Assignment 2
Randomized Search Tree and Partial Persistent Linked List

Jacob Nielsen, jacon18@student.sdu.dk

May 2022

1 Randomized Search Tree

1.1 Implementation

We briefly go over the implementation here, as it very much as already described in the lectures.
When insertion an element, we are inserting recursively (called from a wrapper function), first insertion on key property and then fixing the heap-property using either `right_rotate` or `left_rotate` function. The searching is simply a normal search as in a normal binary tree. Deletion is done by going through the tree recursively and, find the target-node and then replace the node by the merge of its two children. Merge is done with the two functions `merger` and `merge` depending on if you want a tree or a node back. `merger` is used by the function `delete`, that works very much like a normal delete function, by then replacing the target-node with the results of merger.

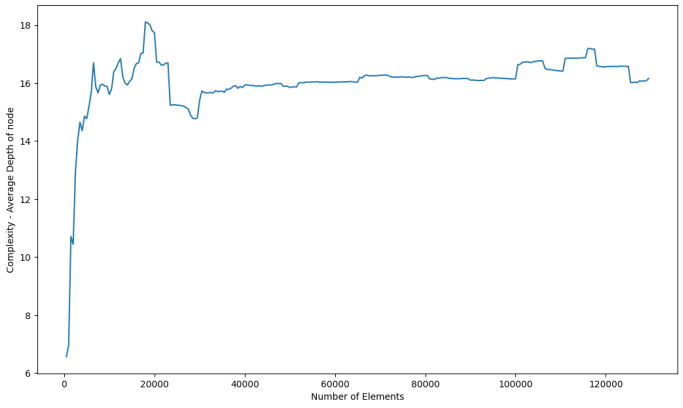
1.1.1 Average Search Complexity

As stated in the assignment, the search complexity is measured from the depth that the key is found. Therefore. A series of **N** number of keys is inserted into the data structure and then **J** number of searches is done within each N. The average dept the keys/nodes is found in, is the average search complexity for that N. All tests is performed with a random-range that is the same size as the size of N, the reader should take this into account.
What we found out, was that the numbers of searches should be of a certain size to give stable results after multiple plots. This is probably to do with, first of all getting a representative representation over a bigger part of the key-domain, but also a consequence of the probability in the data structure. After this discovery, the results is very stable and the behaviour of the algorithms seems be somewhat predictable.
The data in the plot is generated from 500 to 13000 number of elements (N), with a step on 500. Each N is then searched 3000 times (this for having base number of searches, could have been adaptive). We can conclude that the depth of the nodes is not exceeding 18 as complexity in our test - hence the depth in our tree. Overall this really shows how effective this simple data structure is on 130.000 elements. Further i can be seen from the graph, that the number is balanced from a little before 30.000 inserted elements.

1.1.2 Variation in search complexity

As the investigated property in Average Search Complexity is a smoothed result, we would investigate how the data structure behaves in multiple runs - the variation of the search complexity. This allows us to see how much a certain complexity is activate over many sample-searches.
Above is shown 4 figures. The many figures is provided because of the implementor's thoughts in the spike around $i = 15$ in figure 2, and wanted to test this tendency. All tests is done with $N = 130.000$. A change to this would probably

Figure 1: Average Search Complexity



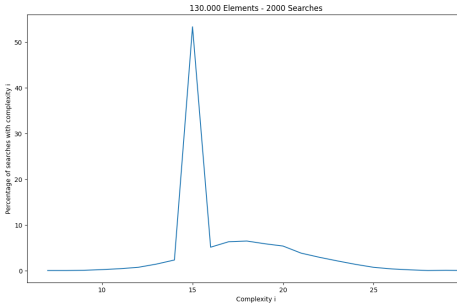


Figure 2: 2000 Searches

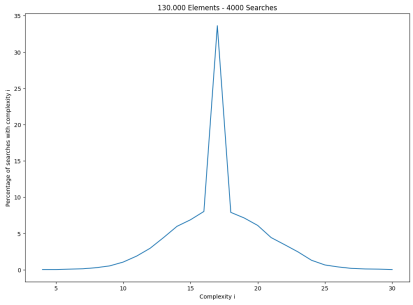


Figure 3: 4000 Searches

push the spike around - or a least a few side-tests seem to show the tendency to the spike to be on a smaller complexity value when N is smaller.
All i all it seems that the data structure has a complexity spike with where a a big percentage of the searches consists of. This makes the structure quite easy to reason on.

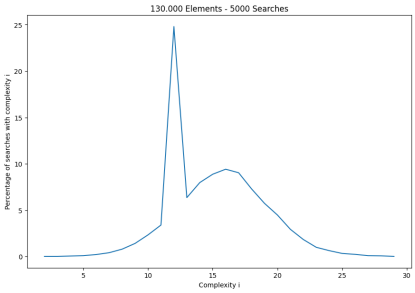


Figure 4: 5000 Searches

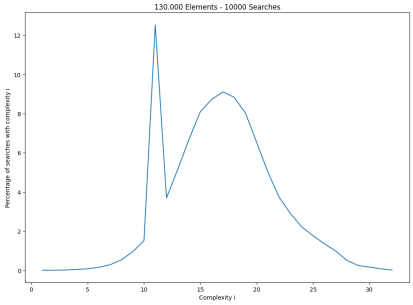


Figure 5: 10000 Searches

2 Partial Persistent Singly Linked List

A partial persistent linked list, is a linked list that that supports partial persistence. This means that while you can only modify/update the latest "live" version of the data structure you can search/lookup in all versions that was live.
What we are describing, implementing and analysing here is not what is expected when one think of a linked list. First of all, while the nodes do contains a `key` attribute, passed when inserting, the key are not used as insertion variant for a sorted list. Instead we are insertion on an index basis. All nodes represent an index, with the first one at index 0. The insertion index is chosen by the client. When inserting the, is it optional to pass another key, `k'`, that determines the allocating of another associated node, that the first node points to via it's `assoc`-field as dictated by the assignment. These associated notes points to each other, via next fields, just as their parent nodes. The parent nodes are not required to have an associated node, but the associated nodes, that do exist, must be linked, "skipped under" the child-less nodes.
Below, the big design-descriptions and mechanisms are explained.

2.1 The Node

The nodes are the driven element in the persistent data-structures. They are the organizers that have the responsibility for each node, keeps tracks on its versions, called a modification in this text and the code, after the initial assigning is made. Node can also be copied, using the `node-copying` technique, copying only the newest data into the newly created *copy*, making new room for modifications. The old node(s) and the new node (the latest node is called the `live`-node) in the paper), is linked via back-pointers, that are changed to point to the new, now live node, as described in the paper. The implementation allows an adaptive number of modifications internally in the nodes, before this mechanism is triggered.

2.2 Insertion

Insertion is not that different other than traversing the list with a number of indices instead of search for a position based on a key. While doing this, we are applying a few tricks of the cost of constant work for each element-iteration, to save us some search time later, if if need to insert a `k'` associated node.
First of all, if a k' mark value is parsed, we are creating an additional node, besides the "parent" node, we are always creating. If the list does not have any versions or an list-head, we are using the first insertion to create a list head and a version.
Otherwise we see if there is existing any versions and a head of the list. If not we are creating this the first version and inserting this first node as head. If, we need to look through the list. This is done with couple of probes keeping track of the "to-be-next", "to-be-prev" and "last-seen-empheral" node, that is helping us keep track in information. to-be-next is pointing at the next node and control the iteration. "to-be-prev" is used to modify to hold a pointer to this new node, where we of course also are inserting a pointer to "to-be-next" from. The last-seen-empheral, is keeping track of last time we saw a node with a not-None assoc field. This is because this node needs to point to our assoc-node, if any.
Further, we need to search to the next-first assoc node, if we are not inserting a node last in the list, therefore we also have a last-node parameter. If this is true, we are searching, potentially, the rest of the list for a pointer to the next assoc-node. Then we are inserting a next-pointer to our new node, from the to-be-prev node before determined the new head status in this insertion, before lastly, finalising the attribute-assignments in the new version and appending the new version to the lists version-array.

2.3 Search

As we during insert are keeping track of the versions head - as mentioned in the paper as an "auxiliary data structure to keep track of heads", in an array, we can easily get the head in a certain version using the `head_in` function, that is simply returning the index of the version's head, which is a node pointer. From there the index (nodes) can be traversed using the nodes own function `next_in(version)` function, that is traversing the nodes modifications finding the next-pointer for the specific version. From here the given node is traversed and from here we can call the `key_in()` and `data_in()` functions on that given node, which is returning the key-value and the data/assoc value from that node. This could have been optimized to be returned directly, but it is not chosen.

2.4 Update

The update-function, takes a value and a val_mark, to update the node and the assoc-node-values (if val_mark is passed). First the functions get's the latest head using the `head()` function of the list. Then the nodes are iterated using

the `next()` function. When we have a pointer to the given node, we can use the function `update_data()`, where we parse the nodes and the potential two update-values.
This function creates a new version, not to modification in earlier version, but to preserve them. Then it calls first an updating on the key field of the node it self and then update on the assoc-node, if val_mark is passed.

2.5 Copy Live Node

The `copy_live_node`-function is called from the `update_field` function, that also can be triggered from `insert()`.
The function is parsed an allocated node, the node to copy, the field name to "write" in the new node, the value to put into the given one of the pointer-types - this is the latest information from the old node, as mentioned in the paper. Lastly the head is also parsed, to linked the live and dead nodes together. The `next()` function, to assign the back pointers to the other nodes to the copy.
If the copied node have a next back-pointer, we are updating this next-back-pointer also. This is done using the `update_field()` function.

2.6 New Versions

To keep easy track on the progress during development it was chosen to update the version after each insertion, but we are required by the assignment to create a version on direct demand. We are doing this by simply taking the state of the list: head and and size, updating the version number, creating a new Version object that we are appending to the list of versions.

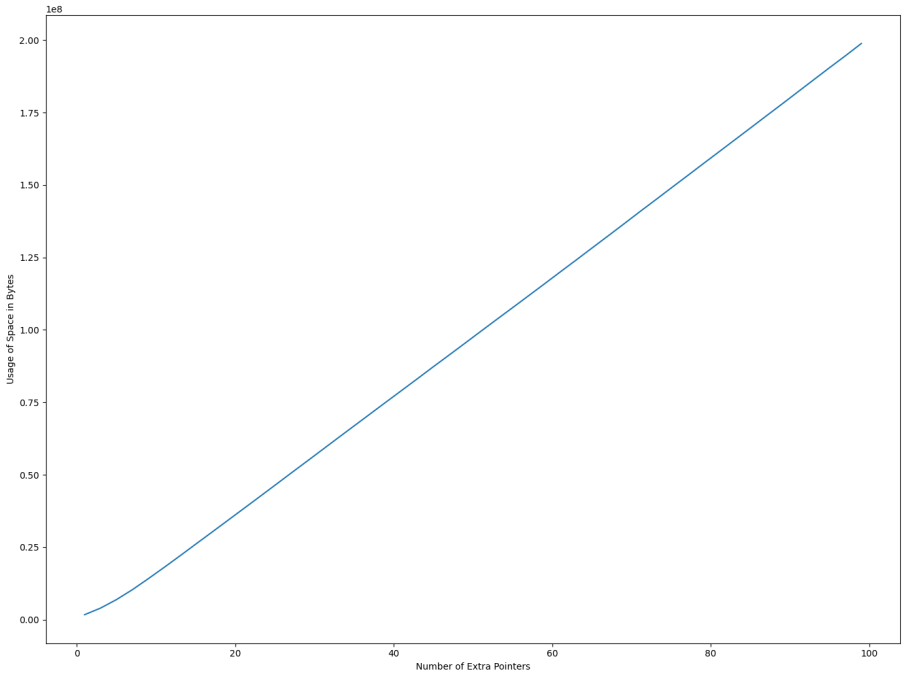
2.7 Analysis

2.7.1 Space Used as function of Allowed Extra Pointers - Bytes

What we are looking for here is, if where is any connection between the number of pointers and space used. Further we are asked to look for an optimal value. Theoretically, we would expect that we would rather one a connection to a new node, than we want to search a list/array of modifications in one node.
A sequential number of indexes with identical keys are created. From here they are pop'ed randomly and inserted into the Linked List. As this not creates the worst case scenario, it creates a scenario that simulates use in practise.

At first glance this seems very wrong and maybe it is? - but there could be some reason behind it. It comes in the following. First of all we are testing on very large numbers here, and the data-structure is not made to very large number of extra pointers, as we would then end up making a linear search on them. Further, when the number of pointers is growing, so is the number of allowed bytes, this is also accounted to be expected for the number of allocations. There is a little bend in the line just after 0, probability around 3 extra pointers, before the function scale linearly and this is probably the optimal number of

Figure 6: Space used in bytes used per extra pointers

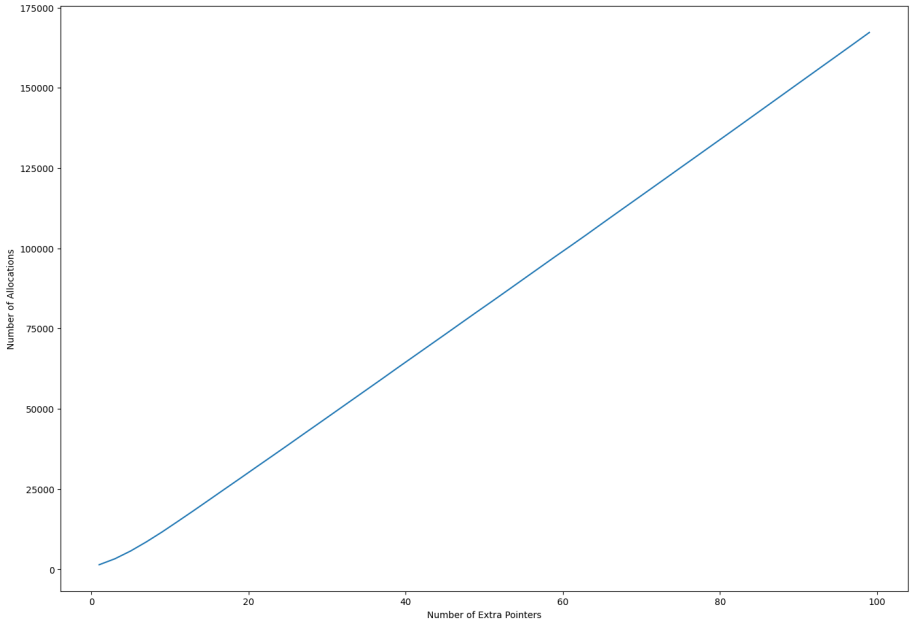


extra pointers - for this implementation at least. What we like in this kind of structure is a few recent pointers to control and be accessed, and older pointers in "dead" nodes to take longer time to traverse through the pointers.

2.7.2 Space Used as function of Allowed Extra Pointers - Number of Allocations

As expected this looks very much like the graph above. Not much surprising here, as the number of allocations follows the space used. So even if this is not quite right, these two follows as expected. Again this graph would suggest the lowest number of allocations with 3 extra pointers. Afterwords is does seem that there is a linear coellation between the number of extra fields (called modifications) and allocations used.

Figure 7: Space used in bytes used per extra pointers



3 Conclusion

Overall a data structure that works and shows the use of different pointers and the copy-node technique is provided. It is tested and works as described in the assignment and paper.
The results of the RST seems to be quite trustable and follows what one might expect from the data structure. However the results form the Partial Linked List seems explainable, but not necessarily convincing, other than the arguments already given. The implementation is looked through and seems to be right - likewise with the test-script.
In both implementation there is provided functionality to show the structures and document their form. But, differently the test and the utilize for this could have been a lot more sophisticated. The main functions in the module/files of the data structures gives a lot better view on how the structures work. This especially account for semantic checking. `main_test.py` for PLL and `rst.py` for RST. The functionality in the `main.py` file is not satisfactory and does a bad job in general. Apologies for that before-hand.