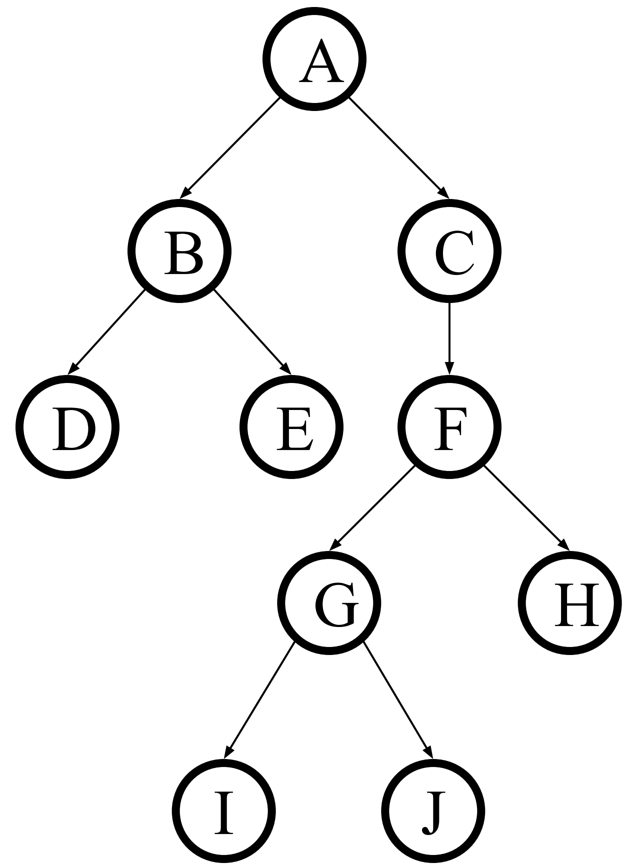
Binary Search Trees

SDP4

Binary Trees

Binary tree is

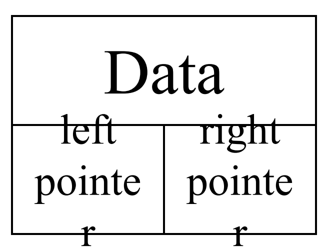
- a root
- left subtree *(maybe empty)*
- right subtree *(maybe empty)*



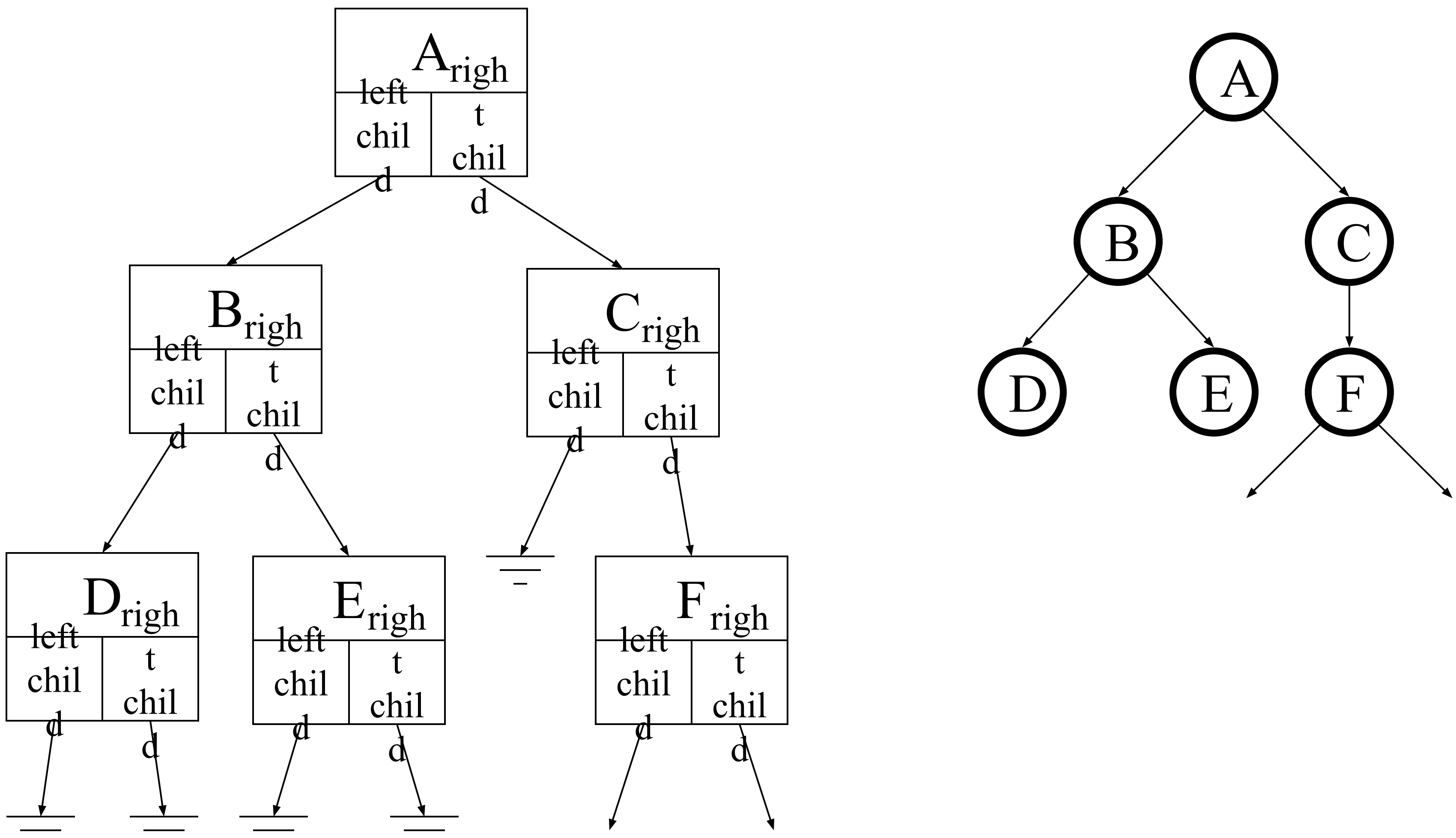
Properties

- max # of leaves:
- max # of nodes:
- average depth for N nodes:

Representation:



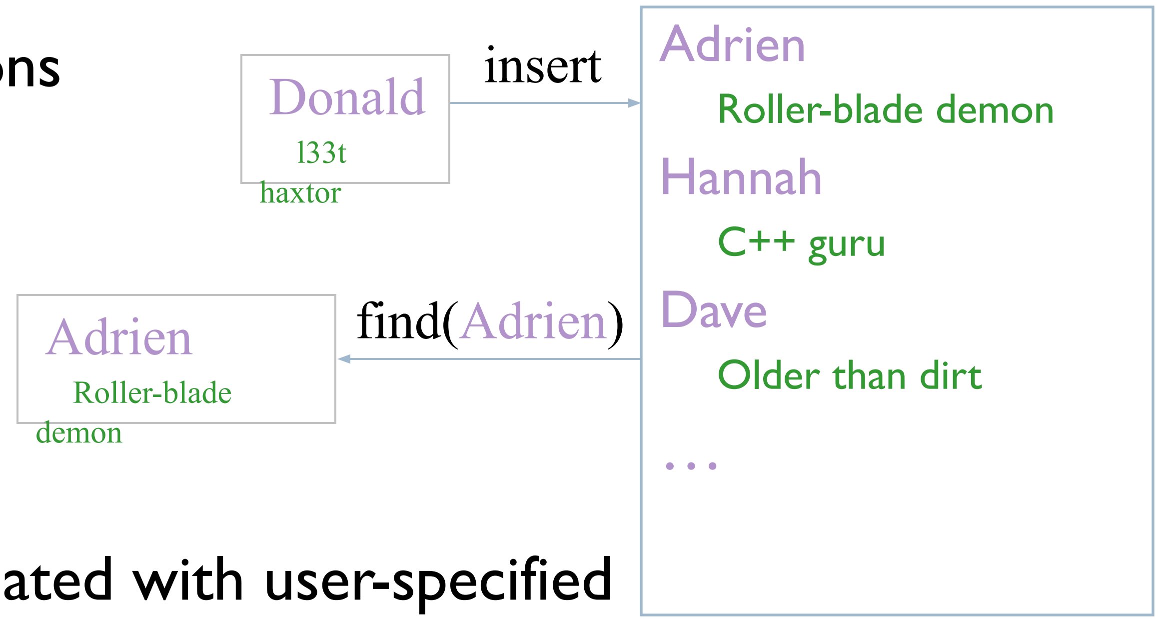
Binary Tree Representation



Dictionary ADT

Dictionary operations

- create
- destroy
- insert
- find
- delete



Stores *values* associated with user-specified *keys*

- values may be any (homogeneous) type
- keys may be any (homogeneous) comparable type

Dictionary ADT: Used *Everywhere*

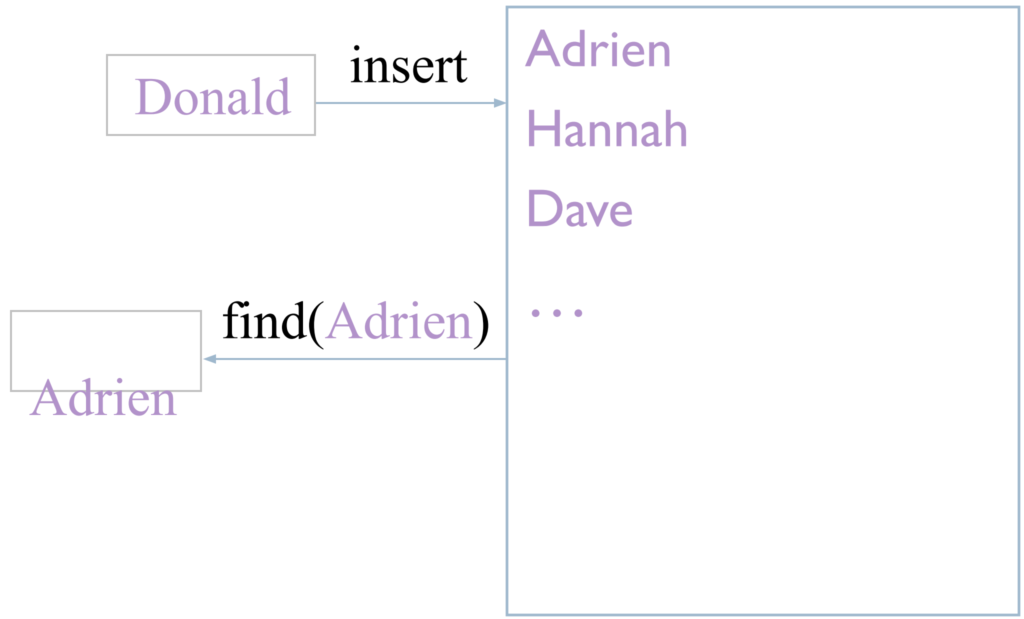
- Arrays
- Sets
- Dictionaries
- Router tables
- Page tables
- Symbol tables
- C++ structures
- …

Anywhere we need to *find* things fast based on a *key*

Search ADT

Dictionary operations

- create
- destroy
- insert
- find
- delete



Stores only the keys

- keys may be any (homogenous) comparable
- quickly tests for membership

Simplified dictionary, useful for examples (e.g. CSE 326)

Dictionary Data Structure: Requirements

- Fast insertion
  - runtime:
- Fast searching
  - runtime:
- Fast deletion
  - runtime:

Naïve Implementations

	unsorted array	sorted array	linked list
insert	O(n)	find + O(n)	O(1)
find	O(n)	O(log n)	O(n)
delete	find + O(1) (mark-as-deleted)	find + O(1) (mark-as-deleted)	find + O(1)

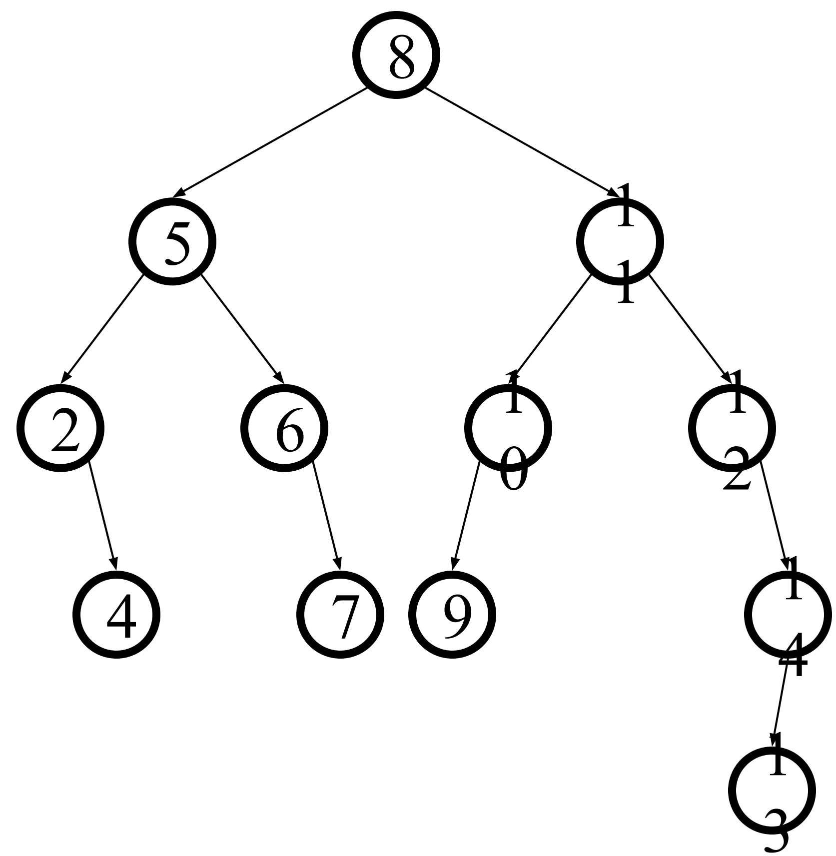
Binary Search Tree Dictionary Data Structure

Binary tree property

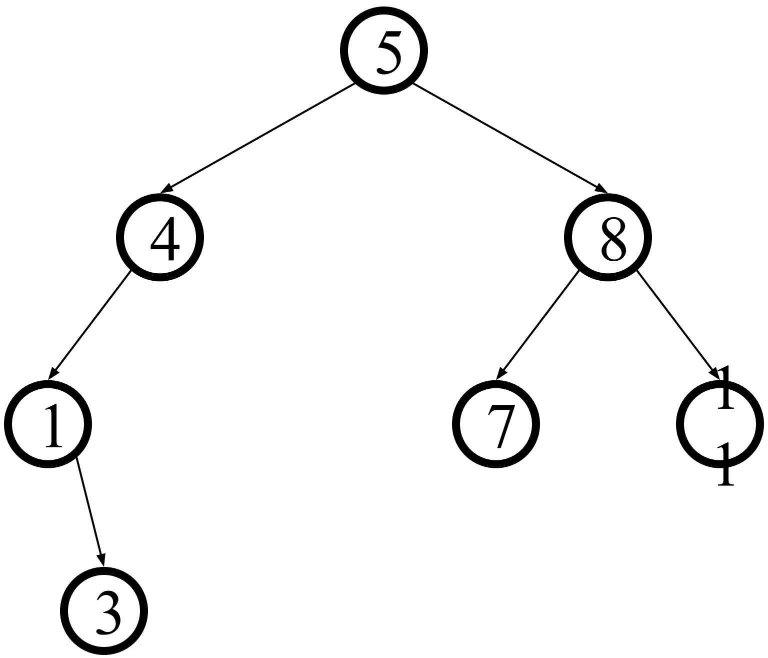
- each node has ≤ 2 children
- result:
  - storage is small
  - operations are simple
  - average depth is small

Search tree property

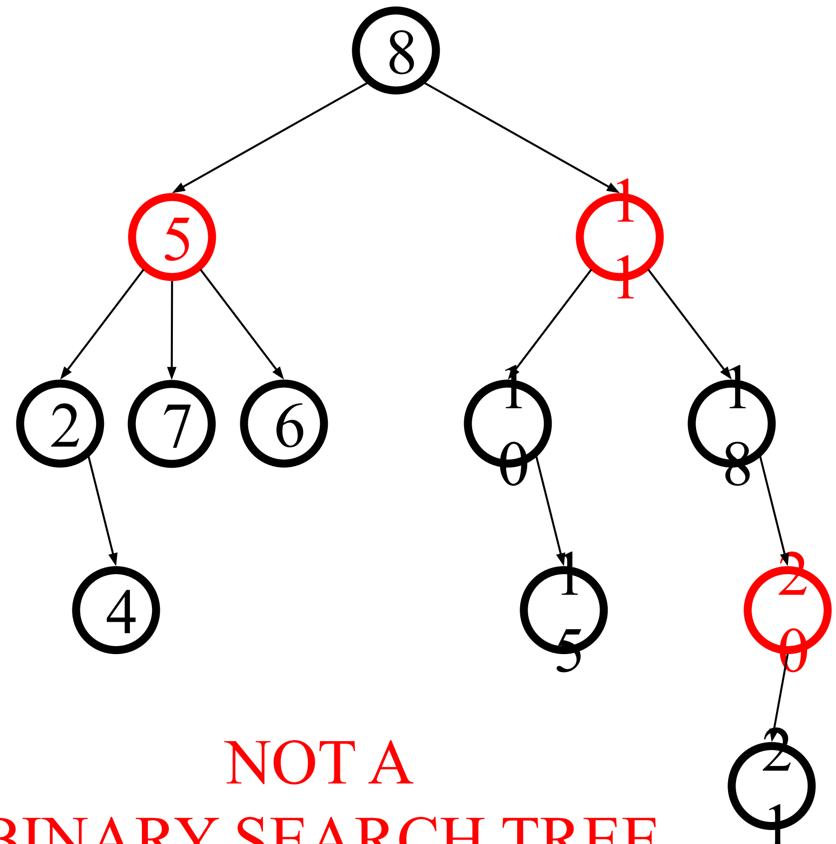
- all keys in left subtree smaller than root's key
- all keys in right subtree larger than root's key
- result:
  - easy to find any given key
  - Insert/delete by changing links



Example and Counter-Example



BINARY SEARCH TREE

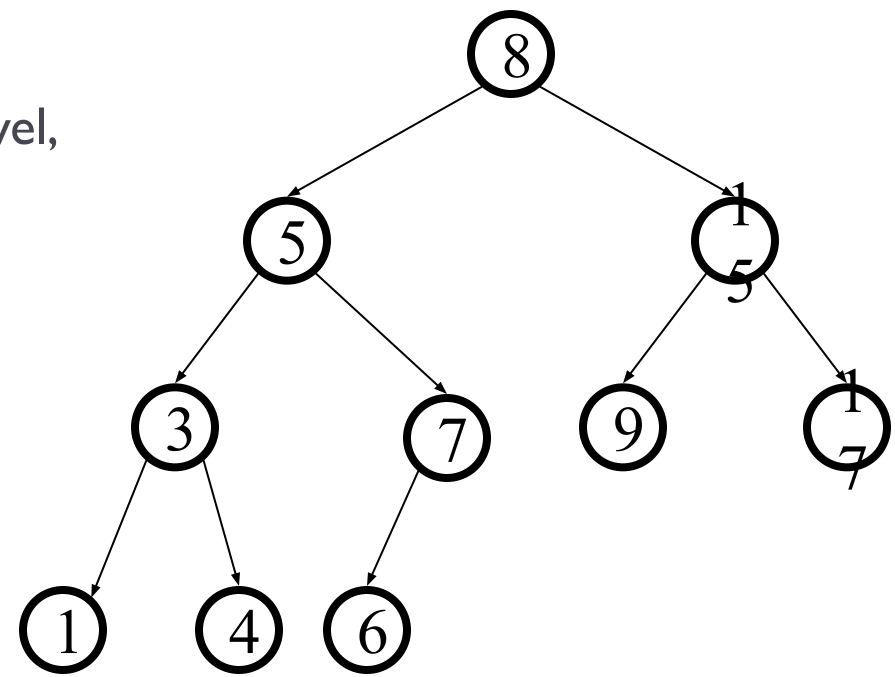


NOT A
BINARY SEARCH TREE

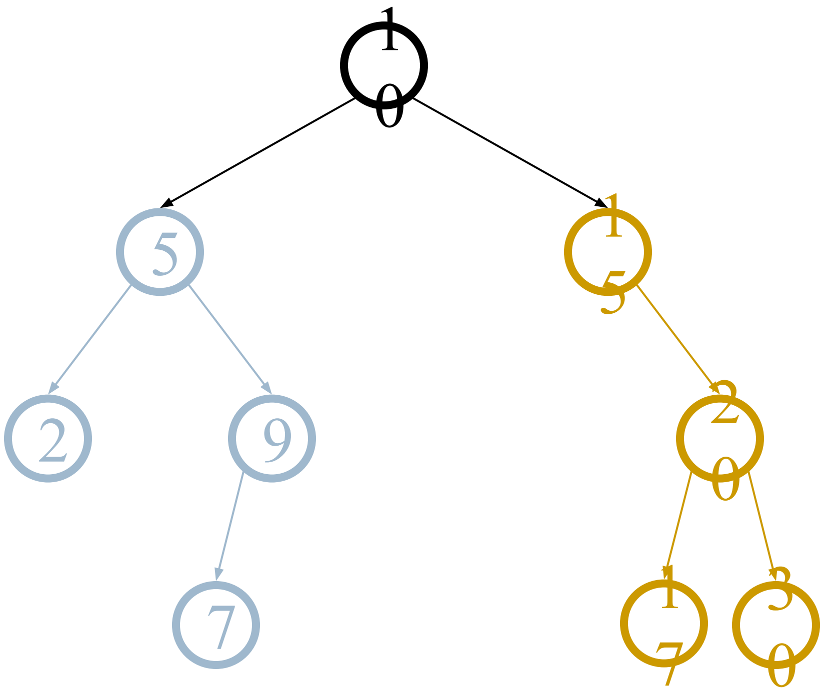
Complete Binary Search Tree

Complete binary search tree
(aka binary heap):

- Links are completely filled, except possibly bottom level, which is filled left-to-right.



In-Order Traversal



visit left subtree

visit node

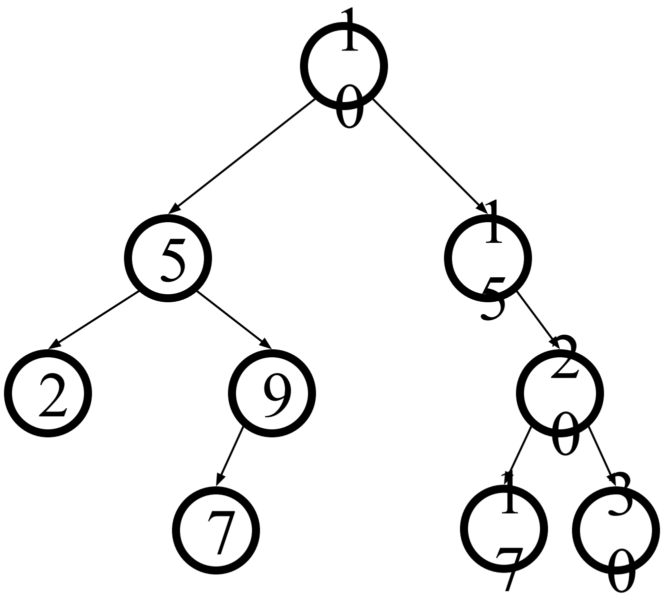
visit right subtree

What does this guarantee with a BST?

In order listing:

2→5→7→9→10→15→17→20→30

Recursive Find



Runtime:

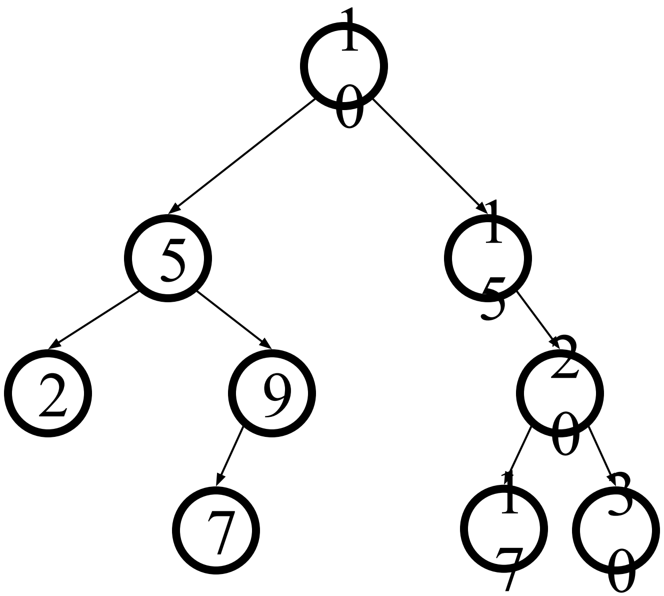
Best-worse case?

Worst-worse case?

f(depth)?

```
Node *
find(Comparable key, Node * t)
{
  if (t == NULL) return t;
  else if (key < t->key)
    return find(key, t->left);
  else if (key > t->key)
    return find(key, t->right);
  else
    return t;
}
```

Iterative Find




```
Node *
find(Comparable key, Node * t)
{
  while (t != NULL && t->key != key)
  {
    if (key < t->key)
      t = t->left;
    else
      t = t->right;
  }

  return t;
}
```

Insert

Concept:
- Proceed down tree as in Find
- If new key not found, then insert a new node at last spot traversed

```
void
insert(Comparable x, Node * t)
{
  if ( t == NULL ) {
    t = new Node(x);

  } else if (x < t->key) {
    insert( x, t->left );

  } else if (x > t->key) {
    insert( x, t->right );

  } else {
    // duplicate
    // handling is app-dependent
}
```

BuildTree for BSTs

- Suppose the data 1, 2, 3, 4, 5, 6, 7, 8, 9 is inserted into an initially empty BST:
  - in order
  - in reverse order
  - median first, then left median, right median, etc.

Analysis of BuildTree

Worst case is $O(n^2)$

$$1 + 2 + 3 + \ldots + n = O(n^2)$$

Average case assuming all orderings equally likely:

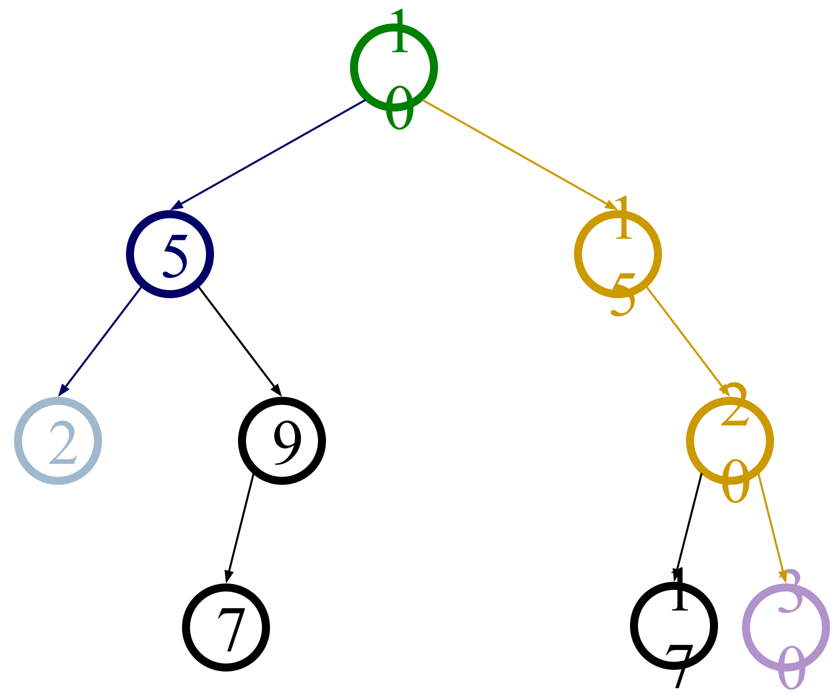
$O(n \log n)$

- averaging over all insert sequences (not over all binary trees)
- equivalently: average depth of a node is log n
- proof: see Introduction to Algorithms, Cormen, Leiserson, & Rivest

BST Bonus: FindMin, FindMax

- Find minimum

- Find maximum

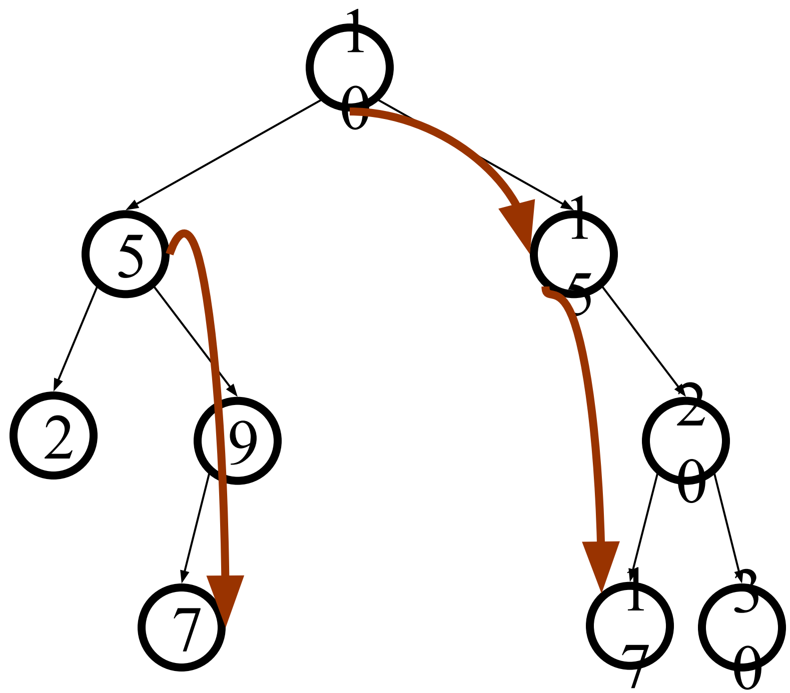


Successor Node

Next larger node

in this node's subtree

```
Node * succ(Node * t) {
  if (t->right == NULL)
    return NULL;
  else
    return min(t->right);
}
```

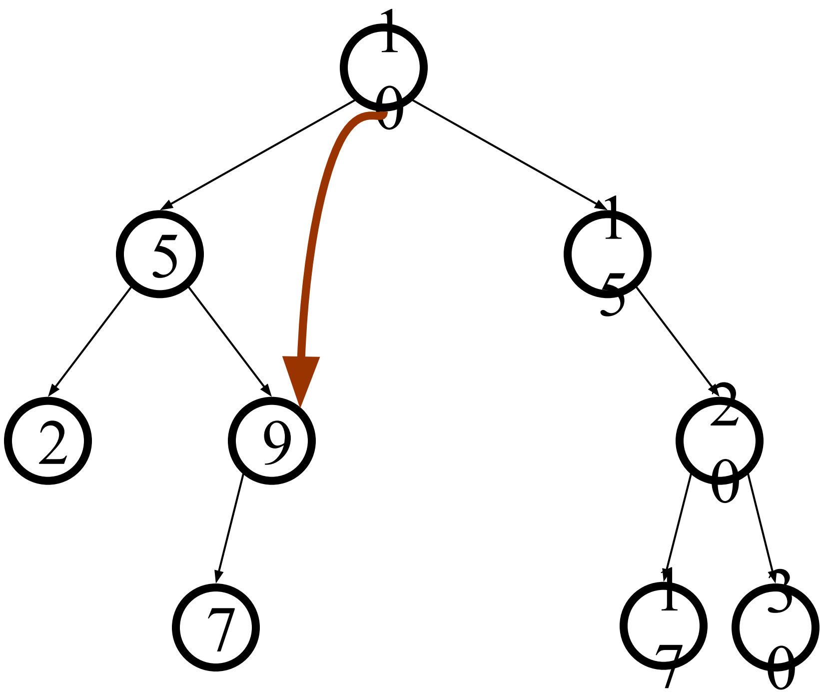


How many children can the successor of a node have?

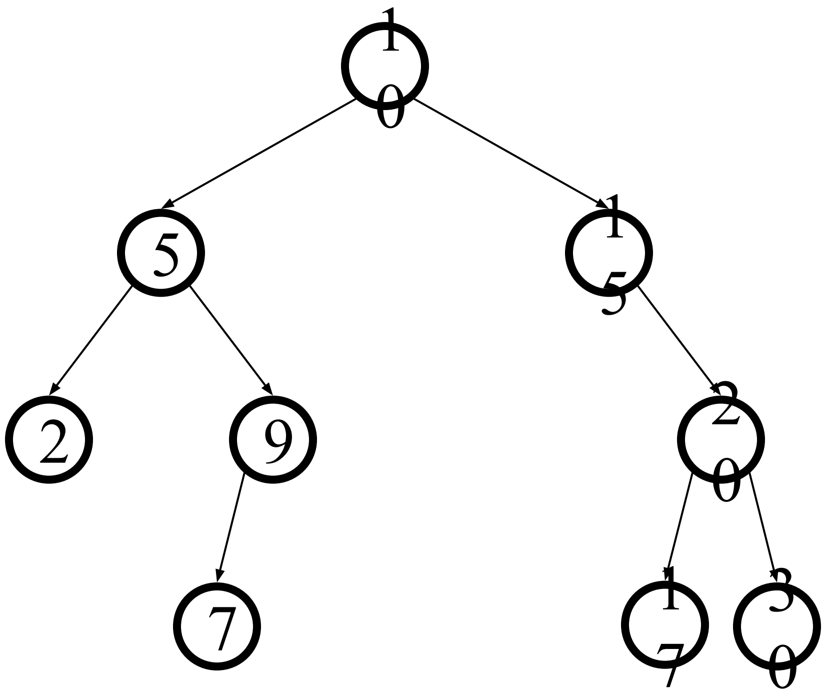
Predecessor Node

Next smaller node
in this node's subtree

```
Node * pred(Node * t) {
  if (t->left == NULL)
    return NULL;
  else
    return max(t->left);
}
```



Deletion

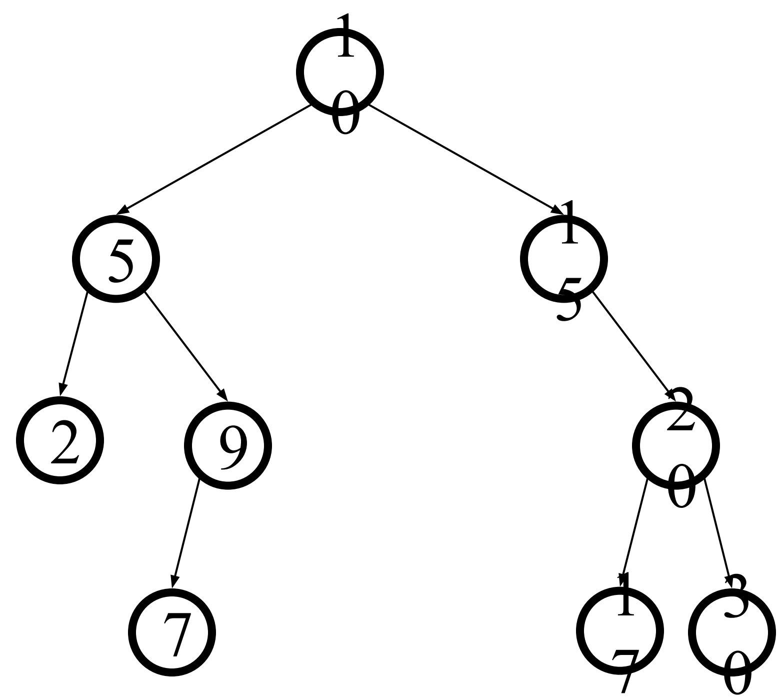


Why might deletion be harder than insertion?

Lazy Deletion

Instead of physically deleting nodes, just mark them as deleted

- simpler
- physical deletions done in batches
- some adds just flip deleted flag
  - extra memory for deleted flag
  - many lazy deletions slow finds
  - some operations may have to be modified (e.g., min and max)



Lazy Deletion

Delete(17)

Delete(15)

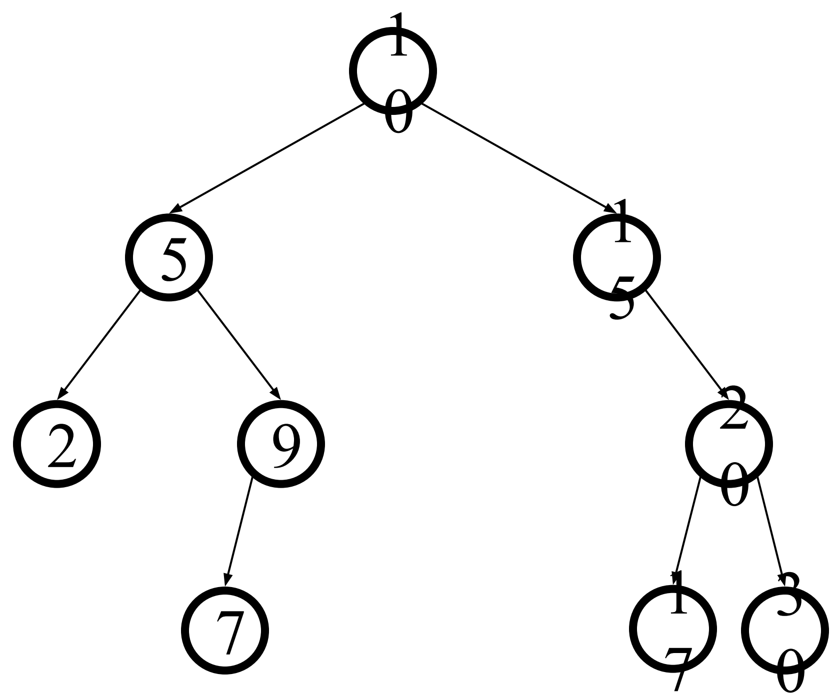
Delete(5)

Find(9)

Find(16)

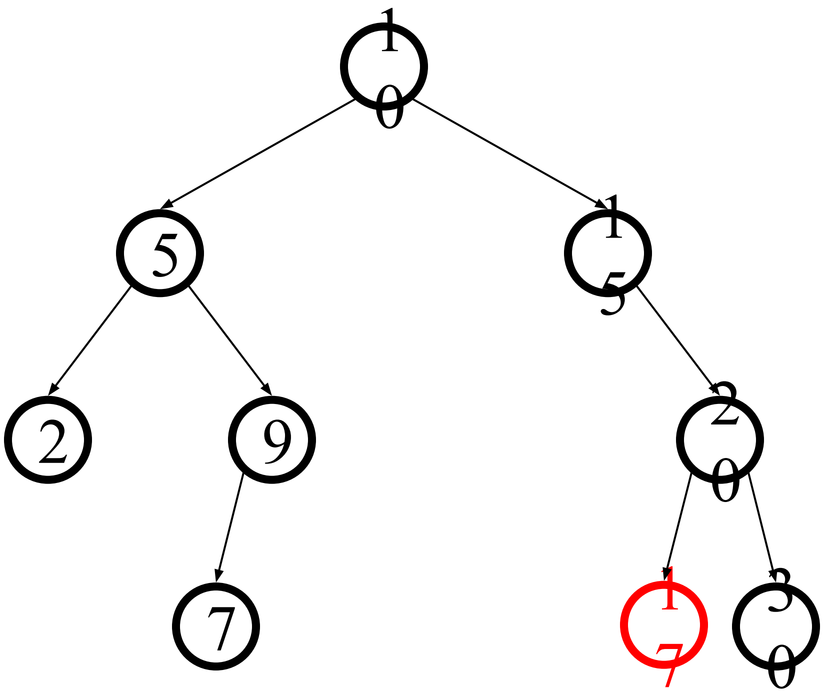
Insert(5)

Find(17)



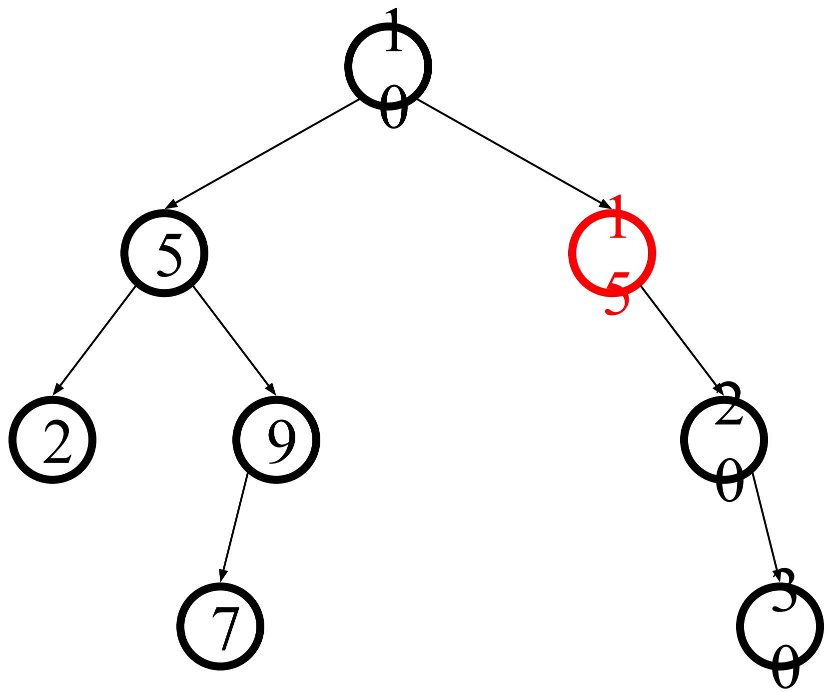
Deletion - Leaf Case

Delete(17)



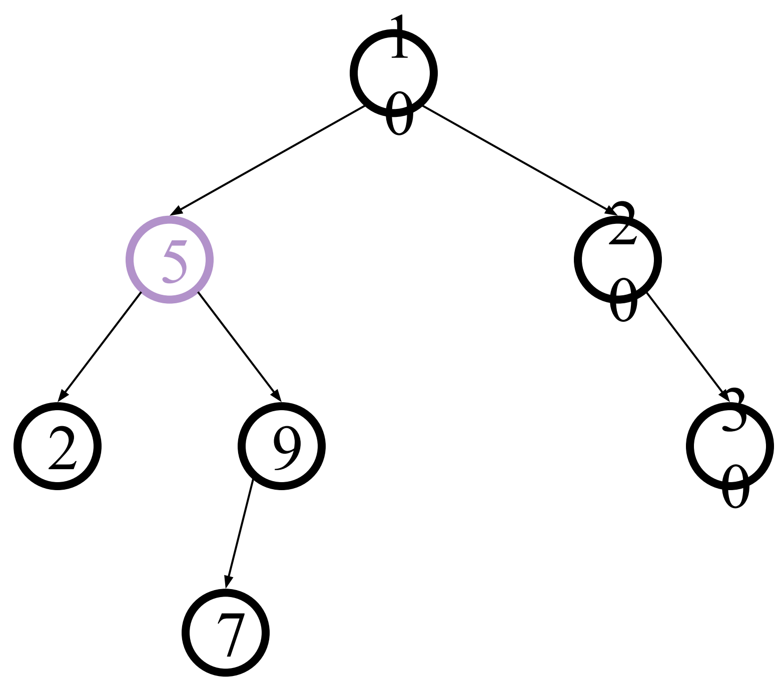
Deletion - One Child Case

Delete(15)



Deletion - Two Child Case

Replace node with descendant whose value is guaranteed to be between left and right subtrees: the successor



Delete(5)

Could we have used predecessor instead?

Delete Code

```
void delete(Comparable key, Node *& root) {
  Node *& handle(find(key, root));
  Node * toDelete = handle;
  if (handle != NULL) {
    if (handle->left == NULL) {          // Leaf or one child
      handle = handle->right;
      delete toDelete;
    } else if (handle->right == NULL) { // One child
      handle = handle->left;
      delete toDelete;
    } else {                             // Two children
      successor = succ(root);
      handle->data = successor->data;
      delete(successor->data, handle->right);
    }
  }
}
```

Thinking about Binary Search Trees

Observations

- Each operation views two new elements at a time
- Elements (even siblings) may be scattered in memory
- Binary search trees are fast *if they're shallow*

Realities

- For large data sets, disk accesses dominate runtime
- Some deep and some shallow BSTs exist for any data

Beauty is Only Θ(log n) Deep

Binary Search Trees are fast if they're shallow:

- perfectly complete
- complete – possibly missing some "fringe" (leaves)
- any other good cases?

What matters?

- Problems occur when one branch is much longer than another
- i.e. when tree is out of balance

Dictionary Implementations

	unsorted array	sorted array	linked list	BST
insert	O(n)	find + O(n)	O(1)	O(Depth)
find	O(n)	O(log n)	O(n)	O(Depth)
delete	find + O(1) (mark-as-deleted)	find + O(1) (mark-as-deleted)	find + O(1)	O(Depth)

BST's looking good for shallow trees, i.e. if Depth is small (log n); otherwise as bad as a linked list!

Digression: Tail Recursion

- Tail recursion: when the tail (final operation) of a function recursively calls the function
- Why is tail recursion especially bad with a linked list?
- Why *might* it be a lot better with a tree? Why *might it not*?

Making Trees Efficient: Possible Solutions

Keep BSTs shallow by maintaining "balance"

- AVL trees

… also exploit most-recently-used (mru) info

- Splay trees

Reduce disk access by increasing branching factor

- B-trees