Balanced Search Trees

Lecture 22
CS211 – Summer 2007

Some Search Structures

- Sorted Arrays
  - Advantages
    - Search in O(log n) time (binary search)
  - Disadvantages
    - Need to know size in advance
    - Insertion, deletion O(n) – need to shift elements
- Lists
  - Advantages
    - No need to know size in advance
    - Insertion, deletion O(1) (not counting search time)
  - Disadvantages
    - Search is O(n), even if list is sorted

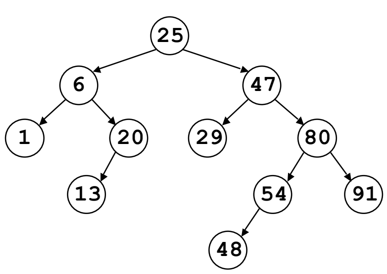
Balanced Search Trees

- Best of both!
  - Search, insert, delete in O(log n) time
  - No need to know size in advance
- Many flavors
  - AVL trees, 2-3 trees, red-black trees, skip lists, random treaps, splay trees...

Review – Binary Search Trees

- Every node has a *left child*, a *right child*, both, or neither
- Data elements are drawn from a totally ordered set (e.g., **`Comparable`**)
- Every node contains one data element
- Data elements are ordered in *inorder*

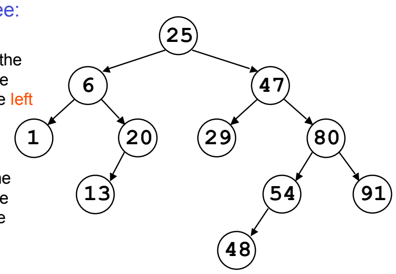
A Binary Search Tree



Binary Search Trees

In any subtree:

- all elements smaller than the element at the root are in the left subtree
- all elements larger than the element at the root are in the right subtree



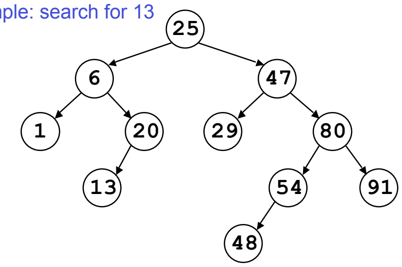
Search

To search for an element x:

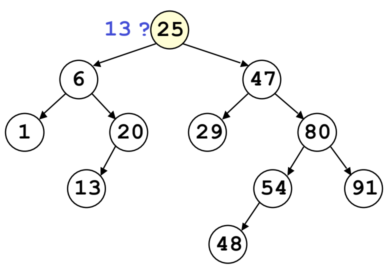
- if tree is empty, return false
- if x = object at root, return true
- If x < object at root, search left subtree
- If x > object at root, search right subtree

Search

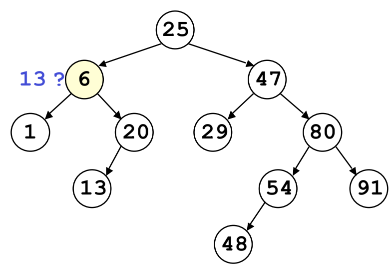
Example: search for 13



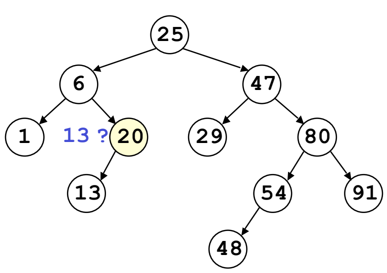
Search



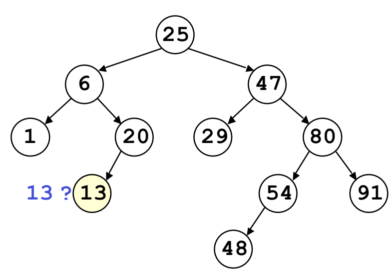
Search



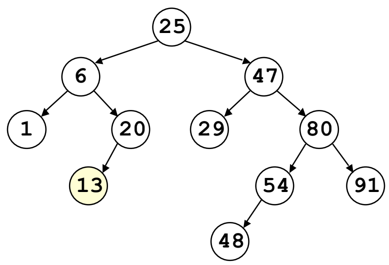
Search



Search



Search



Search

```
boolean treeSearch(Comparable x,
                   TreeNode t) {
  if (t == null) return false;
  switch (x.compareTo(t.data)) {
    case 0: return true; //found
    case 1: return treeSearch(x, t.right);
    default: return treeSearch(x, t.left);
  }
}
```

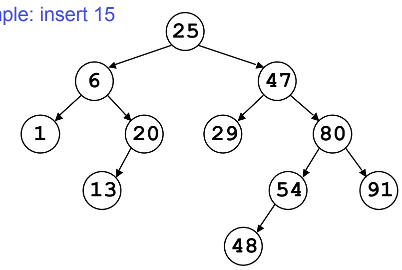
Insertion

To insert an element x:

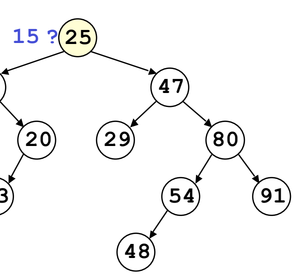
- search for x – if there, just return
- when arrive at a leaf y, make x a child of y
  - left child if x < y
  - right child if x > y

Insertion

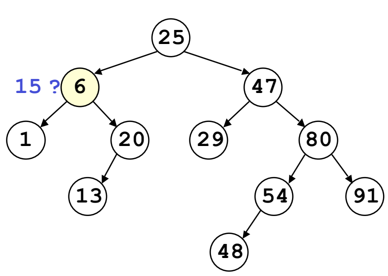
Example: insert 15



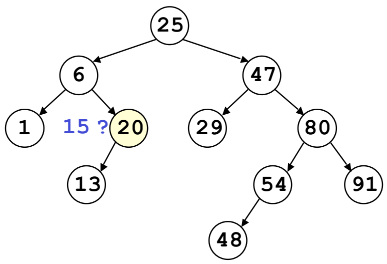
Insertion



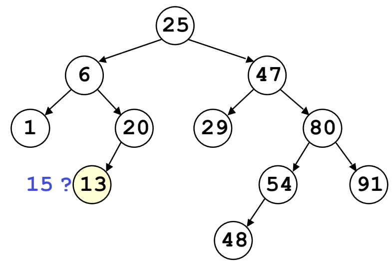
Insertion



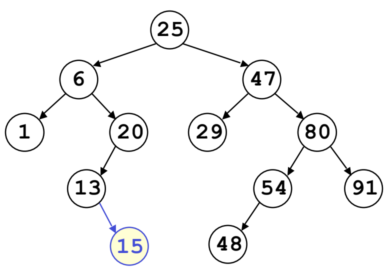
Insertion



Insertion



Insertion



Insertion

```
void insert(Comparable x, TreeNode t) {
  if (x.compareTo(t.data) == 0) return;
  if (x.compareTo(t.data) < 0) {
    if (t.left != null) insert(x,t.left);
    else t.left = new TreeNode(x);
  } else {
    if (t.right != null) insert(x,t.right);
    else t.right = new TreeNode(x);
  }
}
```

Deletion

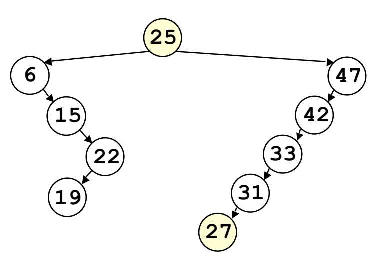
To delete an element x:

- remove x from its node – this creates a hole
- if the node was a leaf, just delete it
- find greatest y less than x in the left subtree (or least y greater than x in the right subtree), move it to x's node
- this creates a hole where y was – repeat

Deletion

To find least y greater than x:

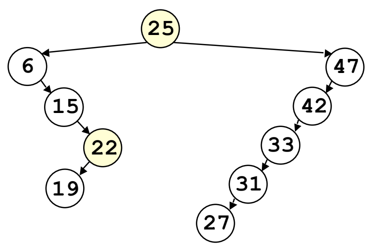
- follow left children as far as possible in right subtree



Deletion

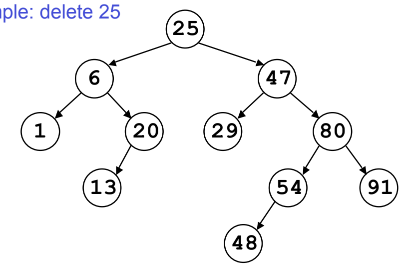
To find greatest y less than x:

- follow right children as far as possible in left subtree

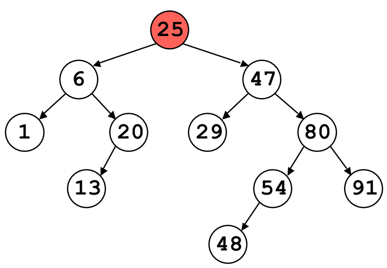


Deletion

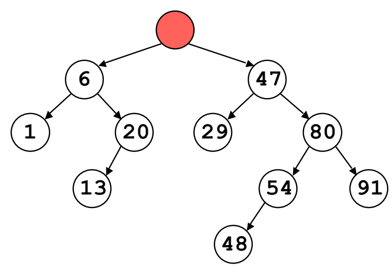
Example: delete 25



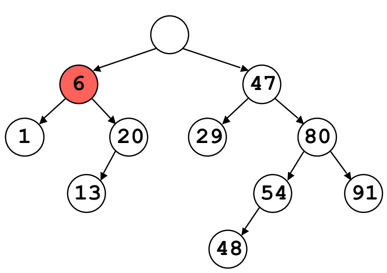
Deletion



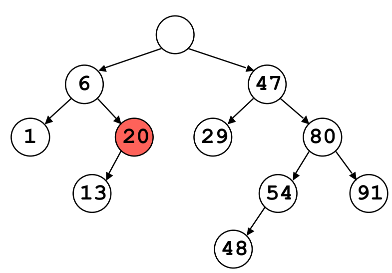
Deletion



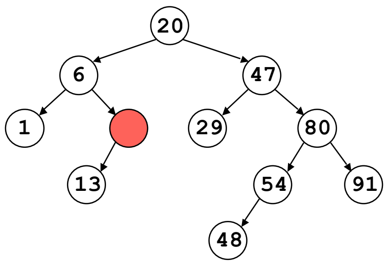
Deletion



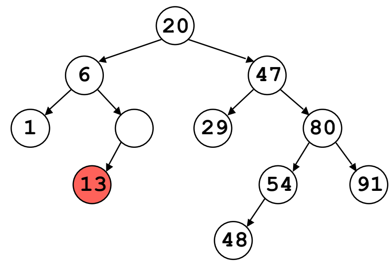
Deletion



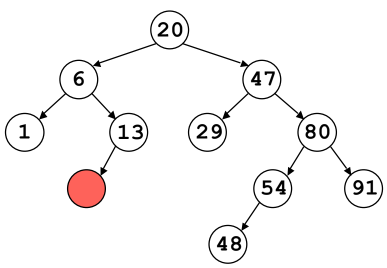
Deletion



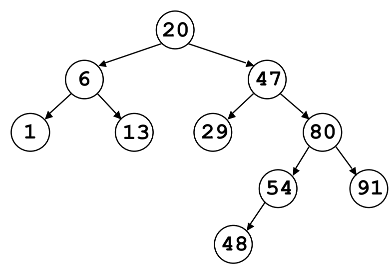
Deletion



Deletion

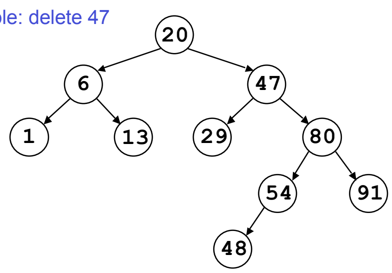


Deletion

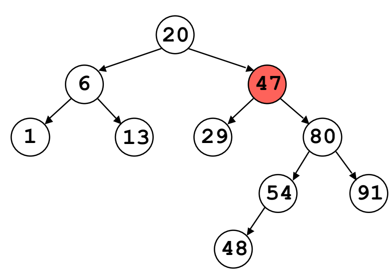


Deletion

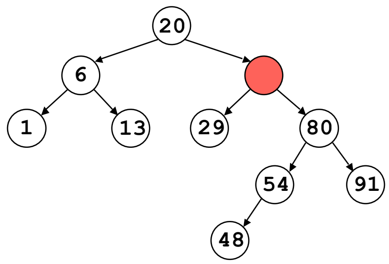
Example: delete 47



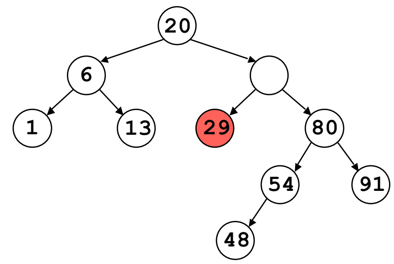
Deletion



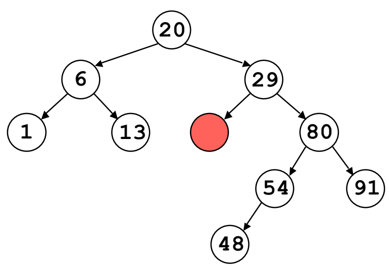
Deletion



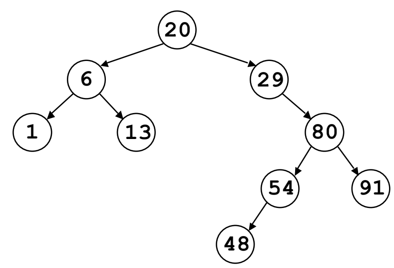
Deletion



Deletion

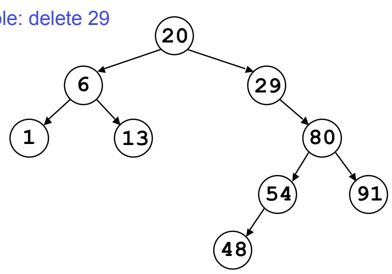


Deletion

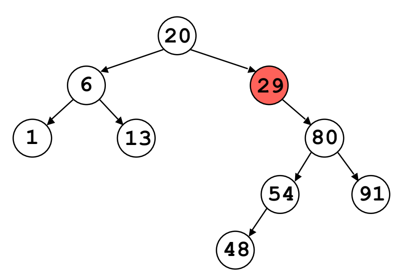


Deletion

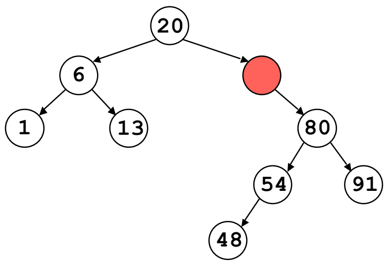
Example: delete 29



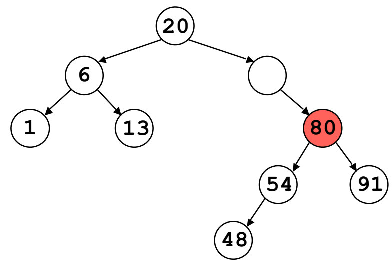
Deletion



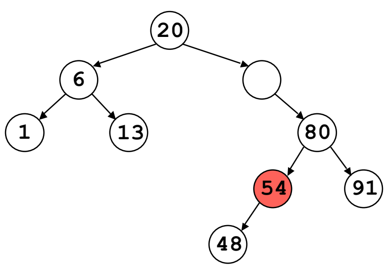
Deletion



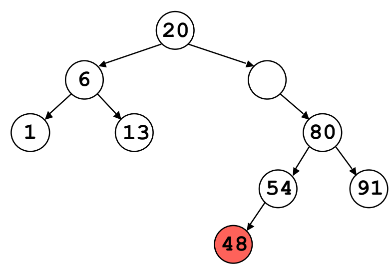
Deletion



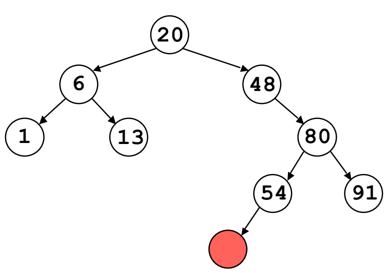
Deletion



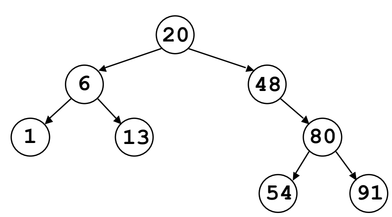
Deletion



Deletion



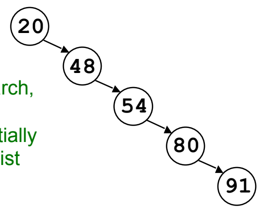
Deletion



Observation

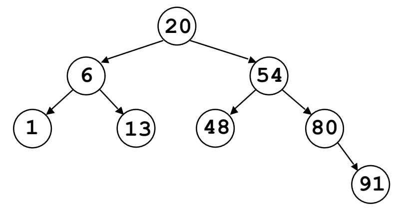
- These operations take time proportional to the height of the tree (length of the longest path)
- O(n) if tree is not sufficiently balanced

Bad case for search, insertion, and deletion – essentially like searching a list



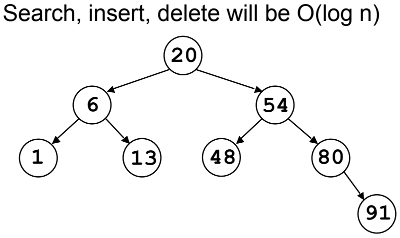
Solution

Try to keep the tree *balanced* (all paths roughly the same length)



Balanced Trees

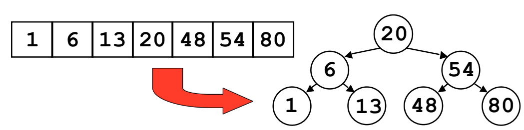
- Size is exponential in height
- Height = $\log_2$(size)
- Search, insert, delete will be O(log n)



Creating a Balanced Tree

Creating one from a sorted array:

- Find the median, place that at the root
- Recursively form the left subtree from the left half of the array and the right subtree from the right half of the array



Keeping the Tree Balanced

- Insertions and deletions can put tree out of balance – we may have to rebalance it
- Can we do this efficiently?

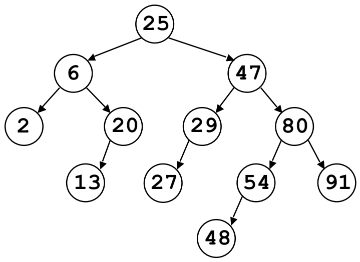
AVL Trees

Adelson-**V**elsky and **L**andis, 1962

AVL Invariant:
The difference in height between the left and right subtrees of any node is never more than one

An AVL Tree

- Nonexistent children are considered to have height -1
- Note that paths can differ in length by more than 1 (e.g., paths to 2, 48)



AVL Trees are Balanced

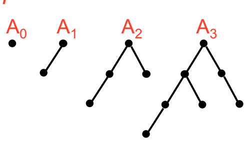
The AVL invariant implies that:

- Size is at least exponential in height
  - $n \geq \varphi^d$, where $\varphi = (1 + \sqrt{5})/2 \sim 1.618$, the golden ratio!
- Height is at most logarithmic in size
  - $d \leq \log n / \log \varphi \sim 1.44 \log n$

AVL Trees are Balanced

AVL Invariant:
The difference in height between the left and right subtrees of any node is never more than one

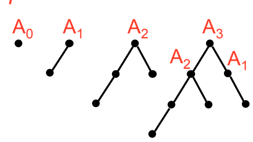
To see that $n \geq \varphi^d$, look at the *smallest possible* AVL trees of each height



AVL Trees are Balanced

AVL Invariant:
The difference in height between the left and right subtrees of any node is never more than one

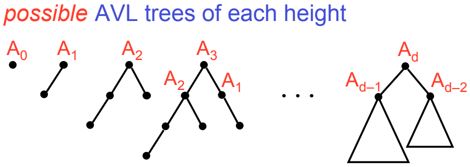
To see that $n \geq \varphi^d$, look at the *smallest possible* AVL trees of each height



AVL Trees are Balanced

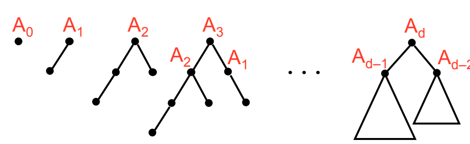
AVL Invariant:
The difference in height between the left and right subtrees of any node is never more than one

To see that $n \geq \varphi^d$, look at the *smallest possible* AVL trees of each height



AVL Trees are Balanced

$A_0 = 1$
$A_1 = 2$
$A_d = A_{d-1} + A_{d-2} + 1, \quad d \geq 2$



AVL Trees are Balanced

$A_0 = 1$
$A_1 = 2$
$A_d = A_{d-1} + A_{d-2} + 1, \quad d \geq 2$

1 2 4 7 12 20 33 54 88 ...

AVL Trees are Balanced

$A_0 = 1$
$A_1 = 2$
$A_d = A_{d-1} + A_{d-2} + 1, \quad d \geq 2$

1 2 4 7 12 20 33 54 88 ...

1 1 2 3 5 8 13 21 34 55 ...
The Fibonacci sequence

AVL Trees are Balanced

$A_0 = 1$
$A_1 = 2$
$A_d = A_{d-1} + A_{d-2} + 1, \quad d \geq 2$

1 2 4 7 12 20 33 54 88 ...

1 1 2 3 5 8 13 21 34 55 ...

$A_d \;=\; F_{d+2} - 1 \;=\; O(\varphi^d)$

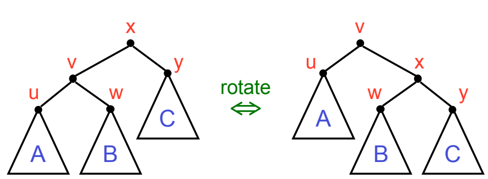
Rebalancing

- Insertion and deletion can invalidate the AVL invariant
- May have to *rebalance*

Rebalancing

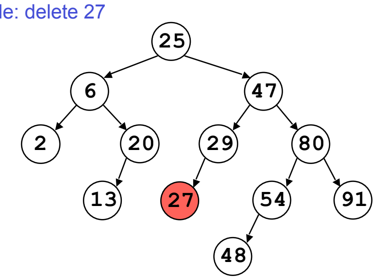
Rotation

- A local rebalancing operation
- Preserves inorder ordering of the elements
- The AVL invariant can be reestablished with at most O(log n) rotations up and down the tree

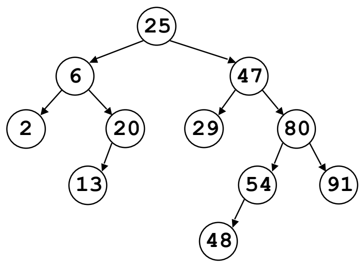


Rebalancing

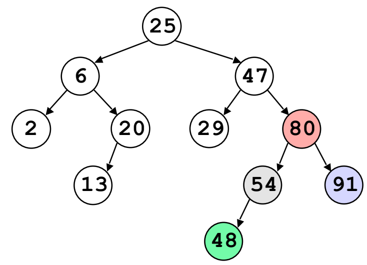
Example: delete 27



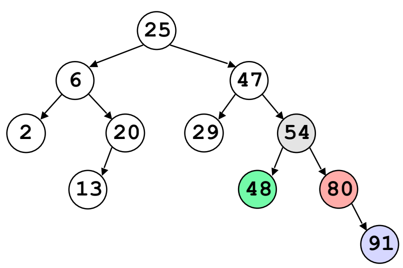
Rebalancing



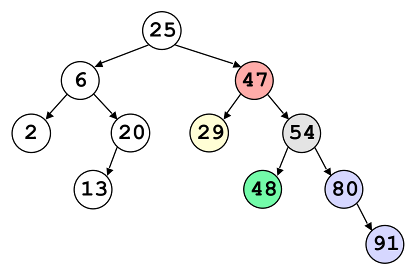
Rebalancing



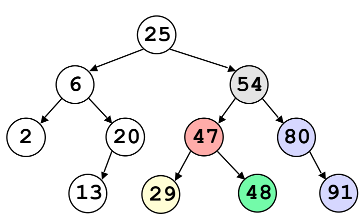
Rebalancing



Rebalancing



Rebalancing

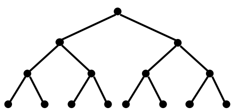


2-3 Trees

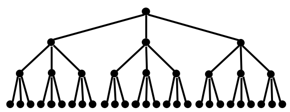
Another balanced tree scheme

- Data stored only at the leaves
- Ordered left-to-right
- All paths of the same length
- Every non-leaf has either 2 or 3 children
- Each internal node has smallest, largest element in its subtree (for searching)

2-3 Trees



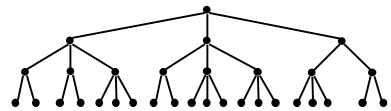
smallest 2-3 tree of height d = 3
2^d = 8 data elements



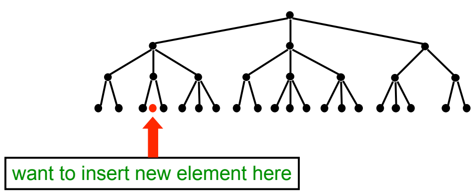
largest 2-3 tree of height d = 3
3^d = 27 data elements

- number of elements satisfies $2^d \le n \le 3^d$
- height satisfies $d \le \log n$

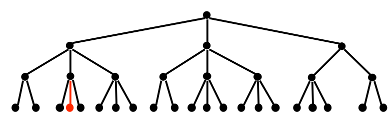
Insertion in 2-3 Trees



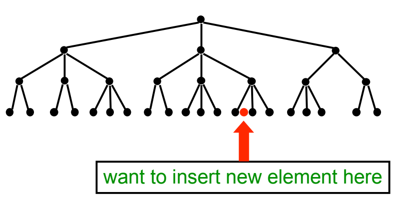
Insertion in 2-3 Trees



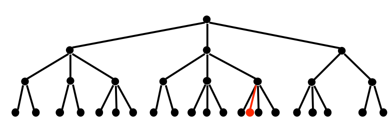
Insertion in 2-3 Trees



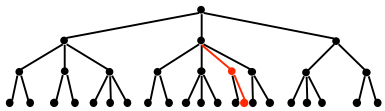
Insertion in 2-3 Trees



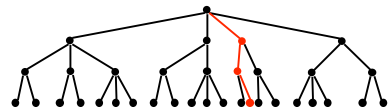
Insertion in 2-3 Trees



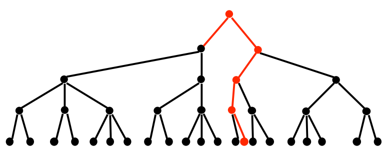
Insertion in 2-3 Trees



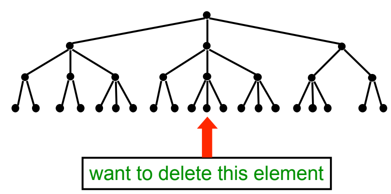
Insertion in 2-3 Trees



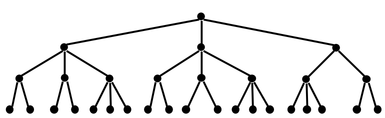
Insertion in 2-3 Trees



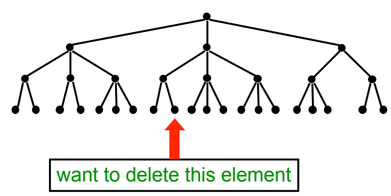
Deletion in 2-3 Trees



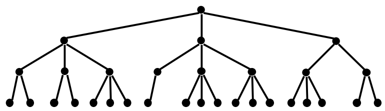
Deletion in 2-3 Trees



Deletion in 2-3 Trees

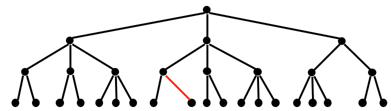


Deletion in 2-3 Trees



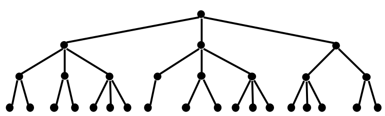
If neighbor has 3 children, borrow one

Deletion in 2-3 Trees



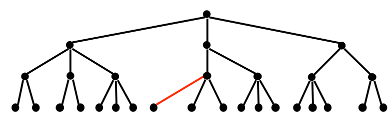
If neighbor has 3 children, borrow one

Deletion in 2-3 Trees



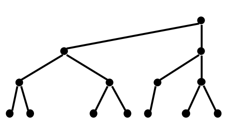
If neighbor has 2 children, coalesce with neighbor

Deletion in 2-3 Trees



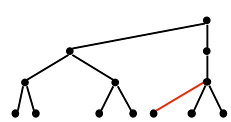
If neighbor has 2 children, coalesce with neighbor

Deletion in 2-3 Trees



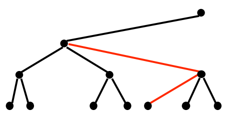
This may cascade up the tree!

Deletion in 2-3 Trees



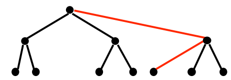
This may cascade up the tree!

Deletion in 2-3 Trees



This may cascade up the tree!

Deletion in 2-3 Trees



This may cascade up the tree!

Splay trees

- Whenever a search() is carried out, the target node is splayed to the root of the tree
  - The splays promote balance
  - Frequently-accessed nodes rise towards the top of the tree
  - Splay trees good choice when searches are biased
- Splay trees are easier to implement than AVL and 2-3 trees, but asymptotically just as fast

Conclusion

Balanced search trees are good

- Search, insert, delete in O(log n) time
- No need to know size in advance
- Several different versions
  - AVL trees, 2-3 trees, red-black trees, skip lists, random treaps, Huffman trees, ...
  - find out more about them in CS482