Practice 3

Exercise levels:
(L1) Reproduce: Reproduce basic facts or check basic understanding.
(L2) Apply: Follow step-by-step instructions.
(L3) Reason: Show insight using a combination of different concepts.
(L4) Create: Prove a non-trivial statement or create an algorithm or data structure of which the objective is formally stated.

▶ Lecture 7 Binary Search trees

Binary Search Trees and Traversals

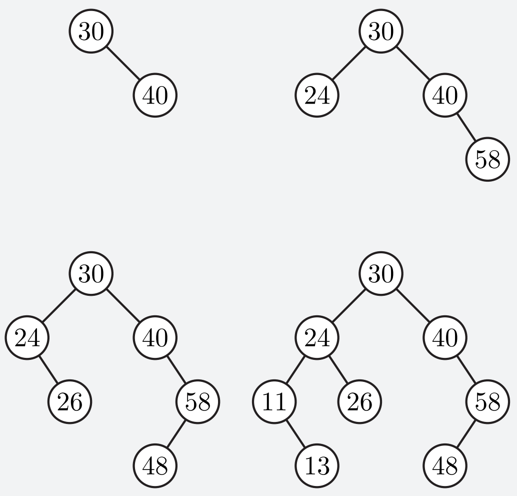
Exercise 1

(a) *(L2)* Insert items with the following keys (in the given order) into an initially empty binary search tree: 30, 40, 24, 58, 48, 26, 11, 13. Draw the tree after any two insertions.

Solution:

Highlights:

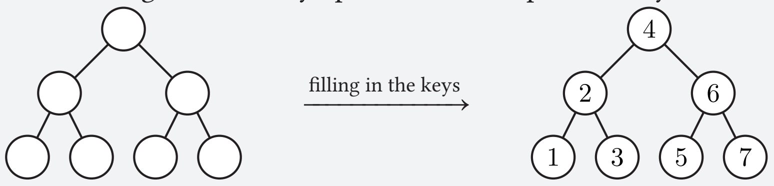
- It is *not* needed to draw the NIL-leaves.



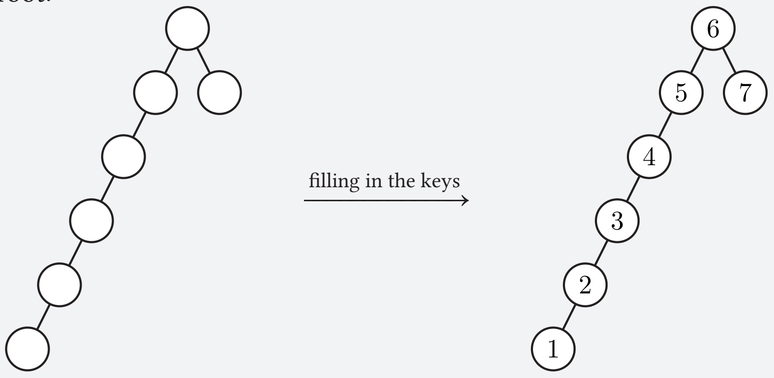
(b) *(L3)* Choose a set of 7 distinct, positive, integer keys. Draw binary search trees of height 2, 5, and 6, excluding NIL-leaves, for your set of keys.

Solution:
Any set of seven distinct integers will work. Let's choose $\{1, 2, 3, 4, 5, 6, 7\}$.
We will tackle this problem by considering what a final solution would look like. So, we will first construct three trees of heights 2, 5, and 6, and then will fill in the values in the nodes of the trees in the order of an *inorder* treewalk, thus ensuring that the binary-search-tree property is satisfied.

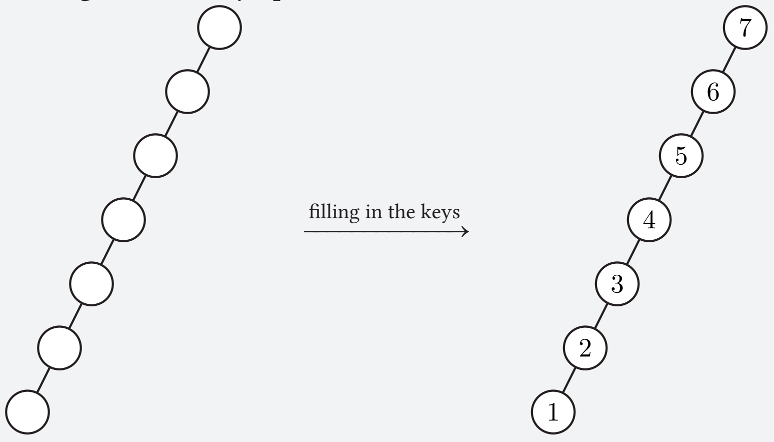
First, for height 2, the only option is the complete binary tree:



For height 5, we start with a chain of six nodes, and add the last node such that we don't increase the height. For example, we can add the last node as the second child of the root:



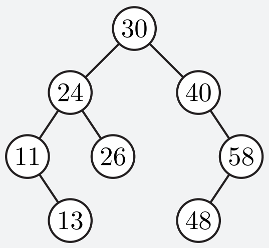
For height 6, the only option is the chain of seven nodes:



Exercise 2 For each of the algorithms PreorderTreeWalk, InorderTreeWalk, and PostorderTreeWalk answer the following questions:

(a) *(L2)* Does the algorithm print the keys stored in a binary search tree in a sorted order? Argue why or give a counter-example.

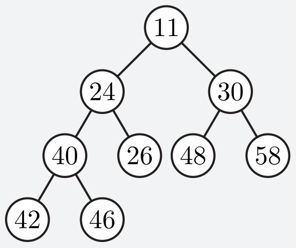
Solution:
InorderTreeWalk does (as discussed in the lecture), PreorderTreeWalk and PostorderTreeWalk do not. For example, for the tree from Problem 1:



PreorderTreeWalk will print the sequence $\{30, 24, 11, 13, 26, 40, 58, 48\}$, and PostorderTreeWalk will print the sequence $\{13, 11, 26, 24, 48, 58, 40, 30\}$.

(b) *(L3)* Does the algorithm print the keys of elements stored in a min-heap in a sorted order? Why or why not?

Solution:
No for all three. For example, for the following min-heap:



InorderTreeWalk will print $\{42, 40, 46, 24, 26, 11, 48, 30, 58\}$, PreorderTreeWalk will print the sequence $\{11, 24, 40, 42, 46, 26, 30, 48, 58\}$, and PostorderTreeWalk will print the sequence $\{42, 46, 40, 26, 24, 48, 58, 30, 11\}$.
Recall the $\Omega(n \log n)$ lower bound on comparison-based sorting. What would the answer "yes" to this problem imply for the lower bound on sorting?

Exercise 3

(a) *(L3)* The *height* of a node v in a rooted tree T is defined as the number of edges on the longest simple downwards path from the node v to a leaf. Write an algorithm that calculates the heights of all nodes in a binary tree in $O(n)$ time. Do not forget to prove the correctness of your algorithm and to argue that it indeed runs in $O(n)$ time.

Solution:

Highlights:

- Often in an algorithm it is useful to use the NIL-leaves as the base case.
- It does not really matter if we count the NIL-leaves in our runtime or not. Note that a binary tree with n internal nodes has exactly $n + 1$ NIL-leaves and thus $2n+1$ nodes in total. Or the other way around, a tree with n nodes has exactly, $\frac{n-1}{2}$ internal nodes. In other words, asymptotically there are equally many internal nodes as there are NIL-leaves and as there are nodes in total. We like counting the runtime disregarding the NIL-leaves, as only at the remaining nodes do we actually store keys. Thus $T(n)$ becomes the runtime for a tree containing n keys.
- It is sufficient to argue that this is an in-order treewalk that uses $O(1)$ time per node, to conclude that the algorithm runs in $O(n)$ time. Here we work out an official substitution proof as an example, but it is not required to analyze the runtime.

The height of a node is the maximum number of edges that need to be traversed from a node to reach a NIL-leaf. Particularly, the height of a NIL-leaf is 0.
If we know the heights h_ℓ and h_r of a node's left and right child, then we can compute the height of the node itself by taking the maximum of h_ℓ and h_r, and adding 1. A PostorderTreeWalk gives exactly the functionality that we need. We first compute the height of the left and right child and then we can use this information to compute the height of the current node.

```
ComputeHeight($x$)
    Input: node $x$
    Output: height of node $x$
1   if $x$ == NIL return 0
2   $h_\ell$ = ComputeHeight($left[x]$)
3   $h_r$ = ComputeHeight($right[x]$)
4   $h[x] = \max\{h_\ell, h_r\} + 1$
5   return $h[x]$
```

Initially we call ComputeHeight on the root.
Proof of correctness: We prove the correctness of the algorithm using strong induction. Namely, we claim that for any value of $h \geq 0$, calling ComputeHeight on the root x of a subtree of height h correctly determines the height values $h[y]$ for all internal nodes y in the subtree rooted at x, and moreover, it correctly returns the height of the subtree

rooted at x.

Base case: $h = 0$. In this case x is a NIL-leaf. A subtree rooted at a NIL-leaf contains no internal nodes so trivially the first part of the claim is achieved. Moreover, as argued above a leaf has height 0 and the algorithm also correctly returns 0. Thus the indeed our claim is true for a subtree with a NIL-leaf as root.

Inductive step:

Induction hypothesis: Assume that ComputeHeight works correctly for all subtrees of height $0 \leq k < h$ for some fixed $h > 0$.

Let x denote the root of the subtree T_x and assume that T_x has height h. Let T_ℓ and T_r denote the subtrees rooted at left[x] and right[x], respectively. By definition, $height(T_x) = \max\{height(T_\ell), height(T_r)\}+1$. Since T_x has height greater than 0, the algorithm does not return in line 1 but proceeds to line 2. As $height(T_x) = \max\{height(T_\ell), height(T_r)\} + 1$, T_ℓ and T_r must both be strictly smaller than $height(T_x) = h$. Therefore, by induction hypothesis, ComputeHeight correctly determines the values h_ℓ and h_r in lines 2 and 3. Line 4 then directly computes the height for T_x by following the definition based on the correct height h_ℓ and h_r and therefore the algorithm also computes $h[x]$ correctly. Finally it thus also returns the correct value in line 5. Thus the claim holds for a subtree of height h as well.

Combining the base case with the inductive step, we conclude that by induction ComputeHeight, when called on the root x of a tree, correctly determines the height values $h[y]$ for all internal nodes y in the tree (and correctly returns the height of the complete tree), for all trees with height $h \geq 0$.

Running time: Let $T(n)$ denote the running time of the algorithm on a tree with n internal nodes (so not counting the NIL-leaves).

When the ComputeHeight is called on a tree with no internal nodes (so just a NIL-leaf), then it returns in the first line. Therefore, it takes a constant time to run. Let's say some positive constant c time. Thus $T(0) = c$.

Now suppose that ComputeHeight is called on a node x, which is the root of a subtree with n internal nodes. Then ComputeHeight line 1 will not be executed and instead line 2-5 are executed. Lines 4-5 require $O(1)$ time. Let's say they take a positive constant d time.

The time spent in line 2-3 depends on the recursive call on the two children of x. Let ℓ be the number of internal nodes in the subtree rooted at x's left child. The there must be $n - \ell - 1$ internal nodes in the subtree rooted at x's right child. Thus we get the following recursion:

$$T(n) = T(\ell) + T(n - \ell - 1) + d\,,$$

for some positive constant d. We show that $T(n) = O(n)$ by substitution method. That is, we show that there exist a positive constant c_1 such that $T(n) \leq c_1 n$ for all sufficiently large n.

Base case: $n = 0$: $T(0) = c \leq c_1$ for all values of c_1 that are not less than c.

Inductive step:

Induction hypothesis: Assume that $T(k) \leq c_1 k$ for all $0 \leq k < n$ for some fixed $n \geq 1$.

We will show that then $T(n) \leq c_1 n$:

$$\begin{aligned} T(n) &= T(\ell) + T(n-\ell-1) + d && \{\text{By IH on } T(\ell) \text{ and } T(n-\ell-1)\} \\ &\leq c_1\ell + c_1(n-\ell-1) + d \\ &= c_1 n + (-c_1 + d) && \{\text{If } -c_1 + d \leq 0\} \\ &\leq c_1 n\,, \end{aligned}$$

This is true when $(-c_1 + d) \leq 0$, so for $c_1 \geq d$. Pick $c_1 = \max\{c, d\}$, then the base case and the inductive step hold. Thus by induction, $T(n) \leq c_1 n$ for all $n \geq 0$. This directly implies that $T(n) = O(n)$.

(b) *(L3)* The *depth* of a node v in a rooted tree T is defined as the number of edges on the simple path from the root of T to the node v. Write an algorithm that calculates the depths of all nodes in a binary tree and has running time $O(n)$. Do not forget to prove the correctness of your algorithm and to argue that it indeed runs in $O(n)$ time.

Solution:
Unlike the height of a node, the depth of a node x does not depend on the subtree rooted at x, but on the path from x to the root of the tree. Thus, we need to know the depth of a parent of x to compute the depth of x. We will modify the PreorderTreeWalk in the following way:

```
ComputeDepth($x$)
    Input: node $x$
1   if $x \neq$ NIL
2       if $p[x] ==$ NIL
3           $d[x] = 0$
4       else
5           $d[x] = d[p[x]] + 1$
6       ComputeDepth($left[x]$)
7       ComputeDepth($right[x]$)
```

Initially we call ComputeDepth on the root node.
Correctness: The correctness follows from the definition of the depth. The length of a simple path from a node x to the root of the tree is one larger than the length of the path from the parent of x to the root of the tree.
When computing the depth of a node x, the depth of a parent of x has already been computed, as the depth of a node is computed in lines 2–5, before the recursive call to ComputeDepth on its left and right children in lines 6 and 7. Thus by simply computing the depth based on the definition for each node the depth is correctly determined.
Running time analysis: The algorithm ComputeDepth is a pre-order treewalk. As each line (except for the recursion) takes $O(1)$ time the algorithm takes $O(1)$ time to process a node. As a pre-order treewalk visits each node once, we conclude that the total algorithm runs in $O(n)$ time.

Balanced Search Trees

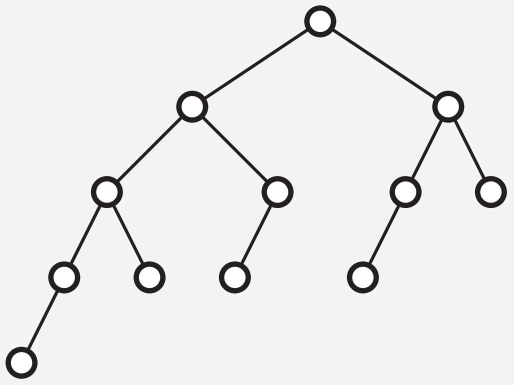
Exercise 4 A *Fibonacci tree* is a binary tree that is defined as follows: a Fibonacci tree F_0 of rank 0 is an empty tree; a Fibonacci tree F_1 of rank 1 is a tree that has 1 node; a Fibonacci tree F_k of rank $k \geq 2$ consists of a root node with F_{k-1} as its left subtree, and F_{k-2} as its right subtree.

(a) *(L2)* Draw the Fibonacci tree of rank 5.

Solution:

Highlights:

- Construct by building all Fibonacci trees upto to rank 5 by following the definition.



(b) *(L3)* What is the height of a Fibonacci tree of rank k?

Solution:
The height of the Fibonacci tree of rank k is $k - 1$.
To prove this, observe that the height of the root of the Fibonacci tree of rank k equals the height of the Fibonacci tree of rank $k - 1$ plus 1. The height of F_1, which consists of one node, is 0. Thus, we get the following recursion:

$$height(F_k) = \begin{cases} 0 & \text{when } k = 1\,, \\ height(F_{k-1}) + 1 & \text{when } k > 1\,. \end{cases}$$

This recursion solves to $height(F_k) = k - 1$. This can be formally proven, for example, by induction. The recursion is so trivial though that directly concluding the right answer is accepted.

(c) *(L3)* Is every Fibonacci tree an AVL-tree? Why or why not?

Solution:
Yes. We will prove that every Fibonacci tree is an AVL-tree by induction on the rank k.
Base case: $k = 1$ and $k = 2$. We can check by hand that F_1 and F_2 are both AVL-trees.
Inductive step:

Induction hypothesis: Assume that F_i is an AVL-tree for all $2 \leq i < k$ for some fixed $k > 2$.
We will show that then F_k is also an AVL-tree. By definition, it consists of a root node with F_{k-1} as its left subtree, and F_{k-2} as its right subtree. By exercise (b) we know that the height of F_{k-1} is $k-2$ and F_{k-2} is $k-3$. Thus the height of the left and right subtree differs by at most one, and the AVL-property holds for the root of F_k. Furthermore, by IH, both subtrees are proper AVL-trees. This implies, that for any node x in F_k other than the root, the AVL-property holds as well. So, the AVL-property holds for all nodes of F_k, and therefore by definition F_k is an AVL-tree.

(d) *(L4)* Is every Fibonacci tree a red-black tree? If so, give a scheme to color the nodes of a Fibonacci tree "red" and "black" so that it becomes a red-black tree. If not, give a counter-example.

Solution:

Highlights:

- This exercise takes some puzzling. The key observations to make are that (1) the root should always be black, and (2) one subtree has a height that is one larger than the height of the other subtree. Particularly to color F_k we need to ensure that both F_{k-1} and F_{k-2} have the same black-height, or we need to adjust the black-height. This can be done by removing one black node from the larger subtree and coloring it red.
- It is not trivial to formally prove this scheme correct.

Yes. We will construct the coloring recursively:
$k = 1$. Color the one node of F_1 black. Then, F_1 satisfies all five red-black properties, and is a red-black tree. Note, that black height of the root is 1: $bh(\text{root}[F_1]) = 1$.
$k = 2$. Color the root black and its child red. Then, F_2 satisfies all five red-black properties, and is a red-black tree. Note, that black height of the root is still 1: $bh(\text{root}[F_2]) = 1$.
$k = 2i + 1$. Color the root black, and color the subtrees rooted at the children of the root exactly as F_{2i} and F_{2i-1}.
$k = 2i + 2$. Color the root black, and color the subtrees rooted at the children of the root exactly as F_{2i+1} and F_{2i}. Recolor the left child of the root into red.
To prove the correctness of the coloring scheme, we use the following claims, which we can prove by induction:

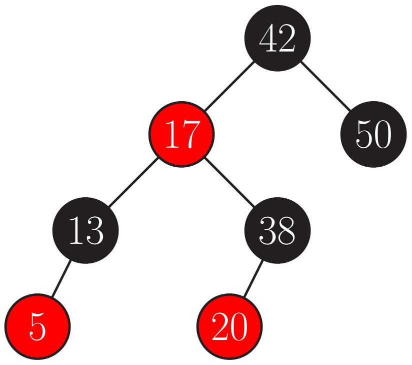
- $bh(\text{root}[F_{2i-1}]) = bh(\text{root}[F_{2i}])$
- the children of the root of each F_{2k+1} are both black (thus, we can change the color of the left child of the root of F_{2i+2} to red without violating the red-black properties)

We leave the proof of these facts as a (difficult) exercise.

▶ Lecture 8 Augmenting data structures

RB-tree operations

Exercise 5 *(L2)* Given the following red-black tree T:



Show all the changes to the tree when executing the following three operations in sequence:

(a) RB-Insert(T, 25)

(b) RB-Insert(T, 18)

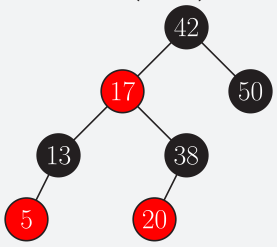
(c) RB-Delete(T, 25)

Solution:

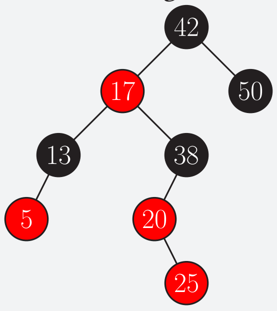
Highlights:

- Note that we assume the algorithm as described in the book (and slides). Particularly some websites use a slightly different algorithm and end up with a different red-black tree.

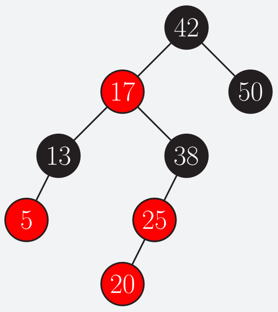
(a) RB-Insert($\boldsymbol{T}$, 25)



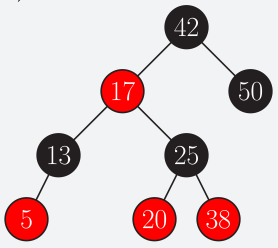
After inserting 25 and coloring the node "red":



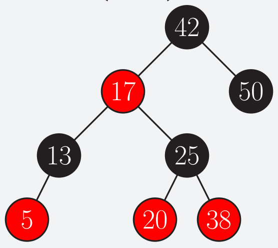
After left rotation around the node with key 20 (case 2):



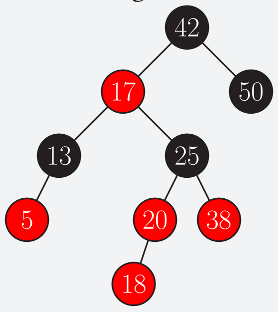
After right rotation around the node with key 38 and recoloring nodes 25 and 38 (case 3):



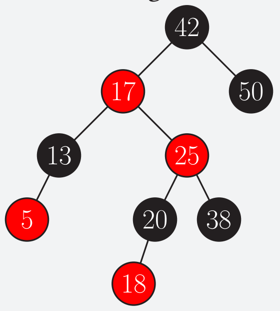
(b) RB-INSERT(T, **18**)



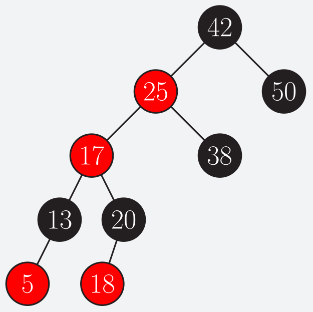
After inserting 18 and coloring the node "red":



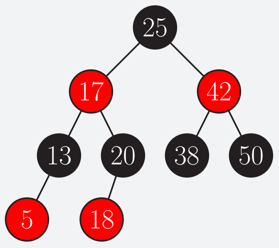
After recoloring the nodes with keys 20, 38, and 25 (case 1):



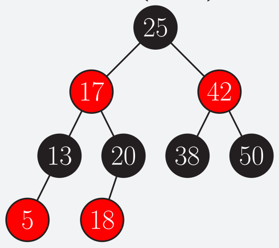
After left rotation around the node with key 17 (case 2):



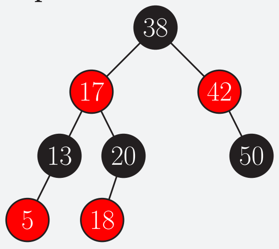
After right rotation around the node with key 42 and recoloring nodes 25 and 42 (case 3):



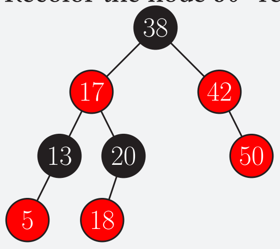
(c) RB-Delete$(T, 25)$



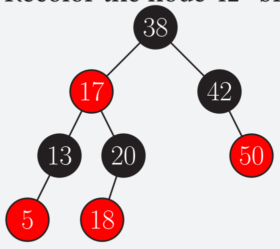
Replace the node 25 with its successor 38:



Recolor the node 50 "red":



Recolor the node 42 "black":



Augmenting Data Structures

Exercise 6 Different data structures have different advantages and disadvantages.

(a) *(L1)* For each of the data structures give the asymptotic worst-case runtimes for all the operations.

	Sorted array	**Min-heap**	**OS-tree**
INSERT(S, t)			
MINIMUM(S)			
EXTRACT-MIN(S)			
OS-SELECT(S, i)			

Solution:

Highlights:

- Side-note: Observe that these *operations* all do only one thing. They are the basic interactions we allow with the data structure. This in contrast to higher-level algorithms like sorting the data (which is *not* an operation).

	Sorted array	**Min-heap**	**OS-tree**
INSERT(S, t)	$\Theta(n)$	$\Theta(\log n)$	$\Theta(\log n)$
MINIMUM(S)	$\Theta(1)$	$\Theta(1)$	$\Theta(\log n)$
EXTRACT-MIN(S)	$\Theta(n)$	$\Theta(\log n)$	$\Theta(\log n)$
OS-SELECT(S, i)	$\Theta(1)$	$\Theta(n)$	$\Theta(\log n)$

(b) *(L3)* Your application never inserts or deletes an element (static data) and you often need to find the element at a given rank. What data structure would be the best to use? Explain your answer.

> **Solution:**
> Sorted array, as the running time of both MINIMUM and OS-SELECT is $O(1)$.

(c) *(L3)* Your application has very many inserts (dynamic data) and you often need to find the element at a given rank. What data structure would be the best to use? Explain your answer.

> **Solution:**
> OS-tree. It is better than the sorted array, as it allows for $O(\log n)$ time INSERT operations, and better than min-heap, as it allows for $O(\log n)$ time OS-SELECT operations.

(d) *(L3)* Your application has very many inserts and you often need to find the minimum and extract the minimum. What data structure would be the best to use? Explain your answer.

> **Solution:**
> Min-heap is better than the sorted array for inserts, and better than OS-tree for finding the minimum.

Exercise 7

(a) *(L2)* We augment every node x in a red-black tree T with a field h that stores the height of the node x. Can field h be maintained without affecting the asymptotic performance of any red-black tree operation? Show how, or argue why not.

> **Solution:**
> Height $h[x] = 0$ if $x = \text{NIL}$, and $h[x] = \max\{h[\text{left}[x]], h[\text{right}[x]]\} + 1$ otherwise. Hence $h[x]$ depends only on the contents of x and contents of its children. Therefore by the RB-tree augmentation theorem it can be maintained without affecting the asymptotic running time of insertion and deletion.

(b) *(L2)* We augment every node x in a red-black tree T with a field d that stores the depth of the node x. Can field d be maintained without affecting the asymptotic performance of any red-black tree operation? Show how, or argue why not.

> **Solution:**

Highlights:

- The augmentation theorem only states that something *can* be maintained in $O(\log n)$ time if the requirements are met. However, it does *not* say that it cannot be maintained if these requirements are not met. (It is an IF statement not an IF AND ONLY IF.) Thus, to argue that the runtime cannot be maintained we *cannot* use the augmentation theorem but really need to give a counter-example.

The field d cannot be maintained without affecting the asymptotic running time. To show why we give a counter-example. A rotation at the root will changes the depth-values in for more than half of the tree and therefore requires $\Omega(n)$ time. Such a rotation can be achieved by repeatedly inserting a new smallest element into an RB-tree. After a linear number of such operations a rotation at the root will occur.

Exercise 8 *(L3)* Let $S = \{k_1, \ldots, k_n\}$ be a set of distinct integers. Design a data structure for S that supports the following operations in $O(\log n)$ time: $\textsc{Insert}(S, k)$ which inserts the number k into S (you can assume that k is not contained in S yet), and $\textsc{TotalGreater}(S, a)$ which returns the sum of all keys in S that are larger than a, that is, $\sum_{s \in S,\ s>a} s$.
Argue why you can still insert elements in $O(\log n)$ time in your data structure.
Describe the algorithm $\textsc{TotalGreater}(S, a)$, argue its runtime, and prove its correctness.

Solution:
We augment a red-black tree by storing a *sum* field in each node which contains the sum of all keys in the tree rooted at that node. The value of this field depends only on the values stored at its children (and its own key):

$$\mathit{sum}[x] = \mathrm{key}[x] + \mathit{sum}[\mathrm{left}[x]] + \mathit{sum}[\mathrm{right}[x]]\,.$$

Thus, by the theorem from the lecture, the field *sum* can be maintained without influencing running time of the $\textsc{Insert}$ operation, which is $O(\log n)$.

Next, we develop the $\textsc{TotalGreater}$ operation. It will be based on the normal $\textsc{Search}$ operation for binary search trees. Specifically, we will be searching with argument a. For each node x we encounter, there can be three cases:

- If $\mathrm{key}[x] = a$, then all the keys in the subtree rooted at $\mathrm{right}[x]$ are strictly larger than a, and we need to add $\mathit{sum}[\mathrm{right}[x]]$ to the total sum. The node x itself, and the subtree rooted at $\mathrm{left}[x]$ do not contribute to the total sum.

- If $\mathrm{key}[x] > a$, then the keys in the subtree rooted at $\mathrm{right}[x]$ are also larger than a. We add $\mathrm{key}[x] + \mathit{sum}[\mathrm{right}[x]]$ to the total sum, and recurse into the left child of x.

- If $\mathrm{key}[x] < a$, then the key of the node x and the keys of the nodes of the subtree rooted at $\mathrm{left}[x]$ do not contribute to the total sum. We simply recurse into the right child of x.

```
TotalGreater($x, a$)
if $x$ == NIL return 0
if key[$x$] == $a$ return sum[right[$x$]]
if key[$x$] > $a$
    return key[$x$] + sum[right[$x$]] + TotalGreater(left[$x$], $a$)
else
    return TotalGreater(right[$x$], $a$)
```

Initially we call TotalGreater on the root of the tree.
Proof of correctness: We will prove the correctness of TotalGreater by induction on the height of the subtree rooted at x.
Base case: $h = 0$ and $x =$ NIL. Then, TotalGreater correctly returns 0, as $\sum_{s \in S_x,\ s>a} s = 0$, where S_x is the set of keys in the subtree rooted at x.
Inductive step:
Induction hypothesis: Assume that TotalGreater works correctly for all subtrees of height $0 \leq k < h$ for some fixed $h > 0$.
We will show then that TotalGreater works correctly on subtrees of height h as well.
Let x be a node of a subtree of height h, and let S_x denote the set of keys stored in the subtree rooted at x. There are three cases to consider: key[x] $= a$, key[x] $< a$, and key[x] $> a$:

- When key[x] $= a$, by the binary-search-tree property, and because all the keys in the tree are distinct, we know that all the keys stored in the subtree rooted at left[x] are $< a$, and all the keys stored in the subtree rooted at right[x] are $> a$. In this case, TotalGreater correctly returns the value of the field *sum* of the right child of x.

- When key[x] $< a$, by the binary-search-tree property, and because all the keys in the tree are distinct, we know that x itself and all the keys in the subtree rooted at left[x] do not contribute to $\sum_{s \in S_x,\ s>a} s$. In this case, TotalGreater returns the value of the recursive call on the right child of x. By induction hypothesis, this call correctly returns the sum of keys stored in the right subtree that are larger than a, and thus TotalGreater returns the correct sum for the node x as well.

- When key[x] $> a$, by the binary-search-tree property, and because all the keys in the tree are distinct, we know that x itself and all the keys in the subtree rooted at right[x] contribute to $\sum_{s \in S_x,\ s>a} s$. In this case, TotalGreater returns the value of the recursive call on the left child of x, plus key[x] + *sum*[right[x]]. By induction hypothesis, this call correctly returns the sum of keys stored in the left subtree that are larger than a, and thus TotalGreater returns the correct sum for the node x as well.

Therefore, TotalGreater works correctly on subtrees of height h.
Running time: As the height of a RB-tree is $O(\log n)$ and in each iteration of the algorithm we go down one level in the tree, the algorithm is run $O(\log n)$ times. As each line takes $O(1)$ time, so does each iteration of the algorithm. Thus in total all recursion together take $O(\log n)$ time.

Exercise 9 *(L3)* Consider an interval tree T containing n intervals with unique endpoints. Describe an $O(\log n)$ time algorithm that, given a number p, returns the interval from T that contains p and has the maximum high endpoint, or NIL if no interval contains p.

Solution:

Highlights:

- We want to be sure that the intervals that may contain p are left of our search path as using *max* we can quickly check if an interval contains p somewhere inside the subtree without completely traversing the subtree. In other words *max* only gives information about the *high*-endpoint, so our search should give information about the *low*-endpoint of all intervals in the subtrees.

Our algorithm will consist of two stages. In the first stage, we traverse a path down the tree where we ensure that all the intervals that lie in a right subtree of the traversed path have a low endpoint that is larger than p. Thus, every interval that is right of the search path can be discarded as it cannot contain p. Thus the interval we are looking for is either on the search path or left of it. We keep track of the biggest possible *high* endpoints we have seen left of the search path (by looking at the *max* value of the root nodes of these subtrees) or on the search path (by explicitly testing checking the *high* endpoint of the node). We maintain a reference to the node where we found the mention of this largest *high* endpoint.
In the second stage, we start at the node we found and search through the subtree until we have found the interval that has a *high*-time equal to *max*.

```
FindRightmostInterval($T, p$)
    Input: an interval tree $T$ and a number $p$
    Output: an interval from $T$ that contains $p$ with maximum high endpoint
$v_{\max} = $ NIL
$max_{high} = -\infty$
$x = $ root[$T$]
while $x \neq$ NIL
    if $low[\text{int}[x]] > p$
        $x = $ left[$x$]
    else $max_{high} = \max\{max_{high}, high[\text{int}[x]], max[\text{left}[x]]\}$
        $v_{\max} = $ node corresponding to $max_{high}$ (either $v_{\max}$, $x$, or left[$x$])
        $x = $ right[$x$]
if $v_{\max} ==$ NIL or $max_{high} < p$ return NIL
$x = v_{\max}$
while $high[\text{int}[x]] \neq max_{high}$
    if $max[\text{left}[x]] == max_{high}$
        $x = $ left[$x$]
    else $x = $ right[$x$]
return int[$x$]
```

(Note: In the pseudocode we assume that $max[\textsc{nil}] = -\infty$.)
Proof of correctness: (sketch) The correctness of the algorithm can be shown using two loop invariants. (We only show the setup here.)
The loop invariant for the first while-loop is the following:

(a) The value of max_{high} is the maximum high endpoint of all the intervals with low endpoints $< low[x]$;

(b) Node v_{max} is either a node with high endpoint of its interval equal to max_{high}, or v_{max} is a node such that all the intervals stored in the subtree rooted at v_{max} have their low endpoints $< p$ and $max[v_{max}] = max_{high}$.

The loop invariant for the second while-loop is the following: the subtree rooted at x contains the interval with high endpoint equal to max_{high} that contains p.
Running time analysis: Both while-loops have at most $O(\log n)$ iterations as at every step we move a level down in a balanced tree. Each iteration a constant amount of work is performed. Thus, the total running time of the algorithm is $O(\log n)$.

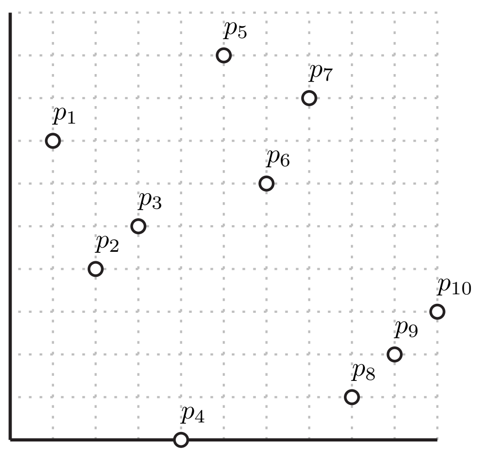
▶ Lecture 9 Range searching

Range Searching

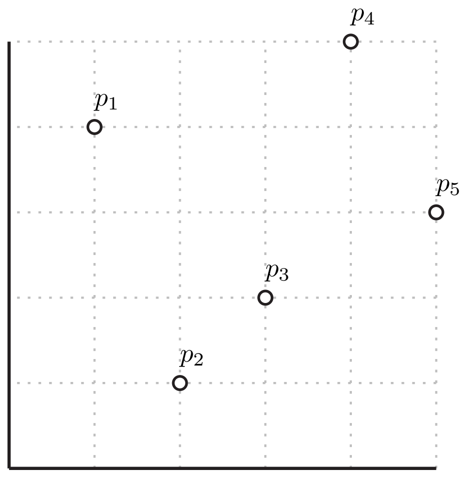
Exercise 10

(a) *(L2)* Build a KD-tree for point set (a). Use the lower median to split the set of points.

(b) *(L2)* Build a 2D range tree (including all associated structures) for point set (b). Use the lower median to split the set of points.



(a)

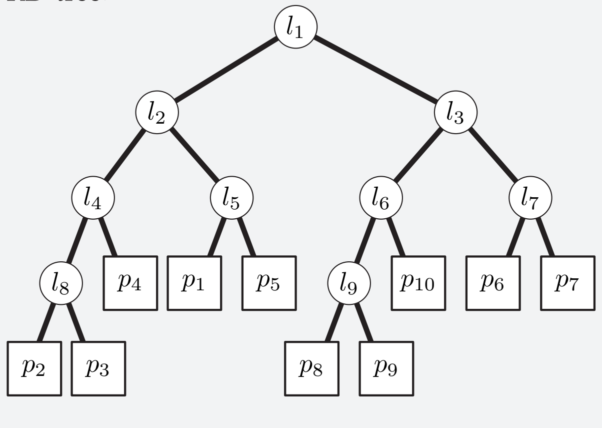


(b)

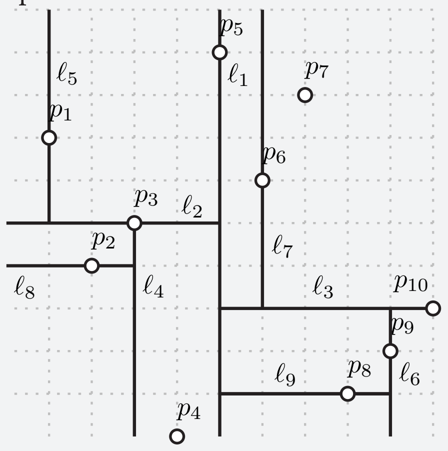
Solution:

(a)

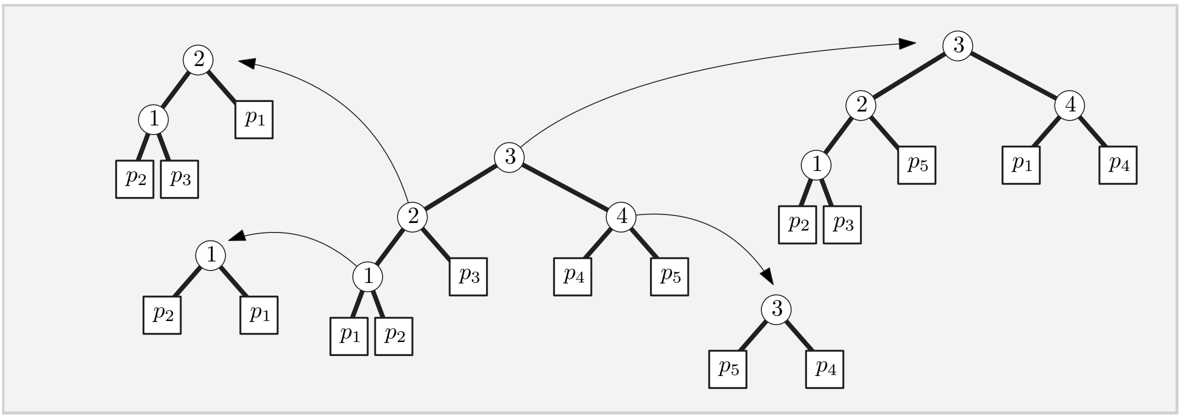
KD-tree:



Splitlines:



(b) 2D range tree:



Exercise 11 *(L3)* Describe an algorithm that, given a (two-dimensional) KD-tree T, returns the point $p \in T$ with minimum x-coordinate. You may assume that all x-coordinates of the points in T are unique.

Solution:
Consider the lines splitting the sets of points when moving down in the KD-tree. For a node z, if the splitting line is vertical, then the point with the minimum x-coordinate cannot lie to the right of this line. Then, we can simply recurse into the left child of z. If the splitting line is horizontal, the point with the minimum x-coordinate can lie on both sides of the line, and we need to check both children of z.

```
MINIMUM($z$, $depth$)
 1  if $z$ is a leaf
 2      return $z$
 3  if $depth$ is even
 4      $minNode$ = MINIMUM(left[$z$], $depth + 1$)
 5  else $minBottom$ = MINIMUM(left[$z$], $depth + 1$)
 6      $minTop$ = MINIMUM(right[$z$], $depth + 1$)
 7      if $minBottom.x < minTop.x$
 8          $minNode$ = $minBottom$
 9      else $minNode$ = $minTop$
10  return $minNode$
```

Initially we call MINIMUM(root[T], 0).
Proof of correctness: We can prove the algorithm correctly finds the point with minimum x-coordinate out of all points stored in the subtree rooted at z by induction on the height of the subtree rooted at z.
Base case: Height of the tree $h = 0$. Then z is a leaf, and the algorithm correctly returns z.
Inductive step:
Induction hypothesis: Assume that MINIMUM works correctly for all subtrees of height $0 \leq k < h$ for some fixed $h > 0$.
We will show that then MINIMUM works correctly for trees of height h as well. As $h > 0$, the algorithm will proceed to lines 3–10. Consider region *region*(z) corresponding to node z.

If $depth(z)$ is even, then the $region(z)$ is split by a vertical line, and the point with minimum coordinate will lie on the left side of this line. In this case the algorithm makes a recursive call to itself on the left child of z. By IH, it will correctly find the point with the minimum coordinate in the region $region(\text{left}[z])$, and thus correctly return the point with the minimum coordinate in line 10.
If $depth(z)$ is odd, then the $region(z)$ is split by a horizontal line, and the point with minimum coordinate could potentially be found in both subregions, below the line and above the line. In this case the algorithm makes a recursive call to itself on the left child of z and on the right child of z. By IH, MINIMUM will correctly find the two points with the minimum coordinates in the regions $region(\text{left}[z])$ and $region(\text{right}[z])$. The algorithm then compares the two points and correctly returns the point with the minimum coordinate.
Thus, MINIMUM works correctly for trees of height h. Therefore, by induction, MINIMUM works correctly for trees of all heights.
Running time analysis: To analyze the running time we will count the number of nodes in the tree visited by the algorithm. Similarly to the analysis of SEARCHKDTREE from the lecture, the number of nodes visited by the algorithm can be counted in the following way: Let $Q(n)$ be the number of nodes visited in a tree of size n. We go down the tree two steps, i.e., consider one vertical and one horizontal split. For a vertical split we only go into the left child, and for horizontal we go into both children. Thus, we get the recursion $Q(n) = 2 + 2Q(n/4)$, which solves to $Q(n) = O(\sqrt{n})$. The amount of work performed at each node is constant, thus the total running time of MINIMUM is $O(\sqrt{n})$.

Exercise 12 *(L3)* Modify the 1DRangeQuery algorithm to report all the numbers stored in a binary search tree that are outside of a given query range. Prove the correctness of your algorithm, and analyze its running time.

Solution:
We modify the 1DRangeQuery and the FindSplitNode in the following way. First, when searching for the v_{split}, when we are going into the left child we will be reporting all the points in the right subtree of the node, and when going down into the right child we will be reporting all the points in left subtree:

```
InvFindSplitNode($T, x, x'$)
v = root[$T$]
while $v$ is not a leaf and ($x' \leq v.x$ or $x > v.x$)
    if $x' \leq v.x$
        ReportSubTree(right[$v$])
        $v$ = left[$v$]
    else ReportSubTree(left[$v$])
        $v$ = right[$v$]
```

After we have found v_{split}, we continue searching for x and x', while reporting the left subtrees along the left branch (when searching for x), and the right subtrees along the right branch (when searching for x'):

```
Inv1DRangeQuery($T, [x : x']$)
$v_{\text{split}}$ = InvFindSplitNode($T, x, x'$)
if $v_{\text{split}}$ is a leaf
    check if the point stored at $v_{\text{split}}$ must be reported
else $v$ = left[$v_{\text{split}}$]
    while $v$ is not a leaf
        if $x \leq v.x$
            $v$ = left[$v$]
        else ReportSubTree(left[$v$])
            $v$ = right[$v$]
        check if the point stored at leaf $v$ must be reported
    $v$ = right[$v_{\text{split}}$]
    while $v$ is not a leaf
        if $x' \geq v.x$
            $v$ = right[$v$]
        else ReportSubTree(right[$v$])
            $v$ = left[$v$]
        check if the point stored at leaf $v$ must be reported
```

Proof of correctness: To prove the correctness of the algorithm we will need to prove the following two claims:

1. Every point that is outside of the interval $[x : x']$ is reported,
2. Every point that is in the interval $[x : x']$ is not reported.

We can proof both of these claims by contradiction. Assume there was a point p outside of the interval $[x : x']$ that was not reported. Without loss of generality let $p < x$. Consider the lowest ancestor of p on the search path from the root to x. We can arrive at a contradiction by arguing about the steps Inv1DRangeQuery performs on this node.
Similarly, assume there was a point p in the interval $[x : x']$ that was reported. We again consider the lowest ancestor of p on the search path from the root to x (or to x'). And we then again arrive at a contradiction by arguing about which exact steps Inv1DRangeQuery performs on this node.
We leave the details of this proof as an exercise.
Running time analysis: The running time of the algorithm is $O(\#\text{nodes visited} + \#\text{points reported})$. The number of visited nodes is $O(\log n)$, as at each step the algorithm either goes into the left or the right child. Then, the total running time is $O(\log n + k)$, where k is the number of reported points.