Department of Informatics, University of Zürich

Vertiefungsarbeit

Red-Black-Trees and Heaps in Timestamp-Adjusting Sweepline Based Algorithms

Mirko Richter
Matrikelnummer: 12-917-175
Email: mirko.richter@uzh.ch

November 11, 2016
supervised by Prof. Dr. Michael Böhlen and Katerina Papaioannou

University of Zurich[UZH]
Department of Informatics

1 Introduction

The problem we study in this Vertiefungsarbeit has an input consisting of a number of intervals $I = [(t_s^1, t_e^1), (t_s^2, t_e^2), ..., (t_s^n, t_e^n),]$. t_s is the starting point of the interval and t_e the ending point. The intervals are sorted by their starting points, but the endings points are generally random. We only can fetch one interval after the other and we cannot know at the beginning how many intervals there are. In Figure 1, we illustrated an input of five intervals with increasing starting points and decreasing ending points. We refer to this input structure as inverse hanoi, because if you draw them into a coordination system with the vertical coordinate proportional to the length of the interval and the horizontal axis as the time, they look like a flipped hanoi tower. We decided to use the inverse hanoi structure because then we ensure that the data structure underlying the *event point schedule* will be forced to insert all tuples before deleting any. Hence, the data structure needs to be able to perform insertion and deletion efficiently even on big numbers of elements. Our goal is then to create subintervals, for which we have determined all relevant intervals, for example in Figure 1 for the subinterval (1,2), there is only $\{t^1\}$ relevant or subinterval (5,7) for which $\{t^1, t^2, t^3, t^4, t^5\}$ are relevant.
The sweepline algorithm needs a data structure to process the ending points efficiently. We will explain what operations we need the data structure to perform and then consider the Red-Black-Tree and the Heap as the underlying data structure. Finally, in section 4, we will analyse theoretically and in an experiment how the Red-Black-Trees and Heaps perform in comparison, when we input datasets with inverse hanoi structure.

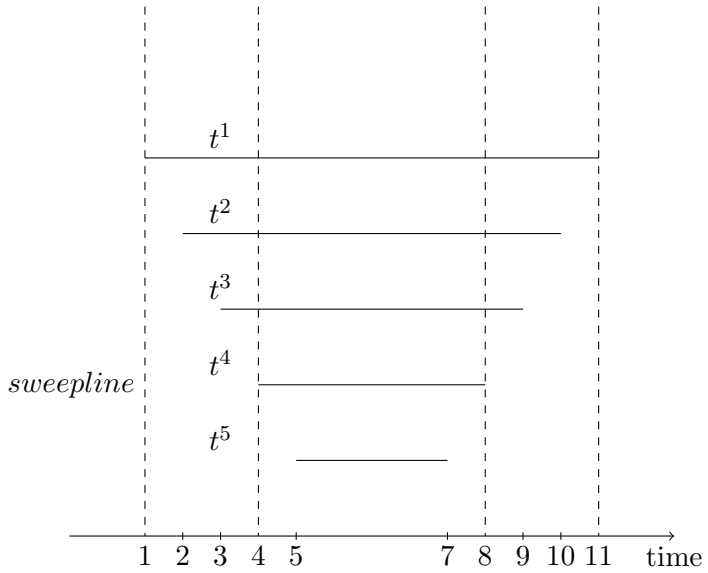


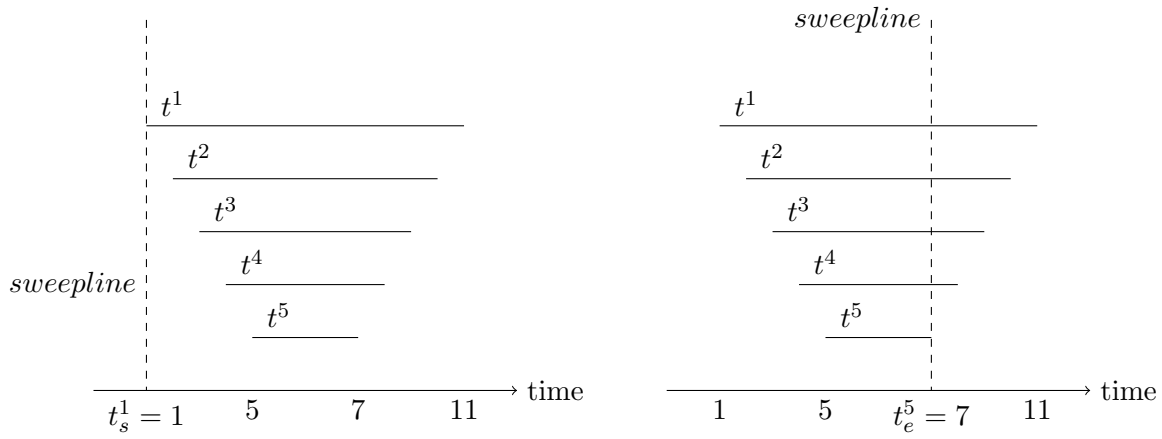
Figure 1: Intervals build *inverse hanoi* and sweepline "sweeps" from left to right through the intervals

The basic intuition of a sweepline algorithm is a vertical line that moves graphically

through the data and updates the *sweepline status* when needed. In our case, the data consists of intervals with a starting and an ending point. Considering Figure 1, you see the sweepline at various time points. To be able to 'move' through the data, the algorithm establishes an *event point schedule*. An event point in defined as a time point at which the algorithm needs to update its *status*[1].

Definition (Relevant Interval)**.** *Lets say A is the set of all input intervals and t is the current time. For every interval $a \in A$, a is relevant if $t_s^a \leq t < t_e^a$.*

Now, we use the *sweepline status* to store the relevant intervals, hence the *status* changes when an interval becomes relevant or stops being relevant. By definition, an interval becomes relevant at its starting point and stops being relevant at its ending point. Therefore the *event point schedule* should consist of all starting and ending points. In our instance, we cannot access the intervals before running the algorithm to create such an schedule. We are bound to fetch one interval after the other. So we consider the example in Figure 2a to explain how we nevertheless can use the concept of the sweepline algorithm.



(a) Sweepline at 1

(b) Sweepline at 7

Figure 2: Sweepline when start fetching intervals and start processing event point schedule

Example. *We start an fetch t^1, we add t^1 to our sweepline status and store t_e^1, which is 11, in our event point schedule. We can already output the subinterval [0,1) with the empty set because there is no relevant interval in time before 1. We proceed by fetching t^2. We add t^2 to the sweepline status, store 10 in the event point schedule and we output the subinterval [1,2) with $\{t^1\}$. The inverse hanoi structure, implies that all starting points are smaller than the smallest ending point. Hence we can fetch all intervals without having to check the event point schedule for an possible ending point. We repeat the*

[1]In this report, *sweepline status* will sometimes just be written as *status*.

fetch process until we have no interval left to fetch.
So now our sweepline status contains all intervals and we have to find the next event point in the ending points we stored in the event point schedule. We can easily see in Figure 2b that the next ending point is 7, which is the smallest of all of them. In fact, we can generalise that. We always have to find the Minimum ending point in the schedule to find the next event point. By taking the minimum we ensure that all other intervals are still relevant, because their ending points are bigger. We reach 7 and remove t^5 *from the sweepline status as well as the 7 from the event point schedule and we output [5,7) with* $\{t^1, t^2, t^3, t^4, t^5\}$*. We proceed in the same fashion until we removed all ending points from our schedule.*

By keeping the *status* updated and defining our subintervals as the interval between two adjacent event points, we get our desired subintervals.

In this report, we want to consider the Red-Black-Trees and Heaps as underlying data structures for the *event point schedule*. As the example has shown, we need the data structure to insert an ending point whenever we fetch an interval, then after we fetched all intervals, we have to repeat finding the minimum and deleting it until we processed all event points. We continue by considering the Red-Black-Tree as the underlying data structure, then we move to the Heap before we compare them in section 4.

2 Red-Black-Trees

As we have established in the previous section, the operations we need to perform in the data structure of the *event point schedule* are insertion, finding the minimum and deletion. We examine these operations on a Red-Black-Tree by applying them on the example we introduce in Figure 1.

2.1 Insertion

The Red-Black-Tree follows the principle of a binary search tree enhanced with five additional properties. (a) Every node is either red or black. (b) The root is black. (c) Every leaf (NIL) is black. (d) If a node is red then both its children are black. (e) For each node, all paths from the node to descendant leaves contain the same number of black nodes.
So to insert a new node in a Red-Black-Tree we first need to follow the tree along the node's keys to the position where the new node belongs, just like when inserting into a binary search tree. New nodes are red, because the property (e) would almost never hold if we inserted black nodes. The only case in which inserting a black node would not violate property (e) is, if the tree was empty and the new node is the root. In all other cases, an inserted black node increases the black height of its subtree by one while the sibling subtree stays at its black height and the root of those two subtrees has not the same black height for every path to the leafs.

When inserting a red node, there are four[2] main cases[3] to restore the tree's properties. We encounter some of them in our example.

We fetch the first interval. At that point the Red-Black-Tree[4] is empty and we insert t_e^1, so (11), as new root. We insert (11) in red, but because the root must be black (Property (b)), we recolour it black and end up with the tree in Figure 3b.

(a) Insert 11 in red

(b) Recolour 11

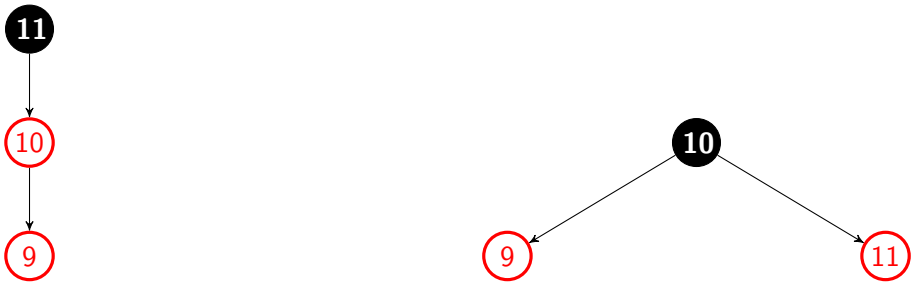
Figure 3: Insert 11 in emtpy tree

Next we need to insert 10. This exposes Case 0, in which we already have a valid red black tree and do not need to restructure.

Figure 4: Sweepline Algorithm at t_s^1, insert 10 in Fig 3b

We proceed to the next interval and insert t_e^2, so (9) in red. Now we are confronted with Case 3. In Figure 5a, nodes (10) and (9) are violating property (d). To restore the property we colour the parent node (10) black and grandparent (11) red and then rotate right over (11) to re-balance the tree. This leads to the tree in Figure 5b which is not violating any properties. In fact, Case 3 always leaves us with a valid Red-Black-Tree.

[2]There are more than four cases in general, but it is enough to know the main four, because the other cases are just symmetrical to one of the four main cases.
[3]The cases are numbered as in the lecture slides of Algorithm & Data-Structures.
[4]For this section, *Red-Black-Tree* also means *event point schedule*.

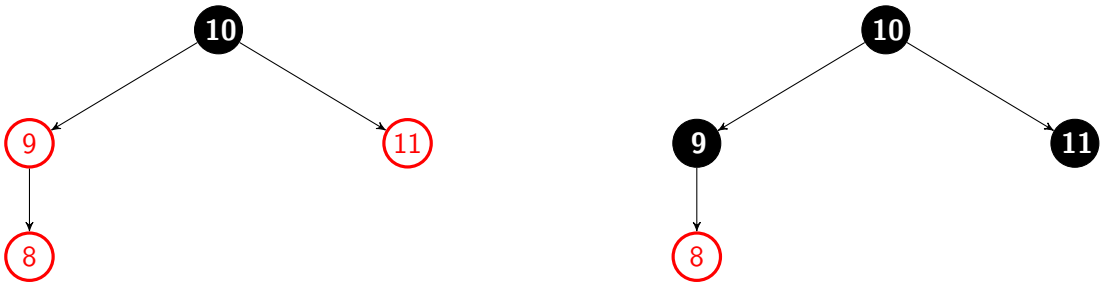


(a) Insert 9 in red

(b) After applying Case 3

Figure 5: Insert 9 in Fig 4

Next, we insert (8) in red and as we can see in Figure 6a, we are confronted with the next property (d) violation. But because the inserted node's brother, in our case node (11), is red we can apply Case 1. This means we colour parent (9) and uncle (11) black and grandparent (10) red. We cannot just recolour the parent node (9) to black, we must recolour the uncle (11) to black as well because otherwise we would violate property (e). We have to check that recolouring grandparent (10) to red, does not violate property (d) further up the tree. Here, grandparent (10) is the root, hence we can recolour it black to resolve all violations.

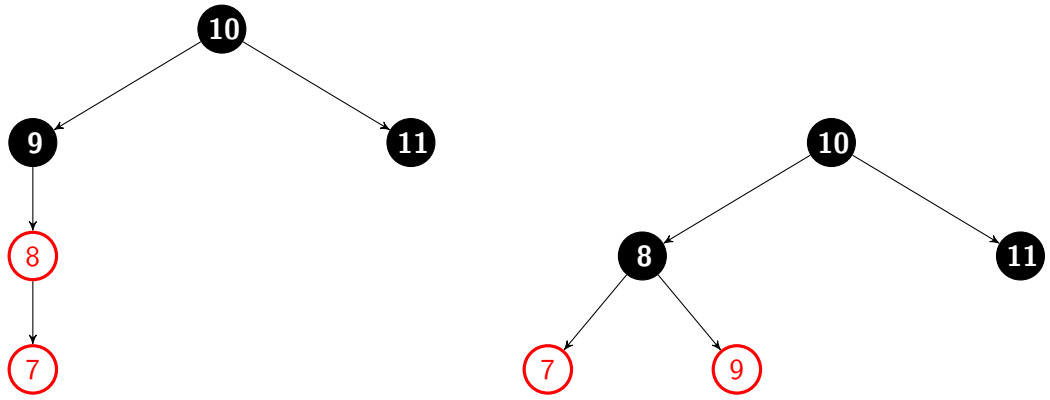


(a) Insert 8 in red

(b) After applying Case 1

Figure 6: Insert 8 in Fig 5b

At last, we need to insert (7) in red and are confronted with applying Case 3 again. We recolour parent (8) to black and grandparent (9) to red and rotate right over grandparent (9). This leads to the valid Red-Black-Tree in Figure 7b.



(a) Insert 7 in red

(b) After applying Case 3

Figure 7: Insert 7 in Fig 6b

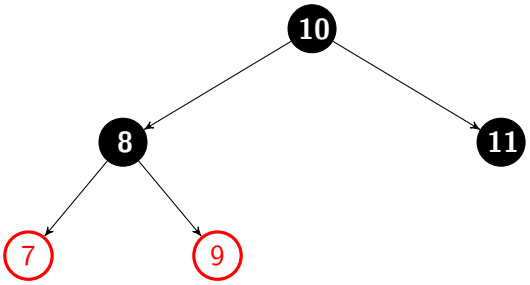
2.2 Find Minimum

The Red-Black-Tree is essentially an enhanced binary search tree. Therefore, we find the minimum if we follow the path of the left child until we reach the leftmost node.
For the example we refer to the next section, in which we will use Find Minimum to remove nodes from the Red-Black-Tree.

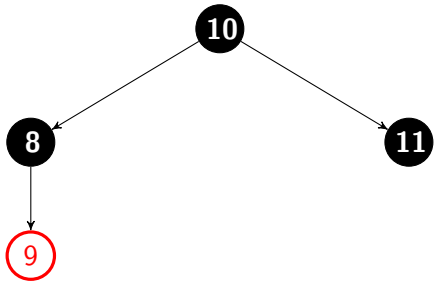
2.3 Deletion

While in insertion the most frequent violated property was property (d), it is property (e) for deletion. In fact, if we want to delete a red node and it is not an internal node[5] we will not have to restructure the tree.
We assume that it is clear how deletion in a binary search tree works, so we can focus on preserving the properties of a red-black tree. In deletion there are five[6] main cases[7].
We continue with our example from section 2.1. We have fetched all intervals and the Red-Black-Tree is filled with all ending points as shown in Figure 8a. We fetch the minimum to get the first ending point we need to process, which is node (7). We take the Red-Black-Tree from Figure 8a and remove node (7). In this instance, node (7) was red and a leaf, hence we do not need to restructure (case 0).

[5] *internal nodes* have two children which are no leaf.
[6] Again, there are more than five main cases, when counting the symmetrical ones as well.
[7] The deletion cases are also numbered as in the lecture slides of Algorithm & Data-Structures



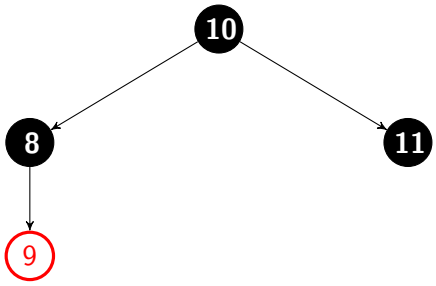
(a) Before deletion of 7



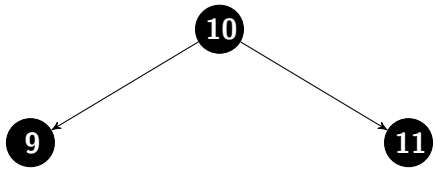
(b) After deletion

Figure 8: Delete 7 in Fig 7b

We proceed and need to remove the minimum in Figure 8b, which is the black node (8). For that we apply the binary tree deletion and swap the 8 with its child 9 and then delete the red 9. Deleting the red leaf does not violate any properties (case 0).



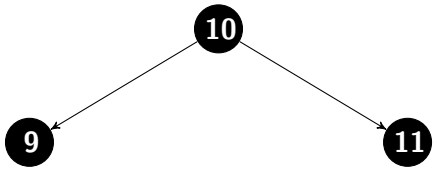
(a) Before deletion of 8



(b) After deletion

Figure 9: Delete 8 in Fig 8b

Next, we remove the black (9). By doing that, we violate property (e). From root (10) going left, the black height will be zero after deleting (9) and going right (through 11), the black height is one. To prevent that we apply Case 2. We recolour the black (11) to red to also reduce the black height of this subtree to zero. But we have to be aware, that if we consider the tree from node (10) a subtree from a bigger tree, we just decreased the black height from the subtree rooted in (10) by one. If node (10) were red, we can recolour it to black to restore the old black height. But if node (10) is already black, we will need to check for further violations up the tree. In our case, (10) is already the root, therefore there is no other subtree with a different black height.



(a) Before deletion of 9

(b) After deletion

Figure 10: Delete 9 in Fig9b

We proceed by deleting the next minimum. But now root has no left child, this means root is the minimum itself. We swap 10 with 11 and delete the red node.

(a) Before deletion of 10

(b) After deletion

Figure 11: Delete 10 in Fig 10b

At last, the sweepline reaches t_e^1 and we delete the root (11). The Red-Black-Tree representing the *event point schedule* is empty.

3 Heaps

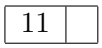
In this section, we consider using a Heap, a Min-Heap to be precise, as the underlying data structure for the sweepline algorithm. We consider a Min-Heap, because one operation we need perform is finding the minimum ending point. In a Min-Heap, we do not need to search for the minimum, but just access the first element in the Heap. Min-Heaps are defined by a nearly complete binary tree with every node being less or equal to its children. A nearly complete binary tree, hence a heap as well can efficiently be stored in an array. To find a node's parent or children we us the following indices

$$\begin{aligned} parent(i) &= \left\lfloor \frac{i}{2} \right\rfloor \\ leftchild(i) &= 2 * i \\ rightchild(i) &= 2 * i + 1 \end{aligned}$$

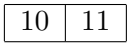
3.1 Insertion

Inserting into a Heap[8] is in comparison to the Red-Black-Tree straight forward. We just append the new ending point to the array[9] that represents the Heap. Then, to find its right position in the Heap, we swap it with its parent until it is bigger than its parent. An array is usually fixed in length. In our problem, we do not know the number of intervals we are going to fetch, that is why we use dynamic memory allocation to lengthen the array when needed.
We use our example directly to illustrate the inserting algorithm.

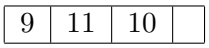
1. Fetching t^1: The Heap is empty and we allocate memory for two elements. We insert 11 as first element.



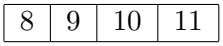
2. Fetching t^2: We append 10 to the array at index $i = 2$. We do not exhaust the size of the array, so we do not need to reallocate memory. We compare the new element to its parent $heap[i] < heap[\lfloor \frac{i}{2} \rfloor] \Rightarrow 10 < 11$. Since it is smaller, we swap them.



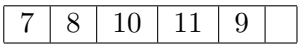
3. Fetching t^3: Insert 9 at index $i = 3$. We try to insert to index 3, but our Heap has only size two. We allocate memory for two more elements appending our existing Heap. Again, because $heap[i] < heap[\lfloor \frac{i}{2} \rfloor] \Rightarrow 9 < 10$ we swap them.



4. t_s^3: We insert 8 at index 4. When comparing it to its parent $heap[4] < heap[2] \Rightarrow 8 < 11$, we see that we again have to swap them. But in contrast to the insertions before, we are not done yet. The new element 8 is now at index 2 and we compare it to its parent at index 1. Because $8 < 9$ we swap again and get



5. t_s^4: We insert 7 at index 5. We are exhausting the size of the Heap, so we allocate memory for two more elements. Then, we first swap with the 9 at index 2 and then with 8 at index 1.



It is easy to see that we always keep the minimum at index 1.

[8] Heap, not otherwise specified, is a Min-Heap.
[9] Array representing a Heap starts with index 1 not 0.

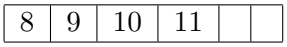
3.2 Find Minimum

As already stated, the minimum of a Min-Heap is at index 1, thus we can fetch it directly, which causes only constant running time.

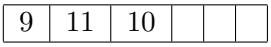
3.3 Deletion

To delete the minimum in the heap we access it and then replace it with the last element. Then we use a similar method as in the insertion to restore the properties. We compare the element at index 1 with its children at swap it with the smaller of the two. We repeat this until both children are bigger. We proceed in our example from section 3.1. We take the minimum in the Heap and remove it.

6. Delete 7: We take the Heap from the last insertion and remove 7. We replace the minimum with the last element (here 9). Compare it to the smaller child, in our case $heap[1]$ to $heap[2] \Rightarrow 9 > 8$, so we swap them. Then 9 has only 11 as child.



7. Delete 8: We proceed in the same manner for the next ending points.



8. Delete 9:

9. Delete 10:

10. Delete 11: We remove the last remaining element and our Heap is empty.

4 Experimental Analysis

In this section we to compare the two data structures and inspect the costs they produce when used in the algorithm theoretically and practically in an experiment.

When inserting a new node in a Red-Black-Tree, we first have to find the correct position for the new node. We know that the height of a Red-Black-Tree has an upper bound of $2 * lg(n + 1)$. To find the correct position to insert, we have to go from root to leaf and on every level we have to compare the existing node to the new node.
This means with c_c representing the cost of a comparison and c_f the cost of fetching a node or element, we have costs $height * c_c * c_f \leq 2 * lg(n + 1) * c_c * c_f$. Then we need to check the new node's parent, uncle and grandparent to determine what case we need to

apply. This gives us three more nodes we need to fetch. The worst case for restoring the properties in a Red-Black-Tree is case 1, because we may have to repeat case 1 for every grandparent until we reach the root. This means we access and recolour 3 nodes on every second level in the Red-Black-Tree. This gives us costs $\frac{height}{2} * 3 * c_f \leq lg(n+1) * 3 * c_f$. We add up all three parts of the insertion and get

$$\begin{aligned} 2 * lg(n+1) * c_c * c_f & \\ +3 * c_f & \\ +lg(n+1) * 3 * c_f & \\ = (2 * lg(n+1) * c_c + 3 + 3 * lg(n+1)) * c_f & \end{aligned} \tag{1}$$

To find the minimum in a Red-Black-Tree, we start at the root and move left until we reach the leftmost node, which must be the minimum. By doing that, we fetch one node one every level. With the height of the Red-Black-Tree $h \leq 2 * lg(n+1)$ we have

$$2 * lg(n+1) * c_f \tag{2}$$

To compute the running time of the deletion we need three parts. First we need to find the minimum, then check what case to apply and then we apply the case. We already established the running time for finding the minimum in Equation 2. To check for the case 2, which is the only case that pushes the violation up the tree, we need to fetch the brother and parent. Then applying case 2 consists only of recolouring the brother. We push the violation up to the parent, so in worst case, we apply case 2 on every level of the tree. This gives us $2 * lg(n+1) * c_f$. All three parts added up results in

$$2 * lg(n+1) * c_f + 2 * c_f + 2 * lg(n+1) * c_f = (4 * lg(n+1) + 2) * c_f \tag{3}$$

for the deletion in a Red-Black-Tree.

For the insertion into a Heap, the worst case is inserting a new minimum which then gets compared and swapped with every parent until it is at index 1. Either by considering that a parent is at index $\lfloor \frac{i}{2} \rfloor$ with length of the array being n = number of elements or that a Heap is binary tree where we have to move the new minimum up to the root from the lowest level, we find that we compare the new element at most lg(n+1) times to existing elements. Hence, the running time for insertion into a Heap is

$$c_f + lg(n+1) * c_c * c_f = (1 + lg(n+1) * c_c) * c_f \tag{4}$$

As we stated, finding the minimum takes only constant time c_f. When deleting the minimum, restoring the properties is very similar to the insertion. We just reverse the direction and we need two comparisons per swap, since we are comparing with both children in contrast to one parent. This means in worst case we have a running time of

$$c_f + 2 * lg(n+1) * c_c * c_f = (1 + 2 * lg(n+1) * c_c) * c_f \tag{5}$$

for deleting an element in a Heap.

	Red-Black-Tree	Heap
Insertion	$(2 * lg(n+1) * c_c + 3 + 3 * lg(n+1)) * c_f$	$(1 + lg(n+1) * c_c) * c_f$
Deletion	$(4 * lg(n+1) + 2) * c_f$	$(1 + 2 * lg(n+1) * c_c) * c_f$

Figure 12: Compare Running Time for Red-Black-Tree and Heap. c_c denotes cost of comparisons and c_f cost of fetch an element.

We have summarised the running times we established in Figure 12. From this running times we can see that both the insertion and deletion for both, Red-Black-Tree and Heap, take the same Order of time, namely O(lg(n)). Now, for every interval we fetch, we insert the corresponding ending point in the data structure and at the ending point, we delete it. This means for every interval we need one insertion and one deletion. For n intervals, we get a complexity of $O(n * lg(n))$ for both, Red-Black-Trees and Heaps. This does not mean that we expect both data structures to perform equally. From the equations in Figure 12, we expect Heap to perform better in insertion and deletion. We ran experiments to show whether this is true or not. We present the results in the following section.

4.1 Experiment

In this section we evaluate the two data structures experimentally. We generated ten datasets starting from 100'000 tuples to 1'000'000 tuples and ten datasets from 1'000'000 increasing to 10'000'000. All have the form of the inverse hanoi, the same structure we saw in the example in the previous sections. For both data structures we measured the time of all insertions, plotted in Figure 13 and all deletions, plotted in Figure 14. The graphs clearly show that both data structure have complexity of O(n*lg(n)). However, the heap is one magnitude faster in inserting the elements than the Red-Black-Tree. If we compare the result of the cost analysis in Figure 12, we have

$$\begin{aligned} &\text{Insertion Red-Black-Tree} <> \text{Insertion Heap} \\ (2 * lg(n+1) * c_c + 3 + 3 * lg(n+1)) * c_f &<> (1 + lg(n+1) * c_c) * c_f \\ lg(n+1)c_c + 3 + 3 * lg(n+1) &> 1 \end{aligned} \tag{6}$$

and we see that we expect the heap to be faster. In Figure 14 we observe that the difference between the Heap and the Red-Black-Tree is smaller in the deletion than in the insertion. We again consider the cost analysis for both data structures and get

$$\begin{aligned} &\text{Deletion Red-Black-Tree} <> \text{Deletion Heap} \\ (4 * lg(n+1) + 2) * c_f &<> (1 + 2 * lg(n+1) * c_c) * c_f \\ 4 * lg(n+1) + 1 &<> 2 * lg(n+1) * c_c \\ 2 + (2 * lg(n+1))^{-1} &> c_c \end{aligned} \tag{7}$$

Here, it is indeed not as apparent as in the insertion. Nonetheless, the Heap should run faster than the Red-Black-Tree, which it did.

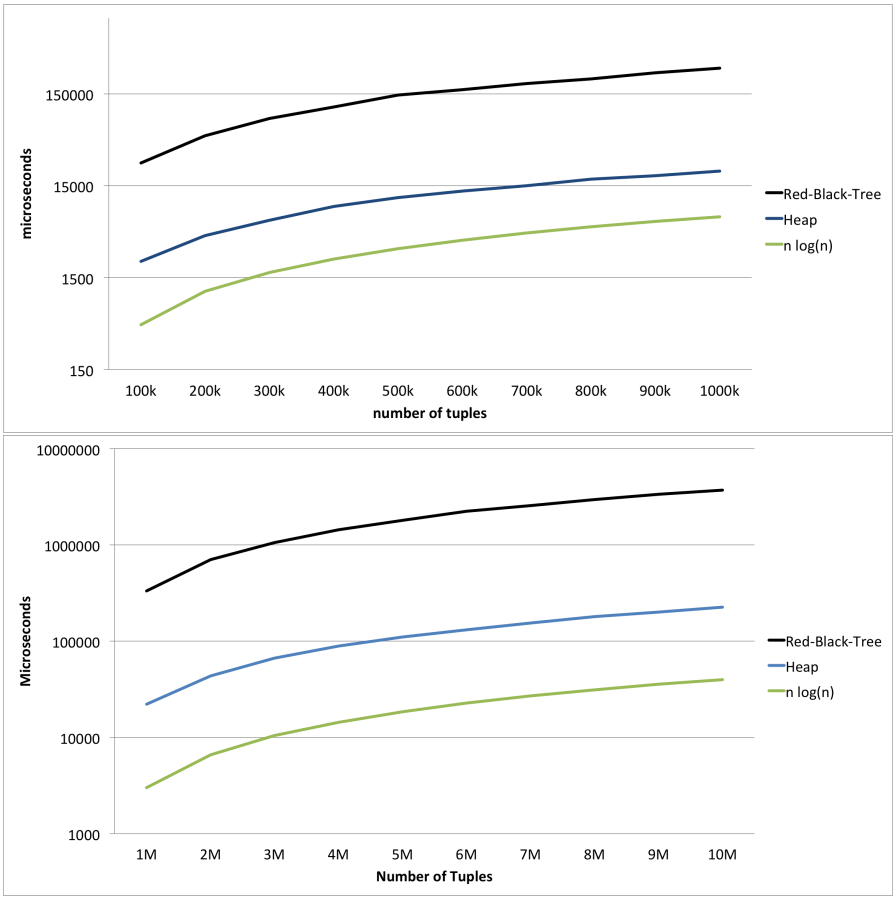


Figure 13: Logarithmic Plot of Red-Black-Tree and Heap insertion and $n * log(n)$ baseline.

5 Conclusion

We have considered two possible data structures to store and process the event points in a sweepline algorithm. We showed that both the Red-Black-Tree and the Heap are suitable candidates for this purpose. Nevertheless, the experiments have shown that the Heap runs more efficiently than the Red-Black-Tree. In a next step, one could examine how efficiently the data structure manages memory allocation to see if the Red-Black-Tree has other advantages over the Heap or one could compare another data structure to the Heap.

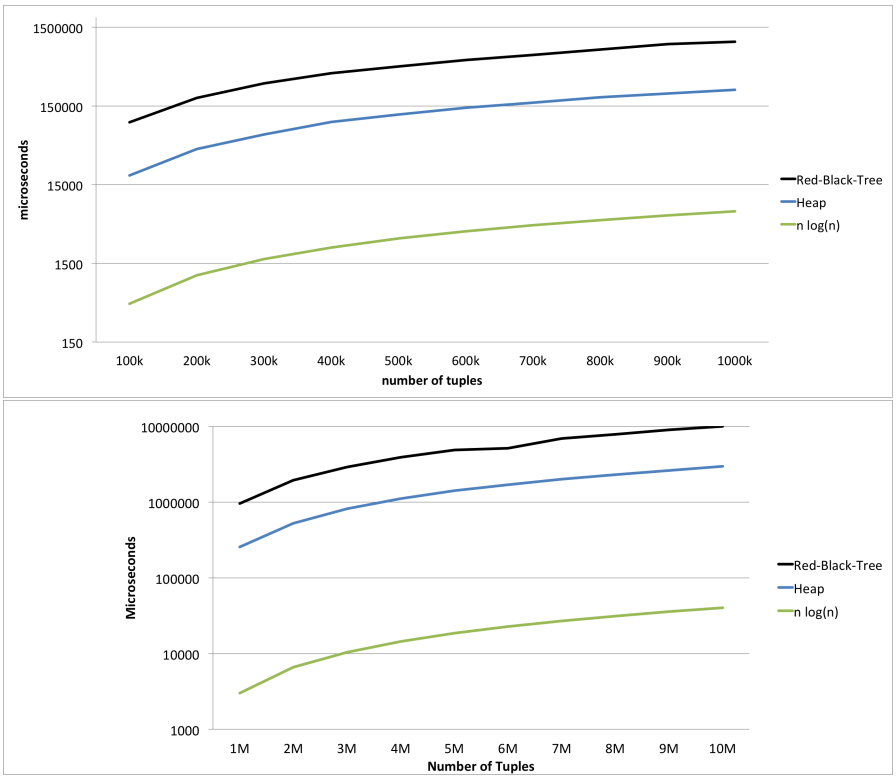


Figure 14: Logarithmic Plot of Red-Black-Tree and Heap deletion and $n*log(n)$ baseline.