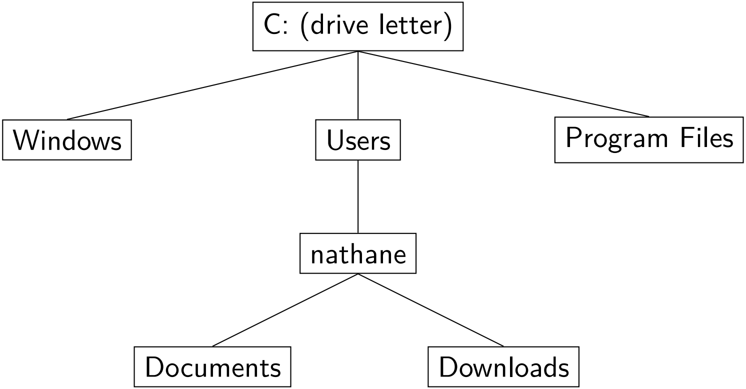
Trees

Trees

- Information is not aways easily modeled with a list
  - Many times a hierarchy type structure is more intuitive
  - Example: File Systems



Trees

- These structures are caled Trees
- Trees:
  - give reasonably efficient searching
  - are frequently the underlying structure of databases
  - are used in compilers to check syntax, semantics, etc
  - can be used to evaluate statements in non-compiled languages
- Trees in and of themselves are not helpful
  - They define a structure for data, not impose rules on how data is inserted, removed, and consumed
  - We will specialize the trees to do some cool things!
- Trees are made of nodes that contain both data and references to other nodes (similar to a Linked List)

Definitions

- Root
  - Single node at the top of the tree
- Parent
  - A node directly above another node
  - In the file system example, Users is the parent of nathane
- Child
  - If node A is node B's parent, then B is A's child (a node directly under another node)
- Siblings
  - Nodes with the same parent
- Descendent
  - Any node that can be reached from a node by only moving down
  - In the example, Users has the descendents nathane, Documents, and Downloads
- Ancestor
  - Any node that can be reached by only moving up the tree

Definitions

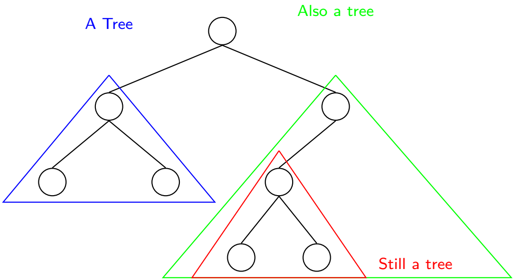
- Subtree
  - A node and all of its descendents
- Level
  - Distance (number of edges) of a node to the root
  - The level of a node is the level of its parent + 1
  - The level of the root is 0
  - In the FS example, C: has a level of 0, Windows, Users, and Program Files have a level of 1, nathane has a level of 2, and Documents and Downloads have a level of 3
- Height (of the tree)
  - The maximum level of any node in the tree
  - The height of the tree in the example is 3

Definitions

- Leaf Node
  - A node with no children
- Internal Node
  - Any node that is not a leaf node
- Binary Tree
  - Tree in which every node has at most two children (left and right)

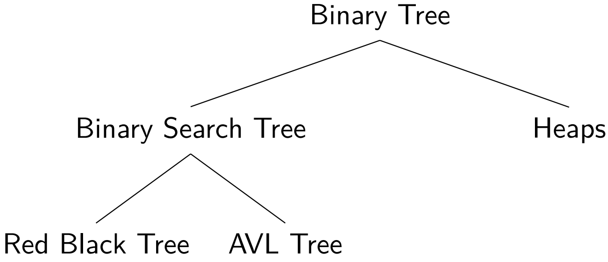
Important Sidebars

1. Some of these definitions will differ from source to source; any good source will clarify the definitions it will use.
2. Trees are inherently recursive. Many (if not most) operations discussed will use recursion



Kinds of Trees

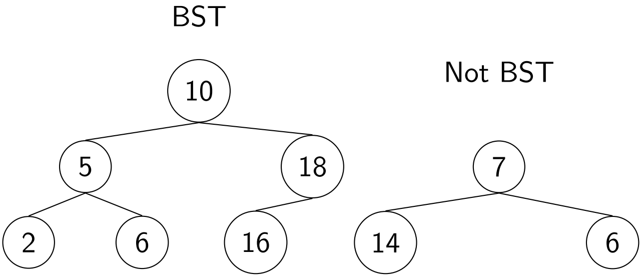
We will discuss...



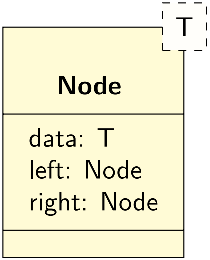
Binary Search Trees

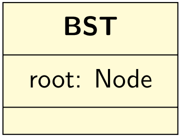
Binary trees without rules are not helpful. We want to define rules for insertion and deletion that let us exploit properties of trees.

- Binary Search Tree (BST)
  - A binary tree where, for any node N, every node in the left subtree has a value less than N's, and every node in the right subtree is greater than N's



Binary Search Tree Nodes



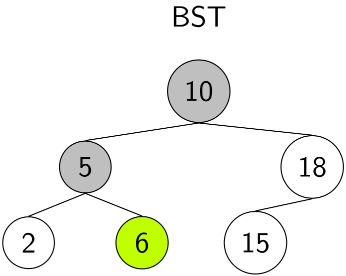


Example

Is 6 in the tree?

- Start at the root (n = root)
- $6 < 10$ (n.data), so move left (n = n.left)
- $6 > 5$ (n.data), so move right (n = n.right)
- $6 == 6$ found!

Takes 3 probes (check against 10, 5, and 6)

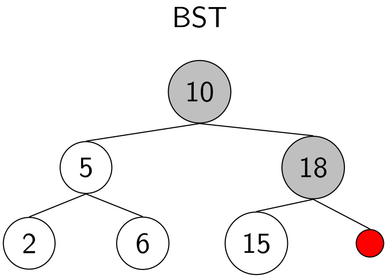


Example

Is 20 in the tree?

- Start at the root (n = root)
- $20 > 10$ (n.data), so move right (n = n.right)
- $20 > 18$ (n.data), so move right (n = n.right)
- n is null, so not found!

Takes 2 probes (check against 10 and 18)



Searching in BSTs

- Performance is ok
  - Faster than lists of unsorted data
  - Slower than hashed data

Recursive BST Search

```
public boolean search(int elm)
{
  return search(elm, root)
}

public boolean search(int elm, Node n)
{
  if (n == null)
    return false;
  if (n.data == elm)
    return true;
  if (elm < n.data) // or (n.data > elm)
    return search(elm, n.left);
  return search(elm, n.right):
}
```

Iterative BST Search

```
public boolean search (int elm)
{
  Node n = root;
  while (n != null)
  {
    if (n.data == elm)
      return true;
    else if (elm < n.data) // or (n.data > elm)
      n = n.left;
    else
      n = n.right;
  }
  return false;
}
```

BST Insertion

- Simple at a high level
    - Find where it should go, and put it there
- New values are always inserted as leaves
- We'll start with a new tree

```
BST<Integer> bst = new BST<>( );
```

    - Note: BST is *not* a JCF class; the notation is just for familiarity

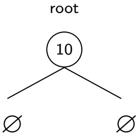
root

$\varnothing$

BST Insertion

```
bst.insert(10);
```

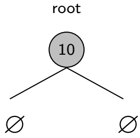
- The tree is empty, so the value gets inserted as the root node



BST Insertion

`bst.insert(18);`

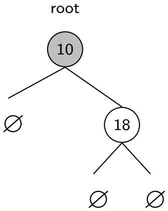
- Start at the root (`n=root`)
  - `Node n` is shaded)



BST Insertion

`bst.insert(18);`

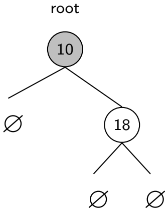
- $18 > 10$
- `n.right == null`
  - insert a new node at `n.right`



BST Insertion

`bst.insert(5);`

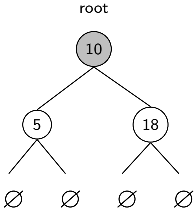
- Start at the root (`n = root`)



BST Insertion

```
bst.insert(5);
```

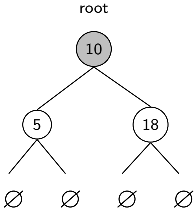
- $5 < 10$
- `n.left == null`
  - insert a new node at `n.left`



BST Insertion

`bst.insert(6);`

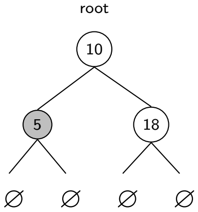
- Start at the root (`n = root`)



BST Insertion

`bst.insert(6);`

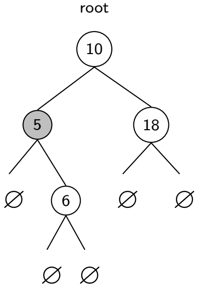
- $6 < 10$
- `n.left != null`
  - `n = n.left`



BST Insertion

```
bst.insert(6);
```

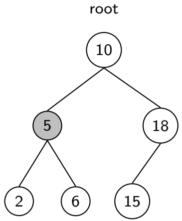
- $6 > 5$
- `n.right == null`
  - `n = n.left`



BST Insertion

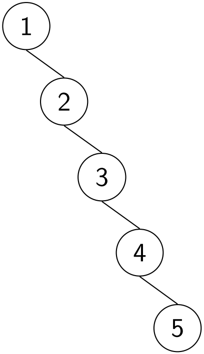
```
bst.insert(15);
bst.insert(2)
```

- null nodes are omitted for clarity
- `n.right == null`



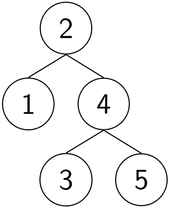
Order Matters

Insert 1, 2, 3, 4, 5 in that order



This is sometimes called a chain, spine, or backbone

Insert 2, 4, 3, 1, 5

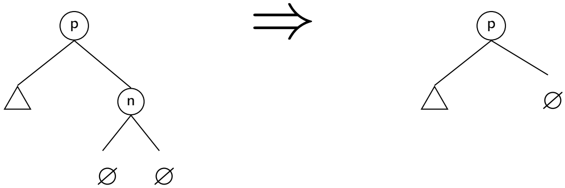


BST Removal

- To delete a node from a BST:
  - Find the node to delete (move left and right until you have a reference to it) (call it `n`)
    - You should also keep track of the parent node
  - If `n` is a leaf node, delete it
  - If `n` has one child, replace `n` with its child
  - If `n` has two children, swap `n.data` with the smallest value larger than it, then delete the node you swapped from as if it has one or no children
    - You can also use largest value smaller than it...
  - Update the parent of `n` to reflect the new node

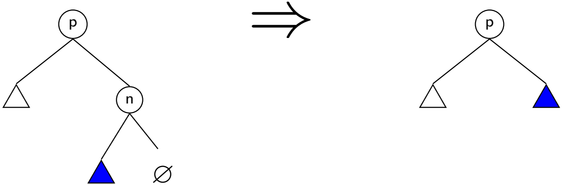
BST Removal

- Find node n and its Parent p
- If n is a leaf node, delete it
  - Note: triangular nodes are subtrees we don't care about. They could have no nodes, one node, or a million nodes.



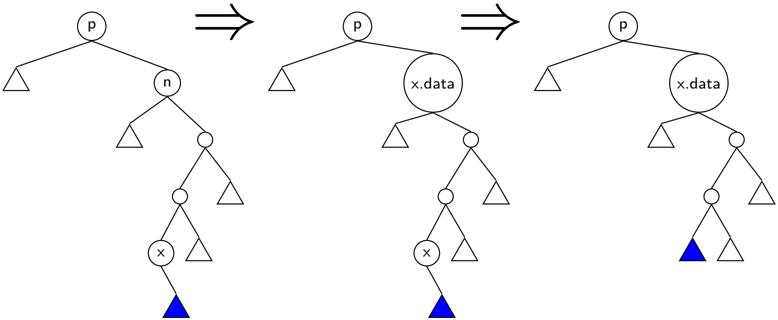
BST Removal

- If n has one child, replace n with its child



BST Removal

- If `n` has two children, find the node (`x`) with the smallest value larger than `n.data`.
  - This will be a single step right, then as far left as possible
- Set `n.data = x.data`
- Remove `x` as it has either one or no children



Iterating Through Trees

- There are three main ways we iterate through *any* Binary Tree
  - InOrder
  - PreOrder
  - PostOrder
- These traversals work for all binary trees, not just the BST

InOrder Traversal

To InOrder traverse a tree with root (callled n)

1. Process the Left subtree InOrder
2. Process n
3. Process the Right subtree InOrder

- If n's left subtree is called L, and n's right subtree is called R, then the ordering is LnR
- n is In Between L and R
- In the following examples, we will process the nodes by printing their values

InOrder Print

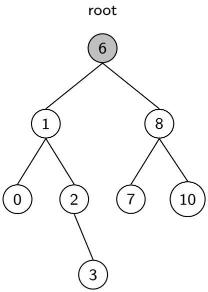
```
void inorder_print()
{
  inorder_print(root);
}

void inorder_print(Node n)
{
  if (n == null) return;
  inorder_print(n.left);
  System.out.print(n.data + " ");
  inorder_print(n.right);
}
```

InOrder Print

```
bst.inorder_print();
```

- calls
  `bst.inorder_print(root);`
- Shaded nodes have been n, but not printed

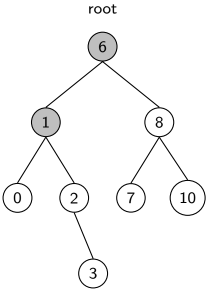
Output:



InOrder Print

- Calls `bst.inorder_print(n.left);`

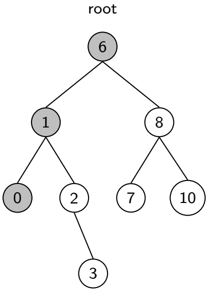
  Output:



InOrder Print

- Calls `bst.inorder_print(n.left);`

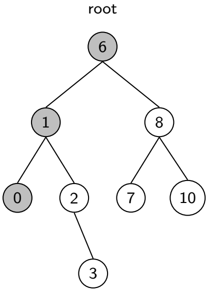
  Output:



InOrder Print

- Calls `bst.inorder_print(n.left);`
- `n==null`, so nothing is printed

Output:

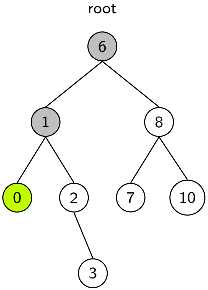


InOrder Print

- `n.data` is printed
- Printed nodes are green

Output:

`0`

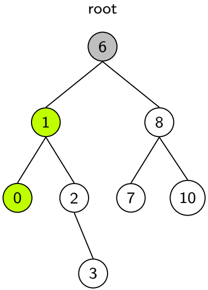


InOrder Print

- `n.data` is printed

  Output:

  0 1

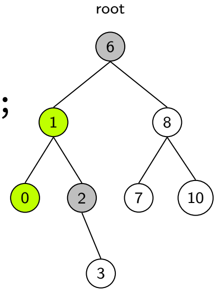


InOrder Print

- Calls
  `bst.inorder_print(n.right);`

  Output:

  0 1

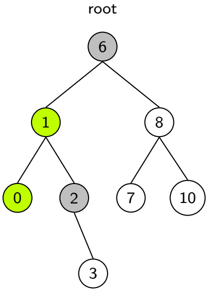


InOrder Print

- Calls `bst.inorder_print(n.left);`
- `n==null`, so nothing printed

Output:

`0 1`

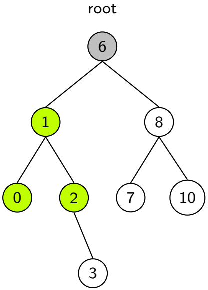


InOrder Print

- `n.data` is printed

  Output:

  `0 1 2`

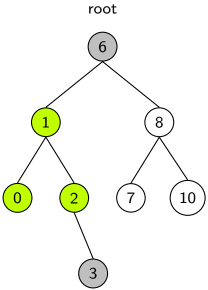


InOrder Print

- Calls `bst.inorder_print(n.right)`

  Output:

  `0 1 2`

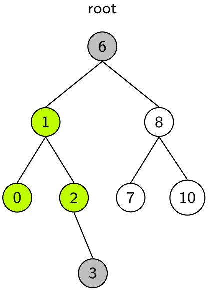


InOrder Print

- Calls `bst.inorder_print(n.left)`
- `n == null`, so nothing printed

Output:

`0 1 2`

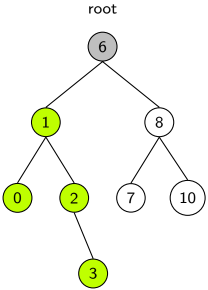


InOrder Print

- `n.data` is printed
- Will recurse right, but it is null, so those slides ommitted

Output:

`0 1 2 3`

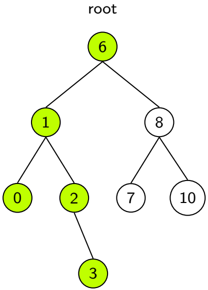


InOrder Print

- `n.data` is printed

  Output:

  `0 1 2 3 6`

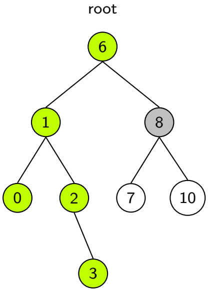


InOrder Print

- Calls `bst.inorder_print(n.right)`

  Output:

  `0 1 2 3 6`

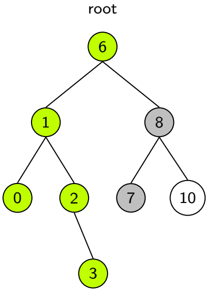


InOrder Print

- Calls `bst.inorder_print(n.left)`

  Output:

  `0 1 2 3 6`

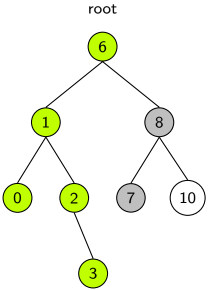


InOrder Print

- Calls `bst.inorder_print(n.left)`
- `n == null`, so nothing printed

Output:

`0 1 2 3 6`

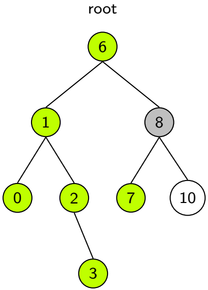


InOrder Print

- `n.data` is printed
- Will recurse right, but it is null, so those slides ommitted

Output:

`0 1 2 3 6 7`

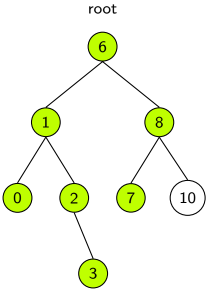


InOrder Print

- `n.data` is printed

  Output:

  `0 1 2 3 6 7 8`

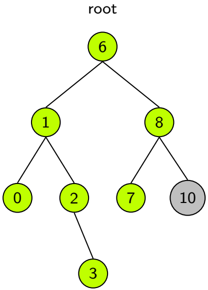


InOrder Print

- Calls `bst.inorder_print(n.right)`

  Output:

  0 1 2 3 6 7 8

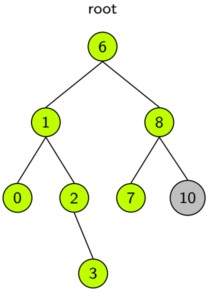


InOrder Print

- Calls `bst.inorder_print(n.left)`
- `n == null`, so nothing printed

Output:

`0 1 2 3 6 7 8`

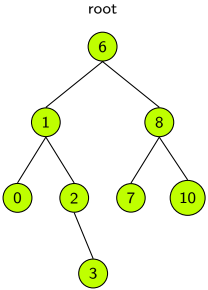


InOrder Print

- `n.data` is printed
- Will recurse right, but it is null, so those slides ommitted

Output:

`0 1 2 3 6 7 8 10`



InOrder and BST

An InOrder of a BST will *always* yield a sorted ordering of the data in the tree

PreOrder Traversal

To PreOrder traverse a tree with root (callled n)

1. Process n
2. Process the Left subtree PreOrder
3. Process the Right subtree PreOrder

- If n's left subtree is called L, and n's right subtree is called R, then the ordering is nLR
- n is processed Previously to L and R
- As an exercise, generate the output from the previous example using PreOrder instead of InOrder

Exercise Solution

6 1 0 2 3 8 7 10

PostOrder Traversal

To PostOrder traverse a tree with root (callled n)

1. Process the Left subtree PreOrder
2. Process the Right subtree PreOrder
3. Process n

- If n's left subtree is called L, and n's right subtree is called R, then the ordering is LRn
- n is processed after (Post) L and R
- As an exercise, generate the output from the previous example using PostOrder instead of InOrder

Exercise Solution

0 3 2 1 7 10 8 6