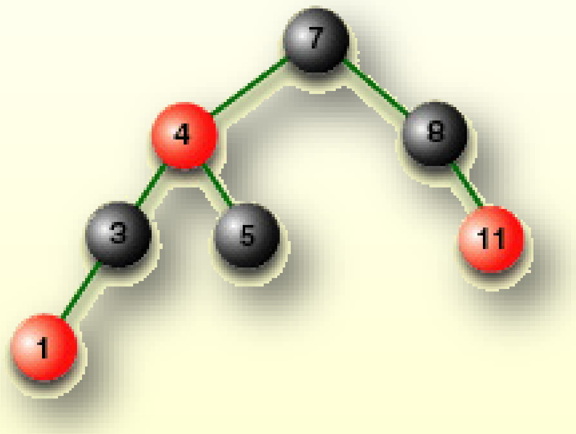
CS161:
Design and Analysis of Algorithms



Lecture 9
Leonidas Guibas

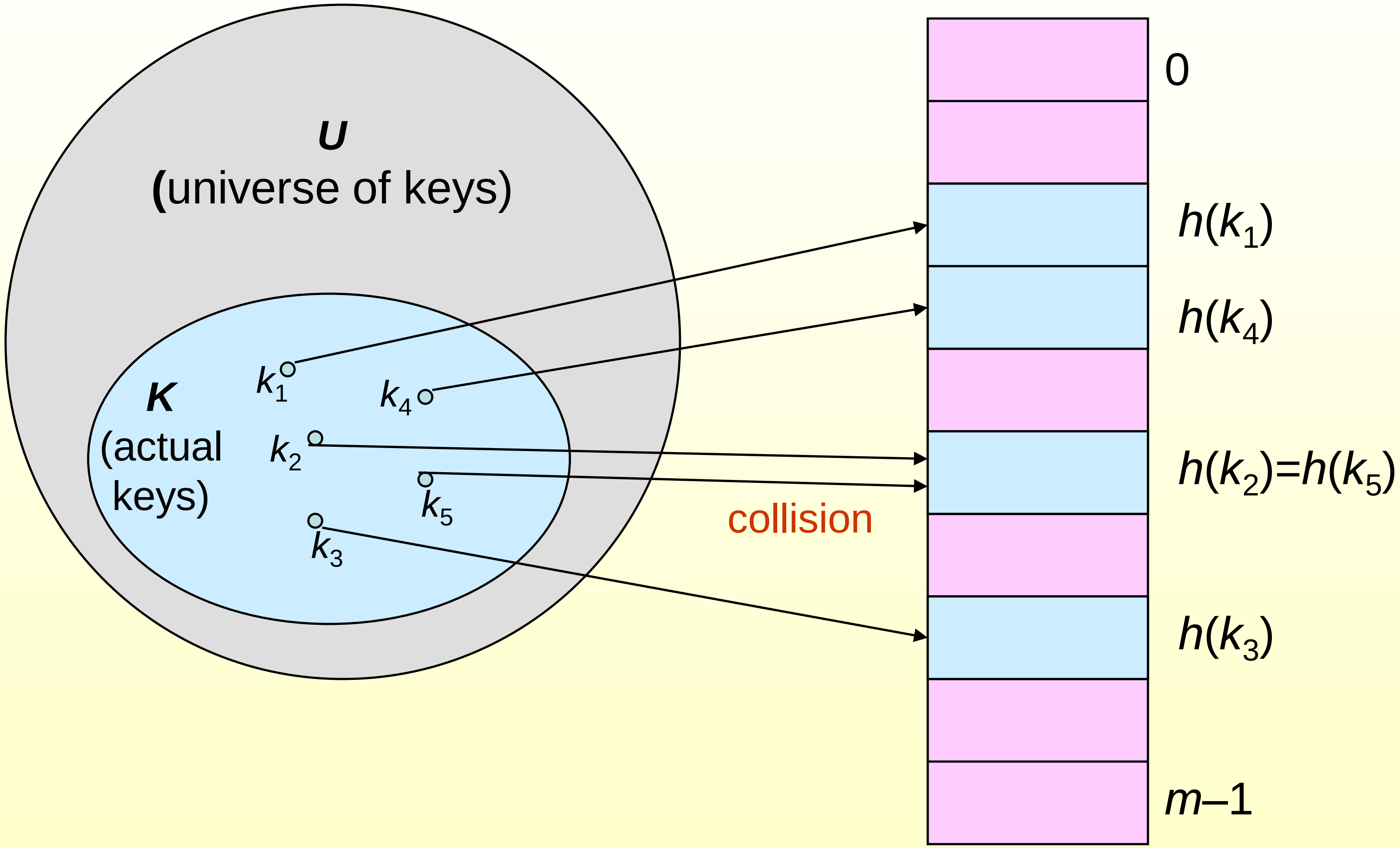
Outline

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

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

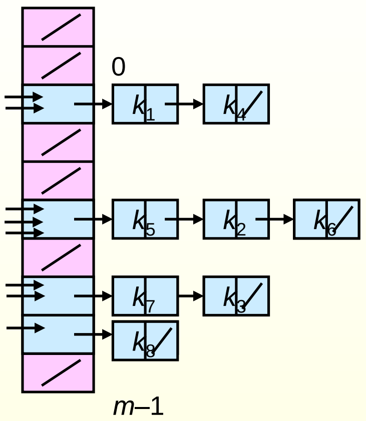
Hashing



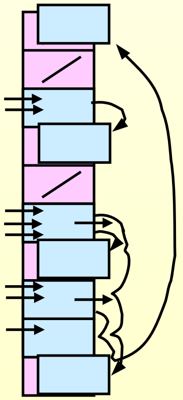
U
(universe of keys)
K
(actual keys)
k_1
k_4
k_2
k_5
k_3
collision
0
$h(k_1)$
$h(k_4)$
$h(k_2)=h(k_5)$
$h(k_3)$
m–1

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.



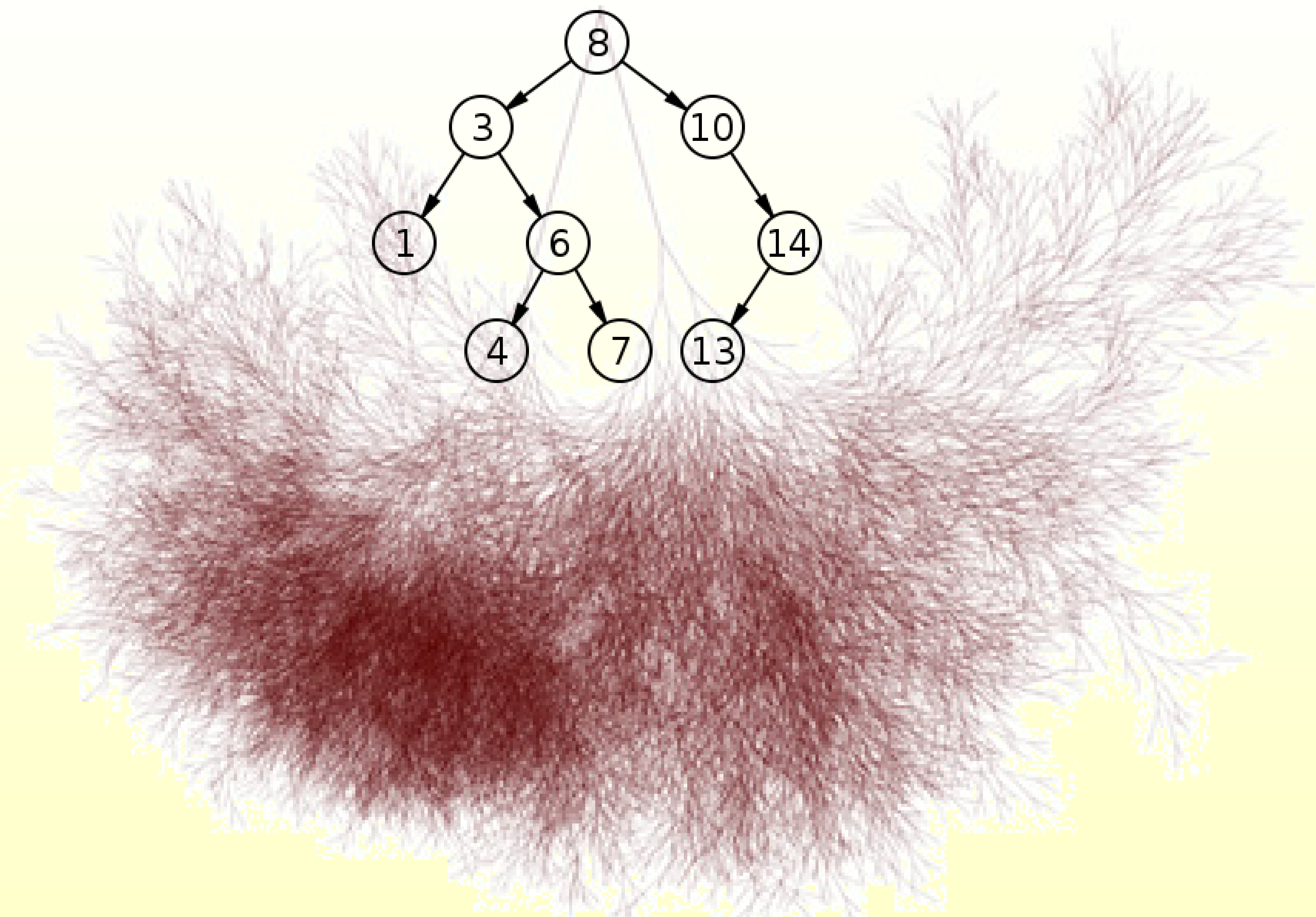
- Open Addressing:
  - All elements stored in hash table itself.
  - When collisions occur, use a systematic (consistent) procedure to store (and search for) elements in free slots of the table.



Choosing A Hash Function

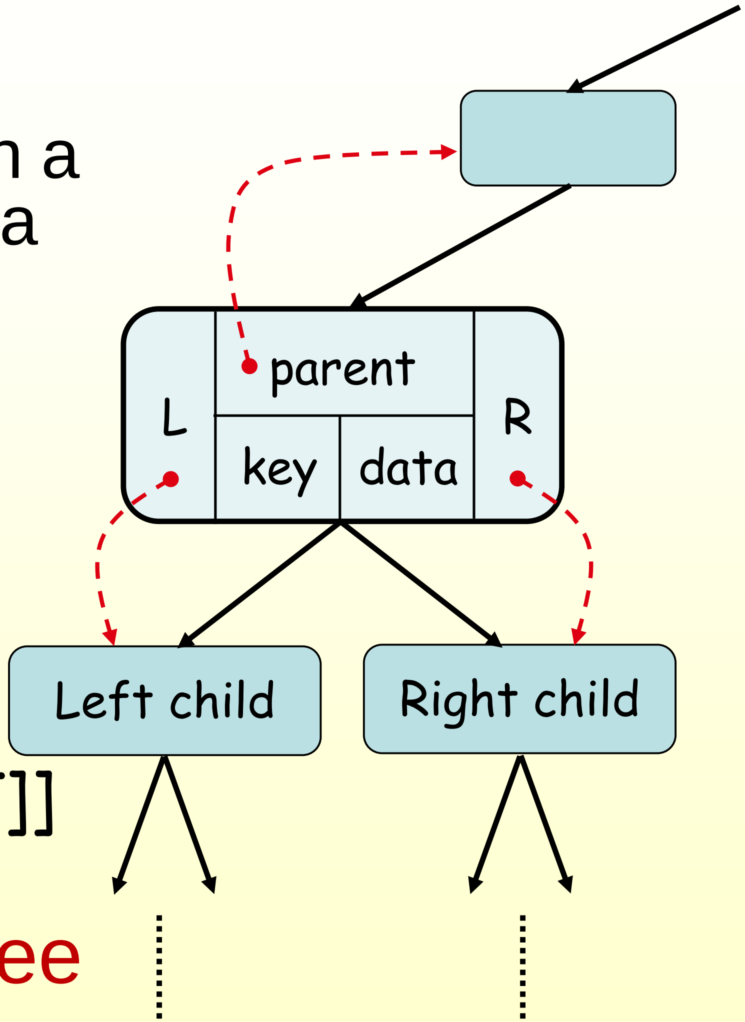
- Choosing the hash function well is crucial
  - Bad hash function puts all elements in same slot
  - A good hash function:
    - Should distribute keys uniformly into slots
    - Should not depend on patterns in the data
- We discussed two methods:
  - Division method
  - Multiplication method
- One more in worth mentioning
  - Universal hashing

Binary Search Trees



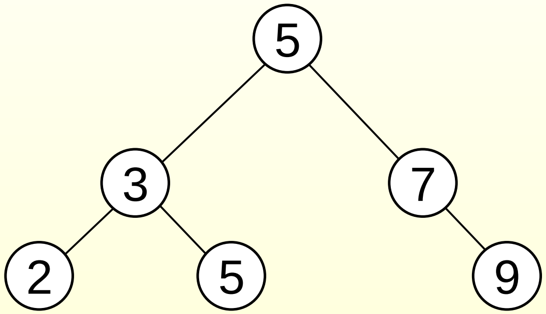
Binary Search Trees

- Tree representation:
  - A linked data structure in which a set of nodes is connected into a tree
- Node representation:
  - Key field
  - Satellite data
  - Left: pointer to left child
  - Right: pointer to right child
  - p: pointer to parent (p [root [T]] = NIL)
- Satisfies the binary-search-tree property



Binary Search Tree Property

- Binary search tree order property:
  - If y is in left subtree of x, then $key[y] \leq key[x]$
  - If y is in right subtree of x, then $key[y] \geq key[x]$



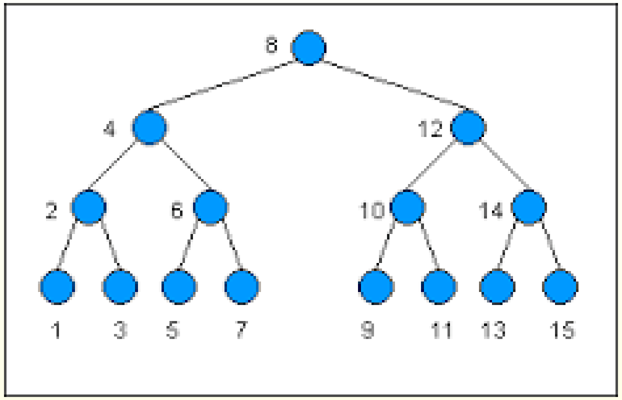
Binary Search Trees

- 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

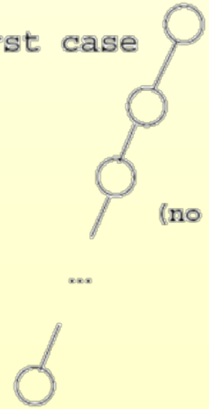
- Running time of basic operations on binary search trees
  - On average: $\Theta(\lg n)$ [Really $O(h)$, h = height of tree]
    - The expected height of the tree is $\lg n$ (balanced case)
  - In the worst case: $\Theta(n)$
    - The tree is a linear chain of n nodes (unbalanced case)

Best/Worst Case



balanced case

- If the tree is very **unbalanced**, then running time will be $O(n)$.

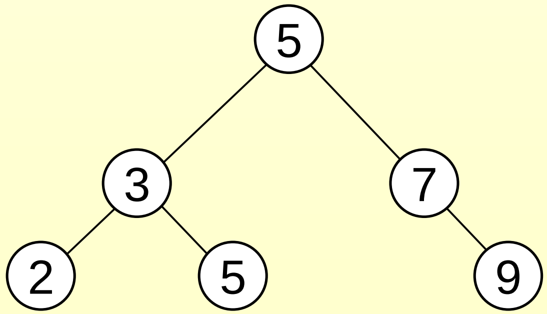


h=n-1

(no better than a linked list)

Traversing a Binary Search Tree

- **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

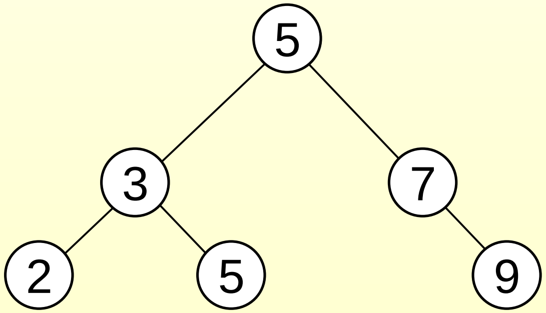
Postorder: 2 5 3 9 7 5

Traversing a Binary Search Tree

Alg: INORDER-TREE-WALK(x)

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

- *E.g.:*

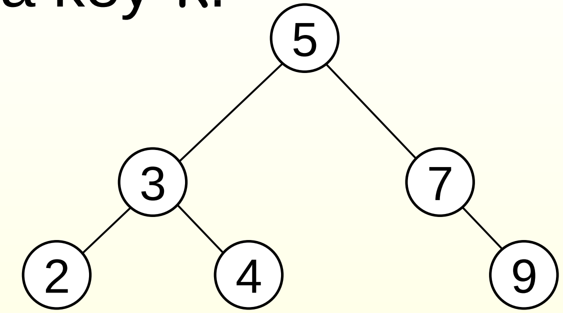


Output: 2 3 5 5 7 9

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

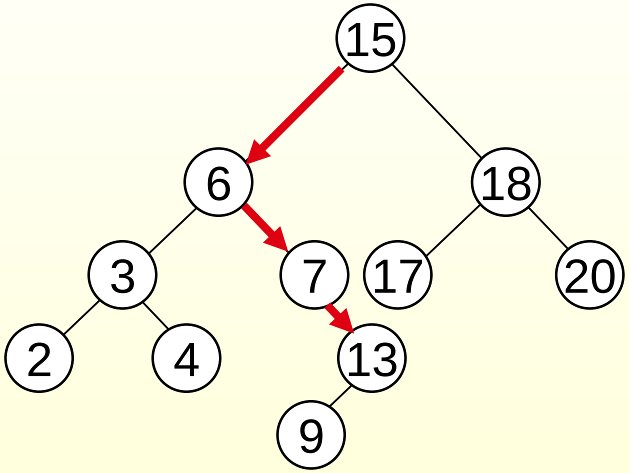
Searching for a Key

- 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

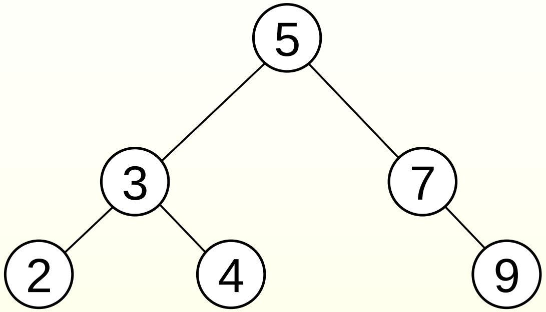
Example: TREE-SEARCH



- Search for key 13:
  - 15 → 6 → 7 → 13

Searching for a Key

Alg: TREE-SEARCH(x, k)

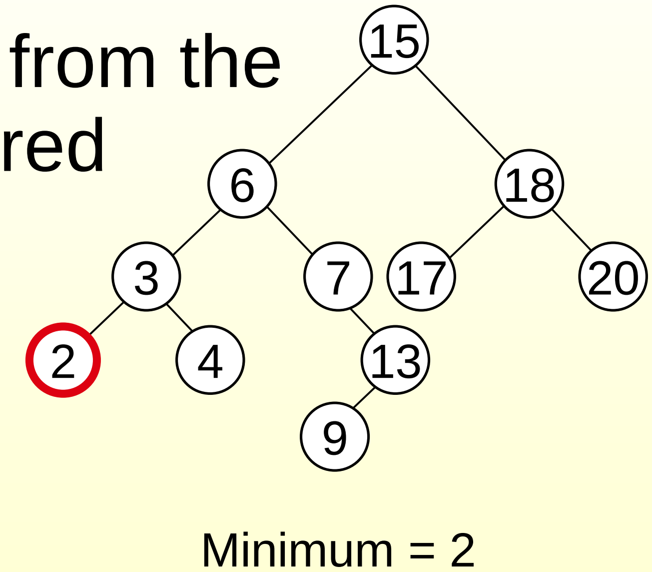



```
if x = NIL or k = key [x]
        then return x
if k < key [x]
        then return TREE-SEARCH(left [x], k)
        else return TREE-SEARCH(right [x], k)
```

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

Finding the Minimum in a Binary Search Tree

- Goal: find the minimum value in a BST
  - Following left child pointers from the root, until a NIL is encountered



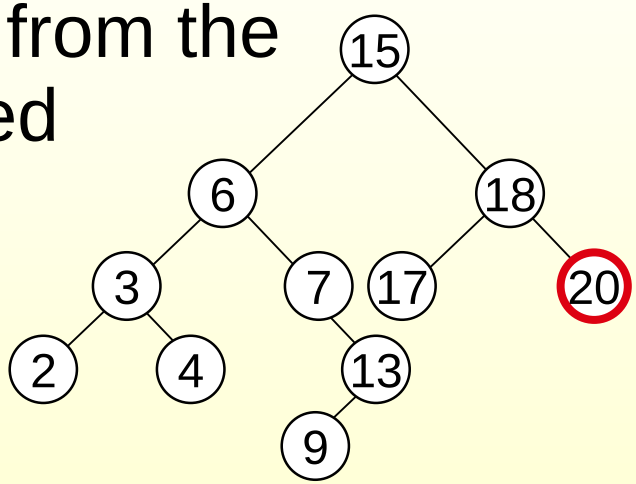
Alg: TREE-MINIMUM(x)

1. **while** $left[x] \neq$ NIL
2. **do** $x \leftarrow left[x]$
3. **return** x

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

Finding the Maximum in a Binary Search Tree

- Goal: find the maximum value in a BST
  - Following right child pointers from the root, until a NIL is encountered



Alg: TREE-MAXIMUM(*x*)

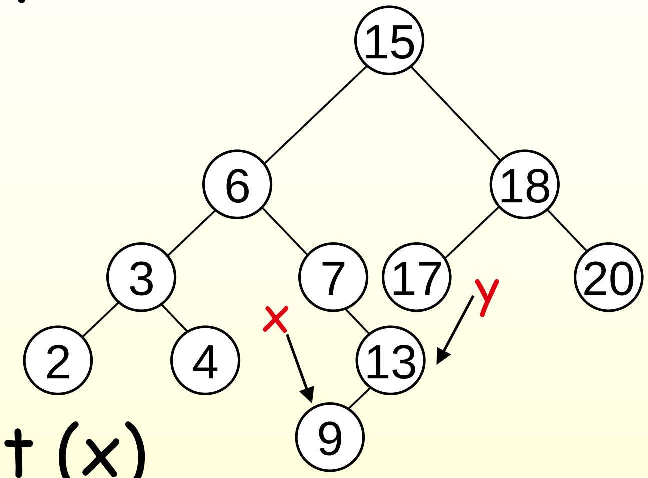
1. **while** *right* [*x*] $\neq$ NIL
2. **do** *x* $\leftarrow$ *right* [*x*]
3. **return** *x*

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

Successor

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

- *E.g.:* $successor(15) = 17$
  $successor(13) = 15$
  $successor(9) = 13$

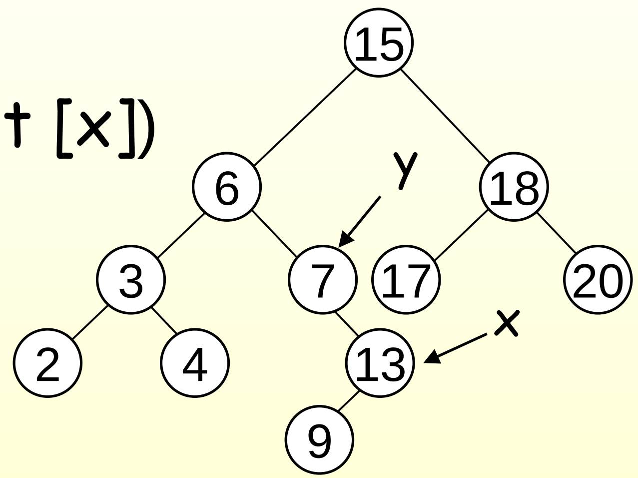


- Case 1: right (x) is non empty
  - $successor(x)$ = the minimum in right (x)
- Case 2: right (x) is empty
  - go up the tree until the current node is a left child: $successor(x)$ is the parent of the current node
  - if you cannot go further (and you reached the root): x is the largest element

Finding the Successor

Alg: TREE-SUCCESSOR(*x*)

```
if right [x] ≠ NIL
   then return TREE-MINIMUM(right [x])
y ← p[x]
while y ≠ NIL and x = right [y]
   do x ← y
      y ← p[y]
return y
```

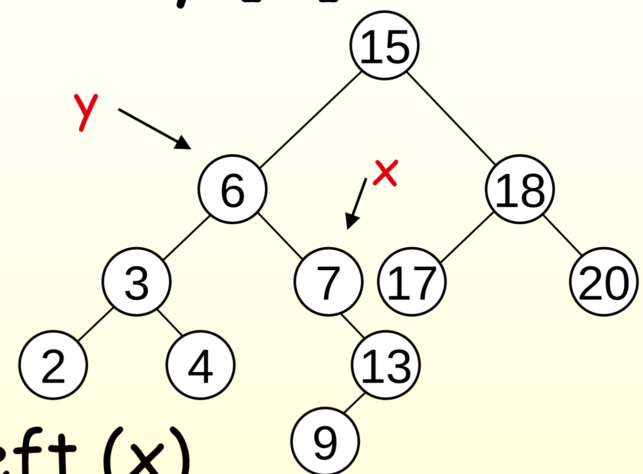


Running time: O (h), h = height of the tree

Predecessor

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

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



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

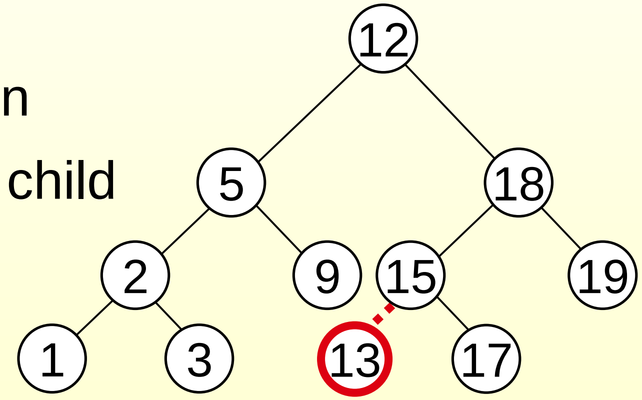
Insertion

- 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
  - If v < key [y] insert the new node as y's left child

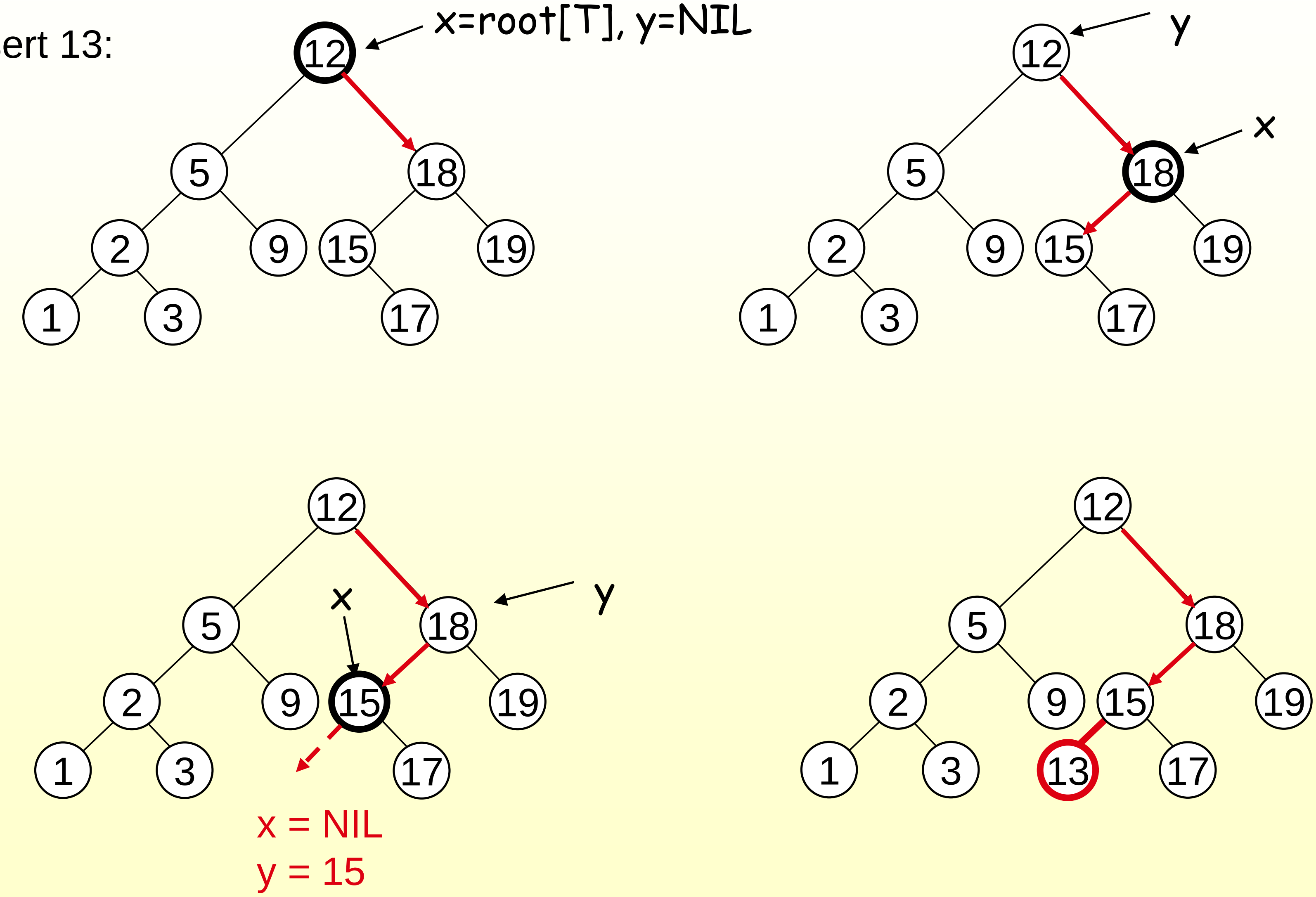
    else insert it as y's right child
  - Beginning at the root, go down the tree and maintain:
    - Pointer x : traces the downward path (current node)
    - Pointer y : parent of x ("trailing pointer")

Insert value 13



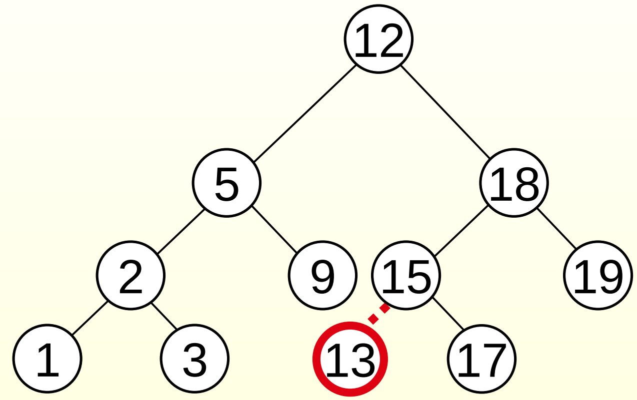
Example: TREE-INSERT

Insert 13:



Alg: TREE-INSERT(T, z)

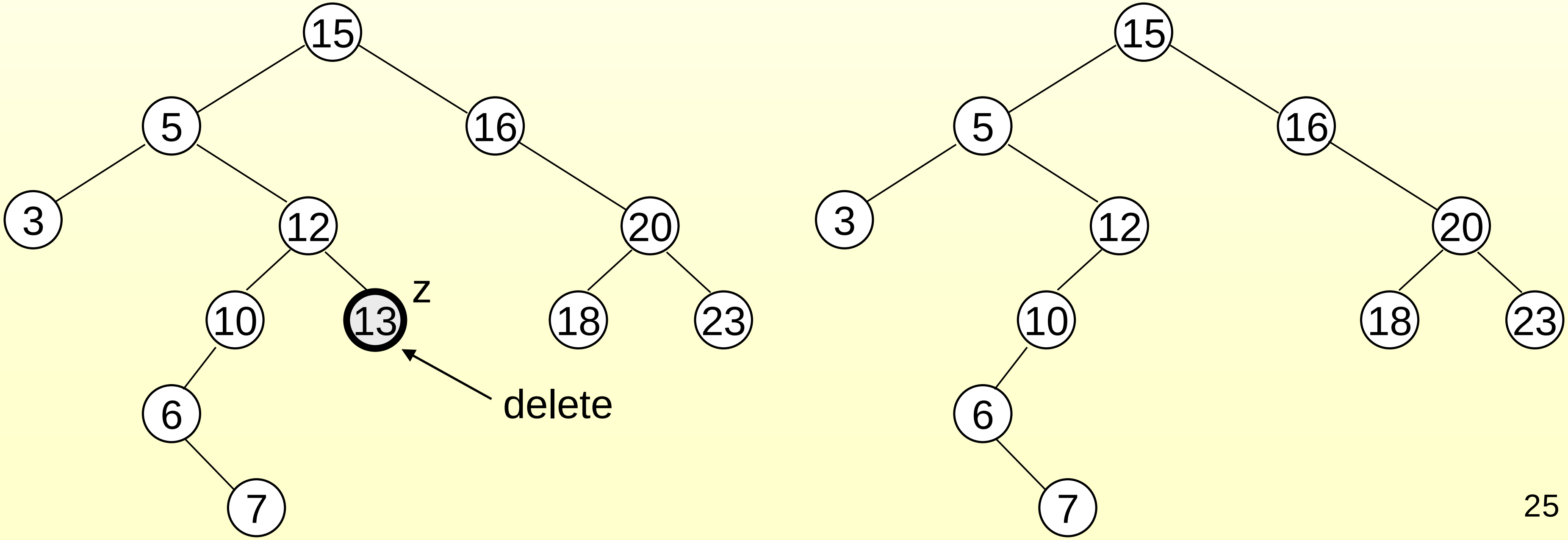
```
y ← NIL
x ← root [T]
while x ≠ NIL
     do y ← x
          if key [z] < key [x]
             then x ← left [x]
             else x ← right [x]
p[z] ← y
if y = NIL
   then root [T] ← z                    ▷ Tree T was empty
   else if key [z] < key [y]
            then left [y] ← z
            else right [y] ← z
```



Running time: *O*(h)

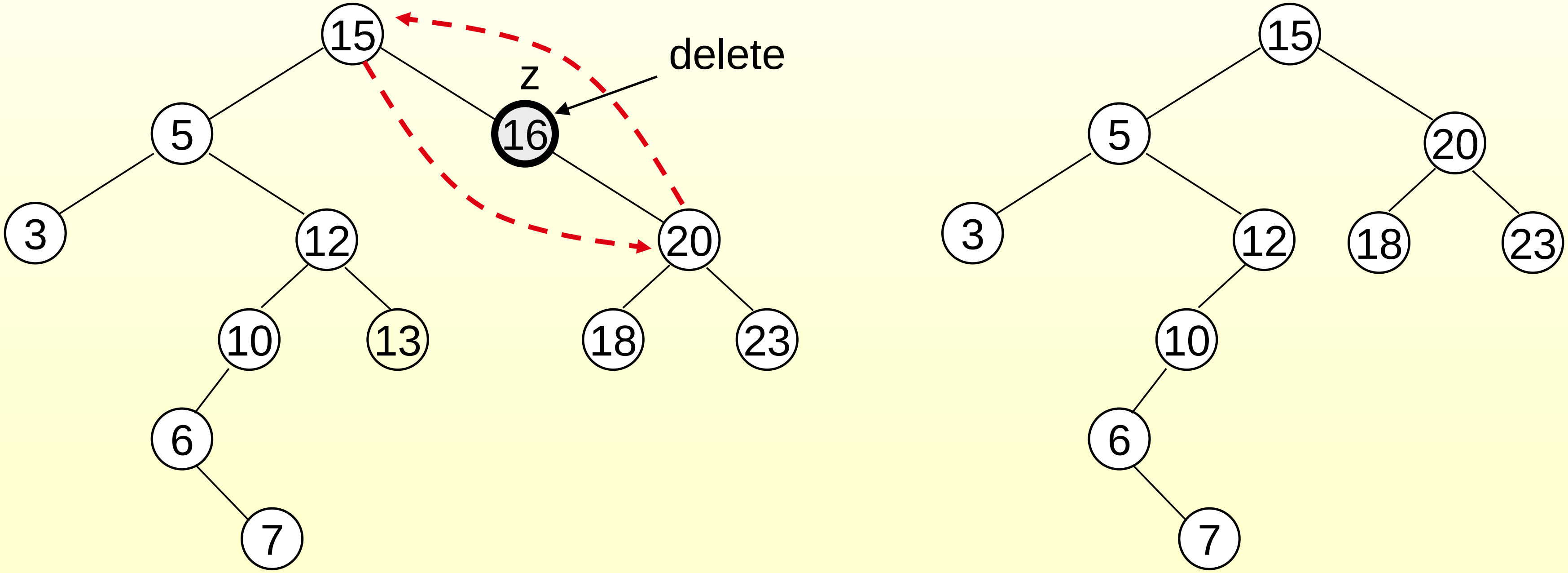
Deletion

- 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



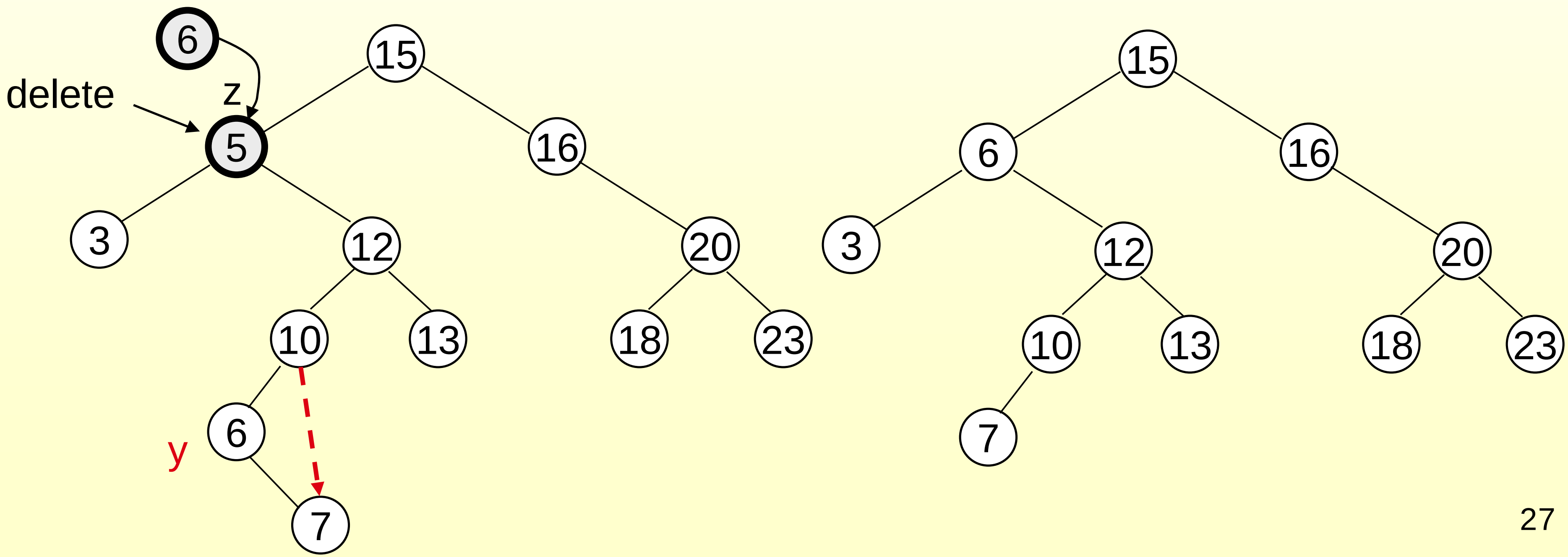
Deletion

- **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.

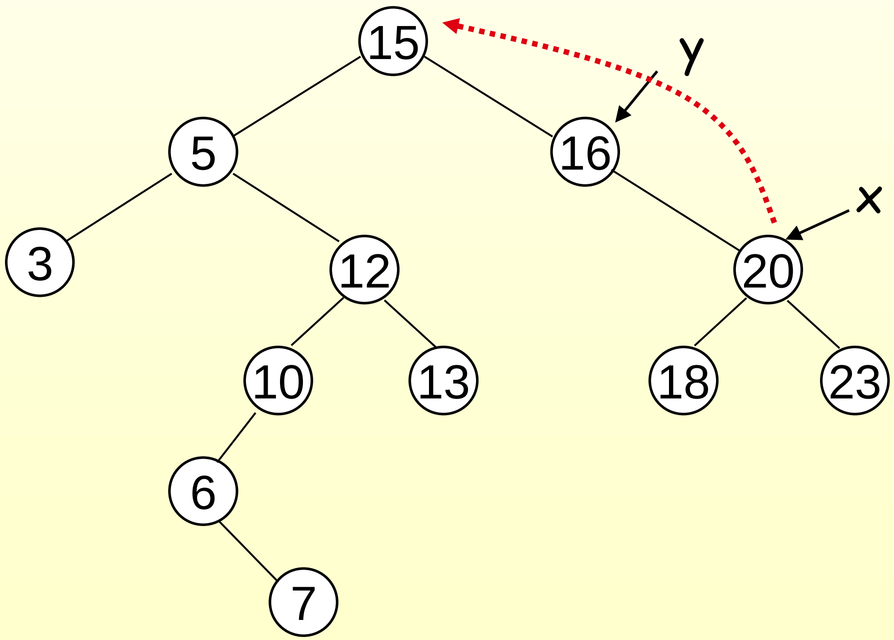


TREE-DELETE(T, z)

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]

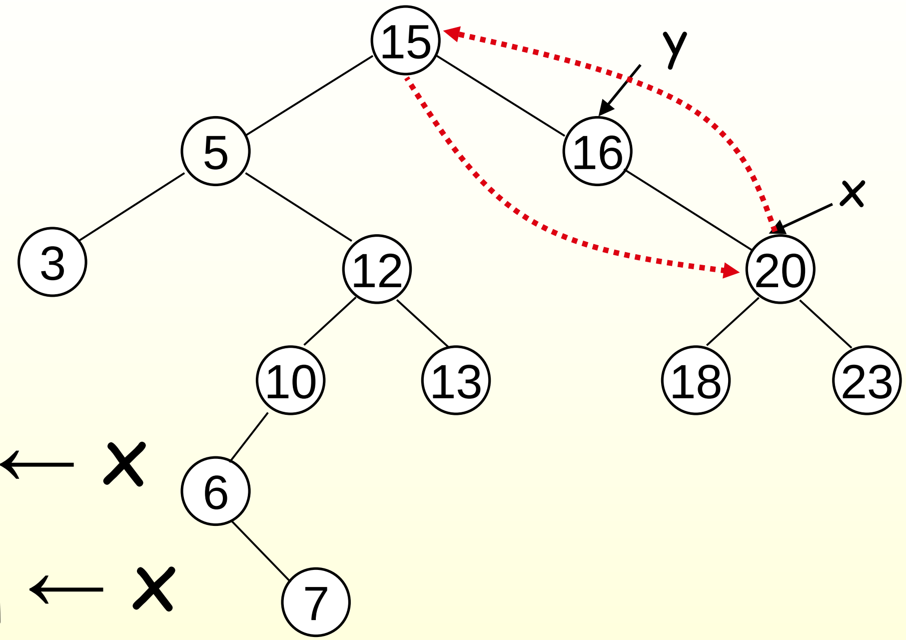
z has one child

z has 2 children



TREE-DELETE(T, z) – cont.

```
if p[y] = NIL
   then root[T] ← x
   else if y = left[p[y]]
           then left[p[y]] ← x
           else right[p[y]] ← x
if y ≠ z
   then key[z] ← key[y]
        copy y's satellite data into z
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 Binary Search Trees

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

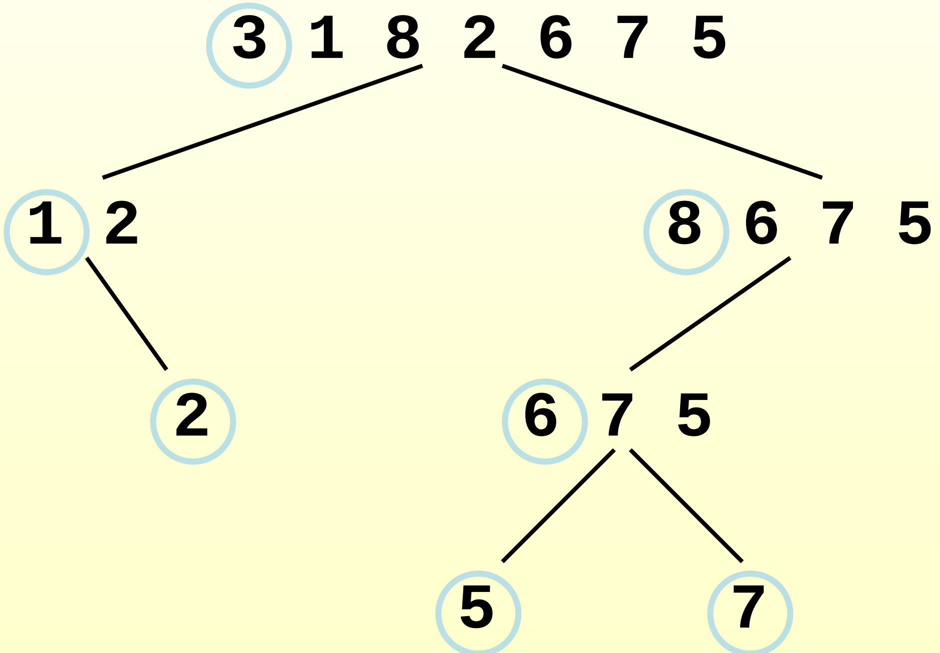
```
TreeSort(A)
    for i=1 to n
        TreeInsert(A[i]);
    InorderTreeWalk(root);
```

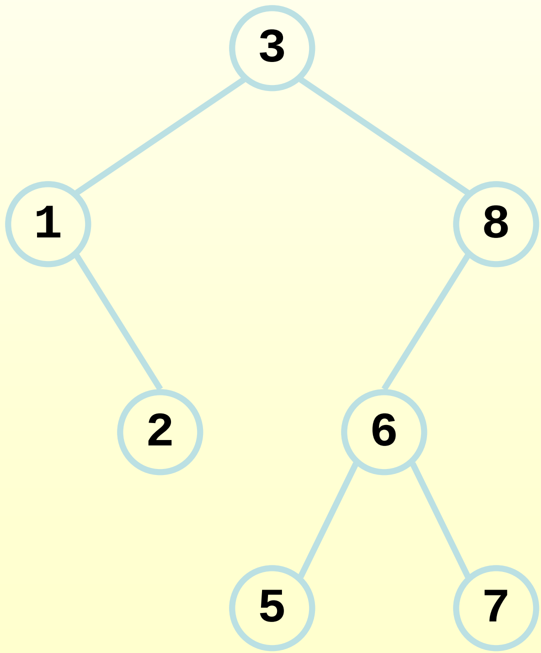
- *Argue that this is $\Omega(n \lg n)$*
- *What will be the running time in the*
  - *Worst case?*
  - *Average case? (hint: remind you of anything?)*

Sorting With BSTs

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

- Average case analysis
  - It’s a form of QuickSort



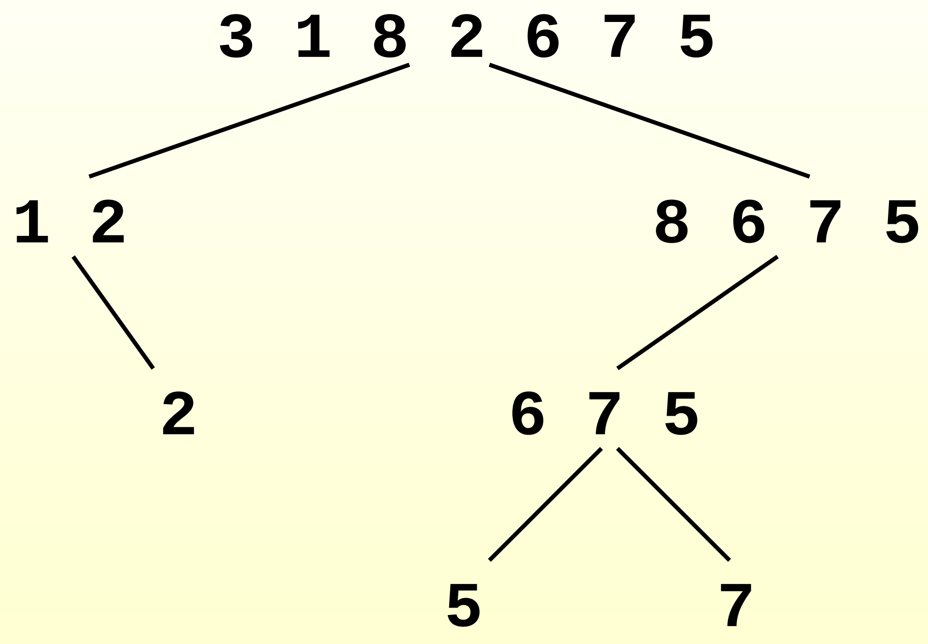


Sorting with BSTs

- Same partitions are done as with QuickSort, but in comparisons happen in a different order
  - In previous example
    - Everything was compared to 3 once
    - Then those items < 3 were compared to 1 once
    - Etc.
  - Same comparisons as QuickSort
    - 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

Node Depth

The depth of a node = the number of comparisons made during TREE-INSERT. Assuming all input permutations are equally likely, we have

Average node depth

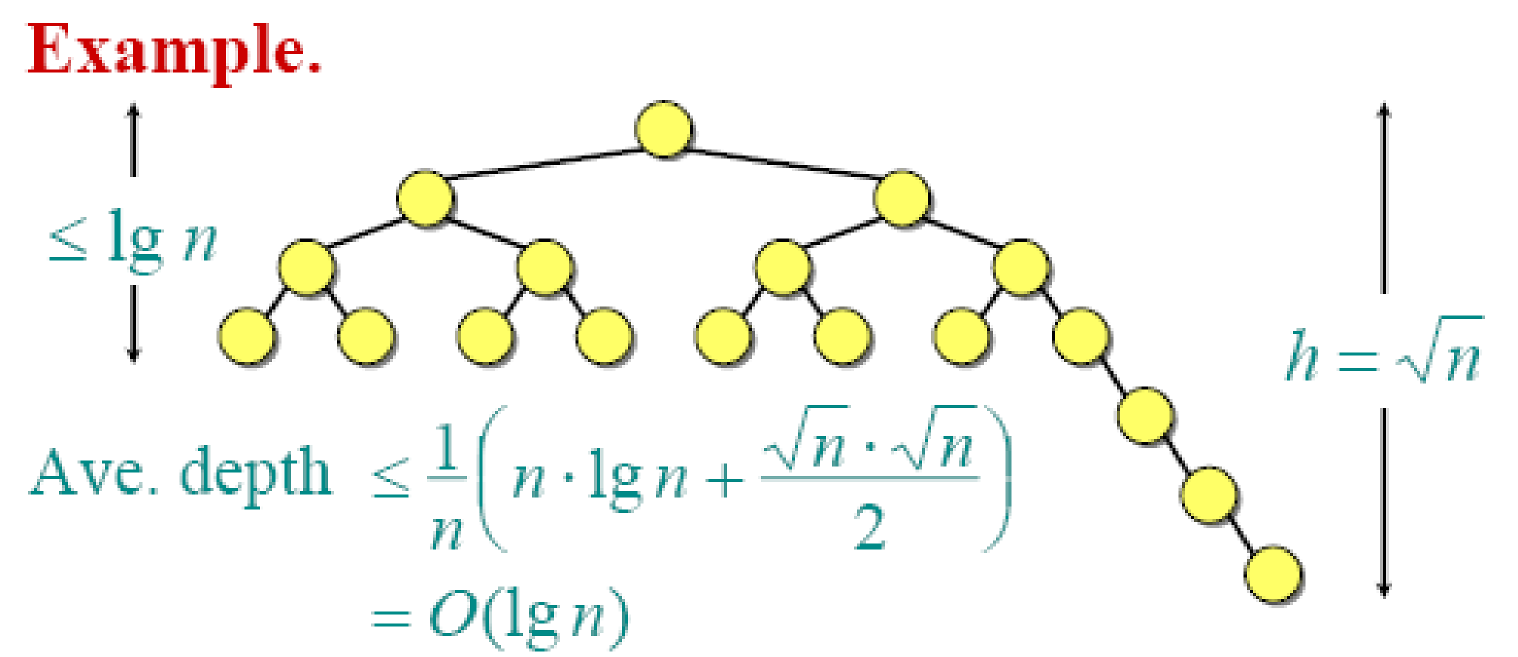
$$= \frac{1}{n} E\left[\sum_{i=1}^{n} (\#\text{comparisons to insert node } i)\right]$$

$$= \frac{1}{n} O(n \lg n) \qquad \text{(quicksort analysis)}$$

$$= O(\lg n) \ .$$

Expected Tree Height

But, the fact that the average node depth of a randomly built BST = $O(\lg n)$ does not necessarily mean that its expected height is also $O(\lg n)$ (although it is).



Expected Tree Height – How to Estimate?

Outline of the analysis:

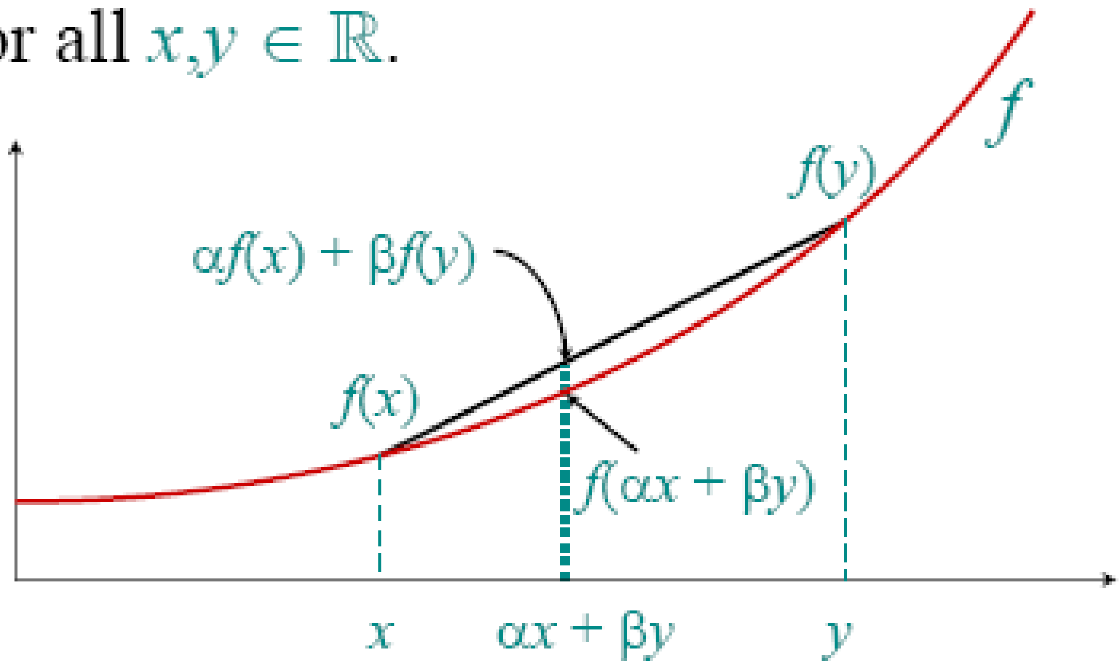
- Review ***Jensen's inequality***, which says that $f(E[X]) \le E[f(X)]$ for any convex function f and random variable X.
- Analyze the ***exponential height*** of a randomly built BST on n nodes, which is the random variable $Y_n = 2^{X_n}$, where X_n is the random variable denoting the height of the BST.
- Prove that $2^{E[X_n]} \le E[2^{X_n}] = E[Y_n] = O(n^3)$, and hence that $E[X_n] = O(\lg n)$.

Convex Functions

A function $f : \mathrm{R} \rightarrow \mathrm{R}$ is ***convex*** if for all $\alpha,\beta \geq 0$ such that $\alpha + \beta = 1$, we have

$$f(\alpha x + \beta y) \leq \alpha f(x) + \beta f(y)$$

for all $x,y \in \mathbb{R}$.



Convexity Lemma

Lemma. Let $f : \mathrm{R} \rightarrow \mathrm{R}$ be a convex function, and let $\{\alpha_1, \alpha_2, \ldots, \alpha_n\}$ be a set of nonnegative constants such that $\Sigma_k \alpha_k = 1$. Then, for any set $\{x_1, x_2, \ldots, x_n\}$ of real numbers, we have

$$f\left(\sum_{k=1}^{n} \alpha_k x_k\right) \leq \sum_{k=1}^{n} \alpha_k f(x_k).$$

Proof. By induction on n. Omitted.

Jensen's Inequality

Lemma. Let f be a convex function, and let X be a random variable. Then, $f(E[X]) \le E[f(X)]$.

Proof.

$$f(E[X]) = f\left(\sum_{k=-\infty}^{\infty} k \cdot \Pr\{X = k\}\right)$$

$$\le \sum_{k=-\infty}^{\infty} f(k) \cdot \Pr\{X = k\}$$

$$= E[f(X)].$$ □

Analysis of BST Height

Let X_n be the random variable denoting the height of a randomly built binary search tree on n nodes, and let $Y_n = 2^{X_n}$ be its exponential height.

If the root of the tree has rank k, then

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

since each of the left and right subtrees of the root are randomly built. Hence, we have

$$Y_n = 2 \cdot \max\{Y_{k-1}, Y_{n-k}\} .$$

Analysis (Continued)

Define the indicator random variable Z_{nk} as

$$Z_{nk} = \begin{cases} 1 & \text{if the root has rank } k, \\ 0 & \text{otherwise.} \end{cases}$$

Thus, $\Pr\{Z_{nk} = 1\} = E[Z_{nk}] = 1/n$, and

$$Y_n = \sum_{k=1}^{n} Z_{nk} \left(2 \cdot \max\{Y_{k-1}, Y_{n-k}\}\right) .$$

Exponential Height Recurrence

$$E[Y_n] = E\left[\sum_{k=1}^{n} Z_{nk}\left(2 \cdot \max\{Y_{k-1}, Y_{n-k}\}\right)\right]$$

Take expectation of both sides.

Exponential Height Recurrence

$$E[Y_n] = E\left[\sum_{k=1}^{n} Z_{nk}\left(2 \cdot \max\{Y_{k-1}, Y_{n-k}\}\right)\right]$$

$$= \sum_{k=1}^{n} E\left[Z_{nk}\left(2 \cdot \max\{Y_{k-1}, Y_{n-k}\}\right)\right]$$

Linearity of expectation.

Exponential Height Recurrence

$$E[Y_n] = E\left[\sum_{k=1}^{n} Z_{nk}\left(2 \cdot \max\{Y_{k-1}, Y_{n-k}\}\right)\right]$$

$$= \sum_{k=1}^{n} E\left[Z_{nk}\left(2 \cdot \max\{Y_{k-1}, Y_{n-k}\}\right)\right]$$

$$= 2\sum_{k=1}^{n} E[Z_{nk}] \cdot E[\max\{Y_{k-1}, Y_{n-k}\}]$$

Independence of the rank of the root from the ranks of subtree roots.

Exponential Height Recurrence

$$E[Y_n] = E\left[\sum_{k=1}^{n} Z_{nk}\left(2 \cdot \max\{Y_{k-1}, Y_{n-k}\}\right)\right]$$

$$= \sum_{k=1}^{n} E\left[Z_{nk}\left(2 \cdot \max\{Y_{k-1}, Y_{n-k}\}\right)\right]$$

$$= 2\sum_{k=1}^{n} E[Z_{nk}] \cdot E[\max\{Y_{k-1}, Y_{n-k}\}]$$

$$\leq \frac{2}{n}\sum_{k=1}^{n} E[Y_{k-1} + Y_{n-k}]$$

The max of two nonnegative numbers is at most their sum, and $E[Z_{nk}] = 1/n$.

Exponential Height Recurrence

$$E[Y_n] = E\left[\sum_{k=1}^{n} Z_{nk}\left(2 \cdot \max\{Y_{k-1}, Y_{n-k}\}\right)\right]$$

$$= \sum_{k=1}^{n} E\left[Z_{nk}\left(2 \cdot \max\{Y_{k-1}, Y_{n-k}\}\right)\right]$$

$$= 2\sum_{k=1}^{n} E[Z_{nk}] \cdot E[\max\{Y_{k-1}, Y_{n-k}\}]$$

$$\leq \frac{2}{n}\sum_{k=1}^{n} E[Y_{k-1} + Y_{n-k}]$$

$$= \frac{4}{n}\sum_{k=0}^{n-1} E[Y_k]$$

Each term appears Twice – re-index.

Solving the Recurrence

Use substitution to show that $E[Y_n] \le cn^3$ for some positive constant c, which we can pick sufficiently large to handle the initial conditions.

$$E[Y_n] = \frac{4}{n} \sum_{k=0}^{n-1} E[Y_k]$$

Solving the Recurrence

Use substitution to show that $E[Y_n] \le cn^3$ for some positive constant c, which we can pick sufficiently large to handle the initial conditions.

$$E[Y_n] = \frac{4}{n}\sum_{k=0}^{n-1} E[Y_k]$$

$$\le \frac{4}{n}\sum_{k=0}^{n-1} ck^3$$

Substitution.

Solving the Recurrence

Use substitution to show that $E[Y_n] \le cn^3$ for some positive constant c, which we can pick sufficiently large to handle the initial conditions.

$$E[Y_n] = \frac{4}{n}\sum_{k=0}^{n-1} E[Y_k]$$

$$\le \frac{4}{n}\sum_{k=0}^{n-1} ck^3$$

$$\le \frac{4c}{n}\int_0^n x^3\,dx$$

Integral method.

Solving the Recurrence

Use substitution to show that $E[Y_n] \le cn^3$ for some positive constant c, which we can pick sufficiently large to handle the initial conditions.

$$E[Y_n] = \frac{4}{n}\sum_{k=0}^{n-1} E[Y_k]$$

$$\le \frac{4}{n}\sum_{k=0}^{n-1} ck^3$$

$$\le \frac{4c}{n}\int_0^n x^3 dx$$

$$= \frac{4c}{n}\left(\frac{n^4}{4}\right)$$

Compute the integral.

Solving the Recurrence

Use substitution to show that $E[Y_n] \le cn^3$ for some positive constant c, which we can pick sufficiently large to handle the initial conditions.

$$E[Y_n] = \frac{4}{n}\sum_{k=0}^{n-1} E[Y_k]$$

$$\le \frac{4}{n}\sum_{k=0}^{n-1} ck^3$$

$$\le \frac{4c}{n}\int_0^n x^3 dx$$

$$= \frac{4c}{n}\left(\frac{n^4}{4}\right)$$

$$= cn^3.$$

Algebra.

The Grand Finale

Putting it all together, we have

$$2^{E[Xn]} \leq E[2^{Xn}]$$

from Jensen's inequality, since $f(x) = 2^x$ is convex.

The Grand Finale

Putting it all together, we have

$$2^{E[X_n]} \le E[2^{X_n}]$$
$$= E[Y_n]$$

Definition.

The Grand Finale

Putting it all together, we have

$$\begin{aligned} 2^{E[X_n]} &\le E[2^{X_n}] \\ &= E[Y_n] \\ &\le cn^3 . \end{aligned}$$

What we just showed.

The Grand Finale

Putting it all together, we have

$$\begin{aligned} 2^{E[X_n]} &\le E[2^{X_n}] \\ &= E[Y_n] \\ &\le cn^3 . \end{aligned}$$

Taking the lg of both sides yields

$$E[X_n] \le 3 \lg n + O(1) .$$

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

$$\max\{a, b\} \le a + b\ .$$

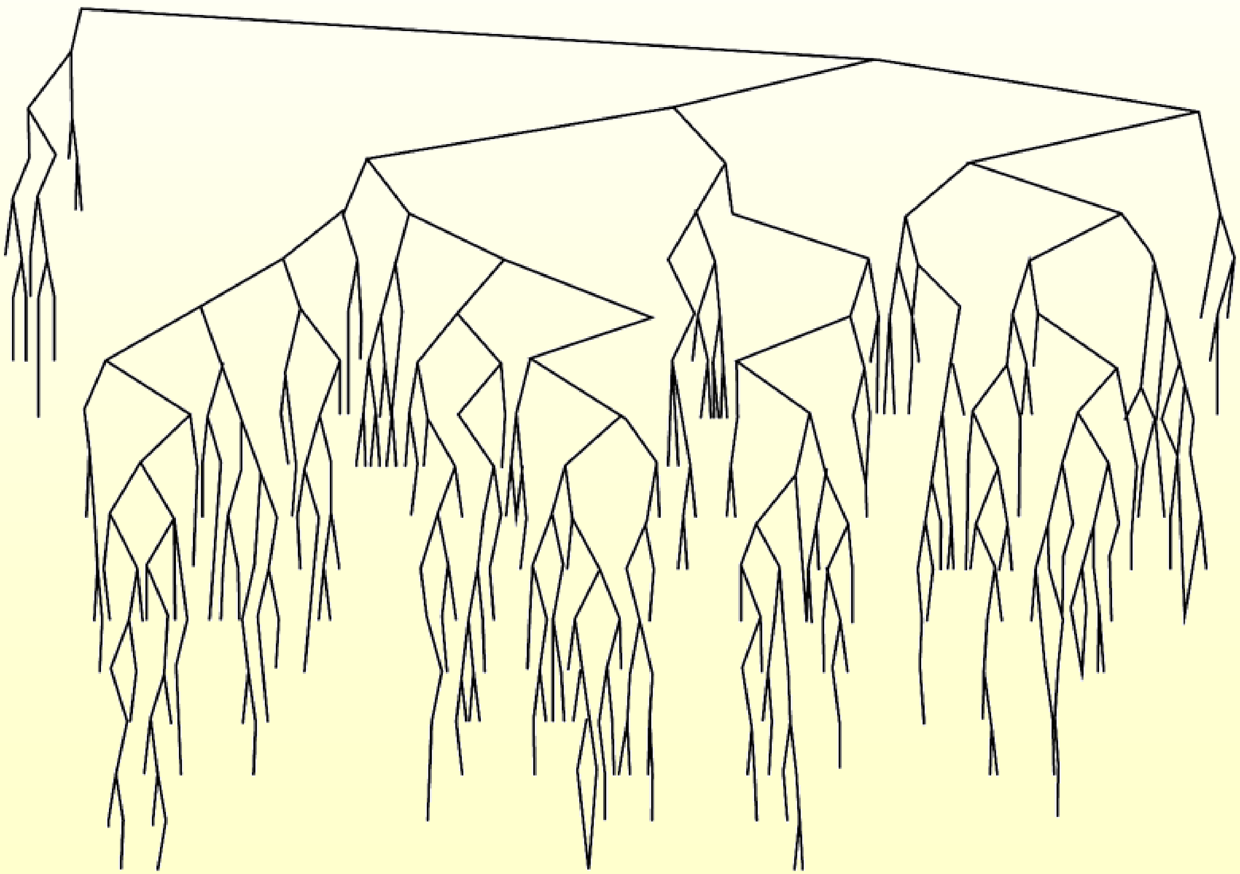
provides a poor upper bound, since the RHS approaches the LHS slowly as $|a - b|$ increases. The bound

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

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

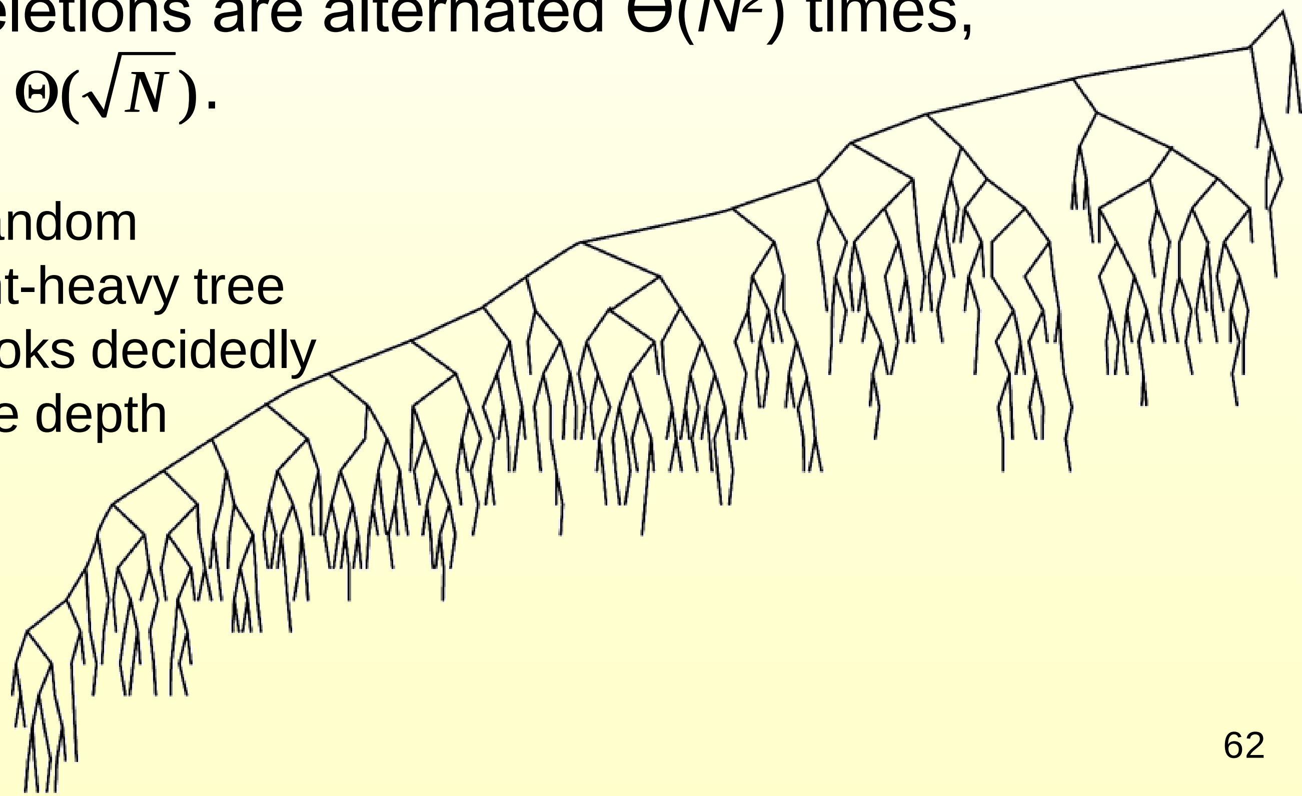
- An example of a randomly generated 500 node BST
- nodes at expected depth 9.98, height = 17.



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).