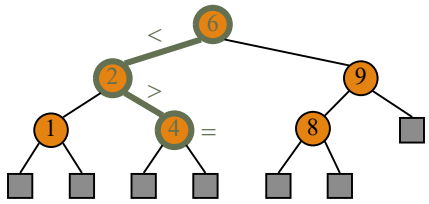
Binary Search Trees



1

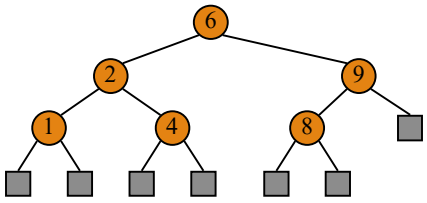
Binary Search Trees

A binary search tree is a proper binary tree storing keys (or key-value entries) at its internal nodes and satisfying the following property:

- Let u, v, and w be three nodes such that u is in the left subtree of v and w is in the right subtree of v. We have $key(u) \leq key(v) < key(w)$

External nodes do not store items

An inorder (left, node,right) traversal of a binary search trees visits the keys in increasing order



2

Search

To search for a key k, we trace a downward path starting at the root

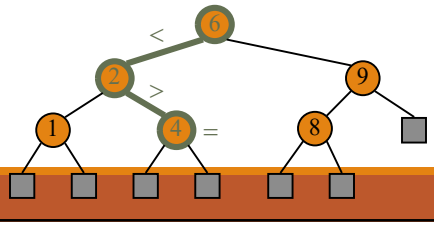
The next node visited depends on the comparison of k with the key of the current node

If we reach a leaf, the key is not found

Example: get(4):
- Call TreeSearch(4,root)

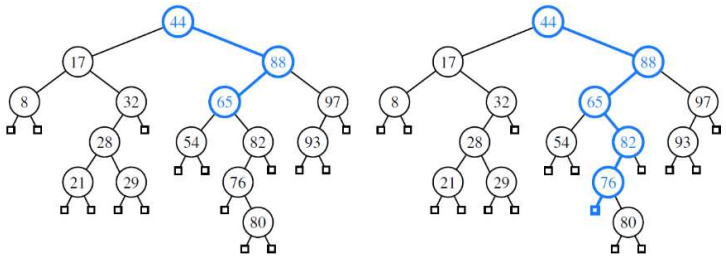
The algorithms for nearest neighbor queries are similar

```
Algorithm TreeSearch(k, v)
    if T.isExternal (v)
        return v
    if k < key(v)
        return TreeSearch(k, left(v))
    else if k = key(v)
        return v
    else { k > key(v) }
        return TreeSearch(k, right(v))
```



3

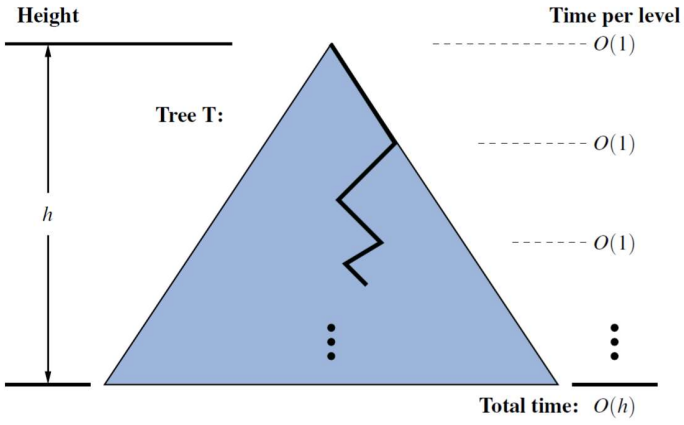
Ex. Search In Binary Search Tree



(a) A successful search for key 65 in a binary search tree; (b) an unsuccessful search for key 68 that terminates at the leaf to the left of the key 76.

4

Analysis of Binary Tree Searching



5

Insertion

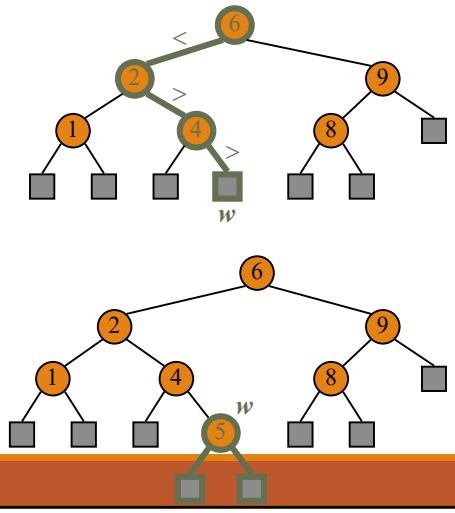
To perform operation put(k, o), we search for key k (using TreeSearch)

Assume k is not already in the tree, and let w be the leaf reached by the search

We insert k at node w and expand w into an internal node

Algorithm TreeInsert(k, v):
Input: A search key k to be associated with value v
p = TreeSearch(root( ), k)
if k == key(p) **then**
    Change p's value to (v)
else
    expandExternal(p, (k,v)

Example: insert 5



6

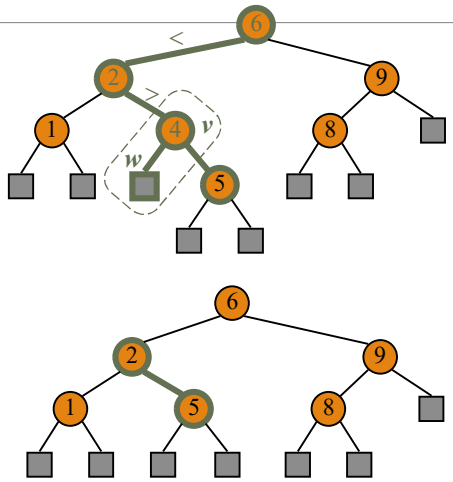
Deletion

To perform operation remove(***k***), we search for key ***k***

Assume key ***k*** is in the tree, and let let ***v*** be the node storing ***k***

If node ***v*** has a leaf child ***w***, we remove ***v*** and ***w*** from the tree with operation removeExternal(***w***), which removes ***w*** and its parent

Example: Remove 4



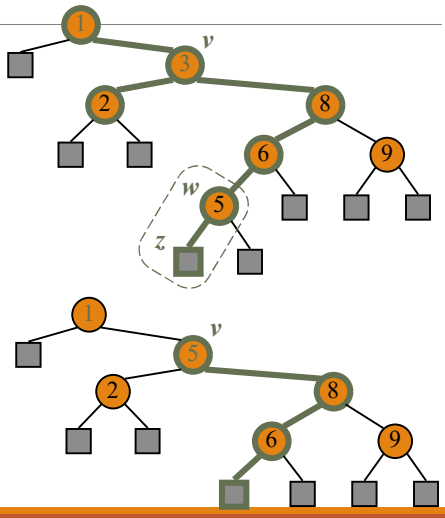
7

Deletion (cont.)

We consider the case where the key k to be removed is stored at a node v whose children are both internal

- we find the internal node w that follows v in an inorder traversal
- we copy $key(w)$ into node v
- we remove node w and its left child z (which must be a leaf) by means of operation removeExternal(z)

Example: Remove 3



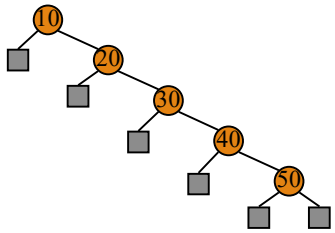
8

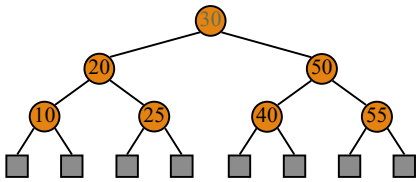
Performance of a Binary Search Tree

Consider an ordered map with n items implemented by means of a binary search tree of height h

- the space used is $O(n)$
- methods get, put and remove take $O(h)$ time

The height h is $O(n)$ in the worst case and $O(\log n)$ in the best case





9

Performance

Method	Running Time
size, isEmpty	O(1)
get, put, remove	O(h)
firstEntry, lastEntry	O(h)
ceilingEntry, floorEntry, lowerEntry, higherEntry	O(h)
subMap	O(s+h)
entrySet, keySet, values	O(n)

Table : Worst-case running times of the operations for a TreeMap. We denote the current height of the tree with h, and the number of entries reported by submap as s. The space usage is O(n), where n is the number of entries stored in the map.

10

Balanced Search Trees

Assume a random series of insertions and removals, the standard binary search tree supports O(logn) expected running times for the basic map operations. However, we may only claim O(n) worst-case time, because some sequences of operations may lead to an unbalanced tree with height proportional to n.

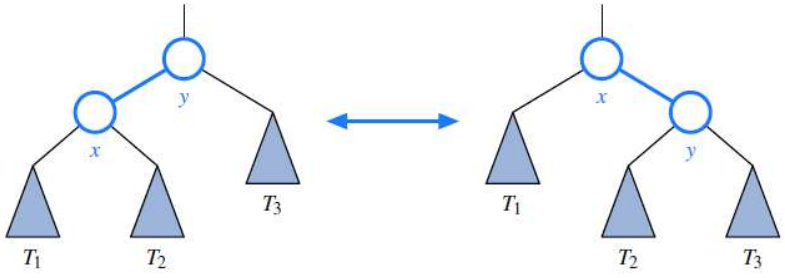
In the remainder of this chapter, we will explore four search-tree algorithms that provide stronger performance guarantees.

Three of the four data structures (**AVL trees**, **splay trees**, and **red-black trees**) are based on augmenting a standard binary search tree with occasional operations to reshape the tree and reduce its height. **The primary operation to rebalance a binary search tree is known as a rotation.**

11

Rotation Operation in a Binary Search Tree

During a rotation, we "rotate" a child to be above its parent, as diagrammed below



A rotation operation in a binary search tree. A rotation can be performed to transform the left formation into the right, or the right formation into the left. Note that all keys in subtree T1 have keys less than that of position x, all keys in subtree T2 have keys that are between those of positions x and y, and all keys in subtree T3 have keys that are greater than that of position y.

12

Rotation

A single rotation modifies a constant number of parent-child relationships, it can be implemented in O(1) time with a linked binary tree representation.

A rotation allows the shape of a tree to be modified while maintaining the search-tree property.

13

The Trinode Restructuring Operation in a Binary Search Tree

Algorithm restructure(x):

Input: A position x of a binary search tree T that has both a parent y and a grandparent z

Output: Tree T after a trinode restructuring (which corresponds to a single or double rotation) involving positions x, y, and z

1: Let (a, b, c) be a left-to-right (inorder) listing of the positions x, y, and z, and let (T1, T2, T3, T4) be a left-to-right (inorder) listing of the four subtrees of x, y, and z not rooted at x, y, or z.

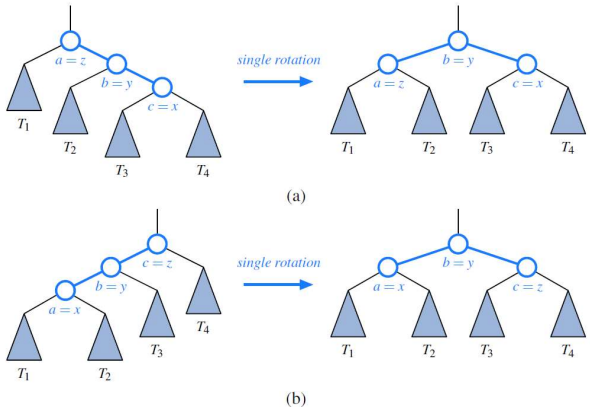
2: Replace the subtree rooted at z with a new subtree rooted at b.

3: Let a be the left child of b and let T1 and T2 be the left and right subtrees of a, respectively.

4: Let c be the right child of b and let T3 and T4 be the left and right subtrees of c, respectively.

14

Single Rotation



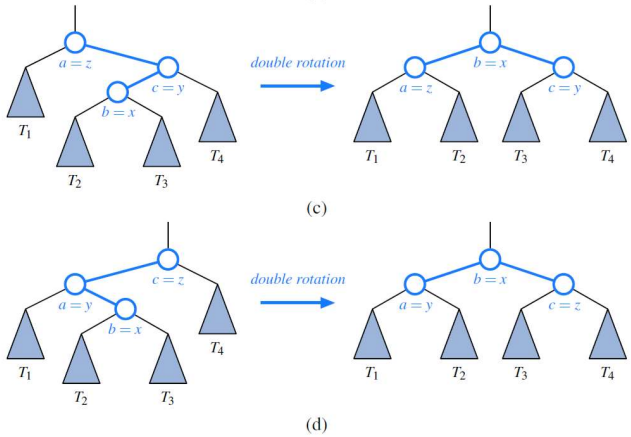
15

Left Rotation Pseudocode

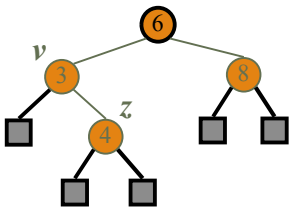
```
LEFT-ROTATE(T, x)
 1  y = x.right                  // set y
 2  x.right = y.left             // turn y's left subtree into x's right subtree
 3  if y.left ≠ T.nil
 4      y.left.p = x
 5  y.p = x.p                    // link x's parent to y
 6  if x.p == T.nil
 7      T.root = y
 8  elseif x == x.p.left
 9      x.p.left = y
10  else x.p.right = y
11  y.left = x                   // put x on y's left
12  x.p = y
```

16

Double Rotation



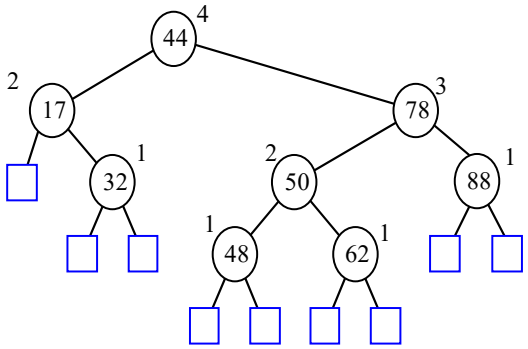
17



AVL Trees

18

AVL Tree Definition



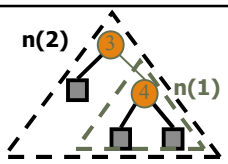
An example of an AVL tree where the heights are shown next to the nodes

AVL trees are balanced

An AVL Tree is a binary search tree such that for every internal node v of T, the **heights of the children of v can differ by at most 1**

19

Height of an AVL Tree



Fact: The height of an AVL tree storing n keys is O(log n).

Proof (by induction): Let us bound n(h): the minimum number of internal nodes of an AVL tree of height h.

We easily see that n(1) = 1 and n(2) = 2

For n > 2, an AVL tree of height h contains the root node, one AVL subtree of height n-1 and another of height n-2.

That is, n(h) = 1 + n(h-1) + n(h-2)

Knowing n(h-1) > n(h-2), we get n(h) > 2n(h-2). So
n(h) > 2n(h-2), n(h) > 4n(h-4), n(h) > 8n(n-6), … (by induction),
n(h) > 2^in(h-2i)

Solving the base case we get: n(h) > $2^{h/2-1}$

Taking logarithms: h < 2log n(h) +2

Thus the height of an AVL tree is O(log n)

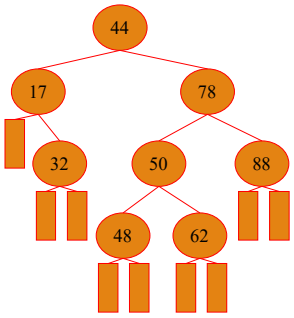
20

Insertion

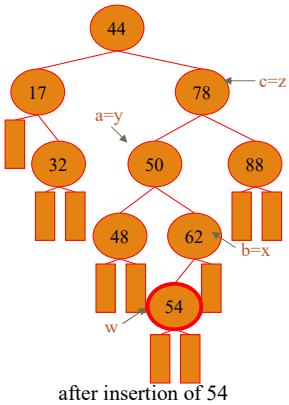
Insertion is as in a binary search tree

Always done by expanding an external node.

Example:



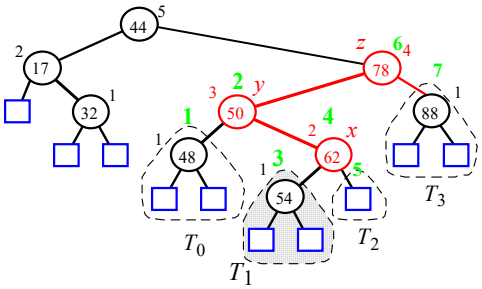
before insertion



after insertion of 54

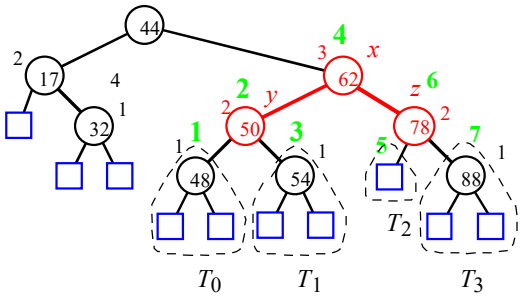
21

Insertion (54) Example, continued



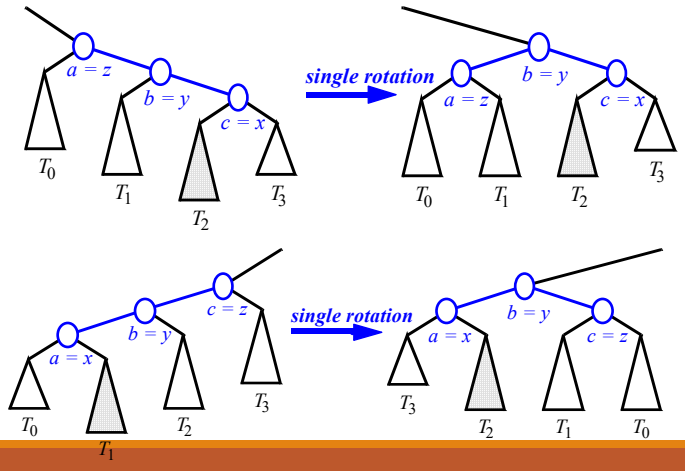
unbalanced...

...balanced



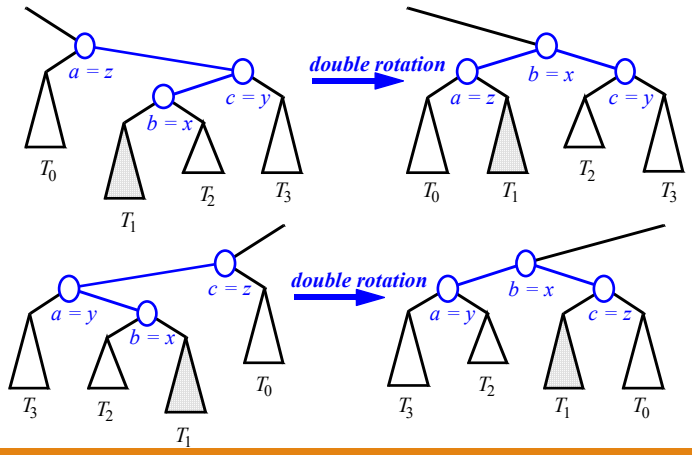
23

Restructuring (as Single Rotations)



24

Restructuring (as Double Rotations)

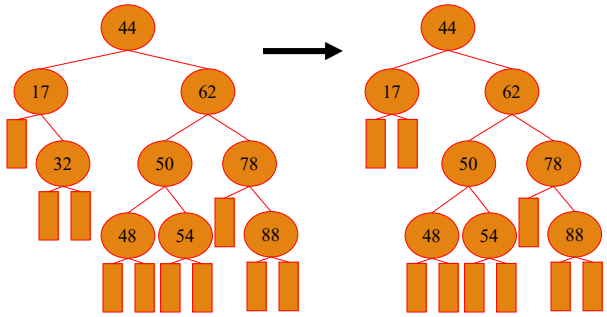


25

Removal

Removal begins as in a binary search tree, which means the node removed will become an empty external node. Its parent, w, may cause an imbalance.

Example:



before deletion of 32 after deletion

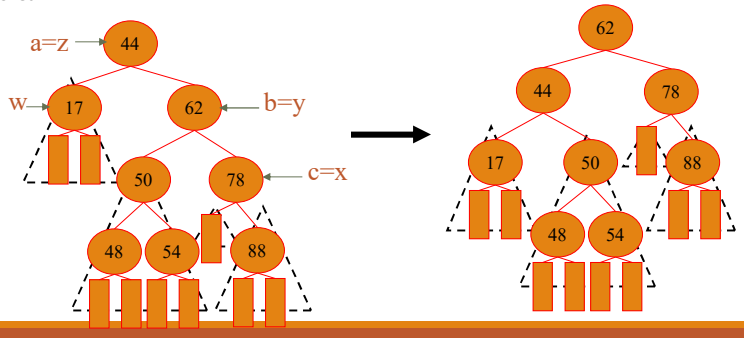
26

Rebalancing after a Removal

Let z be the first unbalanced node encountered while travelling up the tree from w. Also, let y be the child of z with the larger height, and let x be the child of y with the larger height

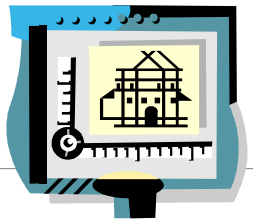
We perform a trinode restructuring to restore balance at z

As this restructuring may upset the balance of another node higher in the tree, we must continue checking for balance until the root of T is reached



27

AVL Tree Performance



AVL tree storing n items

- The data structure uses O(n) space
- A single restructuring takes O(1) time
  - using a linked-structure binary tree
- Searching takes O(log n) time
  - height of tree is O(log n), no restructures needed
- Insertion takes O(log n) time
  - initial find is O(log n)
  - restructuring up the tree, maintaining heights is O(log n)
- Removal takes O(log n) time
  - initial find is O(log n)
  - restructuring up the tree, maintaining heights is O(log n)

28

```
1  /** An implementation of a sorted map using an AVL tree. */
2  public class AVLTreeMap<K,V> extends TreeMap<K,V> {
3    /** Constructs an empty map using the natural ordering of keys. */
4    public AVLTreeMap( ) { super( ); }
5    /** Constructs an empty map using the given comparator to order keys. */
6    public AVLTreeMap(Comparator<K> comp) { super(comp); }
7    /** Returns the height of the given tree position. */
8    protected int height(Position<Entry<K,V>> p) {
9      return tree.getAux(p);
10   }
11   /** Recomputes the height of the given position based on its children's heights. */
12   protected void recomputeHeight(Position<Entry<K,V>> p) {
13     tree.setAux(p, 1 + Math.max(height(left(p)), height(right(p))));
14   }
15   /** Returns whether a position has balance factor between −1 and 1 inclusive. */
16   protected boolean isBalanced(Position<Entry<K,V>> p) {
17     return Math.abs(height(left(p)) − height(right(p))) <= 1;
18   }
```

Java Implementation

29

```
19   /** Returns a child of p with height no smaller than that of the other child. */
20   protected Position<Entry<K,V>> tallerChild(Position<Entry<K,V>> p) {
21     if (height(left(p)) > height(right(p))) return left(p);          // clear winner
22     if (height(left(p)) < height(right(p))) return right(p);         // clear winner
23     // equal height children; break tie while matching parent's orientation
24     if (isRoot(p)) return left(p);                                   // choice is irrelevant
25     if (p == left(parent(p))) return left(p);                        // return aligned child
26     else return right(p);
27   }
```

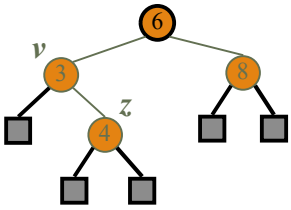
Java Implementation, 2

30

```
33   protected void rebalance(Position<Entry<K,V>> p) {
34     int oldHeight, newHeight;
35     do {
36       oldHeight = height(p);                          // not yet recalculated if internal
37       if (!isBalanced(p)) {                           // imbalance detected
38         // perform trinode restructuring, setting p to resulting root,
39         // and recompute new local heights after the restructuring
40         p = restructure(tallerChild(tallerChild(p)));
41         recomputeHeight(left(p));
42         recomputeHeight(right(p));
43       }
44       recomputeHeight(p);
45       newHeight = height(p);
46       p = parent(p);
47     } while (oldHeight != newHeight && p != null);
48   }
49   /** Overrides the TreeMap rebalancing hook that is called after an insertion. */
50   protected void rebalanceInsert(Position<Entry<K,V>> p) {
51     rebalance(p);
52   }
53   /** Overrides the TreeMap rebalancing hook that is called after a deletion. */
54   protected void rebalanceDelete(Position<Entry<K,V>> p) {
55     if (!isRoot(p))
56       rebalance(parent(p));
57   }
58 }
```

Java Implementation, 3

31



Splay Trees

32

Splay Trees

Splay tree is conceptually quite different from the other balanced search trees we will discuss in this chapter, for a splay tree does not strictly enforce a logarithmic upper bound on the height of the tree. In fact, no additional height, balance, or other auxiliary data need be stored with the nodes of this tree.

The efficiency of splay trees is due to a certain move-to-root operation, called splaying, that is performed at the bottommost position p reached during every insertion, deletion, or even a search.

Intuitively, a splay operation causes more frequently accessed elements to remain nearer to the root, thereby reducing the typical search times.

The surprising thing about splaying is that it allows us to guarantee a logarithmic amortized running time, for insertions, deletions, and searches.

33

Splaying

Given a node x of a binary search tree T, we splay x by moving x to the root of T through a sequence of restructurings.

The particular restructurings we perform are important, for it is not sufficient to move x to the root of T by just any sequence of restructurings. The specific operation we perform to move x up depends upon the relative positions of x, its parent y, and x's grandparent z (if it exists). There are three cases that we will consider.

34

Splaying Cases – Zig Zig

The node x and its parent y are both left children or both right children. We promote x, making y a child of x and z a child of y, while maintaining the inorder relationships of the nodes in T.

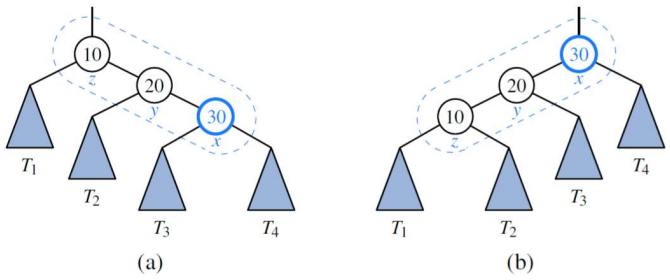


Figure 11.15: Zig-zig: (a) before; (b) after. There is another symmetric configuration where x and y are left children.

35

Splaying Cases – Zig Zag

One of x and y is a left child and the other is a right child. In this case, we promote x by making x have y and z as its children, while maintaining the inorder relationships of the nodes in T.

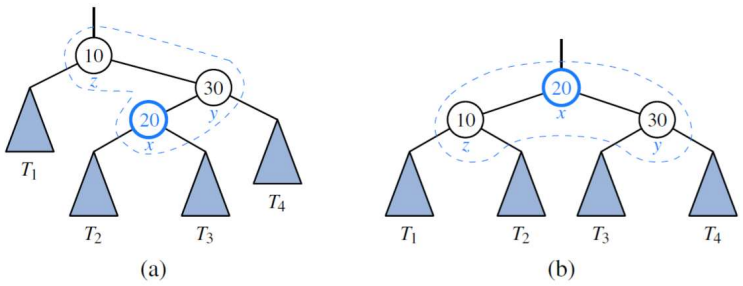


Figure 11.16: Zig-zag: (a) before; (b) after. There is another symmetric configuration where x is a right child and y is a left child.

36

Splaying Cases – Zig

x does not have a grandparent. In this case, we perform a single rotation to promote x over y, making y a child of x, while maintaining the relative inorder relationships of the nodes in T.

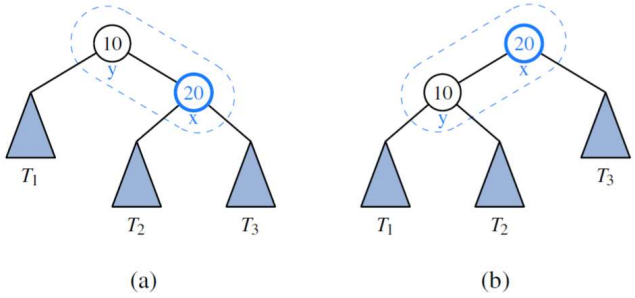
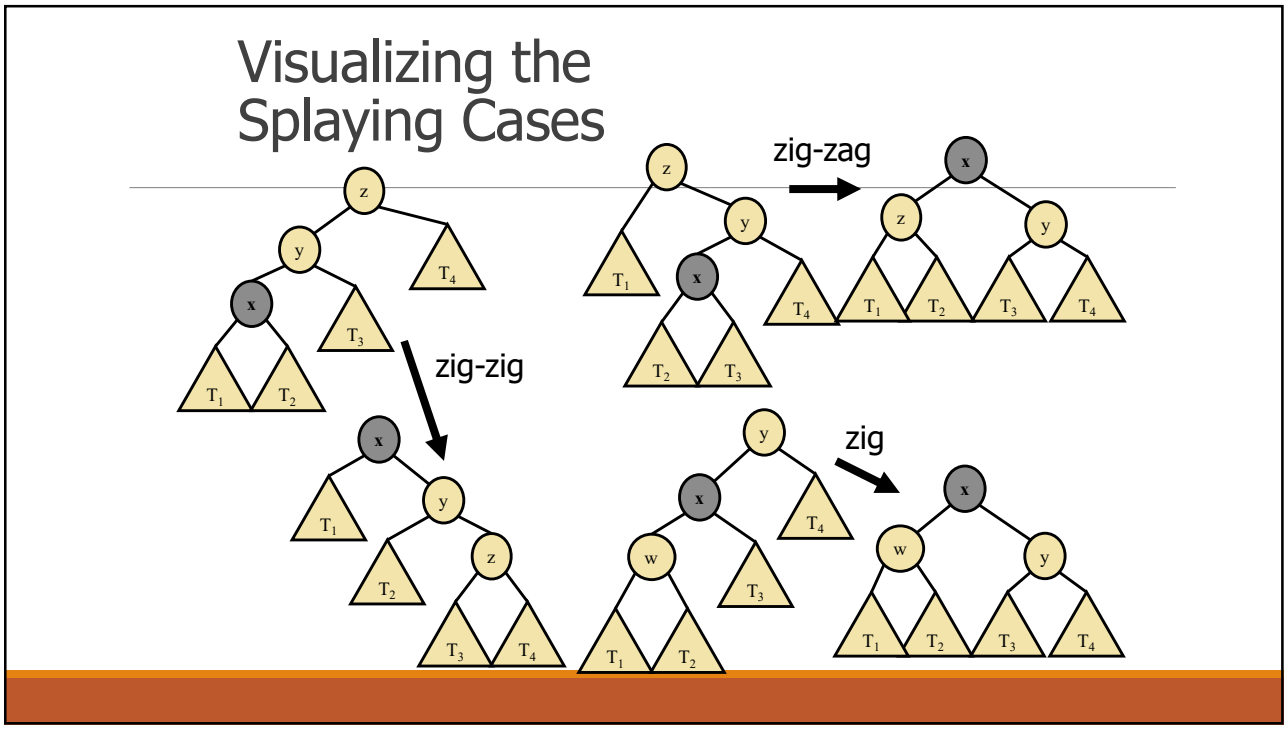


Figure 11.17: Zig: (a) before; (b) after. There is another symmetric configuration where x is originally a left child of y.

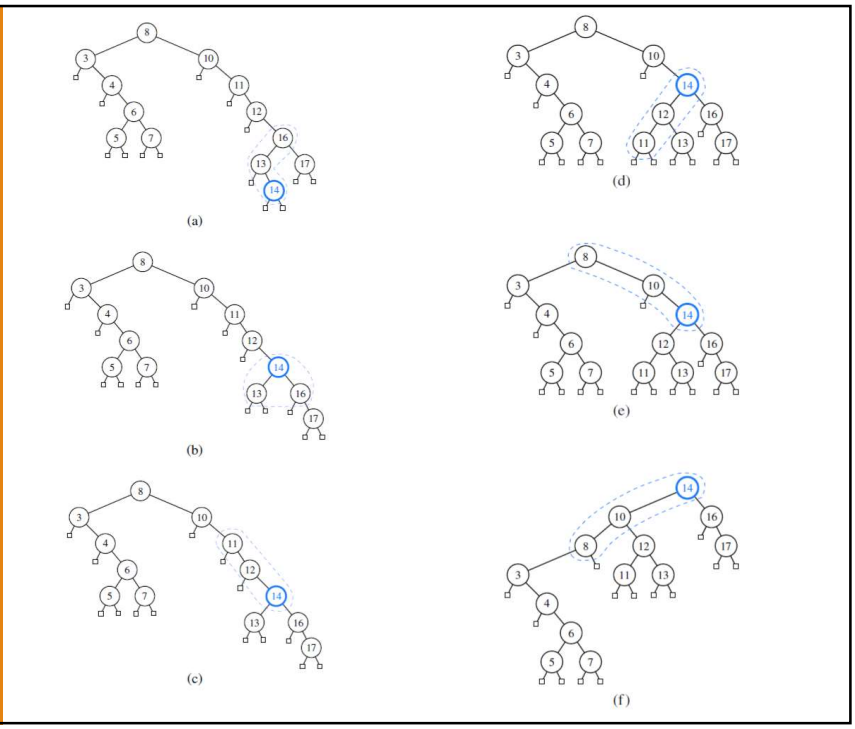
37



38

We perform a zig-zig or a zig-zag when x has a grandparent, and we perform a zig when x has a parent but not a grandparent. A splaying step consists of repeating these restructurings at x until x becomes the root of T.

Example of splaying a node: (a) splaying the node storing 14 starts with a zig-zag; (b) after the zig-zag; (c) the next step will be a zig-zig. (d) after the zig-zig; (e) the next step is again a zig-zig; (f) after the zig-zig.



39

When to Splay

The rules that dictate when splaying is performed are as follows:

- When searching for key k, if k is found at position p, we splay p, else we splay the parent of the leaf position at which the search terminates unsuccessfully. For example, the splaying in the previous slide would be performed after searching successfully for key 14 or unsuccessfully for key 15.
- When inserting key k, we splay the newly created internal node where k gets inserted. For example, the splaying in the previous figure would be performed if 14 were the newly inserted key.

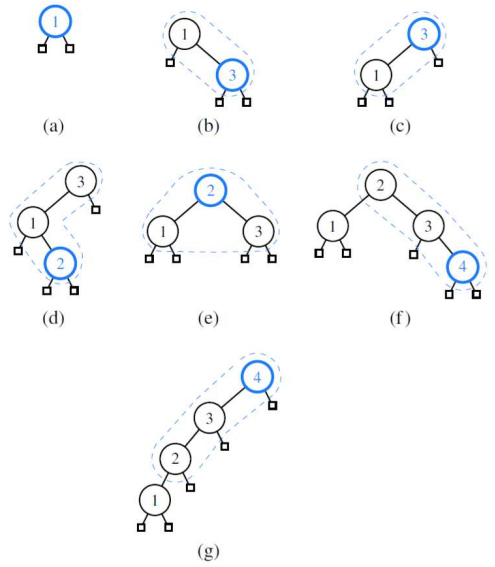
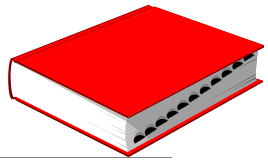


Figure 11.20: A sequence of insertions in a splay tree: (a) initial tree; (b) after inserting 3, but before a zig step; (c) after splaying; (d) after inserting 2, but before a zig-zag step; (e) after splaying; (f) after inserting 4, but before a zig-zig step; (g) after splaying.

40

Splay Trees

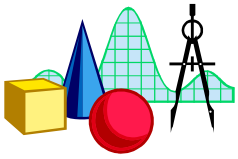


which nodes are splayed after each operation?

method	splay node
Search for k	if key found, use that node if key not found, use parent of ending external node
Insert (k,v)	use the new node containing the entry inserted
Remove item with key k	use the parent of the internal node that was actually removed from the tree (the parent of the node that the removed item was swapped with)

42

Amortized Analysis of Splay Trees



Running time of each operation is proportional to time for splaying.

Define rank(v) as the logarithm (base 2) of the number of nodes in subtree rooted at v.

Costs: zig = $1, zig-zig = $2, zig-zag = $2.

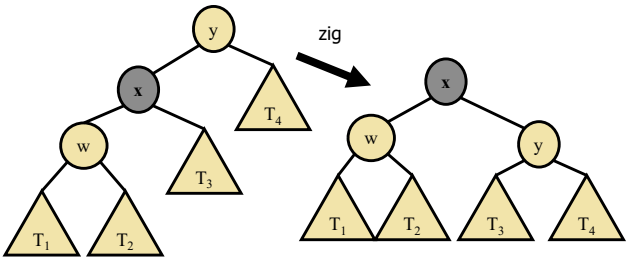
Thus, cost for playing a node at depth d = $d.

Imagine that we store rank(v) cyber-dollars at each node v of the splay tree (just for the sake of analysis).

43

Cost per zig

Rank of key k the number of keys that are less than k. In other words, it returns the index of k in a sorted array, or an ordered tree.

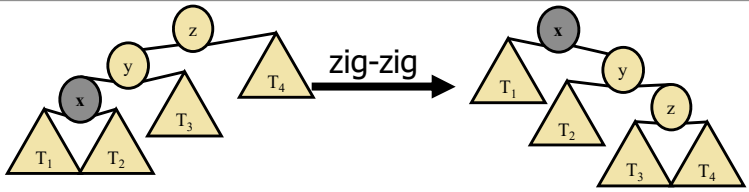


Doing a zig at x costs at most rank' (x) - rank(x):

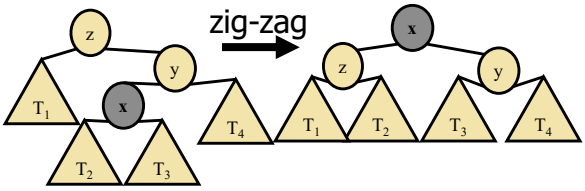
- cost = rank' (x) + rank' (y) - rank(y) - rank(x) ≤ rank' (x) - rank(x).

44

Cost per zig-zig and zig-zag



Doing a zig-zig or zig-zag at x costs at most
3(rank' (x) - rank(x)) - 2



45

Cost of Splaying

Cost of splaying a node x at depth d of a tree rooted at r:
- at most 3(rank(r) - rank(x)) - d + 2:
- Proof: Splaying x takes d/2 splaying substeps:

$$\begin{aligned}\text{cost} &\le \sum_{i=1}^{d/2} \text{cost}_i \\ &\le \sum_{i=1}^{d/2} (3(\text{rank}_i(x) - \text{rank}_{i-1}(x)) - 2) + 2 \\ &= 3(\text{rank}(r) - \text{rank}_0(x)) - 2(d/d) + 2 \\ &\le 3(\text{rank}(r) - \text{rank}(x)) - d + 2.\end{aligned}$$

46

Performance of Splay Trees

Recall: rank of a node is logarithm of its size.

Thus, amortized cost of any splay operation is O(log n)

In fact, the analysis goes through for any reasonable definition of rank(x)

This implies that splay trees can actually adapt to perform searches on frequently-requested items much faster than O(log n) in some cases

47

```
1  /** An implementation of a sorted map using a splay tree. */
2  public class SplayTreeMap<K,V> extends TreeMap<K,V> {
3    /** Constructs an empty map using the natural ordering of keys. */
4    public SplayTreeMap( ) { super(); }
5    /** Constructs an empty map using the given comparator to order keys. */
6    public SplayTreeMap(Comparator<K> comp) { super(comp); }
7    /** Utility used to rebalance after a map operation. */
8    private void splay(Position<Entry<K,V>> p) {
9      while (!isRoot(p)) {
10       Position<Entry<K,V>> parent = parent(p);
11       Position<Entry<K,V>> grand = parent(parent);
12       if (grand == null)                                    // zig case
13         rotate(p);
14       else if ((parent == left(grand)) == (p == left(parent))) {    // zig-zig case
15         rotate(parent);          // move PARENT upward
16         rotate(p);               // then move p upward
17       } else {                                              // zig-zag case
18         rotate(p);               // move p upward
19         rotate(p);               // move p upward again
20       }
21     }
22   }
```

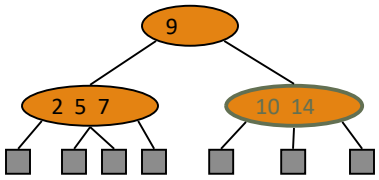
Java Implementation

48

```
23    // override the various TreeMap rebalancing hooks to perform the appropriate splay
24    protected void rebalanceAccess(Position<Entry<K,V>> p) {
25      if (isExternal(p)) p = parent(p);
26      if (p != null) splay(p);
27    }
28    protected void rebalanceInsert(Position<Entry<K,V>> p) {
29      splay(p);
30    }
31    protected void rebalanceDelete(Position<Entry<K,V>> p) {
32      if (!isRoot(p)) splay(parent(p));
33    }
34  }
```

Java Implementation

49



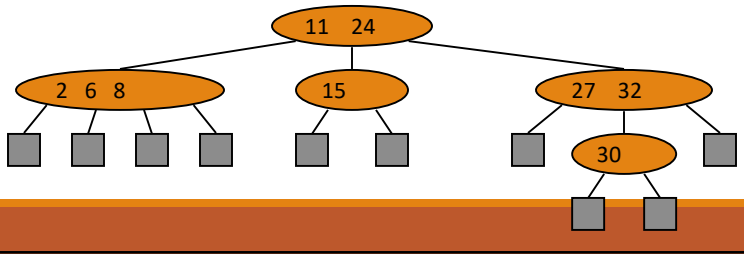
(2,4) Trees

50

Multi-Way Search Tree

A multi-way search tree is an ordered tree such that

- Each internal node has at least two children and stores $d-1$ key-element items (k_i, o_i), where d is the number of children
- For a node with children $v_1\, v_2 \ldots v_d$ storing keys $k_1\, k_2 \ldots k_{d-1}$
  - keys in the subtree of v_1 are less than k_1
  - keys in the subtree of v_i are between k_{i-1} and k_i $(i = 2, \ldots, d-1)$
  - keys in the subtree of v_d are greater than k_{d-1}
- The leaves store no items and serve as placeholders



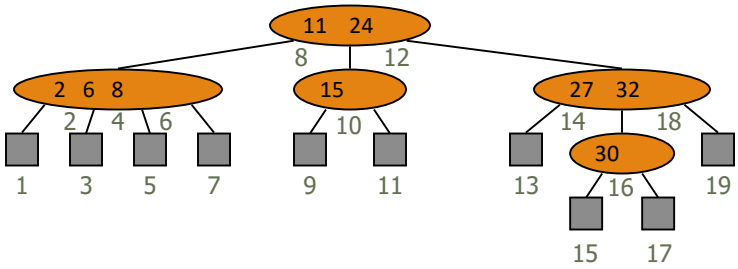
51

Multi-Way Inorder Traversal

We can extend the notion of inorder traversal from binary trees to multi-way search trees

Namely, we visit item (k_i, o_i) of node v between the recursive traversals of the subtrees of v rooted at children v_i and v_{i+1}

An inorder traversal of a multi-way search tree visits the keys in increasing order



52

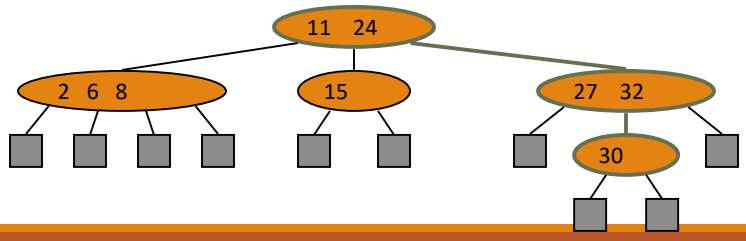
Multi-Way Searching

Similar to search in a binary search tree

A each internal node with children $v_1 v_2 \ldots v_d$ and keys $k_1 k_2 \ldots k_{d-1}$
- $k = k_i$ $(i = 1, \ldots, d-1)$: the search terminates successfully
- $k < k_1$: we continue the search in child v_1
- $k_{i-1} < k < k_i$ $(i = 2, \ldots, d-1)$: we continue the search in child v_i
- $k > k_{d-1}$: we continue the search in child v_d

Reaching an external node terminates the search unsuccessfully

Example: search for 30

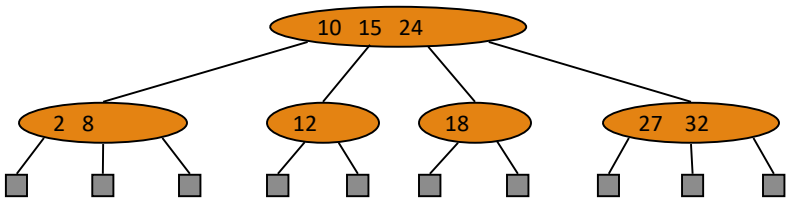


53

(2,4) Trees

A (2,4) tree (also called 2-4 tree or 2-3-4 tree) is a multi-way search with the following properties
- Node-Size Property: every internal node has at most four children
- Depth Property: all the external nodes have the same depth

Depending on the number of children, an internal node of a (2,4) tree is called a 2-node, 3-node or 4-node



54

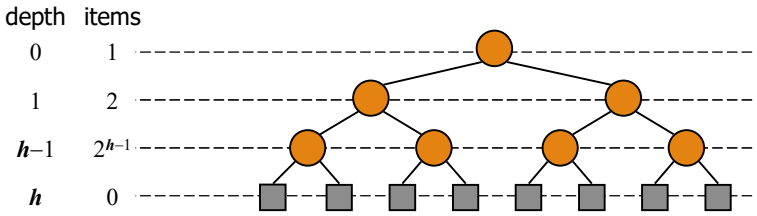
Height of a (2,4) Tree

Theorem: A (2,4) tree storing n items has height $O(\log n)$

Proof:

- Let h be the height of a (2,4) tree with n items
- Since there are at least 2^i items at depth $i = 0, \ldots, h-1$ and no items at depth h, we have
  $n \geq 1 + 2 + 4 + \ldots + 2^{h-1} = 2^h - 1$
- Thus, $h \leq \log(n+1)$

Searching in a (2,4) tree with n items takes $O(\log n)$ time



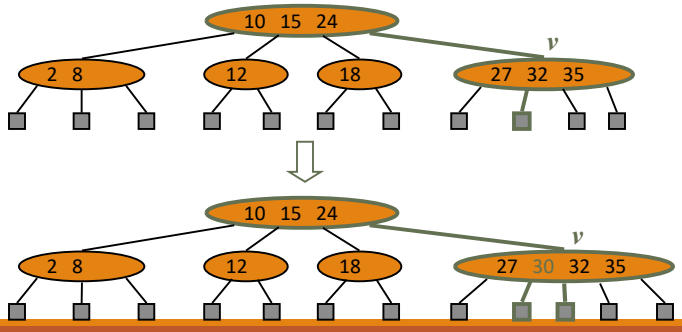
55

Insertion

We insert a new item (k, o) at the parent v of the leaf reached by searching for k

- We preserve the depth property but
- We may cause an overflow (i.e., node v may become a 5-node)

Example: inserting key 30 causes an overflow



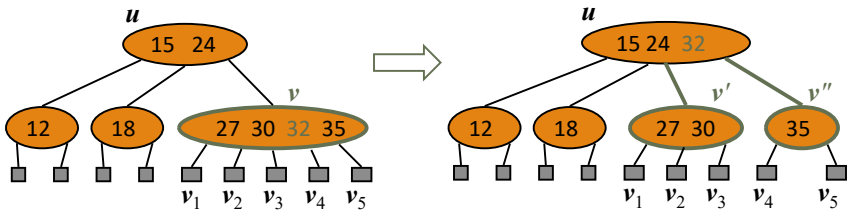
56

Overflow and Split

We handle an overflow at a 5-node v with a split operation:

- let $v_1 \ldots v_5$ be the children of v and $k_1 \ldots k_4$ be the keys of v
- node v is replaced nodes v' and v''
  - v' is a 3-node with keys k_1 k_2 and children v_1 v_2 v_3
  - v'' is a 2-node with key k_4 and children v_4 v_5
- key k_3 is inserted into the parent u of v (a new root may be created)

The overflow may propagate to the parent node u



57

Analysis of Insertion

Algorithm *put*(k, o)

1. We search for key k to locate the insertion node v
2. We add the new entry (k, o) at node v
3. **while** *overflow*(v)
   - **if** *isRoot*(v)
     - create a new empty root above v
   - $v \leftarrow$ *split*(v)

Let T be a (2,4) tree with n items

- Tree T has $O(\log n)$ height
- Step 1 takes $O(\log n)$ time because we visit $O(\log n)$ nodes
- Step 2 takes $O(1)$ time
- Step 3 takes $O(\log n)$ time because each split takes $O(1)$ time and we perform $O(\log n)$ splits

Thus, an insertion in a (2,4) tree takes $O(\log n)$ time

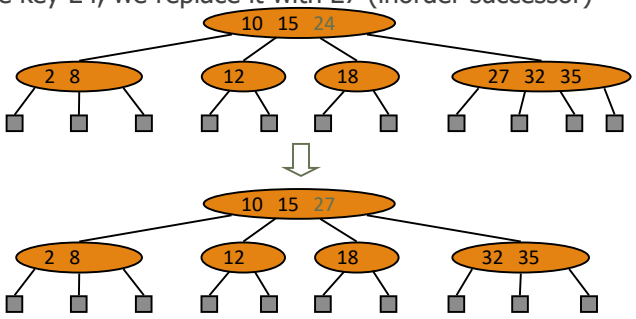
58

Deletion

We reduce deletion of an entry to the case where the item is at the node with leaf children

Otherwise, we replace the entry with its inorder successor (or, equivalently, with its inorder predecessor) and delete the latter entry

Example: to delete key 24, we replace it with 27 (inorder successor)



59

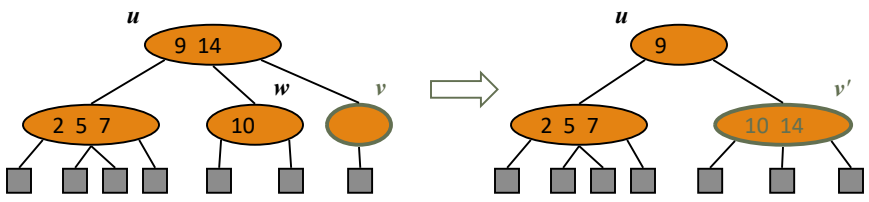
Underflow and Fusion

Deleting an entry from a node v may cause an underflow, where node v becomes a 1-node with one child and no keys

To handle an underflow at node v with parent u, we consider two cases

Case 1: the adjacent siblings of v are 2-nodes

- Fusion operation: we merge v with an adjacent sibling w and move an entry from u to the merged node v'
- After a fusion, the underflow may propagate to the parent u



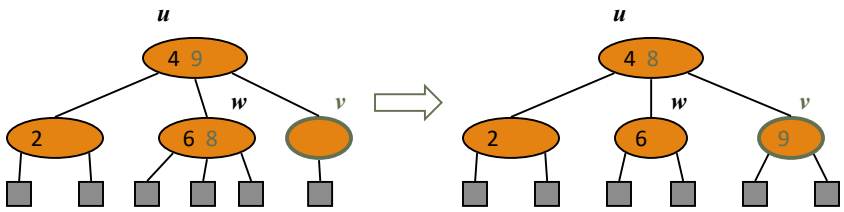
60

Underflow and Transfer

To handle an underflow at node v with parent u, we consider two cases

Case 2: an adjacent sibling w of v is a 3-node or a 4-node

- Transfer operation:
  1. we move a child of w to v
  2. we move an item from u to v
  3. we move an item from w to u
- After a transfer, no underflow occurs



61

Analysis of Deletion

Let T be a (2,4) tree with n items

- Tree T has $O(\log n)$ height

In a deletion operation

- We visit $O(\log n)$ nodes to locate the node from which to delete the entry
- We handle an underflow with a series of $O(\log n)$ fusions, followed by at most one transfer
- Each fusion and transfer takes $O(1)$ time

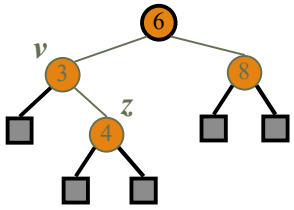
Thus, deleting an item from a (2,4) tree takes $O(\log n)$ time

62

Comparison of Map Implementations

	Search	Insert	Delete	Notes
Hash Table	1 expected	1 expected	1 expected	○ no ordered map methods ○ simple to implement
Skip List	$\log n$ high prob.	$\log n$ high prob.	$\log n$ high prob.	○ randomized insertion ○ simple to implement
AVL and (2,4) Tree	$\log n$ worst-case	$\log n$ worst-case	$\log n$ worst-case	○ complex to implement

63



Red-Black Trees

64

Red-Black Trees

Although AVL trees and (2,4) trees have a number of nice properties, they also have some disadvantages. For instance, AVL trees may require many restructure operations (rotations) to be performed after a deletion, and (2,4) trees may require many **split** or **fusing** operations to be performed after an insertion or removal.

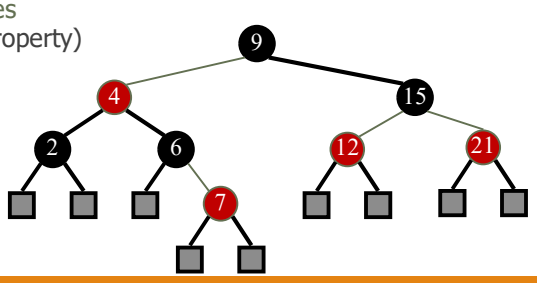
The red-black tree, does not have these drawbacks; it uses $O(1)$ structural changes after an update in order to stay balanced.

65

Red-Black Trees

A red-black tree can also be defined as a binary search tree that satisfies the following properties:

- Every node is red or black
- **Root** is black (if it is not we make it black)
- New insertions are always red
- Every path from root to leaf has the same # of **BLACK** nodes (all the leaves (nil) have the same black **depth**)
- No path can have two consecutive RED nodes
- Every leaf is black (nil is black) (**external** property)



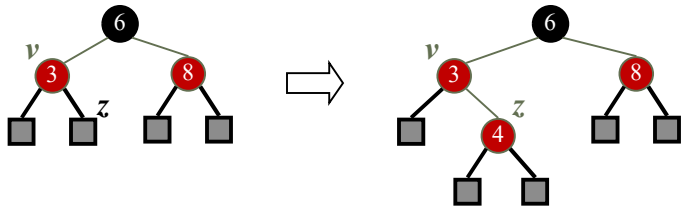
66

Insertion

To insert (k, o), we execute the insertion algorithm for binary search trees and color red the newly inserted node z unless it is the root

- We preserve the **root**, **external**, and **depth** properties
- If the **parent v of z is black**, we also preserve the internal property and we are done
- Else (v **is red**) we have a double red (i.e., a violation of the internal property), which requires a reorganization of the tree

Example where the insertion of 4 causes a double red:



67

Remedying a Double Red

Consider a double red with child z and parent v, and let w be the sibling of v

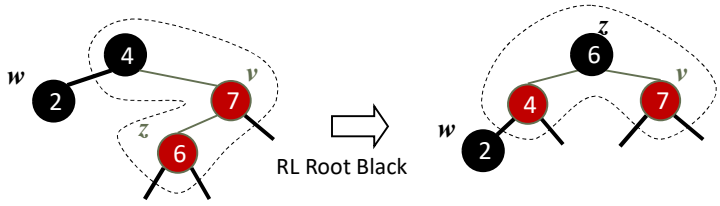
Case 1: w is black (Aunt is Black) Case 2: w is red (Aunt is Red)

68

Restructuring

A restructuring remedies a child-parent double red when the parent red node has a black sibling. (Aunt is Black)

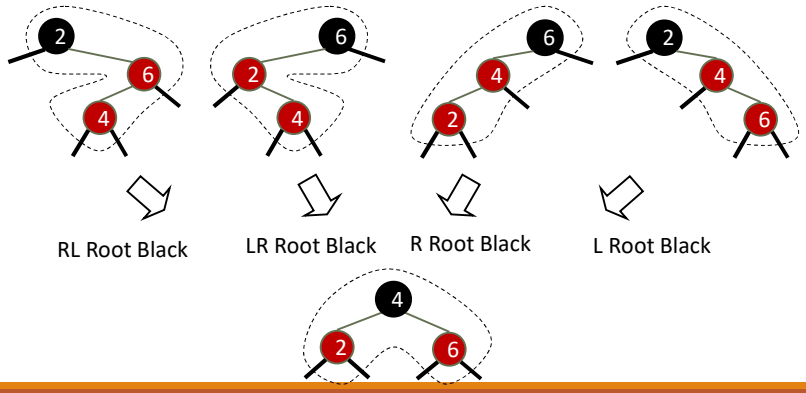
The internal property is restored and the other properties are preserved



69

Restructuring (cont.)

There are four restructuring configurations depending on whether the double red nodes are left or right children

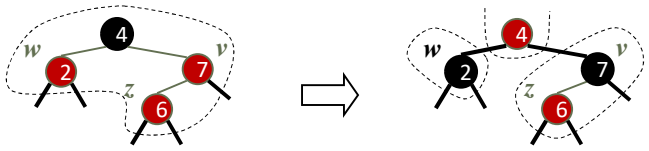


70

Recoloring (Aunt is Red)

A recoloring remedies a child-parent double red when the parent red node has a red sibling

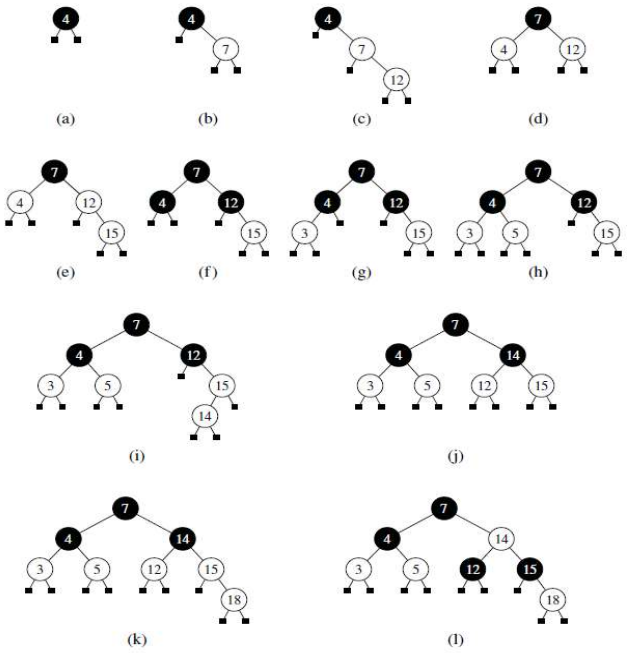
The parent v and its sibling w become black and the grandparent u becomes red, unless it is the root



71

Example

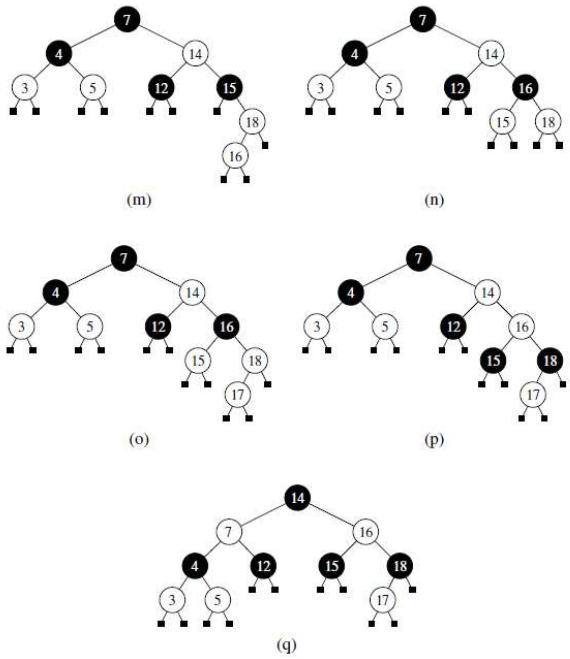
A sequence of insertions in a red-black tree: (a) initial tree; (b) insertion of 7; (c) insertion of 12, which causes a double red; (d) after restructuring; (e) insertion of 15, which causes a double red; (f) after recoloring (the root remains black); (g) insertion of 3; (h) insertion of 5; (i) insertion of 14, which causes a double red; (j) after restructuring; (k) insertion of 18, which causes a double red; (l) after recoloring.



72

Example

A sequence of insertions in a red-black tree (continued from previous figure): (m) insertion of 16, which causes a double red; (n) after restructuring; (o) insertion of 17, which causes a double red; (p) after recoloring there is again a double red, to be handled by a restructuring; (q) after restructuring.



73

Height of a Red-Black Tree

Theorem: A red-black tree storing n items has height $O(\log n)$

Proof:

- The height of a red-black tree is at most twice the height of its associated (2,4) tree, which is $O(\log n)$

The search algorithm for a binary search tree is the same as that for a binary search tree

By the above theorem, searching in a red-black tree takes $O(\log n)$ time

74

Analysis of Insertion

Algorithm *insert*(k, o)

1. We search for key k to locate the insertion node z
2. We add the new entry (k, o) at node z and color z red
3. **while** *doubleRed*(z)
   if *isBlack*(*sibling*(*parent*(z)))
   $z \leftarrow$ *restructure*(z)
   return
   else { *sibling*(*parent*(z) is red }
   $z \leftarrow$ *recolor*(z)

Recall that a red-black tree has $O(\log n)$ height

Step 1 takes $O(\log n)$ time because we visit $O(\log n)$ nodes

Step 2 takes $O(1)$ time

Step 3 takes $O(\log n)$ time because we perform

- $O(\log n)$ recolorings, each taking $O(1)$ time, and
- at most one restructuring taking $O(1)$ time

Thus, an insertion in a red-black tree takes $O(\log n)$ time

75

Deletion

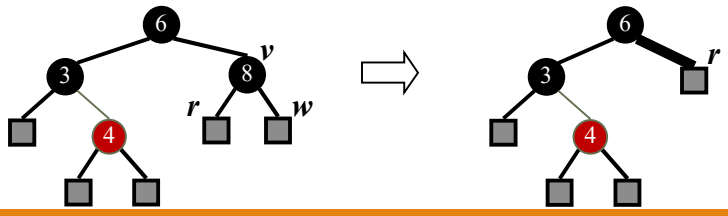
Reminder: Every path from root to leaf has the same # of **BLACK** nodes (all the leaves (nil) have the same black **depth**)

To perform operation remove(k), **we first execute the deletion algorithm for binary search trees**

Let v be the internal node removed, w the external node removed, and r the sibling of w

- If either v **or** r was RED, we color r **BLACK** and we are done
- Else (v **and** r were both black) we color r ***double black***, which is a violation of the internal property requiring a reorganization of the tree

Example where the deletion of 8 causes a double black:



76

Remedying a Double Black

The algorithm for remedying a double black node w with sibling y considers three cases

Case 1: y **is black and has a red child**
- We perform a restructuring, equivalent to a transfer , and we are done

Case 2: y **is black and its children are both black**
- We perform a recoloring, equivalent to a fusion, which may propagate up the double black violation

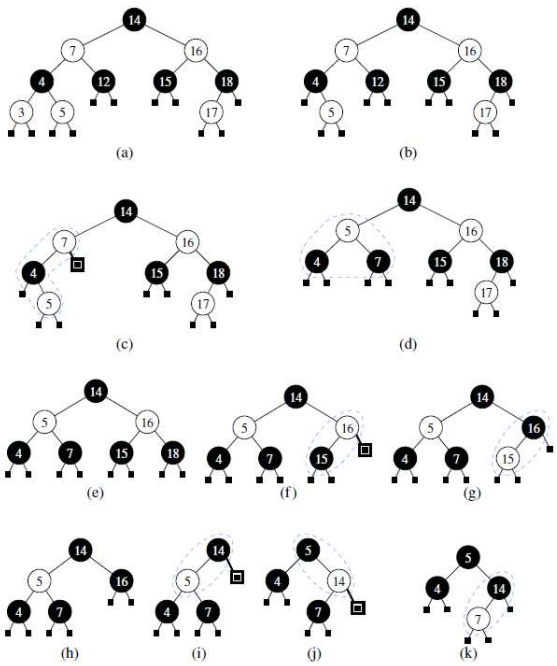
Case 3: y **is red**
- We perform an adjustment, equivalent to choosing a different representation of a 3-node, after which either Case 1 or Case 2 applies

Deletion in a red-black tree takes $O(\log n)$ time

77

Example

A sequence of deletions from a red-black tree: (a) initial tree; (b) removal of 3; (c) removal of 12, causing a black deficit to the right of 7 (handled by restructuring); (d) after restructuring; (e) removal of 17; (f) removal of 18, causing a black deficit to the right of 16 (handled by recoloring); (g) after recoloring; (h) removal of 15; (i) removal of 16, causing a black deficit to the right of 14 (handled initially by a rotation); (j) after the rotation the black deficit needs to be handled by a recoloring; (k) after the recoloring.



78

Red-Black Tree Reorganization

Insertion	remedy double red	
Red-black tree action	**(2,4) tree action**	**result**
restructuring	change of 4-node representation	double red removed
recoloring	split	double red removed or propagated up

Deletion	remedy double black	
Red-black tree action	**(2,4) tree action**	**result**
restructuring	transfer	double black removed
recoloring	fusion	double black removed or propagated up
adjustment	change of 3-node representation	restructuring or recoloring follows

79

```
1  /** An implementation of a sorted map using a red-black tree. */
2  public class RBTreeMap<K,V> extends TreeMap<K,V> {
3    /** Constructs an empty map using the natural ordering of keys. */
4    public RBTreeMap( ) { super( ); }
5    /** Constructs an empty map using the given comparator to order keys. */
6    public RBTreeMap(Comparator<K> comp) { super(comp); }
7    // we use the inherited aux field with convention that 0=black and 1=red
8    // (note that new leaves will be black by default, as aux=0)
9    private boolean isBlack(Position<Entry<K,V>> p) { return tree.getAux(p)==0;}
10   private boolean isRed(Position<Entry<K,V>> p) { return tree.getAux(p)==1; }
11   private void makeBlack(Position<Entry<K,V>> p) { tree.setAux(p, 0); }
12   private void makeRed(Position<Entry<K,V>> p) { tree.setAux(p, 1); }
13   private void setColor(Position<Entry<K,V>> p, boolean toRed) {
14     tree.setAux(p, toRed ? 1 : 0);
15   }
16   /** Overrides the TreeMap rebalancing hook that is called after an insertion. */
17   protected void rebalanceInsert(Position<Entry<K,V>> p) {
18     if (!isRoot(p)) {
19       makeRed(p);                    // the new internal node is initially colored red
20       resolveRed(p);                 // but this may cause a double-red problem
21     }
22   }
```

Java Implementation

80

```
23  /** Remedies potential double-red violation above red position p. */
24  private void resolveRed(Position<Entry<K,V>> p) {
25    Position<Entry<K,V>> parent,uncle,middle,grand; // used in case analysis
26    parent = parent(p);
27    if (isRed(parent)) {                              // double-red problem exists
28      uncle = sibling(parent);
29      if (isBlack(uncle)) {                           // Case 1: misshapen 4-node
30        middle = restructure(p);                      // do trinode restructuring
31        makeBlack(middle);
32        makeRed(left(middle));
33        makeRed(right(middle));
34      } else {                                        // Case 2: overfull 5-node
35        makeBlack(parent);                            // perform recoloring
36        makeBlack(uncle);
37        grand = parent(parent);
38        if (!isRoot(grand)) {
39          makeRed(grand);                             // grandparent becomes red
40          resolveRed(grand);                          // recur at red grandparent
41        }
42      }
43    }
44  }
```

Java Implementation 2

81

```
45  /** Overrides the TreeMap rebalancing hook that is called after a deletion. */
46  protected void rebalanceDelete(Position<Entry<K,V>> p) {
47    if (isRed(p))                                     // deleted parent was black
48      makeBlack(p);                                   // so this restores black depth
49    else if (!isRoot(p)) {
50      Position<Entry<K,V>> sib = sibling(p);
51      if (isInternal(sib) && (isBlack(sib) || isInternal(left(sib))))
52        remedyDoubleBlack(p);              // sib's subtree has nonzero black height
53    }
54  }
55
56  /** Remedies a presumed double-black violation at the given (nonroot) position. */
57  private void remedyDoubleBlack(Position<Entry<K,V>> p) {
58    Position<Entry<K,V>> z = parent(p);
59    Position<Entry<K,V>> y = sibling(p);
60    if (isBlack(y)) {
61      if (isRed(left(y)) || isRed(right(y))) {        // Case 1: trinode restructuring
62        Position<Entry<K,V>> x = (isRed(left(y)) ? left(y) : right(y));
63        Position<Entry<K,V>> middle = restructure(x);
64        setColor(middle, isRed(z)); // root of restructured subtree gets z's old color
65        makeBlack(left(middle));
66        makeBlack(right(middle));
67      } else {                                        // Case 2: recoloring
68        makeRed(y);
69        if (isRed(z))
70          makeBlack(z);                               // problem is resolved
71        else if (!isRoot(z))
72          remedyDoubleBlack(z);                       // propagate the problem
73      }
```

Java Implementation 3

82

```
48        makeBlack(p);                                  // so this restores black depth
49      else if (!isRoot(p)) {
50        Position<Entry<K,V>> sib = sibling(p);
51        if (isInternal(sib) && (isBlack(sib) || isInternal(left(sib))))
52          remedyDoubleBlack(p);                        // sib's subtree has nonzero black height
53      }
54    }
55
56    /** Remedies a presumed double-black violation at the given (nonroot) position. */
57    private void remedyDoubleBlack(Position<Entry<K,V>> p) {
58      Position<Entry<K,V>> z = parent(p);
59      Position<Entry<K,V>> y = sibling(p);
60      if (isBlack(y)) {
61        if (isRed(left(y)) || isRed(right(y))) {       // Case 1: trinode restructuring
62          Position<Entry<K,V>> x = (isRed(left(y)) ? left(y) : right(y));
63          Position<Entry<K,V>> middle = restructure(x);
64          setColor(middle, isRed(z)); // root of restructured subtree gets z's old color
65          makeBlack(left(middle));
66          makeBlack(right(middle));
67        } else {                                       // Case 2: recoloring
68          makeRed(y);
69          if (isRed(z))
70            makeBlack(z);                              // problem is resolved
71          else if (!isRoot(z))
72            remedyDoubleBlack(z);                      // propagate the problem
73        }
74      } else {                                         // Case 3: reorient 3-node
75        rotate(y);
76        makeBlack(y);
77        makeRed(z);
```

Java Implementation 4

83

Practice Questions

R-11.15 Perform the following sequence of operations in an initially empty splay tree and draw the tree after each set of operations.

a. Insert keys 0, 2, 4, 6, 8, 10, 12, 14, 16, 18, in this order.
b. Search for keys 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, in this order.
c. Delete keys 0, 2, 4, 6, 8, 10, 12, 14, 16, 18, in this order.

R-11.24 Consider a tree T storing 100,000 entries. What is the worst-case height of T in the following cases?

a. T is a binary search tree.
b. T is an AVL tree.
c. T is a splay tree.
d. T is a (2,4) tree.
e. T is a red-black tree.

84