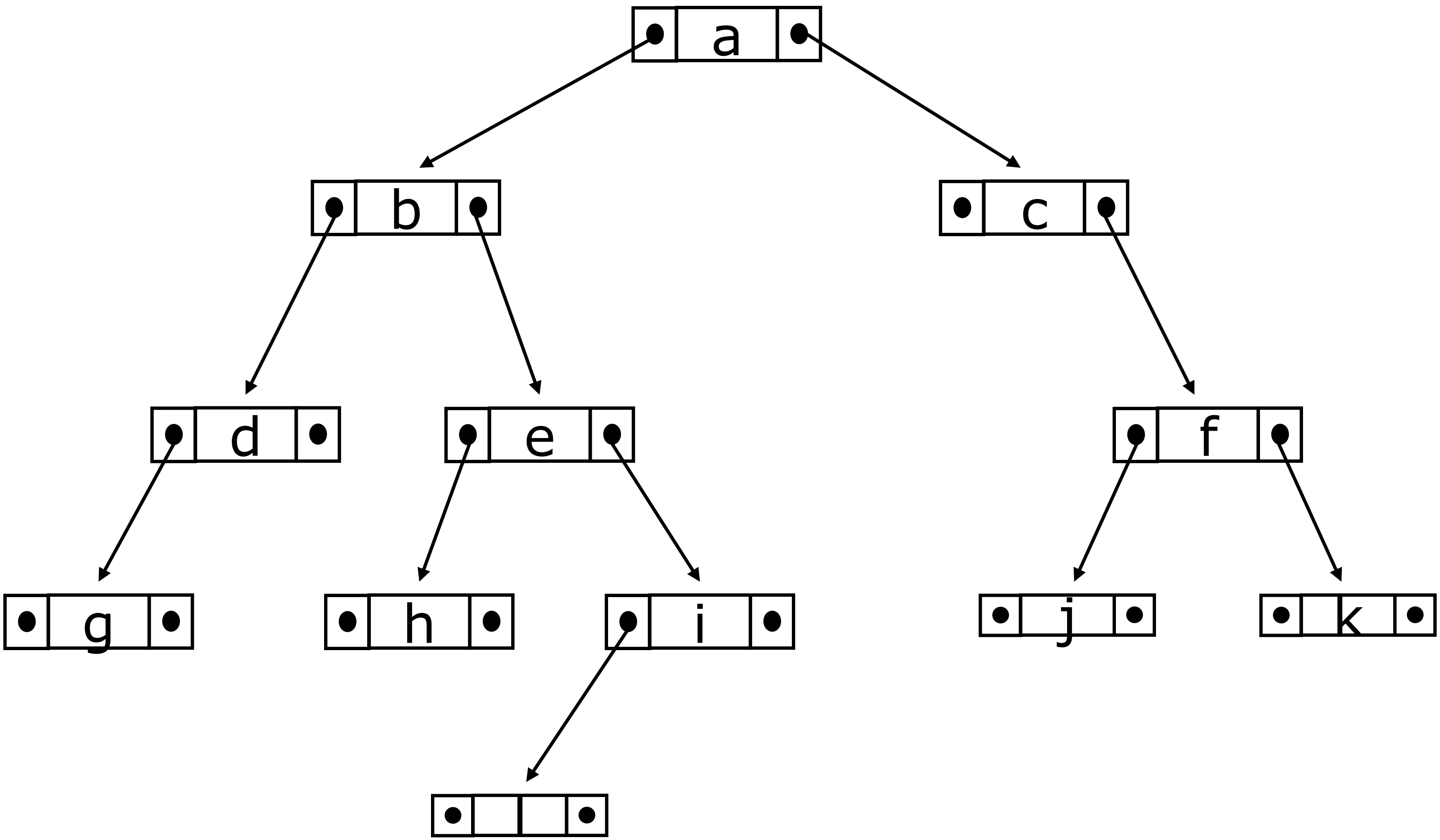
Trees

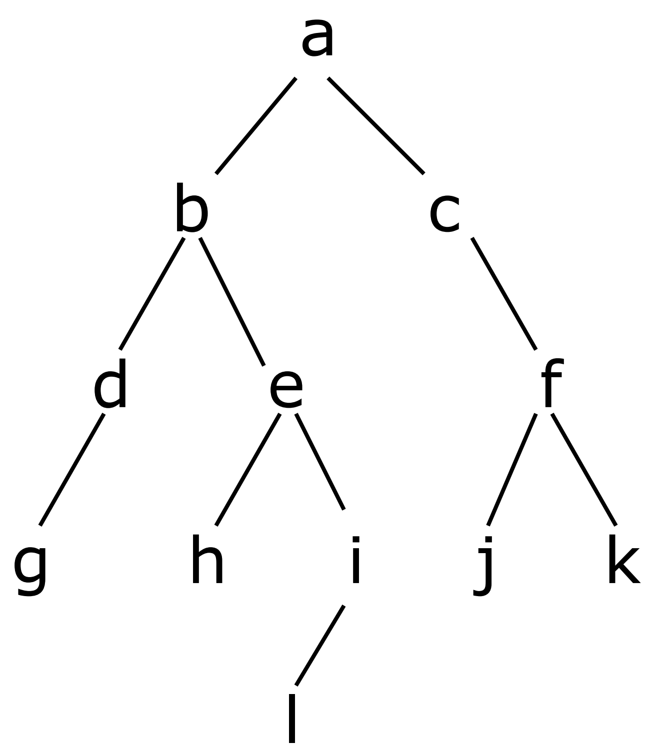
Parts of a binary tree

- A binary tree is composed of zero or more nodes
- Each node contains:
  - A value (some sort of data item)
  - A reference or pointer to a left child (may be null), and
  - A reference or pointer to a right child (may be null)
- A binary tree may be *empty* (contain no nodes)
- If not empty, a binary tree has a root node
  - Every node in the binary tree is reachable from the root node by a *unique* path
- A node with neither a left child nor a right child is called a leaf
  - In some binary trees, only the leaves contain a value

Picture of a binary tree

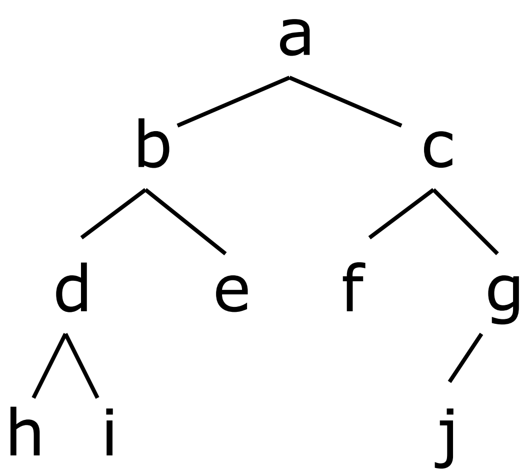


Size and depth

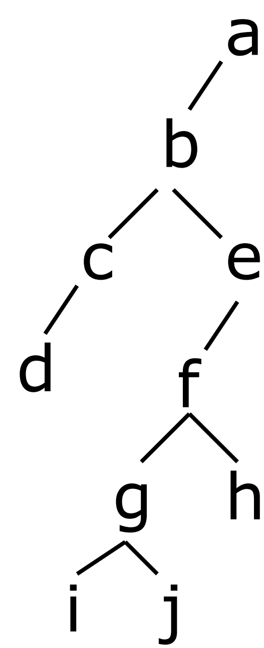


- The size of a binary tree is the number of nodes in it
  – This tree has size 12
- The depth of a node is its distance from the root
  – a is at depth zero
  – e is at depth 2
- The depth of a binary tree is the depth of its deepest node
  – This tree has depth 4

Balance



A balanced binary tree

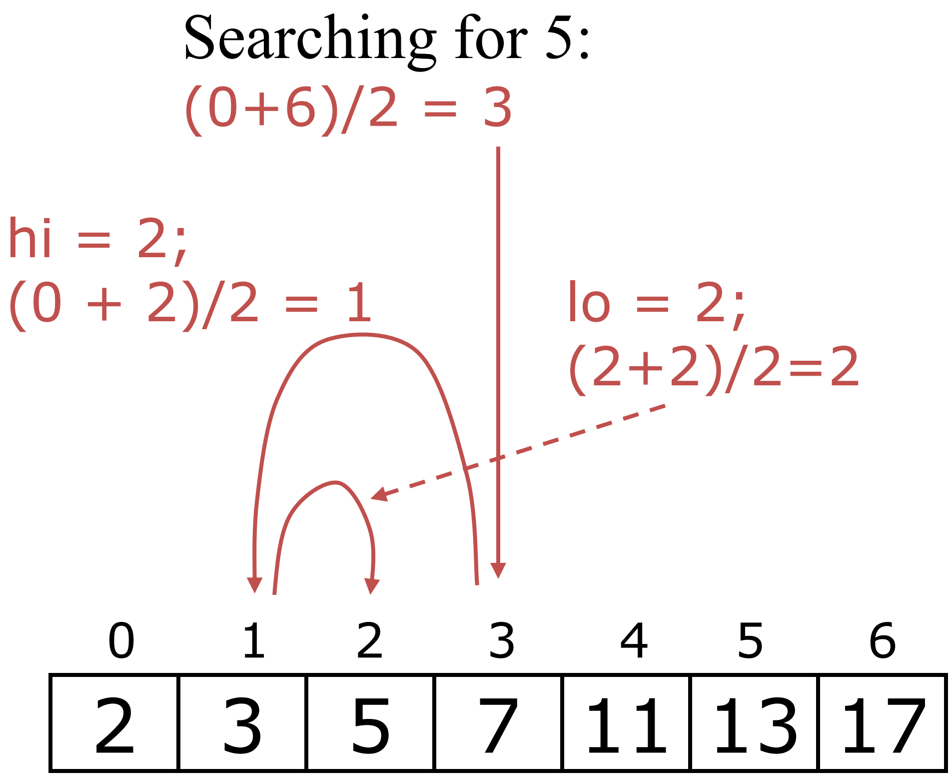


An unbalanced binary tree

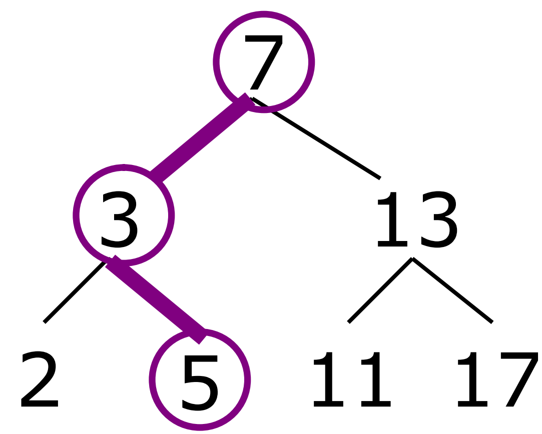
- A binary tree is balanced if every level above the lowest is “full” (contains 2^n nodes)
- In most applications, a reasonably balanced binary tree is desirable

Binary search in an array

- Look at array location (lo + hi)/2



Using a binary search tree

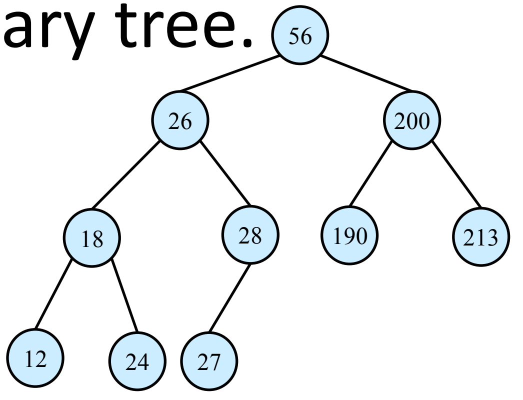


Binary Search Trees

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?

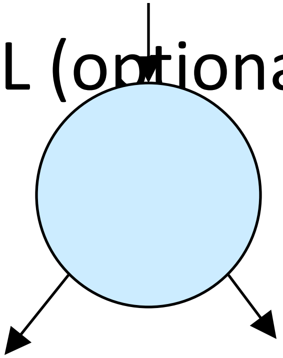


Binary Search Trees

- View today as data structures that can support dynamic set operations.
  - Search, Minimum, Maximum, Predecessor, Successor, Insert, and Delete.
- Can be used to build
  - Dictionaries.
  - Priority Queues.
- Basic operations take time proportional to the height of the tree – $O(h)$.

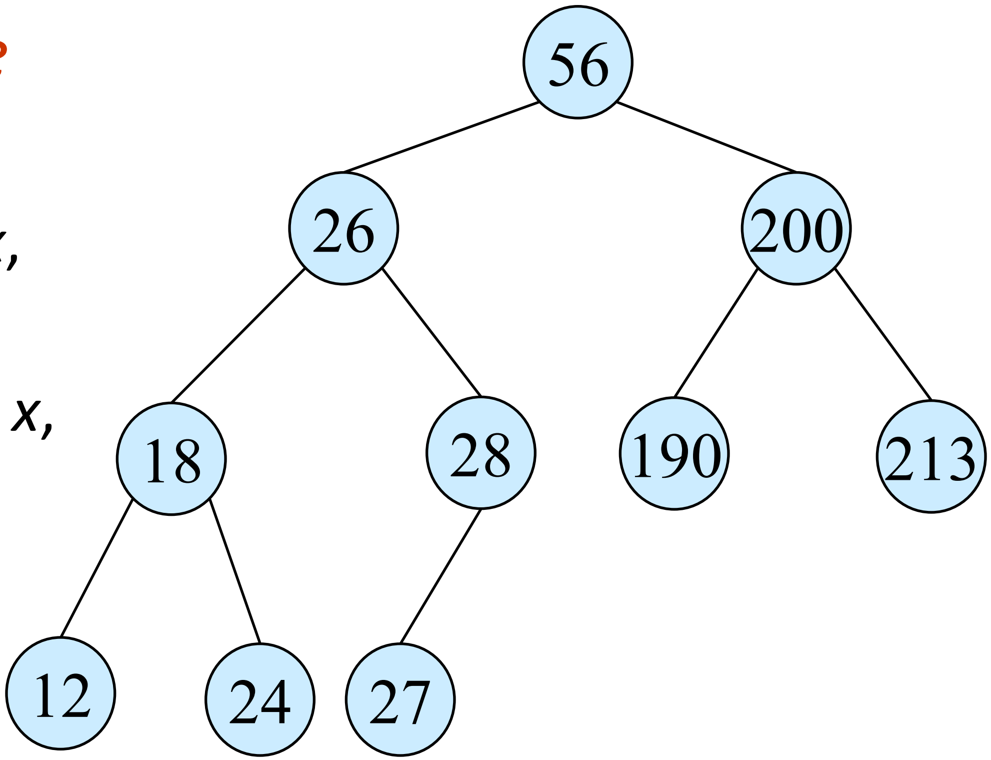
BST – Representation

- 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*].

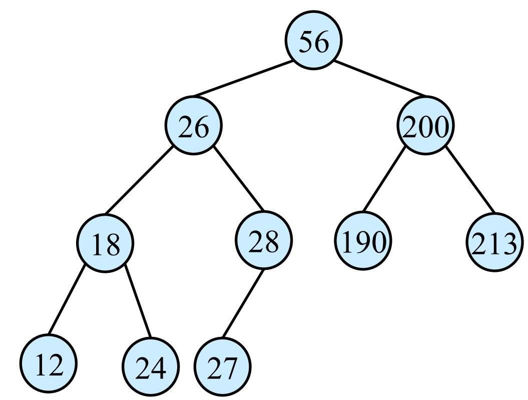


Inorder Traversal

The binary-search-tree property allows the keys of a binary search tree to be printed, in (monotonically increasing) order, recursively.

<u>Inorder-Tree-Walk (x)</u>

1. **if** $x \neq$ NIL
2. **then** Inorder-Tree-Walk(*left*[*p*])
3. print *key*[*x*]
4. Inorder-Tree-Walk(*right*[*p*])



- How long does the walk take?
- Can you prove its correctness?

Correctness of Inorder-Walk

- Must prove that it prints all elements, in order, and that it terminates.
- By induction on size of tree. Size=0: Easy.
- Size >1:
  - Prints left subtree in order by induction.
  - Prints root, which comes after all elements in left subtree (still in order).
  - Prints right subtree in order (all elements come after root, so still in order).

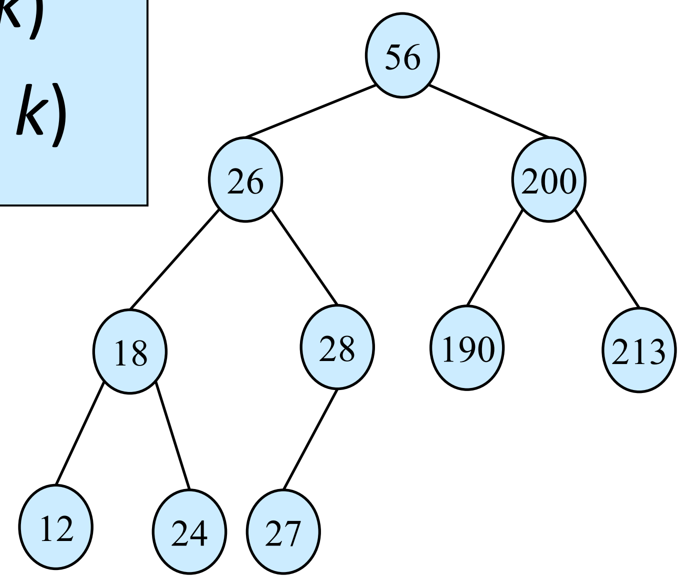
Querying a Binary Search Tree

- All dynamic-set search operations can be supported in $O(h)$ time.
- $h = \Theta(lg\ n)$ for a balanced binary tree (and for an average tree built by adding nodes in random order.)
- $h = \Theta(n)$ for an unbalanced tree that resembles a linear chain of n nodes in the worst case.

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)
```



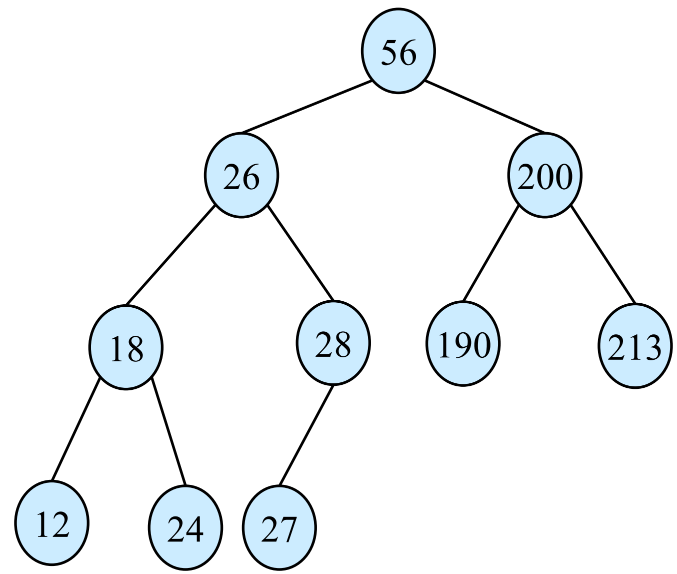
Running time: *O*(*h*)

Aside: tail-recursion

Iterative Tree Search

Iterative-Tree-Search(*x*, *k*)

```
while x ≠ NIL and k ≠ key[x]
    do if k < key[x]
        then x ← left[x]
        else x ← right[x]
return x
```



The iterative tree search is more efficient on most computers.
The recursive tree search is more straightforward.

Finding Min & Max

- The binary-search-tree property guarantees that:
  - » The minimum is located at the left-most node.
  - » The maximum is located at the right-most node.

Tree-Minimum(x)

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

Tree-Maximum(x)

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

Q: How long do they take?

Predecessor and Successor

- Successor of node *x* is the node *y* such that *key*[*y*] is the smallest key greater than *key*[*x*].
- The successor of the largest key is NIL.
- Search consists of two cases.
  - If node *x* has a non-empty right subtree, then *x*'s successor is the minimum in the right subtree of *x*.
  - If node *x* has an empty right subtree, then:
    - As long as we move to the left up the tree (move up through right children), we are visiting smaller keys.
    - *x*'s successor *y* is the node that *x* is the predecessor of (*x* is the maximum in *y*'s left subtree).
    - In other words, *x*'s successor *y*, is the lowest ancestor of *x* whose left child is also an ancestor of *x*.

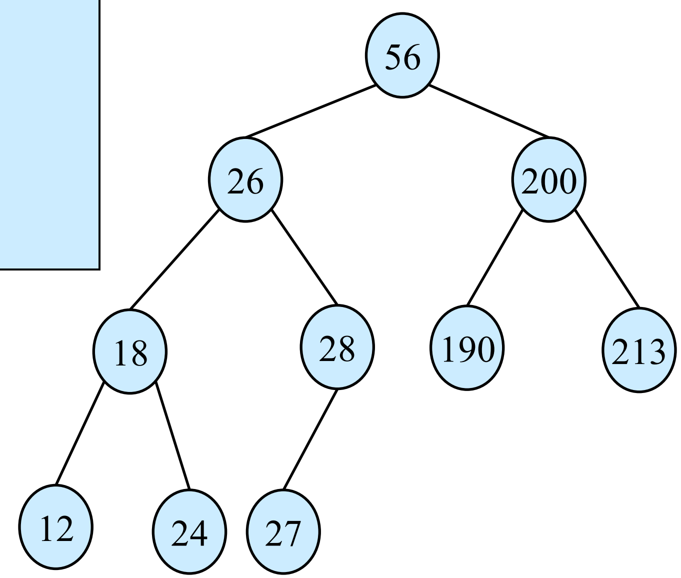
Pseudo-code for Successor

<u>Tree-Successor(*x*)</u>

```
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
```

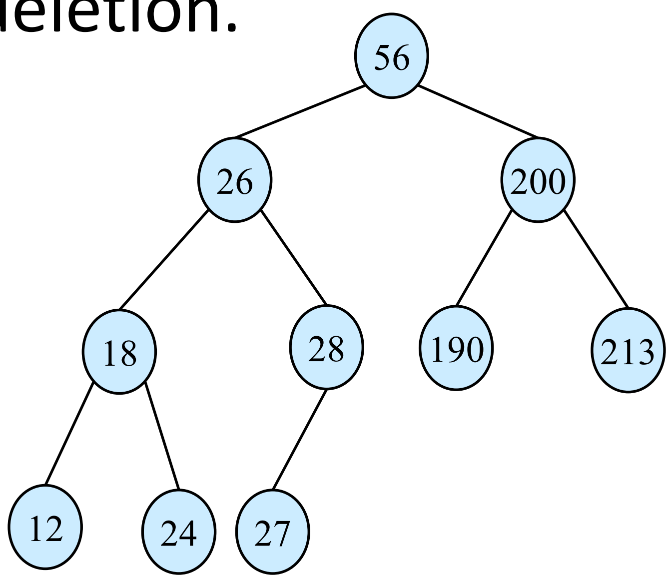
Code for ***predecessor*** is symmetric.

Running time: $O(h)$



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
```

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: $O(1)$

⇒ 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
```

Exercise: Sorting Using BSTs

Sort (A)

for $i \leftarrow 1$ to n

do tree-insert($A[i]$)

inorder-tree-walk(*root*)

- What are the worst case and best case running times?
- In practice, how would this compare to other sorting algorithms?

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

```
Tree-Delete(T, z)
/* Determine which node to splice out: either z or z's successor.
   */
    if left[z] = NIL or right[z] = NIL
       then y ← z
       else y ← Tree-Successor[z]
/* Set x to a non-NIL child of x, or to NIL if y has no children. */
    if  left[y] ≠ NIL
        then x ← left[y]
        else x ← right[y]
/* y is removed from the tree by manipulating pointers of  p[y]
   and x */
    if x ≠ NIL
       then p[x] ← p[y]
/* Continued on next slide */
```

Deletion – Pseudocode

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

```
if p[y] = NIL
    then root[T] ← x
    else if  y ← left[p[i]]
        then left[p[y]] ← x
        else right[p[y]] ← x
/* If z’s successor was spliced out, copy its data into z */
if y ≠ z
    then  key[z] ← key[y]
          copy y’s satellite data into z.
return y
```

Correctness of Tree-Delete

- How do we know case 2 should go to case 0 or case 1 instead of back to case 2?
  - Because when *x* has 2 children, its successor is the minimum in its right subtree, and that successor has no left child (hence 0 or 1 child).
- Equivalently, we could swap with predecessor instead of successor. It might be good to alternate to avoid creating lopsided tree.

Binary Search Trees

- View today as data structures that can support dynamic set operations.
  - Search, Minimum, Maximum, Predecessor, Successor, Insert, and Delete.
- Can be used to build
  - Dictionaries.
  - Priority Queues.
- Basic operations take time proportional to the height of the tree – $O(h)$.

Tree traversals

- A binary tree is defined recursively: it consists of a root, a left subtree, and a right subtree
- To traverse (or walk) the binary tree is to visit each node in the binary tree exactly once
- Tree traversals are naturally recursive
- Since a binary tree has three "parts," there are six possible ways to traverse the binary tree:
  - root, left, right
  - left, root, right
  - left, right, root
  - root, right, left
  - right, root, left
  - right, left, root

Preorder traversal

- In preorder, the root is visited *first*
- Here's a preorder traversal to print out all the elements in the binary tree:

```
public void preorderPrint(BinaryTree bt) {
    if (bt == null) return;
    System.out.println(bt.value);
    preorderPrint(bt.leftChild);
    preorderPrint(bt.rightChild);
}
```

Inorder traversal

- In inorder, the root is visited *in the middle*
- Here's an inorder traversal to print out all the elements in the binary tree:

```
public void inorderPrint(BinaryTree bt) {
    if (bt == null) return;
    inorderPrint(bt.leftChild);
    System.out.println(bt.value);
    inorderPrint(bt.rightChild);
}
```

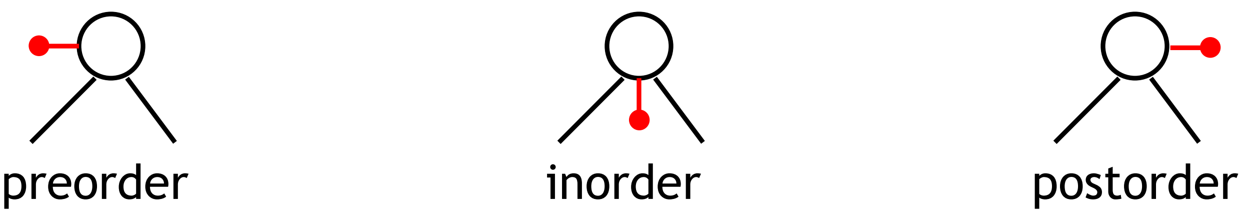
Postorder traversal

- In postorder, the root is visited *last*
- Here's a postorder traversal to print out all the elements in the binary tree:

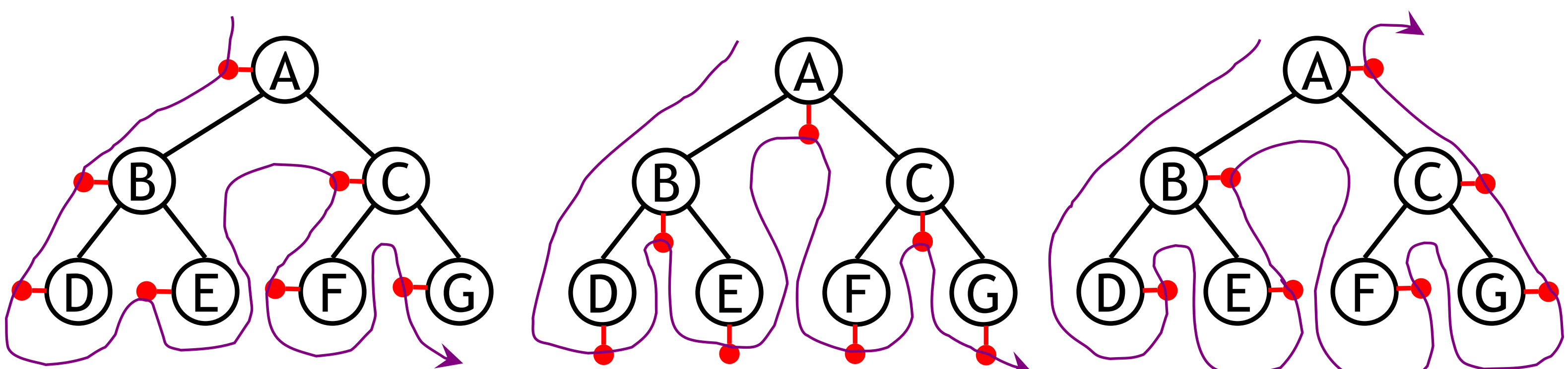
```
public void postorderPrint(BinaryTree bt) {
    if (bt == null) return;
    postorderPrint(bt.leftChild);
    postorderPrint(bt.rightChild);
    System.out.println(bt.value);
}
```

Tree traversals using "flags"

- The order in which the nodes are visited during a tree traversal can be easily determined by imagining there is a "flag" attached to each node, as follows:



- To traverse the tree, collect the flags:



A B D E C F G   D B E A F C G   D E B F G C A

Copying a binary tree

- In postorder, the root is visited *last*
- Here's a postorder traversal to make a complete copy of a given binary tree:

```
public BinaryTree copyTree(BinaryTree bt) {
    if (bt == null) return null;
    BinaryTree left = copyTree(bt.leftChild);
    BinaryTree right = copyTree(bt.rightChild);
    return new BinaryTree(bt.value, left, right);
}
```

Other traversals

- The other traversals are the reverse of these three standard ones
  - That is, the right subtree is traversed before the left subtree is traversed
- Reverse preorder: root, right subtree, left subtree
- Reverse inorder: right subtree, root, left subtree
- Reverse postorder: right subtree, left subtree, root