Laboratory Report for the Algorithms and Data Structures Laboratory Class
of the University of Florence

A Comparative Benchmark of Multiset Binary Search Tree Variants: Regular, Alternating and List-Grouped

Besnik Nuro

Abstract
The Binary Search Tree is a fundamental data structure used for its logarithmic ordered search time complexity. While traditionally not meant to store duplicate keys, certain applications, for example ones involving colliding hashed keys or word frequency counters may warrant handling of duplicates in a certain manner. In this laboratory report three multiset implementations are put against each-other to measure performance when duplicates are used: Regular Insertion BST, Alternating Insertion BST and a multiset BST with a duplicate collection.

Keywords
Binary Search Tree – Multiset – Benchmark

Contents

A bit of theory

Multiset collections are sets in which duplicates are allowed. A Multiset Binary Search Tree would be a Binary Search Tree (BST) with the ability to insert and manage duplicates.

As regular implementation of a BST allows for duplicates but they are made inaccessible unless deletions are used.

An Alternating BST would be a BST in which insertion of duplicates is alternated between the left and right nodes.

For example, three equivalent keys inserted in succession degenerate to a linked list in a regular BST but in an Alternating BST they are equivalent to the first insertion being a parent with the other two nodes as left and right siblings. In fact it can be trivially proven that for a succession of n equivalent key insertions, a balanced subtree of duplicates forms.

An untitled Multiset BST implementation, here called List-Grouped BST, stores duplicates in a list (optionally linked) pointed to by the first equivalent node inserted. This implementation has the advantage of easy retrieval of the values and shorter heights which imply shorter search times. If a succession of n equivalent keys increases height by n in a regular BST implementation, $\log_2(n)$ in an Alternating BST, in a List-Grouped BST the height is increased by one level and the rest of the duplicates are stored in multiset collections pointed to by the nodes. This implementation is also more space-efficient if keys are not stored in node containers inside the collection.

1. Basis and Experiments

To have an idea of why the data structures exist and how experiments were formulated, a description of the problem is in order.

1.1 The need for multiple keys

Multiple key handling is required when BSTs are used to implement more advanced data structures.

1.1.1 Priority Queues

Priority Queues are data structures often used for task scheduling in operating systems. More generally, they can be used in any application where a data structure of elements ordered by priority is needed. For example, a CPU with distinct performance and efficiency cores can make use of a task scheduler which uses a Multiset BST-based Priority Queue to retrieve tasks with lower (but sometimes equal) priority for the efficiency cores. This is preferable to a heap implementation which favors highest priority retrieval.

1.1.2 Maps

Maps are sometimes implemented as self-balancing BSTs. This implementation is simpler compared to using a hash table. `std::map` uses a self-balancing BST in its implementation. The motive for handling duplicate keys arises when hashed keys can collide with each-other.

1.1.3 Order Statistics

Order-Statistics Binary Search Trees (OS-BST) are used for order statistics to efficiently find elements by their rank and perform range queries. A multiset implementation is needed since there can be elements with the same key. Examples are text analyzers and analytics/statistics in different fields.

1.2 Why a regular BST is not enough

The advantage Binary Search Trees offer is that they have fast ($O(\log_2(n))$) query times, as long as they are balanced. In case of left-skewed subtrees (one produced by successive duplicate insertion), the query time within that subtree is $O(n)$. This is because it is no different from a linked list search.

The search time is proportional to the height of a given tree, so the more balanced a Binary Search Tree, the faster the query time. By accounting for duplicates in the implementation of a Binary Search Tree, we can reduce this height via alternate insertion or collection insertion in accordance with the application intended for the data structure.

1.3 Implementations

We consider the three implementations mentioned in the introduction.

1.3.1 Regular BST with left-biased insertion

A regular BST node is a structure containing:

- a parent node reference;
- two (`left_node` and `right_node`) references to sibling nodes;
- a key which is for sorting values;
- a value contained or pointer to it;
- a root reference;

In these implementations, for the sake of simplicity, `value` refers to both the key and value contained. There is no reason why these implementations wouldn't work for different key and value attributes.

Python code for BST node structure

```
class BinarySearchTree():
    def __init__(self, value):
        self.parent = None
        self.left_node = None
        self.right_node = None
        self.value = value
        self.root = self
```

Initializing a tree as `BinarySearchTree(n)` creates a root node containing n as its value. It and every successive node points to itself when queried for the root node. This way subtrees are conceptually implemented.

A `print(node)` will print the subtree with `node` as its parent key. This is because `__str__` returns the string representation of a class which the print function uses. By overriding `__str__` we can give a tree an presentable visual representation.

For the documentation of the code, check Appendix A.

```
def __str__(self):
    return (self
    ._BinarySearchTree__tree_walk_helper())
```

Insertion in this case would be as the left node of the equivalent parent. When queried for a key, the tree returns the first node matching the queried key. That means only one node reference is returned for any amount of equivalent keys and the keys not returned are inaccessible unless the any nodes coming before them in order are deleted.

1.3.2 BST with alternating insertions

The implementation of this structure is the same as for the regular BST, with the exception of an additional attribute `preferLeft` and a few more lines in the logic of the `insert` method. For each successful insertion, the value of `preferLeft` is alternated between `False` and `True`.

1.3.3 BST with collection for duplicates

This implementation differs from the regular and the alternating one in that each node points to an collection for duplicates. Insertion here does not inser t in the tree if there are duplicates, but in an appropriate collection instead. In our implementation we have used a list, but a deque could be used instead for fast insertions and deletions.

Querying for a key contained in the tree returns the node and a flag which asserts whether there are dupes for that key. `get_duplicates` returns a reference to the collection of duplicates.

1.3.4 Expectations

We can expect a regular BST's height to increase linearly for each successive duplicate insertion. For low collision rates this should not be a problem.

An alternating BST's height should increase logarithmically for each successive duplicate insertion. For example, for 5 equivalent insertions, the height should increase by 3 and for 100 it should increase by 7. This would be the same rate of increase as a regular BST with random insertions.

A list-grouped BST's height should increase by one node for any succession of equivalent insertions. This means that for n equivalent insertions, the height increases by one and the list of duplicates pointed to by the node increases in size by $n-1$.

1.3.5 Experiments

Different test cases were devised in order to confirm or deny the expectations. There is a baseline test case for randomly inserted keys. This will be used to compare against. Another reference would be the upper bound of the worst case (all duplicate insertions), which will not be graphed and has already been mentioned in the introduction and problem formulation.

There are two test cases intended to mimic low collision rates of samples and hashed keys:

- 0.01% collision rate
- 0.05% collision rate

Two more cases intended for moderate collision rates of samples:

- 0.1% collision rate
- 0.5% collision rate

Three cases are intended for high collision rates of samples:

- 1% collision rate
- 5% collision rate
- 10% collision rate

The data structures will be tested for the insertion of 1000, 5000, 10000 elements with the appropriate collision rates for each test case. Collision rate is $\frac{1}{\texttt{max\_value}}$.

Test Case for 0.1% Collision Rate

```
    max_value = 999
    size = 10000
    root_value = max_value//2

    regular = \
    BinarySearchTree(root_value)
    alternating = \
    AlternatingBinarySearchTree(root_value)
    listgrouped = \
    ListGroupedBinarySearchTree(root_value)

    for i in \
        SyntheticData. \
        generate_integer_list(size,
                max_value):
        regular.insert(i)
        alternating.insert(i)
        listgrouped.insert(i)
```

The height for each tree is recorded after the insertions and exported to a file. This way tests can be run in batch.

```
    output = \
    str(regular.get_height()) + " " + \
    str(alternating.get_height()) \
    + " " + \
    str(listgrouped.get_height())

    Export.export_values_to_file(output)
```

The exported data is listed in columns, each column containing values for a BST's test case results.

```
# cat data/export
18 13 6
20 11 5
19 10 5
```

A shell loop was used to run many cases sequentially so that a mean value could be extracted. For this report, a hundred runs were made.

```
for i in {1..n}
        do python bst/main.py
done
```

2. Results

The data was generated via a few bash scripts using awk for processing of the data and gnuplot for plot generation. The data were approximated with bezier curves.

2.1 Low Collision Rates

For low collision rates, the regular and the alternating variants of the BST perform similarly and there is a slight gain of performance only for list-grouped BSTs.

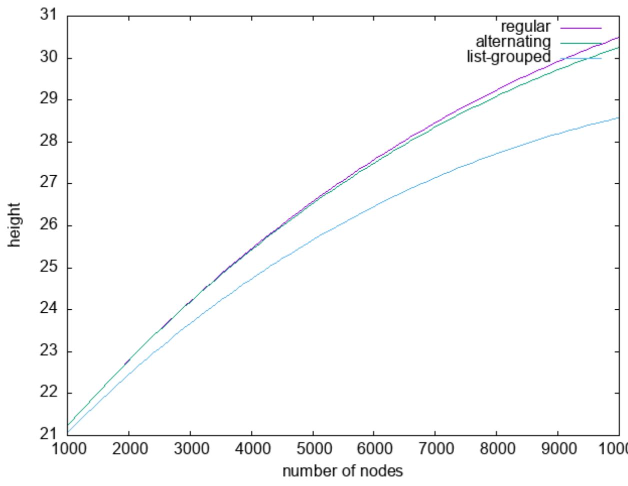

Figure 1. 0.01% Collision Rate Plot

2.1.1 0.01% Collision Rate

The regular and alternating BSTs perform almost identically, with the alternating BST height being slightly lower. The list-grouped version has a lower height, but not by much.

2.1.2 0.05% Collision Rate

Results are similar to the 0.01% collision rate case.

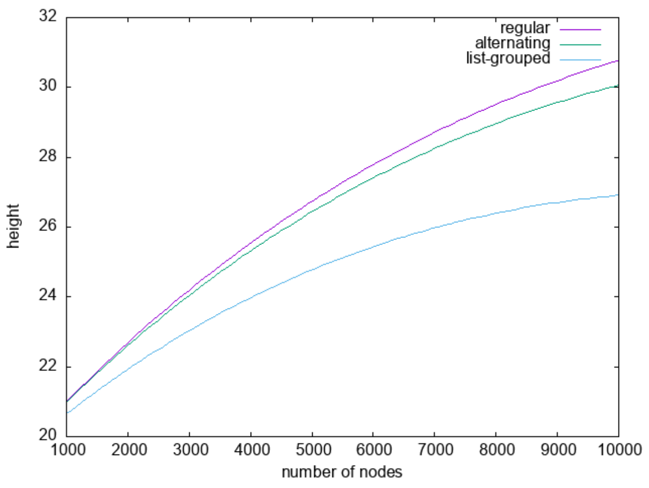

Figure 2. 0.05% Collision Rate Plot

2.2 Moderate Collision Rates

In moderate collision rate test cases, alternating BSTs and list-grouped BSTs start to show their resistance to height scaling with respect to duplicates.

2.2.1 0.1% Collision Rate

Here we see the first considerable difference between the three BST variants. While the regular BST scales almost linearly, the alternating BST performs much better, still retaining the logarithmic scaling of the height. List-grouped BST has an almost linear height scaling.

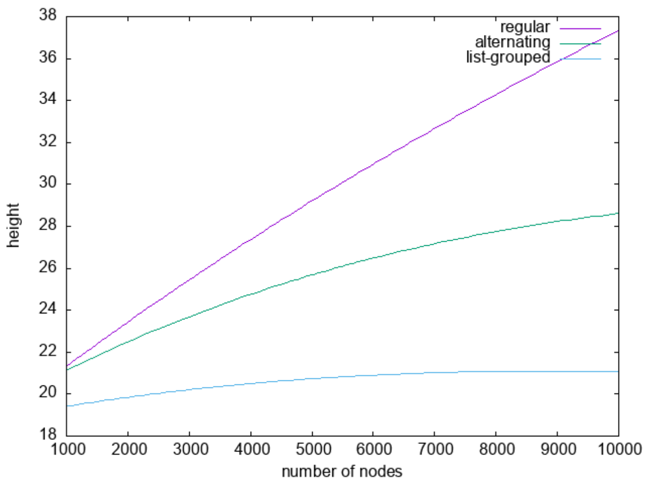

Figure 3. 0.1% Collision Rate Plot

2.2.2 0.5% Collision Rate

In the 0.5% test case, the difference is even more pronounces. While the regular implementation continues to scale linearly, the other two variants fare better: there are enough repetitions for these two BST implementations to make use of their properties. Interestingly, the height of the list-grouped BST remains constant on average with respect to the lower bound of 1000 elements. This may imply that for this collision rate any application that foresees a structure with more elements than this lower bound, the use of the list-grouped implementation is the most beneficial one.

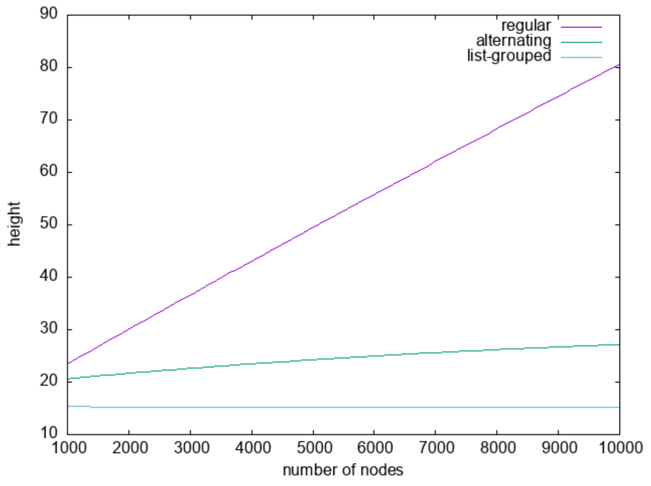

Figure 4. 0.5% Collision Rate Plot

2.3 High Collision Rates

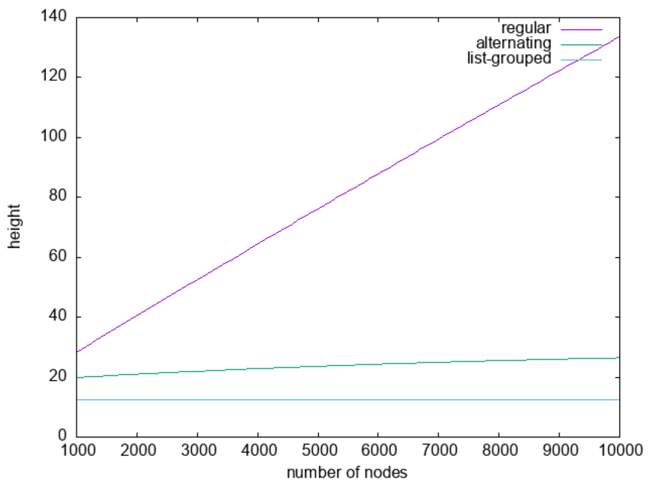

Figure 5. 1% Collision Rate Plot

The common denominator for this class of test cases is that the regular implementation of the Binary Search Tree maintains a linear scaling of its height, whereas the alternating and list-grouped BSTs show a constant or almost constant height on average. In fact:

- for a 1% collision rate, the height of list-grouped had an average of 12.6 and did not surpass 17;
- for a 1% collision rate, the height of alternating had an average of 26.4 and did not surpass 31;
- for a 5% collision rate, the height of list-grouped had an average of 7.2 and did not surpass 9;

- for a 5% collision rate, the height of alternating had an average of 24.8 and did not surpass 31;
- for a 10% collision rate, the height of list-grouped had an average of 4.9 and did not surpass 6;
- for a 10% collision rate, the height of alternating had an average of 24.2 and did not surpass 27;

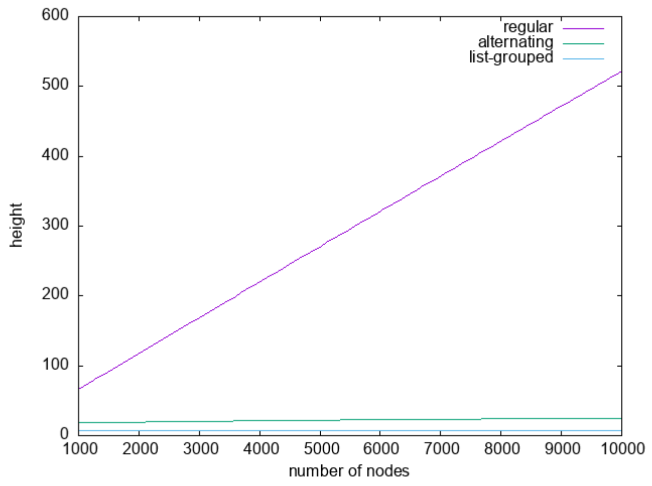

Figure 6. 5% Collision Rate Plot

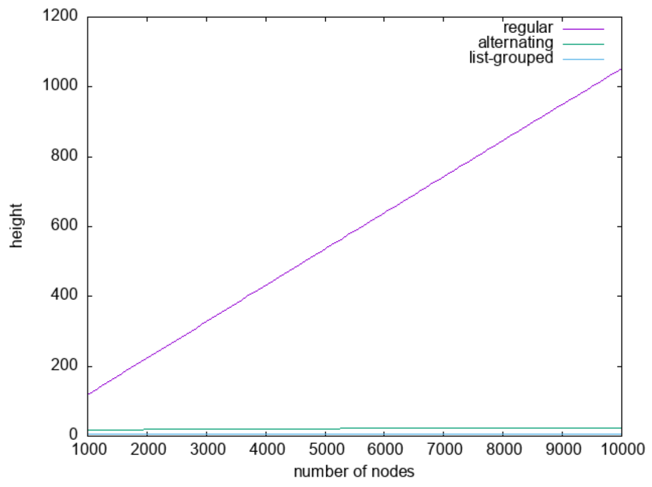

Figure 7. 10% Collision Rate Plot

3. Analysis and Interpretation

As hinted before, low collision rate test cases were meant to mimic applications where keys rarely collide. Hashed keys are one such example. Typically they have very low collision rates, but we can already see that in the 0.01% collision rate test case the regular and alternating BSTs perform almost identically, leading us to believe that for lower rates of collision this should be true as well. We can say the alternating BST implementation does not have any significant benefits in these conditions with respect to the regular implementation. However, the list-grouped BST scales a bit better in the 0.05% collision rate. With this in mind we can see that list-grouped BSTs may be worth it where performance is more of an issue and we don't have access to better alternatives.

For moderate collision rates the trade-offs become more apparent:

- height of the regular BST starts to increase linearly with respect to the previous test class. This means that for uniformly-distributed ranges of keys with collision rates equal or higher to 0.1 percent, the regular BST start to throttle in performance.
- height of the list-grouped BST starts to look more like a constant line. We can take this to mean that from this point on the collection BST implementation fairs better for searches than its competitors.
- the alternating BST is somewhere in the between: whether of not to use it remains to the be decided by further circumstances. For example, in a application where performance is critical this variant wouldn't be useful, whereas in another where we want moderate performance for searches it can be decently performant.

High collision rate test cases were meant to roughly simulate instances in which repetitions are common or probable. One such case could be a frequency analyzer for a text or for a corpus of population data with respect to certain attributes such as name, surname or region of provenance. Regardless of the actual application, this class of tests shows that even an alternating Binary Search Tree can be useful, almost as much as a list-grouped Binary Search Tree. Regular implementations, on the other hand, scale with a complexity between linear and logarithmic. Locally, between the 1000 and 10000 elements it behaves almost linearly. Even insertion times during tests were increased noticeably for this class of collision rates with respect to the other test cases.

4. Appendix A: Code Documentation

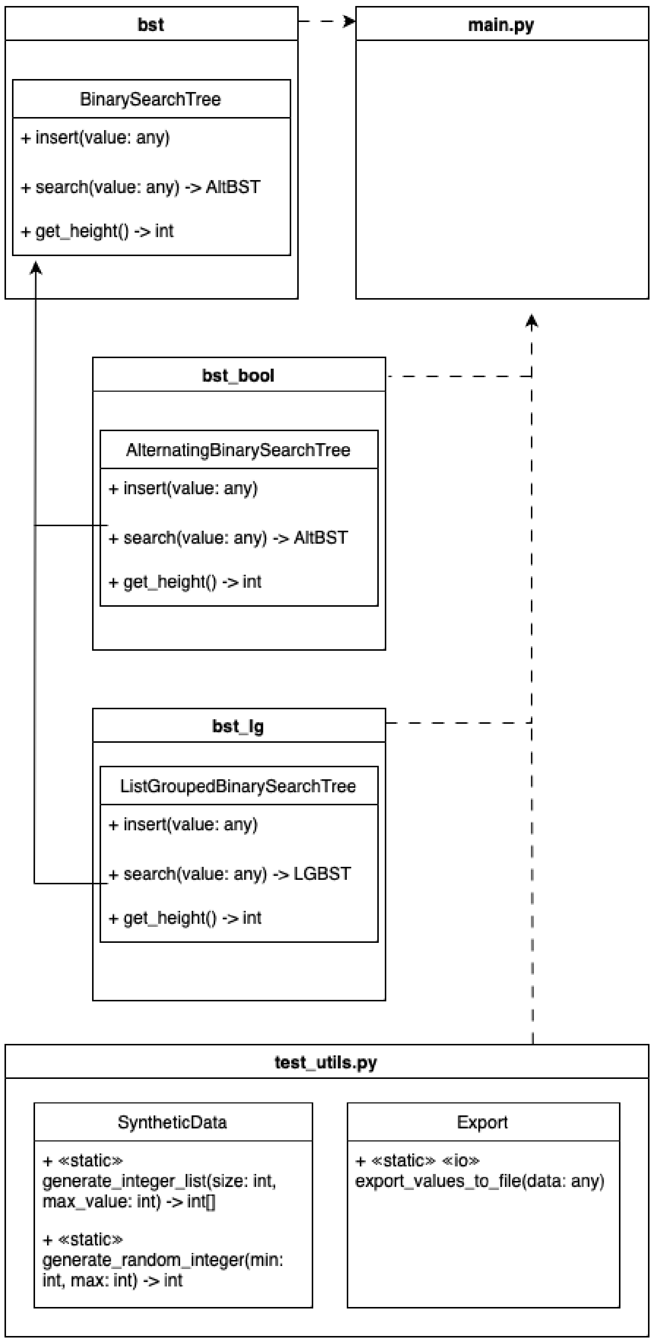
bst
BinarySearchTree
+ insert(value: any)
+ search(value: any) -> AltBST
+ get_height() -> int
main.py
bst_bool
AlternatingBinarySearchTree
+ insert(value: any)
+ search(value: any) -> AltBST
+ get_height() -> int
bst_lg
ListGroupedBinarySearchTree
+ insert(value: any)
+ search(value: any) -> LGBST
+ get_height() -> int
test_utils.py
SyntheticData
+ «static» generate_integer_list(size: int, max_value: int) -> int[]
+ «static» generate_random_integer(min: int, max: int) -> int
Export
+ «static» «io» export_values_to_file(data: any)

Figure 8. Module and Class Interaction Diagram

BinarySearchTree

__init__(self, value)

Initializes a tree with the root node as itself, contained value/key with the given value, null pointers to siblings nodes and parent node.

__str__(self)
Override of the `__str__` method, returns a visual string representation via a tree walk.

__tree_walk_helper(self)
Helper wrapper for the `__tree_walk` method.

__tree_walk(self, buffer, vbranch)
Calls itself recursively to construct a visual tree representation stored in the buffer via an in-order tree traversal. Returns a string representation of the tree for visual debugging.

insert(self, new_value)
Performs a textbook Binary Search Tree insertion. Left-biased for duplicates. Returns a reference to the new node.

search(self, value)
Performs a recursive binary search for a given value on the tree. Returns a reference to the first equivalent node if found, else returns `None`.

get_height(self)
Returns the height of the tree.

AlternatingBinarySearchTree

__init__(self, value)
Initializes a tree with the root node as itself, contained value/key with the given value, null pointers to siblings nodes and parent node. Initializes `preferLeft` to `True`.

__str__(self)
Override of the `__str__` method, returns a visual string representation via a tree walk.

__tree_walk_helper(self)
Helper wrapper for the `__tree_walk` method.

__tree_walk(self, buffer, vbranch)
Calls itself recursively to construct a visual tree representation stored in the buffer via an in-order tree traversal. Returns a string representation of the tree for visual debugging.

insert(self, new_value)
Performs a Binary Search Tree insertion. Duplicate insertion is alternated by checking `preferLeft`, toggling it and inserting accordingly. Returns a reference to the new node.

search(self, value)
Performs a recursive binary search for a given value on the tree. Returns a reference to the first equivalent node if found, else returns `None`.

get_height(self)
Returns the height of the tree.

ListGroupedBinarySearchTree

__init__(self, value)
Initializes a tree with the root node as itself, contained value/key with the given value, null pointers to siblings nodes and parent node, and a null pointer for a list of duplicates to be initialized upon duplicate insertion.

__str__(self)
Override of the `__str__` method, returns a visual string representation via a tree walk.

__tree_walk_helper(self)
Helper wrapper for the `__tree_walk` method.

__tree_walk(self, buffer, vbranch)
Calls itself recursively to construct a visual tree representation stored in the buffer via an in-order tree traversal. Returns a string representation of the tree for visual debugging.

insert(self, new_value)
Performs a Binary Search Tree insertion. When the first duplicate for a key is inserted, it initializes the pointer to the list. Otherwise, it puts the duplicate inside the list. Returns a reference to the new node.

search(self, value)
Performs a recursive binary search for a given value on the tree. Returns a reference to the first equivalent node if found, otherwise `None`. Also returns a flag indicating whether there are duplicates as the second positional return value.

get_height(self)
Returns the height of the tree.

get_duplicates(self)
Returns a reference to the list of duplicates for its node.

SyntheticData

generate_integer_list(size, max_value)
Returns a list of length equal to `size` and with values ranging from 0 to `max_value` inclusive.

generate_random_integer(min, max)
Returns an integer between `min` and `max` inclusive. Originally used for debugging.

Export

export_values_to_file(data)
Creates a file called `export` inside the `data` directory and writes the string representation of the passed argument to it.

5. Appendix B: Explanations about choices

Why isn't there a Node class?

Nodes and BSTs are represented as the same structure. The reason for this is that programming the logic for two separate structures increased the complexity of the code without adding meaningful improvements. However, it is possible for the variants to extend a class representing a vertex or node without affecting any of the functionality provided by this implementation. I didn't do this because it would not be very useful to have a `Node` class with only a string representation method and a value reference inside, especially when the inheriting classes would contain them anyway.

Why is the measure the height of the trees and not the time required for insertions and searches?

Since the upper bound for time of an insertion or search takes is linked to the height of the tree, measuring height offers us a proxy and a way to measure expected performance without actually timing results. This means that regardless of the machine in which tests are run, the expected performance should be proportional to height. Indeed, a higher tree took longer to populate on average when I ran the tests.

How were the data and plots generated?

Each test case was manually ran a hundred times for each population of 1000, 5000 and 10000 nodes. Height was measured and written in columns in a text file. The first column represents a regular BST, the second an alternating BST and the third a list-grouped BST. In this manner a 100×3 matrix was created. The columns were reduced to averages via a `apply_avg.sh`, a shell script using awk to average the columns. These averages were then plotted using `gnuplot` with 1000, 5000 and 10000 insertions as ordinates.

Where can I find the data used for this report?

The `data` directory contains all data used to generate the plots, while `utils` contains scripts and intermediary data relevant to the original data.