Search Trees in Practice

Theis F. Hinz

Kongens Lyngby 2015

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Summary (English)

This thesis considers binary search trees with respect to the performance of searches. In order to study the strong data structures are the competitive analysis applied.
The working-set, dynamic finger and sequential access property are introduced with respect to this analysis.

Two binary search trees are considered. Splay trees are proved to have all of the properties mentioned above, and they are conjectured to have the even stronger unified property.

Tango trees are proved to be $O(log log n)$-competitive, but this thesis shows that it does not compete well against the upper bounds of the optimal offline binary search tree.

Finally, red-black, splay, and tango trees are experimentally compared. The results show that red-black trees are the best of the three types of trees if searches are on random chosen keys. However, splay trees are found to be the best if searches are close in time or key space. In none of the experiments was the tango tree found to be any better than the others.
The experimental results justify the conjecture that splay trees have the unified property.

Summary (Danish)

Denne afhandling undersøger *binary search trees* og deres ydeevne ved søgning. For at analysere stærke datastrukturer er *competitive analysis* benyttet og med hensyn til analysen introduceres *working-set-*, *dynamic finger-* og *sequential access property.*

Afhandlingen fokuserer på to binary search trees. Splay tree er bevist at have alle ovenstående nævnte egenskaber og formodes at have den endnu stærkere *unified property.*

Tango tree er bevisligt $O(log\ log\ n)$-competitive. Men som denne afhandling viser, klarer den sig ikke vel mod de øvre grænser for det optimale offline binary search tree.

Endelig sammenligner afhandlingen splay-, tango- og red-black- trees eksperimentelt. Resultatet underbygger at red-black trees er den bedste af datastrukturerne, hvis søgningerne benytter tilfældigt valgte keys. Splay trees har derimod den bedste ydeevne, hvis søgninger er tætte i *key space*, eller hvis søgninger på samme key er tætte i tid. Undersøgelsen fandt ingen tilfælde, hvor tango trees var bedre end de øvrige search trees.
De eksperimentale resultater underbygger formodningen om, at splay trees har unified property.

Preface

This thesis represents the end of my effort for acquiring an M.Sc. in Engineering at DTU.
The preparation of this thesis is motivated by the interest of finding tight bounds for online search algorithms for binary search trees. And as the study has shown, we are close: Splay trees are conjectured to be dynamically optimal and strong bounds are known.

I am glad that this thesis can contribute to the research with an experimental comparison of tango, splay and red-black tree. The thesis does also introduce minor theorems with respect to tango trees performance against the upper bounds of the optimal offline binary search tree.

Lyngby, 19-June-2015

Theis F. Hinz

Acknowledgement

I would like to express my gratitude to my advisors Philip Bille and Inge Li Gørtz for providing competent academic advises. Your expertise and approach have been an important help for the progress of this thesis.

I also would give special thanks to Kristiane Brogaard Frederiksen for reading my thesis and helping correcting my misspellings. Your assistance has improved my exposition of the theory much.

Contents

Chapter 1

Introduction

Binary search trees are a category of data structures which have several benefits. Most interesting is its support of search operations, by using the property that data is stored in symmetric order by their key. An often asked question is how fast a search can be done.

The worst-case running time depends on the depth of the tree. Red-black trees, as well as others, are able to search in worst-case $O(log\ n)$ time by minimizing the depth of tree, where n is the number of nodes in the tree.
This is the best running time we can get for binary search trees in the worst-case scenario as there always will be nodes of $log\, n$ depth. But in other scenarios can better results be found. To study efficient binary search tree's further, there will be needed a stronger methods to analyze them.

This thesis will use the competitive analysis to investigate online searches [KMRS88]. There will be considered several searches which is executed on the data structure. An access sequence, $X = \{x_1, x_2, ..., x_m\}$, is the collection of these searches sorted by the time they are executed. It is not possible to chose the algorithm to use on the sequence beforehand as the algorithms is online.

A good binary search tree will be efficient for any given sequence. The analysis therefore consist of investigating how they competes against the offline data structure which is optimal for searching on the sequence. Let $OPT(X)$ be the running time of the optimal offline binary search trees for a access sequence X. A binary search tree is then *c-competitive* if its running time is within a c factor of $OPT(X)$ for any X. By using this analysis must a data structure perform well for any cases - also those which is not the worst-case scenario.

This leads to the following very strong property introduced by Tarjan et. al. in 1985 [ST85]:

Definition 1.1 (Dynamic Optimality Property) A binary search tree, D, is *dynamically optimal* if it for any X executes the access sequence in worst case $O(OPT(X))$.

The property is equivalent to being $O(1)$-competitive. It means in other words that its execution time for any given sequence is a constant factor from the best possible binary search tree that can be expected to exist. It is however an open question if such a data structure exists.

$OPT(X)$ depends of the access sequence X and the initial tree, T_0. Some states of the T_0 are better for some access sequences than others. Yet we know algorithms that can transform any binary tree of n nodes into any other tree in $O(n)$ time[CW82]. So by assuming $m = \Omega(n)$ can T_0 be ignored in the asymptotic analyses of data structures.

This report will look into what we know about an optimal offline binary search tree. There will in chapter 2 and chapter 3, be outlined the upper and lower bounds of $OPT(X)$. Afterwards, there will be examined two binary search trees in relation to this analysis.
Finally these binary search trees are compared by experimental results in section 6 in order to find out what tree are best in certain cases.

1.0.1 The Model

There will initially define the binary search tree model which there will be used throughout the thesis.
All nodes have a key which can be searched for. Nodes have an pointer to its left and right child (if existing). A pointer to its parent can as well be stored, if existing.
The nodes may store addition data, but they are not allowed to contain pointers to other data structures.

Searches will in the model always start by having a pointer at the root. It is then allowed to use the following operations:

- Move the pointer to the left child.
- Move the pointer to the right child.
- Move the pointer to the parent.

This thesis focuses on dynamic tree's. The model hence allow modifications (such as rotations) however they may only be conducted on the node at the pointer. A search succeed when the pointer is at the node with the key-value we searched for.

Variables and Assumptions

Throughout this thesis, n will denote the number of nodes in the binary search tree which there will be analyzed.
The access sequence of size m which is executed on the binary search tree is denoted as $X = \{x_1, x_2, \dots, x_m\}$. It is assumed, without loss of generality, that $m = \Omega(n)$.

We denote $OPT(X)$ to be the optimal running time of the execution of X.

To simplify the analysis, we will assume that the considered binary search trees of size n have the keys $\{1, 2, .., n\}$, however other can be used.

CHAPTER 2

Upper Bound

This chapter will consider a set of upper bounds of the optimal offline binary search tree. They each consider access sequences of specific characteristics for which binary search trees can perform better than the general worst-case running time of $O(log\, n)$.
The upper bounds can also be properties for those data structures which obtain the results of the bounds.

It is the goal to find bounds as close to optimal as possible, with the ideal of $\theta(OPT(X))$. In order to proof the tightness one approach may be to proof a factor between an upper and lower bound (see chapter 2.1).

Some of the bounds is generalization of another. An overview can be seen in figure 2.1.

The first considered upper bounds use distance in key space between the current and previous accessed node [Iac01].

THEOREM 2.1 (DYNAMIC FINGER PROPERTY) *Consider any key x_i in X where $i \geq 2$. x_{i-1} is then the previous key in the sequence. A binary search tree have the dynamic finger property if the search of x_i takes $O(log(|x_i - x_{i-1}| + 2))$ amortized time.*

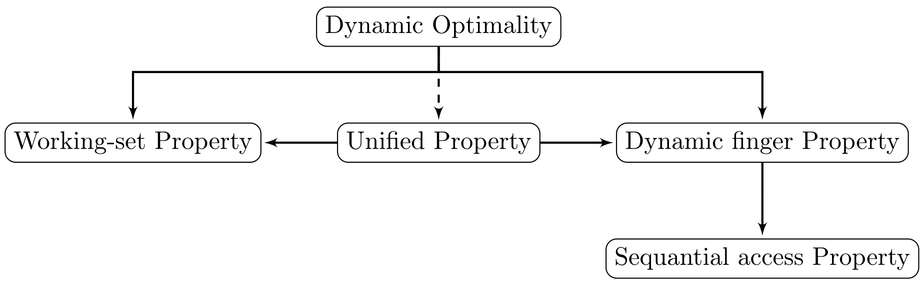


Figure 2.1: The hierarchy of the upper bounds. There is an arrow from one bound to another, if the first bound implies the second. A dashed arrow is implications which is not proven.

As an effect of this does a search take amortized $O(1)$ time, if all keys in X are a constant distance away in key space to their predecessor. There is a "+2" to make sure that the expression never is zero (you always have to pay a constant time).
The dynamic finger property thus implies theorem 2.2.

Theorem 2.2 (Sequential access property) *A binary search tree have the sequential access property if each search in the access sequence $X = \{1, 2, ..., n\}$ takes amortized $O(1)$ time.*

An search tree can also be efficient compared to searches distance in time, instead of space. A data structure with the working-set property is efficient if the time distance between searches of same keys is small. The working-set is defined to be the number of distinct searches in X. [Iac01]

Theorem 2.3 (Working-set property) *Let $t(z)$ be the number of distinct searches since last time a node z were accessed. A binary search tree have then the working-set property if the search of any x_i takes amortized $O(log(t(x_i)+2))$ time.*

A challenge is that there is no connection between a working-set and the dynamic finger property. John Iacono did in 2001 introduce a new bound which combines both bounds [Iac01].

Theorem 2.4 (Unified property) *Let $t(z)$ be the number of distinct nodes accessed since last time the node z were accessed. A binary search tree have then the unified property if a search of any node $x_i \in X$ takes amortized:*

$$O\left(log \min_{z \in X}(t(z) + |x_i - z| + 2)\right)$$

This property assures that a search on x_i is fast if there exists a node in X for which their position in X is close and their keys are close. Notice that the property equals the dynamic finger property if $t(z) = 1$. In similar way does it equal the working-set property if $x_i = z$. The unified property does thus imply both.

For the time being, no existing binary search tree has proved to have the unified property. There have even not been proved that this is a upper bound of $OPT(X)$. There is however made several pointer-based data structures which achieve the property. There will in chapter 5 be considered a binary search tree which is conjectured to have the property.

CHAPTER 3

Lower Bound

We want to be able to analyze how efficient a data structure is compared to an optimal offline binary search tree.

There will in this section be examined the known lower bounds for such a binary search tree. The lower bounds are interesting to consider, because any online binary search tree which is asymptotic as fast as the lower bound is dynamically optimal by definition 1.1. They can as well give a understand of what is not possible to achieve using the binary search tree model.

3.1 Wilbert's First Bound

Robert Wilbert did in 1989 prove the first lower bound for binary search tree [Wil89]. There will be analyzed a binary search tree, T, with the keys S. A lower bound tree of T is then a complete binary tree with leaves that each have a key in S (see figure 3.1). The nodes are ordered symmetrically.

This tree is only used in the purpose of the analyzing T. The tree is static, and its structure will thus not change over time.

For an internal node with a key y, there will be considered the subsequence

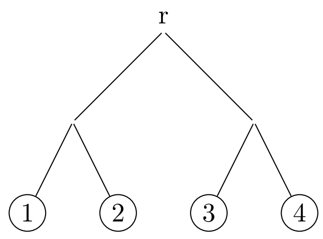


Figure 3.1: For Wilbert's first bound where $n = 4$. Consider the node r. For the access sequence $X = \{2, 3, 4, 1\}$ will $\lambda(r)$ then be 2.

$X' = \{x'_1, x'_2, ..., x'_h\}$ of X for which a node in the subtree's of y is accessed. An access x'_i in X' is defined as a *transition* of y if $x'_{i-1} < key(y) \wedge x'_i \geq key(y)$ or $x'_{i-1} \geq key(y) \wedge x'_i < key(y)$. This means that the access sequence alters between searching for a node in the left or right subtree of y.

Finally we denote the total number of transitions for an internal node y as $\lambda(y) = |x'_i \in X : x'_i \textit{ is a transition of } y|$.

Wilbert's first lower bound is then the sum of transitions for all internal nodes added to the size of access sequences, m.

Theorem 3.1 (Wilbert's first bound) *For a given binary search tree T, there is a lower bound tree with the internal nodes Y. Consider a access sequence X of size m which is executed on T. A lower bound of the optimal execution of X on T is then:*

$$m + \sum_{y \in Y} \lambda(y)$$

Using this lower bound, it can be observed that it is always possible to make a search for which there is a transition at all nodes on the root-to-node path of the searched node. It's search cost is thus worst-case no less than $O(log\, n)$ time.

3.2 Interleave Bound

One variant of Wilbert's first bound was introduced in 2004 for the purpose of easier application in the analysis of data structures[DHIP07].

The difference is that there is used an alternative lower bound tree, P, which is complete and of the same size as T. (see figure 3.2). A key of T, $S = \{1, 2, ..., n\}$, is now stored at each of the nodes, and not only at the leaves. The lower bound tree is still static.

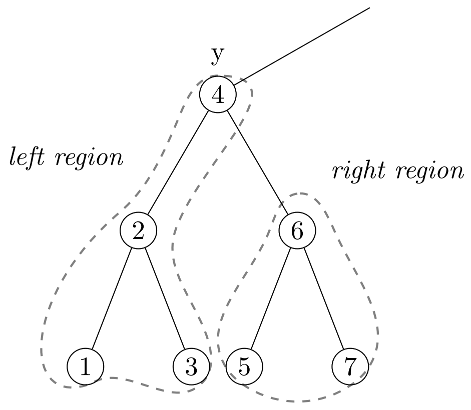


Figure 3.2: Lower bound tree P for the interleave bound. The left and right region is shown for a node y.

We define *transitions* and $\lambda(y)$ as described in section 3.1. Denote the set of internal nodes as Y. The *interleave* of an access sequence X is then defined to be:

$$IB(X) = \sum_{y \in Y} \lambda(y)$$

The interleave lower bound is then:

Theorem 3.2 (Interleave lower bound) *A lower bound of the execution of an access sequence X on any binary search tree T of size n is:*

$$IB(X)/2 - n$$

Regrettably, it becomes rather clear that this lower bound is far from optimal. Consider the root-to-leaf path of the lower bound tree, which contains the child which was last touched. A search of any nodes on this path would not result in any transitions. By the lower bound it should cost at least $O(1)$ time. However, as the path contains *log n* nodes, its is obvious that more that constant time must be used.

Now a proof will be given for the lower bound.

Proof of Interleave Bound

The following lines considers a node y in P and define its *left region* to be all nodes in its left subtree plus y itself (see figure 3.2). It similarly defines y's *right region* to be all nodes in its right subtree.

Let then the *transition point* of y be the node in T with the lowest depth for which its node-to-root path contains a node from both the left and right region.

Now it will be shown that such a node exists for all nodes in P at any time.

Lemma 3.3 *A transition point exists for any node y at any time.*

Proof. Define l to be the lowest common ancestor of all nodes of the left region of y in T (see figure 3.3). The keys of the nodes in the left region covers a subinterval of the key space spanned by T. l must thus be a node in the left region as a lowest common ancestor must be in the interval.
In a similar way can r be defined as the lowest common ancestor of all nodes of the right region. It follows by the same argument that works for l that r is a node in the right region. The nodes l and r is interesting to consider, as they must be visited when there is gonna be accessed a node in its corresponding region.

Lets consider the lowest common ancestor of all nodes in both the left and right region, q. Such a node must be a node in one of the regions, as the union of the keys in the regions covers a subinterval of all keys in the tree. As q is defined to be the node with the lowest depth, it must either be l and r.

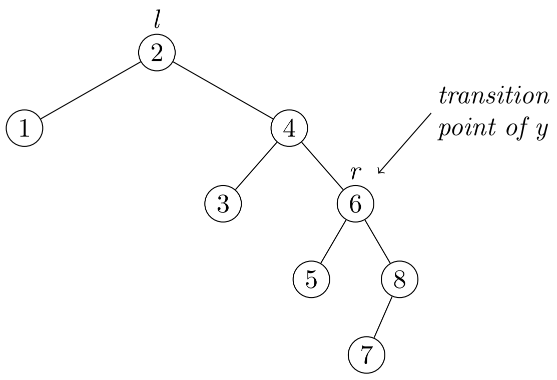


Figure 3.3: T at a certain time for P shown in figure 3.2.

The node which is not the lowest common ancestor of the nodes in both regions must then be the transition point of y. An access of the transition point must visit q and itself, which is nodes in different regions. □

There will for the rest of this section be continued to use the definition of l, r and q.
The transition point of y will not change in a sequence of accesses where the transition point is not touched. This is obvious as no modifications can be done at the transition point or on its subtrees, without accessing it (by definition of the model 1.0.1).
No nodes can thus enter or leave the subtrees of the transition node.

Lemma 3.4 *Let X be a sequence of accesses for which the transition point of a node y is not touched. Then is the transition point of y the same node during the execution of X.*

Finally it will be proved that nodes have a unique transition point. This implies that only one transition can happen when the transition point is touched.

Lemma 3.5 *A node in T is a transition point for at most a single node in P at any time.*

Proof. Let y_1 and y_2 be any nodes in P. It will now be shown that these elements can not have the same transition point in T by considering two cases. The first case is that y_2 is in the subtrees of y_1. The transition point of y_1 can then be a node in the regions of y_1 or not. If it is not then l and r is distinct nodes. If it is, then we would have that the transition point of y_1 is the lowest common ancestor of all nodes in the regions of y_2. But this is the same as q of y_2 and their transition points are thus different. By symmetry can it be shown that their transition points are different for the case there y_1 is in the subtree of y_2.
The second case is that neither y_2 and y_1 is in the subtree's of each other. In this case their regions is distinct and thus the transition point is not the same node. □

The interleave bound (theorem 3.2) can now be proved by the use of three lemmas above:

Proof of Theorem 3.2. We proof this by counting the transitions points which the sequence must touch.

Consider the subsequence $X' = \{x'_1, x'_2, ..., x'_{\lambda(y)}\}$ of X for which a transition happens on a node y. Every second search in X' must access a node in the left region of y, and thus touch l. Similarly, every second search must access a node in right region and thus touch r. The transition point of y is either l or r. The point may change between being l and r but this requires touching the transition point (lemma 3.4). It follows that the transition point of y must be touched at least $\lambda(y)/2 - 1$ times.
There can now be summed over the number of touches for transition point in P. This is okay, as lemma 3.5 proves that the transition points is unique for each node in P. Let Y be the set of nodes in P. The interleave bound then follows:

$$\sum_{y \in Y} \lambda(y)/2 - 1 = IB(X)/2 - n$$

□

3.3 Other Bounds

A second lower bound was also given by Wilbert [Wil89]. The bound has not yet been found usable in practice. However, it have several benefits, for instance it does not depend on a lower bound tree. Furthermore, it is closer to the optimal running time, but it have never proved to be more than a constant factor better that his first bound (which we just have considered) [CD09].

Two other lower bounds exist which are closely related to each other, The Independent Rectangle Lower Bound [DHI+09] and The Rectangle Cover Lower bound [DDS+05]. Both uses a geometric view of tree's and is conjectured to be a constant factor from each other [CD09]. Both is proven to be at most as high as Wilbert's second lower bound.

Chapter 4

Tango Tree

This chapter will consider a binary search tree which is the first to be proved $O(log\,log\,n)$ factor from $O(OPT(X))$. It is also called $O(log\,log\,n)$-competitive. This is an analytical improvement as regular trees are only known to be $O(logn)$-competitive.

Tango trees were proposed in 2004 [DHIP07] and use the interleave bound (see section 3.2) to achieve their result. The main idea is keeping track of the state of the lower bound tree, P.

Consider a P where each node is augmented to know which child was previously accessed (see figure 4.1). We denote this child as its *preferred child*. By convention a node have a left preferred child if it was the node which their previously was searched for. A child that has not previously been accessed is called the *non-preferred child*. Initially no nodes are preferred.

The preferred children may change in order to be up-to-date on what nodes were previous access. There is an important connection between the changes of preferred children and the interleave $IB(X)$ (as defined in section 3.2). Let us now consider lemma 4.1.

Lemma 4.1 *The number of times a node alternate between having a left or right preferred child is equal to the interleave, $IB(X)$, for the access sequence.*

Proof. A preferred child is the left child given that the previous access was in the left region. Similarly, if the previous access was in a node in the right region, the preferred child is also right. A node can only have one preferred child. So a change in which region was previously accessed implies a change of the preferred child and the other way around. The lemma then follows. □

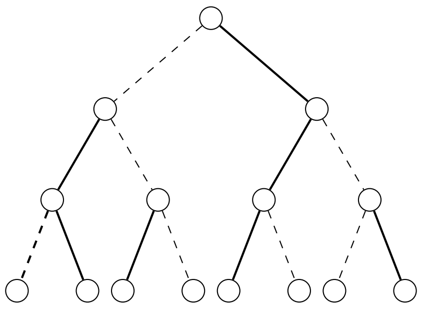


Figure 4.1: Lower bound tree, P. A node have a thick edge to its preferred child, and a dashed edge to its non-preferred child.

Let *the preferred paths* be the paths for which you start at a non-preferred node and move by edges to preferred children. A preferred path have $O(log\ n)$ nodes as P is balanced.

For tango trees to be $O(log\ log\ n)$-competitive the following is required:

Search on preferred path A search which only accesses nodes on a single preferred path would not make any transitions (see section 3.1). It then follows, by the definition of the interleave bound (theorem 3.2), that tango tree's may use worst-case $O(log\ log\ n)$ time.

Access non-preferred child A transition is made for each non-preferred child which is accessed. Thus, the addition of $O(log\ log\ n)$ time is allowed in order to access nodes in the preferred path of which the non-preferred child is in.

Please note that tango trees do not support insertion and deletion of nodes. This is due to the limitation that the interleave bound requires the lower bound tree to be static.
Also, a tango tree is a self-adjustable tree. There is therefore no guarantee that a tango tree is balanced.

4.1 Auxiliary Tree

There is an exact correspondence between the nodes in the lower bound tree and the tango tree. Tango trees store each preferred path of the lower bound tree in an augmented red-black tree, which is denoted *Auxiliary tree.* The augmentation will be described later on.

In P, an edge between a node and its non-preferred child corresponds to an edge between two preferred paths. Such an edge is in T represented by a pointer (stored as an edge) that connects the auxiliary trees which describes the two preferred paths.

Let A_1 and A_2 be the preferred paths representing the auxiliary trees mentioned above. A_2 will then be inserted into A_1, without rebalancing the tree. By doing this A_2 is stored at a leaf of A_1. The connections to all the auxiliary trees make up a single tree T (the tango tree) as shown in figure 4.2.
The root of the tango tree is the root of the auxiliary tree which contains the root of the lower bound tree.

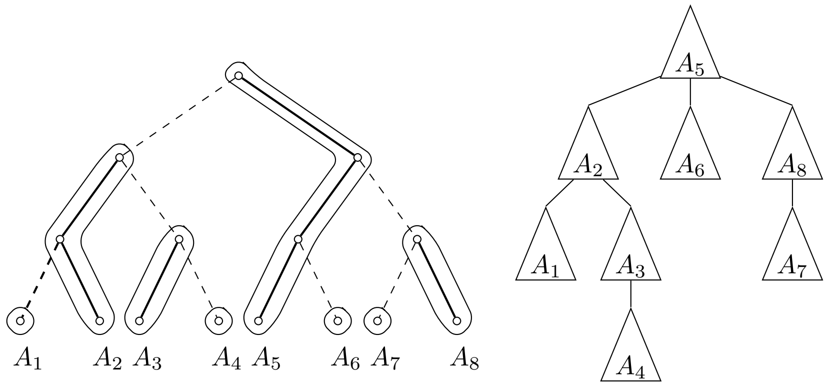


Figure 4.2: On the left: Lower bound tree with highlighted preferred paths. On the right: The Tango tree with its auxiliary trees shown.

Each auxiliary tree will still be treated as an individual tree. It is thus necessary to be aware where the auxiliary trees are. In order to do so, an additional bit is stored on each node which decides if it is a root of an auxiliary tree or not. Note that a tango tree complies with the binary search tree model though it actually has several trees in it.

Additional Data on Nodes

The auxiliary trees are augmented to store additional information about the lower bound tree for T. The data is used by the search algorithm.

First, let each node, v_i in T store the depth of its corresponding node in P. This depth will be denoted $d_P(v_i)$. Each v_i is then augmented to know the maximum and minimum depth (in P) of all nodes in its subtree of the auxiliary tree.

These values must be updated each time a node in T is modified (such as rotated) so that they are always up-to-date. This is simply done without changing of the asymptotic running time as only the nodes on the node-to-root path need to be updated [CLRS09].

4.2 Tango Search Algorithm

The search algorithm for tango trees will now be considered. The algorithm starts by having a pointer at the root and moves the pointer like a classic binary search tree.

The pointer may meet the node u, which is characterised by having an edge to an other auxiliary tree, and it is possible that the pointer moves by the edge into the new auxiliary tree. Such an access will change the preferred path in the corresponding lower bound tree of T. The change of the preferred child in P can be described as in figure 4.3: Firstly, the path cuts into two: One with depth more than $d_P(u)$, and one for the rest. Secondly, the path of the lowest depth is joined with the path which contains the non-preferred child.

The auxiliary trees are intended to always represent a preferred path regardless of the changes that are made during searches. Thus, there will be made changes to the auxiliary trees that are similar to what is done to the preferred paths of the lower bound tree. The cut- and join- step will be described in the next sections.

Finally, the root and the node you search for is in the same auxiliary tree. The search thus ends by finding the wanted node by a regular binary search.

The next sections will consider a transition at a node u. The segment of the preferred path with greater depth than d is denoted D. The preferred path which contains the non-preferred child of u is denoted D'.

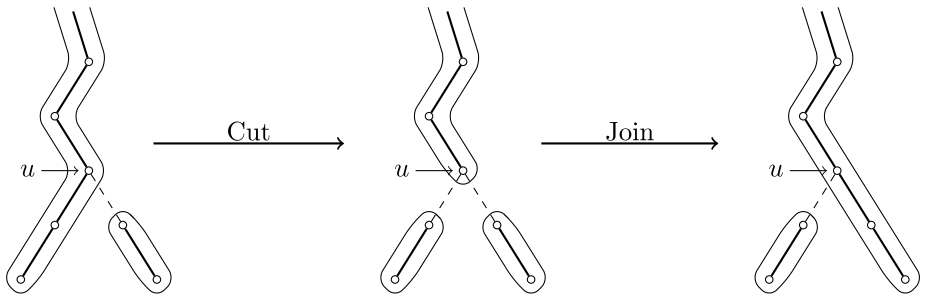


Figure 4.3: The steps of how the preferred path is changed by the transition at u. Notice that the non-preferred children is not shown, except for u.

Cutting Auxiliary Trees

The cut operation turns the auxiliary tree, which contain u, into two trees. The first auxiliary tree A contains all nodes which have a depth in P that is less or equal to $d_P(u)$. The rest of the nodes, D, is in a separate auxiliary tree underneath. The cut operation correspond to turning a preferred child in P into a non-preferred.

To do so the cut operation uses the split and concatenate function:

$split(A, k)$**:** The algorithm takes a tree A and a key k in A. Split then modifies the tree so that the root have the key k. Its left subtree contains all nodes with smaller keys and its right contains the nodes with bigger keys.

$concat(v_k, A_1, A_2)$**:** The concatenation takes a node with key k, and a tree A_1 whose nodes have keys less than k and another tree A_2 whose nodes have keys higher than k. The algorithm then returns a single tree containing all nodes of A_1, A_2 and v_k.

Robert Tarjan proves that such an algorithm exists for Red-Black Tree with worst-case running time of $O(log\ n')$ where n' is the number of nodes in the tree [Tar83].

Now consider how these operations can be used to do a cut operation. Let l' be the node with the biggest key smaller than all keys in D. Similarly, r' is the node with the smallest key which is bigger than all keys in D. All keys between these adjacent keys are in D as a path covers all keys in a interval of the key space. It is later shown how to find these nodes.

Assuming these nodes can be found we are able to do the cut operation as follow (see figure 4.4):

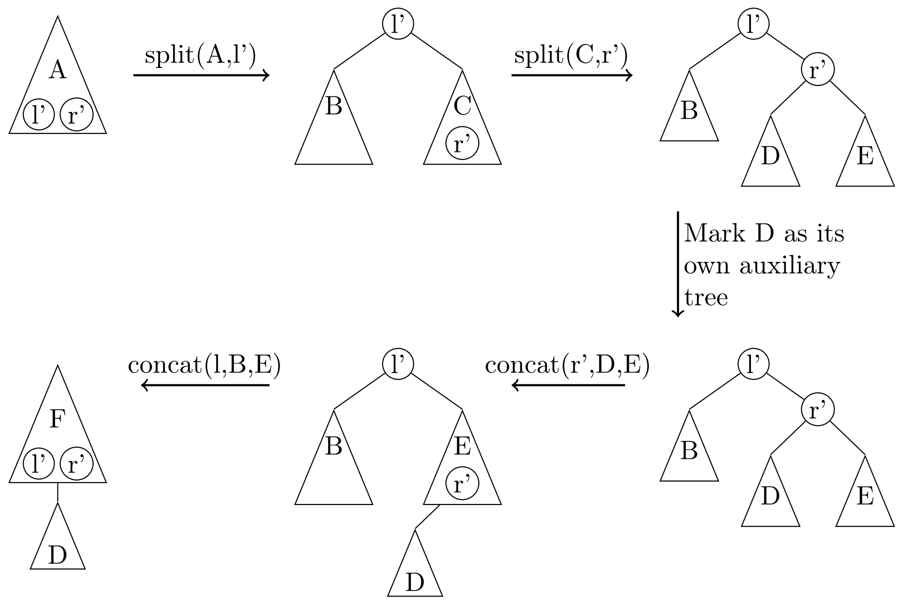


Figure 4.4: Cutting a tree A' into two trees: A and B. The figure is based on a figure from the original paper [DHIP07].

Let the tree be split on the key of l'. Then let the right subtree of the root be split on the key of r'. The left subtree of r' now contains all nodes between l' and r' and must therefore have a depth in P lower than $d_P(u)$. It is thus D.
Let the subtree be its own auxiliary tree by changing the bit of its root so that it represents the root of a new auxiliary tree.
Let r' and its subtrees be concatenated and let thereafter l' and its subtrees be concatenated. This will result in an auxiliary tree with D in another auxiliary tree underneath.

Finding l' and r'

In order to find l' and r' fast a classic search can not be used as the nodes are ordered by their key and not depth. It will now be considered how to find l'. The approach is to find the left most node in D, denoted l. l' is then its predecessor.

First, let d be the lowest depth of the nodes in D. D and D' are both subtrees of u, and the value of d can therefore easily be found as it is equivalent to the lowest depth of D'. Let v be the root of the auxiliary tree to join. d is then either the depth of v or lowest depth in the subtrees of v. Both values are stored on v, and d can be found as by identifying the lesser value.

Knowing d we can find l by following the left most path from the root for which the nodes or their subtrees have depth greater or equal to d.

r' can be found by a similar method by symmetry.

Joining Auxiliary Trees

When two auxiliary tree are joined it turns into a single auxiliary tree containing all the nodes. This is done the same way as cut is done. Once again there can be found an value l' and r' which is adjacent to D'. It requires two split operations in order to have D' in its own subtree of T. The bit on root of D' is then changed so that it is not the root of an auxiliary tree. Finally, two concatenations can turn it into a single balanced tree again.

Figure 4.5 shows the keys of D and D' on a number line. It is obvious that u is either l' and r' during a join as D' is a subtree of u. By symmetry must u either be l' and r' for the cut operation.

This means that there can be one split and one concatenation less if cutting and joining is done at the same time. This means that a join and cut operation can be done by three splits and three concatenations in total.

4.3 Analysis

This section analyzes how tango trees performs. Initially it will consider tango trees worst-case running time as a function of the changes between nodes preferred child.

Theorem 4.2 *The worst-case running time of a tango tree with n nodes is $O((k+1)(log\ log\ n))$ where k is the number of times a node changes its preferred child.*

Proof. By design there is a exact correspondence between a preferred path and an auxiliary tree. A search must thus access nodes from $k+1$ auxiliary trees.

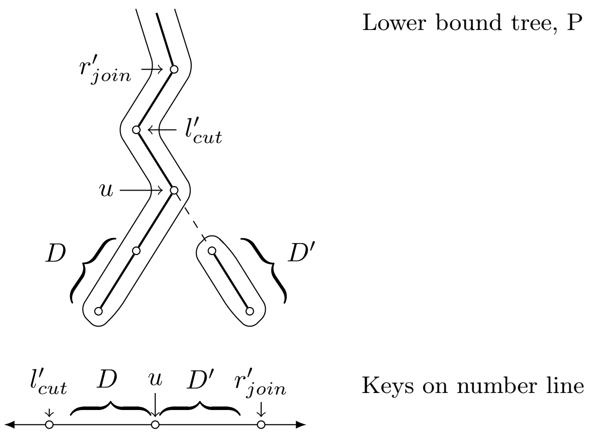


Figure 4.5: Transition of u shown in the lower bound tree, P. The keys are plotted on a number line beneath.

The "+1" is added because there always are accessed nodes in the auxiliary tree containing the root.
A search in an auxiliary tree takes $O(log\ log\ n)$ time as it is an balanced tree with $O(log\ n)$ nodes.

Furthermore, time is spent on updating the trees k times such that they continues to represent the preferred paths. Each update contains a cut and a join operation. This is a constant number of split and concatenation operations and the flip of a bit on two nodes (one to indicate that its a root, and one to indicate that another node is not a root anymore). The auxiliary trees are of size $O(log\ n)$. Hence this takes $O(log\ log\ n)$ time.
Finally, time is spent locating l' and r' which will be used for the cut and join operations. For each of them a simple search is done (where depth is taken into count) and one operation of respectively finding a predecessor or a successor. This is of $O(log\ log\ n)$ time.
The running time must hence be $O((k+1) \cdot log\ log\ n)$. □

In the worst-case scenario can a search execution be touching $O(log\ n)$ non-preferred children. Thus, there exist sequences where each search takes $O((log\ n) \cdot (log\ log\ n))$. This is larger than the most known self-balancing data structures.

Let us now prove tango trees competitive running time.

Theorem 4.3 *Let it be assumed that $m = \Omega(n)$. The execution of a search sequence X takes at most $O(OPT(X) \cdot (log\ log\ n))$ for tango trees.*

Proof. It takes at most n accesses to let all nodes (which is not leaves) have a preferred child. By theorem 3.2 the running time of X is then:

$$O((k + n + m)\ \cdot (log\ log\ n)$$

By lemma 4.1 it is known that $k = IB(X)$. The expression can thus be rewritten as:

$$O((IB(X) + n + m)\ \cdot (log\ log\ n)$$

It can now be deducted that the interleave bound states that $IB(X)/2 - n \leq OPT(X)$. m is always less than $OPT(X)$ (as the node search for must be touched) so the execution takes at least:

$$O((OPT(X) + n)\ \cdot (log\ log\ n)$$

The theorem then follows when the assumption is applied that $m = \Omega(n)$. □

4.3.1 Performance against Upper Bound Properties

Tango trees are closer to optimal than most. But unfortunately, tango trees still do not perform well against the upper bounds described in section 2. This thesis applies the properties to the data structure and finds that tango trees have neither of the properties.

Lemma 4.4 *The execution of the sequential access sequence $X = \{1, 2, ..., n\}$ take $O((n + 1)(log\ log\ n))$ time for tango trees.*

Proof. The strategy is to count the number of alternations between nodes the preferred child when all n nodes is accessed. For every internal nodes must there be one access in the left region and one in the right region. In total are there $O(n)$ changes of preferred children. A sequential access does then use $((n + 1)(log\ log\ n)$ time. □

This is *log log n* of the results for the sequential access sequence. The result does also show that tango trees do not have the dynamic finger property as it is a generalization of the sequential access property.
Lets us consider the working-set property.

Lemma 4.5 *Tango trees does not have the working-set property.*

Proof. The result is found by a disproof. Consider an access sequence with a constant sized working-set. Let every access be on a key of a leaf in the top auxiliary tree. Each search is then equal to a regular search in a red-black tree. Every search then takes $(log\ log\ n)$ time. □

A tango tree does also not have the unified property as it would require that a tango tree also have the working-set and dynamic finger property.

4.4 Tango Inspired Trees

It is a challenge for tango trees that the auxiliary trees use red-black trees where nodes keep a fixed depth. Section 4.3.1 shows that tango trees does not have the working-set property by searching for nodes at $\Omega(log\ log\ n)$ depth.

One idea would be to replace red-black trees with a self-adjusting binary search tree This is possible to do as long as the search, split and concatenation operation is supported.

This is done by for instance multi-splay trees which use splay trees instead [WDS06]. The search algorithm is nearly the same but achieves some better results: Each search takes $O(log\ n)$ amortized time and it is proved to have the working-set property [DSCW09].

CHAPTER 5

Splay Tree

A splay tree is a self-adjusting binary search tree which is proved to have very good results in respect to the upper bounds of the optimal offline binary search trees (see chapter 2). The data structure was introduced by R. E. Tarjan and D. D. Seator [ST85].
It has proven to have the working-set property, dynamic finger property and sequantial access. It is even suspected to have the unified property. Its running time is amortized $O(log\ n)$.

Most interesting is that it is conjectured to be dynamic optimal. However, it is still not proven being any better than a $O(log\ n)$ factor from optimal.

The data structure and search algorithm is very simple however its analysis i very complicated. There will in this chapter first be described the data structure and afterward there will be analyzed its asymptotic running time.

5.1 The Data Structure

The data structure is a clean binary search tree. This means that no additional data is stored at the nodes.

5.2 Search Algorithm for Splay Trees

A search on a key x_i uses a regular binary search in order to find the node. When the node is found, an algorithm called *splay* is applied in order to move the node to the root.

The idea is to keep nodes which have recently been accessed close to the root and in the mean time reduce the depth. Doing this will make future searches on these nodes faster.

Splay

The splay algorithm uses rotations in order to move x_i to root. For every step the node, its parent and grandparent (if existing) is considered. This gives us three cases (and three mirrored cases which is handled similarly). The cases are named *zig*, *zig-zig* and *zig-zag*, and they are shown in figure 5.1.

Each of the steps described above reduce the depth of the subtrees of x_i. Notice that the splay-operation is simply rotating x_i besides for the zig-zig case. This is because the constantly rotating of x_i will not reduce the depth of the subtrees of x_i in this particular case. The steps is conducted repeatedly until x is the root (see figure 5.2).

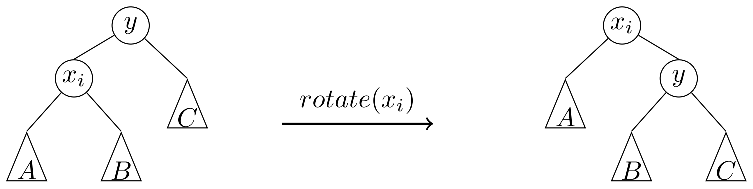
The zig-case (see figure 5.1a) happens no more than once per splay. This is if x_i is a child of the root and therefore do not have a grandparent.

Each step takes $O(1)$ time as they contain no more than two rotation and each of these changes a constant number of pointers.

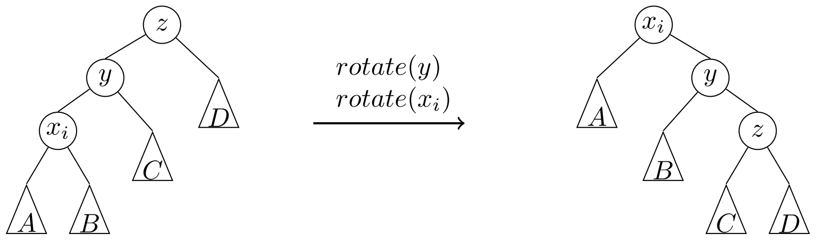
5.3 Analysis

In order to analyze its amortized running the potential function is used. Every node, v, will thus be given a potential which is denoted as its *rank*, $r(v)$.

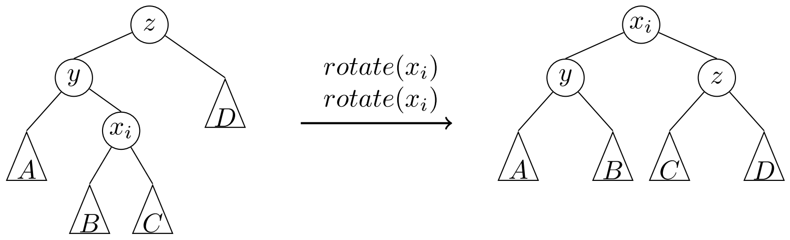
Consider figure 5.3. The weight $w(v)$ of a node v is defined as an arbitrary number. It is equal for all nodes and will not change through time for this specific analysis. The size $s(v)$ of v is then the total weight of all nodes in v's



(a) Zig case.



(b) Zig-zig case.



(c) Zig-zag case.

Figure 5.1: The three cases for splaying x_i. The figure is based on an illustration from the original splay tree paper [ST85].

subtrees and v itself. The rank is then equal to:

$$r(v) = \lfloor log(s(x)) \rfloor$$

Rank Rules

This section will clarify two minor lemmas about nodes rank which are called the Rank Rules [Sle02]. These will later on be used for the prove of the amortized running time.

Lemma 5.1 (Rank Rule 1) *If two siblings have the equal rank r then their*

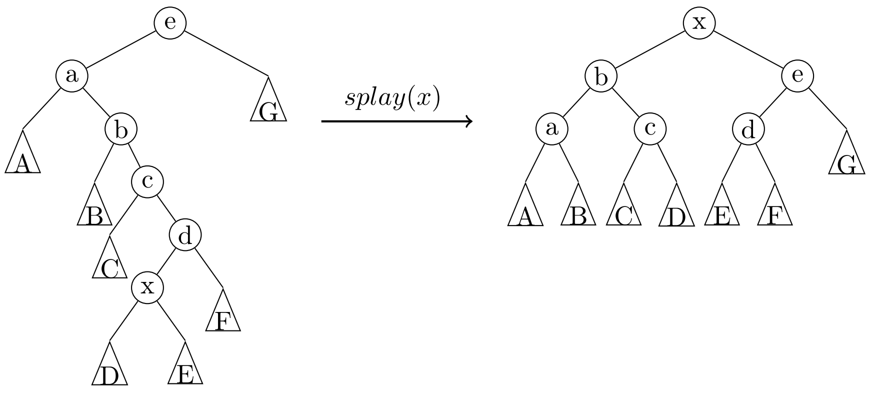


Figure 5.2: The splay of the node x. Firstly, a zig-zag case is executed, then a zig-zig case and finally a zig case.

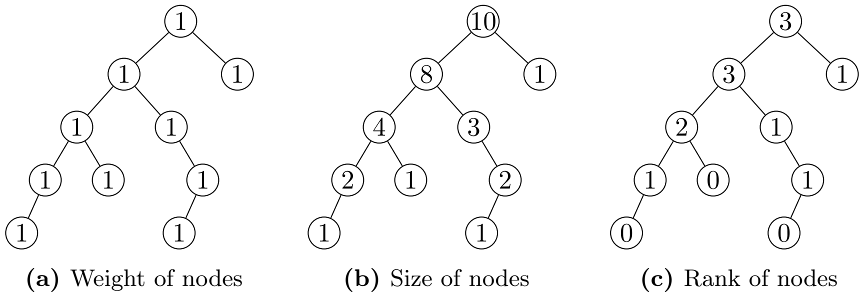


Figure 5.3: The weight, size and rank of nodes in a tree. All nodes in this example have a weight equal to 1.

parent must have a rank higher than r.

Proof. The two siblings must each have a size of minimum 2^r (by definition of size). Their total sizes are therefore at least $2^r + 2^r = 2^{r+1}$, and their parents must thus have a rank of $r + 1$ or higher. □

Lemma 5.2 (Rank Rule 2) *Consider a node v' with the two children v_1 and v_2. If v' and v_1 have the equal rank r then v_2 must have a rank lower than r.*

Proof. v' can have a size no larger than $2^{r+1} - 1$. On the other hand, the size of v_1 is at least 2^r. The largest size which v_2 can have is thus $2^r - 1$. The rank is therefore lower than r. □

Running Time Analysis

Let there now be considered the running time of the splay tree. The access lemma describes splay trees' amortized running time by applying the rank function.

Lemma 5.3 (Access Lemma) *A splay tree with root t has the amortized running time $3(r(t) - r(x)) + 1$ for splaying a node x.*

Proof. The approach is to consider each step of the splay-algorithm (see figure 5.1) one by one. Let $r'(x)$ be the rank of the node to splay after the step. The prove is then to show that each step costs at most $3(r'(x) - r(x))$ beside for the zig-case where an additional constant may be used. The total costs of all steps is $3(r(t) - r(x)) + 1$. Notice that the "+1" is from the zig-case which occur only once or not at all.

Let there now be considered the 3 steps. The parent and grandparent of x is denoted respectively as y and z.
Notice that the rank of these nodes changes during a step, as other nodes' subtrees stay the same. These nodes are therefore only considered. The nodes never have a rank higher than $r'(x)$ during the execution of a step as x is the node with the highest level. It is thus enough to assure that there are $r'(x)$ tokens for each node which increases its rank.

Zig Case
There is paid 1 for the actual cost of the step. The new rank of x is covered by $r(y)$ as they are of equal value. $r'(y)$ is covered by the tokens of $r(x)$ and by letting additional $r'(x) - r(x)$ be paid. This is enough tokens as $r(x) + r'(x) - r(x) = r'(x)$.
The total cost is $r'(x) - r(x) + 1$ which is less that $3(r(t) - r(x)) + 1$.

Zig-zig case
Two situations are considered for for this step: Either the step changes the rank of x or not.
The first situation (the rank does not change) can be illustrated as in figure 5.4.

The rank of a node is written above the node on the figure as a variable. After the first rotation is z a child of y and a sibling of x. By lemma 5.2 the rank

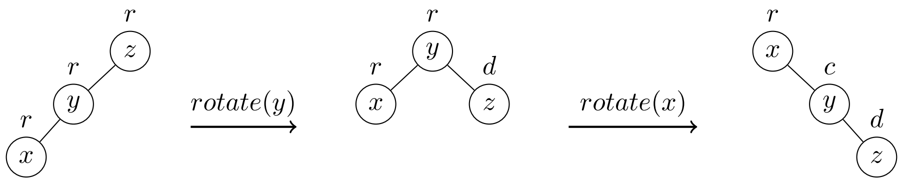


Figure 5.4: Zig-zig case when the rank of x does not change. Based on a figure from [Sle02].

of d must have decreased. The subtrees of z does not change during the next rotation so their rank stays decreased.
So there is release at least 1 token from potential (as $r'(z) - r(z) \leq 1$), and this can be used for paying the actual cost.

The second case (the rank does change) can be seen in figure 5.5.

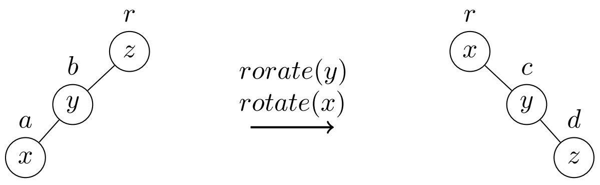


Figure 5.5: The zig-zig case when the rank of x changes. Based on figure from [Sle02].

The increase of x's rank can be covered by the rank of z as $r(z) = r'(x)$.
To cover y and z's rank are there paid for additional $r'(x) - r(x)$ tokens for each to increase $r(x)$ and for $r(y)$ to be equal to $r'(x)$.
Additional $r'(x) - r(x)$ (which is at least 1) is spent for doing the actual job.
The total cost is $3(r'(x) - r(x))$.

Zig-zag Case
Once again, two situations are considered: The rank of x changes during the step or it does not.
The first case (the rank does not change) is illustrated in figure 5.6. In this situation is y and z children of x after executing the (zig-zag) step. Rank Rule 2 says that they cannot both have the same rank as x (lemma 5.2). Therefore, at least one token is released which is used for doing the job.

For the second situation (where x changes its rank) the rank of the nodes can increase.
$r'(x)$ is covered by the tokens of $r(z)$ as they have the same value. Similarly,

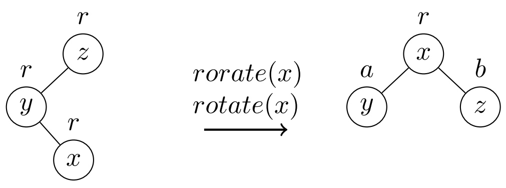


Figure 5.6: Zig-zag case where the rank of x do not change. Based on figure from [Sle02].

$r(y) \geq r'(y)$, so the rank of y can be covered by them.
$r'(x) - r(x)$ is paid in order to increase $r(x)$ enough to cover the tokens for $r'(z)$. Additionally $r'(x) - r(x)$ is paid (which is at least 1) for the actual cost. The total cost is $2(r'(x) - r(x))$.

The zig-case has now been proven to cost no more than $3(r'(x) - r(x)) + 1$ and $3(r'(x) - r(x))$ for the two other cases $3(r'(x) - r(x) + 1$. The prove is thus done. □

If the weight of all nodes is set to be 1 then the root has a rank of $r(t) = log\ n$. A leaf has a rank of 0. The asymptotic running time is thus worst-case:

$$3(r(t) - r(x)) + 1 = 3(log\, n - 0) + 1 = O(log\ n)$$

Chapter 6

Experimental Comparison

In this chapter tango, splay and red-black trees are compared experimentally with different access sequences.

These accesses sequences are among other reasons chosen in order to show their performance against the upper bounds described in section 2.

The trees are implemented in Java and are constructed in such a way that the code is reused between the data structures. Figure 6.1 shows a class diagram of the model representing the nodes. The class *BSTNode* represent the basic model of binary search trees as defined in section 1.0.1.

The splay tree does not use any additional data and is thus using *BSTNode* as its model.

Red-black trees store an additional bit on the nodes to represent their color. The class *RedBlackNode* inherits the *BSTNode* class and adds the additional information to the model.

Tango trees extend red-black trees to store additional augmented data. Their model therefore inherit from the model of red-black trees. Other trees do exist that are inspired by the tango trees and use other data structures than red-black trees (see section 4.4). Therefore, this project use an interface, so that the tango

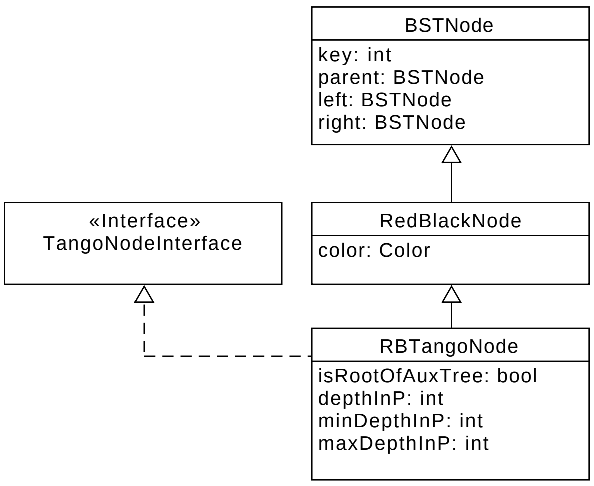


Figure 6.1: Classes representing nodes of the data structures.

tree model can easily be exchanged to other models with the same interface.

The classes representing the trees that are structured in a similar way (see figure 6.2). An abstract class named *BST* is used to describe the public interface and the operations that are shared between the data structures (such as rotation).

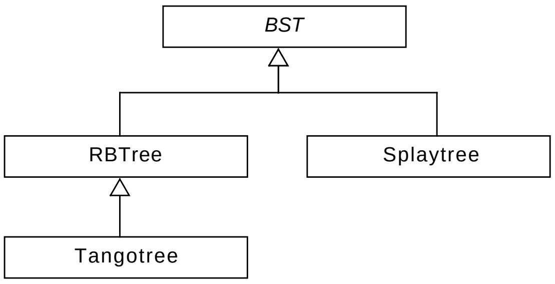


Figure 6.2: Classes representing the trees.

How the Trees are Compared

The experimental execution are conducted by the class *TimeTest* which use the class *SequenceGen* to generate the access sequences.

All trees have their n nodes inserted $\{1, 2, ..., n\}$ in random order. Notice that there might be minor changes in performance if the nodes are inserted in a different order. For instance will the splay tree initially be an unbalanced chain if the nodes are inserted in sequential order. However, the difference in performance should be small as long access sequences are applied.

For tango trees a search on each node of the tree is made prior to the experiment. By doing this does all internal nodes in the corresponding lower bound tree have a preferred child. At this point should the tree perform a $O(log\ log\ n)$ factor from optimal, $OPT(X)$.

The trees are compared by two parameters: The time to execute the access sequence and the number of nodes which are touched during this execution.

A data table with the experimental results can be found in the appendix.

6.1 Dynamic Finger

In order to compare the data structures performance against the dynamic finger property (see section 2) are the following access sequence applied:

$$\{1, 1 + 1i, 1 + 2i, ..., n, (n - 1i) + 1, ...\}$$

This is an access sequence where the previous search always is the distance i away. An exception is when the sequence reaches n for which a search of distance $i - 1$ is taken. This is done in order to assure a working-set of size $O(n)$.

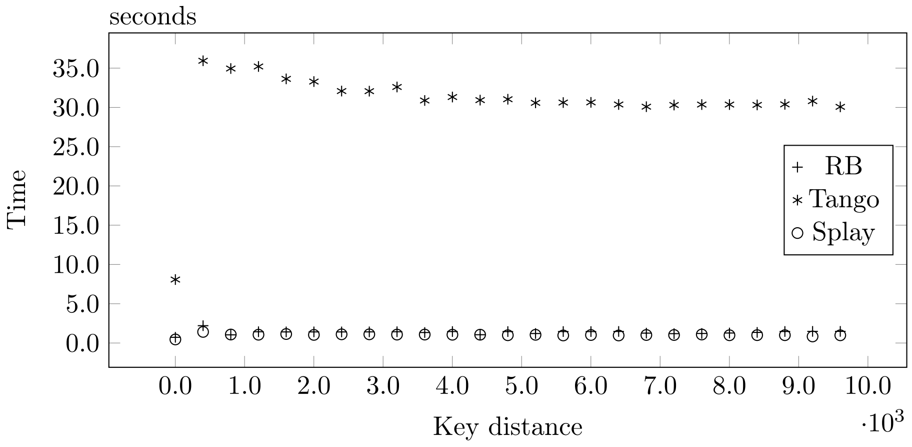
A data structure which have the dynamic finger property should be able to execute each step in amortized $O(log\ i)$ time.

Figure 6.3 plots the result with different key distances, i. A tree is applied which have the size $n = 2500000$, and the length of the access sequence is $m = 5000000$.

The graphs show that the running time of the red-black tree is about constant when i change. This is expected as its running time is independent on i (the depth of the tree is never changed).

Splay trees are proved to have the dynamic finger property. It is thus expected that the running time is increased logarithmically when the key distance grows. The experimental results confirm this.
Notice that the experiment has found that splay trees are the best performing data structures when the key distances is small.

In the experiment, tango trees perform significantly worser than the other data structures. However its performance improves when the key distace grows. We have made following hypotheses for why this happens:
When a search are made will the next accessed nodes be the distance i away. This means that preferred children are unchanged in a large subtree which grows with i. The preferred paths will stay the same in this subtree. At a certain point will the sequence again access nodes in this tree. These searches should thus only follow few non-preferred children in order to succeed.



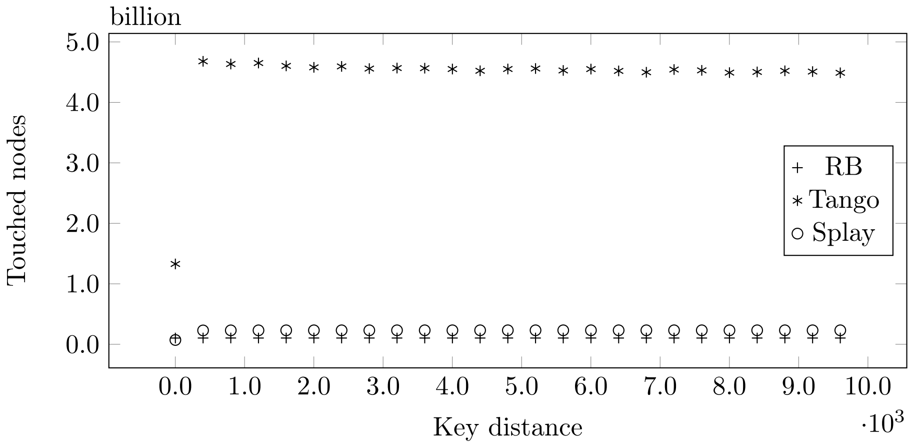
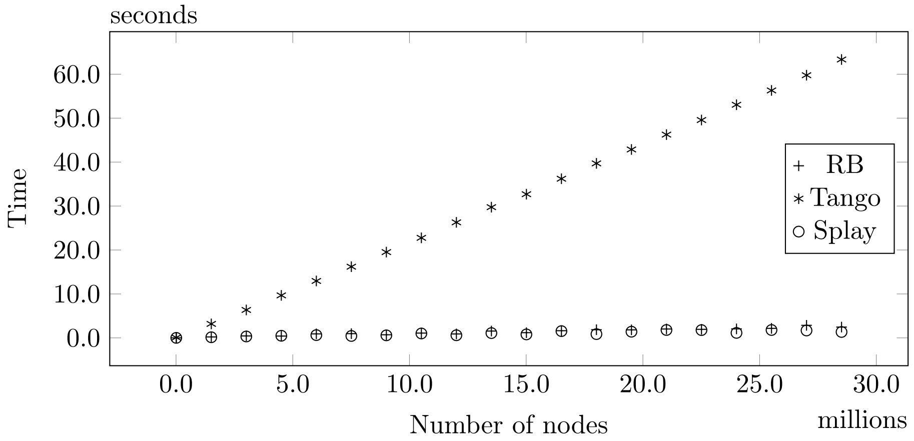


Figure 6.3: Dynamic finger experiment. The upper plot shows the time used for executing the sequence. The lower plots show the touched nodes. For the experiment are $n = 2500000$ and $m = 5000000$.

6.2 Sequential Access

A data structure with sequential access property should be able to visit all n nodes in $O(n)$ time. Figure 6.4 compares the performance when the sequential access sequence $\{1, 2, 3, ..., n\}$ is executed with different values of n.



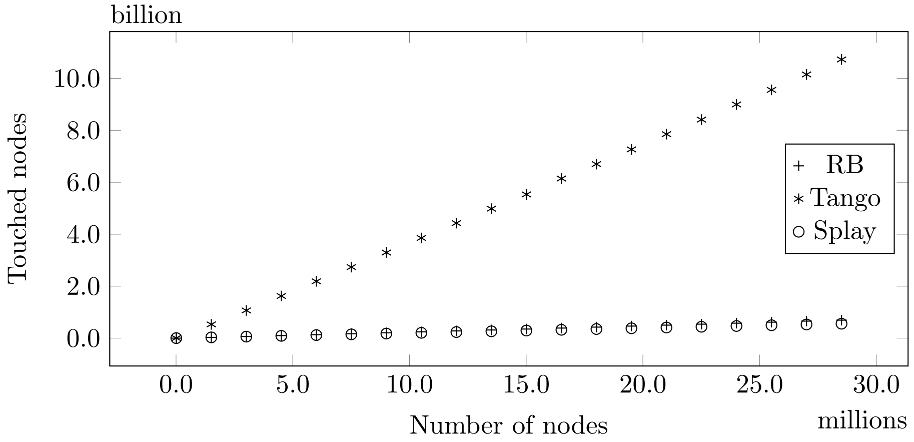


Figure 6.4: Sequential Access experiment. The time usage (upper plot) and the touched nodes (lower plot) for the experiment is shown.

The following is an explanation of the result. Splay trees does have the sequential access property which the experiment also shows as the time increases linearly.

Red-black trees are expected to grow by $O(n\ log(n))$ as m equals n. Our observations confirms this (examine data table in the appendix A.2 for better examination).

Tango trees time usage expected to grow by $O((n+1)(log\ log\ n))$ (see prove for lemma 4.4). This is also the case for the experimental results.

Notice that red-black trees are found to be the best to perform when n is small. This can be explained by the fact that red-black trees don't pay a high cost for adjusting the tree when searching. But when the number of nodes increases are this cost less dominating and splay trees therefore perform best. In none of the cases the tango tree appear to be the best.

6.3 Working-set

This section describes an experimenting comparison of the data structures for variating sizes of the working-set, t (see section 2). This is done by executing an access sequence in which a search for the same key is conducted with a frequency of t.

Figure 6.5 plots the performance for red-black trees and splay trees. For the experiment a tree is used which has $3,000,000$ nodes and an access sequence of length: $10,000,000$. The plots show the data series *RB* and *Splay* which are the execution where random keys are chosen for the working-set.
In addition the best and worst-case scenario shown for red-black trees. This best-case situation is for red-black tree to have a working-set where the keys are of the nodes closest to the root. The worst-case situation is that the working-set contains keys which are all stored on the leafs.

The number of touched nodes can easily be explained by the theory. The time usage for the worst-case situation stays constant as it is bound by its worst-case running time of $O(log\ n)$ where n stays constant. For the best-case situation the time usage grows logarithmically by the size of the working-set. This is because the accessed nodes have a maximum depth of $O(log\ t)$.

Splay trees are proven to have the working-set property and should touch $O(log\ t)$ nodes. This seems to be the case for the experiment.

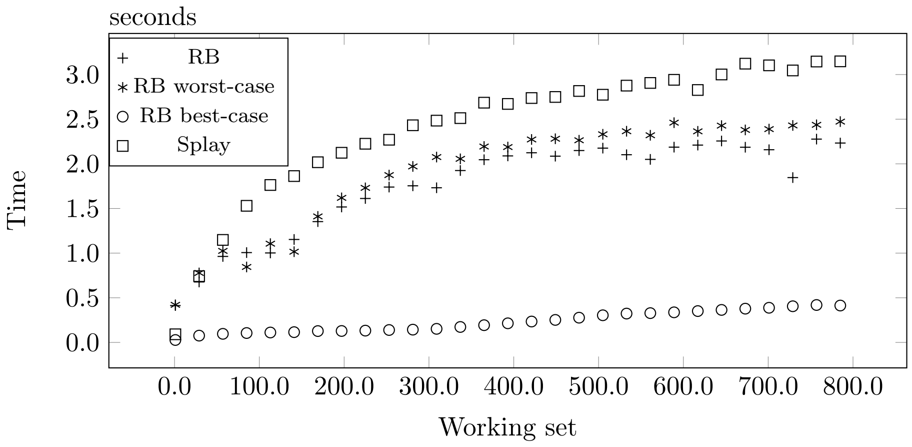
Notice that the plot of the time usage does not clearly reflect the number of touched nodes. It is suspected by us to be caused by the caching of the operating system. It seems likely that cache could have a large impact on the running time when several searches frequently are conducted on the same nodes.

The experimental results for tango trees can be seen in figure 6.6. The experiment uses the same value of n and m as for the other data structures.

The data-serie called *Tango* are execution where a random chosen working-set are used. A search can access no more than t non-preferred children during a search. This is because a search can only switch a single preferred child in respect to a path of interest. We therefore expect the search time to grow with t but stagnates as which nodes is preferred converges toward being random.

The best- and worst-case situation for tango trees are furthermore executed. The best-case scenario is that a search is made for each node on the preferred path. Then the pointer is repeatedly moved by to one non-preferred child. Searches is then made for the new nodes on the preferred path.

The worst-case scenario is to search for nodes for which the pointer only moves by non-preferred children until it is in the auxiliary tree with highest depth. The running time for each search should then be $O(log\, n \cdot log\, log\, n)$ (by the use of theorem 4.2). The only exception for this is the first search in the round of searches as we do not know the state of the lower bound tree. In this case the search time can be less. We suspect this to be the reason for the search time increases as found in the results of the experiment.



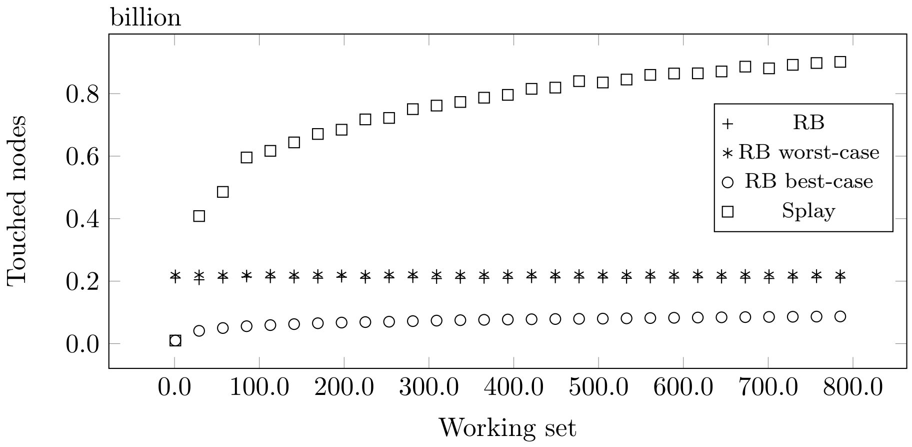
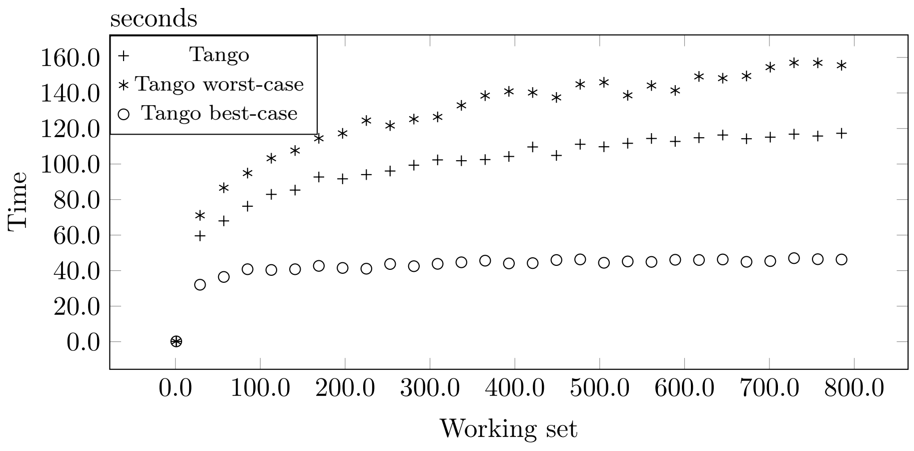


Figure 6.5: Working-set experiment for red-black trees and splay trees. The upper plot shows the time usage and lower plot shows the number of touched nodes. For the experiment is $n = 3000000$ and $m = 10000000$.



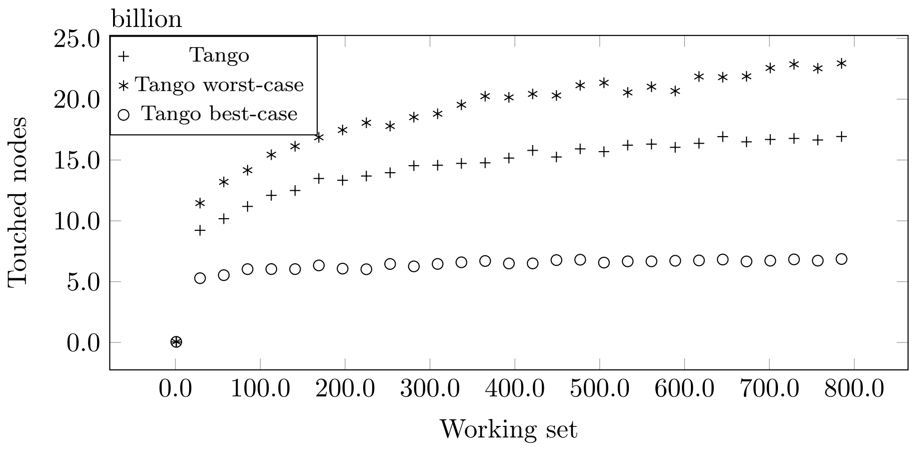


Figure 6.6: Working set experiment results. The upper plot shows the time usage and the lower plot shows the number of touched nodes. For the experiment is $n = 3000000$ and $m = 10000000$.

6.4 Unified Property

In order to consider the data structures performance against the unified property are the following access sequence used:

$$\{1, \frac{n}{2} + 1, 2, \frac{n}{2} + 2, ..., n\}$$

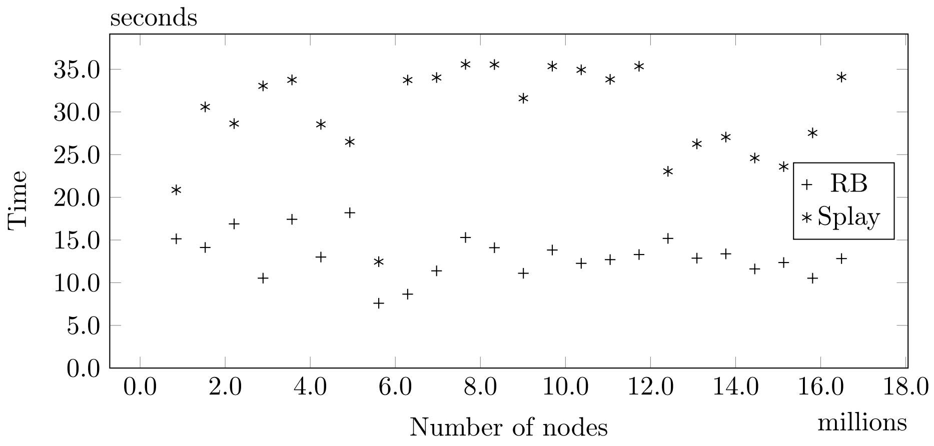
This sequence has the property that every second search is a constant distance away in key-space. A data structure with the unified property should by theorem 2.4 be able to execute the sequence in $O(m\ log(1)) = O(m)$ time. If a data structure has the working-set or dynamic finger property, the running time is $O(m\ log\ n)$.

Figure 6.7 shows the results when executing the access sequence on red-black tree and splay tree. n is changed through the experiment while $m = 100000000$ stays constant.

Once again, red-black trees grow by $O(log\ n)$ because of its worst-case running time.

Splay trees is conjectured to have the unified property, but it is not proven yet. If this is the case the running time should stay constant as it only depends on m. The experimental results justifies this conjecture as the running time is close to constant (it stagnate when higher m was chosen).

Figure 6.8 shows the experimental results for tango trees. The length of the access sequence is the same as for the experiment on the other data structures. The reason it has its own figure is that its time consumption was found to be significantly larger. Its time usage grows slowly. Our hypothesis is that this is caused by every second search being close in key space. The access between the searches can at most change one node from the previous preferred path. Therefore does the searches only need to visit few non preferred children.



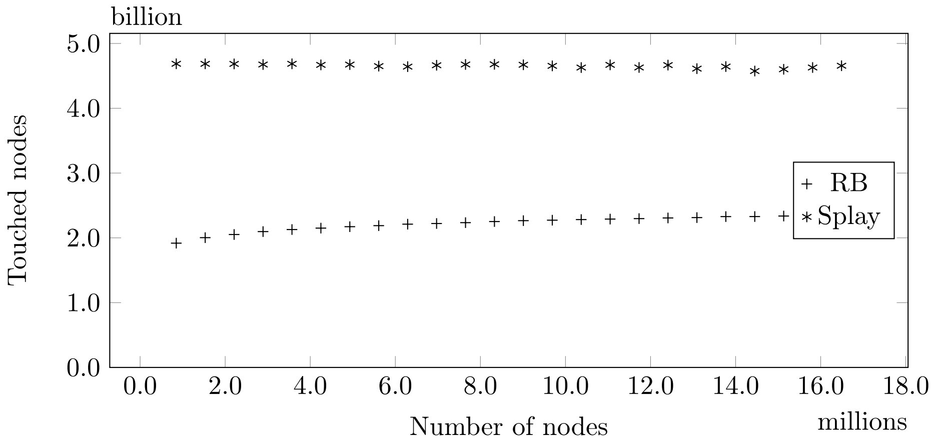
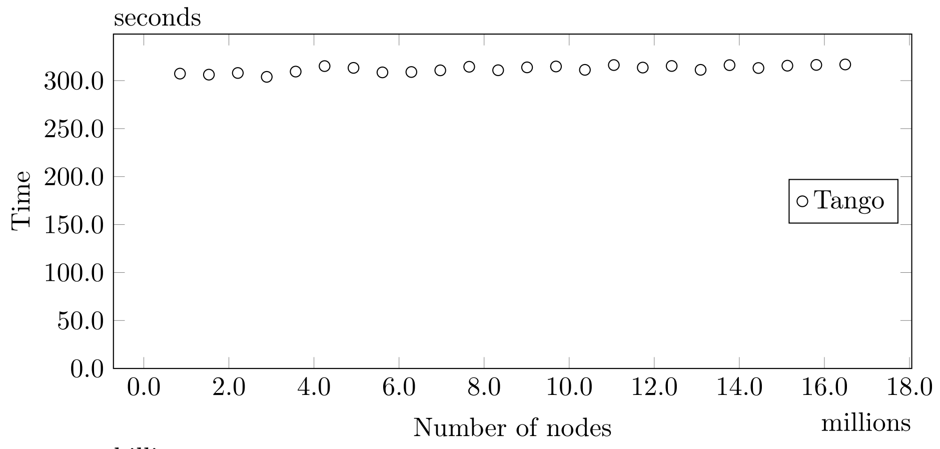


Figure 6.7: Unified property experiment for red-black trees and splay trees. The upper plot shows the time usage and the lower plot shows the number of touched nodes. For the experiment is $m = 100000000$.



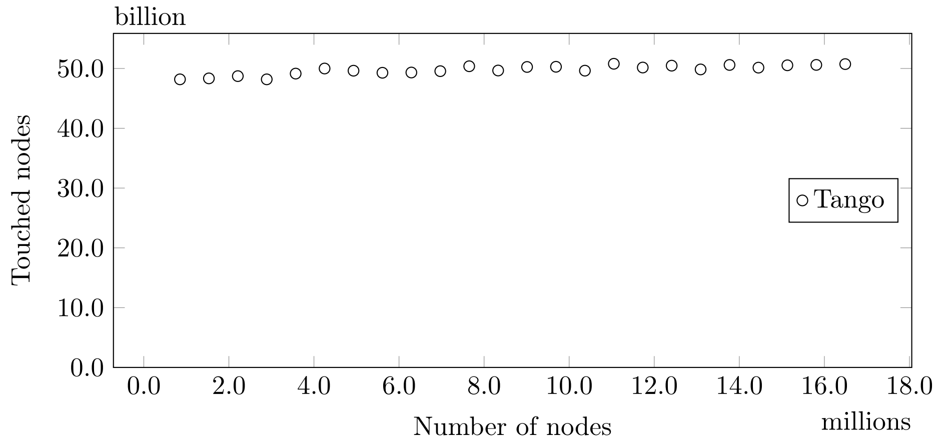


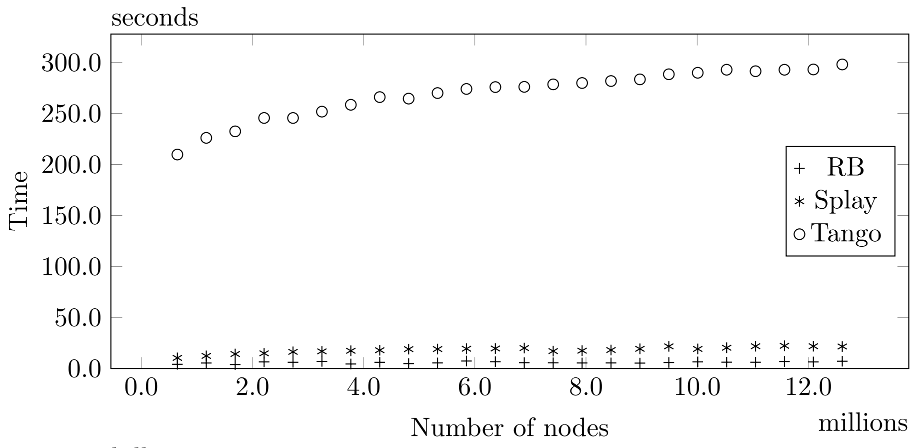
Figure 6.8: Unified property experiment for tango trees. The upper plot shows the time usage and the lower plot shows the number of touched nodes. For the experiment is $m = 100000000$.

6.5 Random Access Sequence

The final experiment compares the data structures when an access sequence is used where keys are chosen randomly. Figure 6.9 shows the results of the execution with different size of n while $m = 10000000$ stays constant.

Red-black trees are expected to always be best for random access sequences. This is because it will not help adjusting the tree. All you can do is to minimize the depth of all nodes and this is what self-balancing binary search trees do.

This experiment shows that self-adjusting trees are only interesting to consider if you know that accesses will come in a systematic order.



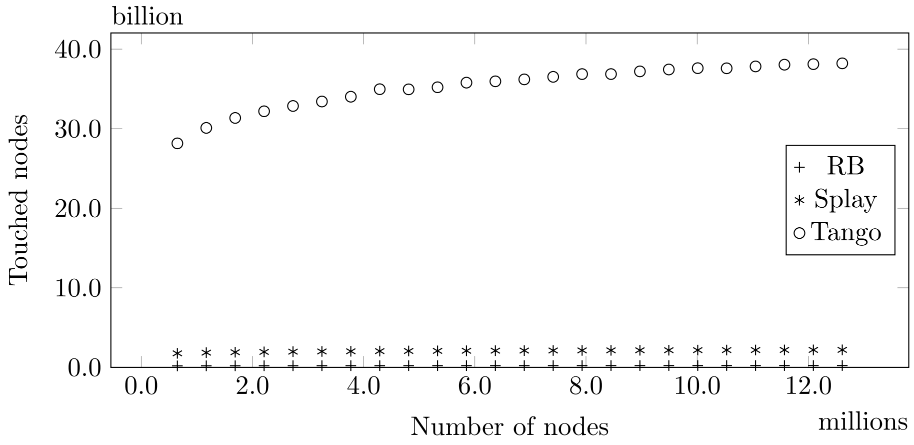


Figure 6.9: Random access sequence experiment with $m = 10000000$. The time usage and the number of touched nodes are plotted as a function of n.

Chapter 7

Conclusion

Tango trees are proved to be $O(log\ log\ n)$-competitive. However, this thesis shows that tango trees do not perform well against the upper bounds of the optimal offline binary search tree. Our results prove that a tango tree uses $(n\ log\ log\ n)$ time to execute the sequential access sequence and does not have the working-set property.

Splay trees, on the other hand, are known to have the dynamic finger, sequential access and working-set property, however it is never proved to be more than $O(log\ n)$-competitive.

The thesis has made an experimental comparison of the two data structures and red-black trees. The results show that red-black trees are the best data structure if the access sequences consist of random chosen keys (in random order).

However, splay trees may be the best if the access sequence contains searches in a systematic order and the tree stores a large number of keys. Splay trees are found to be good if searches in the access sequence are close in times or key space.

It was not possible to find any case where tango trees performed better than the other data structures. Often the time usage was significantly higher than the other. We do therefore not recommend the data structure to be used in practice. However, tango trees' theoretically results are important in respect to the analysis as it shows that $O(log\ log\ n)$-competitiveness is possible.

Splay trees are conjectured to have the unified property. The experimental results are in line with this conjecture.

7.1 Future work

This thesis have argued that the tango-inspired trees have many benefits compared to the original tango trees. Some of these have good results against the upper bounds of $OPT(X)$, and these may be considered in future work.
However, these trees should never be expected to be found any better than $O(log\ log\ n)$-competitive. This is because the interleave bound is not a tight bound and possibly $O(log\,log\,n)$ time away in certain cases.
To find better data structures is it worth considering other lower bounds and design binary search trees which uses these.

APPENDIX A

Experiment data

A.1 Dynamic Finger Data

Table A.1: Data for figure 6.3. Time usage.

	Time (ms)		
Key distance	**Red-black**	**Tango**	**Splay**
1	724	8079	453
401	2204	35938	1413
801	1038	34951	1079
1201	1432	35211	1055
1601	1413	33634	1141
2001	1414	33290	1032
2401	1411	32081	1104
2801	1417	32067	1104
3201	1433	32600	1069
3601	1336	30881	1073
4001	1457	31320	1052
4401	1054	30928	1057
4801	1455	31039	988
5201	1207	30581	1043
5601	1458	30618	962

6001	1454	30642	1017
6401	1468	30372	959
6801	1256	30091	1018
7201	1191	30304	1019
7601	1212	30363	1081
8001	1271	30362	987
8401	1349	30316	1000
8801	1467	30388	994
9201	1479	30802	835
9601	1480	30070	982

Table A.2: Data for figure 6.3. Touched nodes.

	Touched nodes		
Key distance	**Red-black**	**Tango**	**Splay**
1	103377362	1328346740	71159766
401	103656302	4679327709	229245142
801	104072776	4634806378	230347852
1201	103604390	4651237747	230795438
1601	103808294	4603912627	230962082
2001	103997590	4579796046	231138166
2401	103693323	4595115937	231227882
2801	103722655	4559108434	231341190
3201	103525429	4567953697	231322784
3601	103527953	4565643805	231371196
4001	103579184	4551716630	231380532
4401	103593000	4522558167	231422102
4801	103838708	4552251635	231449342
5201	103579815	4560256446	231452970
5601	103675383	4528246186	231444062
6001	103949264	4550972523	231501372
6401	103570504	4520640259	231503780
6801	103688620	4498516582	231490734
7201	103643904	4543631018	231425828
7601	103412006	4529948398	231421160
8001	103696563	4492463259	231527814
8401	103493357	4503583978	231489990
8801	103514686	4522278650	231430688
9201	103597917	4509736983	231425346
9601	103671020	4489571353	231470836

A.2 Sequential Access Data

Table A.3: Data for figure 6.4. Time usage.

	Time (ms)		
Nodes, n	**Red-black**	**Tango**	**Splay**
10000	5	76	3
1510000	216	3184	142
3010000	446	6357	319
4510000	415	9672	482
6010000	912	12953	629
7510000	961	16197	437
9010000	708	19519	576
10510000	1092	22766	1003
12010000	897	26290	596
13510000	1553	29745	1090
15010000	1184	32694	789
16510000	1631	36216	1538
18010000	1858	39708	858
19510000	1827	42874	1407
21010000	2023	46255	1795
22510000	1944	49575	1771
24010000	2050	53047	1138
25510000	2164	56307	1793
27010000	2856	59772	1683
28510000	2458	63346	1347

Table A.4: Data for figure 6.4. Touched Nodes

	Touched nodes		
Nodes, n	**Red-black**	**Tango**	**Splay**
10000	126337	2832396	194542
1510000	30149097	529336039	29392966
3010000	63136788	1065996371	58577302
4510000	97468485	1621644810	87769656
6010000	132148197	2186807035	116979392
7510000	167861711	2734610344	146146332
9010000	203340313	3294242048	175376990
10510000	240199711	3854656282	204552360
12010000	277019115	4427131882	233741464
13510000	313341594	4982750390	262936758
15010000	350362077	5531759620	292152410

16510000	388042405	6142761651	321310234
18010000	425079245	6698602255	350493926
19510000	464071918	7264427849	379743504
21010000	500885990	7850947624	408887300
22510000	538398533	8411049330	438141292
24010000	576711485	8994116534	467283176
25510000	615118348	9555608731	496488048
27010000	655639248	10149213626	525699012
28510000	692302780	10726049690	554861544

A.3 Working-set Data

Table A.5: Data for figure 6.5. Time usage.

Working-set	Time (ms)			
	Red-black			**Splay**
	Random	Worst-case	Best-case	Random
1	413	421	27	91
29	677	778	76	742
57	963	1025	96	1148
85	1006	845	105	1530
113	1002	1106	111	1763
141	1153	1016	115	1863
169	1353	1411	127	2018
197	1517	1619	128	2124
225	1612	1731	132	2225
253	1740	1874	139	2270
281	1754	1971	143	2432
309	1732	2076	152	2484
337	1925	2056	173	2512
365	2046	2194	193	2685
393	2089	2188	214	2671
421	2124	2273	234	2737
449	2086	2281	252	2749
477	2149	2264	277	2816
505	2176	2331	304	2774
533	2102	2365	323	2876
561	2050	2320	328	2906
589	2187	2461	337	2941
617	2210	2364	351	2827
645	2256	2428	364	3001
673	2186	2380	378	3122
701	2158	2389	388	3103
729	1846	2430	405	3046
757	2277	2437	419	3146
785	2233	2473	413	3148

Table A.6: Data for figure 6.5. Touched nodes.

Working-set	Touched nodes			
	Red-black			**Splay**
	Random	Worst-case	Best-case	Random

1	210000000	220000000	10000000	10000000
29	204482762	219999995	41034476	408280428
57	209473673	219473680	49999982	485590370
85	213294124	219529415	55882338	595692242
113	210707950	221858408	59380486	616941566
141	209858153	220212765	62482273	643925054
169	209467450	221065078	65384548	670924808
197	212791892	220253825	67461836	684562014
225	209288895	219377780	69022118	717543736
253	210711460	220909098	70237085	722199912
281	211281140	221814944	72135115	750341510
309	208446624	221229778	73753894	761704538
337	208872393	220267043	75103715	773240660
365	209013707	220931509	76246402	787017612
393	209007585	220661583	77226266	796049676
421	210760086	221615209	78076002	815337818
449	210556800	221135841	78819464	819269820
477	210020947	221215939	79475660	839992640
505	210217821	220712872	80059404	835620050
533	209193263	220487804	80994226	845059318
561	210106954	221782528	81942685	859979076
589	209728355	220780981	82801295	864367424
617	210988681	221620737	83581534	864945492
645	209643413	220790701	84294457	870945814
673	209583951	221471042	84947842	886399562
701	208573464	220813107	85548845	880841152
729	209986270	220864195	86103878	892193644
757	210951129	221096430	86618098	897954716
785	209770730	220573255	87095402	901601250

Table A.7: Data for figure 6.6. Time usage.

Working-set	Time (ms)		
	Tango		
	Random	Worst-case	Best-case
1	146	198	174
29	59591	71118	32066
57	67996	86645	36453
85	76261	94896	40796
113	82929	103250	40374
141	85333	107593	40782
169	92702	114474	42674

197	91672	117270	41510
225	94043	124481	41082
253	96072	121657	43733
281	99355	125297	42498
309	102330	126523	43826
337	101858	133026	44684
365	102498	138496	45603
393	104255	140909	44059
421	109622	140252	44223
449	104834	137482	45972
477	111126	144874	46317
505	109726	146005	44420
533	111673	138669	45180
561	114383	144193	44913
589	112720	141410	46110
617	114793	149294	45993
645	116321	148310	46287
673	114234	149576	44956
701	115129	154528	45369
729	116818	156997	47021
757	115761	156950	46442
785	117307	155594	46267

Table A.8: Data for figure 6.6. Touched nodes.

Working-set	Touched nodes		
	Tango		
	Random	Worst-case	Best-case
1	40000000	60000000	50000000
29	9222829730	11458691345	5284362214
57	10173728188	13200476400	5533674908
85	11186337920	14163190287	6028954718
113	12094792247	15436037714	6035018091
141	12492656696	16129805324	6031779487
169	13480141486	16866067025	6334964992
197	13334016384	17470228867	6071927858
225	13683992882	18062536811	6019864026
253	13955236326	17793579961	6453331278
281	14533447179	18522116666	6257025482
309	14571158641	18803809744	6457194342
337	14721431162	19537345147	6590930461
365	14767994543	20247230490	6697915302
393	15160478444	20149360029	6495616768

421	15791412585	20430197031	6500474408
449	15248652834	20304138684	6771880438
477	15915586886	21143507784	6803182322
505	15687249008	21349081932	6567516435
533	16220039363	20554006036	6669273405
561	16306234718	21025832938	6670189443
589	16043024365	20674045168	6717930128
617	16370999386	21863972285	6744407324
645	16921170920	21807630805	6819743803
673	16493951788	21879280152	6663013318
701	16687796063	22568504182	6724258801
729	16773372842	22867281957	6830562879
757	16647345501	22537285219	6727450938
785	16930064054	22948893537	6862210163

A.4 Unified Property Data

Table A.9: Data for figure 6.7 and 6.8. Time usage.

	Time (ms)		
Nodes, n	**Red-black**	**Splay**	**Tango**
850000	15140	20871	307282
1530000	14123	30591	306209
2210000	16886	28636	307951
2890000	10531	33053	303924
3570000	17423	33755	309409
4250000	13003	28544	315133
4930000	18191	26503	313279
5610000	7591	12464	308576
6290000	8656	33737	308971
6970000	11387	34031	310698
7650000	15294	35569	314437
8330000	14098	35551	310822
9010000	11102	31605	313809
9690000	13827	35364	314736
10370000	12265	34932	311224
11050000	12700	33815	316212
11730000	13296	35352	313592
12410000	15186	23045	315246
13090000	12871	26262	311237
13770000	13380	27058	316120
14450000	11612	24605	313127
15130000	12359	23596	315552
15810000	10523	27544	316364
16490000	12824	34086	316826

Table A.10: Data for figure 6.7 and 6.8. Touched nodes.

	Touched nodes		
Nodes, n	**Red-black**	**Splay**	**Tango**
850000	1918090463	4686592640	48182788855
1530000	2003160404	4685772902	48342391636
2210000	2051632031	4684664786	48720725751
2890000	2096683626	4675164988	48182332671
3570000	2128394167	4686081564	49142518737
4250000	2150224450	4669388498	49993289575
4930000	2172401752	4674225440	49626589420

5610000	2188464008	4650427182	49281242940
6290000	2211148457	4642321152	49323903504
6970000	2221368255	4663646426	49549071414
7650000	2234630907	4675703328	50372932221
8330000	2251714903	4678115340	49658875981
9010000	2265516024	4671036074	50253158836
9690000	2272453445	4654367814	50283734582
10370000	2282124147	4628171422	49632788714
11050000	2289611784	4669862588	50784765965
11730000	2297008153	4629340056	50158372383
12410000	2307109087	4666184222	50470408258
13090000	2312789561	4611486108	49840078571
13770000	2328203155	4643507326	50588786497
14450000	2328729362	4574454580	50138563472
15130000	2336879684	4601671846	50532262122
15810000	2343181387	4629034038	50597218110
16490000	2350395894	4656306072	50731442887

A.5 Random Access Sequence Data

Table A.11: Data for figure 6.9. Time usage.

	Time (ms)		
Nodes, n	**Red-black**	**Splay**	**Tango**
650000	4106	10369	209668
1170000	5343	12322	226032
1690000	3884	14166	232418
2210000	6411	14861	245551
2730000	6081	16278	245528
3250000	6938	16768	251739
3770000	4379	17341	258458
4290000	6050	17878	265991
4810000	4786	18947	264494
5330000	5421	19018	269957
5850000	7291	19349	273987
6370000	6608	19574	275811
6890000	5679	19999	276038
7410000	5539	17012	278461
7930000	5381	17422	279807
8450000	5413	18061	281732
8970000	5260	19262	283359
9490000	5870	21686	288371
10010000	6438	19134	289835
10530000	6160	20320	292818
11050000	6106	21802	291367
11570000	6836	22229	292796
12090000	6394	21807	293071
12610000	7098	21532	297973

Table A.12: Data for figure 6.9. Touched nodes.

	Touched nodes		
Nodes, n	**Red-black**	**Splay**	**Tango**
650000	187346801	1763251374	28154680161
1170000	196059883	1849591382	30098905342
1690000	201452977	1903416668	31350020091
2210000	205296148	1942895706	32186983964
2730000	208237793	1974107896	32854118485
3250000	211354826	1999830998	33414742783
3770000	214241046	2021480370	34017174045

4290000	215238555	2040514210	34962670976
4810000	217066267	2057382312	34944690139
5330000	218384858	2072266974	35206046650
5850000	219853967	2086119308	35785858141
6370000	221054415	2098685836	35952127523
6890000	222309451	2110258358	36196095678
7410000	223493873	2120947204	36517168574
7930000	225143824	2131129230	36862287906
8450000	225085344	2140502214	36863646446
8970000	225830943	2149197854	37200943288
9490000	226875554	2157510384	37439184394
10010000	227270111	2165460900	37605813302
10530000	228661937	2172806670	37590122554
11050000	229043490	2180005392	37824506960
11570000	229641833	2186920770	38043326958
12090000	230115224	2193264674	38111027675
12610000	231497776	2199477524	38212212271

Bibliography

[CD09] Jonathan Carlyle and Derryberry. Adaptive binary search trees. 2009.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 2009.

[CW82] Karel Culik and Derik Wood. A note on some tree similarity measures. 1982.

[DDS$^+$05] Jonathan Derryberry, Daniel Dominic, Sleator, Chengwen Chris, and Wang. A lower bound framework for binary search trees with rotations. 2005.

[DHI$^+$09] Erik D. Demaine, Dion Harmon, John Iacono, Daniel Kane, and Mihai Patrascu. The geometry of binary search trees. 2009.

[DHIP07] Erik D. Demaine, Dion Harmon, John Iacono, and Mihai Patrascu. Dynamic optimality — almost. 2007.

[DSCW09] Jonathan Derryberry, Daniel Sleator, Chengwen Chris, and Wang. Properties of multi-splay trees. 2009.

[Iac01] John Iacono. Alternatives to splay trees with o(log n) worst-case access times. 2001.

[KMRS88] Anna R Karlin, Mark S Manasse, Larry Rudolph, and Daniel D Sleator. Competitive snoopy caching. In *ALGORITHMICA*, 1988.

[Sle02] Daniel Sleator. Splay trees (self-adjusting search trees) - graduate algorithms, carnegie mellon university. 2002.

[ST85] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. 1985.

[Tar83] Robert Endre Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, 1983.

[WDS06] Chengwen Chris Wang, Jonathan Derryberry, and Daniel Dominic Sleator. O (log log n)-competitive dynamic binary search trees. *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, 2006.

[Wil89] Robert E. Wilber. Lower bounds for accessing binary search trees with rotations. IEEE, 1989.