Advanced Topics in Algorithms
Project I

Miguel Ramos
up201101579

Ricardo Pereira
up201604583

1 Introduction

For this project we were asked to implement a subset of the data structures that were taught during the first half of the Advanced Topics in Algorithms course.

We chose to implement the following data structures: Binary Search Tree, for comparison purposes; AVL Tree, because it never gets old; Splay Tree, the one with the cooler operation names; Treap, to play with randomness; and Skip List, to show that we can implement non-tree structures as well.

2 Data structures

In this section, we provide a summary explanation of each data structures.

2.1 Binary Search Trees (BST)

Binary Search Trees are trees for which all of its node respect the following properties:

- Every node is greater than every node in its left sub-tree;
- Every node is smaller than every node in its right sub-tree.

These properties allow us to search, insert and remove any node in a tree of height h in $O(h)$ time. If a tree with n nodes is perfectly balanced, then its height is $O(log(n))$. However, since its height highly depends on the order by which the nodes are inserted, we can have trees as tall as the number of nodes they contain, if the insertion happens in an orderly fashion. Self-balancing BSTs solve this problem by enforcing some additional properties that translate into keeping BSTs balanced (to some degree) after any operation.

2.2 AVL Trees

AVL Trees are self-balancing BSTs. Besides the basic properties of a BST, they guarantee the following:

- For every node, the height of its left and right sub-trees differ by at most one unit.

By enforcing this invariant, we are guaranteed that every AVL Tree of n nodes is of height $O(log(n))$. And, therefore, that we can search, insert and remove any node in $O(log(n))$ time.

When we insert or remove a node from a AVL tree, we may unavoidably break this invariant, so we need to have some mechanisms that allow us to re-balance it. To do this we apply rotations along the path were the affected node is or was. There are two identical unbalances - to the left or to the right - and each one has two distinct possible cases that are solved by applying at most two rotations. Since each rotation can be done in constant time by traversing at most a number of nodes that is equal to its height, then we conclude that enforcing the invariant does not change the overall complexity for each operation, leaving us with the aforementioned $O(log(n))$ time complexity for a AVL tree with n nodes.

2.3 Splay Trees

Splay Trees are self-adjusting BSTs. "Self-adjusting" is a different concept from "self-balancing". It is more general in the sense that it can be applied to any data structure that can rearrange itself after every operation in order to improve the efficiency of any future operation. In the case of Splay Trees, this is achieved through the following idea:

- Accessed nodes are moved to the root of the tree.

This is called "splaying" the node. It allows quick access to recently accessed nodes and guarantees logarithmic access time in an amortized sense.

When we search or insert a node, we splay the affected node. When we remove a node, we start by splaying it, and then we remove it and place the larges node of its left sub-tree in its place. The splay operation is based on rotation. Since rotations can be done in constant time and the number of rotations can be bound by the height the tree, we have that each splay operation takes $O(h)$ time. Once again, the height of the tree highly depends on the insertion order of its node. This means that is some cases the height of the tree is the same as the number of nodes in it, thus giving us a complexity of $O(n)$ time. The real advantage comes analysing the complexity of a sequence of operations. If we have a sequence of m operations, then every operation on a deeper side of the tree brings all the nodes from that side a little closer to the root, giving us a running time of $O(m\ log(n))$, where n is the maximum number of nodes in the tree at any time.

2.4 Treap

A Treap is a randomized alternative to BSTs, such as the previous ones. It is simultaneously a BST and a (Max-)Heap. This is achieved by having assigning to every node, both a search key and a priority that have the following properties:

- The inorder sequence of search keys is always sorted;
- The priority of every node is bigger than the priorities of its children.

After any sequence of insertions and deletions, these properties produce a BST that is very similar in shape to a truly random BST. This means that, with very high probability, the height of a Treap of n nodes is $O(log(n))$. Thus, giving us a search, insertion and deletion time complexity of $O(log(n))$.

Search is done the same way as for any BST. The differences are when inserting or deleting a node. When inserting a node, a random priority is generated for it. It then is inserted as usual for any BST. After that, the heap property is maintained by rotating the node into its father's position while its priority is greater than his. When deleting a node, either: just remove it, if it is a leaf node; remove it and replace it with its child, if it has only one child; or switch it with its immediate successor according to the sorted order, apply the necessary rotation to maintain the heap property, and continue applying this case until resulting into one of the other cases.

2.5 Skip Lists

Skip Lists are randomized data structures. They support the same operations with the same complexities as balanced BSTs, but have a very different underlying structure. It has a linked list-like structure that allows insertions, something that is not possible in arrays. Fast search is achieved by maintaining a linked hierarchy of sub-sequences. Each successive sub-sequence skips over fewer elements than the previous one, eventually reaching the desired node or discovering that it does not exist.

Searching starts in the sparsest sub-sequence. It progresses through it until either the node is found and we are done, or two consecutive nodes such that one is smaller and one is larger than the node we are searching for are found. In this case, we have to keep searching for the node in next sparsest sub-sequence. To do this, we follow the link from the smaller node to the next sub-sequence and continue from there on until we reach the last level. To insert a node, we proceed the same way as if we were searching for the node, but when we reach the last level and find the two consecutive nodes such that one is smaller and one is larger than the node we are inserting, we insert the node between the two, randomly decide on how many of

the previous levels the node will appear, and update all of the link accordingly. To delete a node, we simply search for the node, delete it from all the level in which it appears, and update all of the links accordingly.

This hierarchical organization of sub-sequences and random selection of the number of levels in which a particular node appears, allows us to search, insert and delete nodes in $O(log(n))$ time on a Skip List with n nodes.

3 Implementations

We decided to look at all of the different data structures as different ways of implementing a Set data structure (see Figure 3). To decide which operations we would want our data structures to support, we took inspiration in C++ STL's `Set` class. The only operation that has a different behaviour is `range_search`. We now provide a quick summary of each operation, separated into different groups according to their functionality.

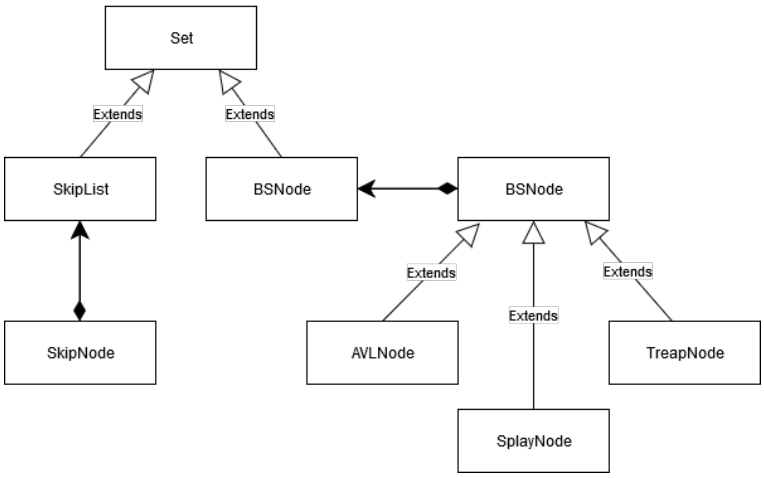

Figure 1: Code structure of our implementations

3.1 Capacity related operation:

- `empty`: Checks if the Set is empty and returns either true or false, accordingly.
- `size`: Returns the number of nodes contained in the Set.

3.2 Modifying operations

- `clear`: Removes all nodes from the Set.
- `insert`: Inserts a given node into the Set.
- `remove`: Removes a given node from the Set.

3.3 Search operations

- `count`: Searches for a node in the Set and returns either true or false, depending on whether the node exists or not in the set.
- `lower_bound`: Searches for the first node that is strictly greater than the provided node. Since the node may not exist, we decided to adapt this operation to return a pair consisting of a node and a boolean: if a node that satisfies the aforementioned restrictions exists, then it is returned along with a true value; if not, then a default node is returned along with a false value.
- `upper_bound`: This operation is very similar to the previous one, but it searches for the first node that is greater or equal to the provided node.
- `min`: Searches for the smallest node in the set and returns a pair with the same properties as for the previous two operations.
- `max`: This operation is very similar to the previous one, but it searches for the greatest node instead.

3.4 Type support

All the data structures were implemented as a template so that they can handle any type which have the operators `==` and `<` defined. In the delivered code, we also require the type to be printable with `std::cin`, but this property is not necessary for the set operations and could be removed if the print functions were taken out.

4 Tests & Results

In this section, we provide some of the results we obtained by analysing the time complexities of each implementation. The results contemplate the time taken by the operations associated with each data structures. Any time spent with I/O operation is disregarded. We will not analyse space complexity since it is trivial to show that the data structures have linear space complexity.

4.1 Integrity Check

In order to assert the correctness of our implementations, we created 3 test files. *test1* was written manually to check the behaviour of the methods in different scenarios (*e.g.* searching for the root, searching for a non-existent node and searching in an empty set). *test2* and *test3* were generated randomly to simulate "real" usage of the data structures, the difference between them being the type of data (`int` and `string`). To generate the answers for this two tests and to compare the efficiency, we wrote a wrapper for the `std::set` which we called STL so that it had the same interface as our definition of Set and used it as ground truth for validity.

4.2 Empirical complexity of basic operations

To test the runtime complexity of our implementation, we generated random files with sizes of different orders of magnitude (from 10^5 to 10^7) to test each operation (namely insertion, removal and search) individually. All of the provided plots are in *log-log* scale and include every chosen data structure, as well as the STL wrapper and the expected time. The expected time assumes a processing speed of 10^7 operations per second.

To verify the complexity of the insertion operation, the test files include a series of insertion operations with integer values in the range of $[1, 10^6]$ picked at random and allowing repetitions.

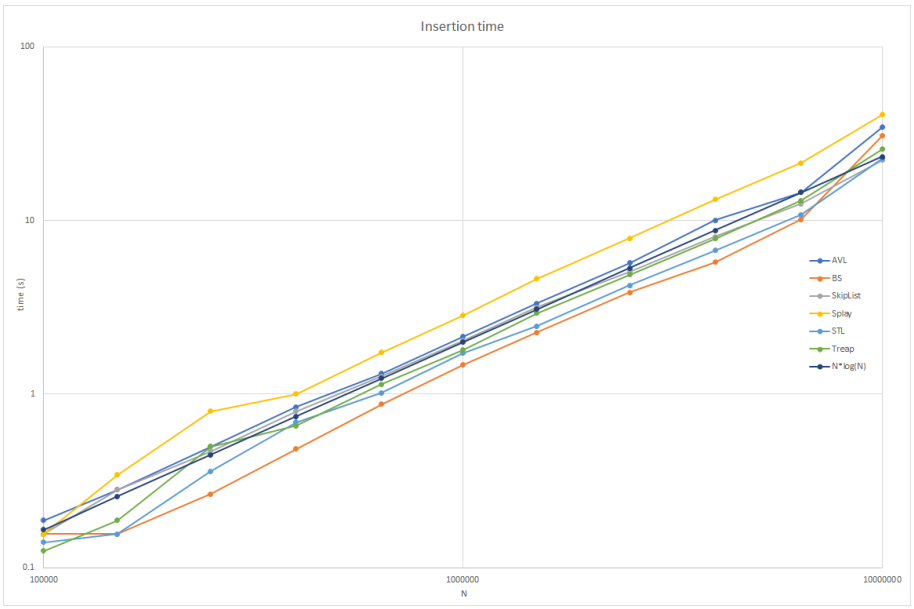

Figure 2: Running time for insertion

In Figure 2 we show the results obtained. The runtime of every data structure for every input is very close too the expected value. However, the Splay Tree seems to be consistently marginally slower than the rest, which is expected given that the data follows a uniform distribution over the domain. Besides, the Splay Tree is expected to be a little heavier than the rest.

Next up, the removal operation. In this case, and given that we need to have something in the Set to make a remove, the run file was changed to include a filling section (in which we insert every node) before starting to process the input. After that, the input files have the same structure as the ones for insertion.

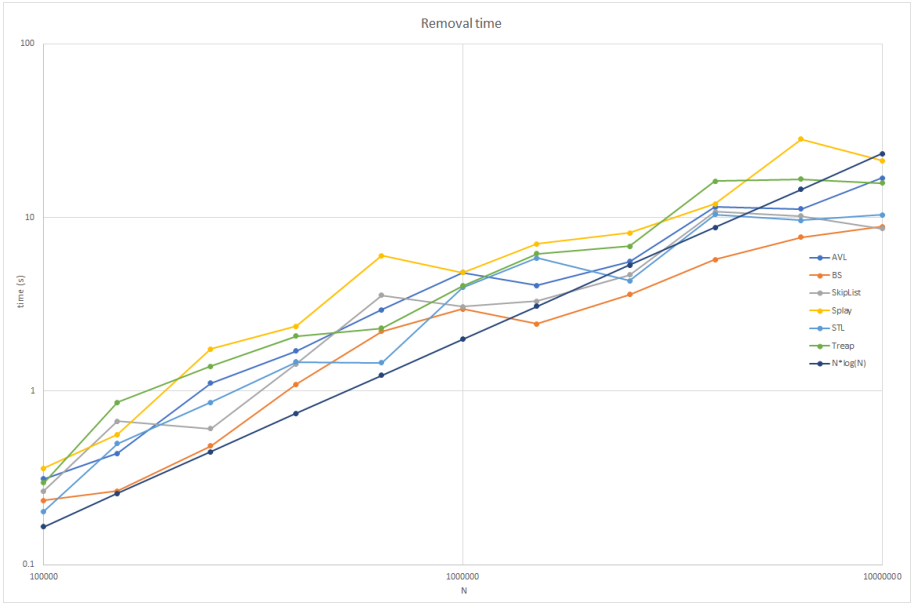

Figure 3: Running time for removal

In Figure 3 we show the results obtained. The runtimes are slightly more dispersed than before, but they are all very close to the expected values. Actually, there is a small downward trend for large values of N. This is probably caused by the decrease in the size of the Set as the total number of successful removals increase.

Last but not least, the search operation. It needs the same filling step as the removal operation and has the same input file structure as the previous operations.

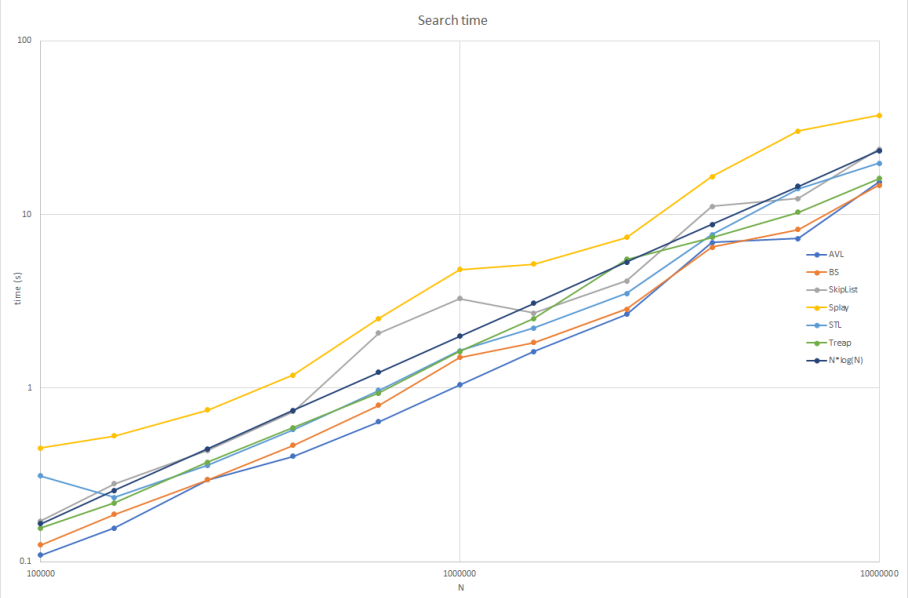

Figure 4: Running time for search

In Figure 4 we show the results obtained. The plot is very similar to the plot for the insertion operation with just minor fluctuations at some points.

4.3 BST Worst case

The whole point of having a balanced BST is to avoid the worst case complexity of $O(n)$ time per operation. Given that it is so easy to test this, it would be a shame to not include it. To do this, we inserted, during the initial filling stage, every value in the domain in order, instead of at random. After the initial stage, we performed 10^5 search operations.

Figure 5: Running time according to insertion ordering

In Figure 5 we show the results obtained. We can clearly see the huge impact that the order had in the runtime of the BST, since it scaled it up by more than two orders of magnitude, while having little impact on the other data structures.

4.4 Access Locality

Given that one of the main advantages of Splay Trees is that they are much more efficient when there exists some sort of locality in the queries, we decided to check if our implementation expressed that same property. To do that, we changed the way we generated the random values by having just 1% of the number unique values as before, while keeping the domain range the same. Then, we performed 10^6 search operations and compared the results with the corresponding previous test.

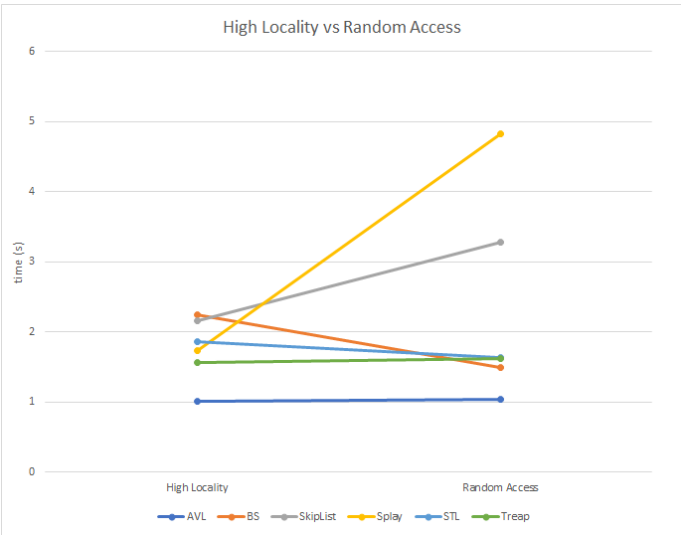

Figure 6: Running time according to access locality

In Figure 6 we show the results obtained. We can see that the Splay Tree was the one that showed the largest drop in runtime when reducing the search domain and becoming one of the fastest approaches.

4.5 SkipList Level Probability

Out of curiosity, we decided to check if changing the probability that determines the level at which a node is inserted would affect the running time of SkipList. To do so, we changed that value while maintaining the number of search operations.

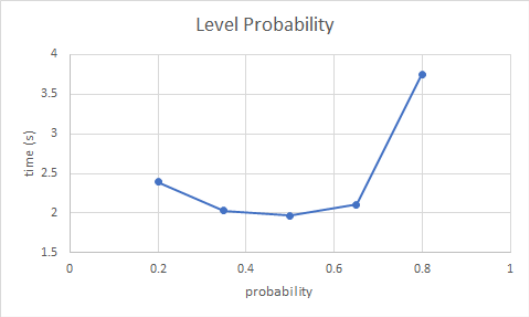

Figure 7: Running time according to the probability of level creation

In Figure 7 we show the results obtained. It looks as though the optimal value is around 0.5, which is interesting because, at that point, its structure resembles that of a BST.

5 Conclusions

In the results section we provided a comparative analysis of the running time complexities of the implementations of the different data structures. This comparative analysis was done not only by comparing between implementations, but also by comparing these implementation against two distinct baselines: the behaviour of C++ STL's `Set`; and the expected theoretical behaviour. Throughout all of the analysis, each one of them with respect to different test cases, no surprises were found. The running times were as expected and all variations that we implemented showed to be nearly as efficient as STL's. A more rigorous statistical analysis of the running times, as well as the analysis of the spacial complexities associated with each implementation would have enriched this work even further. For now, this is left as future work. :)