DOA Week 7 - 8 Exercises

Heaps and Binary Search Trees

Software Engineering
Aarhus University
31. October 2023

Studie Nr
Simon Dybdahl Damulog Andersen: 202204902
Max Glintborg Dall: 202208488
Emil Lydersen: 202207059

Contents

Exercise 1

Using pen and paper, illustrate how a min-heap is built from the array:

[59, 44, 79, 17, 54, 32, 31, 12, 7, 4, 1]

using heapify. You must illustrate all steps of the heapify process. Then illustrate first what happens if remove is called on the resulting heap and then what happens if insert(2) is called on the heap (before the remove).

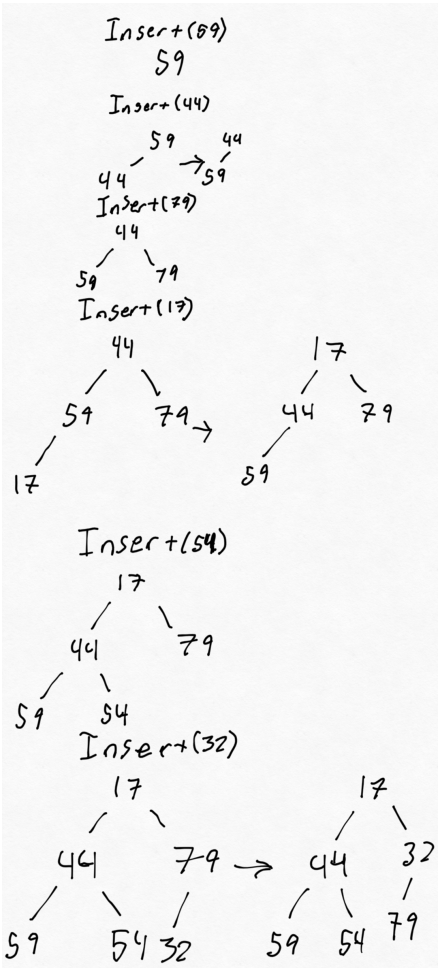

Figure 0.1
Heapify 1. part.

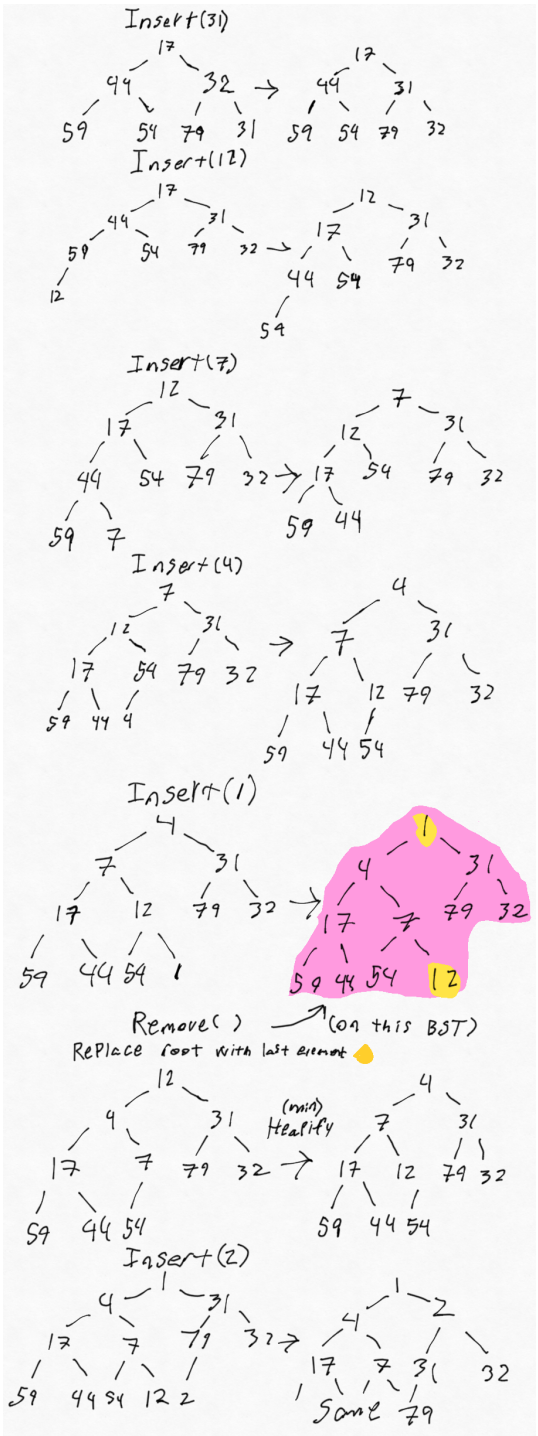

Figure 0.2
Heapify 2. part.

Here is an alternative method, that can be used to show the heapify process. In this version, the whole binary tree is written in the array order, after which the min-heap process begins using an indexing method. This method is used recursively until the BST is in order. After this the same remove and insert methods could have been applied aswell.

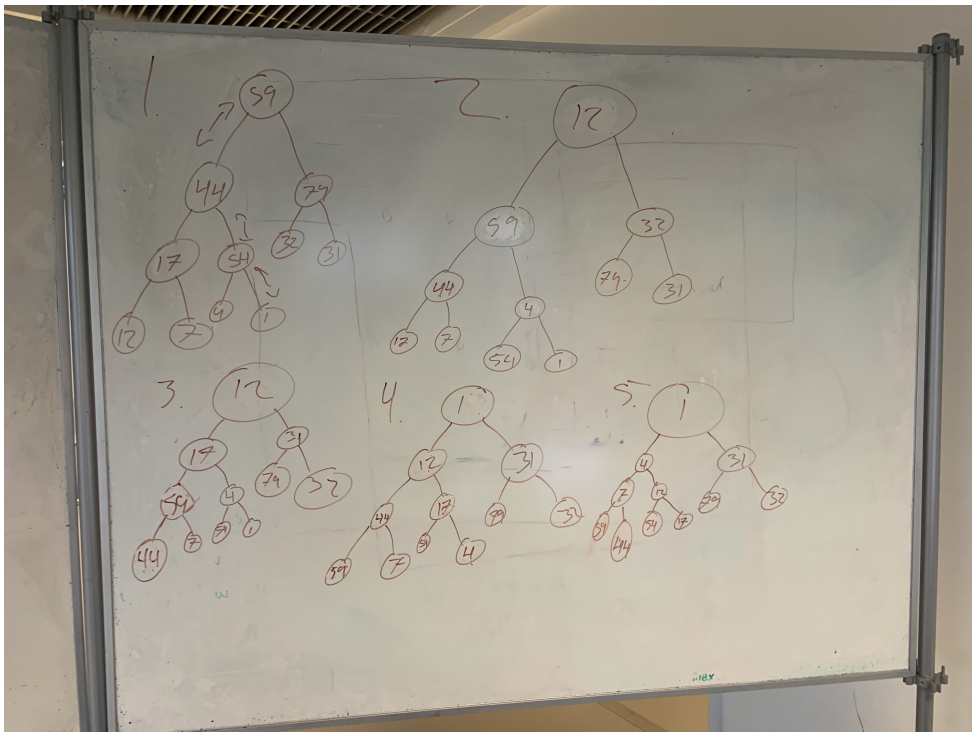

Figure 0.3
Alternative Heapify.

Exercise 2

Modify the heap implementation we saw on class to implement a *max-heap* in which the maximum element is stored in the root. Implement the `priority queue` interface discussed in class using the max-heap, and provide testing code.

Following code is the implementation, made from above assignment's specification:

```
1  /**
2   * Remove the maximum item.
3   * Throws UnderflowException if empty.
4   */
5  template <typename Comparable>
6  void BinaryHeap<Comparable>::deleteMax()
7  {
8      if (isEmpty())
9          throw underflow_error("heap is empty.");
10     int maxIndex = 1;
11     for (int i = 2; i <= currentSize; ++i)
12     {
13         if (array[i] > array[maxIndex])
14             maxIndex = i;
15     }
16     array[maxIndex] = std::move(array[currentSize--]);
17     maxHeapify(maxIndex);
18 }
19
20 /**
21  * Remove the maximum item and place it in maxItem.
22  * Throws Underflow if empty.
23  */
24 template <typename Comparable>
25 void BinaryHeap<Comparable>::deleteMax(Comparable &maxItem)
26 {
27     if (isEmpty())
28         throw underflow_error("heap is empty.");
29     int maxIndex = 1;
30     for (int i = 2; i <= currentSize; ++i)
31     {
32         if (array[i] > array[maxIndex])
33             maxIndex = i;
34     }
35     maxItem = std::move(array[maxIndex]);
36     array[maxIndex] = std::move(array[currentSize--]);
37     maxHeapify(maxIndex);
38 }
39
40 template <typename Comparable>
41 void BinaryHeap<Comparable>::maxHeapify(int node)
42 {
43     int child;
44     Comparable tmp = std::move(array[node]);
45
46     for (; node * 2 <= currentSize; node = child) // percolate down
47     {
48         child = node * 2;
49         if (child != currentSize && array[child + 1] > array[child])
50             ++child;
51         if (array[child] > tmp)
52             array[node] = std::move(array[child]);
53         else
54             break;
55     }
```

```
56      array[node] = std::move(tmp);
57  }
```

The code above is some of the more crucial parts of the implementation. For this reason the rest of the code implementation can be found in the source code.

In the code, deleteMax (two different ones) and maxHeapify has been added. These work essentially the same way as the original deleteMin and minHeapify functions, but have been altered so the maximum element is being found instead of the minimum. Based on the code above, an implementation of priority queue has been made:

```
1   #include "binary_heap.h"
2
3   template <typename Comparable>
4   class PriorityQueue
5   {
6   public:
7       BinaryHeap<Comparable> bh;
8
9       void pop()
10      {
11          bh.deleteMax();
12      }
13
14      void push(const Comparable &x)
15      {
16          bh.insert(x);
17      }
18
19      const Comparable &top() const
20      {
21          return bh.findMax();
22      }
23
24      bool empty()
25      {
26          return bh.isEmpty();
27      }
28
29      void clear()
30      {
31          bh = BinaryHeap<Comparable>();
32          while (!bh.isEmpty())
33          {
34              bh.deleteMax();
35          }
36      }
37  };
```

In this implementation the functions from $binary - heap.h$ has been utilized as-well as the max-heap functions. When running this code, the following is displayed:

```
maxPQ:
5
4
3
PS C:\Users\emilb\DOA-Exercises>
```

Figure 0.4
Terminal screenshot

Exercise 3

What is the running time of *heapsort* on an array A of length N that is already sorted in increasing order? What about decreasing order?

The different running times can be found using the Chrono library to check the running time after the sorted array is created, but before the sort function begins.

Following code is the implementation, made from above assignment's specification:

```
1  using namespace std;
2
3  #include <algorithm>
4  #include <iostream>
5  #include <vector>
6  #include "heap_sort.h"
7  #include <chrono> // Include the chrono library for timing
8
9  void checkSort(const vector<int>& a) {
10     for (int i = 0; i < a.size() - 1; ++i) {
11         if (a[i] >= a[i + 1]) {
12             cout << "Error at " << i << endl;
13         }
14     }
15     cout << "Finished checksort" << endl;
16 }
17
18 /**
19  * @brief This program tests the heapSort function on an array of size NUM_ITEMS.
20  *        The array is initialized with values in increasing order.
21  *        The function is called twice and the time taken for each run is printed.
22  *
23  * @return int
24  */
25 int main() {
26     const int NUM_ITEMS = 1000;
27
28    /* vector<int> a(NUM_ITEMS); // for initializing in decreasing order
29     for (int i = 0; i < a.size(); ++i) {
30         a[i] = NUM_ITEMS - i; //
31     } */
32
33
34     vector<int> a(NUM_ITEMS); // for initializing in increasing order
35     for (int i = 0; i < a.size(); ++i) {
36         a[i] = i;
37     }
38
39     for (int runs = 0; runs < 2; ++runs) {
40
41         auto start_time = chrono::high_resolution_clock::now(); // Record start time
42
43         // No need to call permute(a) since the array is already in increasing order
44         heapSort(a);
45        //checkSort(a);
46
47         auto end_time = chrono::high_resolution_clock::now(); // Record end time
48         auto duration = chrono::duration_cast<chrono::microseconds>(end_time - start_time); // Calculate
               duration in microseconds
49
50         cout << "Time taken for run " << runs << " with N = " << NUM_ITEMS << ":" << endl <<
               duration.count() << " microseconds" << endl;
```

```
51      }
52
53      return 0;
54  }
```

The running times are output like so:

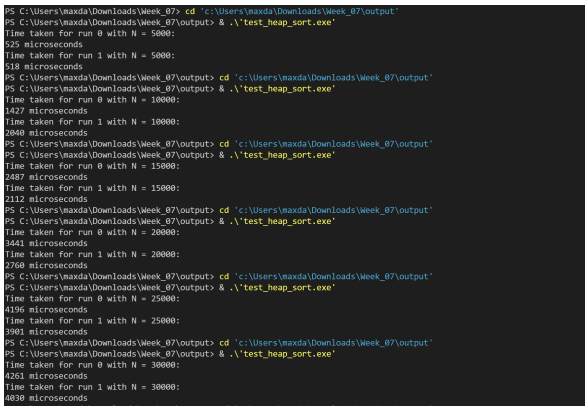

Figure 0.5
Increasing order output.

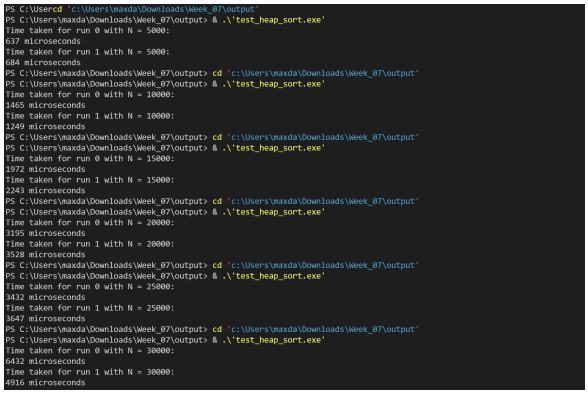

Figure 0.6
Decreasing order output.

From the outputs its possible to see that there are slightly different running times depending on if the way the array is initialized. to further illustrate. the times have been plotted through the use of Excel

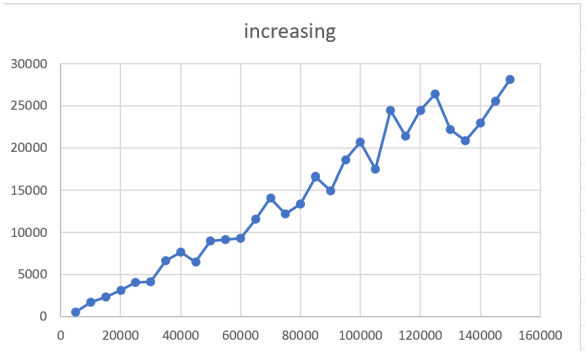

Figure 0.7
Increasing order graph.

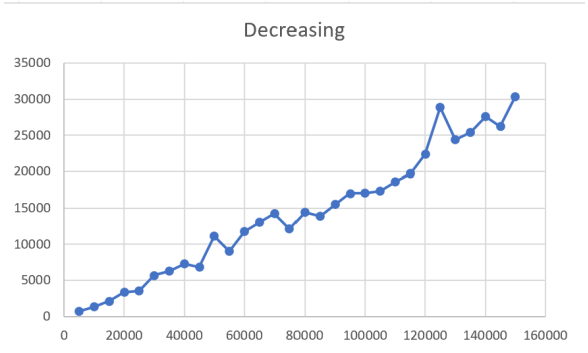

Figure 0.8
Decreasing order graph.

What is able to be gathered and concluded upon seeing comparing the 2 different initializations. is that even though the array was already sorted in increasing order. it took almost just as long time for it to run, compared to when the array is sorted in decreasing order (as shown from the tendency line). this is because of how Heapsort works. Heapsort's time complexity remains O(N log N) because the algorithm relies on the heap structure and the total number of comparisons and swaps required to sort the array will be the same, even though the array starts sorted in increasing order.

Exercise 4

Using pen and paper, illustrate how an AVL tree is build with the following elements

$$[10, 20, 15, 25, 30, 16, 18, 19]$$

(i.e.10 is inserted first, then 20 ...). You must illustrate all rotations etc. Then illustrate what happens when 30 is deleted and then 18

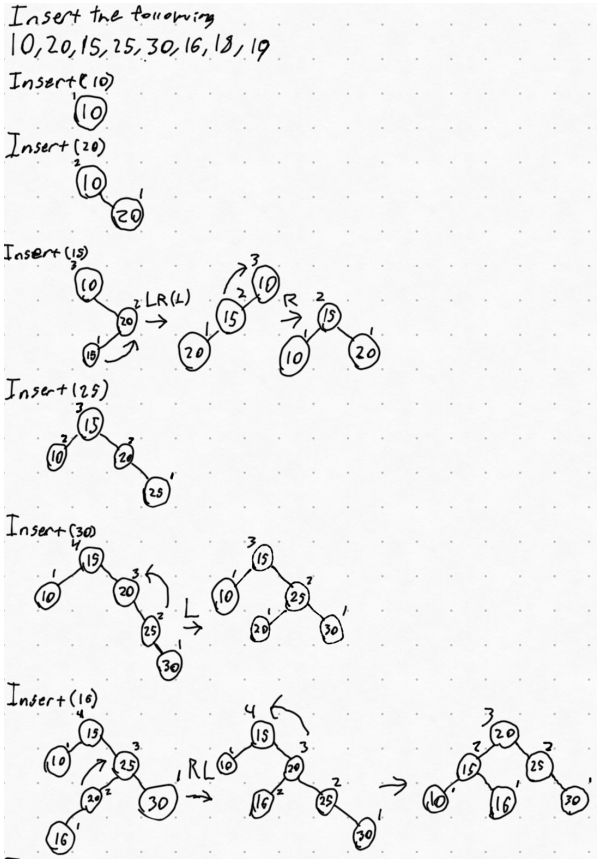

Figure 0.9
AVL tree part 1.

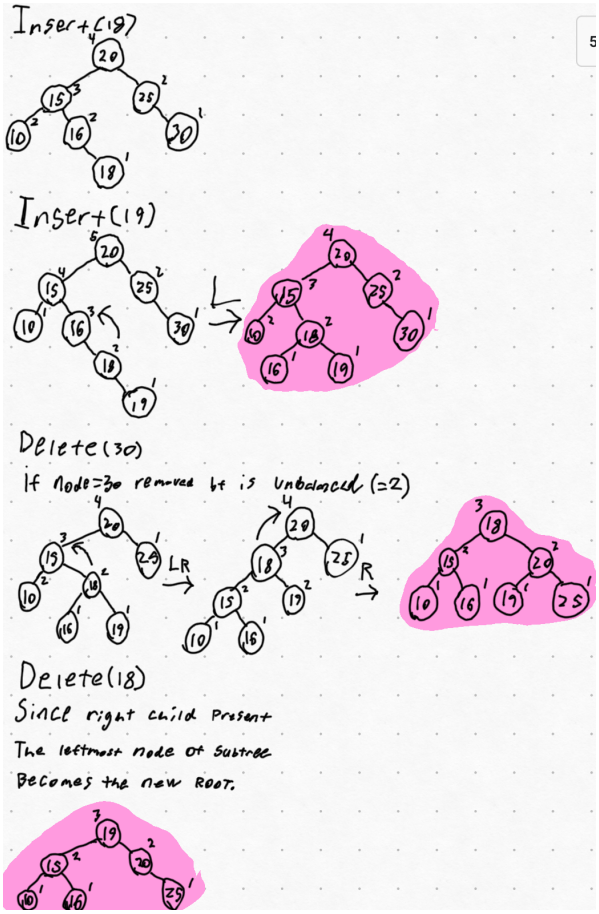

Figure 0.10
AVL tree part 2.

Exercise 5

A node in a binary tree is an only-child if it has a parent node but no sibling node (Note: The root does **not** qualify as an only child). The *loneliness-ratio* (LR) of a given binary tree T is defined as the following ratio:

$$\text{LR(T)} = \frac{\text{The number of nodes in T that are only children}}{\text{The number of nodes in T}}$$

(a) Argue that for any nonempty AVL tree T, we have $LR(T) \leq \frac{1}{2}$.
(b) Is it true for any binary tree T that if $LR(T) \leq \frac{1}{2}$, then $height(T) = \log(n)$

(a) In an AVL tree T, we have $LR(T) \leq \frac{1}{2}$.
This is because an AVL tree is able to rebalance itself - meaning that it will always have a balance of -1, 0 or 1. If it goes past this limit the AVL tree will reconfigure itself, to make sure it stays balanced. A nonempty tree is not empty as it contains one or more nodes. Therefore in the case of loneliness ratio(LR) it tells us how many tree nodes are alone. It measures how many nodes in a tree that has no siblings, and it reflects the distribution of nodes to their respective relationships. This means that it is never possible for parts of the AVL binary tree to only consist of lonely children, as the balance factor will ensure that all nodes will have always have siblings. Therefore for any nonempty AVL tree T the LR will always be less or equal to 1/2, which essentially means that most nodes/children has a sibling.

(b) Is it true for any binary tree T that has $LR(T) \leq \frac{1}{2}$, then it's $height(T) = \log(n)$.
This isn't necessarily true, as the height of a binary tree is often related to how tall the binary tree is. In a balanced binary tree the Loneliness Ratio can be small as most nodes have siblings, which in turn means that the height of the tree is log(n). In most cases this is not true as if the tree is unbalanced then the LR can still be less or equal to 1/2, whilst the height of the tree is not log(n). This could for example be a tree where one of the branches are well developed, and the other one is shallow. This would in turn mean that the LR would still be low, but the height of the tree would not be log(n).

It generally goes to say that the height of a binary and its LR can be related, but its important to remember that they are not equivalent.

Exercise 6

The Set is an ordered container that does not allow duplicates. Write an implementation of the `Set` class using the `BinarySearchTree` implementation we saw in class as the internal data structure. Add to each node a link to the parent node to make traversal easier. The supported operations are:

```
iterator insert(const Object& x);
iterator find(const Object& x )const;
iterator erase(iterator& itr);
```

The `insert` function returns an iterator to that either represents the newly inserted item or the existing duplicate item that is already stored in the container. For searching, the `find` routine returns an iterator representing the location of the item 1 (or the endmarker if the search fails). The `erase` function removes the object at the position given by the iterator, returns an iterator representing the element that followed `itr` immediately prior to the call to erase, and invalidates `itr`.

Following is the implementation of the Set class:

```
1  #include <vector>
2  #include "binary_search_tree.h"
3
4  using namespace std;
5
6  template <typename Comparable>
7  class Set
8  {
9  private:
10         size_t theSize;
11         BinarySearchTree<Comparable> tree;
12
13 public:
14         Set() : theSize(0) {}
15
16         ~Set() { clear(); }
17
18         Set(const Set &s) : theSize(s.theSize), tree()
19         {
20                 // Create an iterator to traverse the source set
21                 iterator it = s.begin();
22                 iterator end = s.end();
23
24                 // Iterate over the source set and insert its elements into the new set
25                 while (it != end)
26                 {
27                         insert(*it);
28                         ++it;
29                 }
30         }
31
32         void clear()
33         {
34                 tree.makeEmpty();
35                 theSize = 0;
36         }
37
38         size_t size() const
39         {
40                 return theSize;
```

```
41          }
42
43          bool empty() const
44          {
45                  if (theSize == 0)
46                          return true;
47                  else
48                          return false;
49          }
50
51          void push(const Comparable &t)
52          {
53                  insert(t);
54          }
55
56          void print() { tree.printTree(); } // used for debugging
57
58          friend class BinarySearchTree<Comparable>;
59          typedef typename BinarySearchTree<Comparable>::iterator iterator;
60
61          iterator begin() const
62          {
63                  return tree.findMin();
64          }
65
66          iterator end() const
67          {
68                  return iterator(nullptr);
69          }
70
71          iterator insert(const Comparable &t)
72          {
73                  if (tree.contains(t) == false)
74                  {
75                          ++theSize;
76                          tree.insert(t);
77                  }
78                  return iterator(tree.find(t));
79          }
80
81          iterator find(const Comparable &t)
82          {
83                  return iterator(tree.find(t));
84          }
85          iterator erase(iterator &itr)
86          {
87                  tree.remove(*itr); // Remove the element using its stored value
88                  --theSize;              // Decrease the size of the set
89                  return itr;             // Returns uninitialized iterator, they didint specify
90          }
91  };
```

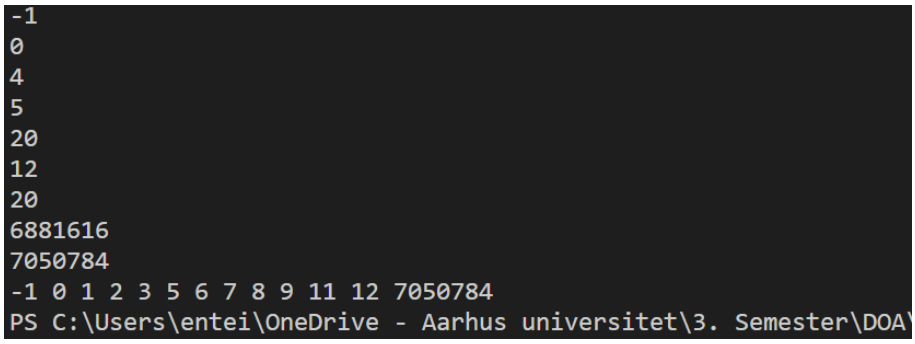

Figure 0.11
Terminal Output

For this exercise many iterations were made. In the end the output managed to get 90% of the way, but the program still throws a segmentation fault when exiting. Some uninitialized memory is also shown, an iteration of the iterator could have been made, such that the next element would have been shown instead, but since the assignment didn't specify the desired print, it was left at this. The specified places in the BST implementation was also implemented and can be seen in the source code.

In the end it was quite the challenge to implement the desired functionality from the BST and Set, but since this was our first experience with using iterators in such a way, and didn't quite get there fully. A lot of knowledge was gained from this exercise.

Exercise 7

Design and implement a linear-time algorithm that verifies that the height information in an AVL tree is correct i.e. the balance property is in order for all nodes.

In this exercise a linear-time algorithm has been designed and implemented:

```
1
2  public:
3  bool ltAlgorithm() const // linear-time algorithm
4          {
5                  return ltAlgorithm(root);
6          }
7
8  private:
9  bool ltAlgorithm(AvlNode *node) const // linear-time algorithm
10         {
11                 if (node == nullptr) // empty tree
12                         return true;
13
14                 int leftHeight = height(node->left); // height of left subtree
15                 int rightHeight = height(node->right); // height of right subtree
16
17                 if (abs(leftHeight - rightHeight) > 1)
18                 {
19                         return false;
20                 }
21                 else if (!ltAlgorithm(node->left) || !ltAlgorithm(node->right))
22                 {
23                         return false;
24                 }
25                 else
26                 {
27                         return true; // if the tree is an AVL tree return true
28                 }
29         }
```

In the implementation above, some of the code for the ltAlgorithm (linear-time algorithm) can be seen. Looking at the code, it can be seen that on line 21 and 22 left and right subtree heights are defined, followed up by the absolute value of the difference between the two subtree heights. Here the logic for the AVL tree is also defined. If the difference between the two subtrees are more than 1, the AVL tree will return false, as it is not possible for an AVL tree to have a height difference larger than 1. An AVL tree can only have the heights - 0,1 and -1 as a difference.

If the left or right subtree is not an AVL tree, it will also return false, otherwise it will return true, meaning the subtree is true and that the linear-time algorithm works. When running the test code the following statement will be printed in the terminal:

```
Checking... (no more output means success)
The linear-time algorithm succeded and it is an AVL tree.
End of test...
```

Figure 0.12
Terminal output

The rest of the code can be found in the source code.

Appendix

Source code in ”Appendix.zip”

Online:

Source code on GitHub - NGK