CITS2200 Project - Dictionary

Samuel Marsh
21324325

1 Design

1.1 Decision - Red-Black Tree

After examining the time complexity for various dictionary implementations, I decided a red-black tree was the most effective implementation for a general-purpose dictionary.

Implementation	**Search**	**Insert**	**Delete**
AVL tree	O(log n)	O(log n)	O(log n)
Red-black tree	O(log n)	O(log n)	O(log n)
B-Tree	O(log n)	O(log n)	O(log n)
Splay tree	(a) O(log n)	(a) O(log n)	(a) O(log n)
Skip list	O(n)	O(n)	O(n)
Trie	O(m)	O(m)	O(m)
Stratified tree	O(log log N)	O(log log N)	O(log log N)

(n = number of elements in data structure, m = length of string, N = size of universe, (a) = amortised)

Table 1: Considered data structures along with their worst-case time complexities.

When deciding on the optimal data structure, I first ruled out splay tree and skip list, since the skip list worst case time is above the required O(log n) and there are other data structures faster than the splay tree where amortisation is not required. I then ruled out Trie, although it is extremely fast (independent of size of dictionary), since I aimed to implement a general-purpose dictionary and Trie only holds strings [5].

Initially I implemented basic functionality of a stratified tree [3], also known as a van Emde Boas tree. The implementation had extremely fast lookup (`contains`, `max`, `min` O(1), and `successor`, `predecessor`, `add`, `delete` O(log log N)). However, my primary aim for this project was a *general-purpose* and *unbounded* dictionary. van Emde Boas trees are extremely fast, however the domain is bounded and each element requires an associated integer key. In the end I decided that van Emde Boas trees are too restrictive, and also more suited for a dictionary interface defined using keys and values (like `java.util.Map.put(K key, V value)`) and thus a more general purpose and unbounded but less efficient O(log n) solution would be more appropriate for this situation.

A red-black tree and a B-Tree seem to be very similar in efficiency, especially since a red-black is simply a type of B-Tree, as implied by the red-black tree's original name: a symmetric binary B-Tree [1]. It was challenging to find a significant advantage of red-black trees over B-Trees, however I chose to rule out B-Trees since they may waste space if a node is not full, whereas red-black trees do not have this disadvantage.

Finally, I chose a red-black tree over an AVL tree after considering the usage of the dictionary described in the project: keeping track of users logged in to a web-service. I assume that in most cases, more users log on and off than search for each other, so *insertions and deletions* are more common than *lookups*. Since red-black trees have generally faster inserts and deletions [2] - whereas AVL trees are faster at searches with slower modifications - a red-black tree would be the most appropriate choice for this scenario.

1.2 Implementation

The majority of my implementation is adapted from the pseudocode in the book 'Introduction to Algorithms: Third Edition' by T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. However, due to the specified requirements of the dictionary, my adaptation of the red-black tree has ended up with various modifications that differ significantly from the CLRS description. In this section, I detail the implementation of the `RedBlackTree` class.

1.2.1 Searching

I considered various options for the `contains()` method. My final implementation runs in logarithmic time by using the typical binary search tree search (traversing down the tree). However, I also considered the option of having a separate internal data structure such as a hash table. A hash table would provide *much* faster lookups on average, but unfortunately still has a O(n) worst case. The project specifications require O(log n) worst case, and storing the data in a hash table as well would also essentially double the space requirement. In the end, I decided a basic tree search would suffice for this method.

1.2.2 Successors/predecessors

This part of the dictionary took several refactorings and was the most challenging part of the project to implement. At first I developed the methods to first locate the node with the given value, and then find the successor of that node using well-known binary search tree methods. Therefore initially, `hasPredecessor(Object)` and `hasSuccessor(Object)` would take logarithmic time, as they would first check whether the element was contained in the dictionary (O(log n)) and then check whether it was within the bounds of the dictionary (not $\leq$ min or $\geq$ max respectively). However after implementation I realised that it was not required that the argument be contained in the dictionary. Therefore `hasPredecessor(Object)` and `hasSuccessor(Object)` could be reduced to constant time - since the argument only has to be less than the maximum element to have a successor, and vice versa for predecessor. My final design for `successor(Object)` uses a method `above(Node)`, which finds the least node strictly greater than the argument (and `below(Node)` is symmetric for `predecessor(Object)`).

1.2.3 Minimum/maximum

Originally I implemented `max()` and `min()` to simply call the helper methods `minimum(Node)` and `maximum(Node)` with the root as the argument. This takes logarithmic time. However, I later modified the class to maintain two extra fields `min` and `max`. Upon insertion, the current values of `min` and `max` are compared with the newly inserted element - if the new node is greater than the max or less than the min, the references are updated. This adds a worst-case cost of 2 extra comparisons to the `add(Object)` method and reduces `min()` and `max()` to O(1).

Upon deletion, if the item being deleted is the minimum or the maximum, then the min/max is found by searching the tree again from the root. This is a minor cost when compared to the performance gains from constant time access to the extremum of the dictionary (and doesn't make the time complexity of `delete(Object)` any worse).

1.2.4 Insertion/deletion

The `add(Object)` and `delete(Object)` methods are where my implementation has strong similarities to the red-black tree detailed in CLRS.

The `add(Object)` method creates a new node and calls the `insert(Node)` method, which simply finds the appropriate location for the new node, inserts it there, colours it red, and then restores the red-black tree properties using `fixInsert(Node)`.

The `delete(Object)` method first locates the node to delete using a basic tree search, and replaces it with one of its children if necessary. Then it calls `fixDelete(Node)` to restore any red-black tree properties that may have been violated.

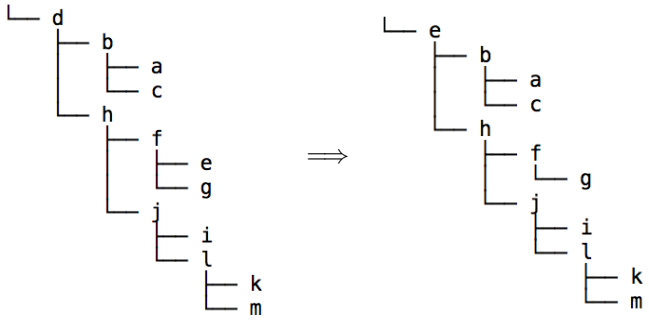

Figure 1: Inserting characters a to m into the red-black tree, and then deleting the root d. Note that in both cases, the height $h = 5$ and for a dictionary with 12-13 nodes the height bound is $h \leq 2\log_2(12+1) \approx 7$ which is satisfied by both structures.

Since detailing how red-black tree insertion and deletion work would take more space than I have room for in this report, I will instead detail some implementation decisions used in regards to `add` and `delete` (the add- and delete-related code is commented in the `TreeIterator` class).

I chose to use a sentinel node `nil` for inserting and deleting, rather than null pointers. This is suggested by CLRS, and it has the advantage of not having to consider exceptional cases as frequently - instead of having to avoid dealing with null as a separate case, the `nil` node can be treated as any other node and have its `parent`, `left`, and `right` fields take on arbitrary values.

I felt one disadvantage to my insertion and deletion related code is that there is a lot of symmetrical/repeated code, that deals with the same case, but having every reference to a 'left' child replaced with a 'right' child, and vice versa. In development I was unable to find a clean way that would enable me to abstract this to deal with both left and right cases.

1.2.5 Iteration

In implementing the above methods, I tried to 'modularise' my code as much as possible - that is, for example, instead of putting the deletion code inside the public `delete(Object)` method, I extracted it into a separate `delete(Node)` method. This gave me the advantage of being able to easily implement the iterator in a short amount of time, since I could rely on `successor(Node)` and `delete(Node)` to perform all operations needed by the `TreeIterator` class.

The iterator constructor takes in a start node, stores a reference to that start node in an instance variable, and then on each call to `next()`, 'increments' the node by replacing it with its successor (using `successor(Node)`) and returns the previous value.

To implement `Iterator#delete()`, I also stored a reference to the last node returned by `next()`. On deletion, I simply call the `delete(Node)` method in the enclosing `RedBlackTree` class, and then set `last` to the `nil` sentinel. Setting this field to `nil` also works as a check to ensure that the iterator state is valid - if `last` is `nil`, then we know that either `next()` hasn't been called yet or `last` has already been deleted. In this case, I throw an `IllegalStateException`.

Ensuring the iterator is fail-fast is also implemented in a fairly simple way - in the `RedBlackTree` class, a field `operations` keeps track of the total number of successful insertions and deletions on the dictionary. In the constructor of `TreeIterator`, this number is recorded. Then, each time a method is called on the iterator class, the two numbers are compared - if the number of operations on the tree is not the same as when the iterator was constructed, then a `ConcurrentModificationException` is immediately thrown.

1.2.6 Logging

Each action made on the dictionary will cause a new line to be appended to the log string. `getLogString()` will return this string and clear the log.

Each line of the log string reports the name of the method called, the value of the argument if applicable, and the number of comparisons used for that method.

Note that an assumption I have made in development is that the number of comparisons is the number of calls to `compareTo`, plus, in general, the number of significant colour comparisons (e.g. in `fixInsert(Node)` the number of times the parent node's colour is checked) and significant reference equality checks (e.g. moving down the tree in the `minimum(Node)` method.

2 Analysis

2.1 Analytical

This section will detail the time complexity of each method.

Red-black trees have five main properties:

1. A node is either black or red.
2. The root node is black.
3. Every leaf node is black.
4. A red node has black children.
5. Every path from a node to a leaf descendent contains the same number of black nodes.

RedBlackTree	
Method	**Time complexity**
`isEmpty()`	O(1)
`contains(Object)`	O(log n)
`hasPredecessor(Object)`	O(1)
`hasSuccessor(Object)`	O(1)
`predecessor(Object)`	O(log n)
`successor(Object)`	O(log n)
`min()`	O(1)
`max()`	O(1)
`add(Object)`	O(log n)
`delete(Object)`	O(log n)
`iterator()`	O(1)
`iterator(Object)`	O(log n)
`toString()`	O(n)

TreeIterator	
Method	**Time complexity**
`hasNext()`	O(1)
`next()`	O(log n)
`remove()`	O(log n)

Table 2: Summary of time complexity of each method.

Since the height of the tree is referenced in demonstrating the worst-case time complexity of almost every method in the class, I first prove the following limit on the height of a red-black tree [2]:

$$h \leq 2\log_2(n+1)$$

Proof. To begin, we prove that a subtree with root node n has a minimum of $2^{h_b(n)} - 1$ internal nodes (where $b_h(n)$, the 'black-height' of n, is the number of black nodes on any path from the node n to any leaf, not including n): we do this by induction. For the base case - if the black height of n is zero, then n must be a leaf - and so $2^0 - 1 = 0$ is the number of internal nodes. So it holds for $b_h(n) = 0$. Now we assume that a node n has a positive height and two children (i.e. n is an internal node). Then each child must have a height of $b_h(n)$ or $b_h(n) - 1$, depending on the colour of the child. We can then apply the inductive hypothesis to the children - then each child has $2^{b_h(n)-1} - 1$ internal nodes. Therefore we can simply sum the number of internal nodes of the two child subtrees to prove the claim - the subtree rooted at n will have at least $(2^{b_h(n)-1} - 1) + (2^{b_h(n)-1} - 1) + 1 = 2^{b_h(n)} - 1$ internal nodes.

With the above combined with red-black tree properties, we can complete the proof: if we have a tree of height h, then according to the fourth property of red-black trees, at least half of the nodes on any path from the root to a given leaf (not including the root) are black. So $b_h(r) \geq \frac{h}{2}$, where r is the root node of the tree. Then $n \geq 2^{\frac{h}{2}} - 1$, and $h \leq 2\log_2(n+1)$ follows. □

2.1.1 `isEmpty()`

This method performs a single null-check of the root node and is independent of the number of elements in the tree. Thus `isEmpty()` runs in constant time O(1).

2.1.2 `contains(Object)`

This method calls the helper method `locate(Node)` which is linear in the height of the tree. `locate(Node)` performs a basic iterative binary tree search, moving down the tree and comparing the node with the argument, then moving to the left or right child depending on the result of `compareTo`. A binary tree search is linear in the height of the tree (O(h)) and since the height of a red-black tree is guaranteed to be logarithmic, the time complexity of this method is thus O(log n).

2.1.3 `hasPredecessor(Object)` and `hasSuccessor(Object)`

These methods make a single comparison with the current minimum/maximum of the dictionary, to ensure that there is a smaller/larger element in the dictionary. Since the extremum of the dictionary can be accessed in constant time, this comparison is independent of the size of the dictionary, so these methods are O(1).

2.1.4 `predecessor(Object)` and `successor(Object)`

These methods call the internal methods `below(Node)` and `above(Node)` respectively: I will show that `below(Node)` runs in logarithmic time, and then `above(Node)` is symmetric to this. `below(Node)` starts at the root and then navigates down the tree in a similar fashion to `locate(Node)`, generally moving to the right child if the argument is larger than the current node, and moving left if it's less than or equal to the current node. Since it moves strictly down the tree, it can iterate at most h times -

therefore being logarithmic. Once it reaches some point in the tree - (where the right/left child is `nil`) then it will, in the worst case, need to return call the `predecessor(Node)` method and return the result. Since the `predecessor(Node)` method also runs in O(log n) time (it either returns the maximum node of the left subtree or moves up the tree until it finds a node which is a right child), then the total time is on the order of O(log n + log n) = O(log n).

2.1.5 `min()` and `max()`

These two methods run in constant time as fields referencing the current minimum and maximum are kept updated upon insertions and deletions to the dictionary. So `min()` and `max()` simply access these fields, making them O(1).

2.1.6 `add(Object)`

`add(Object)` first creates a new `Node` and calls the `insert(Node)` helper method, which is analogous to `RB-INSERT(T, z)` in CLRS. A summary of the non-trivial parts of the method follows: first, the place to insert the new node in the tree is located - this takes at O(h) time since it moves strictly down the tree each iteration, and does constant time work inside the loop. After this is done, the node is made red and then the tree properties are restored using `fixInsert(Node)`. `fixInsert(Node)` is analogous to `RB-INSERT-FIXUP(T, z)` in CLRS. The while loop in `fixInsert(Node)` only repeats if the node's uncle is red, and if this occurs we move two levels up (or possibly terminate the loop). Everything inside the while loop takes constant time (in particular note that `rotateLeft(Node)` and `rotateRight(Node)` do constant work - they have O(1) time complexity). Therefore `fixInsert(Node)` is O(h) $\implies$ the whole insertion procedure takes O(log n) time.

Note that if an item is successfully inserted into the dictionary, it is then compared with the minimum and maximum fields to check if we need to update them. This is a constant time procedure and doesn't affect the time complexity of `add(Object)`.

2.1.7 `delete(Object)`

`delete(Object)` was also implemented using CLRS as a foundation. Firstly note that the `transplant(Node)` method runs in constant time since it simply replaces one subtree as a child of its parent with another subtree by modifying references, and is independent of the size of the tree. With this, we can see that the internal method `delete(Node)`, at worst, takes O(log n) time when it has to find the minimum node of the right child's subtree. Then at the end of the method `fixDelete(Node)` is called, which also takes O(log n) time: the while loop in the method only repeats when node's sibling is black and both of the sibling's children are black - in which case each iteration we move up the tree by assigning `node = node.parent`. This can only happen at most O(h) times, so the overall complexity of `delete(Object)` is O(log n).

NB: if the element that was deleted was the minimum or maximum element in the dictionary, then `minimum(Node)`/`maximum(Node)` will be called with the root as the argument, in order to update the minimum/maximum references. This does not affect the time complexity, as O(2 log n) = O(log n).

2.1.8 `iterator()`

Firstly: the constructor for the internal class `TreeIterator` runs in O(1) time, since it is passed the start node as an argument and stores this reference in a field in preparation for the first call to `next()`. Since for this method we start at the minimum element in the dictionary and iterate through to the maximum, and we can access the minimum element in constant time and simply pass it as a parameter to the `TreeIterator` constructor, the whole method takes O(1) time.

2.1.9 `iterator(Object)`

This method, although similar in functionality to `iterator()` does *not* take constant time: it first needs to find the least node with key greater than the input argument. The method makes a call to the private method `ceiling(Node)`. The method `ceiling(Node)` is very similar in functionality to `above(Node)`, which is used in `successor(Object)` - `ceiling(Node)` just has a different edge case, and so it also runs in O(log n).

In addition to the time complexity analysis of returning a new iterator, I will also briefly detail the time complexity of the methods in the `TreeIterator` class: clearly `hasNext()` takes O(1) since it simply performs a 'nil-check'; `remove()` takes O(log n) since it calls `delete(Node)`, which is shown to be O(log n) above; `next()` also takes O(log n) since it calls the `successor(Node)` method, and as above `successor(Node)` runs in O(log n). Therefore iterating through the dictionary takes time O(n log n).

I expect there is a more efficient way to do iteration - e.g. using a stack and pushing the left subtree onto the stack/pushing right child and its left subtree onto the stack with each call to `next()`. Although I had this strategy working for iteration over the full tree, I could not find a way to successfully iterate in this fashion when given some start node that is not the minimum.

2.1.10 `getLogString()`

`getLogString()` retrieves the string value of the `StringBuilder` log field and then resets the log. This is independent of the number of elements in the dictionary, therefore running in O(1) time.

2.1.11 `toString()`

This method was adapted from a StackOverflow post [4]. The interface method `toString()` calls the `toString()` method of the root node, which is a recursive method that prints out the node itself and the string representation of its two children. This method will therefore consider each node in the dictionary only once, making the method O(n).

2.2 Empirical

This section gives experimental data on the efficiency of the implementation. The efficiency is quantified by the number of comparisons made when performing an operation in the `Dictionary` interface.

For demonstration of experimental correctness, refer to the JUnit tests which can be found in the `/test/java/DictionaryTest.java` file. JUnit 4.12 was used, and all tests passed at the time of project submission.

Experimental data was obtained by repeatedly running a given method and averaging the result. Each run, the elements are randomly shuffled before being inserted into the dictionary to allow for various internal structures. MATLAB R2014a was used for data visualisation in the figures for `add`, `delete` and `contains`.

For `add`, `delete` and `contains`, plot data was averaged over 100 trials. Each trial recorded the number of comparisons made on dictionaries of size ranging from 0 to 10,000.

Plots of the growth of the number of comparisons for adding, deleting and searching are below. Both linear-linear and log-linear graphs are provided to demonstrate the logarithmic growth of the functions.

For the other methods required by the `Dictionary` interface, the average number of comparisons made on a 10,000-element dictionary is given in the table below. These results were averaged over 100 trials. For all tests, a new dictionary was created each time and the 10,000 elements were inserted in random order. This allows for a range of different possible tree structures to even out any possible biases in any individual trial.

Method	**Comparisons**	
	Item in dictionary	**Item not in dictionary**
`isEmpty()`	0.0	
`contains(Object)`	12.8	13.2
`hasPredecessor(Object)`	1.0	
`hasSuccessor(Object)`	1.0	
`predecessor(Object)`	14.3	13.6
`successor(Object)`	14.1	14.0
`min()`	0.0	
`max()`	0.0	
`add(Object)`	12.8	15.3
`delete(Object)`	13.4	13.1
`iterator()`	0.0	
`iterator(Object)`	12.9	14.1
`toString()`	0.0	

Table 3: Mean number of comparisons made on a dictionary of 10,000 elements, averaged over 100 trials and split into two columns depending on whether the argument is in the dictionary or not.

Table 3 is consistent with the analysis section. The constant time methods take 0 or 1 comparisons. `contains(Object)` takes slightly longer if the element is not in the dictionary (since it has to search down the full height of the tree). `add(Object)` takes around 2-3 comparisons more if the item is not already in the dictionary, since it also needs to check if it is necessary to update the `min`/`max` fields. `delete(Object)` takes more comparisons if the element is not in the dictionary, for the same reason as `contains(Object)`.

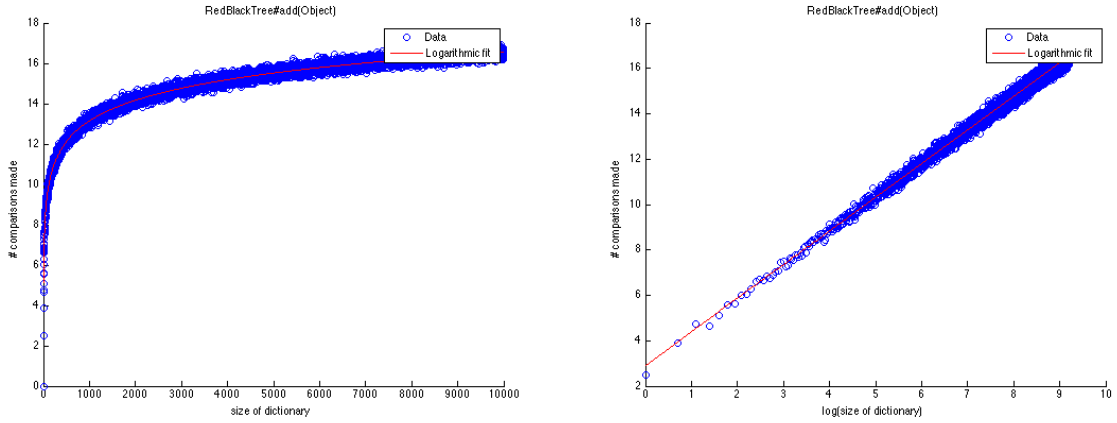

Figure 2: The number of comparisons made when adding a random distinct element to a dictionary of size n (linear and log-linear scale)

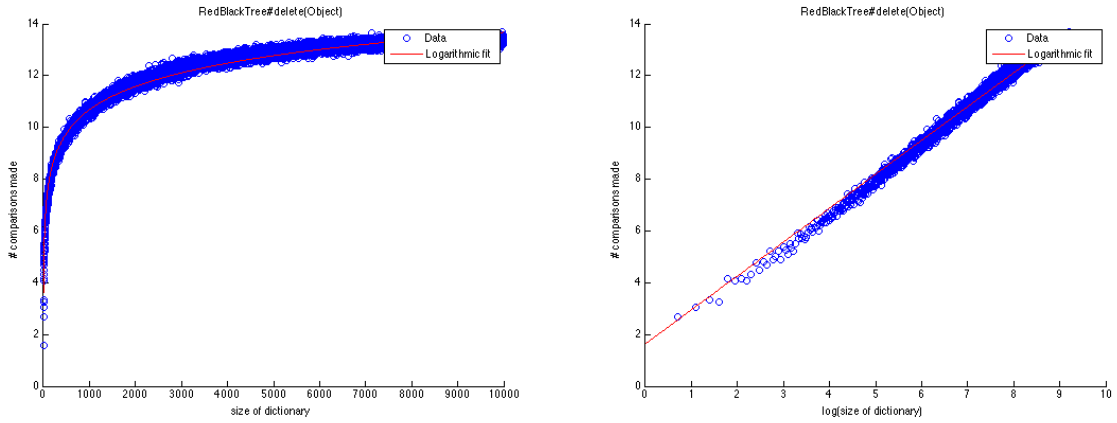

Figure 3: The number of comparisons made when deleting a random element from a dictionary of size n (linear and log-linear scale)

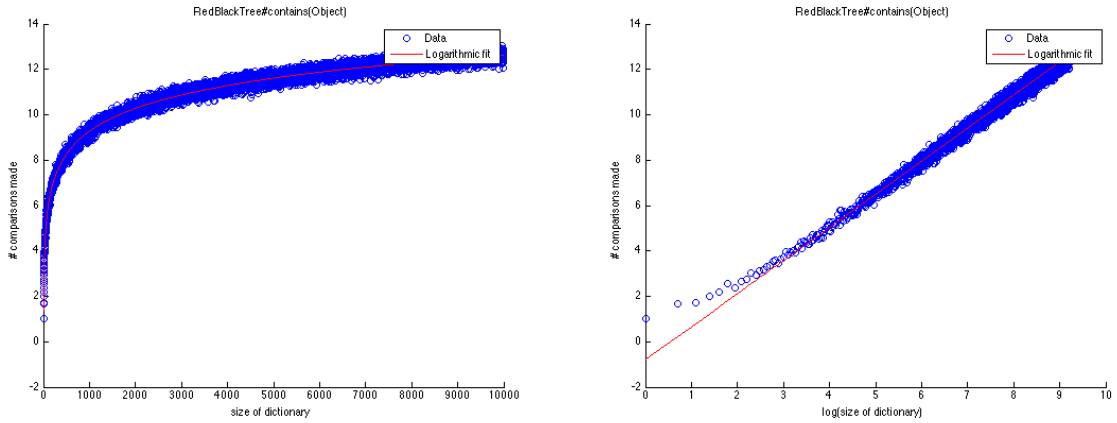

Figure 4: The number of comparisons made when searching for a random element in a dictionary of size n (linear and log-linear scale)

References

[1] Rudolf Bayer. "Symmetric binary B-Trees: Data structure and maintenance algorithms". English. In: *Acta Informatica* 1.4 (1972), pp. 290–306. ISSN: 0001-5903. DOI: 10.1007/BF00289509. URL: http://dx.doi.org/10.1007/BF00289509.

[2] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. The MIT Press, 2009. ISBN: 0262033844, 9780262033848.

[3] Kurt Mehlhorn and Stefan Näher. "Bounded ordered dictionaries in O(log log N) time and O(n) space". In: *Information Processing Letters* 35.4 (1990), pp. 183 –189. ISSN: 0020-0190. DOI: http://dx.doi.org/10.1016/0020-0190(90)90022-P. URL: http://www.sciencedirect.com/science/article/pii/002001909090022P.

[4] Vasya Novikov. *Print a [large] tree by lines.* Sept. 2014. URL: http://stackoverflow.com/questions/4965335/how-to-print-binary-tree-diagram/8948691#8948691.

[5] Dan E. Willard. "New trie data structures which support very fast search operations". In: *Journal of Computer and System Sciences* 28.3 (1984), pp. 379 –394. ISSN: 0022-0000. DOI: http://dx.doi.org/10.1016/0022-0000(84)90020-5. URL: http://www.sciencedirect.com/science/article/pii/0022000084900205.