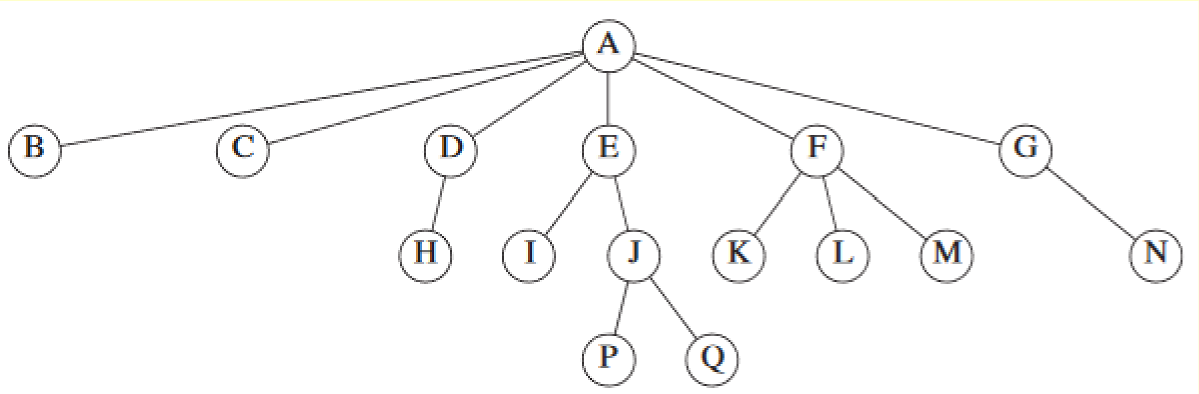
Binary Search Trees

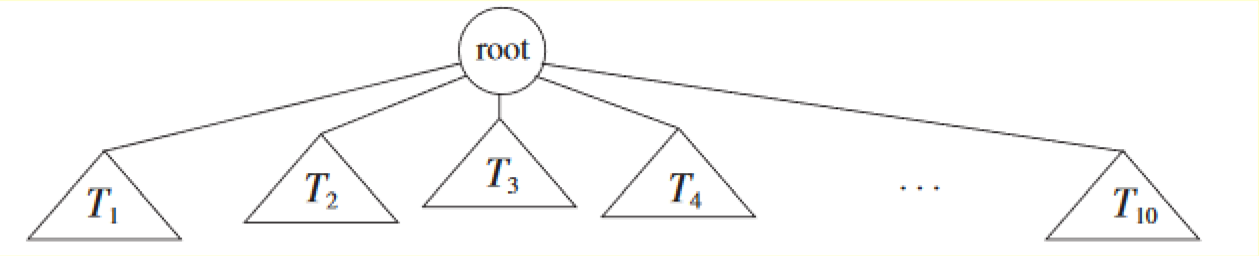
Modified version of
*https://www.cs.unc.edu/~plaisted/comp550/15-b****trees.ppt***

- *For large amounts of input, the linear access time of linked lists is prohibitive* – weiss

  Forms of trees(balanced BST, 2-3 trees) provide an average running time of O(log n) for most operations.



Root node: A
Child of D: H
Parent of D: A



Each node has
a subtree under it.

Depth: The depth of a node is the number of edges in the unique path from the root to the node. Root depth=0

*Depth of a tree is equal to the depth of the deepest leaf.

Height: The height of a node is the length of the longest path from node to a leaf. All leaves are at height 0.
*The height of a tree is equal to the height of the root.

Depth of a tree is equal to the height of the tree.

- **Ancestor-Descendant:** If there is a path from n1 to n 2,then n1 is an ancestor of n2 and n2 is a descendant of n1. If n1 = n2,then n1 is a proper ancestor of n2 and n2 is a proper descendant of n1

Implementation

- When number of children is not known

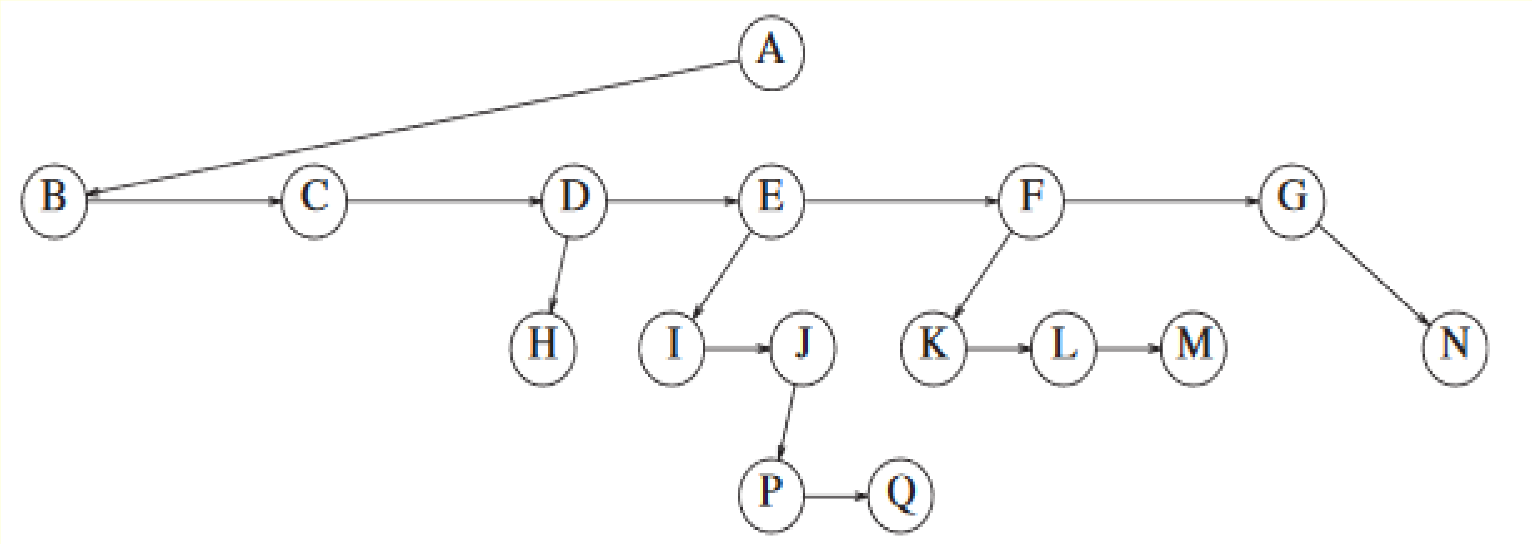
  Struct TreeNode

{ char * string;

TreeNode *firstChild;

TreeNode *nextSibling;

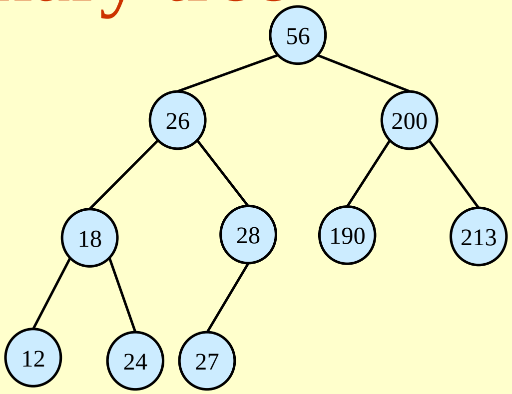
}



Binary Trees

- Recursive definition
  1. An empty tree is a binary tree
  2. A node can have not more than two children.

binary tree



Binary Tree Implementation

- When number of children is atmost two.

```
Struct TreeNode
{ char * string;
TreeNode *firstChild;
TreeNode *secondChild;
}
```

- Max depth of a binary tree ?
- Binary Tree you know of..... Expression tree used to derive postfix, prefix notations.

More definitions

- A **full binary tree** (sometimes proper **binary tree** or 2**-tree**) is a **tree** in which every node other than the leaves has two children.

  A **complete binary tree** is a **binary tree** in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

- Inorder : Left Root Right
- Preorder : Root Left Right
- Postorder: Left Right Root

For a binary tree T, let the preorder traversal be [1, 2, 7, 3, 4, 5, 6] and inorder traversal be [2, 7, 1, 4, 3, 6, 5]. Can you construct the tree T.

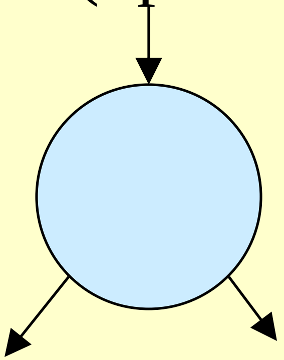
For a FULL binary tree T, let the preorder traversal be [1, 2, 3, 4, 5, 6, 7] and postorder traversal be [2, 4, 6, 7, 5, 3, 1] . Construct the full binary tree T.

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 – $O(h)$.

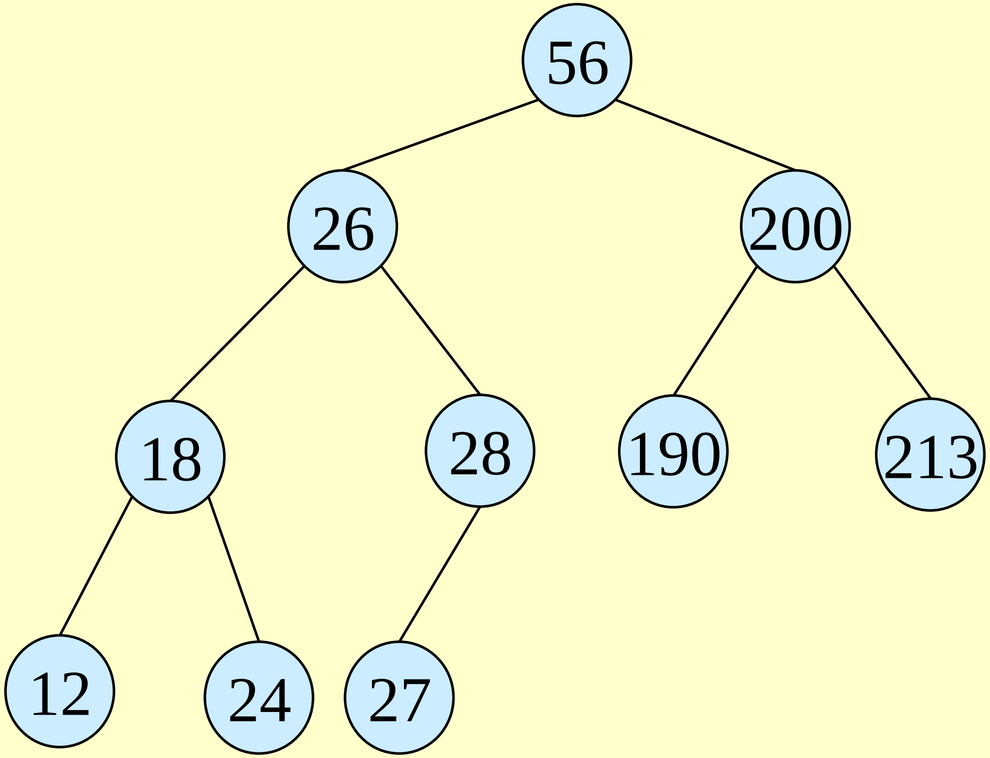
BST – Representation

- 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 $key[y] \le key[x]$.
  - » $\forall$ *y* in right subtree of *x*, then $key[y] \ge key[x]$.



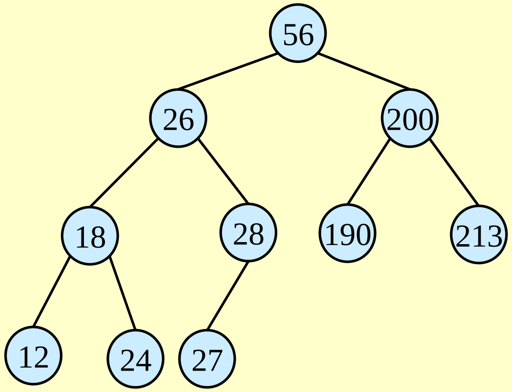
Given a BST, how can I Print the sorted element set?

Inorder Traversal

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

Inorder-Tree-Walk (x)

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?

Operations

- Querying for a key
- Find Min, Max
- Find Successor, Predecessor
- Insert, Delete

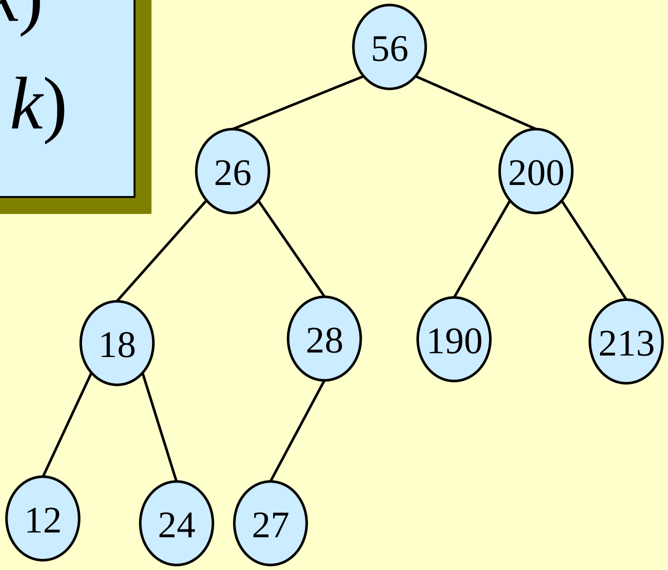
Querying a Binary Search Tree

- 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

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)



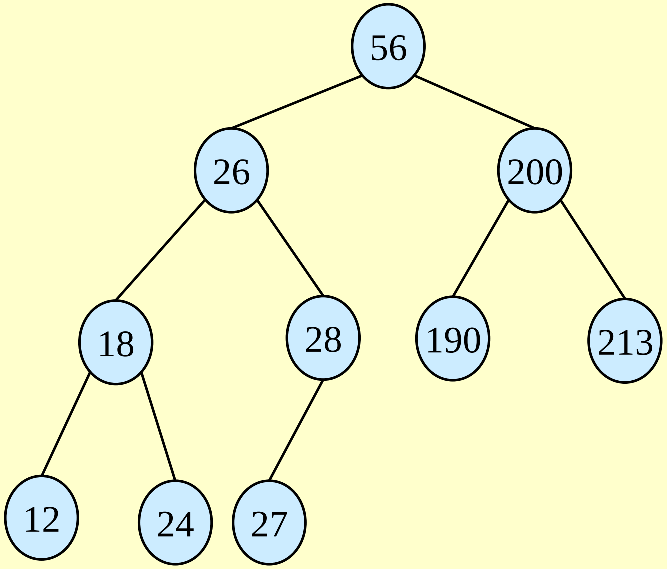
Running time: $O(h)$

Aside: tail-recursion

Iterative Tree Search

Iterative-Tree-Search(x, k)

1. **while** $x \neq NIL$ **and** $k \neq key[x]$
2. **do if** $k < key[x]$
3. **then** $x \leftarrow left[x]$
4. **else** $x \leftarrow right[x]$
5. **return** x



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

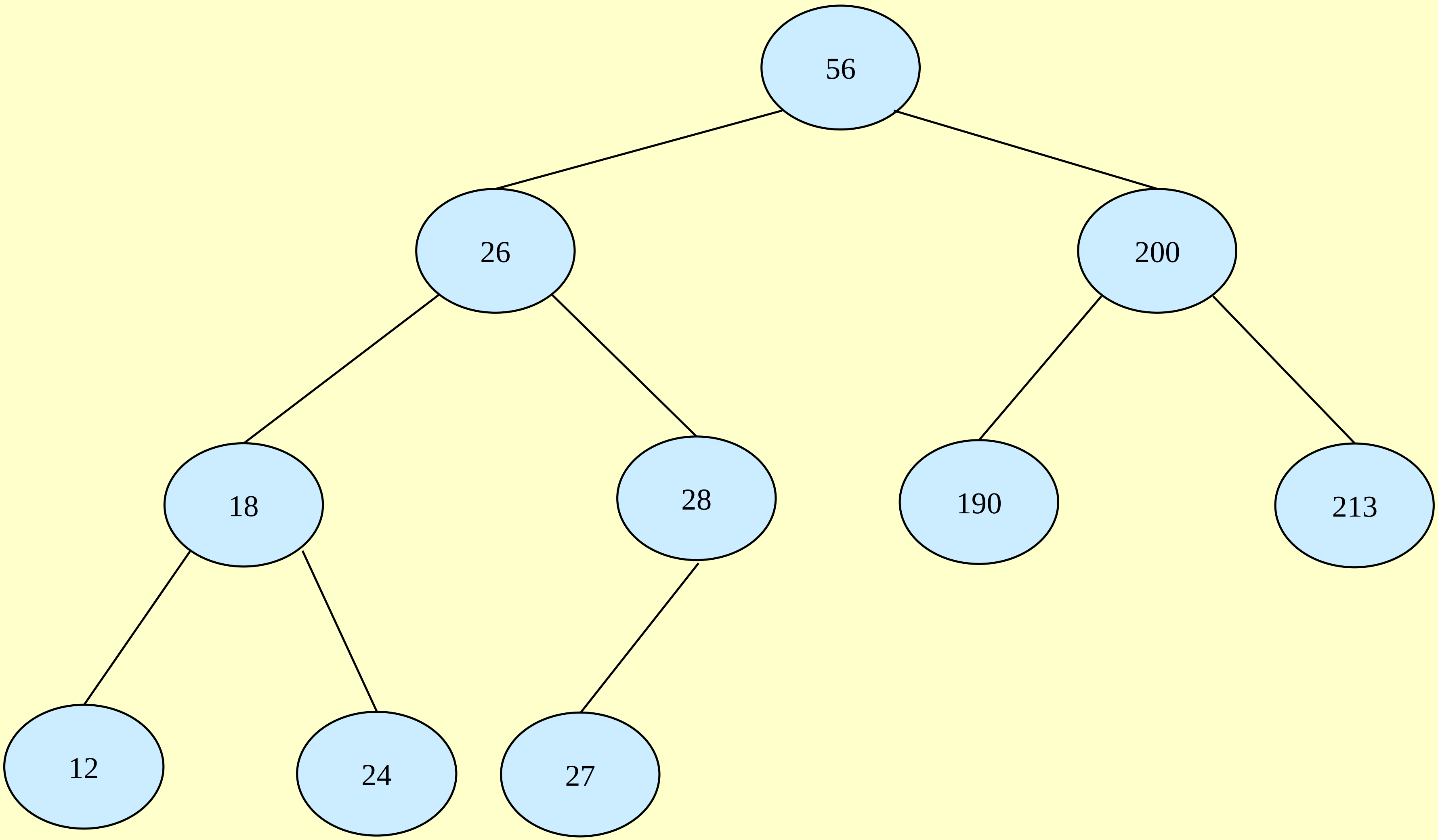
Finding Min & Max

- 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)	Tree-Maximum(x)
1. **while** $left[x] \neq NIL$	1. **while** $right[x] \neq NIL$
2. **do** $x \leftarrow left[x]$	2. **do** $x \leftarrow right[x]$
3. **return** x	3. **return** x

Q: How long do they take?

Predecessor and Successor



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*.

Predecessor and Successor

- If node *x* has a non-empty right subtree, then *x*'s successor is the minimum in the right subtree of *x*. Why ?
  Can it be anywhere else
- A. cant be on the left subtree
- If there is a right subtree, then any ancestor a(x) either has value greater than elements in the right subtree or smaller than x. So successor has to be in the right subtree if there is a right subtree.
- If no right subtree, then,
  -- if x is in left subtree of root, then root has higher value than x. so the successor of x is either root or some ancestor of x on the left of root. If I move left to an ancestor it will be smaller. The first right ancestor y is bigger than x. An even higher ansestor is even bigger than y. so y is the successor.

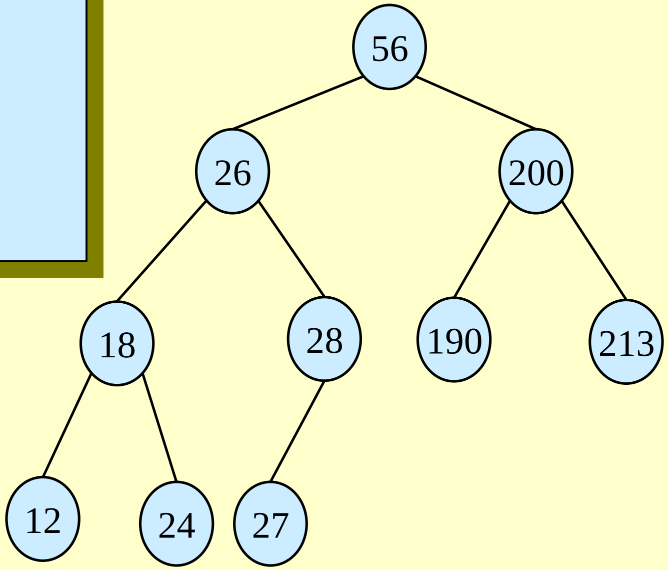
  -- if x is in right subtree of root.

Pseudo-code for Successor

Tree-Successor(x)

- **if** $right[x] \neq NIL$

2. **then** return Tree-Minimum($right[x]$)
3. $y \leftarrow p[x]$
4. **while** $y \neq NIL$ **and** $x = right[y]$
5. **do** $x \leftarrow y$
6. $y \leftarrow p[y]$
7. **return** y

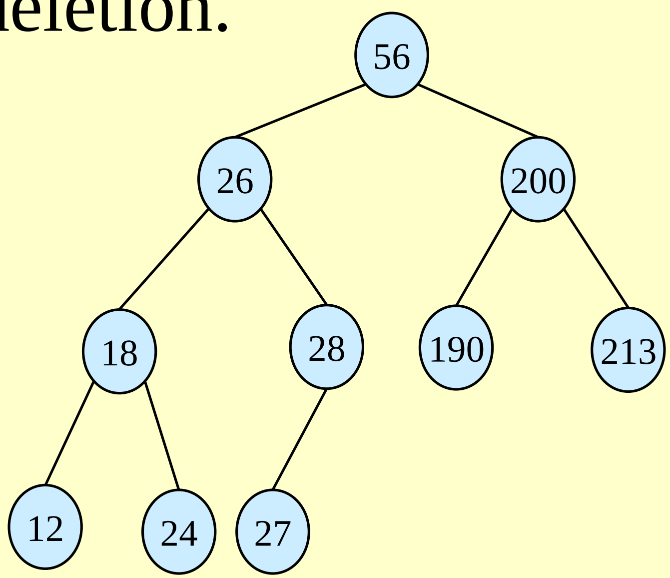


Code for ***predecessor*** is symmetric.

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 (inserting first node)
    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: $O(1)$

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

Tree-Insert(T, z)

1. $y \leftarrow$ NIL
2. $x \leftarrow root[T]$
3. **while** $x \neq$ NIL
4.   **do** $y \leftarrow x$
5.   **if** $key[z] < key[x]$
6.   **then** $x \leftarrow left[x]$
7.   **else** $x \leftarrow right[x]$
8. $p[z] \leftarrow y$
9. **if** $y =$ NIL
10.   **then** $root[t] \leftarrow z$
11.   **else if** $key[z] < key[y]$
12.   **then** $left[y] \leftarrow z$
13.   **else** $right[y] \leftarrow z$

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

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

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

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

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

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

7. **if** $x \neq$ NIL

8. **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

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 – $O(h)$.

More definitions

- A **full binary tree** (sometimes proper **binary tree** or 2-**tree**) is a **tree** in which every node other than the leaves has two children.

  A **complete binary tree** is a **binary tree** in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

Full binary tree(FBT)

- How many nodes in a FBT of height H ?
- How many leaf nodes (L) in FBT of n nodes?
- Number of internal nodes(I) in FBT of n nodes?
- If n nodes then what is the max height ?

- Question:

  I= L-1 ?

  L= (n+1)/2

  $H=\log_2(L)$

Full binary tree(FBT)

- Max nodes in a FBT of height H ? $2^{H+1} - 1$
- If n nodes then what is the min height ? Log(N+1/2)
- Max leaf nodes (L) in FBT of n nodes? 2^{H} (induction)
- Number of internal nodes(I) in FBT of n nodes? $2^{H}-1$ (induction)

- When, I= L-1 and I+ L= n , show L= (n+1)/2

$$I+L=n$$
$$L-1+L=n$$
$$L=(n+1)/2$$

binary tree(BT)

- Max and min height ?
- At most how many nodes are present in a BT of height H ?
- Minimum height of BT with n nodes?
- Prove L<= I+1 using induction?
- At most how many leaf nodes (L) in a tree of n nodes?

Full binary tree(FBT)

- At most how many nodes are present in a FBT of height H ?
  (max nodes when FBT) 2^{h+1} -1

- What is the minimum height of a tree of n nodes?
  N< maximum possible nodes=2^{h+1} -1

  Thus h>=log(n+1)/2

- At most how many leaf nodes (L)?
  L<= I+1 (Prove using induction)
  L+I=n

  Thus, L <= n-L+1

  =>L<= (n+1)/2

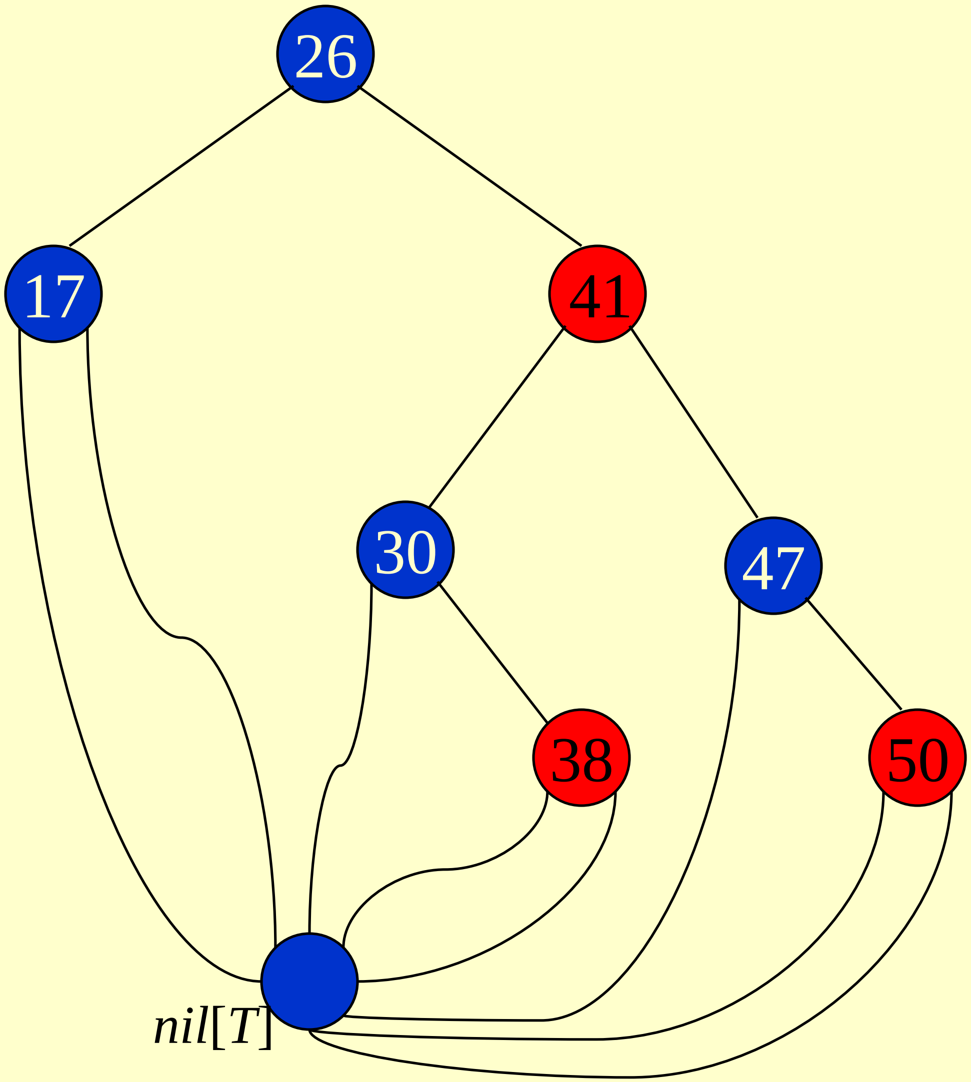
Red-black trees: Overview

- Red-black trees are a variation of binary search trees to ensure that the tree is ***balanced***.
  - » Height is $O(\lg n)$, where n is the number of nodes.
- Operations take $O(\lg n)$ time in the worst case.

Red-black Tree

- Binary search tree + 1 bit per node: the attribute *color*, which is either **red** or **black**.
- All other attributes of BSTs are inherited:
  - » *key*, *left*, *right*, and *p*.
- All empty trees (leaves) are colored black.
  - » We use a single sentinel, *nil*, for all the leaves of red-black tree *T*, with *color*[*nil*] = black.
  - » The root's parent is also *nil*[*T*].

Red-black Tree – Example



Red-black Properties

1. Every node is either red or black.
2. The root is black.
3. Every leaf (*nil*) is black.
4. If a node is red, then both its children are black.

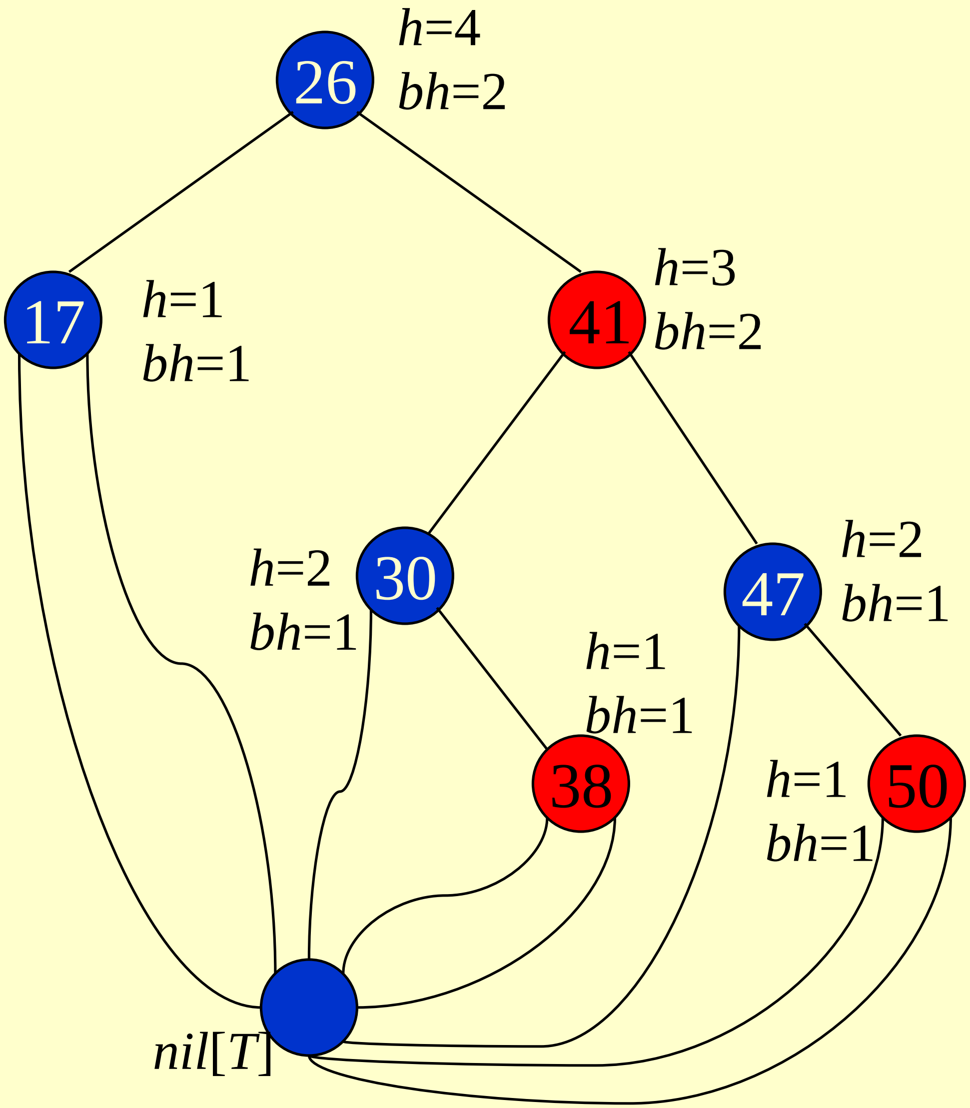
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

Height of a Red-black Tree

- Height of a node:
  - » Number of edges in a longest path to a leaf.
- Black-height of a node x, $bh(x)$:
  - » $bh(x)$ is the number of black nodes (including $nil[T]$) on the path from x to leaf, not counting x.
- Black-height of a red-black tree is the black-height of its root.
  - » By Property 5, black height is well defined.

Height of a Red-black Tree

- Example:
- Height of a node:
  - » Number of edges in a longest path to a leaf.
- Black-height of a node $bh(x)$ is the number of black nodes on path from x to leaf, not counting x.



Hysteresis : or the value of lazyness

- **Hysteresis**, n. [fr. Gr. to be behind, to lag.] a retardation of an effect when the forces acting upon a body are changed (as if from viscosity or internal friction); *especially*: a lagging in the values of resulting magnetization in a magnetic material (as iron) due to a changing magnetizing force