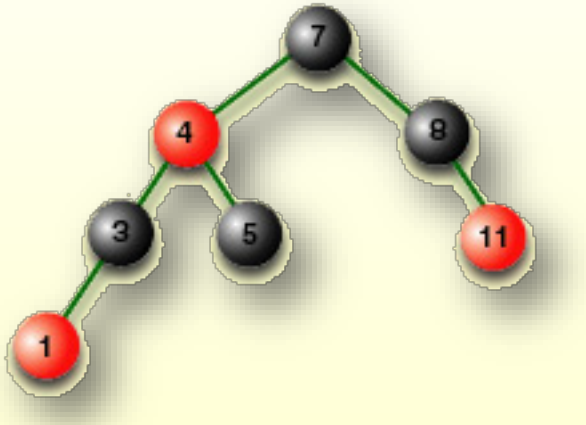
CS161:
Design and Analysis of Algorithms



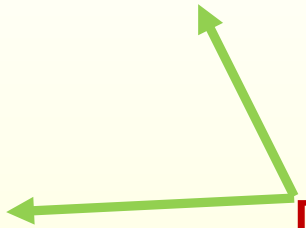
Lecture 9

Leonidas Guibas

(Guest lecturer: Mary Wootters)

Outline

- Review of last lecture: Hashing
- Binary Search Trees
  - Traversals
  - Search/Insertion/Deletion
  - TreeSort
  - Expected height of a tree

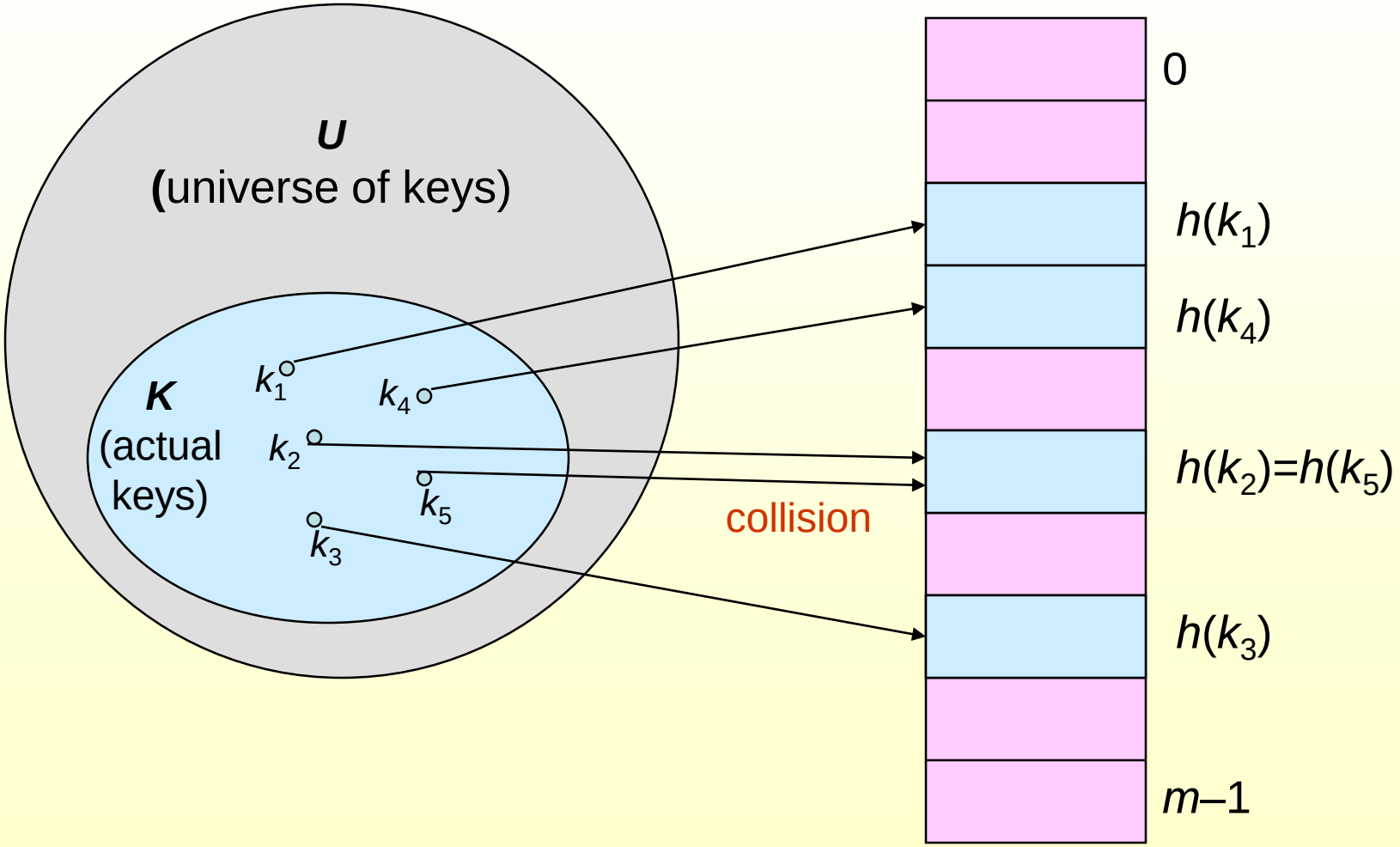


Dictionary Data Structures

Slides modified from

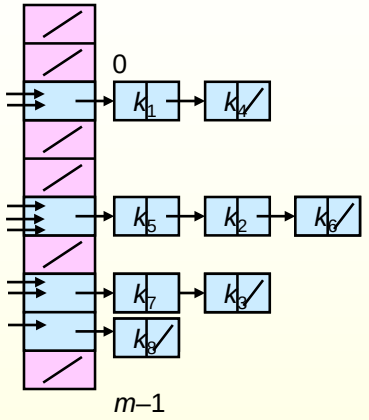
- www.cse.unr.edu/~bebis/CS477/
- homes.ieu.edu.tr/cevrendilek/CE221_week_10_Chapter4_TreesBST.ppt
- http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j

Hashing

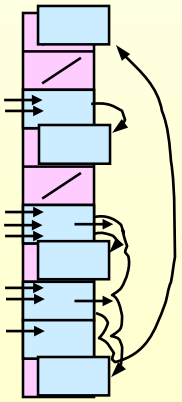

U
(universe of keys)
K
(actual
keys)
k_1
k_4
k_2
k_5
k_3
collision
0
$h(k_1)$
$h(k_4)$
$h(k_2)=h(k_5)$
$h(k_3)$
$m-1$

Methods of Collision Resolution

- Chaining:
  - Store all elements that hash to the same slot in a linked list.
  - Store a pointer to the head of the linked list in the hash table slot.



- Open Addressing:
  - All elements stored in hash table itself.
  - When collisions occur, use a systematic (consistent) procedure to store (and search for) elements in free slots of the table.



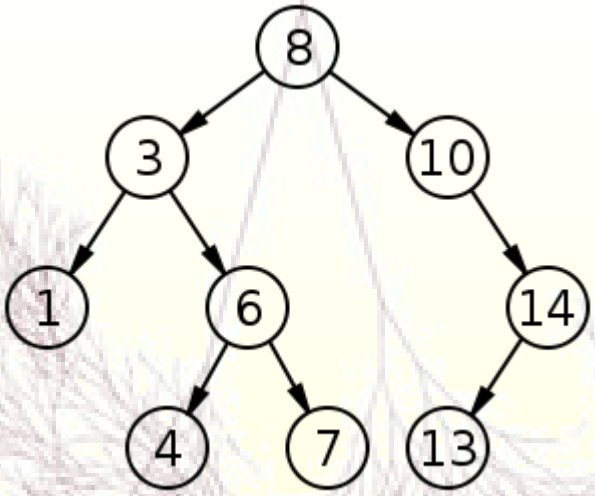
Choosing A Hash Function

- Choosing the hash function well is crucial
  - Bad hash function puts all elements in same slot
  - A good hash function:
    - Should distribute keys uniformly into slots
    - Should not depend on patterns in the data
- Several ways to do this!

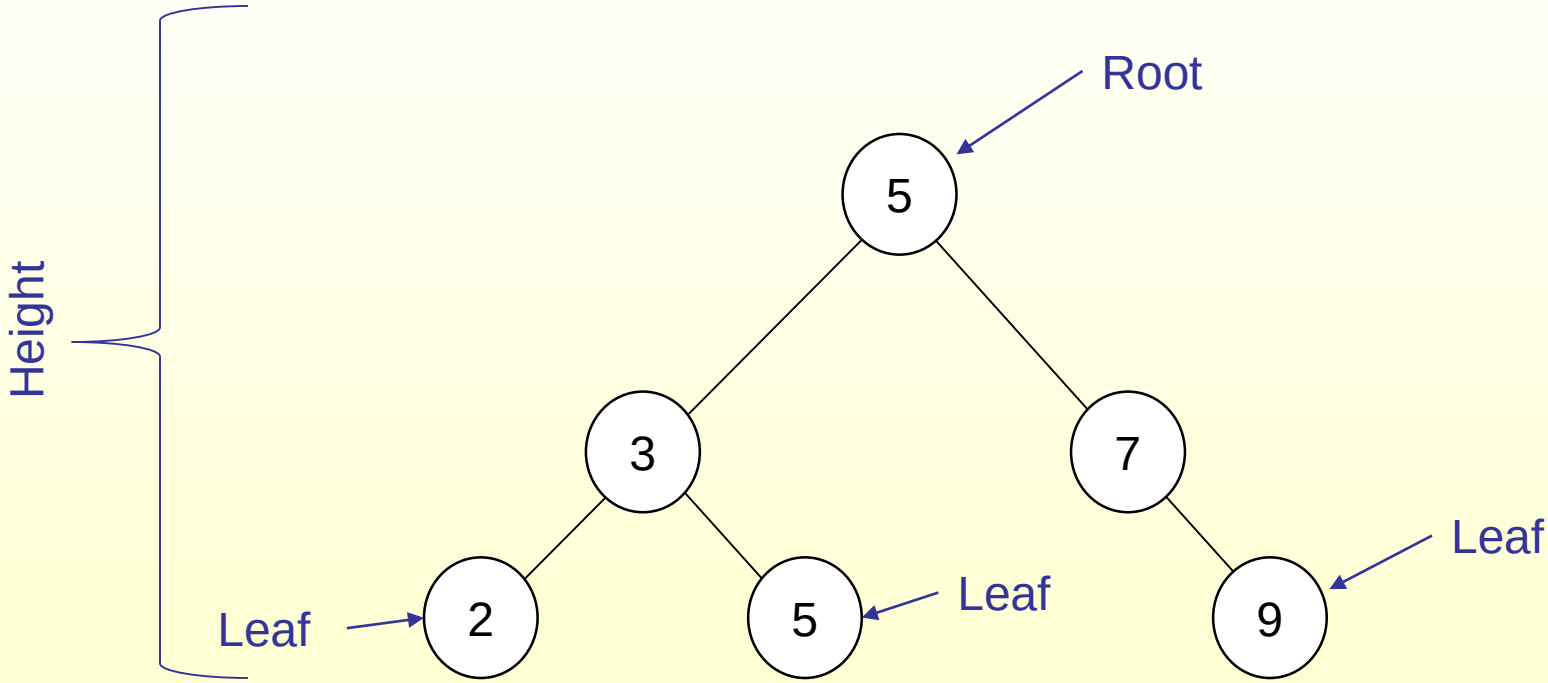
Motivation for today

- Hash tables:
  - Support many operations [SEARCH, INSERT, DELETE, etc…] in expected time O(1).
  - But if we pick a bad hash function it may be much worse!
- Binary Search Trees:
  - Support [SEARCH, INSERT, DELETE, etc…]
  - Eventual goal: O(lg(n)) for all of them.
  - We won't quite get there today, but we'll get a good start.

BINARY SEARCH TREES

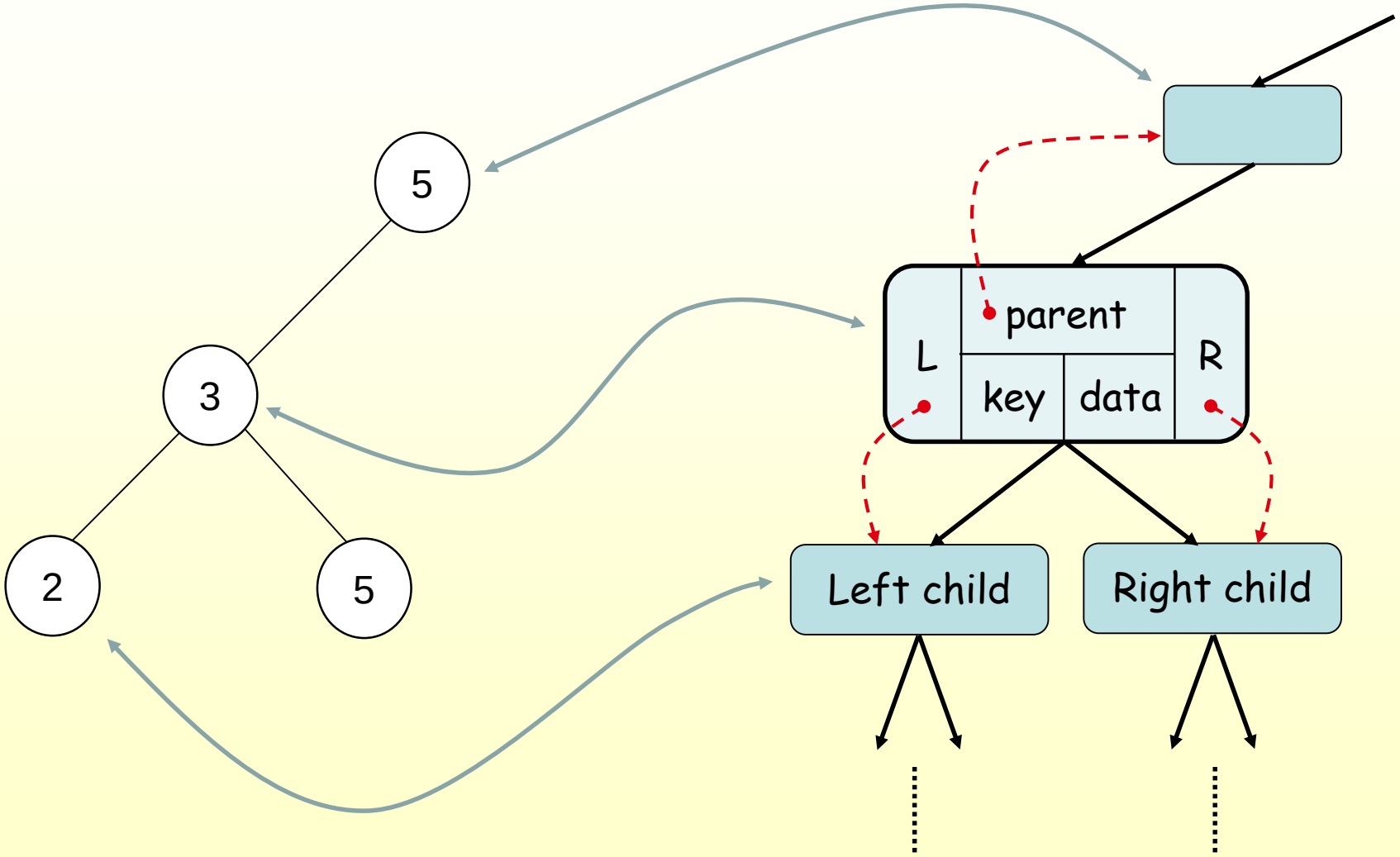


Binary Tree Terminology



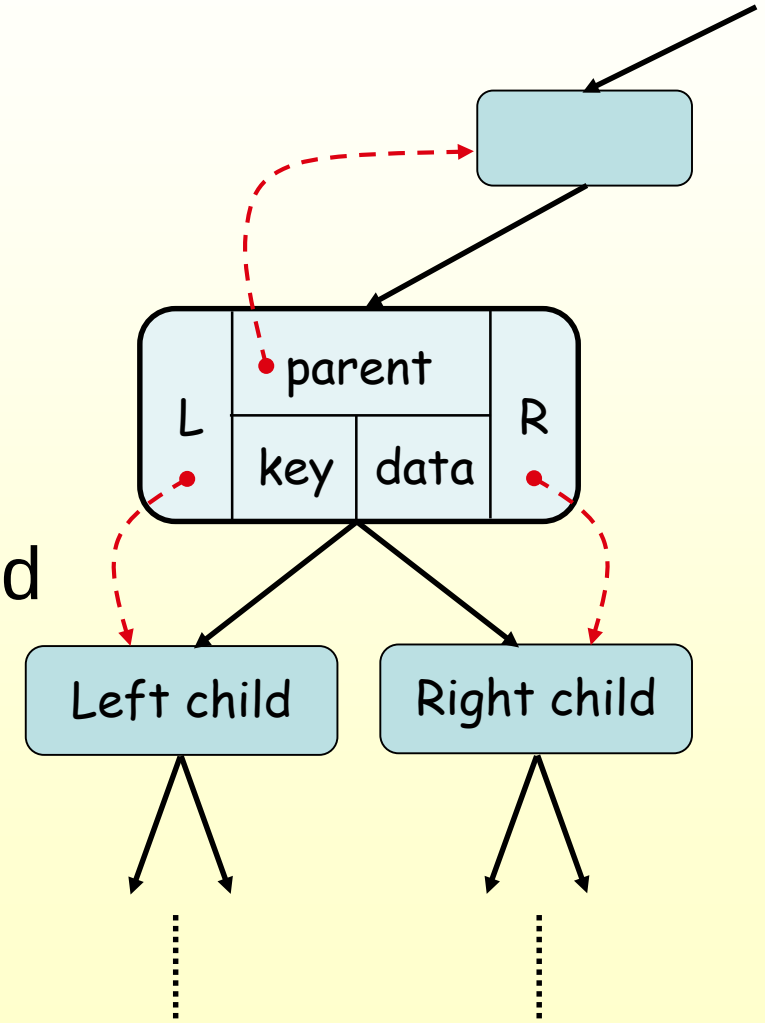
- 3 is the left child of 5
- 7 is the right child of 5
- 5 is the parent of 3

Binary Tree Implementation



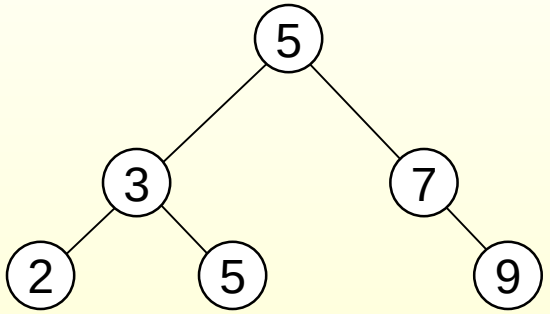
Binary Tree Implementation

- Node representation:
  - Key field
  - Satellite data
  - Left: pointer to left child
  - Right: pointer to right child
  - p: pointer to parent
    - (p [root [T]] = NIL)

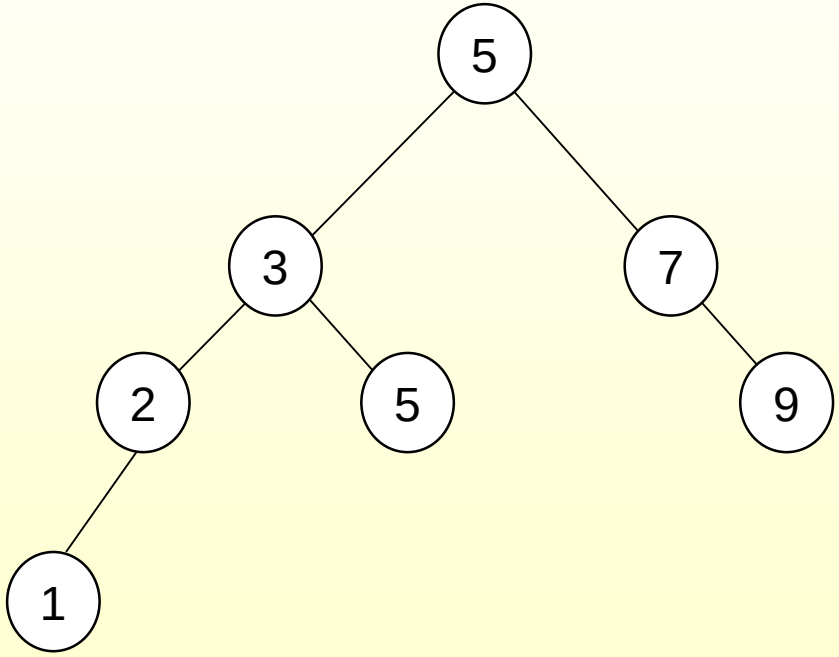


Binary Search Trees

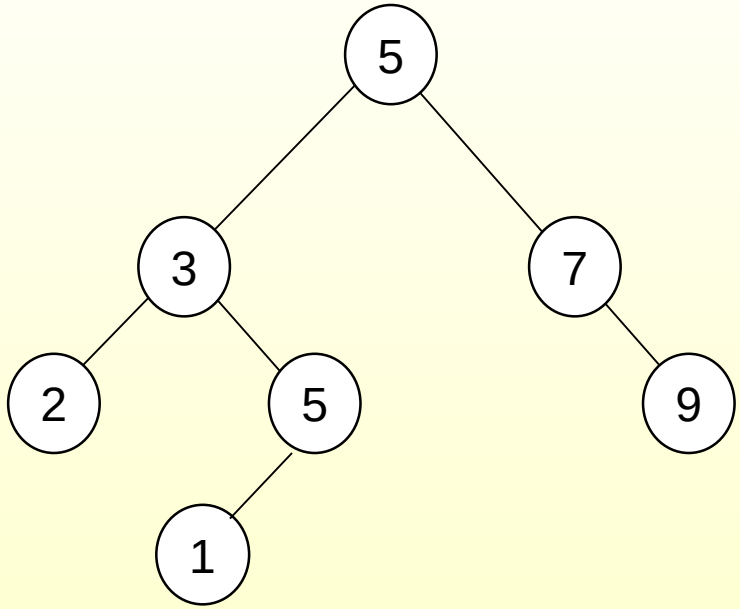
- A Binary Search Tree is a binary tree that satisfies the **Binary search tree property**:
  - If y is in left subtree of x,
    then $key[y] \leq key[x]$
  - If y is in right subtree of x,
    then $key[y] \geq key[x]$



Example and non-example



Binary Search Tree



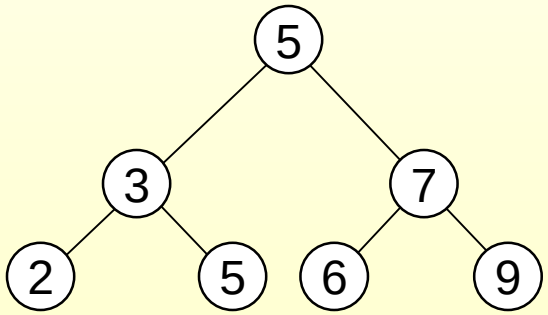
NOT a Binary Search Tree

Binary Search Trees

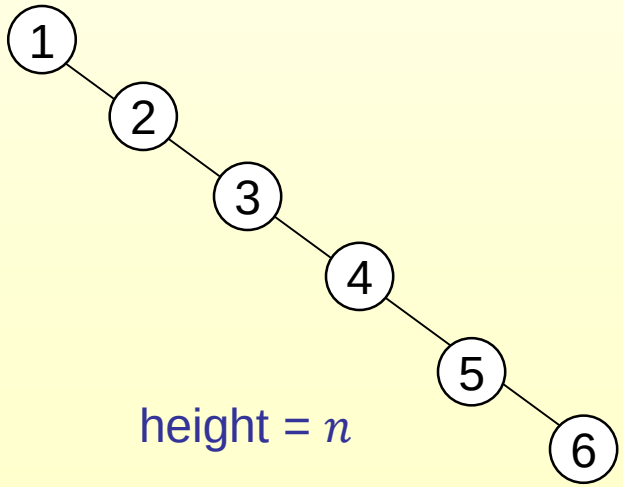
- Support **many** dynamic set operations:
  - SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, DELETE
- In particular, a Binary Search Tree (BST) can implement both the Dictionary and the Priority Queue abstract data types

Binary Search Trees

- Running time of basic operations on BSTs:
  - Depends on the height of the tree.
- What is the height of a BST on n nodes?



height = $\Theta(\log(n))$



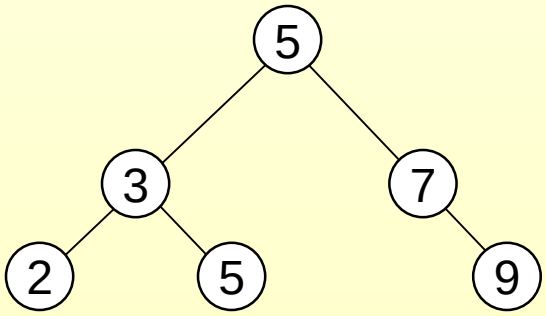
height = n

How to get around a BST?

TRAVERSALS

In-Order Traversal

- When you are at a node x:
  - Recursively explore x's left child
  - Visit x
  - Recursively explore x's right child



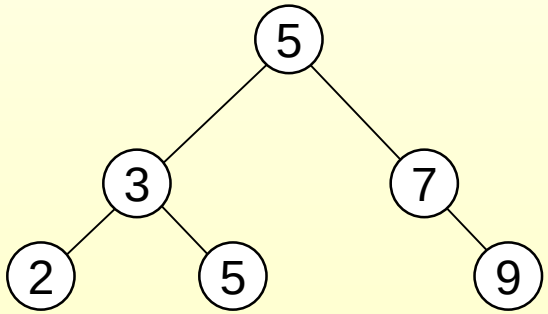
- 2
- 3
- 5
- 5
- 7
- 9

In-Order Traversal

Alg: INORDER-TREE-WALK(x)

1. **if** x ≠ NIL
2. **then** INORDER-TREE-WALK (left [x])
3. visit/print key [x]
4. INORDER-TREE-WALK (right [x])

- *E.g.:*

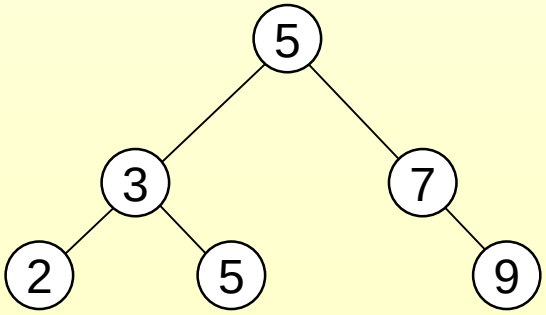


Output: 2 3 5 5 7 9

- Running time:
  - Θ(n), where n is the size of the tree rooted at x

Traversing a BST

- **Inorder** tree walk (traversal):
  - Root is visited between the visits of its left and right subtrees: left, root, right
  - Keys are visited in sorted order
- **Preorder** tree walk:
  - root visited first: root, left, right
- **Postorder** tree walk:
  - root visited last: left, right, root



Inorder: 2 3 5 5 7 9

Preorder: 5 3 2 5 7 9

Postorder: 2 5 3 9 7 5

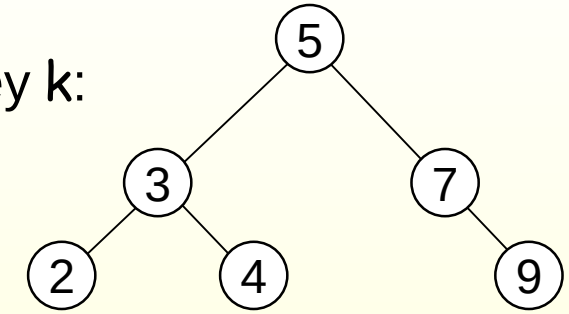
Basic operations on BSTs

SEARCH, MAX, MIN, SUCCESSOR, PREDECESSOR, INSERT, DELETE

Searching for a Key

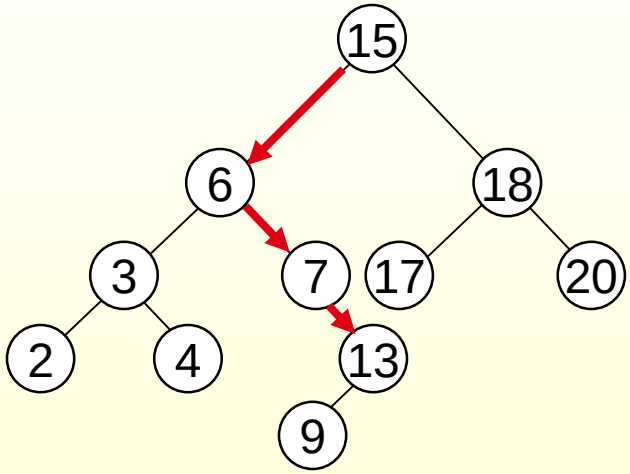
- Goal:
  - Given a pointer to the root of a tree and a key k:
  - Return a pointer to a node with key k

    (if one exists)
  - Otherwise return NIL



- Idea:
  - Starting at the root: trace down a path by comparing k with the key of the current node:
    - If $k = key[x]$, we have found the key
    - If $k < key[x]$, search in the left subtree of x
    - If $k > key[x]$, search in the right subtree of x

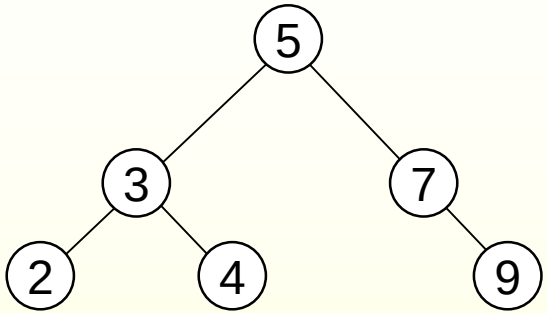
Example: TREE-SEARCH



- Search for key 13:
  - $15 \rightarrow 6 \rightarrow 7 \rightarrow 13$

Searching for a Key

Alg: TREE-SEARCH(x, k)



1. **if** x = NIL or k = key [x]
2. **then return** x
3. **if** k < key [x]
4. **then return** TREE-SEARCH(left [x], k)
5. **else return** TREE-SEARCH(right [x], k)

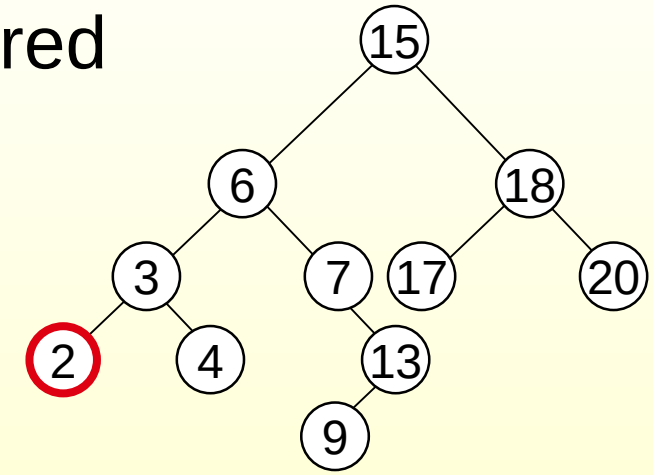
Running Time: *O* (h),
h – the height of the tree

Find the Minimum in a BST

- Idea:
  - Follow left child pointers from the root, until a NIL is encountered

Alg: TREE-MINIMUM(x)

1. **while** left [x] ≠ NIL
2. **do** x ← left [x]
3. **return** x



Minimum = 2

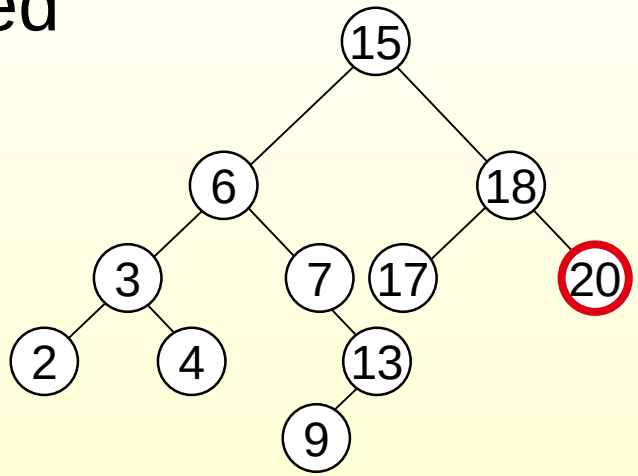
Running time: $O(h)$, h = height of tree

Find the Maximum in a BST

- Idea:
  - Follow right child pointers from the root, until a NIL is encountered

Alg: TREE-MAXIMUM(*x*)

1. **while** right [x] ≠ NIL
2. **do** x ← right [x]
3. **return** x



Maximum = 20

Running time: $O(h)$, h = height of tree

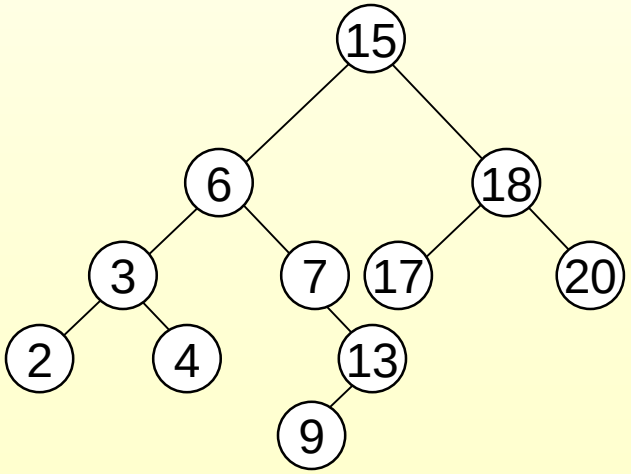
Successor

- *Def:* *successor* (x) = y, such that key [y] is the smallest key with key [y] > key [x]

- *E.g.:* *successor* $(15) = 17$

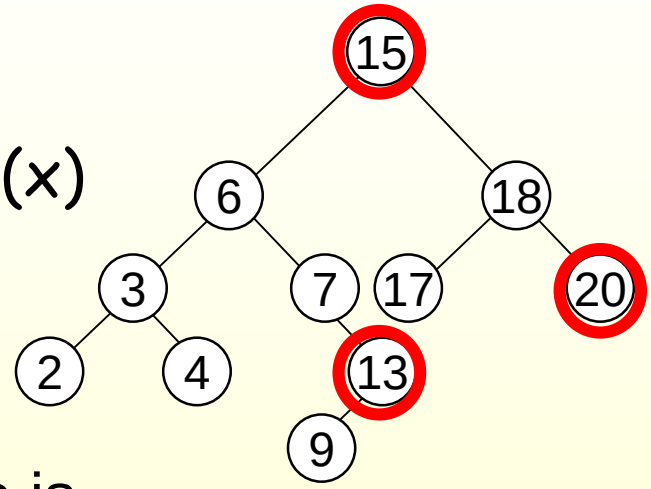
  successor $(13) = 15$

  successor $(9) = 13$



Finding a Successor

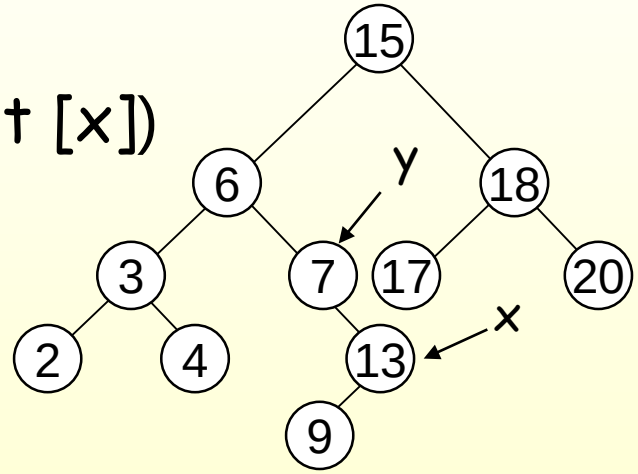
- Case 1: right (x) is non empty
  - Example: *successor(15) = 17.*
  - *successor* (x) = the minimum in right (x)
- Case 2: right (x) is empty
  - *Example: successor(13) = 15.*
  - go up the tree until the current node is a left child: *successor* (x) is the parent of the current node
  - *Example: successor(20) = NIL*
  - if you cannot go further (and you reached the root): x is the largest element



Finding a Successor

Alg: TREE-SUCCESSOR(*x*)

1. **if** right [x] ≠ NIL
2. **then return** TREE-MINIMUM(right [x])
3. y ← p[x]
4. **while** y ≠ NIL and x = right [y]
5. **do** x ← y
6. y ← p[y]
7. **return** y

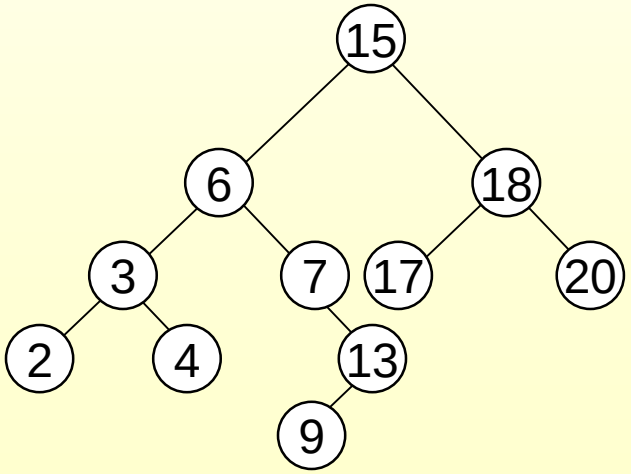


Running time: *O* (h), h = height of the tree

Predecessor

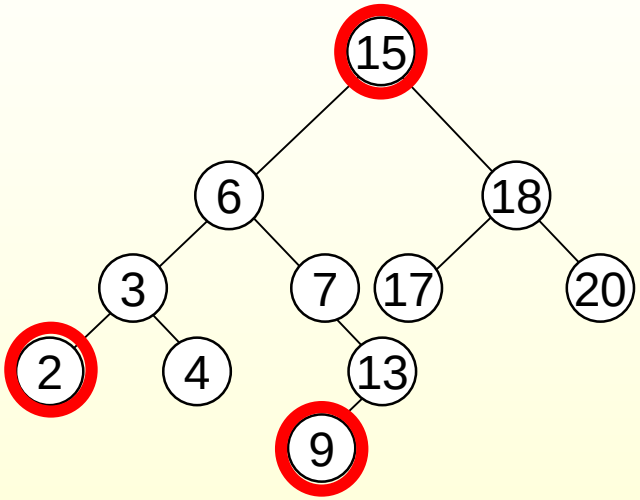
- *Def:* *predecessor* (x) = y, such that key [y] is the largest key with key [y] < key [x]

- *E.g.:* *predecessor (15) = 13*
  predecessor (9) = 7
  predecessor (7) = 6



Finding a Predecessor

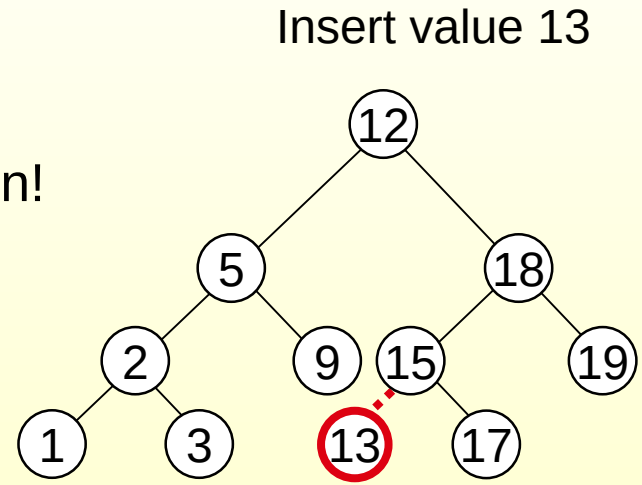
- Case 1: `left (x)` is non empty
  - *Example: predecessor(15) = 13*
  - *predecessor* (x) = maximum in `left (x)`
- Case 2: `left (x)` is empty
  - *Example: predecessor(9) = 7*
  - go up the tree until the current node is a right child: *predecessor* (x) is the parent of the current node
  - *Example: predecessor(2) = NIL*
  - if you cannot go further (and you reached the root): `x` is the smallest element



Insertion

- Goal:
  - Insert value v into a binary search tree
- Idea:
  - If key [x] < v move to the right child of x,

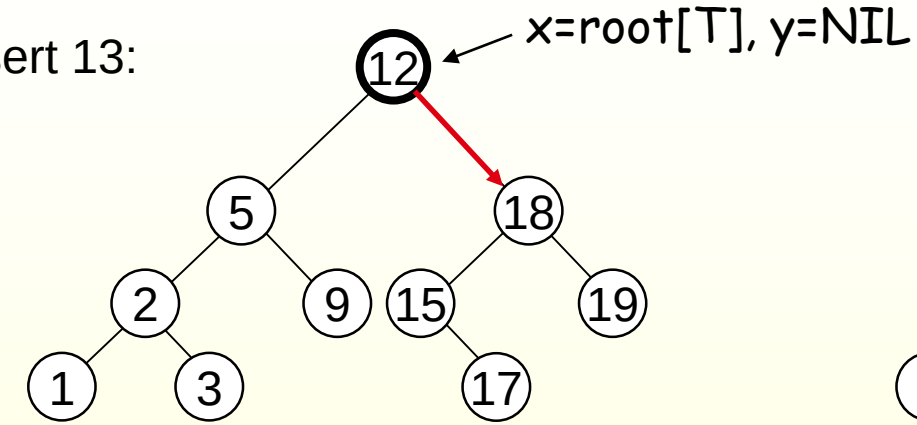
    else move to the left child of x
  - When x is NIL, we found the correct position!

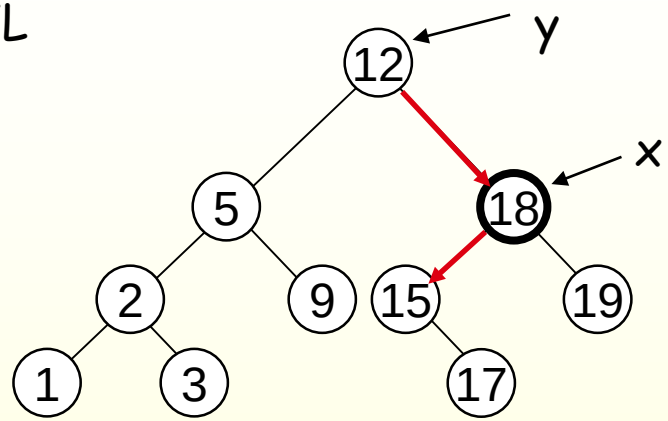


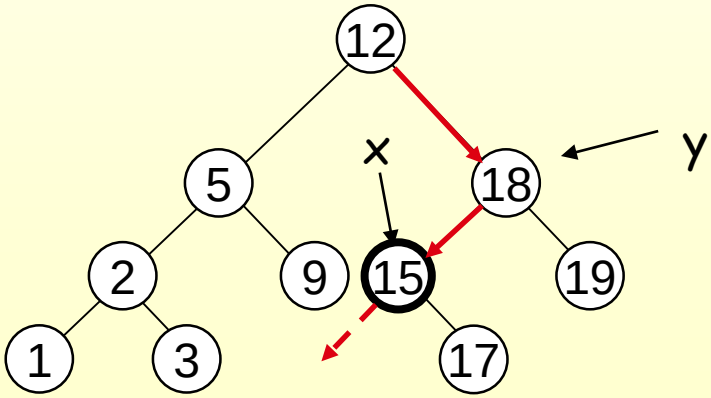
  - Beginning at the root, go down the tree and maintain:
    - Pointer x : current node
    - Pointer y : parent of x (“trailing pointer”)

Example: TREE-INSERT

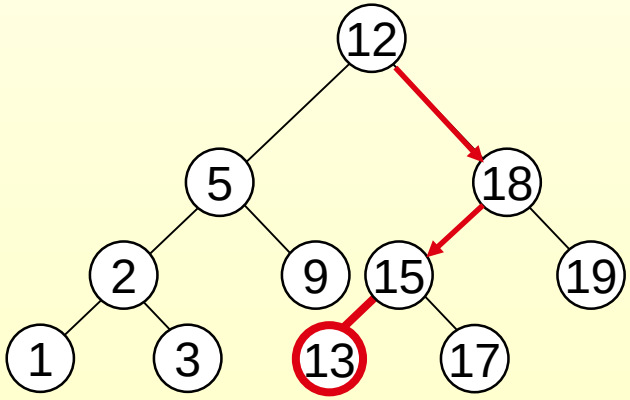
Insert 13:





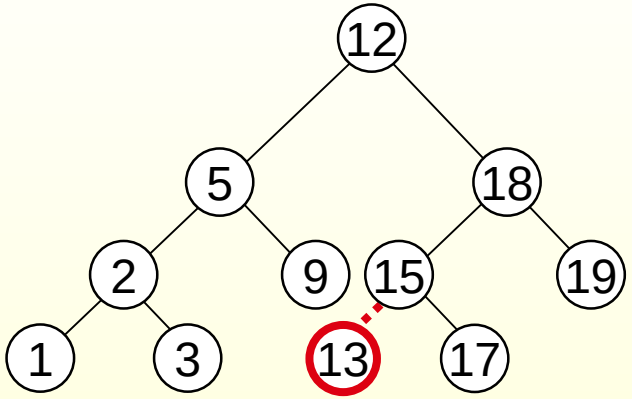


x = NIL
y = 15



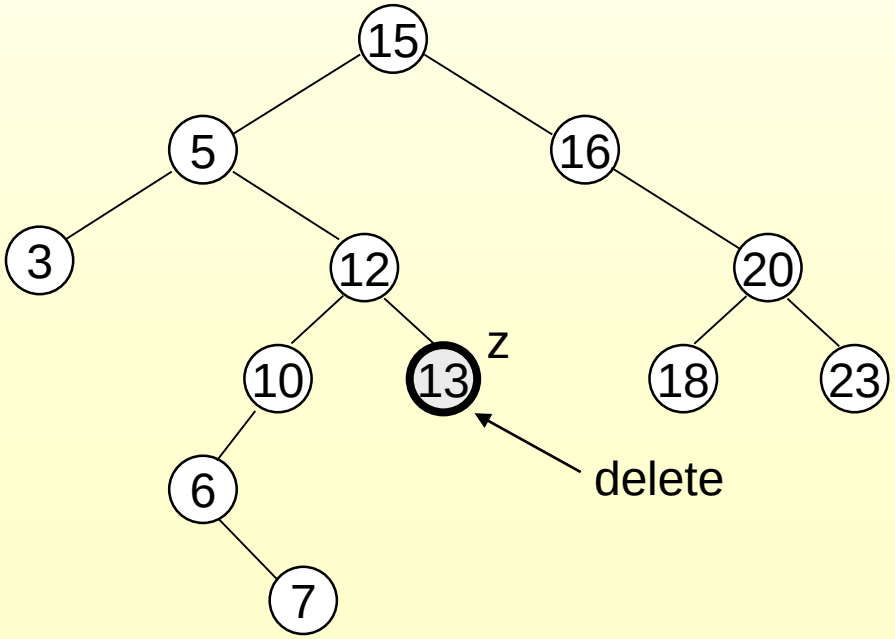
Alg: TREE-INSERT(T, z)

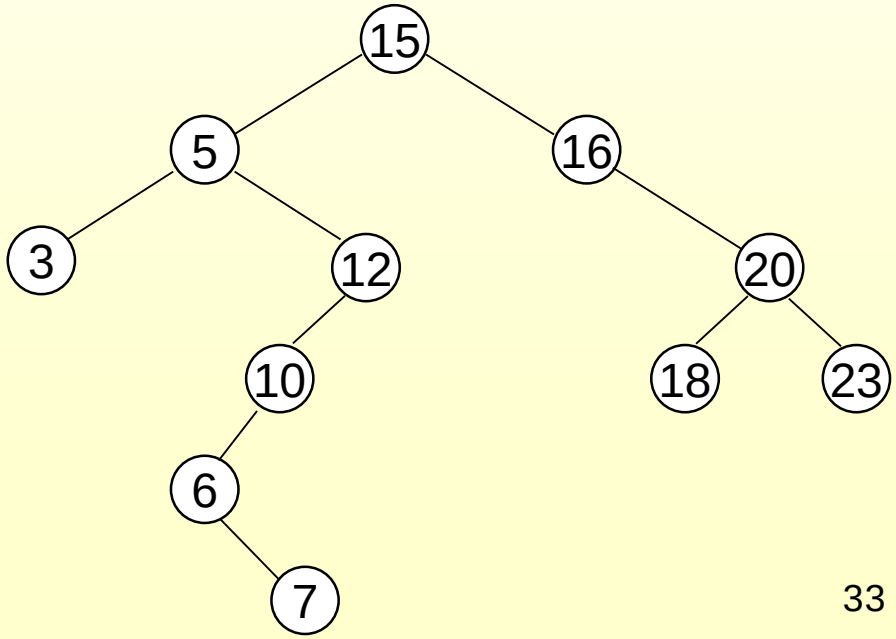
```
1.  y ← NIL
2.  x ← root [T]
3.  while x ≠ NIL
4.      do y ← x
5.          if key [z] < key [x]
6.              then x ← left [x]
7.              else x ← right [x]
8.  p[z] ← y
9.  if y = NIL
10.     then root [T] ← z                ▷ Tree T was empty
11.     else if key [z] < key [y]
12.             then left [y] ← z
13.             else right [y] ← z       Running time: O(h)
```



Deletion

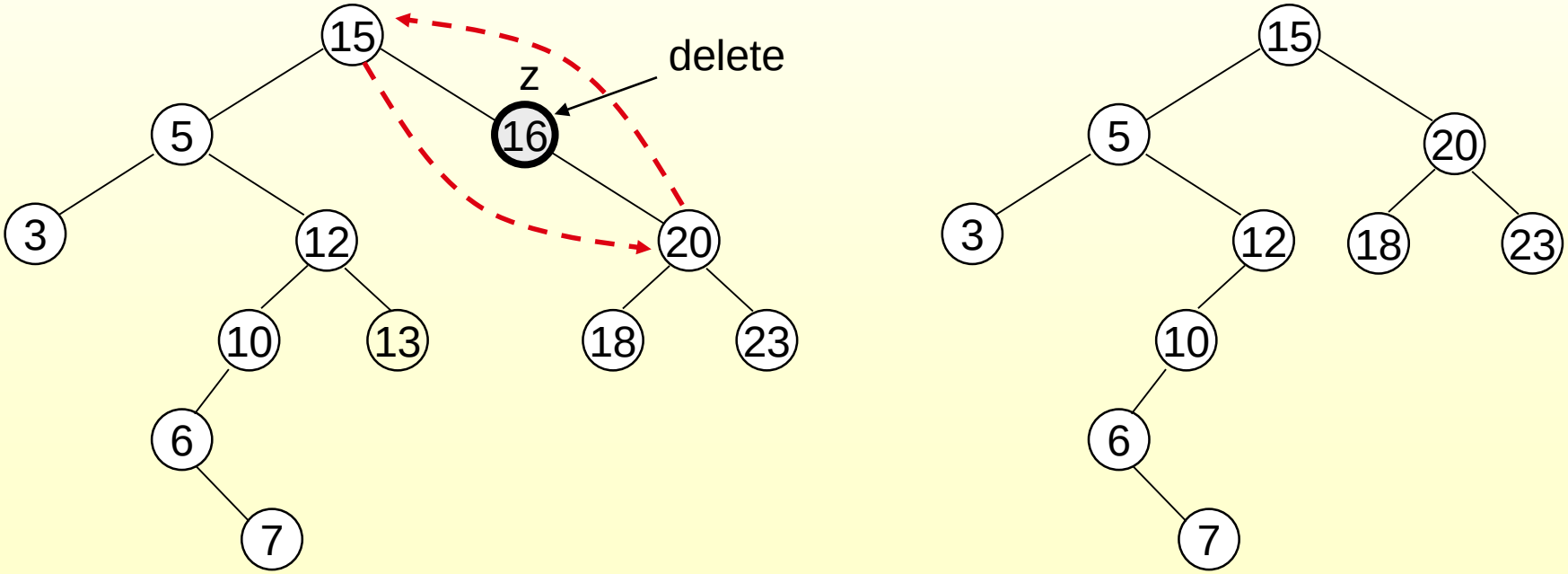
- Goal:
  - Delete a given node **z** from a binary search tree
- Idea:
  - **Case 1:** **z** has no children
    - Delete **z** by making the parent of **z** point to NIL





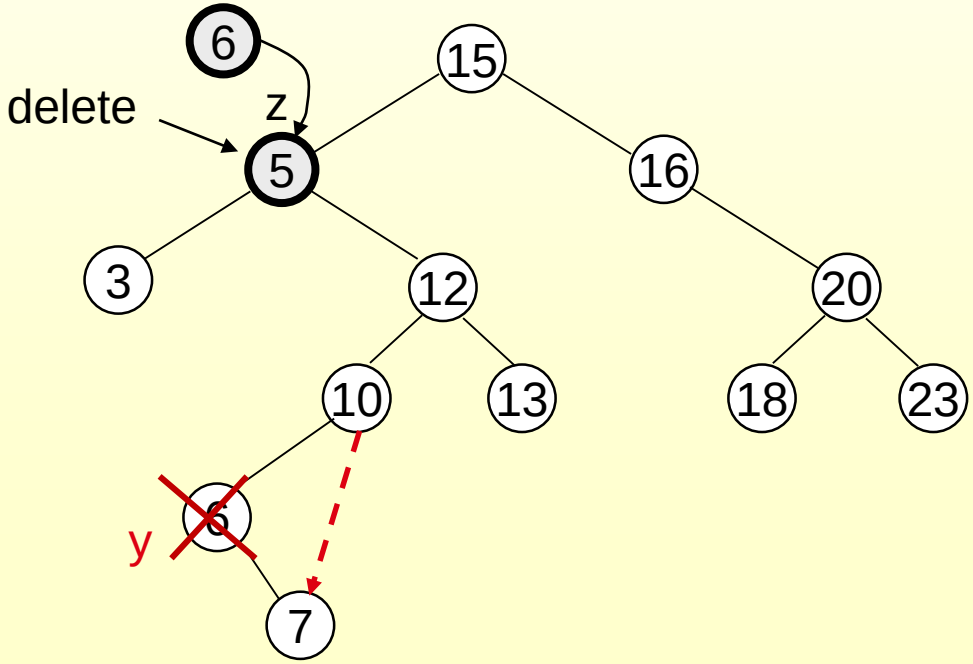
Deletion

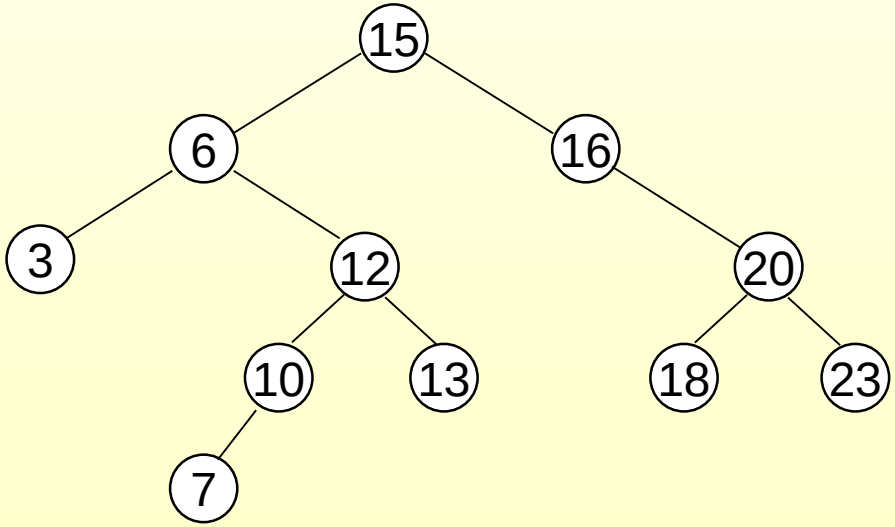
- **Case 2: z** has one child
  - Delete **z** by making the parent of **z** point to **z**'s child, instead of to **z**



Deletion

- **Case 3: z** has two children
  - **z**'s successor (y) is the minimum node in **z**'s right subtree
  - **y** has either no children or one right child (but no left child)
  - Delete **y** from the tree (via Case 1 or 2)
  - Replace z's key and satellite data with y's.



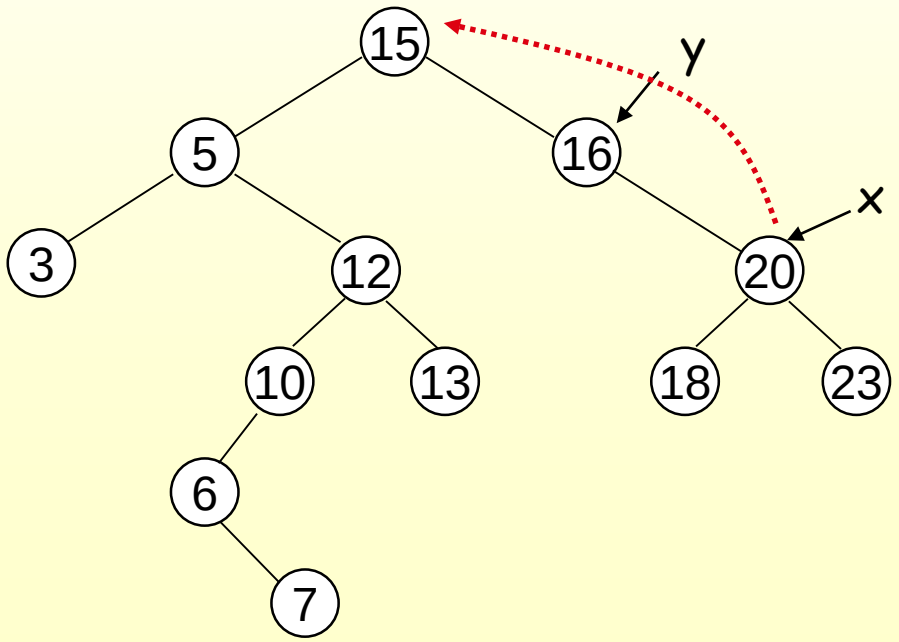


TREE-DELETE(T, z)

1. **if** left[z] = NIL or right[z] = NIL
2. **then** y ← z
3. **else** y ← TREE-SUCCESSOR(z)
4. **if** left[y] ≠ NIL
5. **then** x ← left[y]
6. **else** x ← right[y]
7. **if** x ≠ NIL
8. **then** p[x] ← p[y]

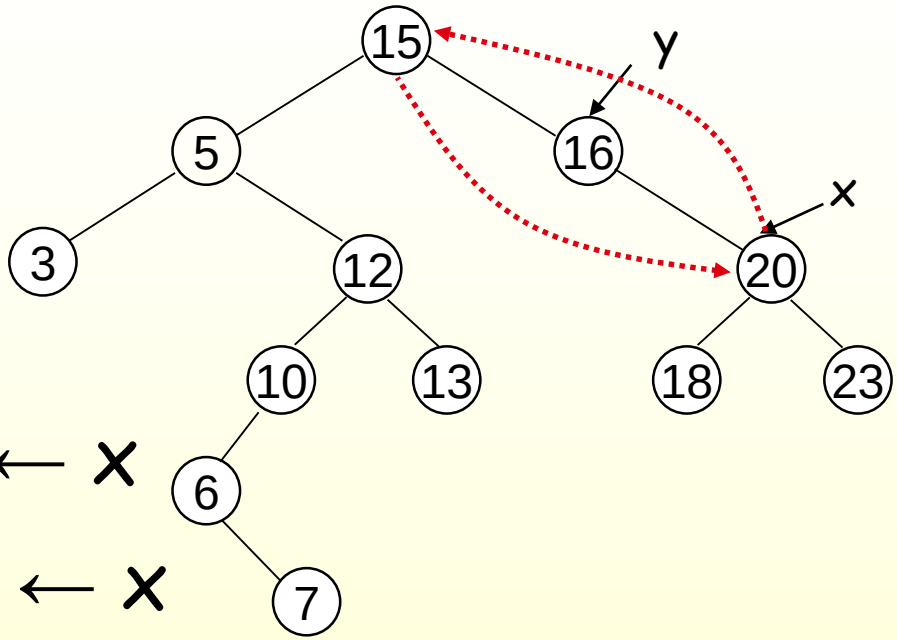
z has one child

z has 2 children



TREE-DELETE(T, z) – cont.

```
9.  if p[y] = NIL
10.    then root[T] ← x
11.    else if y = left[p[y]]
12.              then left[p[y]] ← x
13.              else right[p[y]] ← x
14. if y ≠ z
15.    then key[z] ← key[y]
16.              copy y's satellite data into z
17. return y
```



Running time: $O(h)$

Binary Search Trees - Summary

- Operations on binary search trees:
  - SEARCH $O(h)$
  - PREDECESSOR $O(h)$
  - SUCCESOR $O(h)$
  - MINIMUM $O(h)$
  - MAXIMUM $O(h)$
  - INSERT $O(h)$
  - DELETE $O(h)$
- These operations are fast if the height of the tree is *small* – otherwise their performance is similar to that of a linear linked list

Sorting with BSTs

TREESORT

Sorting With BSTs

- Informal code for sorting array A of length *n*:

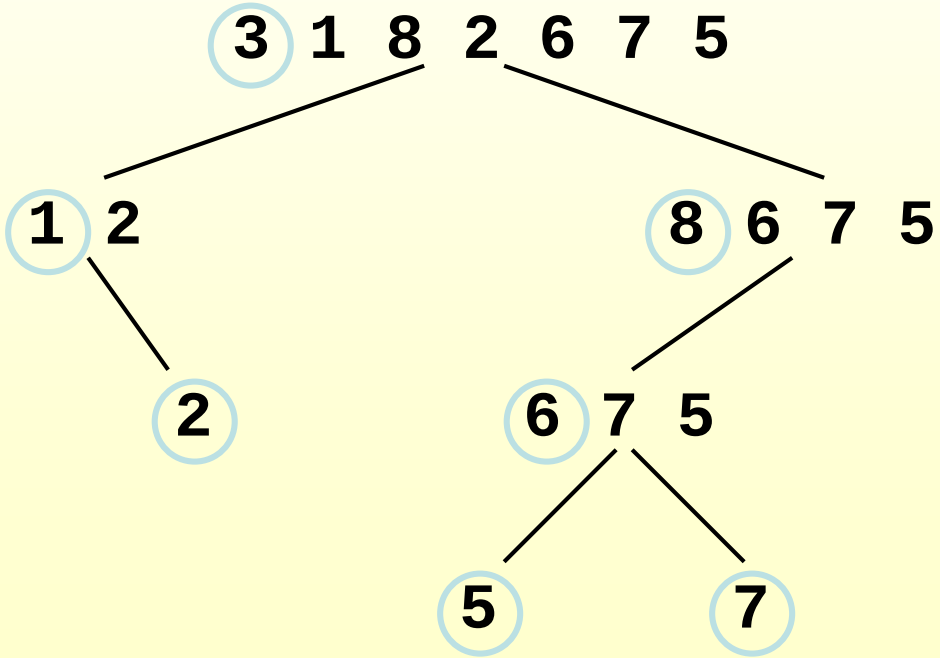
```
TreeSort(A)
    for i=1 to n
        TreeInsert(A[i]);
    InorderTreeWalk(root);
```

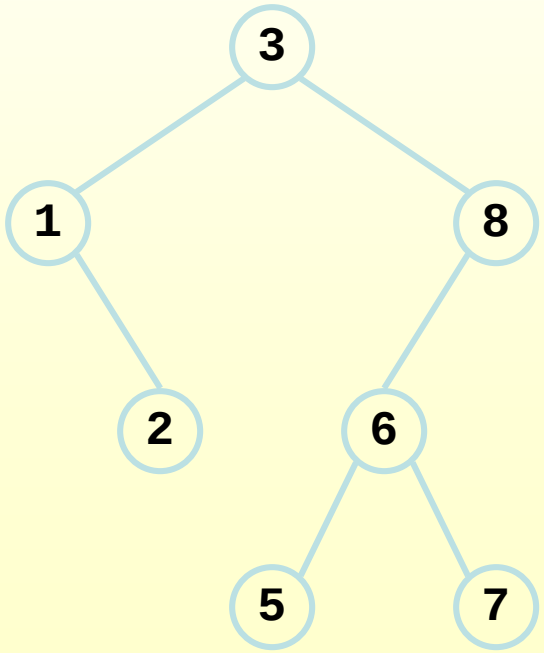
- *Argue that this is $\Omega(n \lg n)$*
- *What will be the running time in the*
  - *Worst case?*
  - *Average case? (hint: remind you of anything?)*

Sorting With BSTs

- A form of QuickSort!

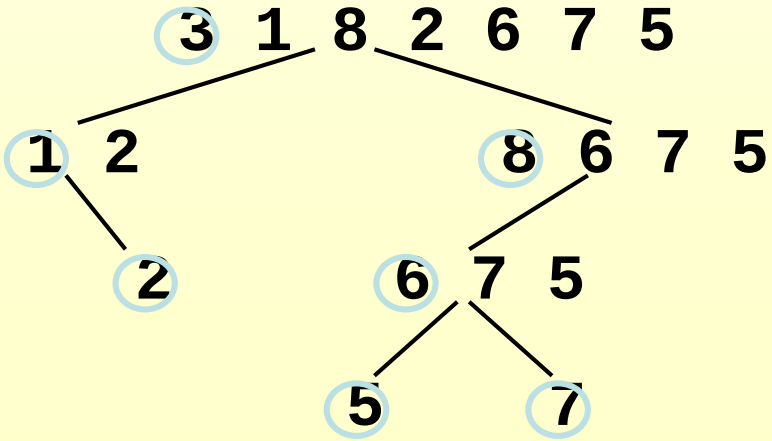
```
for i=1 to n
    TreeInsert(A[i]);
InorderTreeWalk(root);
```

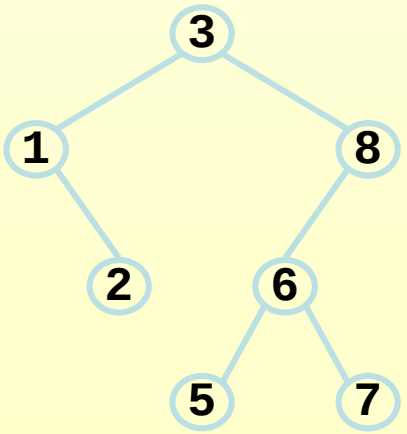




Sorting with BSTs

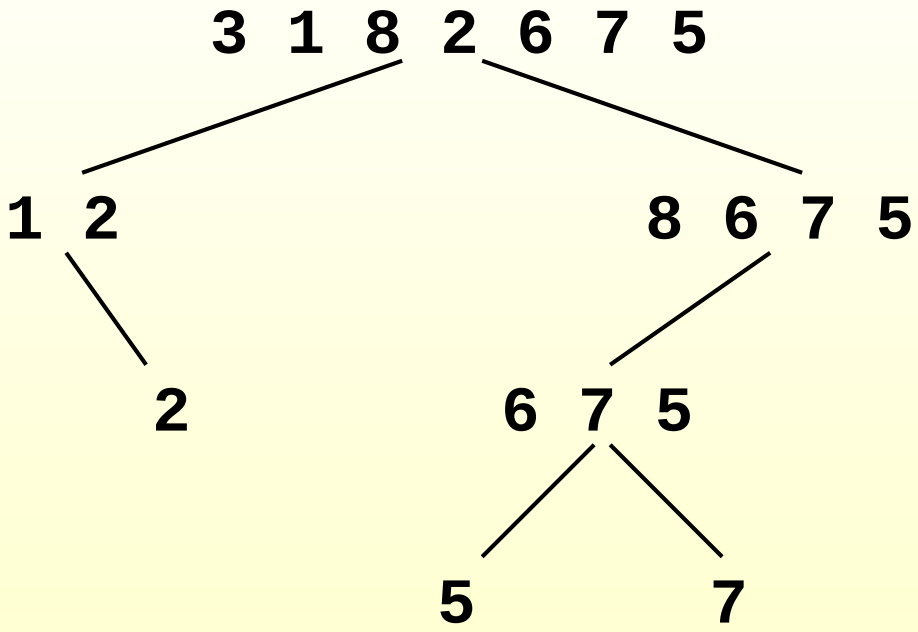
- Same partitions are done as with QuickSort.
- Same comparisons:
  - In previous example
    - Everything was compared to 3 once
    - Then those items < 3 were compared to 1 once
    - Etc.
- But! Comparisons happen in a different order.
  - Example: consider inserting 5





Analysis of TreeSort

TreeSort performs the same comparisons as QuickSort, but in a different order.



The expected time to build the tree is asymptotically the same as the running time of QuickSort.

Sorting with BSTs

- Since run time is proportional to the number of comparisons, same time as QuickSort: O(n lg n)

- *Which do you think is better, QuickSort or TreeSort? Why?*

Sorting with BSTs

- Since run time is proportional to the number of comparisons, same time as quicksort: O(n lg n)
- *Which do you think is better, QuickSort or TreeSort? Why?*
- A: QuickSort
  - Better constants
  - Sorts in place
  - Doesn’t need to build a data structure

Since everything we've done so far depends on the height of the tree, we'd better analyze…

EXPECTED HEIGHT

Expected Height

- Claim: The expected height of a randomly build BST is O(lg(n)).
- Proof: **(NOT QUITE!)**
  - The depth of a node is the number of comparisons made during TREE-INSERT.
  - Average expected node depth:

    $= \frac{1}{n}\sum_{i=1}^{n} \mathbb{E}[\text{number of comparisons to insert node i}]$

    $= O\left(\frac{1}{n} \cdot n \lg(n)\right) = O(\lg(n))$ by the QuickSort analysis.
  - So most nodes aren't too deep, so the expected height is small? **(NOT NECESSARILY!)**

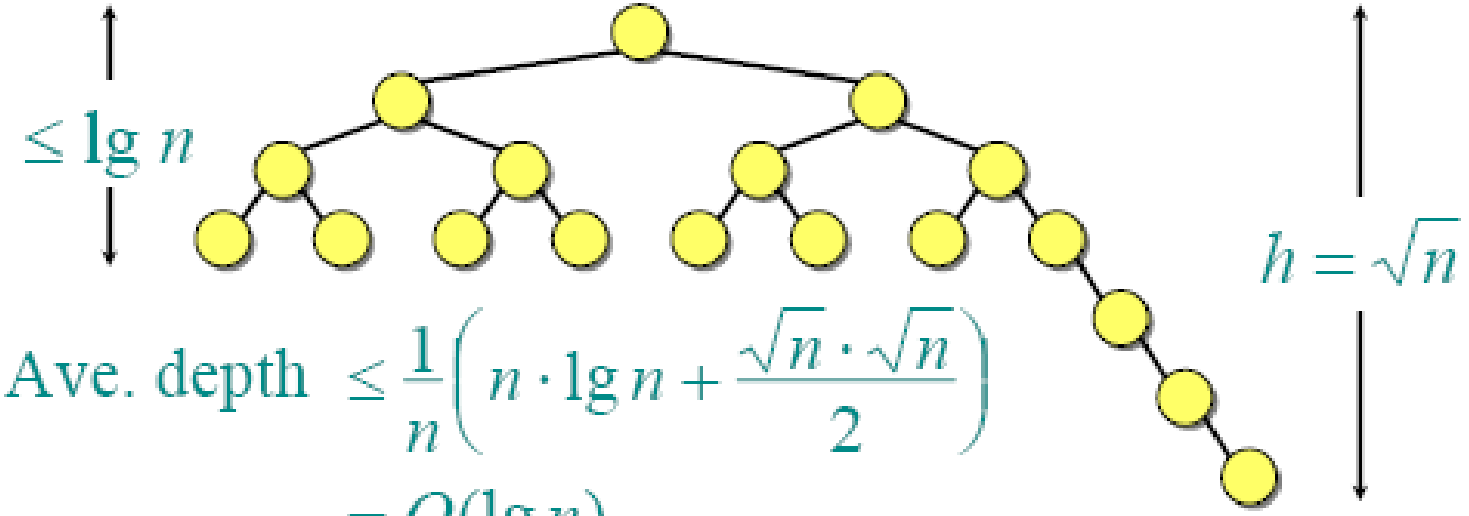
Expected Node Depth

- Claim: The expected average node depth of a randomly built BST is O(lg(n)).
- Proof: previous slide.

Expected Tree Height

- The height of a tree might be much larger than the average depth!
- So we need a more complicated analysis.

Example.



$$\text{Ave. depth} \leq \frac{1}{n}\left(n \cdot \lg n + \frac{\sqrt{n} \cdot \sqrt{n}}{2}\right)$$

$$= O(\lg n)$$

The plan
to estimate Expected Tree Height

- Review ***Jensen's inequality:***
  - $f(\mathbb{E}[X]) \leq \mathbb{E}[f(X)]$ for any convex function f and random variable X.
- Analyze exp(height) of a randomly built BST:
  - $Y_n = 2^{X_n}$, where X_n is the height of a random tree on n nodes.
  - We'll show $\mathbb{E}[Y_n] = O(n^3)$, hence
    $$2^{\mathbb{E}[X_n]} \leq \mathbb{E}[2^{X_n}] = \mathbb{E}[Y_n] = O(n^3).$$
  - so $\mathbb{E}[X_n] = O(\lg(n))$.

The details will be on the board.
There are hidden slides after this for reference later.

Post Mortem

Q. Does the analysis have to be this hard?

Q. Why bother with analyzing exponential height?

Q. Why not just develop the recurrence on

$$X_n = 1 + \max\{X_{k-1}, X_{n-k}\}$$

directly?

Post Mortem (Continued)

A.

- The inequality

$$\max\{a, b\} \leq a + b$$

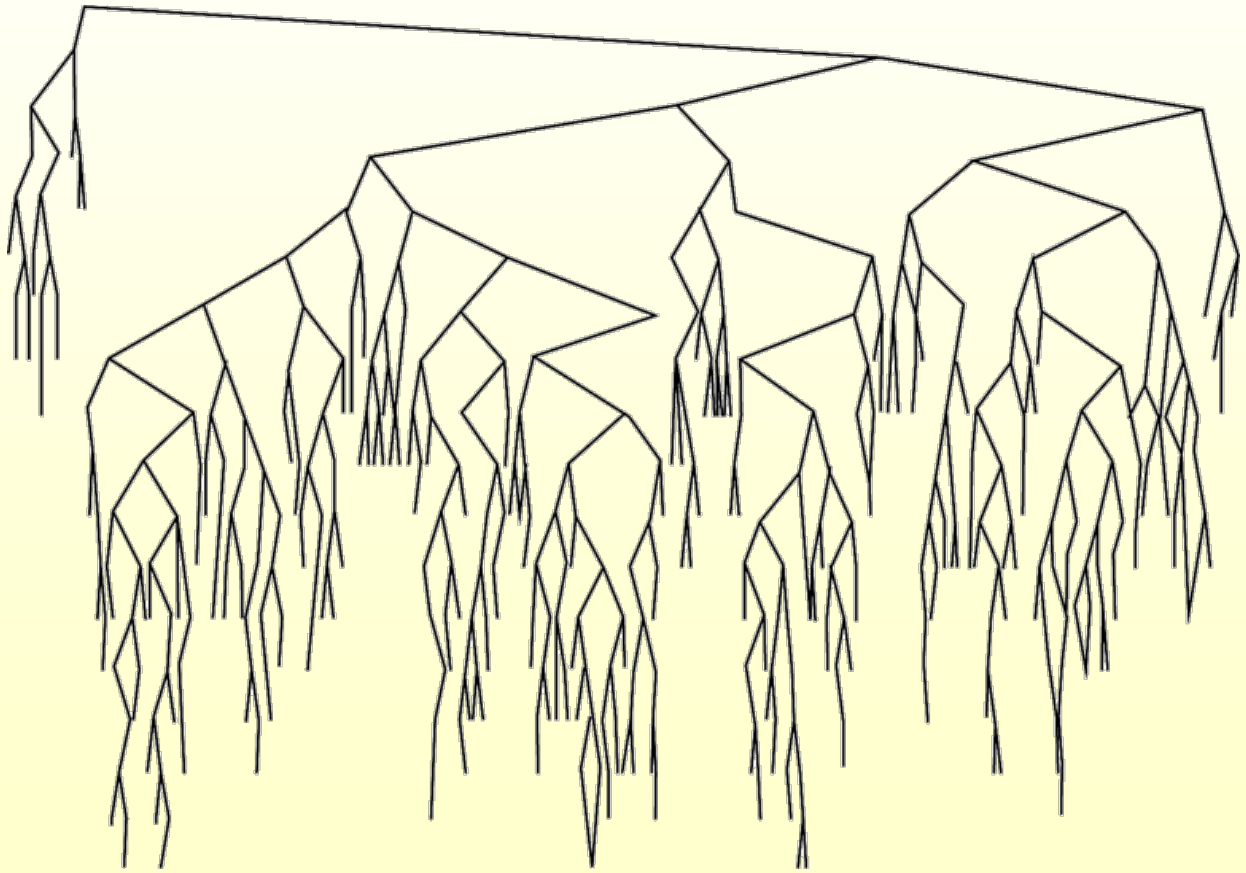
  provides a poor upper bound, since the RHS approaches the LHS slowly as $|a - b|$ increases.
- The bound

$$\max\{2^a, 2^b\} \leq 2^a + 2^b$$

  allows the RHS to approach the LHS more quickly as $|a - b|$ increases.
- By using the convexity of $f(x) = 2^x$ via Jensen's inequality, we can manipulate the sum of exponentials, resulting in a tight analysis.

Example Random BST

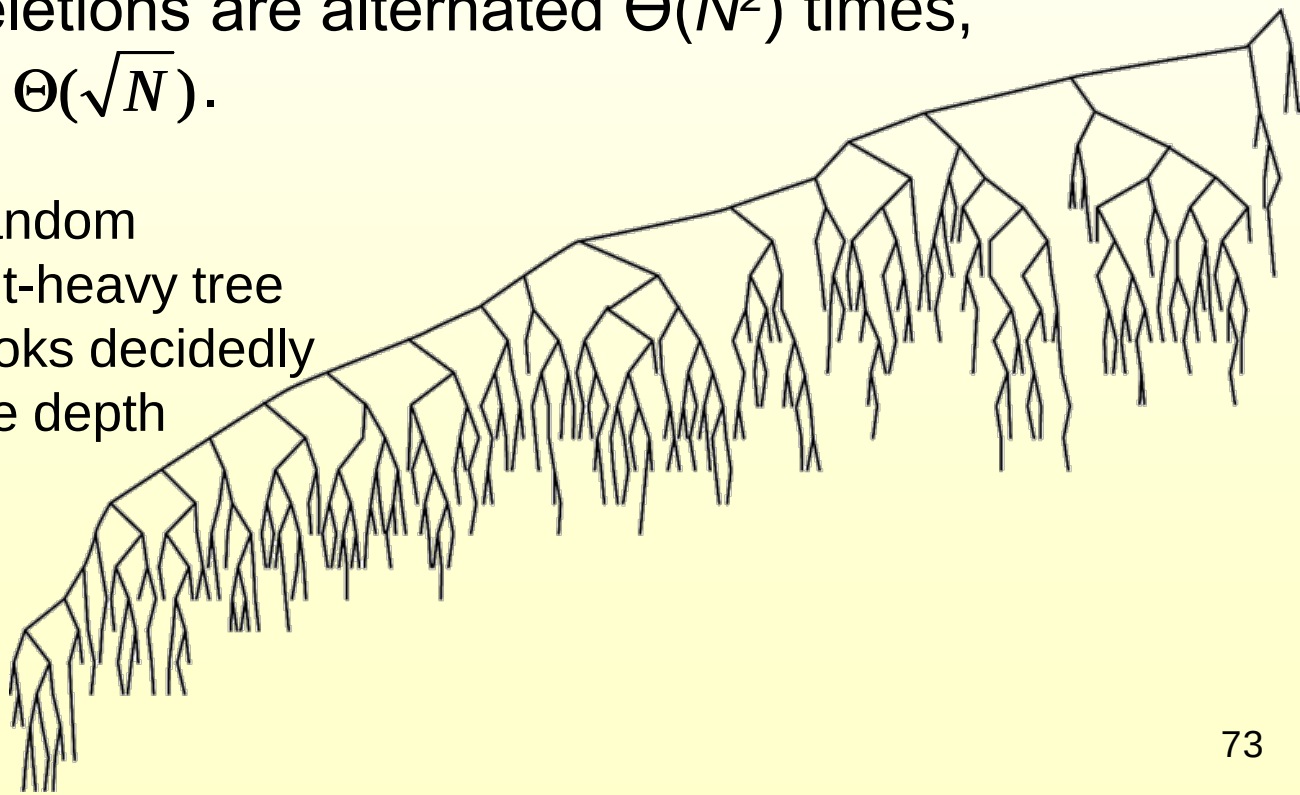
- An example of a randomly generated 500 node BST
- nodes at expected depth 9.98, height = 17.



Not All Tree Operations Preserve Randomness

- Deletion algorithm described favors making left subtrees deeper than right subtrees (a deleted node is replaced with a node from the right). The exact effect of this still unknown, but if insertions and deletions are alternated $\Theta(N^2)$ times, expected depth is $\Theta(\sqrt{N})$.

After a quarter-million random insert/remove pairs, right-heavy tree on the previous slide, looks decidedly unbalanced and average depth becomes 12.51.



BST Tree Height Summary

- The height h of a binary search tree on n items can be as high as n-1
- However, if it is built by inserting the elements in random order, then the expected height is O(lg n).

Wrap-up

- Binary Search Trees
  - Traversals
  - All sorts of operations!
    Insert/Delete/Search/etc…
  - TreeSort

  - All those things depend on the height of the tree, so it's good that the expected height of a randomly build BST = O(lg(n))

Next time

- Prof. Guibas will be back!
- You'll see how to make sure that trees stay balanced!