CS151 Intro to Data Structures

Balanced Search Trees, AVL Trees

Announcements

Last lab will be this week

HW07 due Friday 12/08
    Hashmaps & Sorting

HW08 due 12/14

HW07

- Check correctness on small `ArrayList<Integer>`
- All five of your deduplication methods should return identical lists
- The one that calls `Collections.sort` is probably most trust-worthy
- `doubleHash` should result in better hashing stats over `linearHash`

Faculty Interview/Mock Lecture

- Friday 12/08 – 11-11am
- Binary Search Tree
- Location: TBD
- Tea & Snacks

Outline

Review Balanced Binary Trees

AVL Trees

Splay Trees

Skip List

Sets

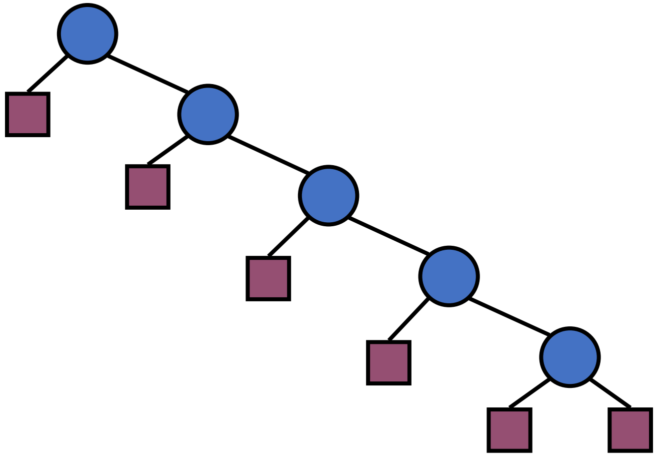
Binary Search Trees

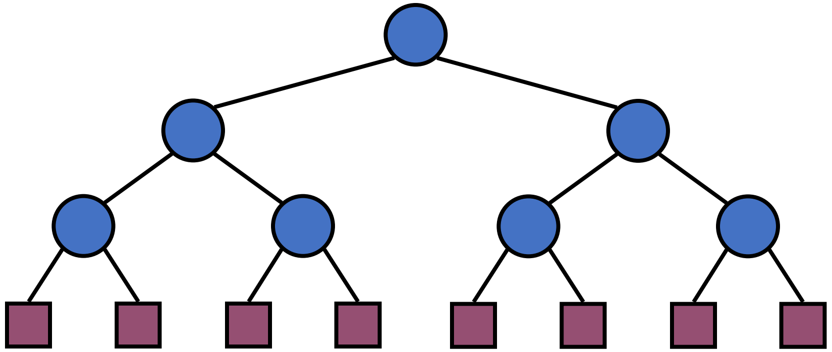
Performance is directly affected by the height of tree

All operations are $O(h)$

- $h = O(n)$ worst case
- $h = O(logn)$ best case

Expected $O(logn)$ if tree is balanced





Balanced Trees

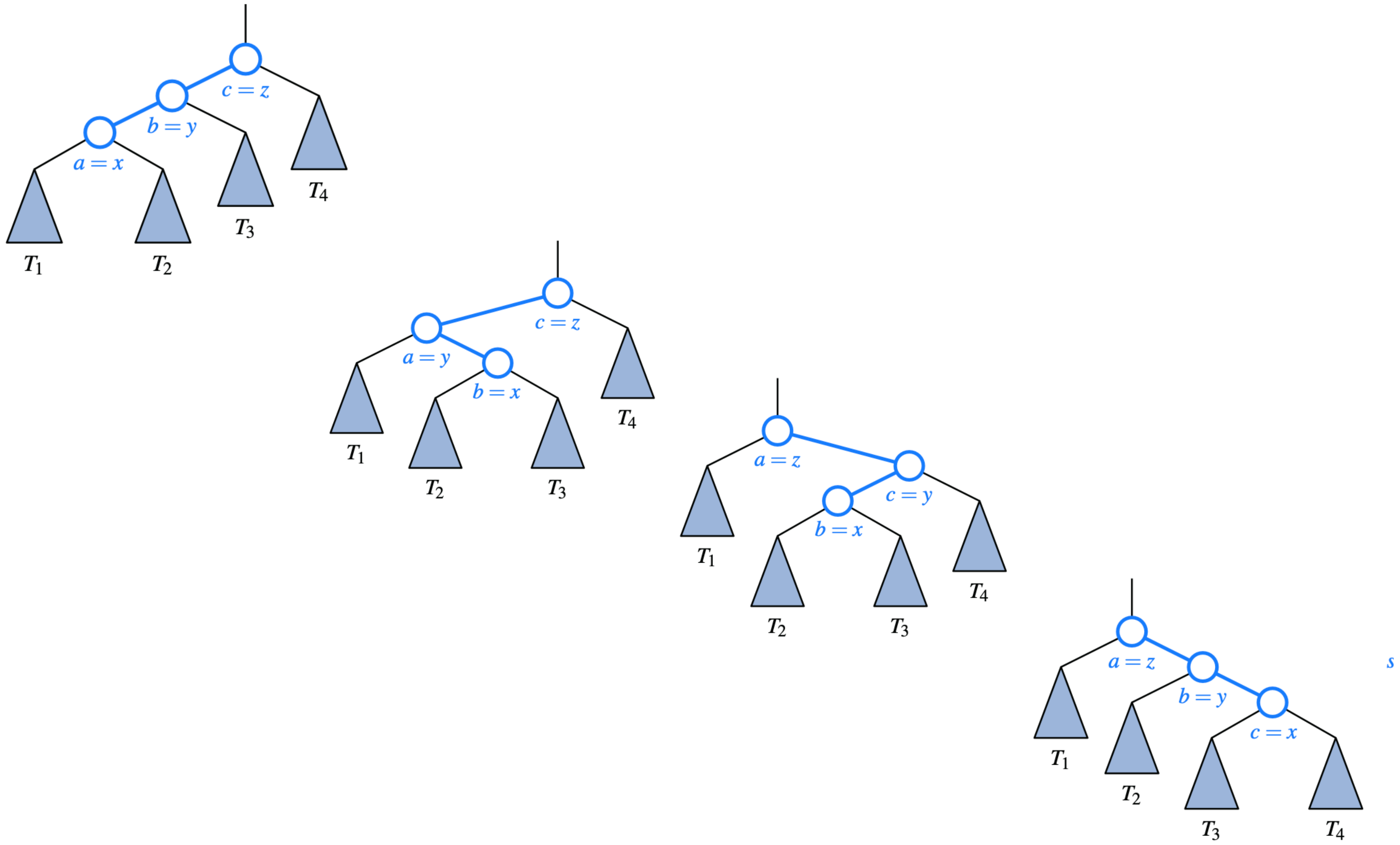
- The difference between the height of the left and right subtree for any node is at most 1
- Left subtree of a node is balanced
- Right subtree of a node is balanced

Types of imbalances

- Left Left
- Left Right
- Right Left
- Right Right

Types of imbalances

- Left Left
- Left Right
- Right Left
- Right Right



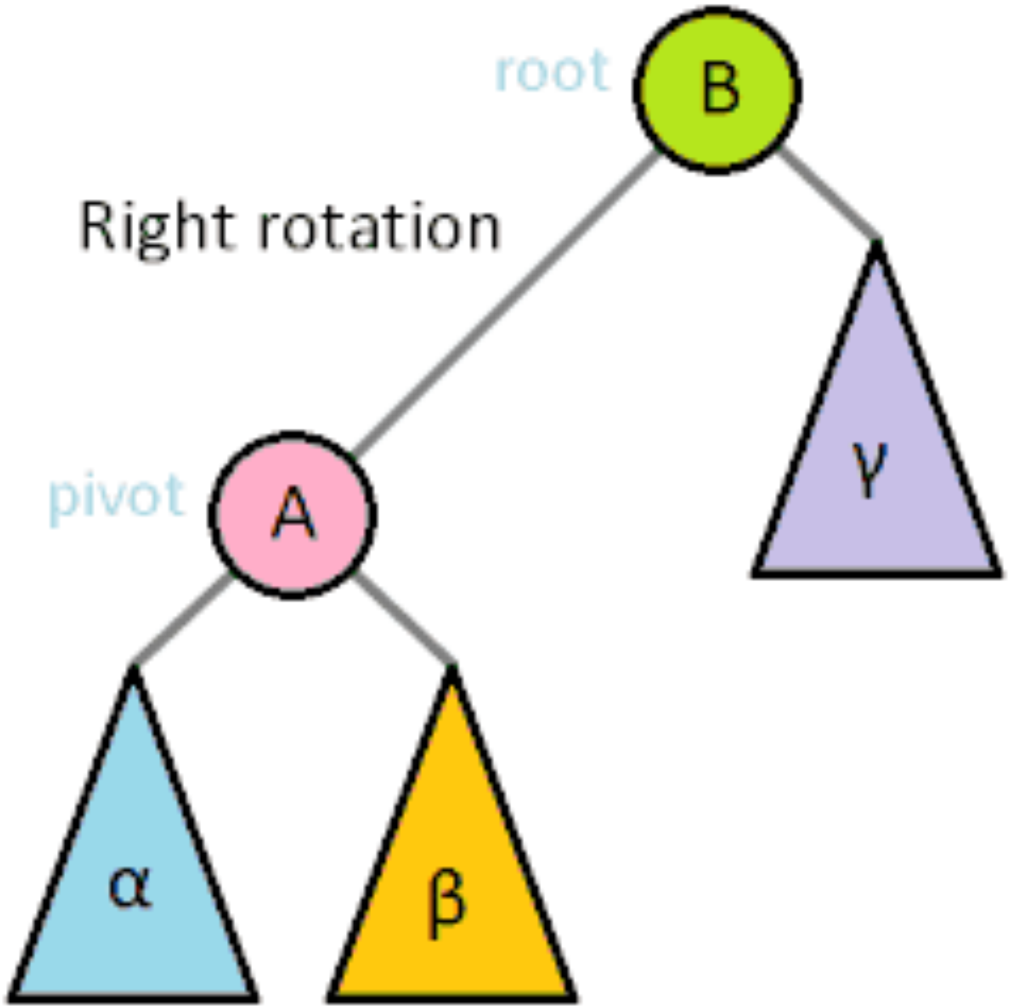
Rotations

Right rotation:

- Root node's left child becomes the new root
- Root node becomes the left child's right child

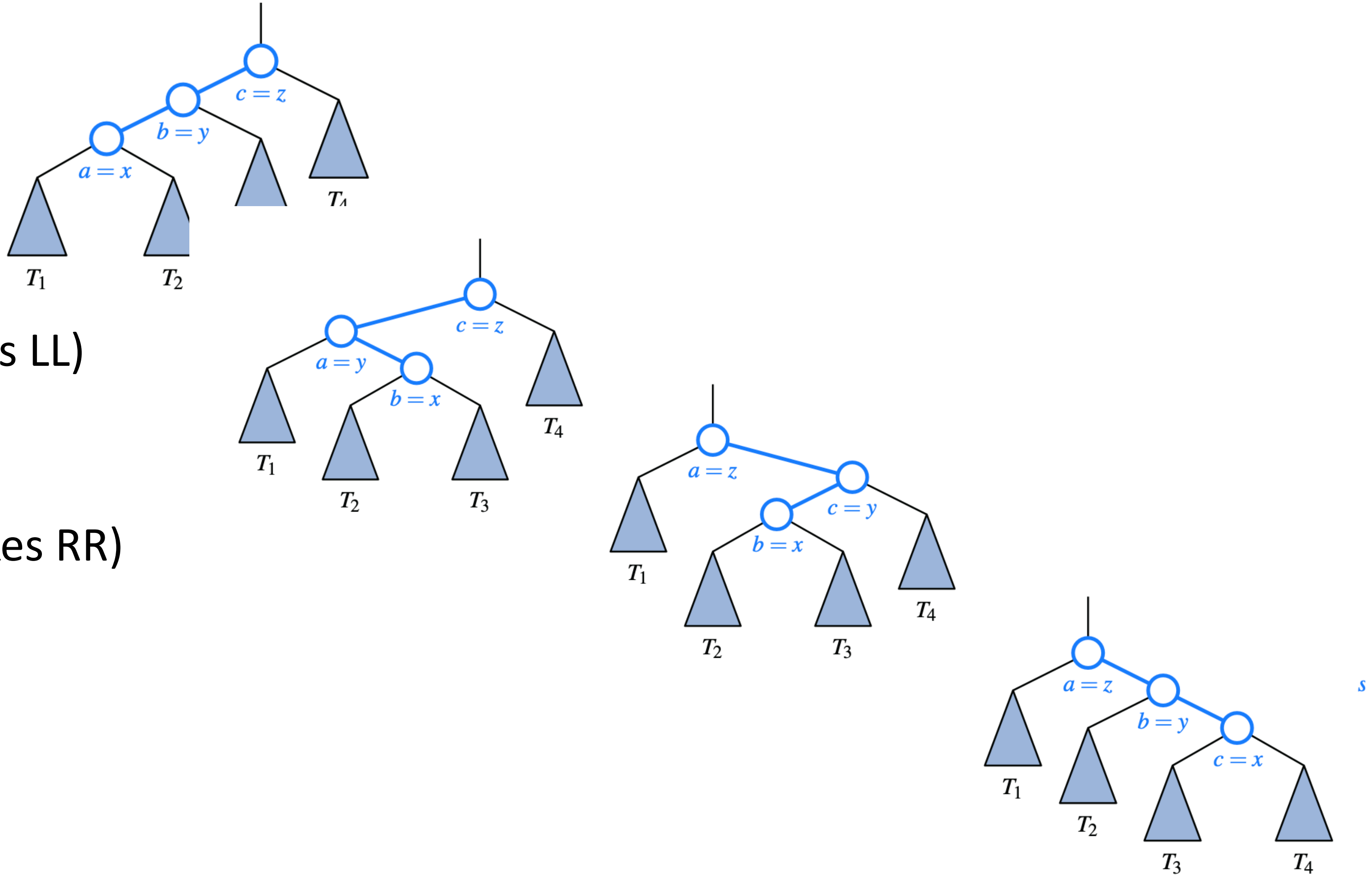
Left rotation:

- Root node's right child becomes the new root
- Root node becomes the right child's left child



Types of imbalances

- Left Left
  - Right rotation
- Left Right
  - Left rotation (makes LL)
  - Then Right rotation
- Right Left
  - Right rotation (makes RR)
  - Then left rotation
- Right Right
  - Left rotation



Outline

Double Hashing Review (Homework 07)

Balanced Binary Trees

AVL Trees

Splay Trees

Red-Black Trees

AVL Tree

Height of a subtree is the number of edges on the longest path from subtree root to a leaf

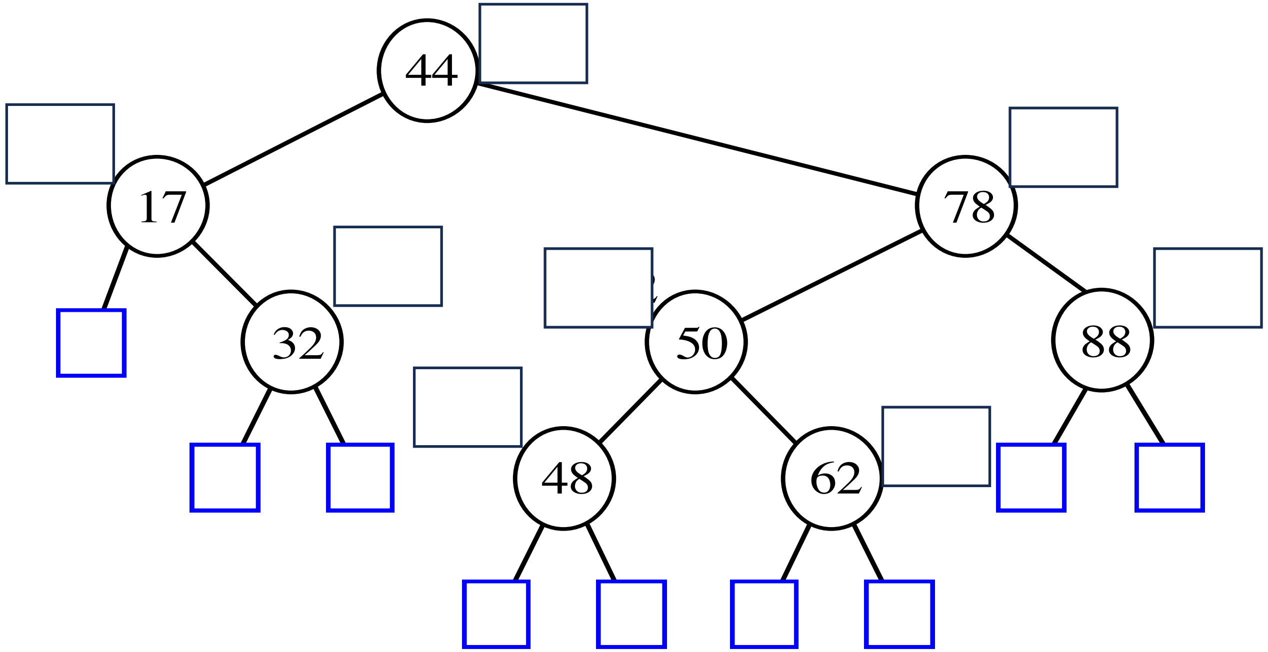
Height-balance property

- For every internal node, the heights of the two children differ by at most 1

Any binary tree satisfying the height-balance property is an AVL tree

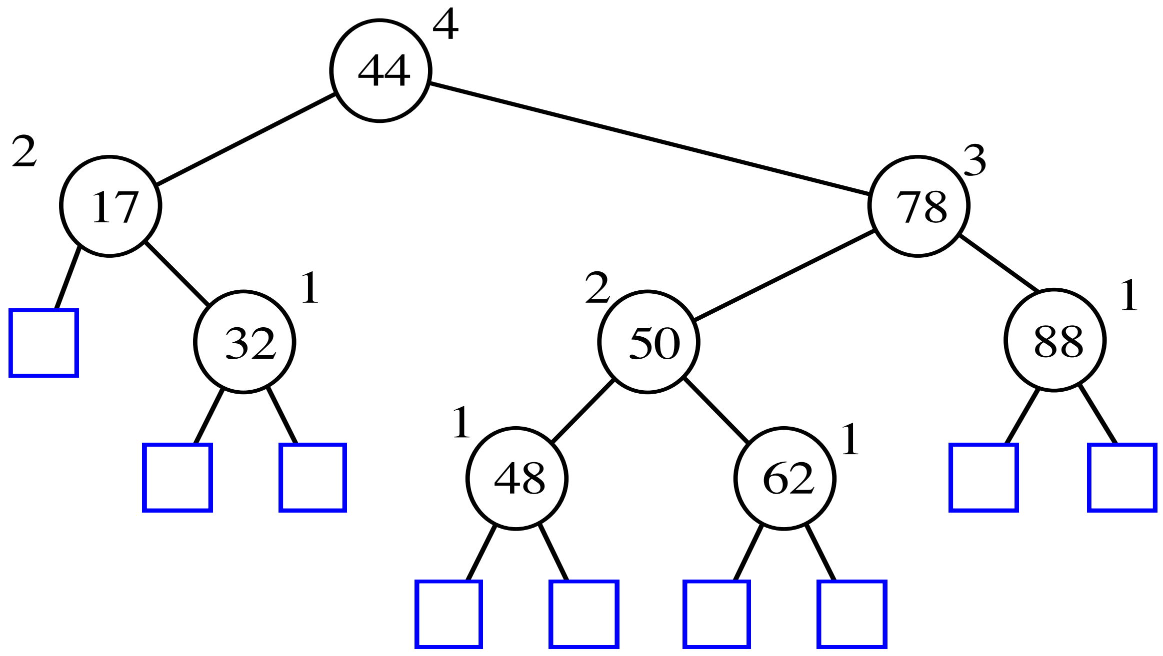
AVL Tree Example

- leaves are sentinels and have height 0



AVL Tree Example

- leaves are sentinels and have height 0



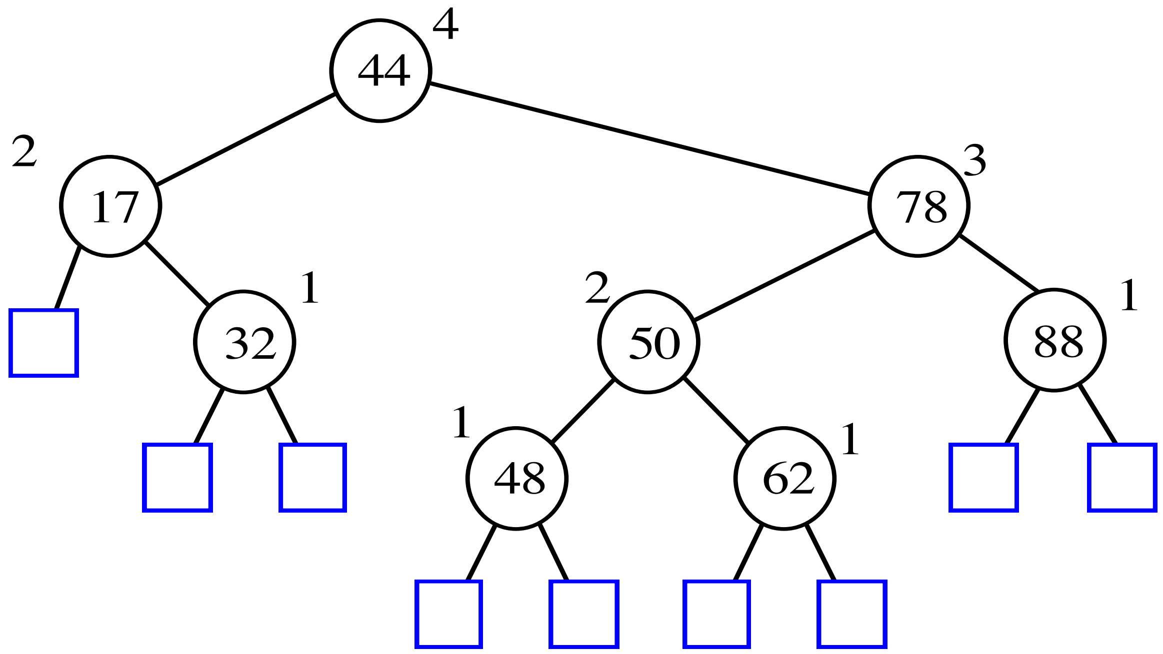
AVL height

The height of an AVL is $O(logn)$

$n(h)$ denotes the number of minimum internal nodes for an AVL with height h

- $n(1) = 1$ and $n(2) = 2$
- $n(h) = 1 + n(h-1) + n(h-2)$
- $n(h) > 2 \cdot n(h-2) > 2^i \cdot n(h-2i)$
- $h - 2i = 1 \implies i = \frac{h}{2} - 1$
- $\log\big(n(h)\big) = \frac{h}{2} - 1 \implies h < 2\log\big(n(h)\big) + 1$

Insert 54



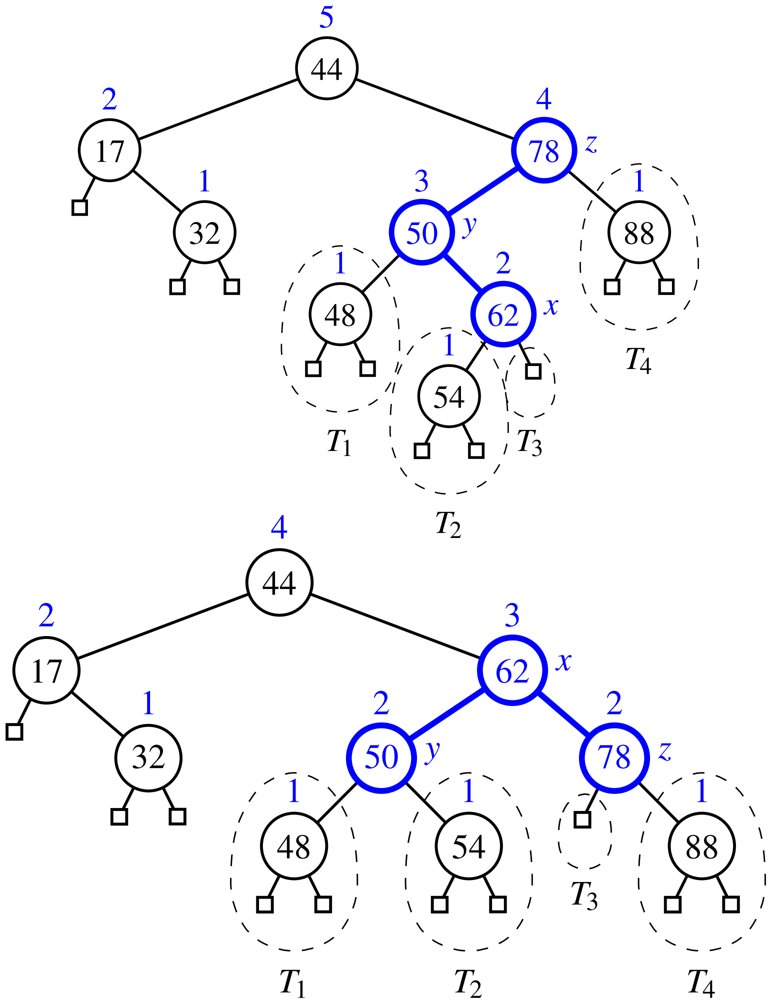
Insertion (54)

New node always has height 1

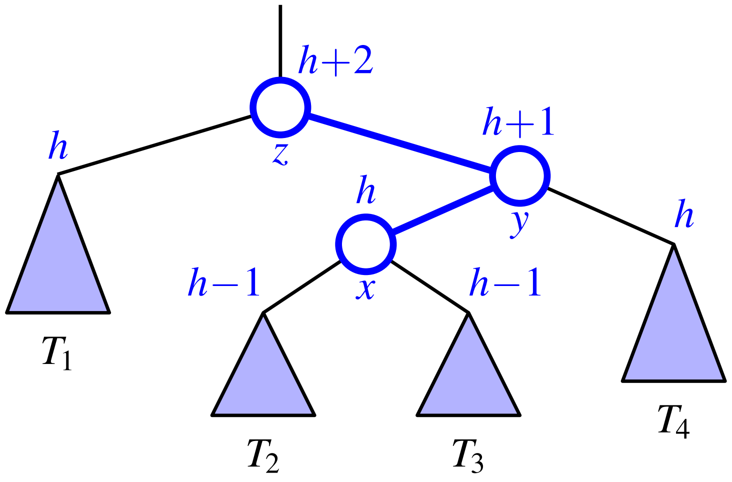
Parent may change height

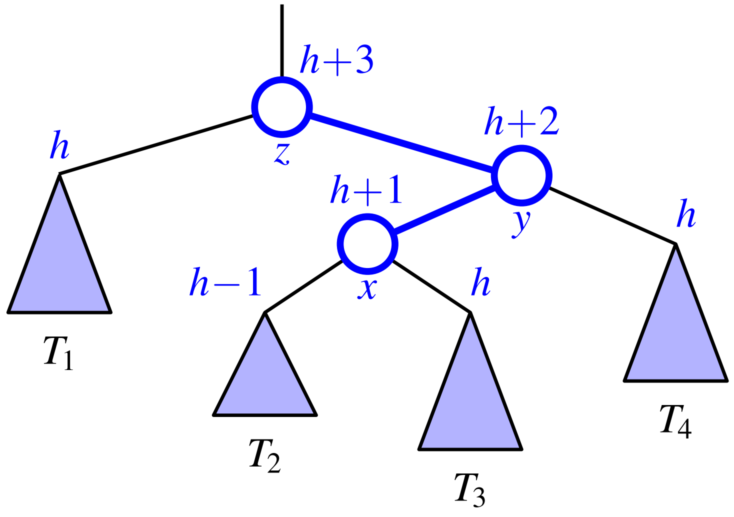
All ancestors may become unbalanced

Perform rotations for unbalanced ancestors



$O(1)$ Rotation Restores Global Balance

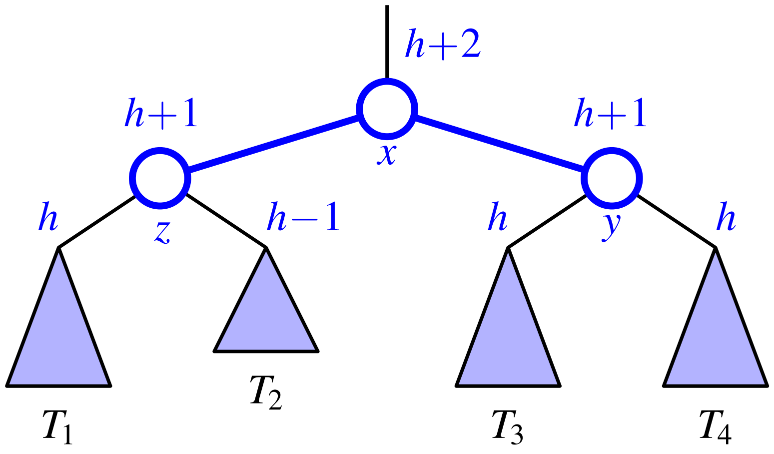




Insert into T_3

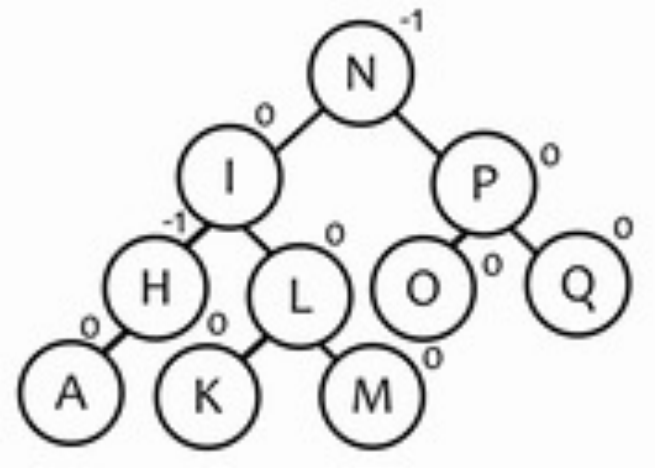
After rebalance:

- x, y and z are balanced after
- root of subtree returns to height $h + 2$, as before



Exercise

- Create an AVL tree by inserting the nodes in this order:
  - M, N, O, L, K, Q, P, H, I, A

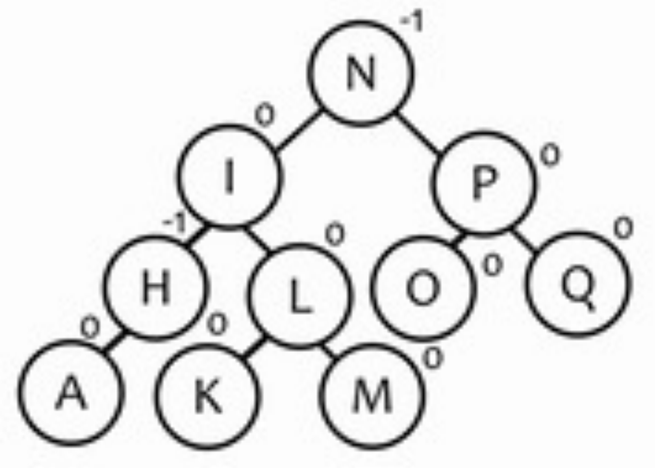


- AVL balance marked on nodes
- balance(n) = height of right subtree – height of left subtree
- AVL balance property: |balance(n)| ≤ 1

AVL Animation

Exercise

- Create an AVL tree by inserting the nodes in this order:
  - M, N, O, L, K, Q, P, H, I, A



- AVL balance marked on nodes
- balance(n) = height of right subtree – height of left subtree
- AVL balance property: |balance(n)| ≤ 1

Rebalance: no null checks

```
rebalance(n):
  updateHeight(n) // update height from children
  lh = n.left.height rh = n.right.height
  if (lh > rh+1) // left subtree too tall
    llh = n.left.left.height lrh = n.left.right.height
    if (llh >= lrh)
      return rotateRight(n)  //left-left
    else
      return rotateLeftRight(n) //left-right
  else if (rh > lh+1) // right subtree too tall
    // … symmetric
  else return n // no rotation
```

Helpers

```
rotateRight(r):
  p = r.left
  r.left = p.right
  p.right = r
  updateHeight(r)
  updateHeight(p)
  // let caller set parent
  // return new subtree root
  return p
```

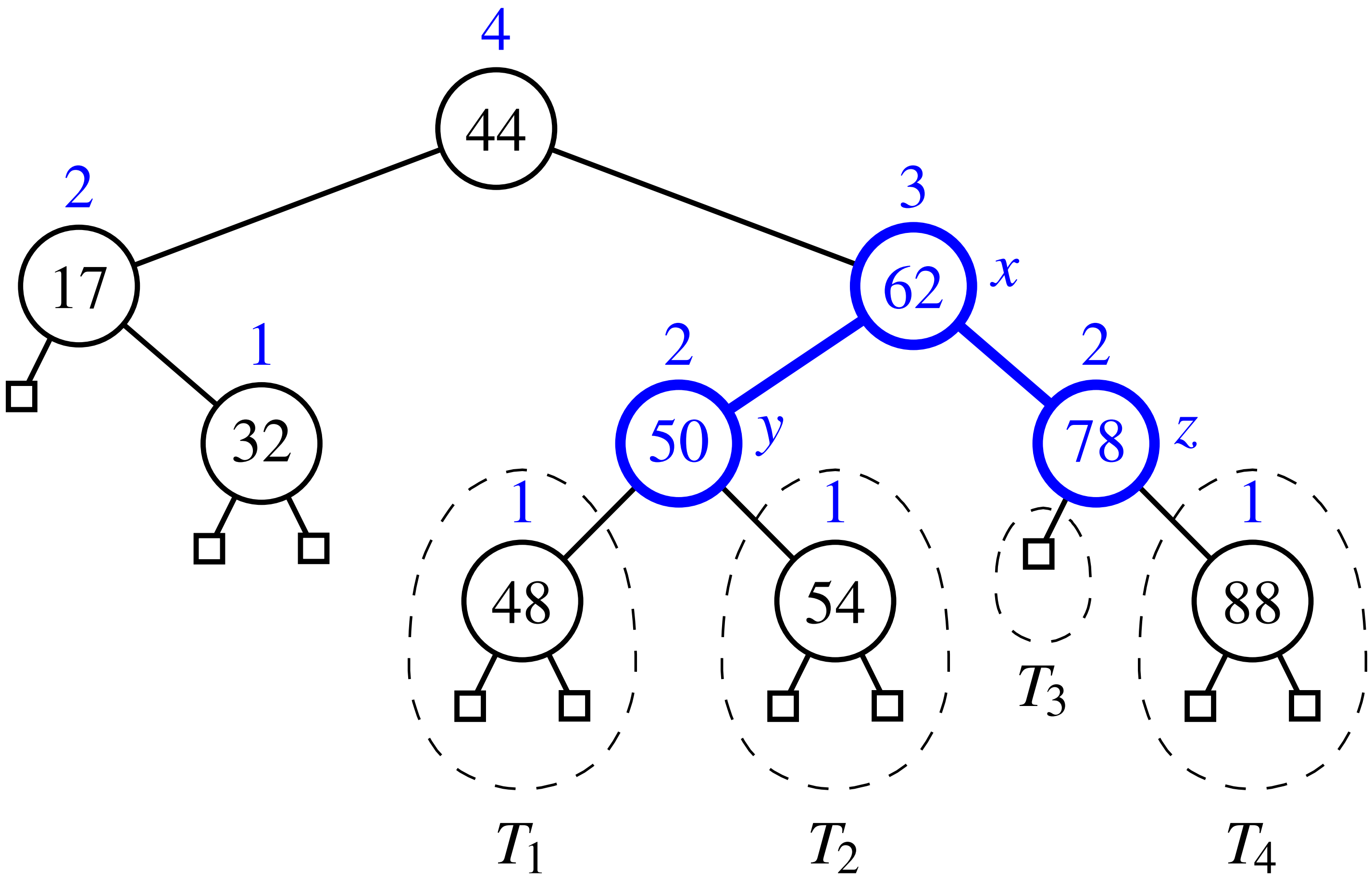
```
updateHeight(n):
  lh = n.left.height
  rh = n.right.height
  height = 1+max(lh, rh)
```

```
rotateLeftRight(r):
  r.left = rotateLeft(r.left)
  return rotateRight(r)
```

Insert with `parent`

```
insertRec(root, key):
  if root == null:
    return new Node(key)
  if root.key > key:
    root.left = insertRec(root.left, key)
    root.left.parent = root
  else
    root.right = insertRec(root.right, key)
    root.right.parent = root
  return root
```

Delete 32



Deletion

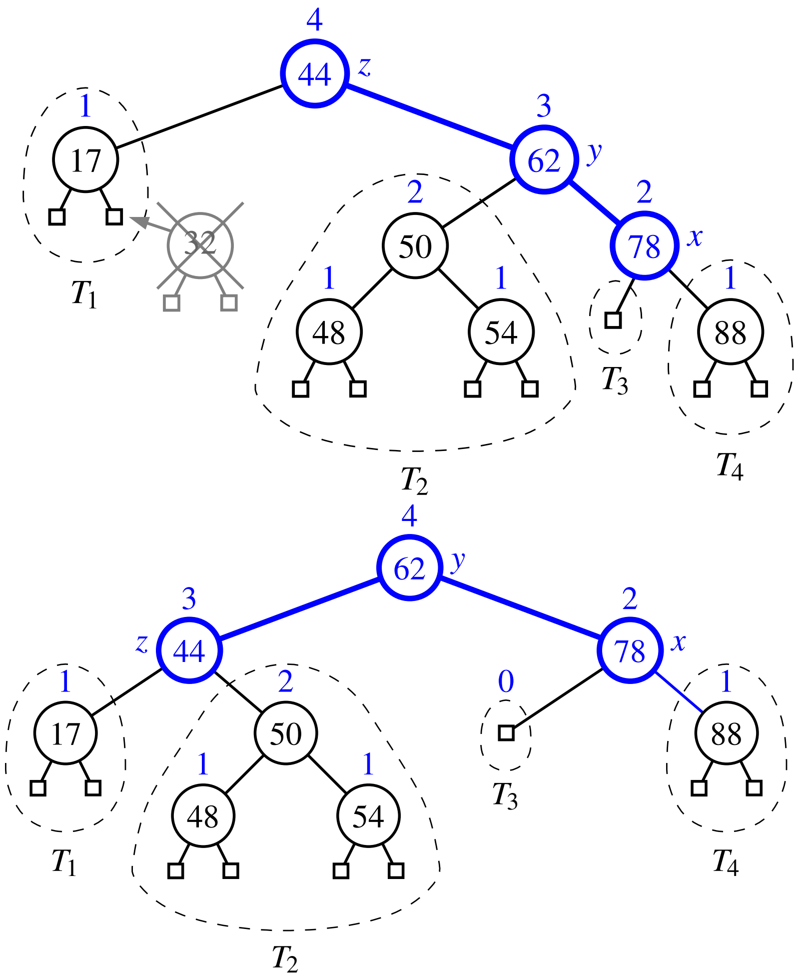
Deletion structurally removes a node with 0 or 1 child

- predecessor has 0 or 1 left child
- successor has 0 or 1 right child

Deletion may reduce the height of parent

Ancestors may become unbalanced

Rotate to rebalance just like insertion



$O(logn)$ Rotations

Unlike insertion where rotation of the nearest unbalanced ancestor restores the balance globally

On deletion, rotation of the nearest unbalanced ancestor only guarantees balance locally to the subtree

Worst-case requires $O(logn)$ rotations up the tree to restore balance globally

Performance of `AVLTreeMap`

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

Book's Implementation of AVL

- 17 classes!
- Interfaces
  - `Entry`
  - `Position`
  - `Queue`
  - `Tree`
  - `BinaryTree`
  - `Map`
  - `SortedMap`
- Abstract classes:
  - `AbstractTree`
  - `AbstractBinaryTree`
  - `AbstractMap`
  - `AbstractSortedMap`
- Concrete classes
  - `SinglyLinkedList`
  - `LinkedQueue`
  - `LinkedBinaryTree`
  - `TreeMap`
  - `AVLTreeMap`

Outline

Review Balanced Binary Trees

AVL Trees

Splay Trees

Skip List

Sets

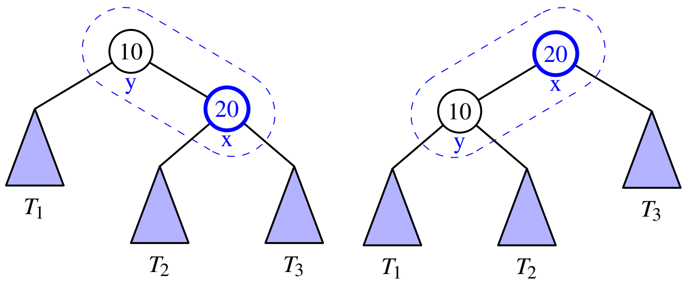
Splay Tree

- A binary search tree that doesn't enforce a $O(logn)$ bound on the height
- Efficiency is achieved due to a move-to-root operation, called splaying
- Performed at the leaf reached during every insert, delete and search
- Causes the more frequently accessed elements to be near the top

Splaying

- Swapping a BST node x up depends on the relative position of x, its parent y and its grandparent z

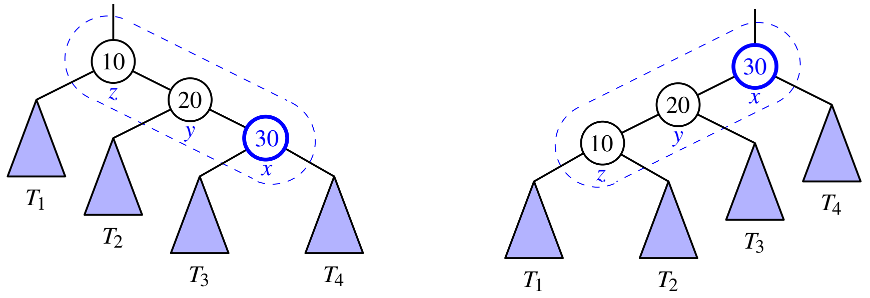
- Zig/zag: y has no parent



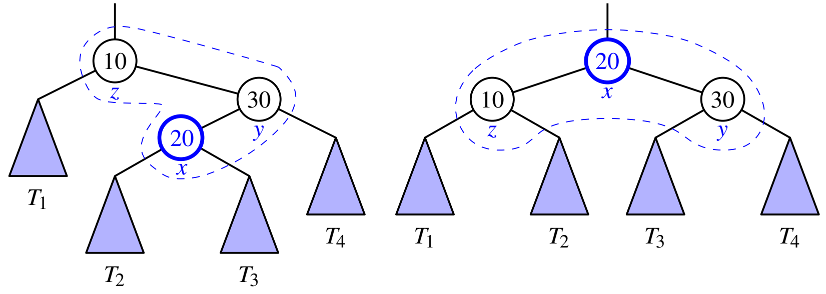
- Splaying will continue these rotations until x becomes root

Splaying

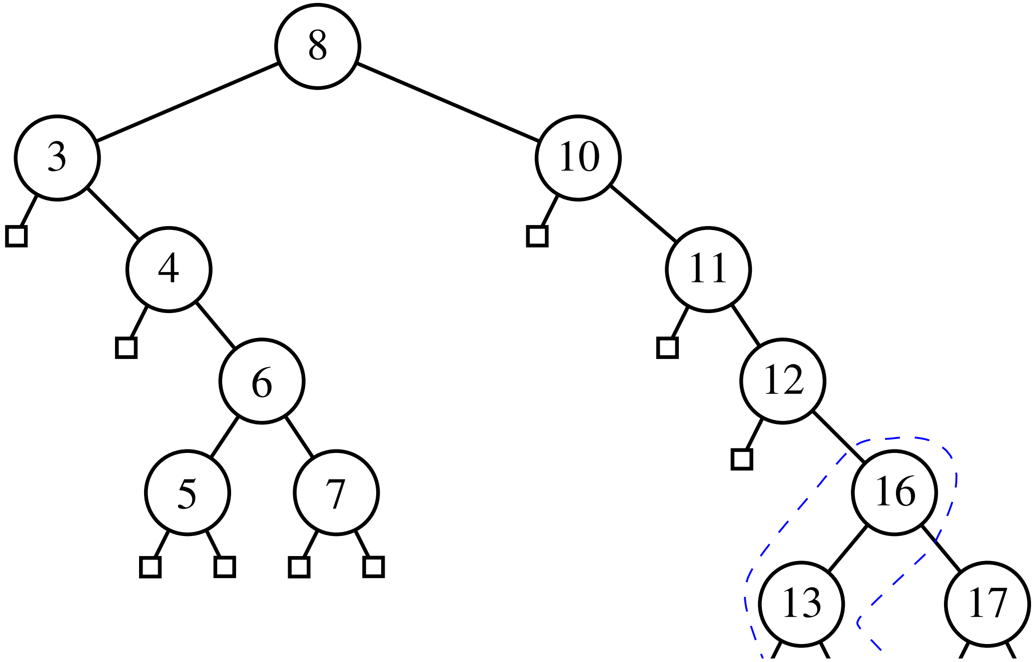
- zig-zig (zag-zag): x and y are both right/left children



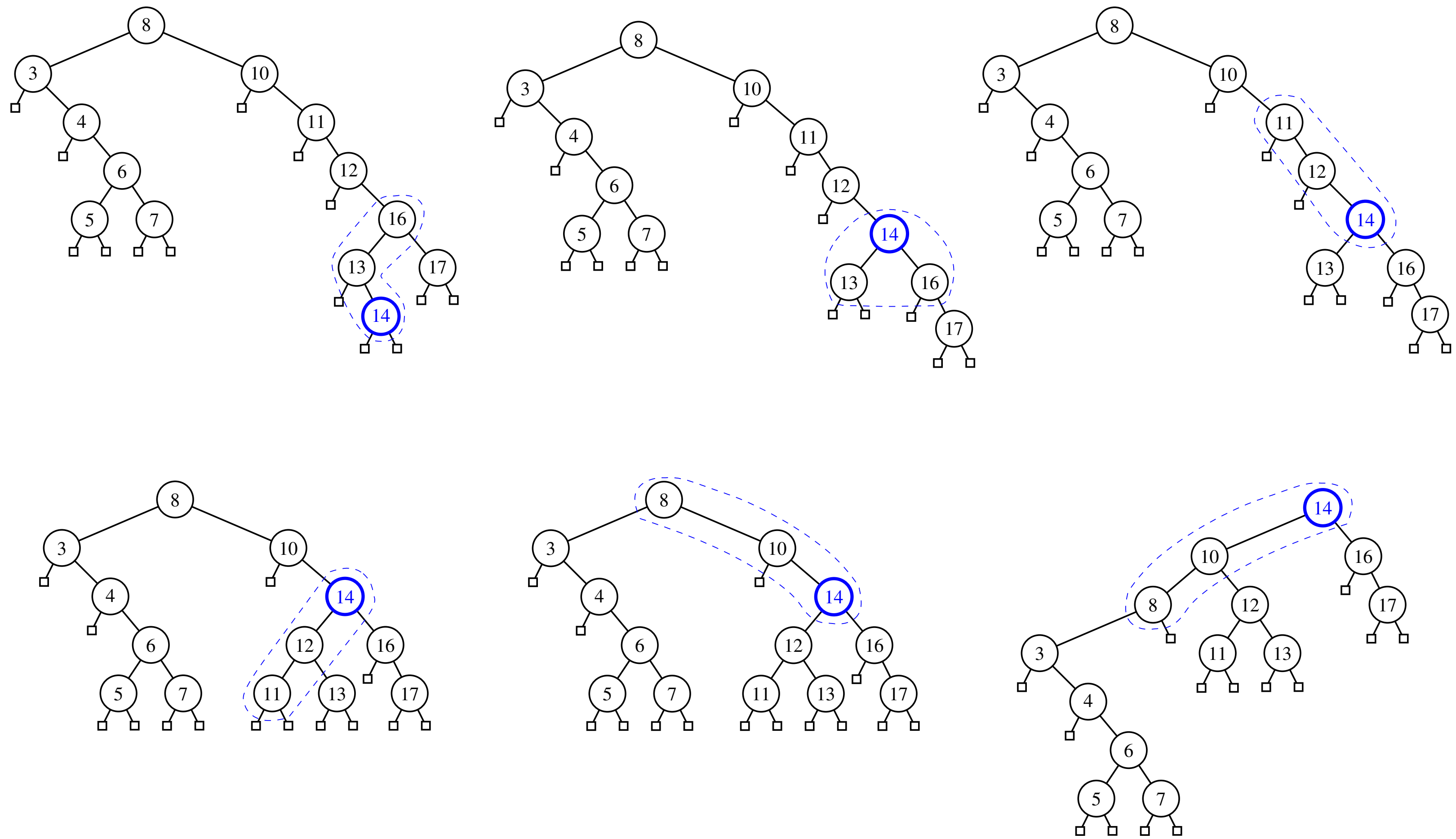
- zig-zag (zag-zig): one right one left



Example – insert 14



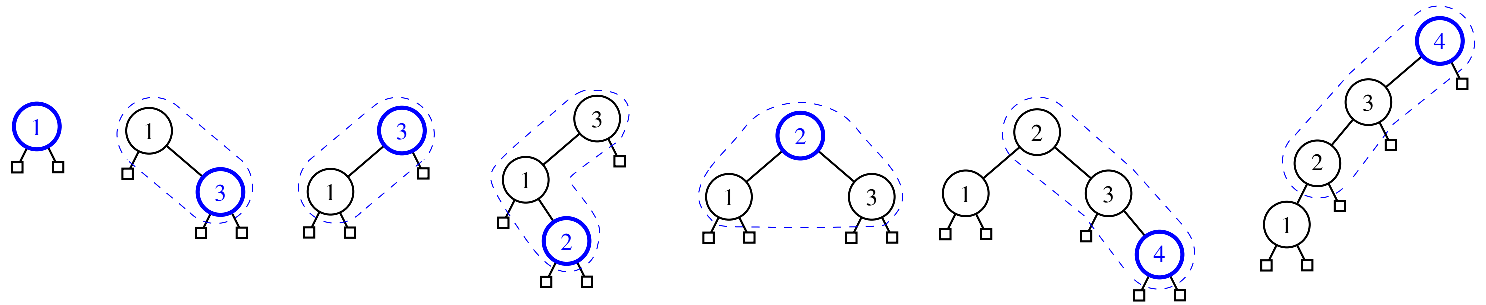
Example



8
3
10
4
11
6
12
5
7
16
13
17
14
8
3
10
4
11
6
12
5
7
14
13
16
17
8
3
10
4
11
6
12
5
7
14
13
16
17
8
3
10
4
14
6
12
16
5
7
11
13
17
8
3
10
4
14
6
12
16
5
7
11
13
17
14
10
16
8
12
17
3
11
13
4
6
5
7

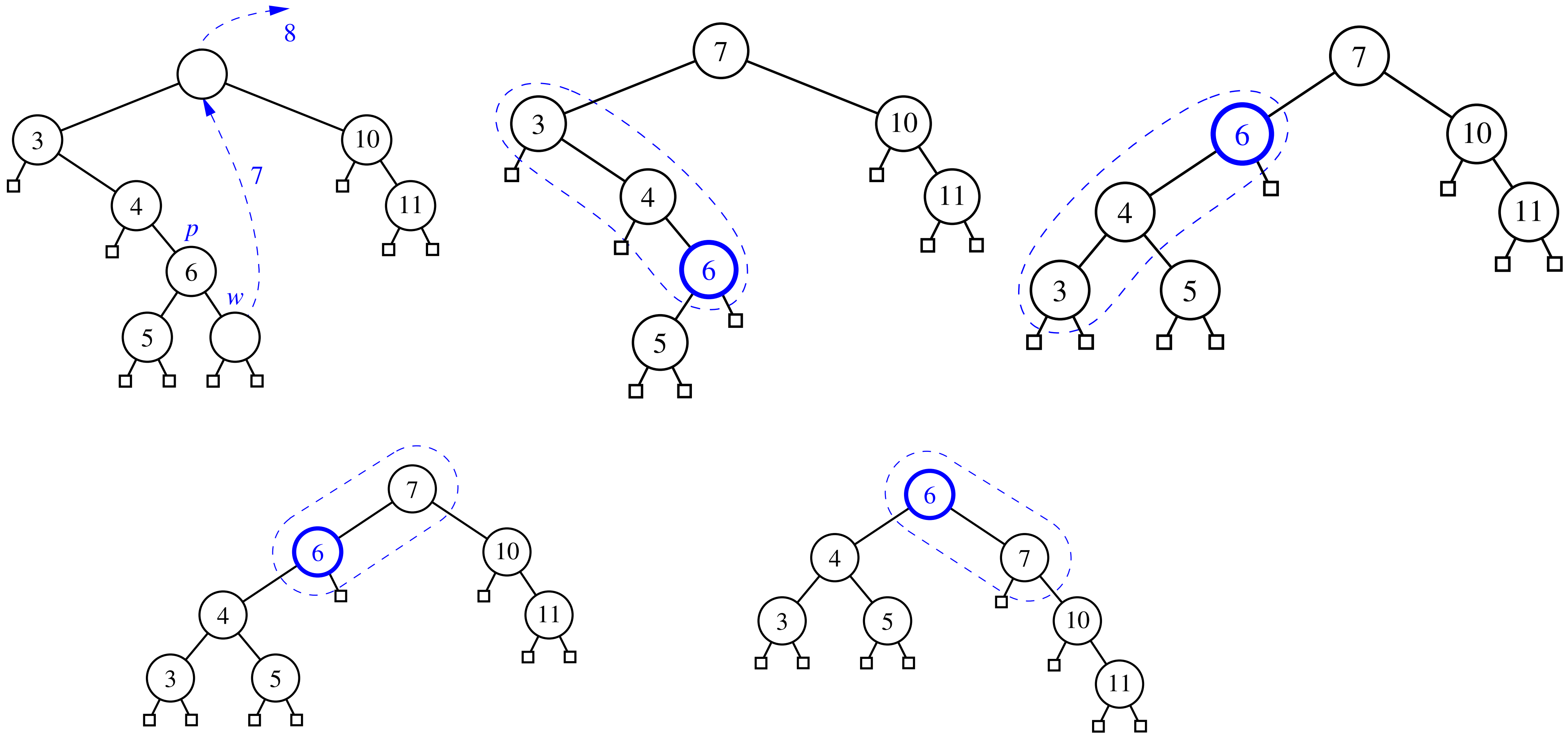
When/what to Splay

- On search for x: if x is found, splay x else splay x's parent
- On insert x: splay x after insertion

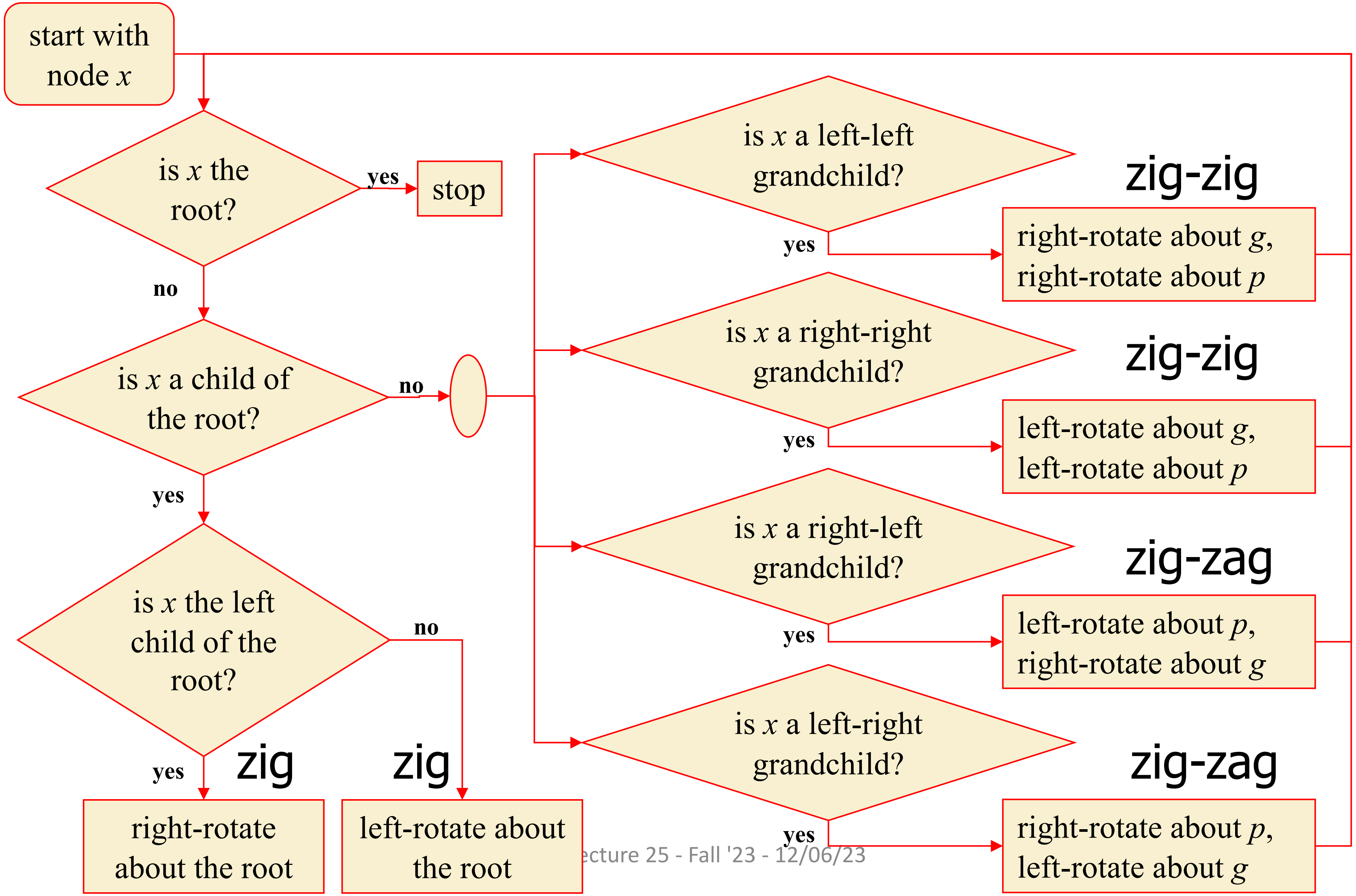


- On delete x: splay parent of removed node
  - x is removed
  - in-order successor/predecessor removed

Deletion



How to Splay



start with node x
is x the root?
yes
stop
no
is x a child of the root?
no
yes
is x the left child of the root?
no
yes
zig
zig
right-rotate about the root
left-rotate about the root
is x a left-left grandchild?
yes
zig-zig
right-rotate about g, right-rotate about p
is x a right-right grandchild?
yes
zig-zig
left-rotate about g, left-rotate about p
is x a right-left grandchild?
yes
zig-zag
left-rotate about p, right-rotate about g
is x a left-right grandchild?
yes
zig-zag
right-rotate about p, left-rotate about g

Analysis of Splaying

- Splay trees do rotations after every operation (even search)
- Runtime of each search/insert/delete is proportional to the time for splaying
- Each zig-zig, zig-zag or zig is $O(1)$
- Splaying a node at height h is $O(h)$
- Worst case height of a splay tree is $O(n)$

Amortized Performance

- A splay tree performs well in amortization – in a sequence of mixed searches, insertions and deletions
- Splay tree performs better for many sequences of non-random operations
- Amortized cost for any splay operation is $O(logn)$
- Must faster search than $O(logn)$ on frequently requested items

AVL Rotations

- AVL insert – $O(logn)$
  - Find the lowest out-of-balance ancestor – also known as the critical node, rotate critical node to balance. Loop ends after single rotation
  - $O(logn)$ search up the tree to find critical node + $O(1)$ rotations
- AVL delete - $O(logn)$
  - $O(logn)$ rotations on delete

Outline

Review Balanced Binary Trees

AVL Trees

Splay Trees

Skip List

Sets

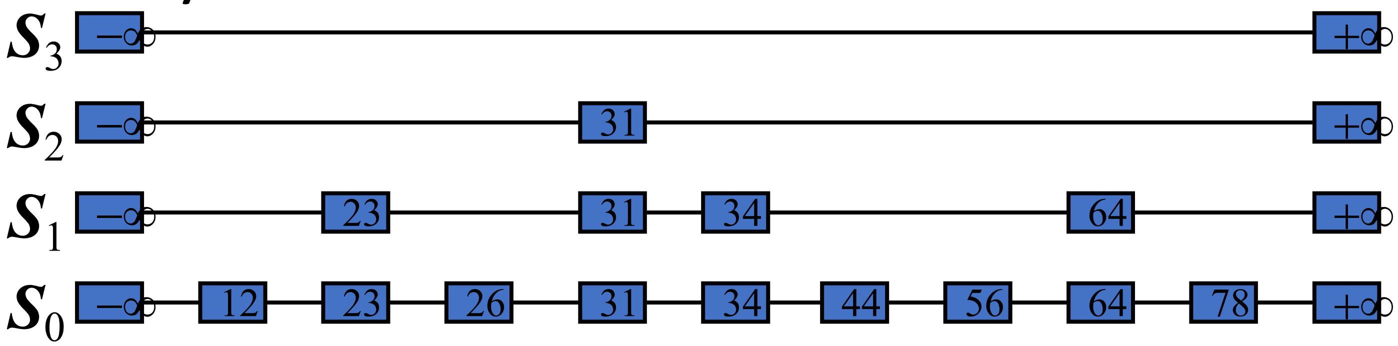
The Problem with lists

- If you must use a linked list
  - because of frequent insertions and deletions
- How do you arrange a fast search?
- What if the list is sorted?
- Still no way to arrange binary search
- `java.util.Concurrent.ConcurrentSkipListMap`

Skip List

A skip list for a set S of (key, value) pairs is a series of lists $S_0, S_1, \ldots, S_h$ such that

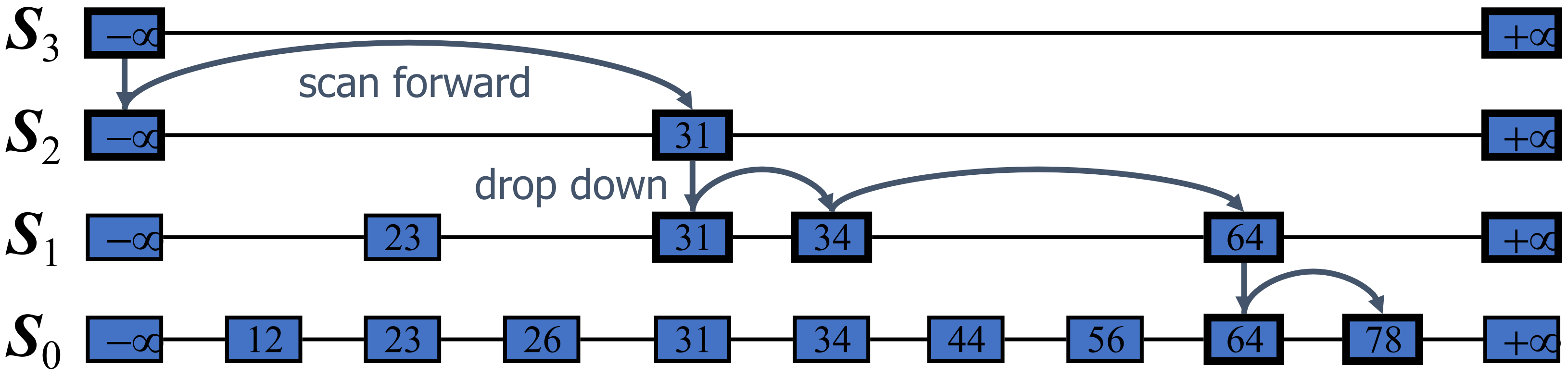
- each list contains special keys -∞ and +∞
- S_0 contains all keys of S in nondecreasing order
- Each list is a subsequence of the one before: $S_0 \supseteq S_1 \supseteq \ \ldots \supseteq S_h$
- S_h only contains the two special keys



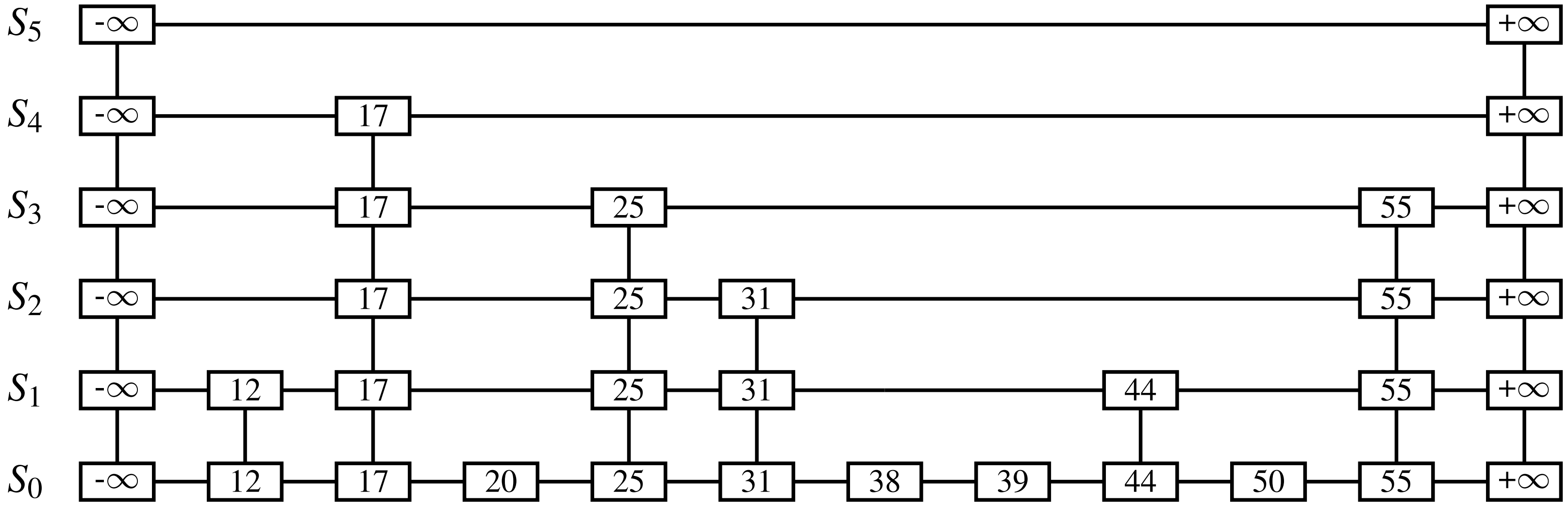
Search

Search for a key x in a skip list:

- start at the first position `p` of S_h
- compare x with `y=next(p)`
  - `x == y` return `y`
  - `x > y`: scan forward `p=next(p)`
  - `x < y`: drop down `p=below(p)`

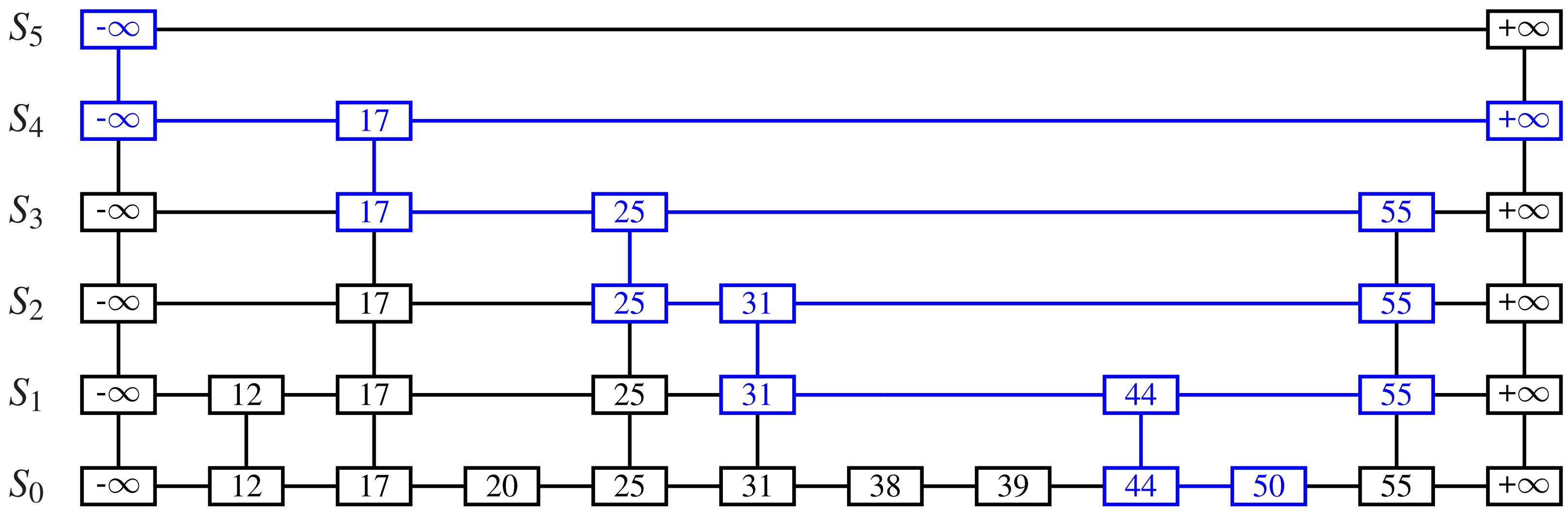


Example



S_5 -∞ +∞
S_4 -∞ 17 +∞
S_3 -∞ 17 25 55 +∞
S_2 -∞ 17 25 31 55 +∞
S_1 -∞ 12 17 25 31 44 55 +∞
S_0 -∞ 12 17 20 25 31 38 39 44 50 55 +∞

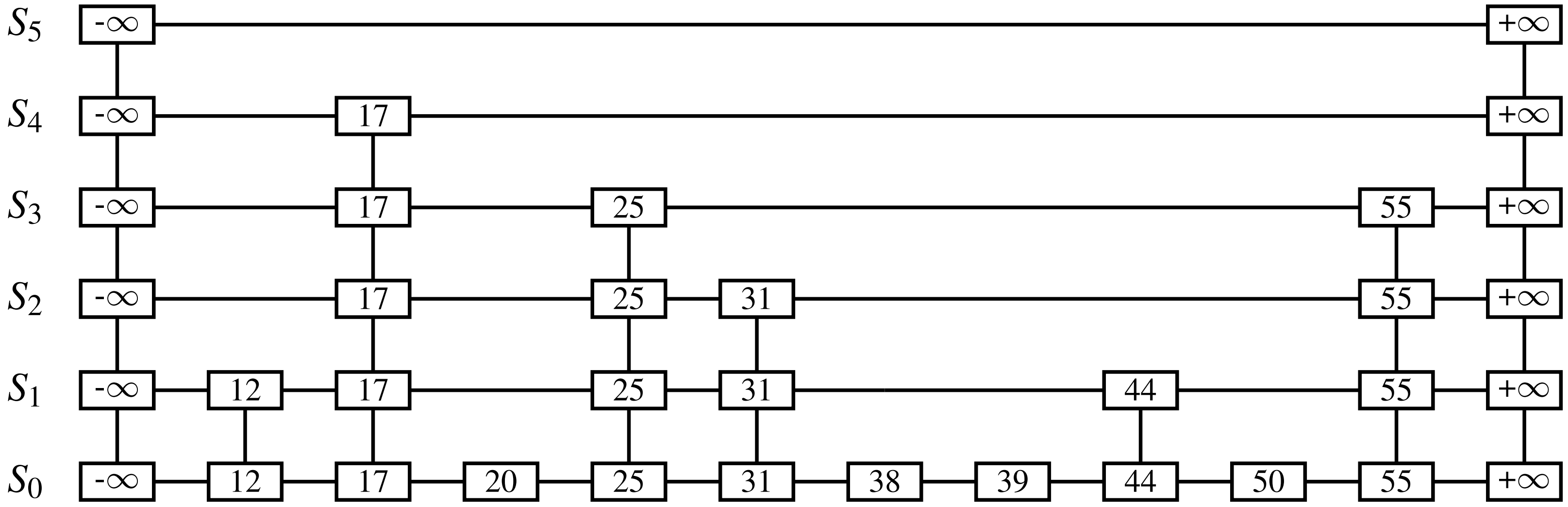
Search for 50



Skip List

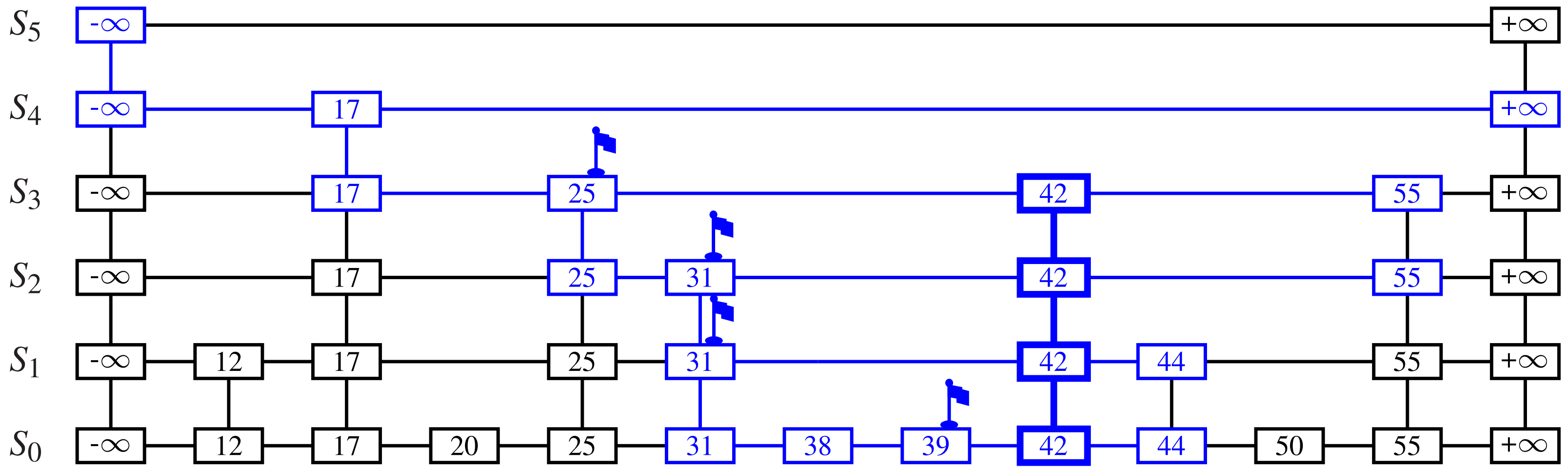
- provides a clever compromise to realize a faster search on a sorted list (map)
- insertion is randomized
  - always insert into s_0
  - flip a coin for how many more lists to insert
  - expected runtime of $O(logn)$
- search is $O(logn)$ expected
- remove via search then up - $O(logn)$ expected

Insert 42



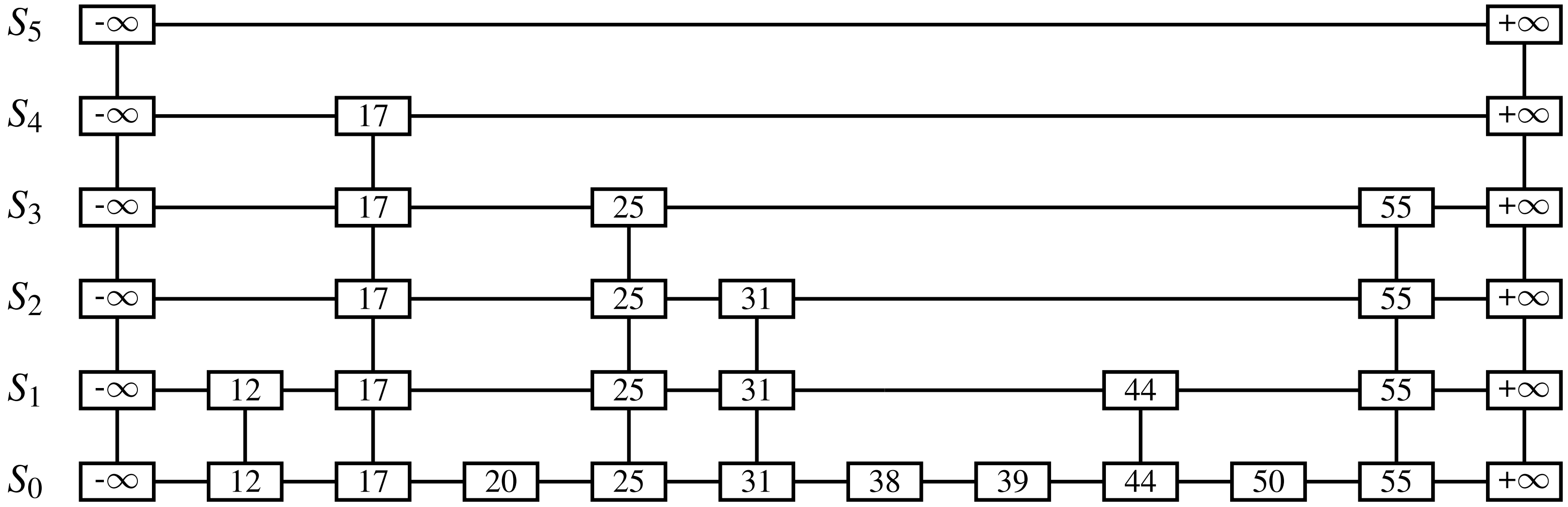
S5
-∞
+∞
S4
-∞
17
+∞
S3
-∞
17
25
55
+∞
S2
-∞
17
25
31
55
+∞
S1
-∞
12
17
25
31
44
55
+∞
S0
-∞
12
17
20
25
31
38
39
44
50
55
+∞

Insert 42



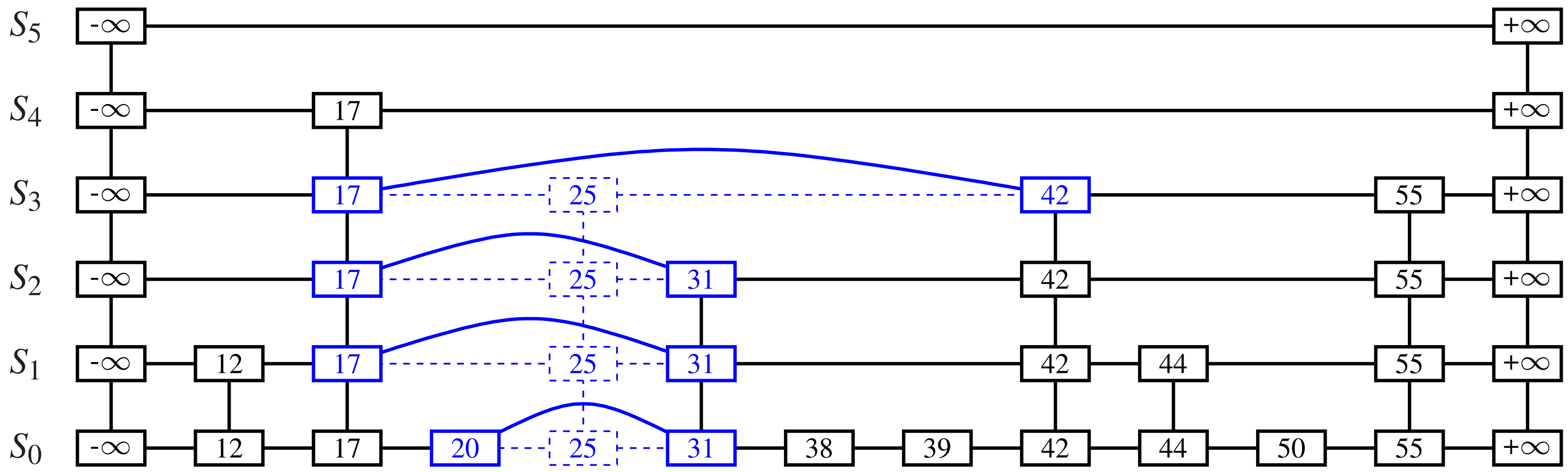
S_5
S_4
S_3
S_2
S_1
S_0
-∞ +∞
-∞ 17 +∞
-∞ 17 25 42 55 +∞
-∞ 17 25 31 42 55 +∞
-∞ 12 17 25 31 42 44 55 +∞
-∞ 12 17 20 25 31 38 39 42 44 50 55 +∞

Remove 25



S_5
-∞
+∞
S_4
-∞
17
+∞
S_3
-∞
17
25
55
+∞
S_2
-∞
17
25
31
55
+∞
S_1
-∞
12
17
25
31
44
55
+∞
S_0
-∞
12
17
20
25
31
38
39
44
50
55
+∞

Remove 25



S_5 -∞ +∞
S_4 -∞ 17 +∞
S_3 -∞ 17 25 42 55 +∞
S_2 -∞ 17 25 31 42 55 +∞
S_1 -∞ 12 17 25 31 42 44 55 +∞
S_0 -∞ 12 17 20 25 31 38 39 42 44 50 55 +∞

Skip List Analysis

Method	Running Time
size, isEmpty	$O(1)$
get	$O(\log n)$ expected
put	$O(\log n)$ expected
remove	$O(\log n)$ expected
firstEntry, lastEntry	$O(1)$
ceilingEntry, floorEntry lowerEntry, higherEntry	$O(\log n)$ expected
subMap	$O(s + \log n)$ expected, with s entries reported
entrySet, keySet, values	$O(n)$

Outline

Review Balanced Binary Trees

AVL Trees

Splay Trees

Skip List

Sets

Set

- A set is an unordered collection of elements, without duplicates
- A set supports an efficient search
- A hashtable is a set
- A multi-set (bag) allows duplicates
- A multi-map allows the same key to be mapped to multiple values

set ADT

add(e): Adds the element e to S (if not already present).

remove(e): Removes the element e from S (if it is present).

contains(e): Returns whether e is an element of S.

iterator( ): Returns an iterator of the elements of S.

There is also support for the traditional mathematical set operations of ***union***, ***intersection***, and ***subtraction*** of two sets S and T:

$$\begin{aligned} S \cup T &= \{e\colon\ e \text{ is in } S \text{ or } e \text{ is in } T\}, \\ S \cap T &= \{e\colon\ e \text{ is in } S \text{ and } e \text{ is in } T\}, \\ S - T &= \{e\colon\ e \text{ is in } S \text{ and } e \text{ is not in } T\}. \end{aligned}$$

addAll(T): Updates S to also include all elements of set T, effectively replacing S by $S \cup T$.

retainAll(T): Updates S so that it only keeps those elements that are also elements of set T, effectively replacing S by $S \cap T$.

removeAll(T): Updates S by removing any of its elements that also occur in set T, effectively replacing S by $S - T$.

Implementation

- Recall that maps do not allow duplicate keys
- A set is simply a map in which keys have no associated values (or null)
- `java.util.HashSet`
- `java.util.Concurrent.ConcurrentSkipListSet`
- `java.util.TreeSet`

Java Built-ins: `java.util.*`

- Linked List
  - `LinkedList`
- Stack
  - `Stack` (linked)
- Queue
  - `ArrayDqueue`
- BST (unbalanced)
  - none
- Heap
  - `PriorityQueue`
- Hashtable
  - `HashMap` (chained)
- Set
  - `HashSet`
- Balanced BST
  - `TreeMap` (R&B)
- Search/Sort
  - `Collections.binarySearch`
  - `Collections.sort`

Framework Diagram