existent linked list the value is added to the end of the linked list. The time complexity for adding and removing elements will almost always be $O(1)$. There is no real limitation to number of keys for this approach since there can be a countless number of values stored for every key. But this could lead to a problem, which is waste of space. Having many collisions and just a few successful inserts could leave most of the memory located to the hash table empty. The worst case in that scenario would be inserting an element that is stored at the end of a very long linked list. This could potentially bring it to have the worst-case scenario of $O(N)$.

Linear probing hash table

Another one is a linear probing hash table, in which the collisions are handled by finding a new available spot for the value instead. This approach can become a problem when there are no available spots left after everything has been added sequentially and it must go check a lot of spots. That would bring a $O(N)$ worst case scenario where N is the number of elements before the free spot. This is also called primary clustering. Otherwise, if the table is empty, it will be the best-case $O(1)$ time.

Quadratic probing hash table

An alternative for linear probing would be a quadratic probing hash table, in which the collisions are handled by finding the next available spot by adding x in the power of two, where x is the counter of how many times there has been the same collision. This could be a solution to the previous problem with linear probing. But there is another problem with this – it might still only check the spots

that are filled with something. Therefore, it might keep looping for long periods of time bringing the worst-case scenario of $O(N)$, otherwise the time complexity will be $O(1)$ in the best case. This is when there are many slots. This brings a problem of only checking around like 60 % of the slots and skipping the rest. This could be solved by letting it loop again from the beginning, but if there are no spots left there is nowhere to put the key.

Double hashing table

A completely different approach would be using double hashing. In case of a collision there is another hash function through which the new spot will be found. But this does not always guarantee that there will be an available spot. This must be complemented with another method for hashing to work even better. Which brings us to the next approach.

Double hashing table size

Using a double hash and entering them to a new table with double the size is the best solution. Especially, when the table size is a prime number. The time complexity will always be $O(1)$ until there is no rehashing. If there is a collision, we would have to increase the table size to the double of the current size we would have to rehash all the values. This would be of $O(N)$ complexity. But this must be done only every now and then, making it very efficient.

Other rehashing functions

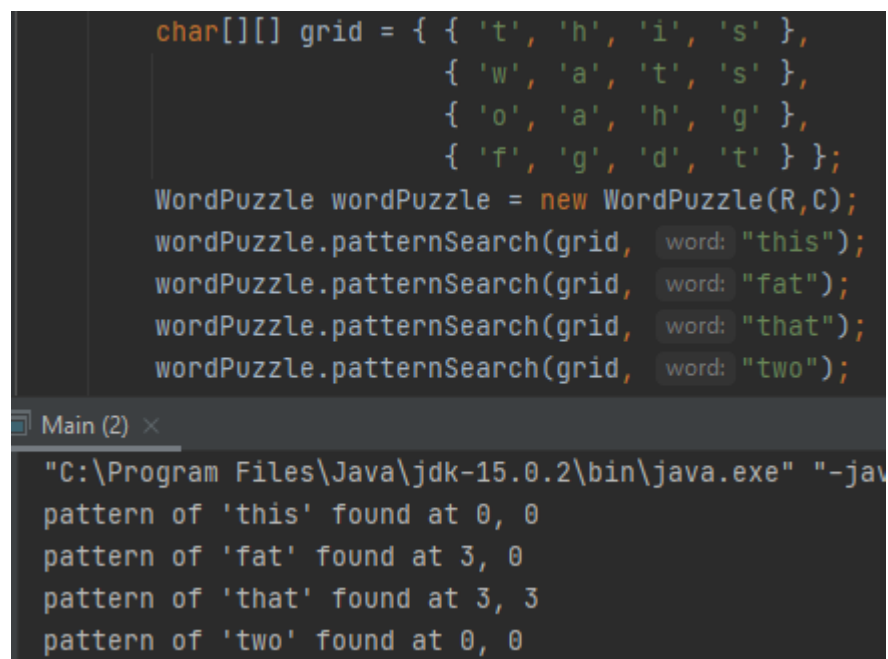
There are a lot of different hashing methods and algorithms. An example of one would be cuckoo hashing mentioned in task1. Cuckoo hashing uses two hash tables and two different hash functions. This prevents collisions because when a collision is raised the next hash function and table will be used to solve it. This will be repeated until there are no more collisions[1].

Task 3

For input to the 2d word puzzle there is a two-dimensional array with letters and a word that is searched for. In this task it was decided to go with the implementation of the triple (row column and orientation).

In the WordPuzzle class there are arrays with all the possible directions which in this case is 8 and 2 variables that indicate the number of rows as well as columns. This class has two methods. In the first one it is a nested for loop that checks all the rows and columns and calls the other method in which every new character is checked if it's the starting character for the word, if it is it does all the checks needed for the search in all directions. Finally, if the word is found it returns true otherwise it returns false. This caused a lot of nested loops, therefore a very high complexity $O(n^4)$ and $O(n^2)$ as lowest.

The results for searching and finding the words that were also searched for in the example are shown below in Figure 7.



```

char[][] grid = { { 't', 'h', 'i', 's' },
                  { 'w', 'a', 't', 's' },
                  { 'o', 'a', 'h', 'g' },
                  { 'f', 'g', 'd', 't' } };

WordPuzzle wordPuzzle = new WordPuzzle(R,C);
wordPuzzle.patternSearch(grid, word: "this");
wordPuzzle.patternSearch(grid, word: "fat");
wordPuzzle.patternSearch(grid, word: "that");
wordPuzzle.patternSearch(grid, word: "two");
  
```

```

Main (2) ×
"C:\Program Files\Java\jdk-15.0.2\bin\java.exe" "-jav
pattern of 'this' found at 0, 0
pattern of 'fat' found at 3, 0
pattern of 'that' found at 3, 3
pattern of 'two' found at 0, 0
  
```

Figure 7 Result of running task3.

Task 4

a) First, the complexity of building the different tree structures should be compared. Therefore, I decided to run tests for running time for each algorithm with inputs from ten to one million over 5 cycles and get the average it took. See figure 8 and table 2, where BST is the Binary Search Tree, BH is the binary Heap, RBT is the Red Black Tree and finally AVT is the AVL Tree. All the data in this task is measured in nanoseconds.

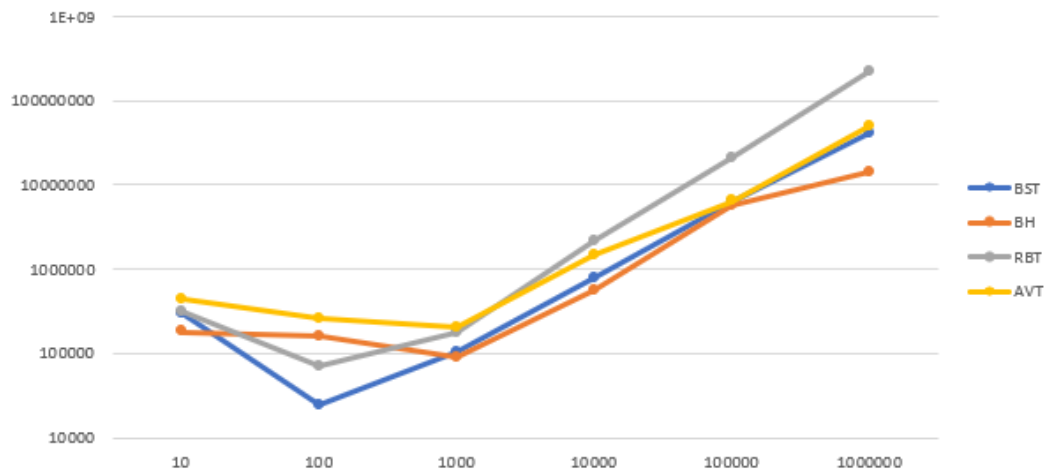


Figure 8 Average running times of all the algorithms for different input sizes over five cycles.

algorithm/input size	10	100	1000	10000	100000	1000000
BST	306400	24680	103700	806760	6191420	40597980
BH	181740	159240	88860	577440	5705520	14000000,00
RBT	323320	70180	173800	2208520	21089980	225593460
AVL	446100	258600	206540	1476520	6457000	50694080

Table 2 Average running times for the algorithms for different input sizes from ten to one million in nanoseconds.

It seems to be that at the start the time elapsed seemed to be a bit higher than the next step, this could be due to the JVM making more space for the operations. Therefore, we should look more from the data that is from one hundred and upwards.

In figure 4 binary heap seems to be the fastest to build for larger inputs, this could be due to that it only has to check that that the number is smaller than the parent and the children are smaller than the current node in case of a minimum heap. While the other trees must sort everything perfectly after their rules, which are more needy than binary heaps are.

The previous time was just for determining how long the tree structures take to build. But let's look at the operations all the trees have in common – insertion and deletion.

Insertion

Inserting a small number into the trees is different to inserting a very large number to the tree. So, let's compare the two. First in figure 9 and table 3 we see inserting the smallest value of 1 into the trees.

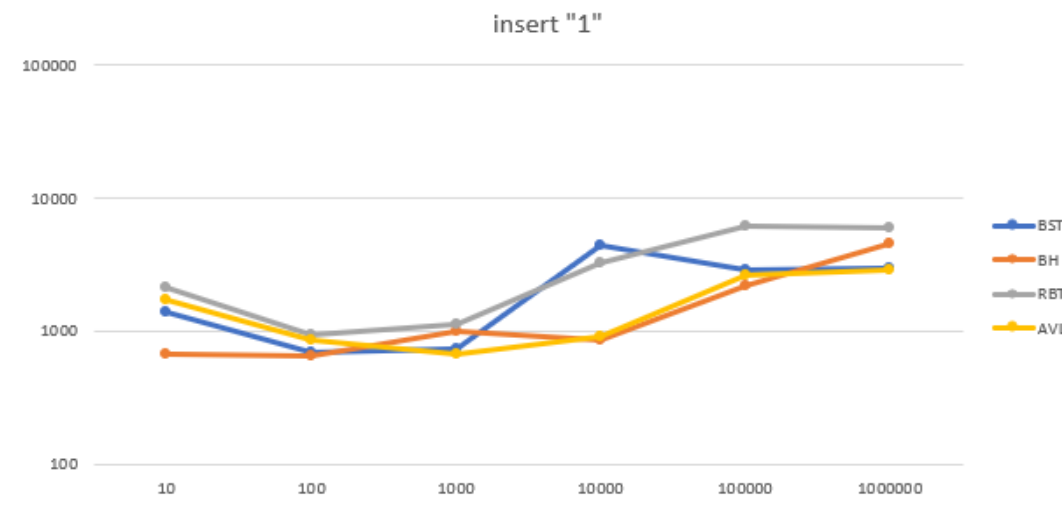


Figure 9 The running times for different algorithms of inserting a small value.

algorithm/input size	10	100	1000	10000	100000	1000000
BST	1380	700	740	4340	2860	2940
BH	680	640	1000	860	2160	4500
RBT	2120	940	1120	3240	6220	5960
AVL	1700	860	660	920	2600	2900

Table 3 Average running times for the four algorithms from inputs ten to one million in nanoseconds.

Second, in figure 10 and table 4 we see inserting the largest value of one million into the trees.

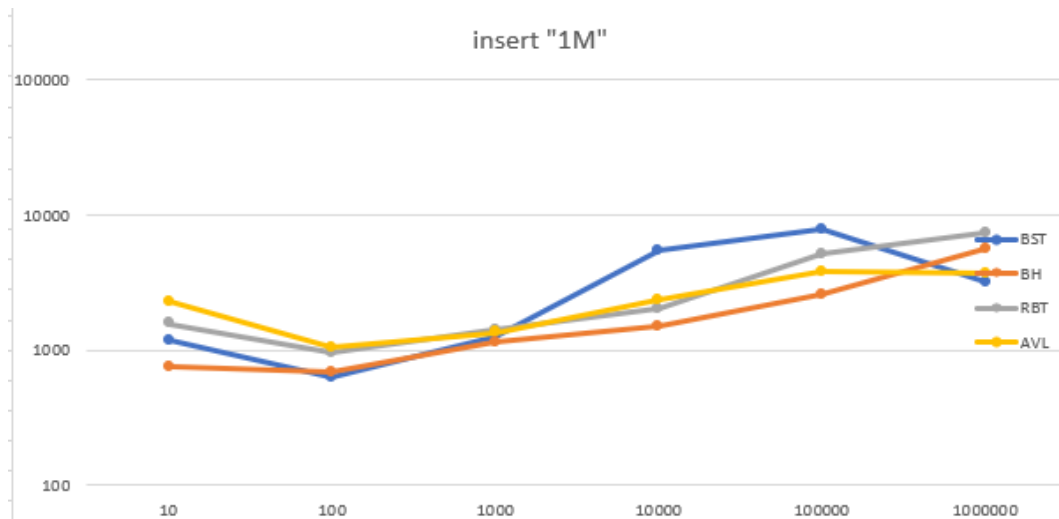


Figure 10 The running times for different algorithms of inserting a very large value.

algorithm/input size	10	100	1000	10000	100000	1000000
BST	1180	640	1280	5500	7800	3200
BH	760	700	1140	1500	2620	5640
RBT	1580	960	1440	2060	5100	7360
AVL	2320	1060	1360	2340	3800	3680

Table 4 Average running times for the four algorithms from inputs ten to one million in nanoseconds.

In the binary tree the average case for inserting an element is $O(\log N)$, if the number is very small it will go to the most left node and if the element is large it will go to the rightest side. The worst-case scenario of inserting in a binary tree is when all the elements are on one side of each node, creating a linked list like scenario with a time complexity of $O(N)$ for all operations. Then when inserting a value to the end of the “linked list” it must look through all the elements before putting it to the end of the rightest side.

In the binary heap inserting a very small value means that the root must be replaced by a new one, this means that the tree must be moved around (the root must be swapped with the child and so on) leaving a complexity of worst-case $O(\log N)$. Whilst inserting a very large number it will most likely already be in its place which gives us a $O(1)$ best case scenario.

In the red-black tree there are many cases for insertion since the red and black colored nodes must be correct. The average case would be $O(1)$ complexity and the worst case is $O(\log N)$ when all the nodes are the same color.

In the AVL tree insertion is the average complexity of $O(\log N)$ since when the tree becomes unbalanced it has to be moved around.

For constantly inserting large values the binary heap should be used since it seems to almost always take the least amount of time according to my tests.

Deletion

For the deletion operation it was decided to only look at deleting the minimum value in the trees. Figure 11 and table 5 shows the differences in running times for deleting the minimum value element.

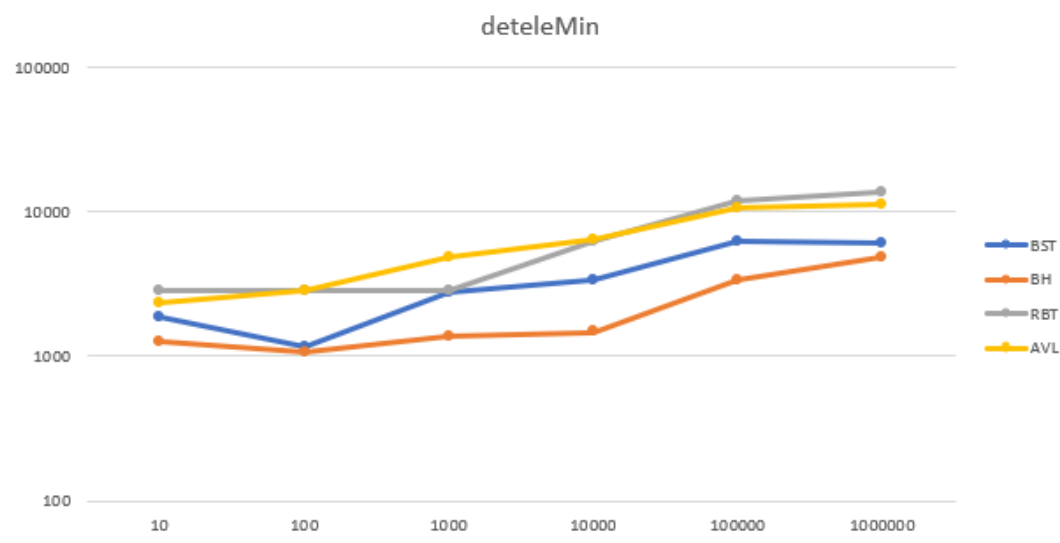


Figure 11 Running times of deleting the minimum element in the different algorithms over 5 cycles.

algorithm/input size	10	100	1000	10000	100000	1000000
BST	1900	1180	2820	3380	6220	6100
BH	1260	1080	1380	1480	3420	4860
RBT	2880	2880	2860	6220	11840	13700
AVL	2360	2860	4840	6480	10760	11240

Table 5 Average running times for deletion of the minimum element for the four algorithms in nanoseconds.

In a binary search tree, the minimum value is going to be on the most left bottom side, which means that the minimum must be searched first, which on average is $O(\log N)$. When the node is found it can be deleted and its $O(\log N)$ on average.

In the minimum binary heap, the root is the minimum element, so no search operation must be done. So, finding the minimum is always $O(1)$. Deleting min is also of $O(\log N)$ as the previous tree.

In the red-black tree the search for an element takes $O(\log N)$ since the root is not the minimum element similarly to the binary search tree. The deletion is also $O(\log N)$ on average.

The AVL tree's complexity for deletion is $O(\log N)$ as searching is $O(\log N)$ complexity. But there must be some configurations done in the tree, so it takes a bit longer than the others.

Due to not having to search for the minimum element the Binary Heap is the fastest of all the trees for deleting the minimum. The AVL tree is also the slowest since it must balance the elements every now and then.

b) All of the trees mentioned above have different properties. But some trees are better to use in some scenarios than the others.

The binary search tree can be used for implementing search algorithms like binary search that has a complexity of $O(\log N)$. Using a binary heap for that purpose would not work as well since there can be multiple nodes with the same values and the nodes can be in completely different places. In the binary heap it, all depends on when the value is added to the heap meaning that the nodes are relatively unsorted at their levels, which would cause $O(N)$ time in the worst case to only find a node.

But in case of a needing to access the minimum or the maximum node most of the time, the binary heap would be the best option since in one minimum or maximum heap the min or max is kept as the root node and the complexity of getting access to the root is then $O(1)$ complexity, while in the binary search tree it would be $O(\log N)$ since it has to get the element that is located at the most left or right of the tree.

As seen in the previous time measurements the time for insertion and deletion is the best out of all four tree algorithms. This could mean that the binary heap could be used for quickly storing different data. This could be used for schedulers or

similar kinds of programs that use of priority queue. Since in schedulers it must get the earliest or highest priority task first every time. While adding more of the lower priority tasks at the bottom.

As seen in the graphs, the red-black tree takes more time build and most of the operations also take more time than the AVL tree, this is due to the re-black tree being more complex than the AVL tree. The results were very close to each other therefore I think the AVL tree and red-black tree could be used for the same purpose. But as I investigated further the AVL tree seems to be more balanced and usually offers a better lookup time for the nodes. Therefore, the AVL tree seems to be used in memory management subsystems in Linux kernel to search memory regions of processes during preemption[2]. The red-black tree in the other hand is used for computational geometry as well as the completely fair scheduler in the Linux kernel[2].

Conclusion

This seminar was more confusing than the previous seminars. It was very stressful being under such time press. However, I learned a lot about different kind of hash tables, trees and how to use them and in which scenarios. I think I also became better at “benchmarking” the different data structures and using excel.

Out of the two data structures hash tables was most definitely the easier to understand, while trees were bringing a lot of confusion and I oftentimes had to make notes on a scrap of paper to get a full understanding and overview of everything.

References

[1] Askitis N. Fast and Compact Hash Tables for Integer Keys. Wellington: 2009 January; p. 113-122.

https://hkr.instructure.com/courses/4156/files/744460?module_item_id=191059

[2] Mallawaarachichi V. 8 Useful Tree Data Structures Worth Knowing[Internet]. Towards Data Science[updated date: 2020-05-11; cited date: 2021-12-16].

Available from: <https://towardsdatascience.com/8-useful-tree-data-structures-worth-knowing-8532c7231e8c>

[3] Weiss, Mark. A. (2012), Data structures and algorithm analysis in Java. 3rd edition Harlow, Essex : Pearson. p 84.

The code(also mentioned in each java file):

Task1:

1. Algorithm 1: HackerRank. Data Structures: Heaps[Internet]. YouTube[updated 2016-09-27; cited date: 2021-12-10]. Available from: <https://www.youtube.com/watch?v=t0Cq6tVNRBA>
2. Algorithm 2: Edpresso Team. How to build a heap from an array[Internet]. Educative[cited date: 2021-12-10]. Available from: <https://www.educative.io/edpresso/how-to-build-a-heap-from-an-array>
3. Insertion&Deletion: GeeksforGeeks. Insertion and Deletion in Heaps[Internet]. GeeksforGeeks[updated: 2021-11-03; cited date: 2021-12-11]. Available from: <https://www.geeksforgeeks.org/insertion-and-deletion-in-heaps/>
4. Traversals: GeeksforGeeks. Tree Traversals (Inorder, Preorder, Postorder)[Internet]. GeeksforGeeks[updated: 2021-11-30; cited date: 2021-12-11]. Available from: <https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/>

Task3:

5. GeeksforGeeks. Search a Word in a 2D Grid of characters[Internet]. GeeksforGeeks[updated: 2021-07-22; cited date 2021-12-11]. Available from: <https://www.geeksforgeeks.org/search-a-word-in-a-2d-grid-of-characters/>

Task4:

6. AVLTree: GeeksforGeeks. AVL Tree Set 2(Deletion)[Internet]. GeeksforGeeks[updated: 2021-09-13; cited date 2021-12-11]. Available from: <https://www.geeksforgeeks.org/avl-tree-set-2-deletion/>
7. BinaryHeap: references 2. and 3.
8. BinarySearchTree: GeeksforGeeks. Binary Search Tree Set 2 (Delete)[Internet]. GeeksforGeeks[updated: 2021-08-25; cited date 2021-12-12]. Available from: <https://www.geeksforgeeks.org/binary-search-tree-set-2-delete/>
9. Red-Black tree: Programiz. Deletion from a Red-Black Tree[Internet]. Programiz.[cited date: 2021-12-13]. Available from: <https://www.programiz.com/dsa/deletion-from-a-red-black-tree>

Appendix A

insert "1"						
	10	100	1000	10000	100000	1000000
Algorithm 1	7740	1260	520	540	3840	620
Algorithm 2	680	640	1000	860	2160	4500
insert "1M"						
	10	100	1000	10000	100000	1000000
Algorithm 1	5740	820	460	760	7180	380
Algorithm 2	760	700	1140	1500	2620	5640
deleteMin						
	10	100	1000	10000	100000	1000000
Algorithm 1	2200	2760	5460	9800	18400	18000
Algorithm 2	1260	1080	1380	1480	3420	4860