

BINARY SEARCH TREES AND BALANCED BINARY SEARCH TREES

Chapters 3-3.1, 4-4.2 of GT

DRAFT

DISCLAIMER: *Report an error / typo to the author so that he resolves it.*
These notes are NOT to be distributed outside of NJIT without the expressly written permission of A. Gerbessiotis

Search Trees

Introduction

Search trees are data structures that support a variety of operations that include the following dynamic-set operations. Such operations were defined earlier in Chapter 1. We call them dynamic-set operations because set size might change: an Insert increases the size (cardinality) of the set by one, and Delete decreases it by one. We also call these operations modifying. The remaining operations are probing: the size of the set remains the same prior and after the operation. Moreover if a set has some property (e.g. a search property) prior to the operation, it must also maintain that property after the operation. The ****** operations are probing operations.

Definition 1 (Operations on sets). The following is a list of operations on sets.

- (1) Search**,
- (2) Minimum**,
- (3) Maximum**,
- (4) Predecessor**,
- (5) Successor**,
- (6) Insert,
- (7) Delete.

Definition 2 (Representing a search tree). A search tree is a rooted and ordered tree. For binary search trees in addition they are binary trees. In a search tree we have internal nodes and external nodes. The external nodes are also known as leaves and are NULL nodes. The internal are associated with key values and have children that are external nodes or internal nodes. In some representations of search trees NULL are not represented at all. In the AVL trees that we are going to introduce in this section, but also in red-black trees that we will not, which are classes of balanced binary search trees, we represent NULL nodes.

Definition 3 (Bounded height (search) tree). A tree T is called a bounded height (search) tree if its height is bounded by a logarithmic function of the number of its internal nodes. Therefore $h = O(\lg n)$, where h is the height of the tree, which is the height of its root, and n is the number of internal nodes of T .

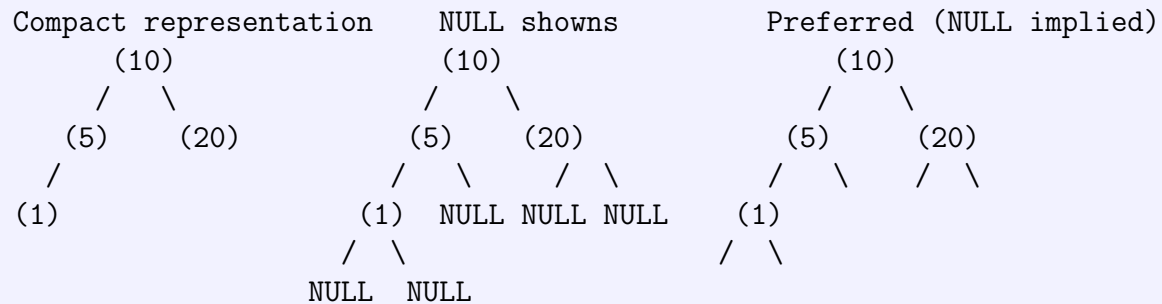
Bounded height search trees that we are going to discuss would be binary search trees. To cover other cases we introduce bounded depth search trees. As a reminder the height of a tree is the height of its root. The height of a node is the length (in edges) of the longest path from the node to a descendant leaf of that node.

Definition 4 (Bounded depth (search) tree). A tree T is called bounded depth (search) tree if the depth of every node is bounded by a logarithmic function of the number of its internal nodes. Therefore $D(u) = O(\lg n)$, where $D(u)$ is the depth of node u which is the distance (in terms of edge) of u from the root of the tree, and n is the number of internal nodes of T . Moreover, we assume that all the external nodes are at the same level. Additional conditions might be imposed on trivial cases where the number of key values (and thus of internal nodes) is very small.

Search trees

Representation

Example 1. In the compact representation below the height of 10 is 2 defined by the path leading to node with key value 1. When NULLs are shown, the height of 10 becomes 3. The height of the root is the height of the tree. (Sometimes height is defined in terms of nodes visited, but it has been previously defined to count edges.)



Search trees

Performance

When we express the running-time of various operations on a search tree T , we first express the running time in terms of the height of T . The objective is to have an algorithm whose running time is $O(h)$.

Another objective is for T to maintain a **balancing property** before and after every operation and especially the modifying operations. It is the balancing property that can lead to the running time of all these operations on T to become $O(\lg n)$, where n is the number of keys in T .

Later, we are going to show that all the dynamic-set operations on a search tree T that were introduced earlier could run in time proportional to the height h of T i.e. in time proportional to $O(h)$.

If the tree T maintains n keys and for binary search trees this implies that the number of internal nodes is also n , then the height h of T can range from $\Theta(\lg n)$ all the way to $\Theta(n)$ thus determining accordingly the running time of all previously mentioned dynamic-set operations.

If the tree is not binary but every node of the tree has a constant number of children (instead of just two) a similar observation is true. Thus an $O(h)$ operation described in term of the height h of T can be mapped to time $O(\lg n)$ or $O(n)$ in terms of the number of key values associated (stored) in T .

By the way, a height of $\Theta(n)$ maps to a degenerate linked list structure, and a height $\Theta(\lg n)$ maps to a complete binary tree if T is a binary tree.

Definition 5 (Binary Search Tree). A binary search tree (BST) T

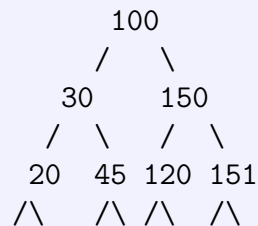
- is a search tree that is organized as a binary tree,
- is represented by a linked data structure every node x of which is an object with
 - a **key** field,
 - a **left** and **right** pointers pointing to the left and right children of x , or to NULL if a child does not exist, and
 - (potentially) a pointer to the **parent** of x . (For $r = \text{root}(T)$, $\text{parent}(r) = \text{NULL}$.)
- Each node of BST T has the **binary-search-tree** property (BSTP) that states that for any node x of a binary search tree,
 - for every node y of the left subtree of x we have $\text{key}(y) \leq \text{key}(x)$, and
 - for every node y of the right subtree of x we have $\text{key}(y) \geq \text{key}(x)$.

Binary search trees

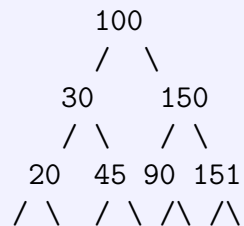
Example

Example 2. We show two examples of a binary tree. One is a BST the other it is not. In one of them the node with key value 90 is to the right of the node with key value 100 that happens to be the root. If the node with 90 is on the right subtree of the node with 100 it is supposed to hold a value at least equal to 100 not less!

A Binary Search Tree



Is this a BST?



Lemma 1. Given n (distinct comparable) keys arranged over the internal nodes of a binary search tree T we can sort those keys in time $\Theta(n)$.

Proof. We perform an inorder traversal of T in linear time $\Theta(n)$. The in-order traversal gives the sorted output sequence. ■

This does not mean that sorting can be done in $\Theta(n)$ time. Building the binary search tree T would require $\Omega(n \lg n)$.

BST operations

Search

```
BSTSearch(T, value)
    TSearch(root(T),value);
```

```
TSearch(x,value)
1. if (x==NULL) or (value == key(x))
2.   return(x);
3. if value < key(x)
4.   return  TSearch(left(x),value);
5. else
6.   return  TSearch(right(x),value);
```

Theorem 1. *For a BST T , operation $BSTSearch$ requires time $O(h)$.*

In most cases we do not know the height of the tree and the available problem size is n , the number of its keys. Thus an $O(h)$ bound does not make sense. In the worst case, a BST T is of height $n - 1$, it looks like a linked list, and for those case running time $O(h)$ translates into an $O(n)$.

Unless we are given the height h of the tree in terms of n , does it make sense to use the $O(h)$ bound.

Theorem 2. *For a BST T with n keys, operation $BSTSearch$ requires time $O(n)$.*

BST operations

Min and Max

An implementation of a `BSTMax` and similarly of a `BSTMin` operation in a BST follows.

```
BSTMin(T)
    TMin(root(T));

TMin(x) /* returns an element not a key */
1. while (left(x) != NULL)
2.   x = left(x);
3. return(x);
```

Whereas a Min operation traverses T through `left(x)` pointers, a Max operation traverses T through `right(x)` pointers.

Theorem 3. *For a BST T , operation `BSTMin` and also `BSTMax` each requires time $O(h)$.*

Theorem 4. *For a BST T with n keys, operation `BSTMin` and `BSTMax` each requires time $O(n)$.*

BST operations

Successor and Predecessor

A `BSTSuccessor` (similarly a `BSTPredecessor`) operation returns the element which is the successor (predecessor) of x i.e. it returns the element whose value is the smallest (largest) one among the keys whose values are greater (smaller) than $\text{key}(x)$. The code for `BSTPredecessor(T,x)` is analogous and symmetric to that of `BSTSuccessor(T,x)`.

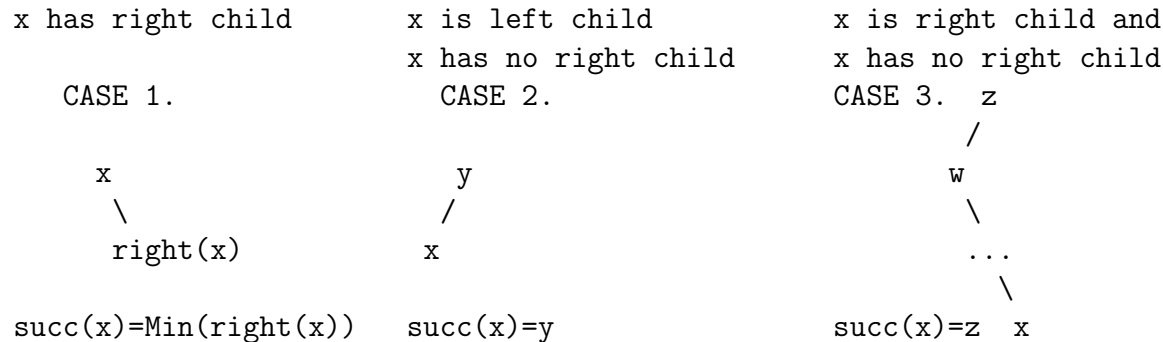
```
BSTSuccessor(T,x)
1. if (right(x) != NULL) return(Min(right(x)));
2. y=parent(x);
3. while (y !=NULL) and (x==right(y)) {
4.     x=y
5.     y=parent(y); }
6. return y;
```

Theorem 5. *For a BST T , operation `BSTSuccessor` and also `BSTPredecessor` each requires time $O(h)$.*

BST operations

Successor and Predecessor

Proof. There are three major cases to consider. If none of them is applicable then x has no successor and x is Max of T .



Case 1 (**x has a right child**). Then successor(x) is in the tree rooted at $\text{right}(x)$ and it is the element with minimum key value in that subtree. It is retrieved through the line 1 call to Min.

Case 2 (**x has no right child**). Let x be the left child of its parent y . Then the successor of x is its parent y and it is retrieved through line 2 (the loop of lines 3-5 is not executed and y is returned).

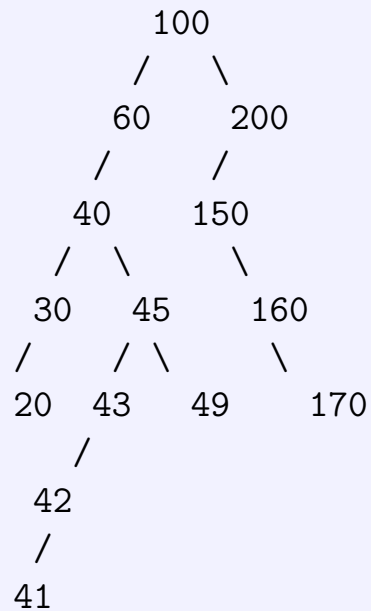
Case 3 (**Otherwise x has no right child and x is the right child of its parent y**). We move upwards until we find an ancestor w of x that is the left child of his parent. If such a node exists then $\text{succ}(x) = \text{parent}(w)=z$; if not, node x has no successor (i.e. it is the maximum). ■

Theorem 6. For a BST T with n keys, operation *BSTsuccessor* and *BSTPredecessor* each requires time $O(n)$.

BST operations

Examples

Example 3.



Search : Search(100), Search(40), Search(41), Search(44)

Min vs Max

Predecessor

symmetric to

Successor : Successor(45) , Successor(100) ,

Successor(41)

Successor(43)

Successor(49)

Successor(200)

Search(44) vs Insert(44)

BST operations

Insert

Operation **BSTInsert** inserts a value v into a binary search tree T rooted at $r = \text{root}(T)$. Insert works similarly to BSTSearch by traversing a path of T starting at the root until it reaches the node where it is to be inserted. For a search the conclusion would have been a not found. For Insert we just insert the key as needed.

```
BSTInsert(T,w)
1. y=NULL; x=root(T)
2. while x!=NULL {
3.   y=x;
4.   if key(v) < key(x)
5.     x=left(x);
6.   else
7.     x=right(x);
8. }
9. parent(v)=y;
10 if y==NULL
11.  root(T)=v;
12.else {
13.  if key(v) < key(y)
14.    left(y) = v;
15.  else
16.    right(y)=v;
  }
```

Theorem 7. *For a BST T , operation **BSTInsert** requires time $O(h)$.*

Theorem 8. *For a BST T with n keys, operation **BSTInsert** requires time $O(n)$.*

BST operations

Delete

```
BSTDelete(T,z)
1. if (left(z)==NULL || right(z) ==NULL)
2.     y=z;
3. else y=successor(z); // y is node to splice out
4. if (left(y)!= NULL)
5.     x=left(y);
6. else x=right(y); // node to be attached to parent(y)
7. if (x !=NULL)
8.     parent(x)=parent(y); //splice out y in lines 7-13
9. if (parent(y)==NULL)
10.    root(T)=x;
11.else if (y==left(parent(y)))
12.    left(parent(y))=x;
13.    else right(parent(y))=x;
14.if (y !=z) // we work on the successor of y
15.    key(z)=key(y) //copy successor(z).key into z.key
16.return y
```

Theorem 9. For a BST T , operation $BSTDelete$ requires time $O(h)$.

BST operations

Delete

Proof. BSTDelete deletes a node z from BST T and takes as an argument a tree T and a pointer to the node to be deleted z (i.e. we do not need to search for z). Three cases related to a deletion operation are considered. In fact the first two cases are treated the same way. We could combine them together by just saying (a) z has a child that is external node (if z has no or one child), (b) z has two children (i.e. no child is an external node).

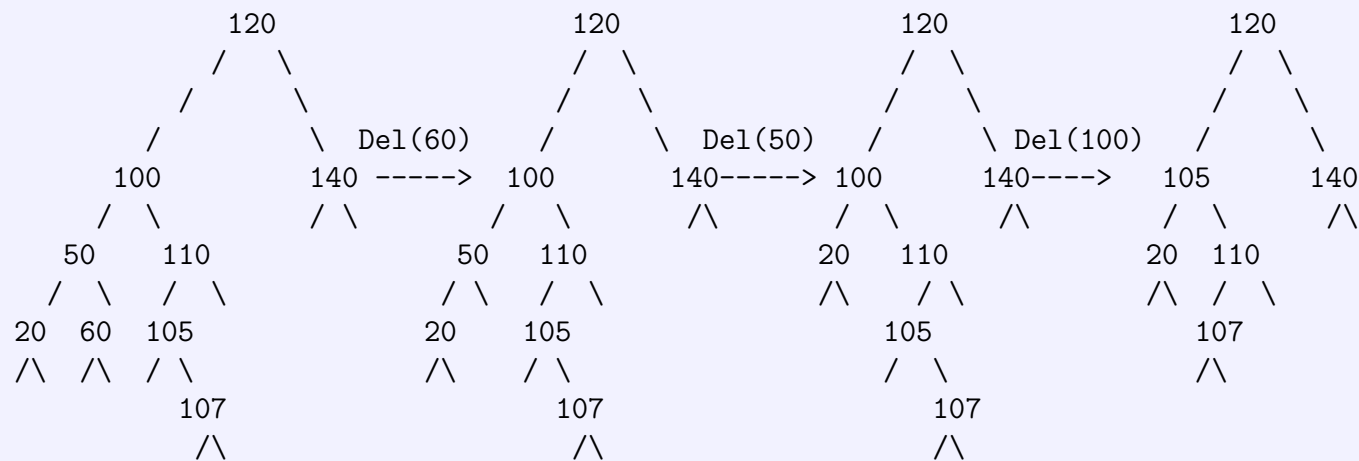
- z has no children. Splice out z so that $\text{left}(\text{parent}(z))$ or $\text{right}(\text{parent}(z))$ point to NULL and not to z .
- z has a single child. Then we spliceout/remove z by making its $\text{parent}(z)$ point to $\text{child}(z)$ rather than z .
- z has two children. Splice out $\text{successor}(z)$ and replace the contents of z ($z.\text{key}$) with those of $\text{successor}(z)$ ($\text{successor}(z).\text{key}$). [The predecessor(z) can ALSO BE USED in deletion; the use of the predecessor would be more prevalent on general search trees in the next chapter.]



Theorem 10. *For a BST T with n keys, operation BSTDelete requires time $O(n)$.*

BSTDelete Example

Example 4. Some examples of BSTDelete follow.



Proof. (In the example the name Delete or the name Del refer to BSTDelete.)

Case 1. Delete($z = 60$). z has no children. Remove z by making $\text{right}(\text{parent}(z)) = \text{NULL}$ deletion is completed.

Case 2. Delete($z = 50$). After 60 is deleted, 50 has only one child. Case 2 applies. Splice out 50, by making $\text{left}(100) = \text{left}(50)$.

Case 3. Delete($z = 100$). Node 100 has two children 20 (after the two previous deletions) and 110. Find $\text{succ}(100)$ in its right subtree, which is 105, replace key of node 100 with the 105 key, and splice out node 105 (effectively make 107 the left child of 110). ■

Theorem 11. *All BST operations have running time that is proportional to the height h of the binary search tree T . Therefore, if h is problem size and $T(h)$ (symbol overload) is the running time of an operation for a binary search tree of height h , then $T(h) = O(h)$ and thus in the worst-case performance is also $\Theta(h)$ when the tree is traverse from the root to the leaf that determines the height. For a binary search tree T of height h with n keys and thus n internal nodes, if problem size is n instead, and express running time in terms of n we conclude that $T(n) = O(h) = O(n)$. This is because a binary tree with n nodes has maximum height $h = n - 1$.*

DRAFT

Balanced binary search trees

Introduction

Although BSTs look interesting, their worst case time performance is no better than that of an array or a linked list. However, BSTs perform better on the average, as on the average case with random insertions, the height of the formed BST is $O(\lg n)$.

In the following section we are going to introduce classes of binary search trees whose worst case performance is optimal, i.e. the worst case running time of any of the seven introduced operations is $O(\lg n)$ independently of any assumptions on the distribution of the keys. Note that n indicates the number of keys but also the number of internal nodes of the BST cited.

Definition 6 (Balanced BST). A BST on n keys (and internal nodes) whose height is $O(\lg n)$ is called a **balanced** binary search tree (BBST).

Definition 7 (Balancing property). Every BBST maintains a balancing property that guarantees that its height is $O(\lg n)$ for a BBST with n keys.

Balanced binary search trees

Operations

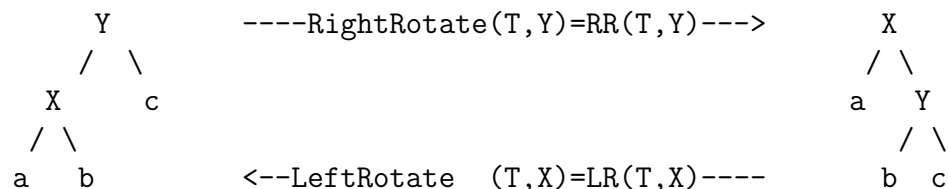
Proposition 2 (BBST probing operations). *Operations $BBSTMin$, $BBSTMax$, $BBSTPredecessor$, $BBSTSuccessor$, $BBSTSearch$ are identical to Min , Max , $Predecessor$, $Successor$, $Search$ defined on a BST . However the running time of $O(h)$ for those BST operations (that used to translate into an $O(n)$ time bound) now translates into an $O(\lg n)$ running time for the corresponding $BBST$ operation.*

Proposition 3 ($BBSTInsert$ and $BBSTDelete$). *Operations $BBSTInsert$ and $BBSTDelete$ have a first phase identical to BST operations $Insert$ and $Delete$ respectively. Thereafter if a violation of the balancing property is detected then a $FixInsert$ or $FixDelete$ operation is initiated in a second phase. The combined costs of both phases is $O(\lg n)$, where n is the number of keys (and also the number of nodes) in the $BBST$.*

BBST

Maintaining the balancing property: Rotations

The Insert and Delete operations on some BBST T require that a certain structural operation that is known as rotation be performed on a subtree of T . A rotation can be of two types a single rotation or a double rotation. A double rotation consists of two single rotations. Any single rotation can be one of two kinds: a left or a right rotation. Left and Right rotations are defined in picture format as follows.



a, b, c , are subtrees (or NULL) , X and Y are nodes of T

FIGURE: SINGLE LEFT AND RIGHT ROTATIONS.

Rotation operations take constant time $O(1)$ to perform and can easily be described through pointer assignments of a $O(1)$ number of nodes. A more complex operation is a double rotation shown below in terms of two rotations.

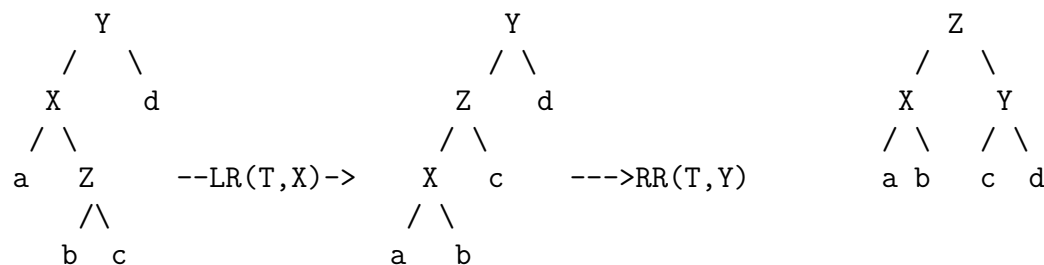


FIGURE: A DOUBLE ROTATION.

Definition 8 (AVL tree structure). An AVL tree T is a BST tree. It has the structure of a BST tree and thus there is a $key(x)$, $left(x)$, $right(x)$ and $p(x)$ at every internal node $x \in T$. Moreover an AVL tree maintains for every internal node x , the balance (or balancing) factor $b(x)$ of x defined as follows:

$$\forall T \in AVL, \forall x \in T : b(x) = h(left(x)) - h(right(x)).$$

By default for the external NULL nodes, $h(\text{NULL})$ is 0. The balancing property of an AVL tree maintains that

$$\forall T \in AVL, \forall x \in T : |h(left(x)) - h(right(x))| \leq 1.$$

In other words,

$$\textbf{Balancing property} : \quad \forall x \in AVL(T) : |b(x)| \leq 1.$$

In order to keep things simple, we would maintain at every node x the height $h(x)$. It would then be straightforward to compute $b(x)$. We shall also refer to as the b-factor of a node x for $b(x)$.

Corollary 1. A $b(x)$ at every node requires two extra bit to store the three possible states 0,+1 and -1. A $height(x)$ at every node requires $O(\lg(\lg n))$ bit at every node since the height of x is $O(\lg n)$.

Note that for an AVL tree on $n < 2^{128}$ keys, we have $\lg n < 128$ and $height(x)$ would require one or maybe two byte of storage space per node.

Definition 9 (Red-black tree structure). A red-black tree is another class of a balanced Binary Search Tree. In order to maintain a balancing property we need to store one extra bit at every node x , the color $c(x)$ of node x . The two color options of red or black require one bit only.

Even if $c(x)$ requires one bit, that bit is to be stored into one byte. The C++ Standard Template Library implements for example a Red-Black Tree data structure. And in Java version 8 and onwards, a HashMap which is the implementation using a Hash table of the Map interface of Java uses red-black trees instead of variable length arrays that were used previously.

DRAFT

AVL trees

Introduction to a class of BBST

The “AVL” of an AVL tree comes from the initials of the last names of its three creators: Adelson, Velskii, Landis. Let us summarize the attributes of an AVL tree below.

Note 1 (AVL tree description). The root of an AVL tree T is represented by $r(T)$ or just r . The left and right children of x will be denoted by $left(x)$ and $right(x)$ and are the roots of the left and right subtree of x . The height of node x is denoted by $h(x)$. The height $h(T)$ of tree T is the height of its root, i.e. $h(T) = h(r)$. The balance factor of x is $b(x)$, where $b(x) = h(left(x)) - h(right(x))$. In AVL trees for every node x , $|b(x)| \leq 1$. NULLs are represented as (external) nodes of height 0.

The following serves as a recursive definition of an AVL tree.

Proposition 4 (AVL tree). *An empty binary search tree T is an AVL tree. If T is not empty and has root r , and T_L and T_R are its left and right subtrees with height $h(left(r))$ and $h(right(r))$ respectively, then T is an AVL tree if and only if*

1. T_L and T_R are AVL trees, and
2. $|h(left(r)) - h(right(r))| = |b(r)| \leq 1$.

Sometimes we call an AVL tree an AVL BST tree. In that case AVL emphasizes its structure ($b(x)$ at every node), where as AVL BST emphasizes the property $|b(x)| \leq 1$ that is present on every x . In that case the following might make sense.

Proposition 5 (AVL BST tree). *An AVL BST is an AVL tree with the binary search tree property. However we shall use interchangeably the terms AVL BST and AVL to mean the same thing.*

Theorem 12 (Height of an AVL tree). *The height of an AVL tree T with n keys is $O(\lg n)$.*

Proof. Let N_h be the minimum number of **internal** nodes in an AVL tree of height h . The internal nodes of an AVL tree are the nodes holding keys. Thus the number of keys is equal to the number of internal nodes. Furthermore, NULL nodes are represented and are external nodes, and of course they do not have keys.

Obviously, $N_0 = 0$ and $N_1 = 1$. If this is disorienting, then be reminded that a single NULL node maps to an empty AVL tree consisting of one external node, the NULL, and 0 internal nodes. Its height is zero and this leads to $N_0 = 0$. Furthermore, a single root internal node with key value that has two NULL left and right subtrees (children) gives rise to an AVL tree of height 1 (distance from the internal node root to the external node NULL) with one key value (internal node).

If the height of the tree is h so is the height of its root by definition. For root r , nodes $left(r)$ and $right(r)$ can have heights $h - 1$ and $h - 1$, or $h - 1$ and $h - 2$, or $h - 2$ and $h - 1$ respectively, and are also roots of AVL trees themselves. As both subtrees are AVL trees the total number of nodes in the subtrees is at least $N_{h-1} + N_{h-1}$, or $N_{h-1} + N_{h-2}$, or $N_{h-2} + N_{h-1}$. The smallest number of nodes in the two subtrees is thus $N_{h-1} + N_{h-2}$ given the monotonicity of N_i . Adding to this internal node count the root r we obtain for N_h the following.

$$\begin{aligned} N_h &= N_{h-1} + N_{h-2} + 1 \Rightarrow N_h + 1 = (N_{h-1} + 1) + (N_{h-2} + 1) \Rightarrow \\ F_h &= F_{h-1} + F_{h-2} \end{aligned}$$

with boundary conditions $N_0 = 0$, $N_1 = 1$. If we substitute $N_h + 1 = F_{h+2}$ we get a Fibonacci recurrence as shown above with $F_0 = 0$, $F_1 = 1$, as discussed in Chapter 1. Therefore if the Fibonacci sequence is 0, 1, 1, 2, 3, 5, 8, 13 the sequence of the (minimum internal node) sizes of AVL trees of height 0, 1, 2, 3, 4, 5 is 0, 1, 2, 4, 7, 12 etc. This also means that

$$N_h = F_{h+2} - 1 = \frac{\phi^{h+2} - \bar{\phi}^{h+2}}{\sqrt{5}} - 1 \geq \phi^{h+2}/\sqrt{5} - 3/2$$

where $\phi = 1.61803\dots$ and $\bar{\phi} = -0.61803\dots$, and we used above the fact that $\bar{\phi}^{h+2}/\sqrt{5} < 1/\sqrt{5} < 1/2$.

Since $N_h \geq \phi^{h+2}/\sqrt{5} - 3/2$, the number of nodes n in an AVL tree of height h would be at least the minimal number N_h i.e. $n \geq N_h$, thus implying that $h \leq 1.44 \lg(n + 2)$ after some calculations. This is equivalent to saying that $h = \Theta(\lg n)$ for an AVL tree of n keys and thus n internal nodes. ■

AVL trees Properties

Example 5.

NOTE: NULL nodes are represented (and count towards the height)

N_h : minimum number of nodes with key values in an AVL tree of height h

$N_0 = 0$: NULL $h=0$

$N_1 = 1$: o $h=1$
 / \
 NULL NULL

N_h	:	o	h		o	h		o	h
		x	x	OR	x	x	OR	x	x
$h-1$	---		---	$h-2$	---		---	$h-1$	---
	---		---		---		---		---

MINIMAL TREE of height h : KEY VALUES

Root contributes 1 node

One of 2 subtrees MUST be height $h-1$ and contributes N_{h-1}

Other subtree is of height $h-1$ or $h-2$ and contributes $\geq N_{h-2}$

Thus $N_h = N_{h-1} + N_{h-2} + 1$

AVL trees

Properties alternative proof

Proof. **A simpler asymptotic analysis.** A simpler derivation is based on the derived equation

$$N_h = N_{h-1} + N_{h-2} + 1.$$

Then

$$N_h = N_{h-1} + N_{h-2} + 1 \geq N_{h-2} + N_{h-2} + 0 \geq 2N_{h-2}.$$

Furthermore, by using the iteration method for the recurrence

$$N_h \geq 2N_{h-2}$$

we obtain that

$$N_h \geq 2N_{h-2} \geq 2^2 N_{h-4} \geq \dots \geq 2^i N_{(h-2i)}.$$

The base case is $h - 2i = 2$ or $h - 2i = 1$ and thus $i = h/2 - 1$ or $i = (h - 1)/2$. This leads to the observation that $N_h \geq 2^{h/2-1}$ i.e. with $n \geq N_h$ as before we have $h \leq 2 \lg n + 2$, a less tight upper bound than through the Fibonacci sequence considerations.

$$n \geq N_h \geq 2^i N_{(h-2i)} \implies n \geq N_h \geq 2^{h/2-1} \implies h \leq 2 \lg n + 2.$$

■

We know from Subject 3 that a binary tree with n internal nodes (that have key values in an BST or an AVL tree) has height at least $\lg n$. Therefore we can strengthen the previous result as expressed in Theorem 12 asymptotically as follows in the form of Theorem 13.

Theorem 13 (Height of an AVL tree). *The height of an AVL tree T with n keys is $\Theta(\lg n)$.*

Probing operations and Modifying operations

The following is a consequence of Theorem 12.

Lemma 6 (Probing operations in an AVL tree). *All probing operations in an AVL tree require $O(\lg n)$ time if performed on an AVL tree with n keys. $BBSTMin$, $BBSTMax$, $BBSTPredecessor$, $BBSTSuccessor$, and $BBSTSearch$ directly translate into $AVLMin$, $AVLMax$, $AVLPredecessor$, $AVLSuccessor$, and $AVLSearch$ without any modification. And of course $BBSTMin$, $BBSTMax$, $BBSTPredecessor$, $BBSTSuccessor$, and $BBSTSearch$ are indeed Min , Max , $Predecessor$, $Successor$, $Search$ of a BST .*

Proof. This is a consequence of Proposition 2 and Theorem 12 ■

Lemma 7 (Modifying operations in an AVL tree). *Insert and Delete operations in an AVL tree referred to hereafter as $AVLInsert$ and $AVLDelete$ also require $O(\lg n)$ time if performed on an AVL tree with n keys.*

Proof. This is a consequence of Proposition 3 and Theorem 12 ■

Probing operations and Modifying operations

Note that AVLInsert and AVLDelete might disturb the balancing property of nodes affected by the insertion or deletion operation. Thus after an insertion of a key or deletion of a key in an AVL tree, the tree might cease to be an AVL tree even if it is a BST tree. The AVL property needs to be reconstituted before the completion of the AVLInsert or AVLDelete operation. Thus an AVLInsert operation performs not just an insertion but also a maintenance of the balancing property following the insertion of a key. Likewise an AVLDelete operation performs not just a deletion but also a maintenance of the balancing property following the deletion of a key.

Proposition 8 (AVL tree that is empty). *An empty tree is an AVL tree.*

Lemma 9 (AVL tree existence). *AVL trees exist for any $n \geq 0$.*

Proof. This is a consequence of Proposition 8 serving as the base case of an inductive argument, and of Proposition 3 for Insertion serving as the inductive step. ■

We have already proved Theorem 12. Then we are going to describe operations AVLInsert and AVLDelete and show that Lemma 7 holds.

AVL trees

AVL Insert operation preliminaries

Proposition 10. *For any node w , $b(w)$ can be $-2, -1, 0, +1, +2$ after the Insertion.*

Proof. Before the Insertion every node had a b -factor of $-1, 0, +1$. The insertion of a single node can cause the subtraction of a $+1$ or a -1 from the b -factor of any node. ■

Proposition 11. *If for some node y , $b(y) = 2$ after the Insert, then $b(y) = 1$ before the Insert. Similarly, if $b(y) = -2$ after the Insert, it was $b(y) = -1$ before.*

Proof. Obvious. ■

Proposition 12. *The b -factors (balance factors in short) of nodes from the root to the inserted node w can only be affected.*

Proof. Obvious. ■

Proposition 13. *For an inserted node w , let $ca(w)$ be the closest to w ancestor of w which is unbalanced i.e. has a b -factor $+2, -2$. Let $fr(w)$ be the node farthest from the root in the path from r to w that is $+1, -1$ BEFORE the insertion (when the determination where to insert w has been finalized but no pointers have been changed nor height values updated). Then $fr(w) = ca(w)$.*

Proof. Node $fr(w)$ must exist in an unbalanced tree. If such a node does not exist, this means that all nodes in the path from the root to w have b -factor 0; but then the insertion of w could not change that 0 into a $+2$ or -2 and thus the tree remains balanced, a contradiction. Therefore, **if the tree is unbalanced after the Insertion of w , $fr(w)$ MUST EXIST.** $fr(w)$ will be the first node whose value will be changed into a $+2$ or -2 , but since all other nodes from $fr(w)$ to w must have a b -factor of 0 before the insertion, (due to the maximality of $fr(w)$), then $ca(w) = fr(w)$. ■

Proposition 14. *$fr(w)$ can be identified while the traversal from the root r to the insertion position of w takes place. Then $ca(w) = fr(w)$ is also identified.*

Proof. Obvious. ■

Proposition 15. *All nodes from $fr(w)$ to w have b -factor 0 (before the insertion).*

Proof. Obvious. ■

Theorem 14 (AVLInsert). *Insertion into a AVL tree with n keys requires $O(\lg n)$ time.*

Proof. When a node is inserted into an AVL tree, it is inserted according to the algorithm we described for unbalanced BST that is BSTInsert which is also identical to the first phase of BBSTInsert implied in a prior discussion. The insertion however, may cause an imbalance because of the increased height in the subtree of the inserted node; in some cases it may even cause a previously unbalanced subtree to become balanced. If there are no balancing problems i.e. all nodes in the path from the root to a leaf before the insertion of w have b -factor 0, then nothing needs to be done. Otherwise, according to Proposition 12, the balance factors of nodes in the path from the root to the inserted node w will be affected after the insertion of w . We can restore the balance of these nodes by a simple scheme that locates unbalanced nodes and restores the balanced property. We identify separately all the important nodes during that Insertion.

- **1st Node w** : The node that is being inserted. By Proposition 13, node $ca(w) = fr(w)$ is uniquely defined after the insertion of w in the tree; this identification can take place while w is being inserted into the tree. Let us refer to this node, the closest ancestor of w that is not balanced, as z from now on.
- **2nd Node z** : The ancestor of w that satisfies $ca(w) = fr(w)$. z has a child of higher height. If such child does not exist, z is balanced, which leads to a contradiction; everything is balanced and we are wasting our time! In addition this child is in the same subtree of z as w . Call this child of z , node y .
- **3rd Node y** : The child of z in the path to w . Let x be the child of y of higher height; the height tiebreaker between the two possible choices for x is the node that is an ancestor of w . It may or may not be the case that x is w . Note that the reason z became unbalanced was that y had a height of two more than its other sibling in the tree after the insertion of w . We then FIX the imbalance by calling Restructure with argument node x .
- **4th Node x** : The child of y of higher height (tiebreaker the node in the path to w).

The proof continues on the next page. ■

AVL trees

AVLInsert operation continued

Proof continued.

- // Reestablishes the balance of the subtree rooted at A.
Restructure (x) // z and y also exist and are parent, grandparent of x.
1. (a,b,c) be an inorder listing of x,y,z.
 2. Let T1, T2, T3, T4, be an inorder listing of the subtrees of x,y,z not rooted at these nodes.
 3. Replace the subtree rooted at z with the one rooted at b.
 4. Make a the left child of b and make T1/T2 the left/right subtrees of a.
 5. Make c the right child of b and make T3/T4 the left/right subtrees of c.

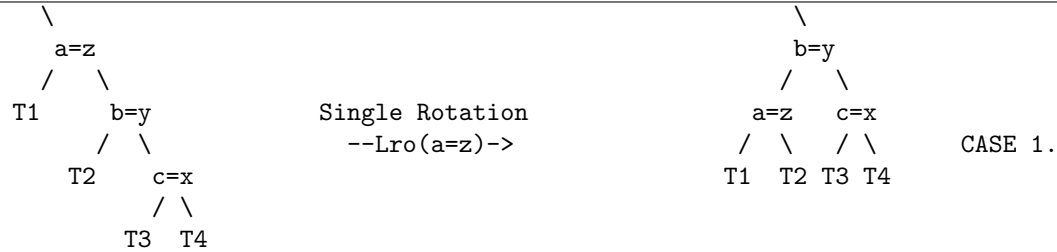
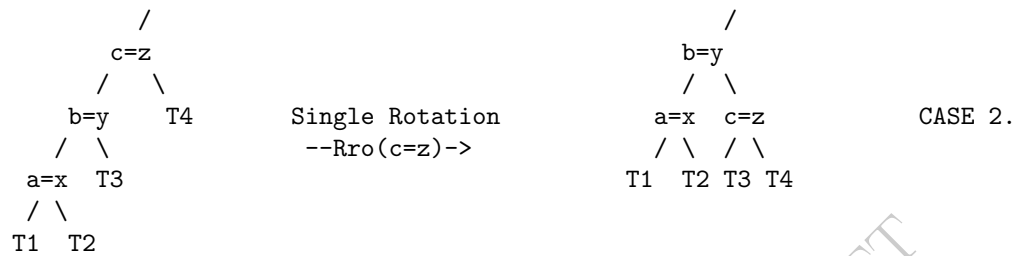
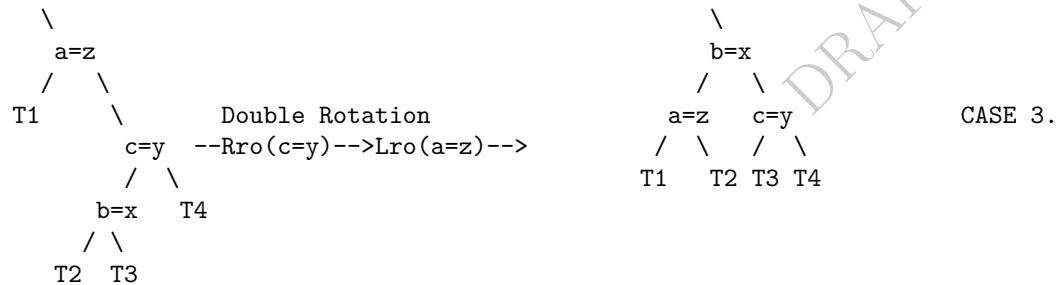
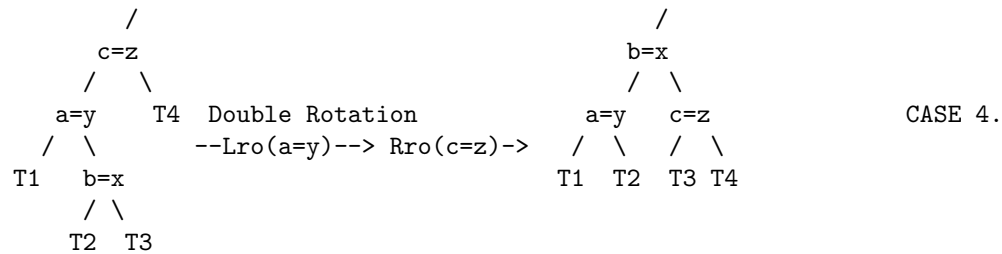
There are four cases to be considered as shown in the Figure on the next page.

We deal with each one of the four cases by performing a rotation operation (single or double as needed). Note that if $b = y$ we need a single rotation; if $b = x$ a double rotation is required. Note also that the inorder traversal of all the nodes remains the same and only $O(1)$ nodes are affected by the rebalancing/restructuring operation. Finally, the restructuring operation that results b being the root of the subtree whose original root was c rebalances the tree globally.

Note that before the rotation (single or double) the subtree rooted at z became unbalanced and this caused imbalance in other nodes higher up than z towards the root. By rearranging that tree that becomes rooted at b things got ironed out not only locally but globally, i.e. b -factors of $+2, -2$ also got eliminated on nodes closer to the root than z . **Therefore for insertion only one rotation (single or double) is needed.** The cost of Insertion including the rebalancing/restructuring is obviously $O(\lg n)$.

The figures involving the four cases are on the next page. ■

AVLInsert operation (cases depicted)



Theorem 15 (AVLDelete). *Deletion from an AVL tree with n keys (and thus n internal nodes) requires $O(\lg n)$ time.*

Proof. The AVLDelete operation proceeds like the BSTDelete operation and its three cases. Given that the AVL tree balancing property might get violated after deletion at the conclusion of AVLDelete node x is identified and Restructure(x) is issued similarly to AVLDelete. However in AVLDelete a Restructure higher in the tree might result. Given that a single Restructure is a constant time operation and the recursive calls triggered higher up in the tree are $O(\lg n)$ the additive cost is asymptotically no more than the $O(\lg n)$ contributed by BSTDelete.

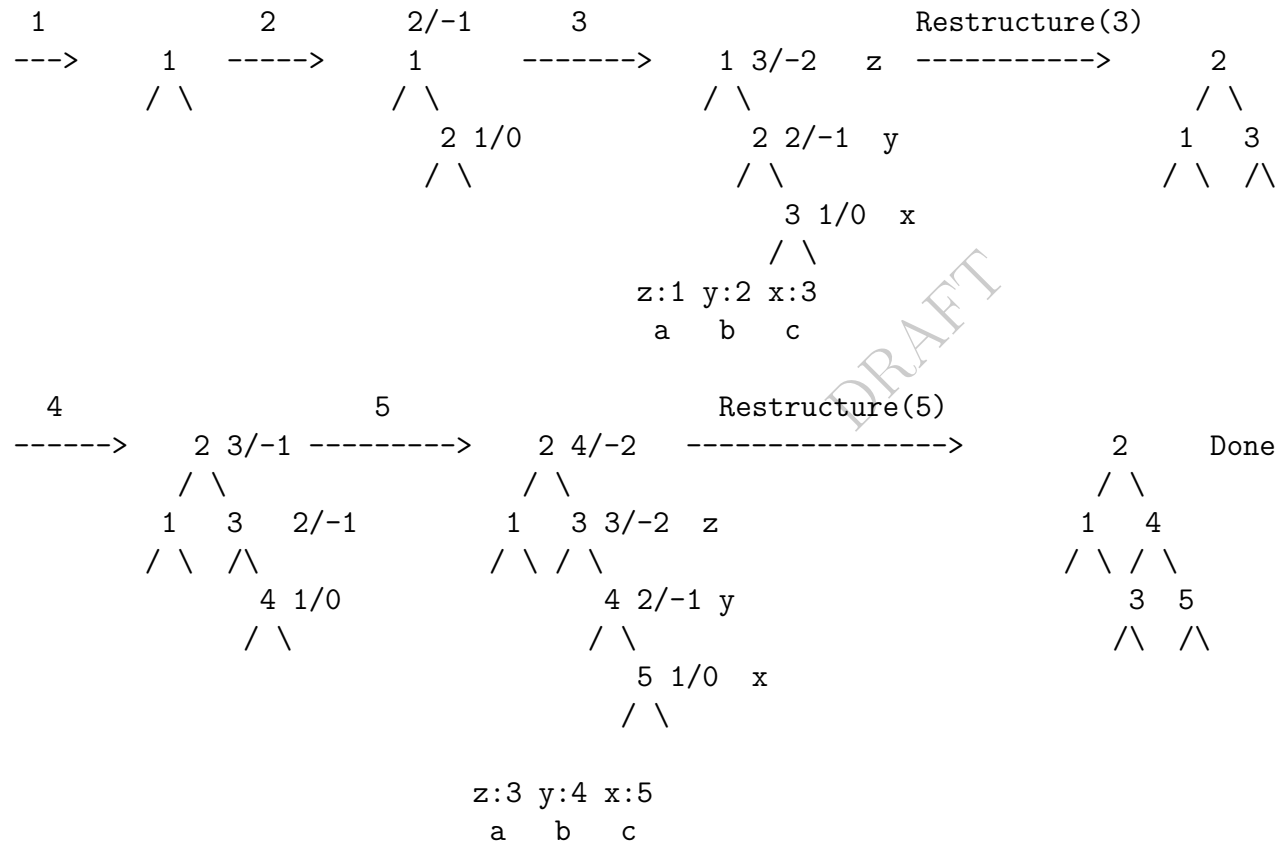
- **1st Node v** : Node that is being deleted. During a Delete(v) operation the node that is physically removed, i.e. spliced out may or may not be v . It can be v (cases 1 or 2 of Deletion) or the successor of v (case 3).
- **2nd Node q** : Node that is being spliced out (v or the successor of v). The node that is important however, is the parent of the spliced out node. We call this node w in general. Depending on which of the three cases was followed, w can be found easily during the Delete(v) operation.
- **3rd Node w** : The parent of q .
- **4th Node z** : We identify the first node in the path from w to the root that is unbalanced. Similarly to AVLInsert we call this node z .
- **5th, 6th Nodes y, x** : Its (of z) child on the other side of higher height (not the one to which w belongs which has lower height due to the deletion) **is called** y and one of the two children of y of higher height **is called** x .

We similarly to Insert perform a **Restructure** operation. Such a Restructure operation might decrease the height of the subtree previously rooted at z but now rooted at the node called b . Therefore we need to move higher up in the tree identifying further imbalances. This way the running time of Delete on an AVL tree might require time $O(\lg n)$ because so many rebalancing/restructuring operations need to be performed. ■

AVL trees

Examples of AVLInsert

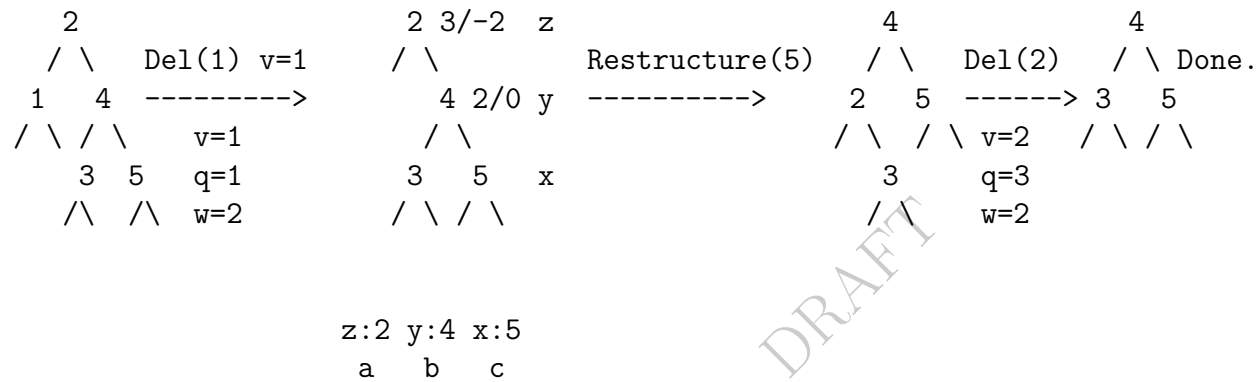
Example 6. We insert keys 1,2,3,4,5 in an initially empty AVL tree. At each node, we occasionally show two numbers x/y , where x is the height and y is $b(x)$.



AVL trees

Examples of AVLDelete

Example 7. We delete keys 1,2 from the AVL tree constructed in the previous example.



Notice Problem Set with Solutions

These Problem Sets are not for credit. They are provided as are for your practice. Several problems are provided with solutions. These can be discussed with the instructor after a student has studied the relevant material. Some other problems are provided with either no solutions at all or a hint of a solution. They are there for your reference and will not be discussed further. For some solutions provided you might have to fill in details related to calculations or intermediate steps.

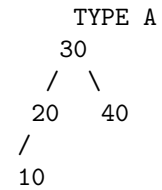
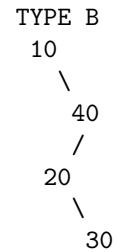
Before you try these problems: (a) Study the designated textbook, (b) use the Subject notes (typewritten notes available through the course web-page from the password-protected section C) to review your study of the designated textbook material, (c) resolve the examples done in class, (d) then practice with some of these problems PS (Problem Set), and then if you have time, (e) solve some of the problems of the designated textbook, and (f) solve more problems from the PS especially the ones with minimal solution information.

Several PS problems appeared in past exams or homeworks.

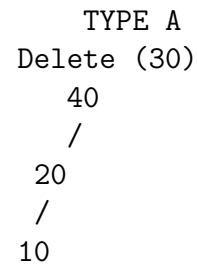
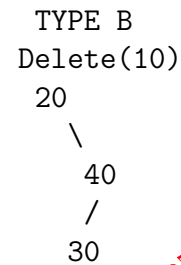
Copyright (c) 2020-2025. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set Problem 1.

Delete the root from the following BST.



Proof.



Copyright (c) 2020-2025. Alex. Gerbessiotis. All rights reserved
Not to be displayed outside of copyright holder's web-page



Problem Set
Problem 2.

a. Suppose that we have numbers between 1 and 1000 in a binary search tree and want to search for the number 363. Which ones of the following sequences (one or more) could NOT be the sequence of nodes examined?

- i. 2, 242, 301, 398, 330, 399, 363.
- ii. 924, 229, 911, 244, 898, 258, 243, 363.
- iii. 925, 202, 911, 240, 900, 245, 363.
- iv. 2, 390, 383, 219, 265, 380, 379, 278, 363.

Justify your answer.

Proof. Sequence (i) cannot be a search sequence as 399 is in the left subtree of 398. Sequence (ii) cannot also be a search sequence as 243 is in the right subtree of 244. ■

Copyright (c) 2020-2025. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page.

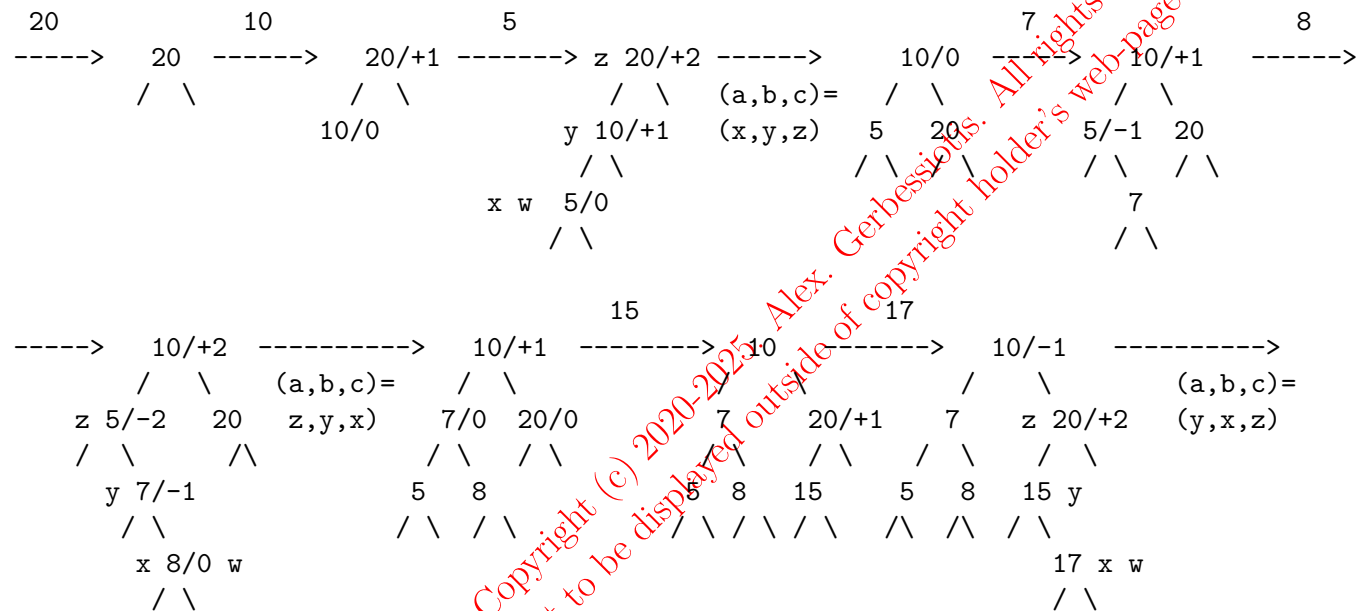
Problem Set Problem 3.

20, 10, 5, 7, 8, 15, 17, 22, 21

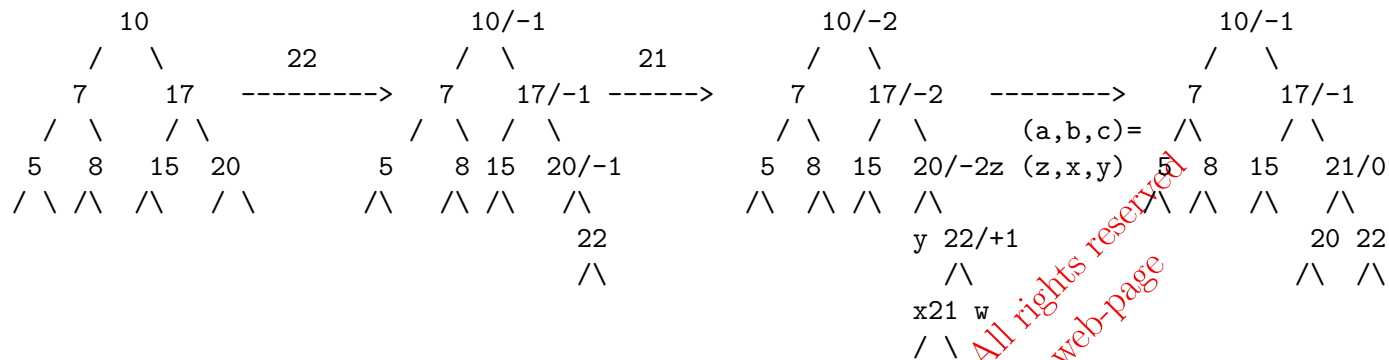
Starting with an empty AVL tree insert the following keys in the order shown (leftmost key first). Show the state of the tree (a) after the key is inserted and before a rebalancing takes place, and (b) after the key is inserted and a rebalancing operation caused by the insertion is completed. (c) Following the insertions, delete 8, 5, 17. Then delete 8, 5, 17.

Proof.

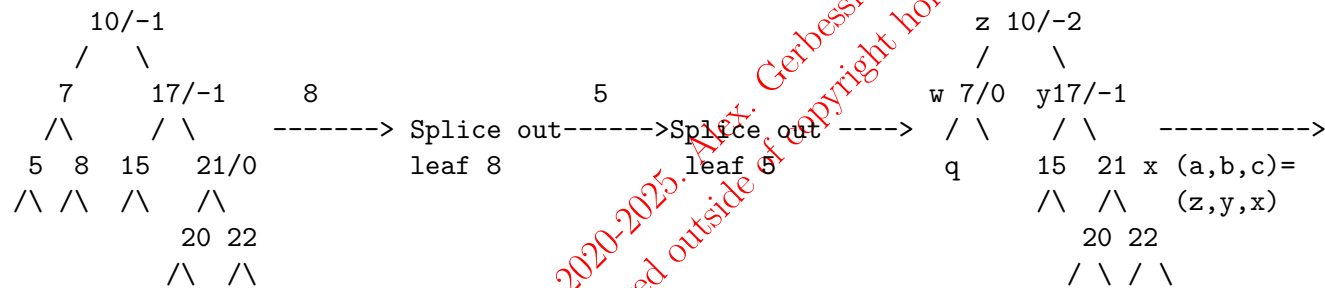
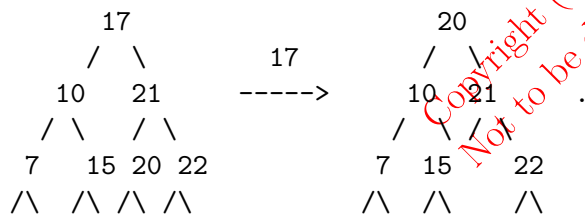
Insertion:



Problem Set Continued



Deletion:



■

Problem Set
Problem 4.

For one Insertion into an AVL tree with n keys prior to the insertion, what is an upper bound on the number of single rotations that need to be performed to make sure it is still an AVL tree after the Insertion?

Proof. Two. A single or a double rotation needs to be performed; the latter is two single rotations! ■

Copyright (c) 2020-2025. Alex. Gerbessiotis. All rights reserved
Not to be displayed outside of copyright holder's web-page

Problem Set
Problem 5.

For the tree of Problem 3 after deletion, label each vertex with a number from $1, \dots, 6$ that shows the order this vertex is visited in (i) a preorder, (ii) postorder, and (iii) inorder tree traversal.

Proof. Skipped. ■

Copyright (c) 2020-2025. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set
Problem 6.

Problem 7.

Before you solve this problem understand carefully the semantics of the operations. In particular in $\text{Delete}(S, z)$ z is a pointer to an element NOT a KEY VALUE. Let n be the size of the data structure before the operation is performed.

- (a) Insertion of a key x in a data structure S , i.e. operation $\text{Insert}(S, x)$.
- (b) Deletion of a record of S pointed by pointer z , i.e. operation $\text{Delete}(S, z)$,
- (c) Search of a key x , i.e. operation $\text{Search}(S, x)$,
- (d) FindMin of S i.e. operation $\text{FindMin}(S)$,
- (e) ExtractMax of S i.e. operation $\text{ExtractMax}(S)$,

Note: All keys in S are distinct. If S is sorted, it is sorted in increasing order. If a linked list (singly or doubly linked) is used, assume the existence of a head pointer; the existence or not of a tail pointer depends on the description of the data structure. **Express the running time in terms of the number of elements n in the data structure just before the application of the operation.**

Proof. (1) All $O(n)$.
(2) All $O(\lg n)$.

■

Problem Set
Problem 8.

Starting with an empty binary search tree insert 1, then 5, then 3, then 4, and finally 2. Show the tree after each insertion. (We don't ask you to build an AVL tree.) Then perform inorder, preorder and postorder traversals on the resulting binary search tree. List the nodes in the order visited for each traversal.

Proof. Practice makes perfect! You practice. ■

Copyright (c) 2020-2025. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set

Problem 9.

Argue that since sorting n keys using comparisons, takes $\Omega(n \lg n)$ time in the worst case in the comparison model, any comparison-based algorithm for constructing a binary search tree from an arbitrary sequence of n keys takes $\Omega(n \lg n)$ time in the worst case.

Proof. Suppose we can construct a binary search tree on n nodes in T time. Then we can run an inorder traversal of the BST in $\Theta(n)$ time and thus get in total time of $T + \Theta(n)$ the nodes in sorted form, since we know that an inorder traversal of a BST produces a sorted sequence of the keys stored in the tree.

If $T = o(n \lg n)$ this means we can sort n keys in $o(n \lg n) + \Theta(n) = o(n \lg n)$ time: first build a binary search tree and then perform an inorder traversal of its nodes. However we know from the lower bound for sorting that comparison-based sorting can not be performed in $o(n \lg n)$ time, i.e. it requires $\Omega(n \lg n)$. Therefore we cannot have $T = o(n \lg n)$, i.e. we must have $T = \Omega(n \lg n)$.

Thus any algorithm for creating a BST of n keys must have running time (at least) $T = \Omega(n \lg n)$. This bound is tight for AVL trees (and also red-black trees not covered in class but material related to them is available in the textbook) since the time an AVLTreeInsert requires to insert n keys in an empty tree is also tight $\Theta(n \lg n)$. ■

Problem Set
Problem 10.

Give a data structure S that implements a MIN priority queue (operations ExtractMin and Insert are implemented efficiently) and a MAX priority queue (operations ExtractMax, Insert) efficiently and simultaneously (i.e. in worst case $O(\lg n)$ time). You therefore need to implement efficiently three operations: Insert(x), ExtractMin, and ExtractMax.

A complete solution first describes the data structure S , and then presents the algorithms for the three operations. Finally an analysis of the worst case running time is performed and shown to be $O(\lg n)$, where n is the number of elements in the data structure before the operation is performed.

Note: You are allowed to use without proof all operations that have been introduced in this class. If for example you wish to use MaxHeapify, you do not need to present its code, you just call it using the class described interface MaxHeapify(A, i).

Proof. Use an AVL tree as the data structure. Insert is an AVL tree insert. ExtractMin, ExtractMax, are implemented by first issuing a FindMin or FindMax and then a Delete on the found min or max key. Insert, Delete on AVL trees are in the notes/textbook, and so are FindMin, FindMax. Worst case performance of all these operations on an AVL tree of n nodes is $O(\lg n)$. ■

Copyright (c) 2020-2025. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web page

Problem Set
Problem 11.

You want to sort the n keys stored into an AVL tree. How fast can you do it? Explain.

Proof. Linear time to the number of keys $\Theta(n)$. Run an inorder traversal! ■

Copyright (c) 2020-2025. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set
Problem 12.

(Precursor to the previous problem.) You have n keys and you want to use an AVL tree to sort them. Do so and explain so.

Proof. Insert the keys into an AVL tree and then run the algorithm of Problem 5 above. The latter step is $O(n)$. The former step is $O(\lg n)$ per key or $O(n \lg n)$ total. Total running time is $O(n \lg n)$ as well. ■

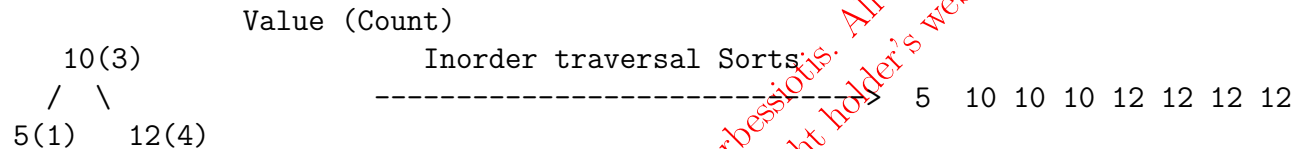
Copyright (c) 2020-2025. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set

Problem 13.

We have n keys that take k distinct values (integer or not). Give an algorithm that sorts those keys in $O(n \lg k)$ time. (Assume $n \gg k$.)

Proof. Use an AVL tree of k nodes. Each node has a linked list attached to it containing all the keys with that value. Radix-sort does not (necessarily) work. We don't know the range of the key values. Let alone the fact that the keys might not be integer. Build an AVL tree and for each node maintain also a counter (how many keys have that specific value). Maintaining this AVL is $O(\lg k)$ per operation as the number of distinct values is k . Thus run the algorithm in Problem 6, noting that only k keys are inserted into the AVL tree, even if we run n Search operations. In addition, when you do the inorder traversal and you are about to print the value of node x you print that value as many times as indicated by the counter. (See example below.)



■

Problem Set Continued

Let A be a set of distinct binary strings whose total length sum is n . Show how to use a radix tree to sort A in lexicographic order in $\Theta(n)$ time. Sorting the tree above with $n = 13$ would give the strings 0, 011, 10, 100, 1011 in lexicographic order.

Proof. First insert the strings and then perform a preorder traversal. In the preorder traversal a string is printed only for the red nodes (not for the black nodes that do not correspond to strings).

Note that the preorder is the correct traversal as a node is printed before its descendants. A printed node x is a red node and the string corresponding to that node should be printed before the strings corresponding to its descendants (in the left or right subtree of x). Strings corresponding to nodes of x 's left subtree must be printed before the ones corresponding to nodes of x 's right subtree as up to node x both strings are the same, but just after x the strings in the left subtree have a leading zero and the ones in the right subtree have a leading 1; the former must be printed before the latter ones. Preorder traversal guarantees that.

Running time of algorithm is $O(n)$, as insertion takes time proportional to length of individual string being inserted and sum of all strings is n . ■

Copyright (c) 2020-2025. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page.

Problem 15.

TRUE or FALSE? Anything other than a TRUE or FALSE is wrong! References are to a tree containing n keys and of height h . How h varies with n depends on the individual question.

1. Every binary search tree on n nodes has height $O(\lg n)$.
2. An AVL tree has height between $\lg n$ and $\lg n + 10$.
3. BSTInsert is $O(\lg n)$.
4. BSTInsert is $O(h)$.
5. BSTInsert is $O(n)$.
6. AVLInsert is $O(n)$.
7. AVLInsert is $O(\lg n)$.
8. AVLInsert is $\Omega(n)$.

Proof. 1. FALSE?

2. FALSE!
3. FALSE
4. TRUE
5. TRUE
6. TRUE
7. TRUE
8. FALSE.

Copyright (c) 2020-2025. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

■

Problem 16.

We have three AVL trees T_1, T_2, T_3 of a, b, c keys each. We would like to create one AVL tree T that stores all the keys of T_1, T_2, T_3 in it i.e. $T = T_1 \cup T_2 \cup T_3$ of size $a + b + c$. Give an algorithm that can do that whose worst-case running time is $O(a + b + c)$. Explain.

Proof. 3 in-order plus intelligent AVL creation after a double merging. ■

Copyright (c) 2020-2025. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page