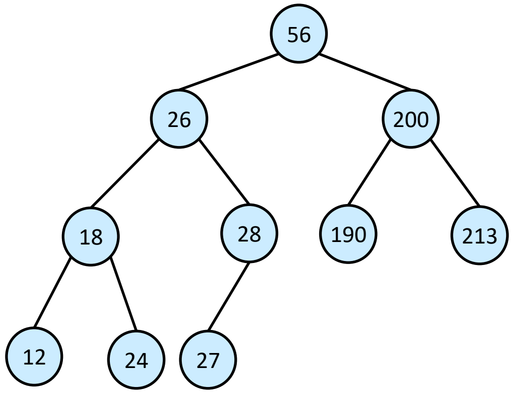
Dictionaries

Implementation Using BST, Direct Mapping, Intro to Hashing

Binary Trees

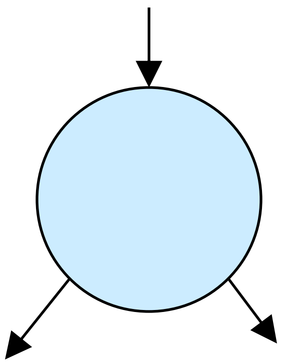
- Recursive definition
  1. An empty tree is a binary tree
  2. A node with two child subtrees is a binary tree
  3. Only what you get from 1 by a finite number of applications of 2 is a binary tree.

Is this a binary tree?



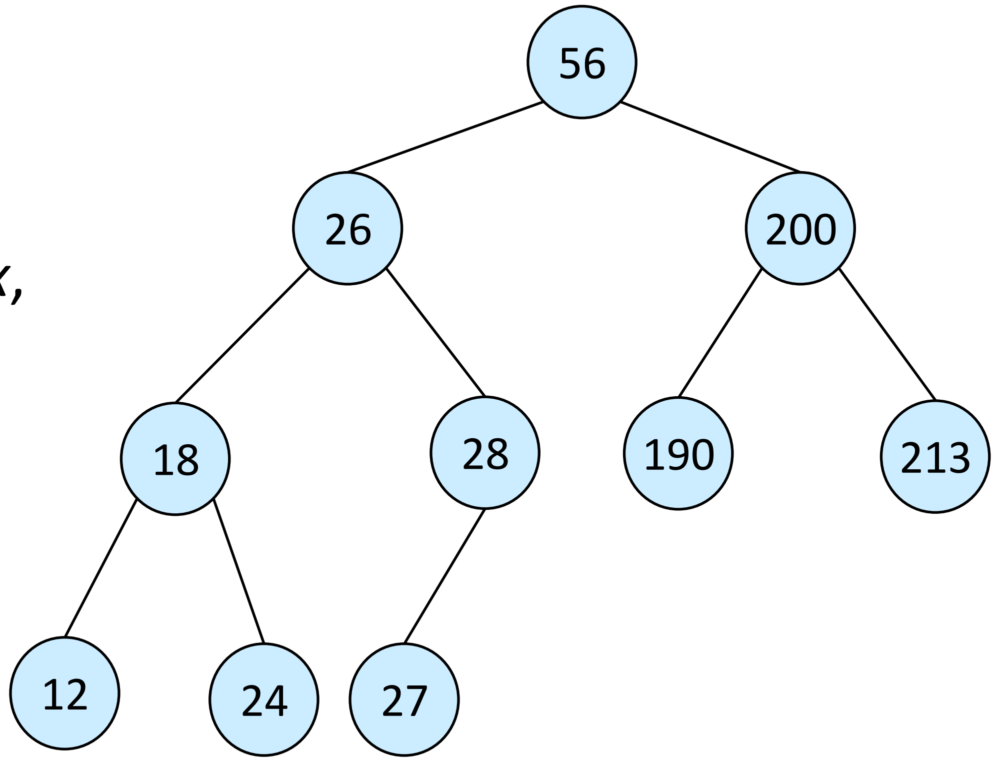
BST – Representation

- Represented by a linked data structure of nodes.
- *root*(*T*) points to the root of tree *T*.
- Each node contains fields:
  - *key*
  - *left* – pointer to left child: root of left subtree.
  - *right* – pointer to right child : root of right subtree.
  - *p* – pointer to parent. *p*[*root*[T]] = NIL (optional).



Binary Search Tree Property

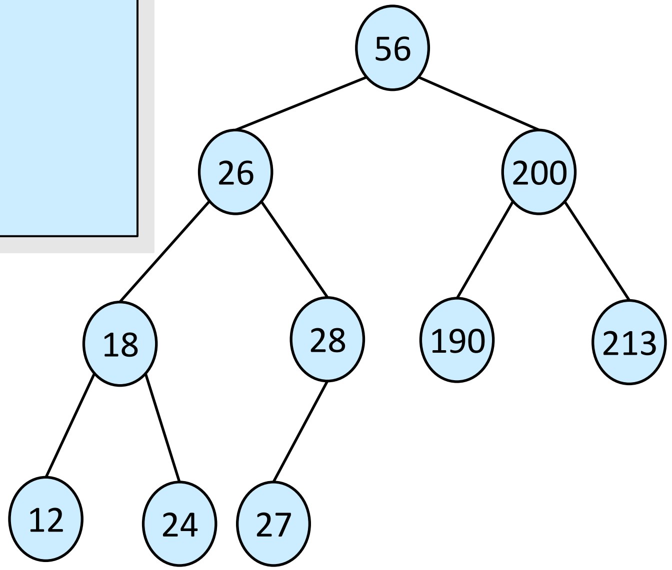
- Stored keys must satisfy the *binary search tree* property.
  - $\forall$ *y* in left subtree of *x*, then $key[y] \leq key[x]$.
  - $\forall$ *y* in right subtree of *x*, then $key[y] \geq key[x]$.



Tree Search

<u>Tree-Search(*x*, *k*)</u>

```
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)
```

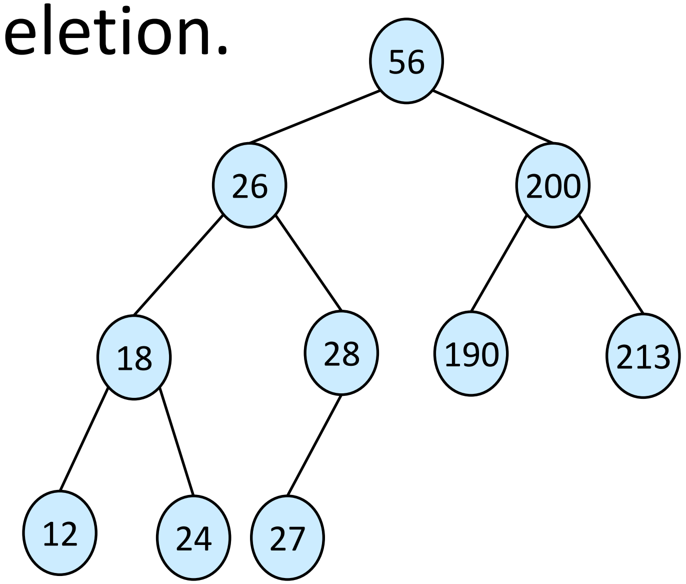


Running time: *O*(*h*)

Aside: tail-recursion

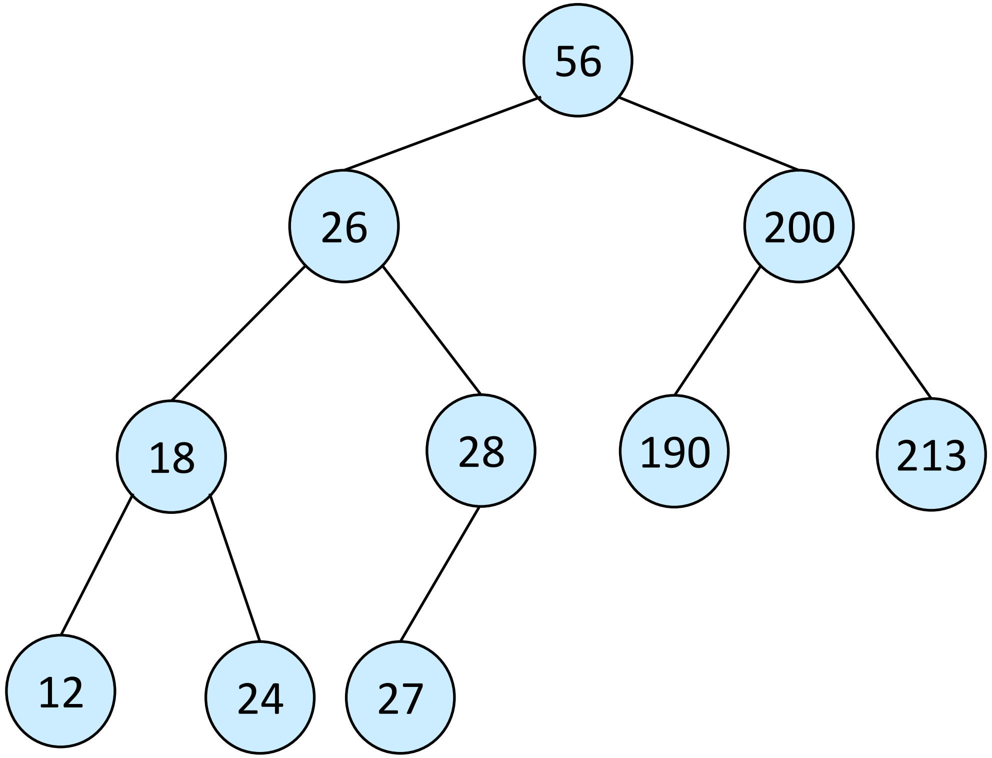
BST Insertion – Pseudocode

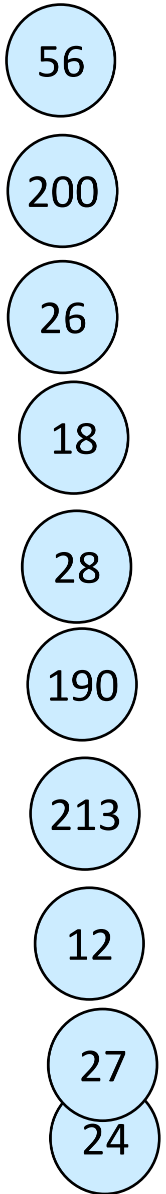
- Change the dynamic set represented by a BST.
- Ensure the binary-search-tree property holds after change.
- Insertion is easier than deletion.



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
    else if key[z] < key[y]
        then left[y] ← z
        else right[y] ← z
```


56
26
200
18
28
190
213
12
24
27


56
200
26
18
28
190
213
12
27
24

Analysis of Insertion

- Initialization: $O(1)$
- While loop in lines 3-7 searches for place to insert z, maintaining parent y.
  This takes $O(h)$ time.
- Lines 8-13 insert the value: $O(1)$

$\Rightarrow$ TOTAL: $O(h)$ time to insert a node.

```
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
11.     else if key[z] < key[y]
12.         then  left[y] ← z
13.         else right[y] ← z
```

Tree-Delete (T, x)

if x has no children ♦ case 0

then remove x

if x has one child ♦ case 1

then make $p[x]$ point to child

if x has two children (subtrees) ♦ case 2

then swap x with its successor

perform case 0 or case 1 to delete it

$\Rightarrow$ TOTAL: $O(h)$ time to delete a node

Deletion – Pseudocode

<u>Tree-Delete(T, z)</u>

/* Determine which node to splice out: either z or z's successor. */

• **if** *left*[z] = NIL **or** *right*[z] = NIL

• **then** $y \leftarrow z$

• **else** $y \leftarrow$ Tree-Successor[z]

/* Set x to a non-NIL child of x, or to NIL if y has no children. */

if *left*[y] $\neq$ NIL

then $x \leftarrow$ *left*[y]

else $x \leftarrow$ *right*[y]

/* y is removed from the tree by manipulating pointers of $p[y]$ and x */

if $x \neq$ NIL

then $p[x] \leftarrow p[y]$

/* Continued on next slide */

Deletion – Pseudocode

Tree-Delete(T, z) (Contd. from previous slide)

if $p[y]$ = NIL
    then $root[T] \leftarrow x$
    else if $y \leftarrow left[p[i]]$
        then $left[p[y]] \leftarrow x$
        else $right[p[y]] \leftarrow x$

/* If z's successor was spliced out, copy its data into z */

if $y \neq z$
    then $key[z] \leftarrow key[y]$
        copy y's satellite data into z.
return y

Binary Search Trees

- Average case and worst case Big O for
  - insertion
  - deletion
  - access
- Balance is important. Unbalanced trees give worse than log N times for the basic tree operations
- Can balance be guaranteed?

BST Average Case Analysis

For simplicity assume that keys are unique.

Assume that every permutation of n elements inserted to BST is equally likely[3] it can be proved that average height of BST is $O(logn)$.

Two cases for operations concerning a key k:

- k is not present in BST: in this case the complexities are bounded by **average height** of a BST
- k is present in BST: in this case the complexities of operations are bounded by **average depth** of a node in BST

An expected height of a random-permutation model BST can be proved to be $O(logn)$ by analogy to QuickSort (the proof is omitted in this lecture)

[3] If we assume other model: i.e. that every n-element BST is equally likely, the average height is $\Theta(\sqrt{n})$. This model seems to be less natural.

Average Depth of a Node in BST

We will explain that the average depth is $O(logn)$ (formal proof is omitted but it can be easily derived from the explanation)
For a sequence of keys $\langle k_i \rangle$ inserted to a BST define:
$G_j = \{k_i : 1 \leq i < j \text{ and } k_l > k_i > k_j \text{ for all } l < i \text{ such that } k_l > k_j\}$
$L_j = \{k_i : 1 \leq i < j \text{ and } k_l < k_i < k_j \text{ for all } l < i \text{ such that } k_l < k_j\}$
Observe, that the path from root to k_j consists exactly from $G_j \cup L_j$ so that the depth of k_j will be $d(k_j) = |G_j| + |L_j|$
G_j consists of the keys that arrived before k_j and are its direct successors (in current subsequence). The $i-th$ element in a random permutation is a current minimum with probability $1/i$. So that the expected number of updating minimum in $n-element$ random permutation is $\sum_{i=1}^{n} 1/i = H_n = O(logn)$. Being a current minimum is necessary for being a direct successor. Analogous explanations hold for L_j. So that the upper bound holds: $d(k_j) = O(logn)$.

Direct Addressing

Assume potential keys are numbers from some universe $U \subseteq N$.

An element with key $k \in U$ can be kept under index k in a $|U|$-element array:

search: O(1); insert: O(1); delete: O(1)

This is extremely fast! What is the price?

Direct Addressing

Assume potential keys are numbers from some universe $U \subseteq N$.

An element with key $k \in U$ can be kept under index k in a $|U|$-element array:

search: O(1); insert: O(1); delete: O(1)

This is extremely fast! What is the price?

n - number of elements currently kept. What is space complexity?

Direct Addressing

space complexity: O(|**U**|) ($|U|$ can be very high, even if we keep a small number of elements!)

Direct addressing is fast but waists a lot of memory (when $|U| >> n$)

Hashtables

The idea is simple.

Elements are kept in an m-element array $[0, ..., m-1]$, where $m << |U|$

The index of key is computed by fast **hash function**:

hashing function: $h : U \to [0..m-1]$

For a given key k its position is computed by $h(k)$ before each dictionary operation.