Dictionaries

Data structures (recap)

> Data structure = data + operations on it

Data:

- An ID (key) + associated data (often implied, also in these slides).

Operations:

- The properties of the data structure consist of the provided operations (API for accessing data) and their runtimes (different implementations of the same API can result in different runtimes).

Data structures (recap)

We have already seen the Priority Queue data structure, which supports the following operations:

- **Extract-Min():** Removes an element with the smallest key from the priority queue and returns it.
- **Insert**(key): Adds a new element to the priority queue.
- **Build** (list of elements): Builds a priority queue containing the given elements.
- **Decrease-Key**(key, reference to element in queue): Sets the key for the element to min{key, old key}.

Dictionaries

Today: Dictionary. A data structure that supports the following operations:

- **Search**(key): Returns the element with the key key (or indicates if it does not exist).
- **Insert**(key): Inserts a new element with the key.
- **Delete**(key): Removes the element with the key.
- **Predecessor**(key): Finds the element with the highest key < key.
- **Successor**(key): Finds the element with the lowest key > key.
- **OrderedTraversal**(): Prints elements in sorted order.

For the last three operations, the keys must have an ordering.

If only the first three operations are supported, it is called an unordered dictionary. If all six operations are supported, it is called an ordered dictionary.

Dictionaries

Dictionaries in Java: interface Map.

Dictionaries in Python: dict.

Implementations:

- Balanced binary search trees: Support all the above operations (and many more, e.g., by adding extra information to the nodes) in O(log n) time. Search, Insert, Delete, Predecessor, Successor, Ordered Traversal and so on ...

- Hashing: Supports the first three operations in expected O(1) time.
  search, insert, delete

These implementations exist in Java as TreeMap and HashMap, respectively. In Python, the built-in dict datatype is implemented using hashing. There are no balanced binary search trees in Python's standard modules, but such implementations can be found from third-party sources.

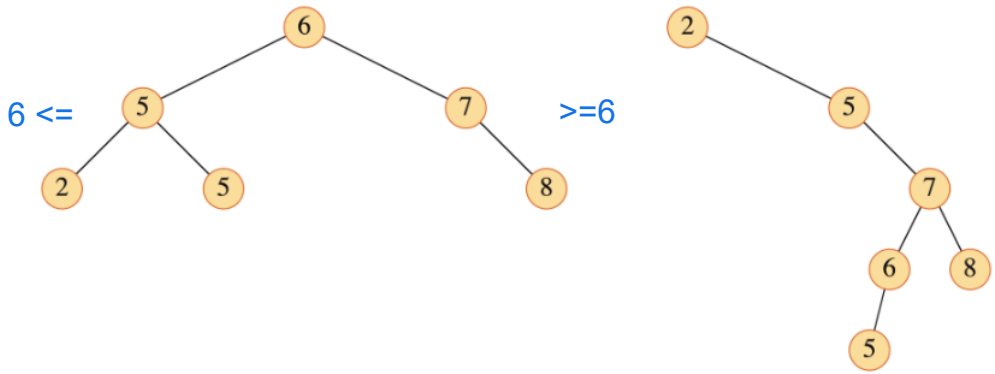
Binary search tree

- A binary tree
- With nodes in inorder

A binary tree with keys in all nodes follows inorder if, for all nodes v, the following holds:

> keys in v's left subtree ≤ key in v ≤ keys in v's right subtree

Examples:



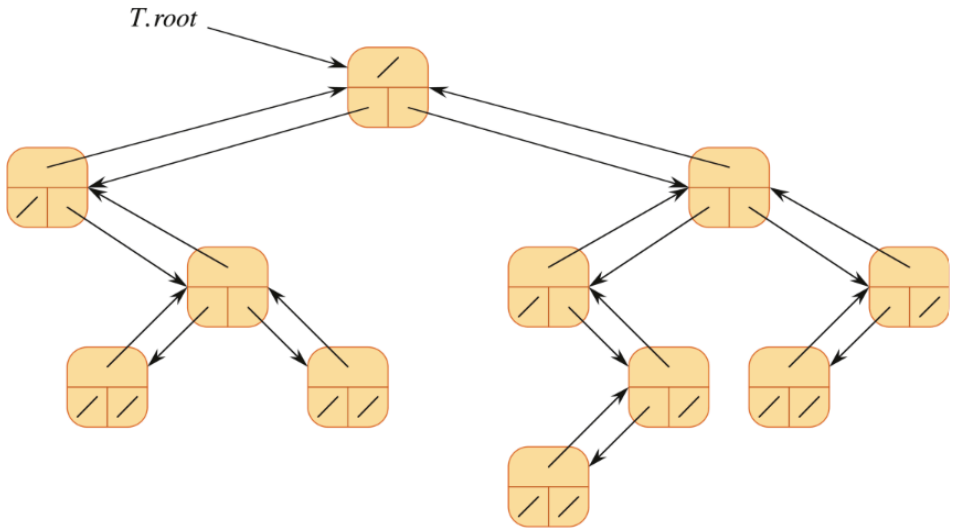
Binary search trees

Typical implementation: Node objects with:

- Reference to parent
- Reference to left subtree
- Reference to right subtree

as well as a tree-object with a reference to the root.

represent the entire binary search tree.
Reference to the root (T.root): This is a pointer to the very first node in the
If the tree is empty, this reference will be NIL/null



if no left child
null

if no right child
null

Node-objects

Java class for nodes:

```
class Node {
  int key;
  Node leftchild;
  Node rightchild;
  Node parent;
  .
  .
  (constructor)
  (other methods)
  .
}
```

Python class for nodes:

```
class Node:
  def __init__(self):
    self.key = None
    self.leftchild = None
    self.rightchild = None
    self.parent = None
  .
  .
  (other methods)
  .
```

Python via lists:

```
Node = [key,leftchild,rightchild,parent]
```

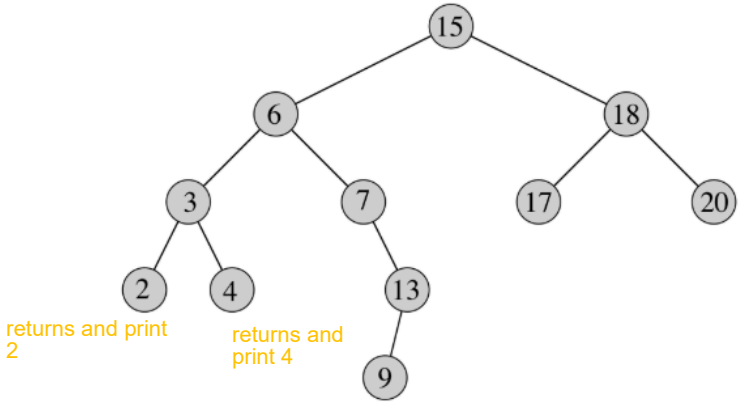
Binary search trees

Due to the definition of inorder

> keys in v's left subtree ≤ key in v ≤ keys in v's right subtree

Binary search trees can be said to contain data in sorted order.

More precisely, an inorder traversal will print the keys in sorted order.

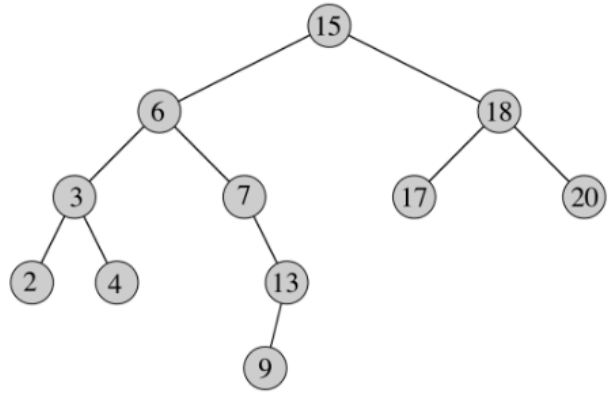


x = node

```
INORDER-TREE-WALK(x)
  if x ≠ NIL                    basecase/bottom
      INORDER-TREE-WALK(x.left)
      print key[x]
      INORDER-TREE-WALK(x.right)
```

Running time: O(n) (O(1) work is done per node in the tree).

Searching in binary search trees



bestcase O(1) (the top node)

worst case runtime
height of the tree is O(height)

```
TREE-SEARCH(x, k)
  if x == NIL or k == key[x]
      return x
  if k < x.key
      return TREE-SEARCH(x.left, k)
  else return TREE-SEARCH(x.right, k)
```

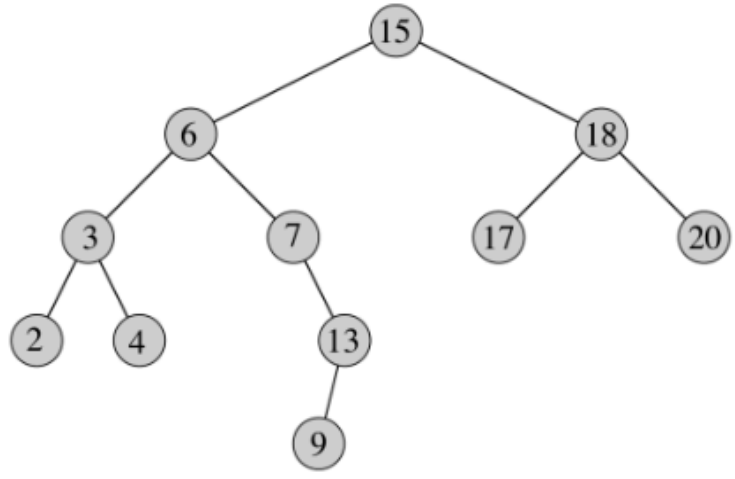
k == key[x]: If the key we are searching for (k) is equal to the key of the current node (key[x]), then we have found the key!

Principle:

> If the searched item is found, it is in the subtree we have reached.

otherwise we will each an empty subtree if the item is not there

Several types of searches in binary search trees



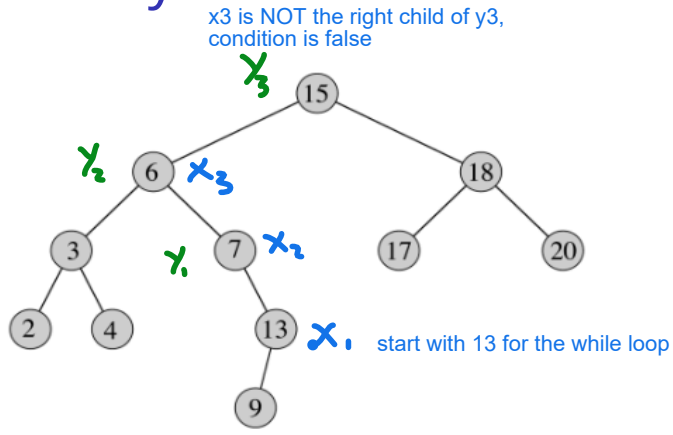

```
TREE-MAXIMUM(x)
  while x.right ≠ NIL
      x = x.right
  return x
```

```
TREE-MINIMUM(x)
  while x.left ≠ NIL
      x = x.left
  return x
```

Principle:

> The searched item is located in the subtree we have reached.

Several types of searches in binary search trees

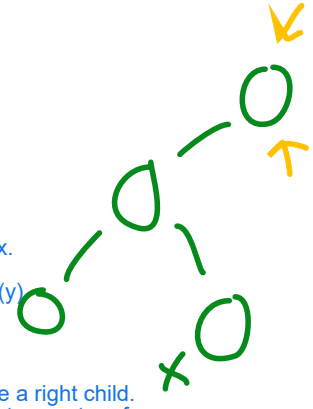



```
TREE-SUCCESSOR(x)
  if x.right ≠ NIL
      return TREE-MINIMUM(x.right)
  y = x.p
  while y ≠ NIL and x == y.right
      x = y
      y = y.p
  return y
```

of the bigger keys, find the smallest

if no right child, set to parent

This while loop is climbing up the tree from x. It continues to go up as long as the current node (x) is a right child of its parent (y).

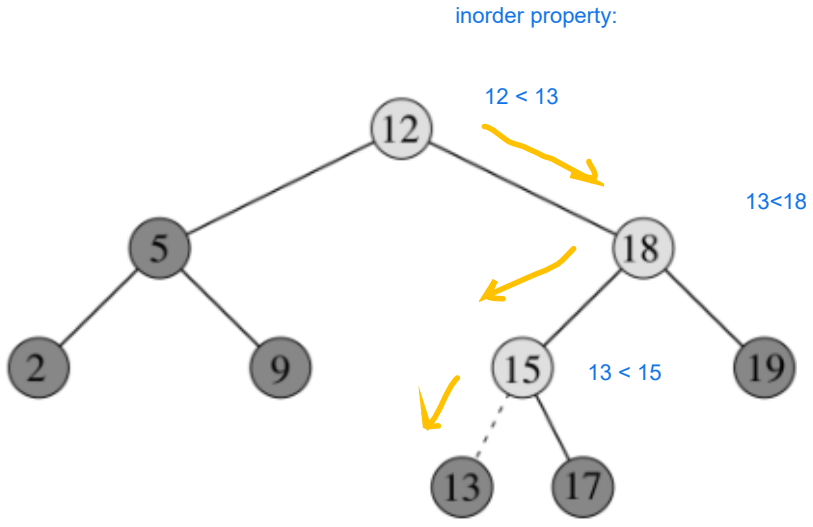


Imagine you are at a node x that doesn't have a right child. To find the successor, we need to find the first ancestor of x that is a left child of its own parent.

Principle:

> Look at the path from x to root. No side trees on it can contain the searched item (because of in-order).

Insertions in binary search trees



- Search downward from the root: go into each node and continue downward into the subtree (right/left), where new elements should be according to the inorder requirements for the node.

- When a leaf (NIL/empty subtree) is reached, replace it with the new node (with two empty subtrees).

Inorder is maintained for nodes on the search path (due to the search rule), and for all other nodes (because they have not received any new descendants in their two subtrees).

Deletions in binary search trees

Deletion of node z:

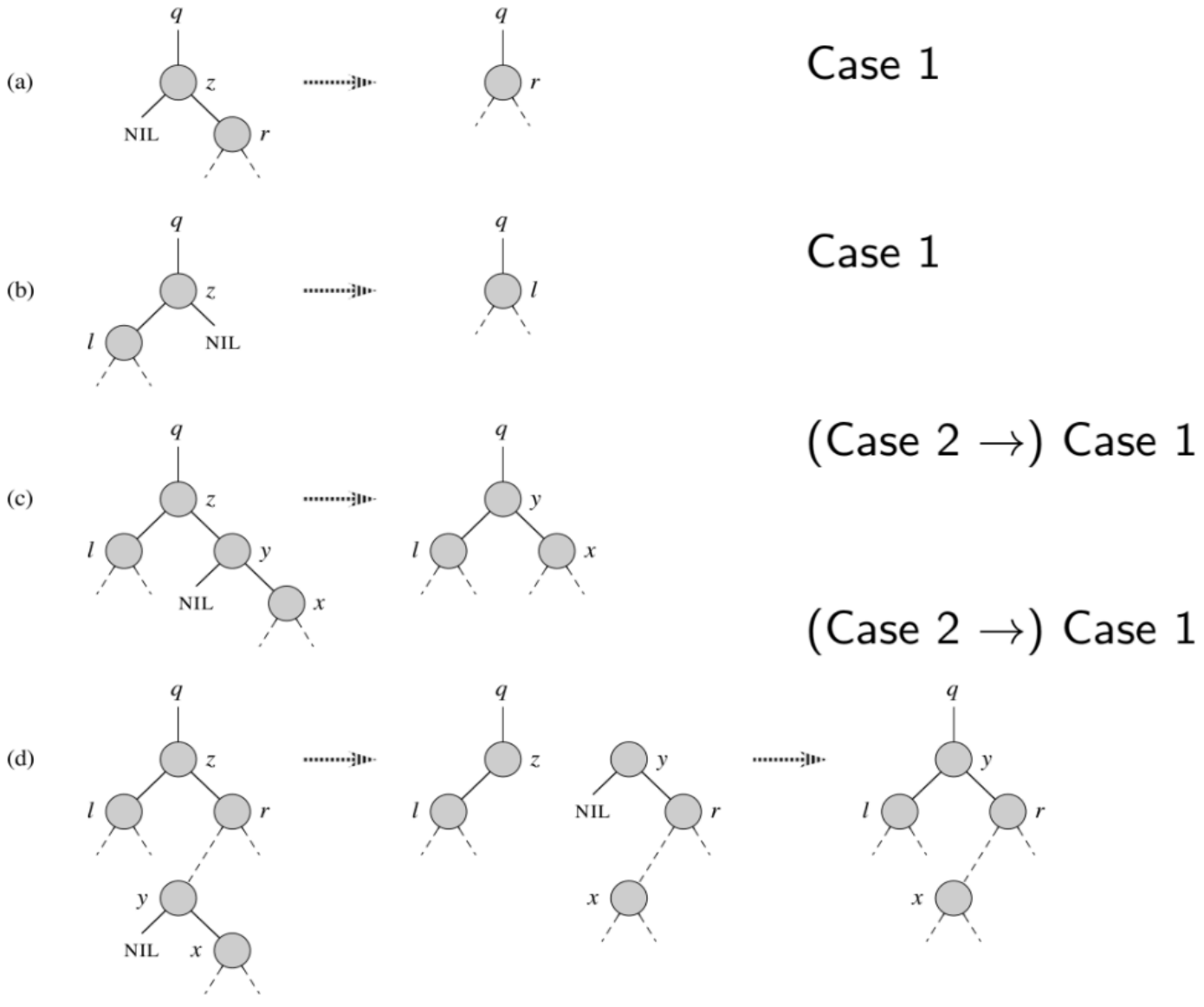
- Case 1: At least one child is NIL: Remove z and this child, let the other child take z's place.

- Case 2: No children are NIL: The successor node y of z is the smallest node in z's right subtree. Remove y (which is a Case 1 deletion, since its left child is NIL), and insert it in z's place.

Both cases leave the tree in inorder: In **Case 1**, no nodes gain new descendants in their two subtrees. In **Case 2**, y (and only y) gains new descendants in its two subtrees, but since y is z's successor, there are no keys in the tree with values between z's and y's keys, so the keys in y's new subtrees maintain the inorder relationship with y, just as they did with z.

Note that structurally, in the tree, all deletions are Case 1 deletions.

Deletions in binary search trees (cases in the book)



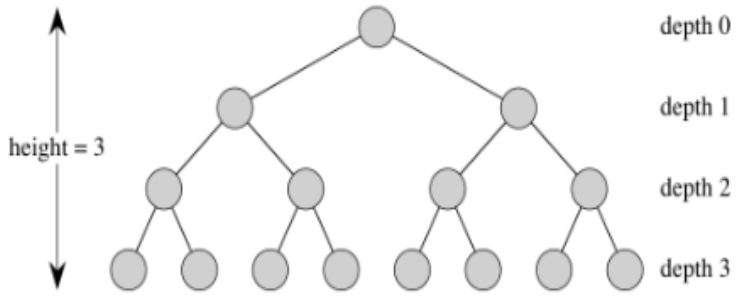
Time for operations in binary search trees

For all operations (except in-order traversal):

Traverse the path from root to leaf.

That is, runtime = O(height).

A tree with height h cannot contain more nodes than the full tree with height h. This contains 2^{h+1} -1 nodes (see slides on heaps).



So for a tree with *n* nodes and height *h*, the following holds:

$$n \leq 2^{h+1} - 1 \Leftrightarrow \log_2(n+1) - 1 \leq h$$

That is, the best possible height is $\log_2 n$ (±1)

Can we keep the height close to optimal – e.g., O(log n) – during updates (insertions and deletions)?

Balanced binary search trees

Can we maintain a height of O(log n) during updates (insertions and deletions)?

Rebalancing (restructuring of the tree) is required after updates, as deep trees may otherwise arise:

Maintenance of O(log n) height first achieved with **AVL trees** [1961].

Many later proposals. One proposal consists of:

- Balance information in nodes.
- Structural requirements based on balance information that ensure O(log n) height.
- Algorithms that restore the structure after an update.

Red-black trees

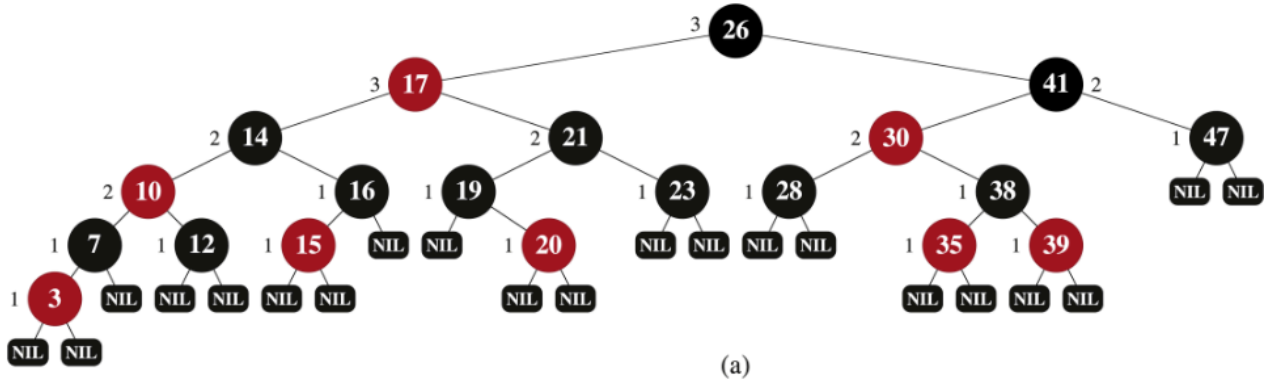
istead of 1/0

Balance information in nodes: 1 bit (called red/black color).

Structural requirements:

- Root and leaves are black. top and bottom
- The same number of blacks on all root-leaf paths.
- No two reds in a row on any root-leaf path. can make the path longer but only double the length, compare to only black nodes

Example:

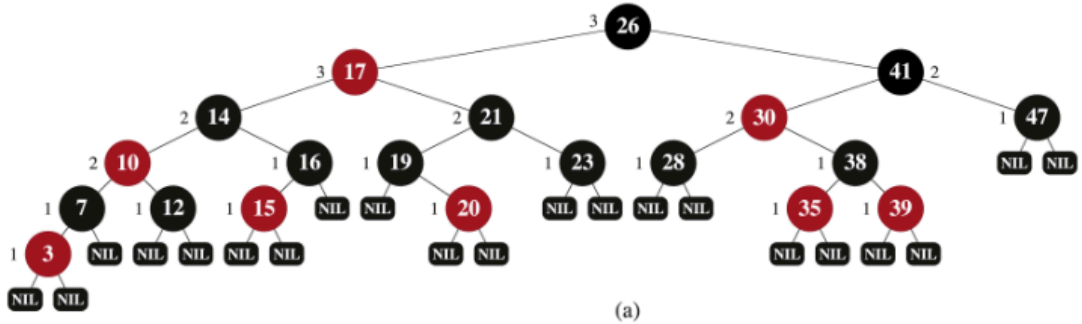


empty

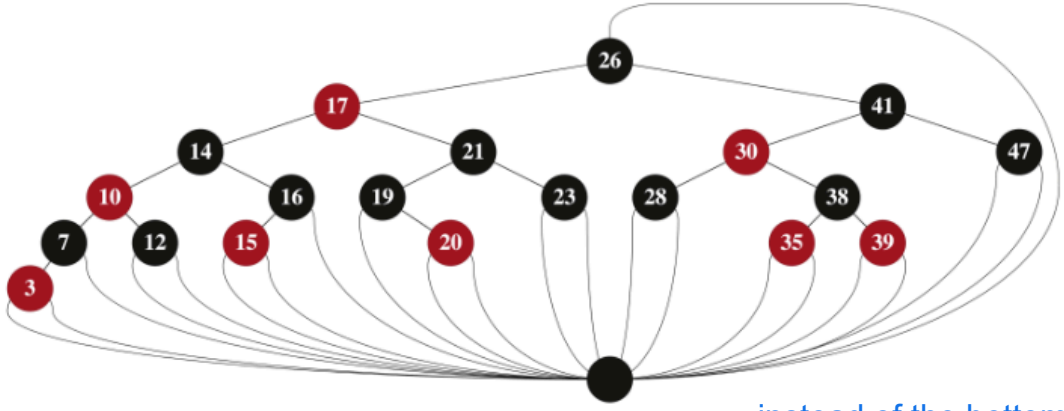
Note: The concept of leaves is used in red-black trees to refer to NIL subtrees (which technically increases the height of a tree by one).

Red-black trees

Other representations in the book (same tree):

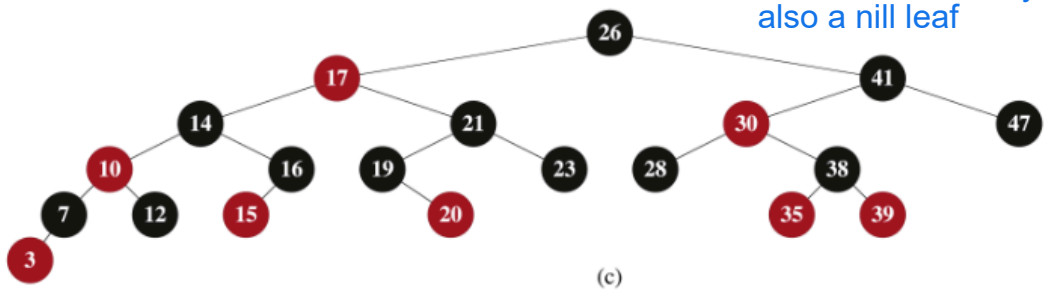


(a)



(b)

instead of the bottom nodes having different empty leaves (nill/null), they can all point to the same. And actually over the top node there is also a nill leaf



(c)

Red-black trees

Structural requirements (recap):

- Roots and leaves are black.
- The same number of black nodes on all root-leaf paths.
- No two red nodes in a row on any root-leaf path.

Do these structural requirements ensure a height of O(log n)? Yes:

k= "black height"

Let the number of black nodes on all root-leaf paths be k. Then all root-leaf paths contain at least k − 1 nodes. Therefore, there are at least k − 1 full levels of nodes in the tree.

cant have less than the blacks

Therefore, $n \geq 2^0 + 2^1 + 2^2 + \cdots + 2^{k-2} = 2^{k-1} - 1$.

From this follows $\log(n+1) \geq k - 1$.

Since there are no two red nodes in a row, the longest root-leaf path contains at most 2(k − 1) edges.

lcould also be written as log k-1

Thus, the height is at most $2(k-1) \leq 2\log(n+1) = O(\log n)$

lower bound

upper bound

Insertion

1. Insert a node in the tree.
2. Remove any existing imbalance (violation of the red-black structure requirements).

Recall insertion: a leaf (NIL) is replaced by a node with two children as leaves.

Imbalance?

Recall insertion: A leaf (NIL) is replaced by a node with two leaves as children.

Violation of red-black structure requirements?

- Root and leaves are black.
- Same number of black nodes on all root-to-leaf paths.
- No two red nodes in a row on any root-to-leaf path.

The two new leaves must be black.

We choose to make the newly inserted node red.

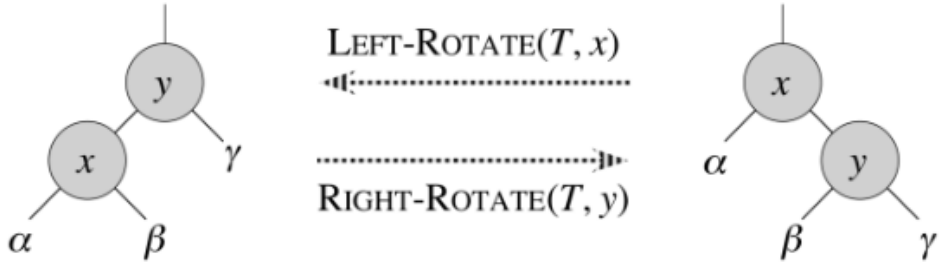
> A possible violation of structure requirements is now: Two consecutive red nodes on a root-to-leaf path somewhere in the tree.

Idea for a plan: If the problem cannot be resolved immediately, push it up in the tree until it can be (hopefully easy to do if it reaches the root).

Rebalancing

Detailed plan: Push the red-red problem upward in the tree, using recolorings and restructurings of the tree.

The basic restructuring will be a rotation (α, β, γ are subtrees, possibly empty):

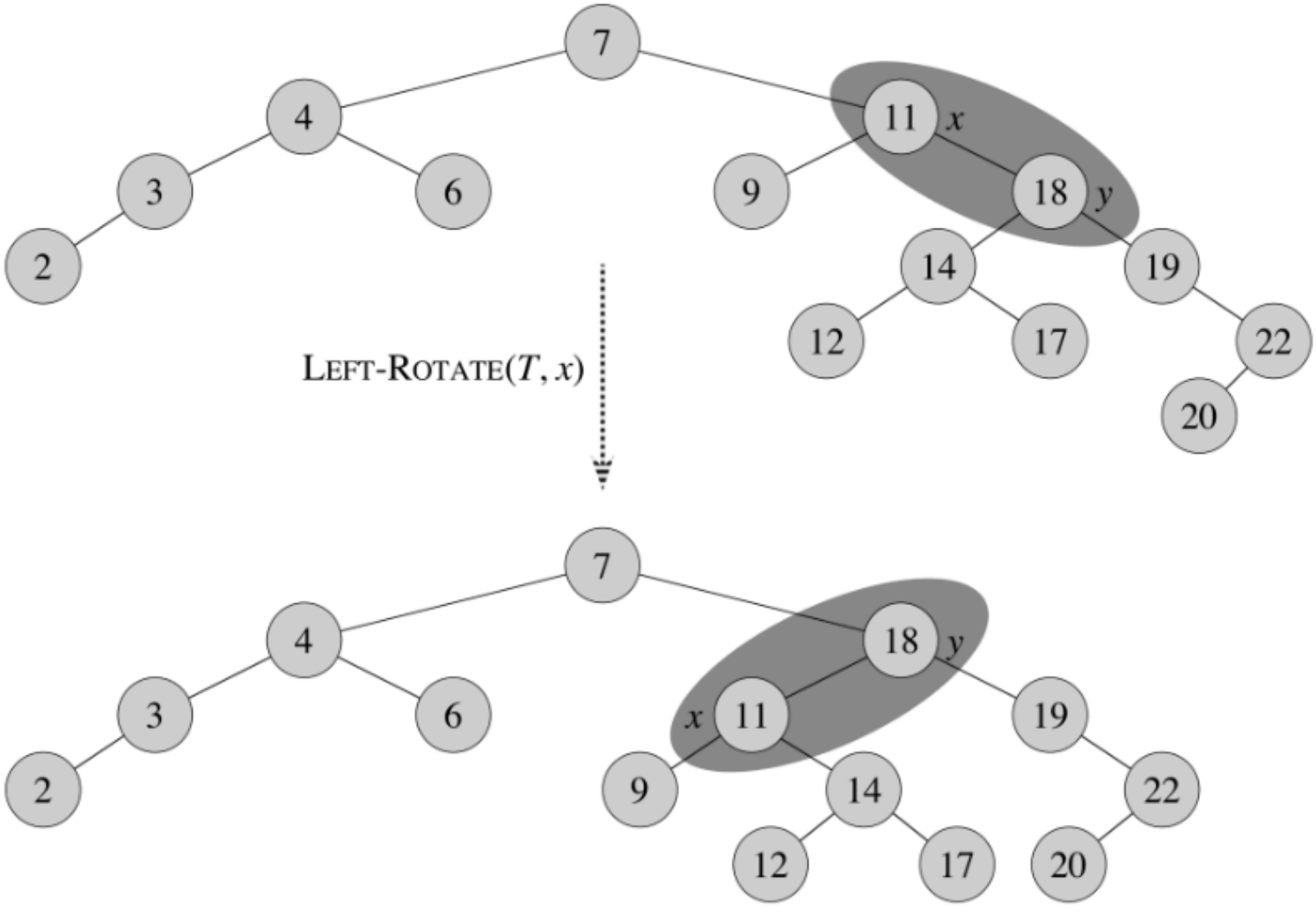


Central observation: Rotations cannot disrupt the in-order sequence in the tree:
Only x and y can have their in-order property violated (all other nodes retain the same elements in their subtrees before and after). However, this does not happen, since the following holds both before and after a rotation:

keys in α ≤ x ≤ keys in β ≤ y ≤ keys in γ

Thus, we do not need to worry about preserving the in-order sequence if we only restructure using rotations.

Example of rotation



Rebalancing plan (after deployment)

Recap of the plan: Push the red-red problem upward in the tree using recoloring and restructuring (rotations) of the tree.

Principle along the way:

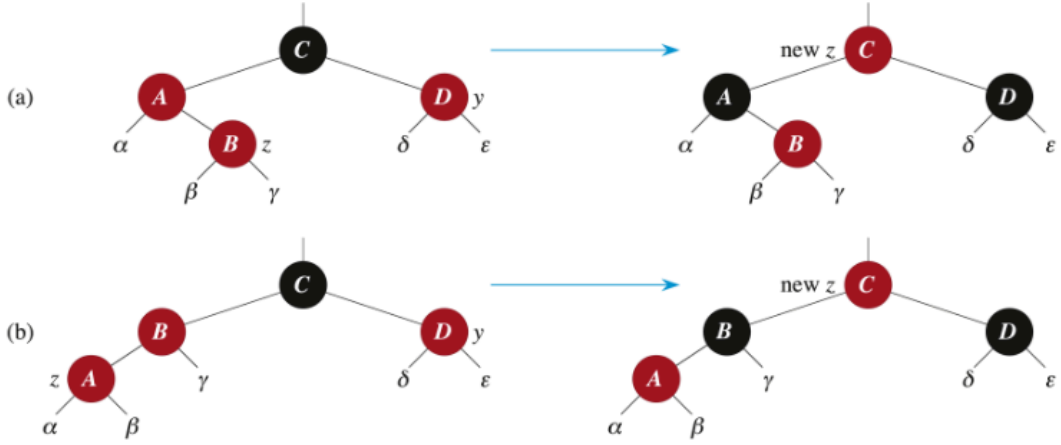
- Two red nodes in a row can occur at most once on a root-to-leaf path in the tree.
- Apart from this, the red-black tree properties are maintained.

Goal: In O(1) time, either remove the problem or push it one step closer to the root.

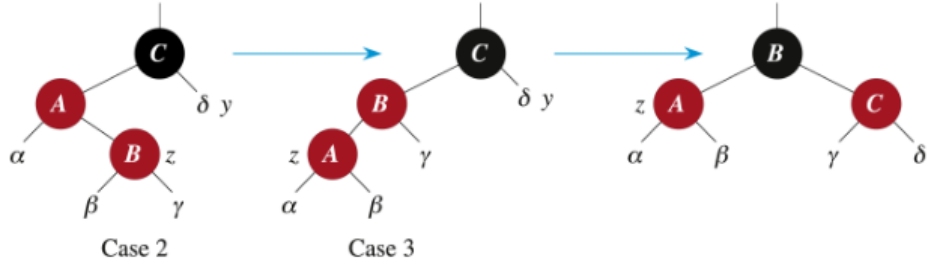
This will result in rebalancing in O(height) = O(log n) time.

Cases in rebalancing (after insertion)

Case 1: Red uncle to z (uncle = parent's sibling).



Case 2: Black uncle to z (uncle = parent's sibling).

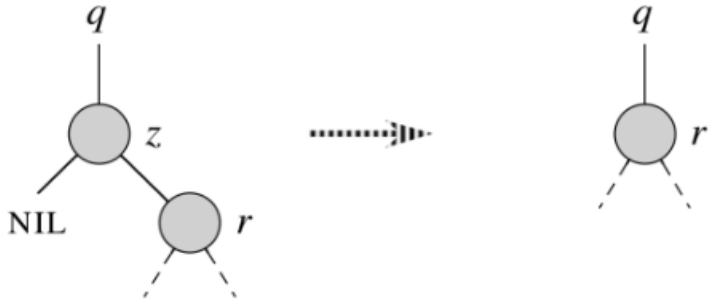


Here, z is the lowest node in the red-red problem. Check that the principle is maintained. Check that the problem is either removed or moved closer to the root (one less black node on the path to the root). If z becomes equal to the root, it can simply be colored black (⇒ all paths gain one more black node).

Deletion

1. Delete a node in the tree.
2. Remove any imbalance that may have arisen (violation of the red-black requirements).

Recall deletion: Structurally, a node is always removed if one of its children is a leaf (NIL), which is also removed.



Imbalance?

Violation of Red-Black Requirements?

- Root and leaves are black.
- The same number of black nodes on all root-to-leaf paths.
- No two consecutive red nodes on any root-to-leaf path.

Removed node is red: All red-black requirements are still met.

Removed node is black: The same number of black nodes is no longer maintained on all paths.

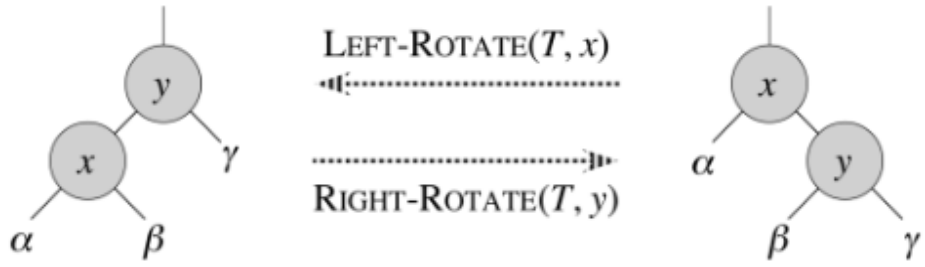
A very useful formulation:

> Let the second child of the removed node be "blackened" and count as "one more" black than its actual color indicates when counting black nodes on paths (blackened black = 2 black, blackened red = 1 black). Then, all requirements are met, except for the existence of a blackened node.

Idea for a plan: If the problem cannot be solved immediately, push it upward in the tree until it can be resolved (hopefully easy to do if it reaches the root).

Rebalancing

Push the blackened node upward in the tree using recoloring and rotations:



Principle along the way:

- At most one node in the tree is blackened.
- If the blackening is counted, the red-black requirements are met.

Simple stopping cases:

- The blackened node is red ⇒ the blackening can be removed by coloring the node black.
- The blackened node is the root ⇒ the blackening can simply be removed (⇒ all paths get one less black node).

(So, it is enough to consider the case where the blackened node is black.)

Rebalancing

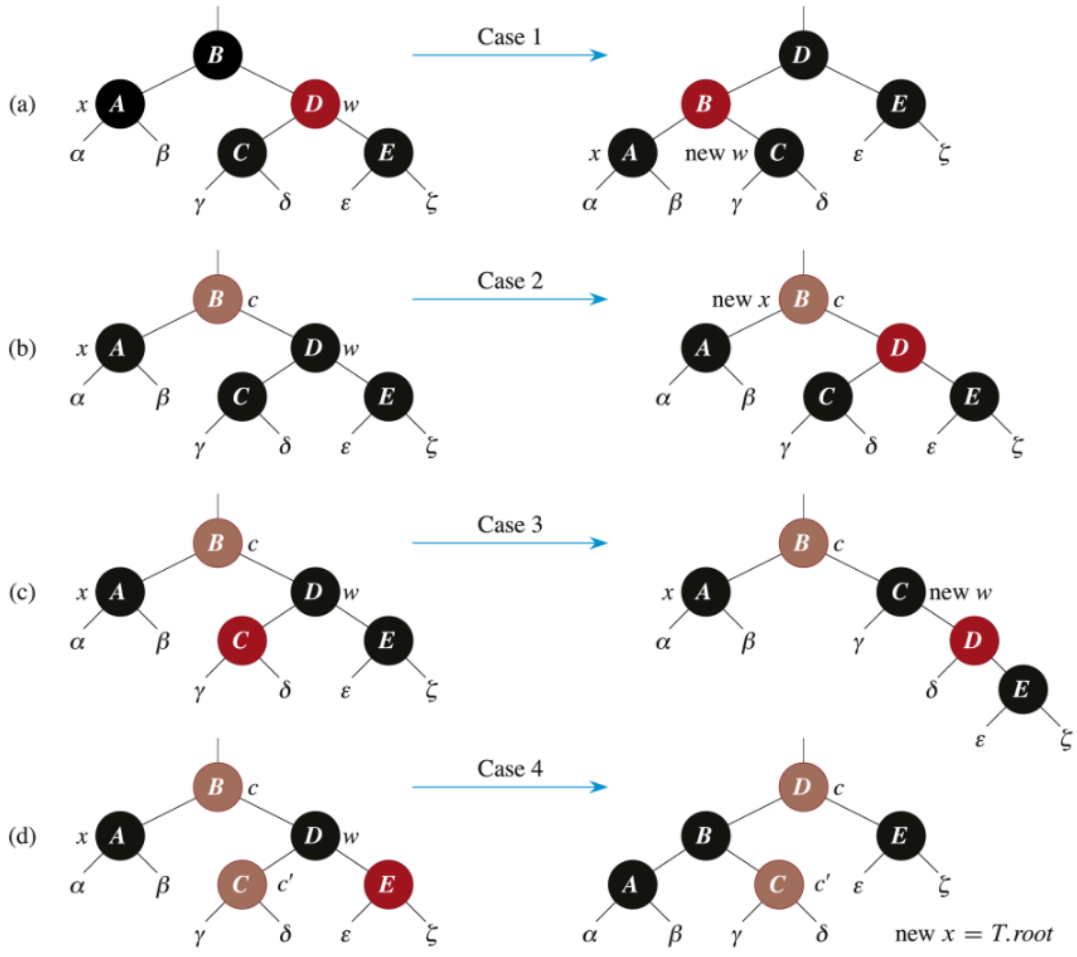
Goal: In O(1) time, remove the problem or push it one step closer to the root.

This will result in rebalancing in O(height) = O(log n) time.

Cases for the blackened red-black node x with sibling w:

1. Red sibling.
2. Black sibling, and it has two black children.
3. Black sibling, and its nearest child is red, while the farthest is black.
4. Black sibling, and its farthest child is red.

Cases in rebalancing (after deletion)



Here is x a blackened node. Check that the principle is maintained. Check O(1) time before blackening is removed or moved one step closer to the root.