Programming Languages and Techniques (CIS120)

Lecture 7

September 11th, 2015

Binary Search Trees

(Lecture notes Chapter 7)

Announcements

- Homework 2 is online
  - due Tuesday, Sept. 15th

- Recitation Section 208 Weds. 5-6 has *moved* from Moore 100B to Moore 207
  - Note: Section 207, also Weds. 5-6, remains in Moore 100A

- My office hours next week: *Tuesday* 3:30 – 5:00
  - (not Monday, this should be the last such change)

Trees as containers

Big idea: find things faster by searching less

Trees as Containers

- Like lists, trees aggregate (possibly ordered) data
- As we did for lists, we can write a function to determine whether the data structure *contains* a particular element

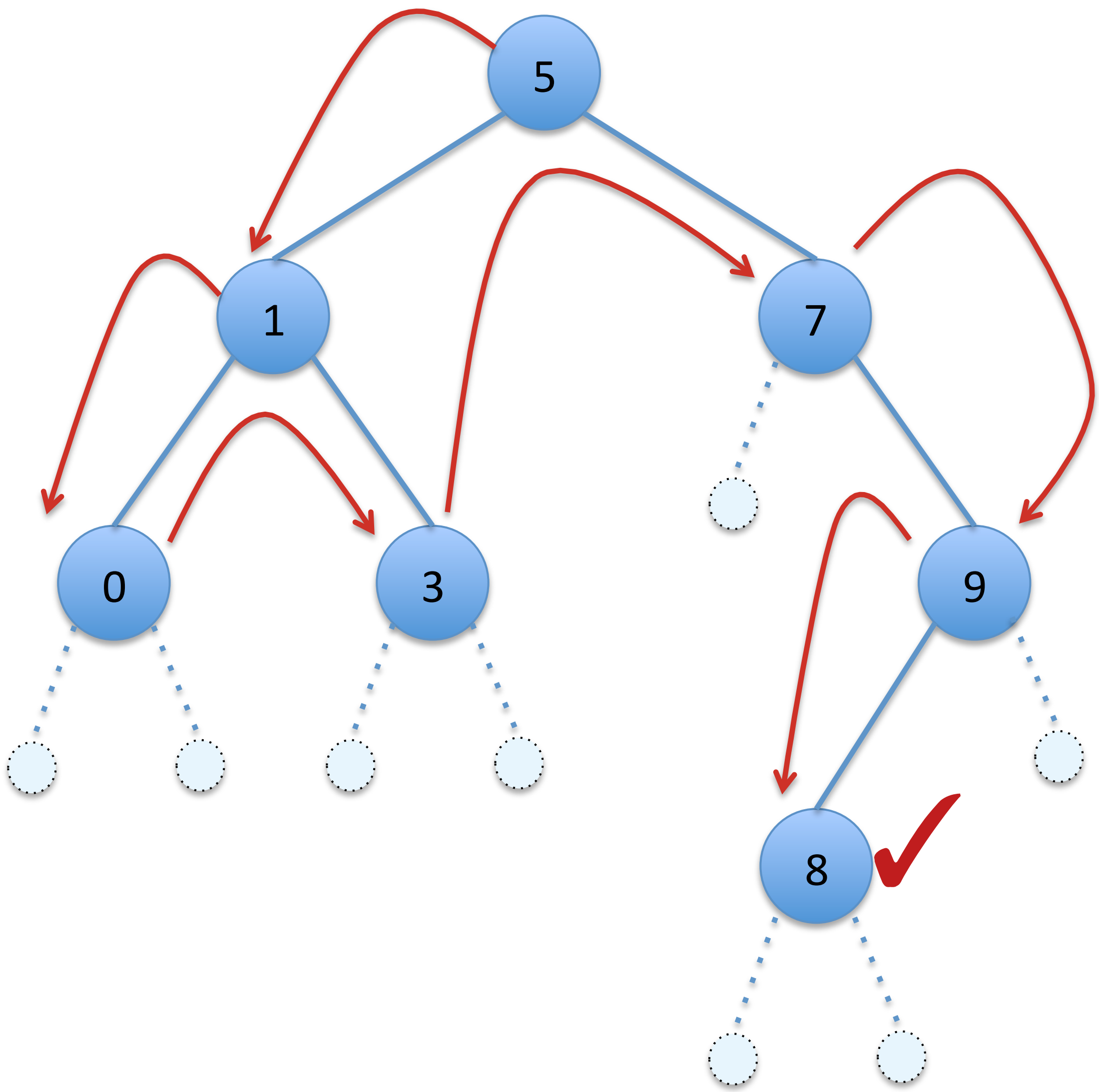
```
type tree =
| Empty
| Node of tree * int * tree
```

Searching for Data in a Tree

```
let rec contains (t:tree) (n:int) : bool =
  begin match t with
  | Empty -> false
  | Node(lt,x,rt) -> x = n ||
           (contains lt n) || (contains rt n)
  end
```

- This function searches through the tree, looking for n
- In the worst case, it might have to traverse the *entire* tree
  - This version uses pre-order traversal
    (other traversal orders have the same worst case traversal time…why?)

Search during `(contains t 8)`



Challenge: Faster Search?

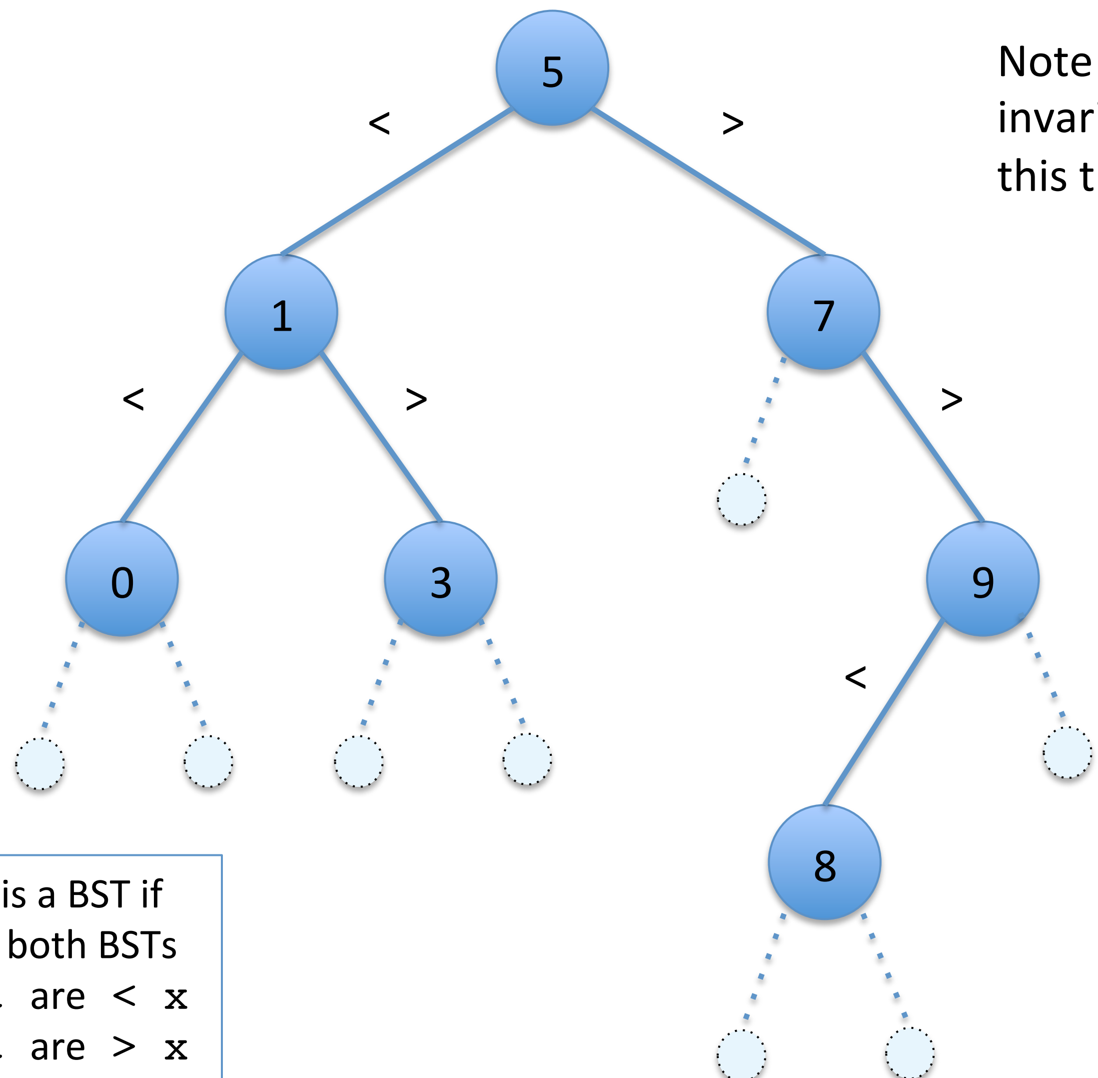
Binary Search Trees

- Key insight:
  Ordered data can be searched more quickly
  - This is why telephone books are arranged alphabetically
  - But requires the ability to focus on *half* of the current data
- A *binary search tree* (BST) is a binary tree with some additional *invariants*:*

> - `Node(lt,x,rt)` is a BST if
>   - `lt` and `rt` are both BSTs
>   - all nodes of `lt` are `<` `x`
>   - all nodes of `rt` are `>` `x`
> - `Empty` is a BST

*An data structure *invariant* is a set of constraints about the way that the data is organized. "types" (e.g. list or tree) are one kind of invariant, but we often impose additional constraints.

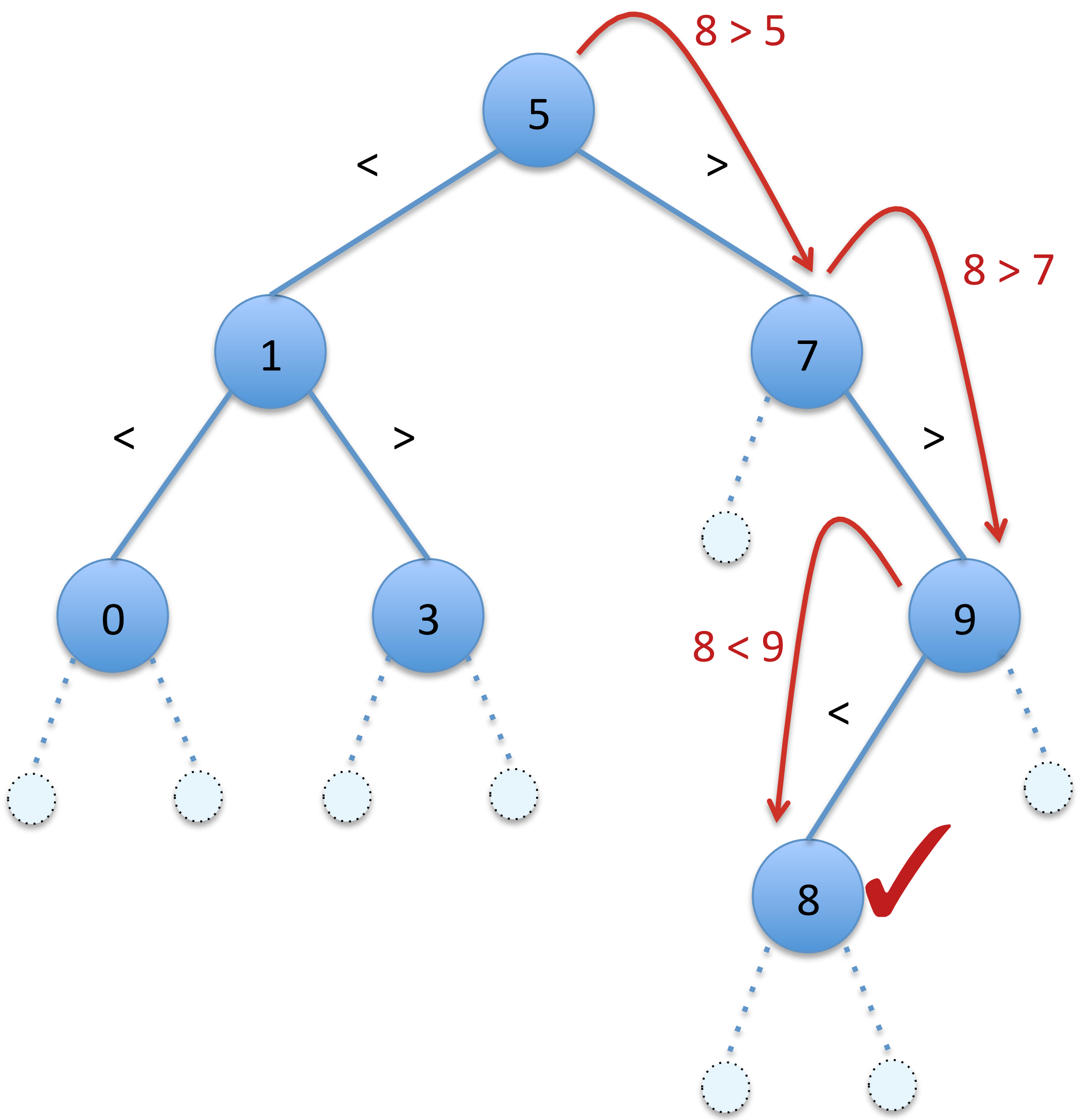
An Example Binary Search Tree



Note that the BST invariants hold for this tree.

- `Node(lt,x,rt)` is a BST if
  - `lt` and `rt` are both BSTs
  - all nodes of `lt` are < `x`
  - all nodes of `rt` are > `x`
- `Empty` is a BST

Search in a BST: `(lookup t 8)`



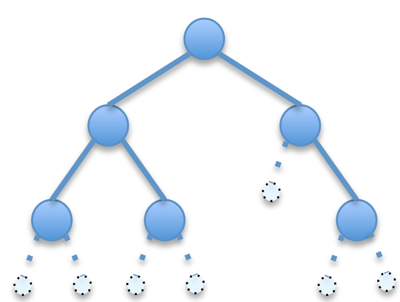
Searching a BST

```
(* Assumes that t is a BST *)
let rec lookup (t:tree) (n:int) : bool =
  begin match t with
  | Empty -> false
  | Node(lt,x,rt) ->
      if x = n then true
      else if n < x then (lookup lt n)
      else (lookup rt n)
  end
```

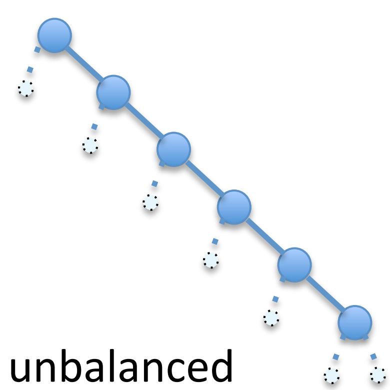
- The BST invariants guide the search.
- Note that lookup may return an incorrect answer if the input is *not* a BST!
  - This function *assumes* that the BST invariants hold of t.

BST Performance

- `lookup` takes time proportional to the *height* of the tree.
  - not the *size* of the tree (as it does with `contains`)
- In a *balanced tree*, the lengths of the paths from the root to each leaf are (almost) *the same*.
  - no leaf is too far from the root
  - the height of the BST is minimized
  - the height of a balanced binary tree is roughly $\log_2(N)$ where N is the number of nodes in the tree



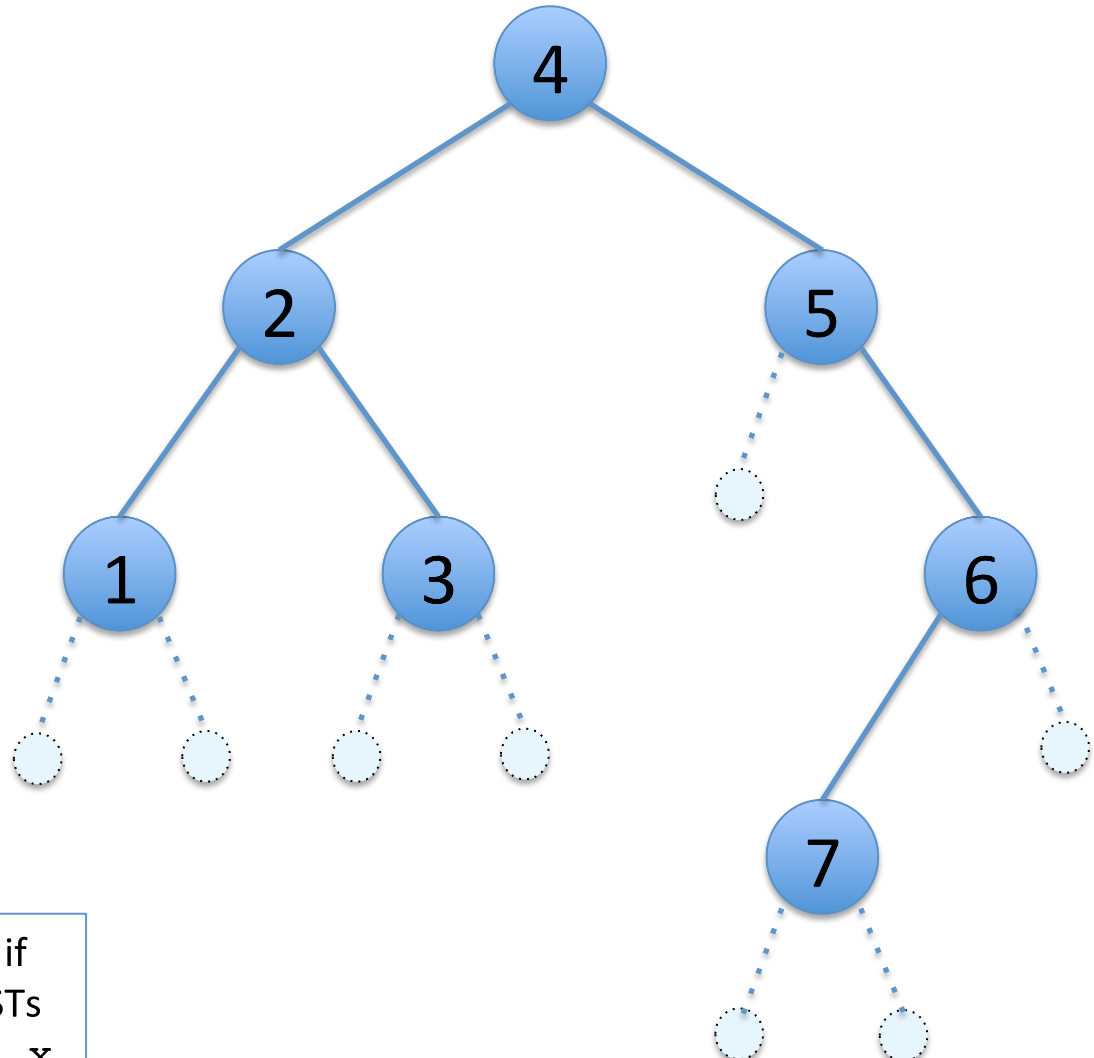
balanced



unbalanced

Is this a BST??

1. yes
2. no

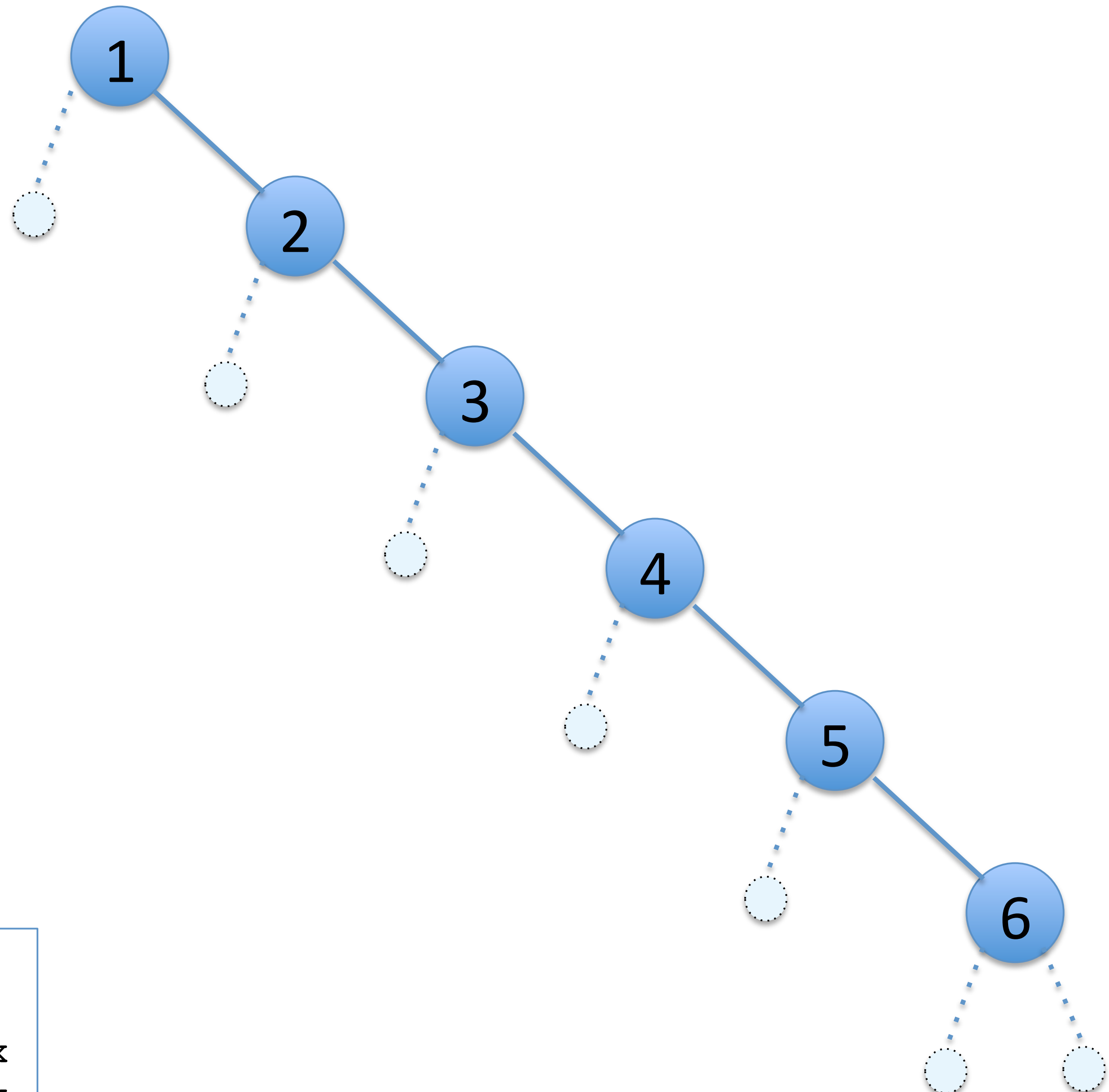


- `Node(lt,x,rt)` is a BST if
  - `lt` and `rt` are both BSTs
  - all nodes of `lt` are `<` `x`
  - all nodes of `rt` are `>` `x`
- `Empty` is a BST

Answer: no, 7 to the left of 6

Is this a BST??

1. yes
2. no

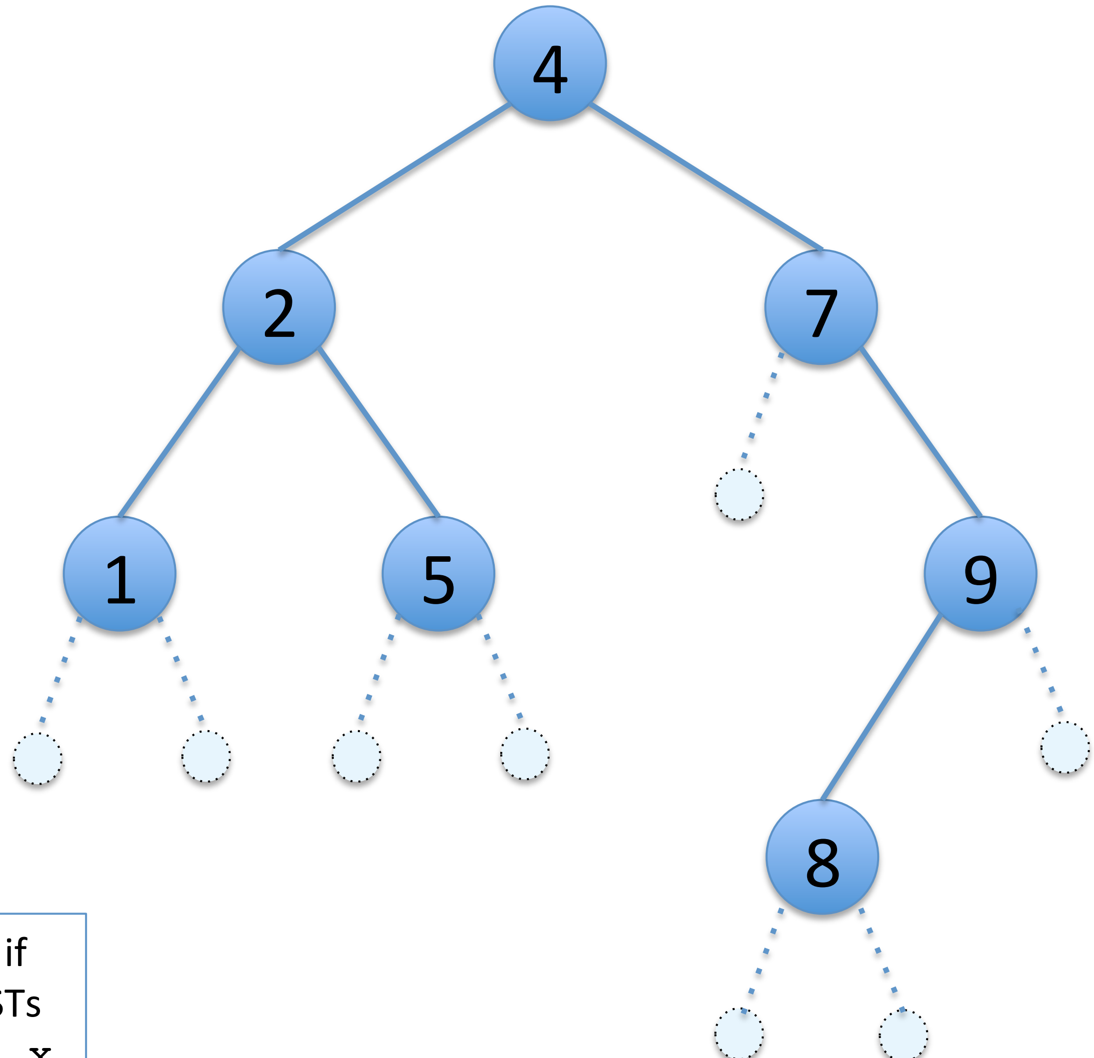


- `Node(lt,x,rt)` is a BST if
  - `lt` and `rt` are both BSTs
  - all nodes of `lt` are < `x`
  - all nodes of `rt` are > `x`
- `Empty` is a BST

Answer: Yes

Is this a BST??

1. yes
2. no

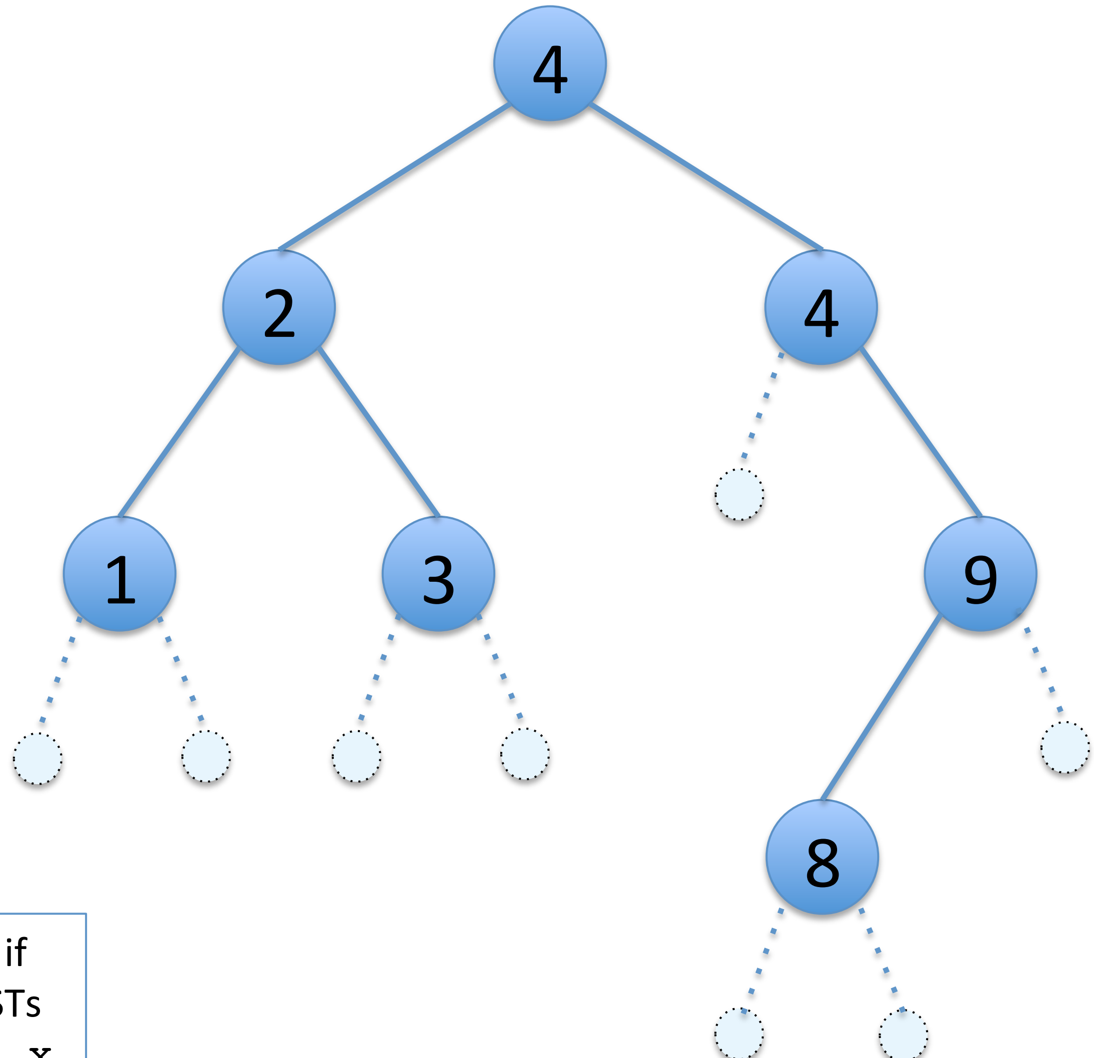


- `Node(lt,x,rt)` is a BST if
  - `lt` and `rt` are both BSTs
  - all nodes of `lt` are `<` `x`
  - all nodes of `rt` are `>` `x`
- `Empty` is a BST

Answer: no, 5 to the left of 4

Is this a BST??

1. yes
2. no

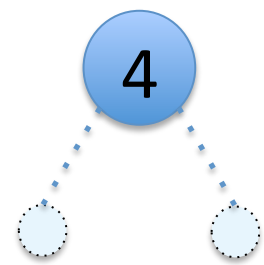


- `Node(lt,x,rt)` is a BST if
  - `lt` and `rt` are both BSTs
  - all nodes of `lt` are $<$ `x`
  - all nodes of `rt` are $>$ `x`
- `Empty` is a BST

Answer: no, 4 to the right of 4

Is this a BST??

1. yes
2. no

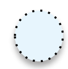


- `Node(lt,x,rt)` is a BST if
  - `lt` and `rt` are both BSTs
  - all nodes of `lt` are `<` `x`
  - all nodes of `rt` are `>` `x`
- `Empty` is a BST

Answer: yes

Is this a BST??

1. yes
2. no



- Node(lt,x,rt) is a BST if
  - lt and rt are both BSTs
  - all nodes of lt are < x
  - all nodes of rt are > x
- Empty is a BST

Answer: yes

Constructing BSTs

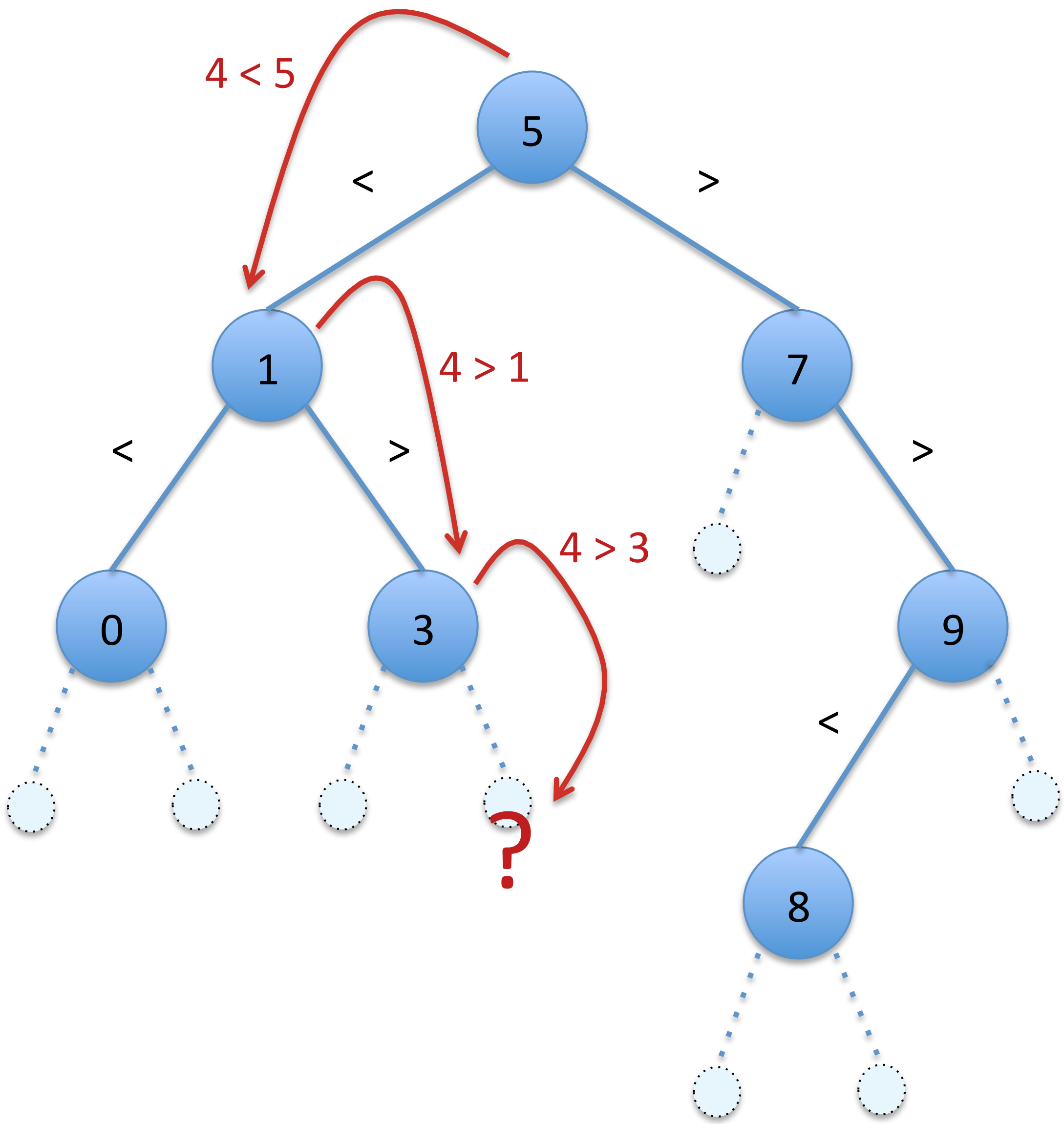
Inserting an element

How do we construct a BST?

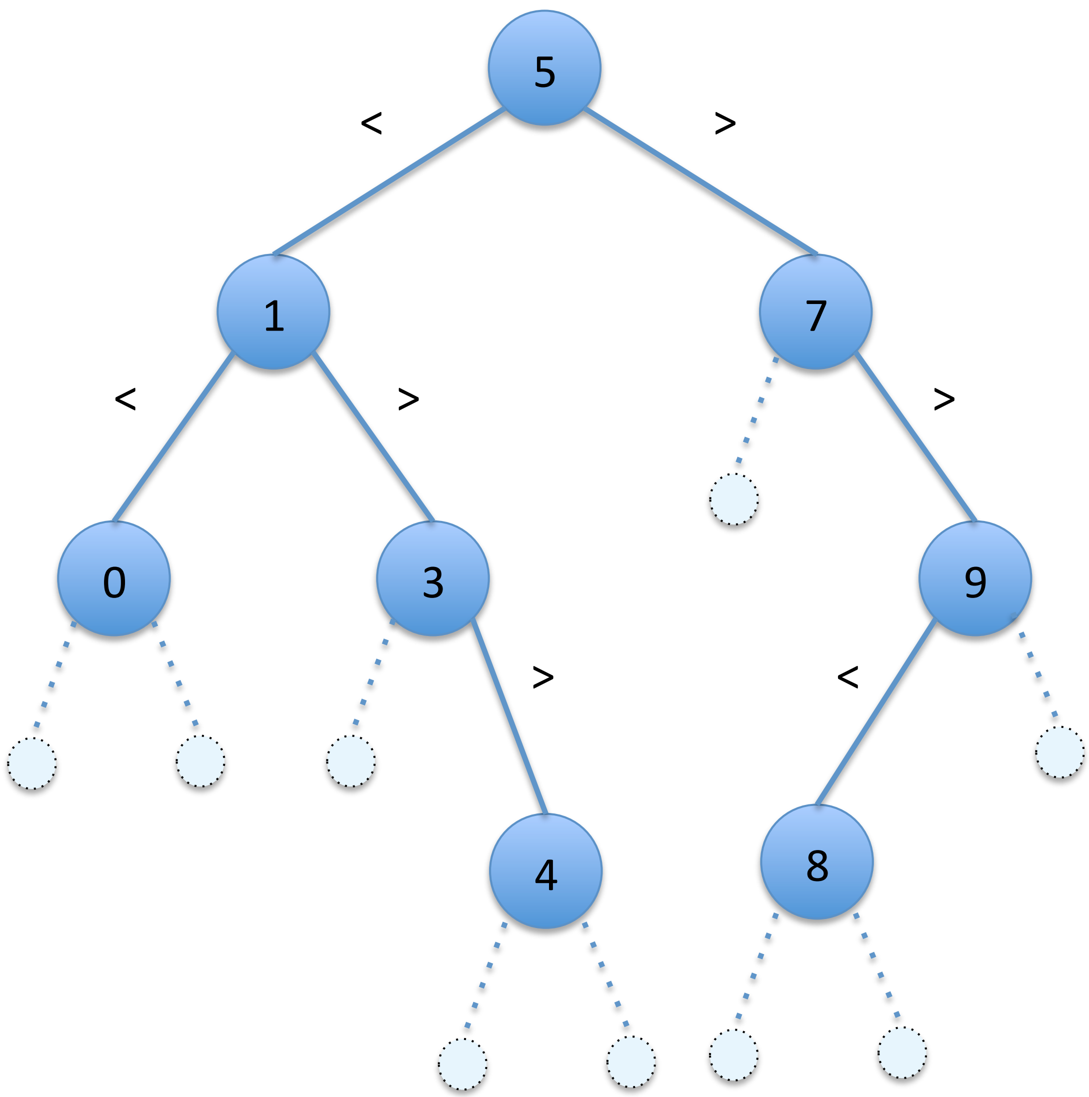
- Option 1:
  - Build a tree
  - Check that the BST invariants hold (unlikely!)
  - Impractically inefficient

- Option 2:
  - Write functions for building BSTs from other BSTs
    - e.g. "insert an element", "delete an element", …
  - Starting from some trivial BST (e.g. `Empty`), apply these functions to get the BST we want
  - If each of these functions *preserves* the BST invariants, then any tree we get from them will be a BST *by construction*
    - No need to check!
  - Ideally: "rebalance" the tree to make lookup efficient (NOT in CIS 120, see CIS 121)

Inserting a new node: `(insert t 4)`



Inserting a new node: `(insert t 4)`



Inserting Into a BST

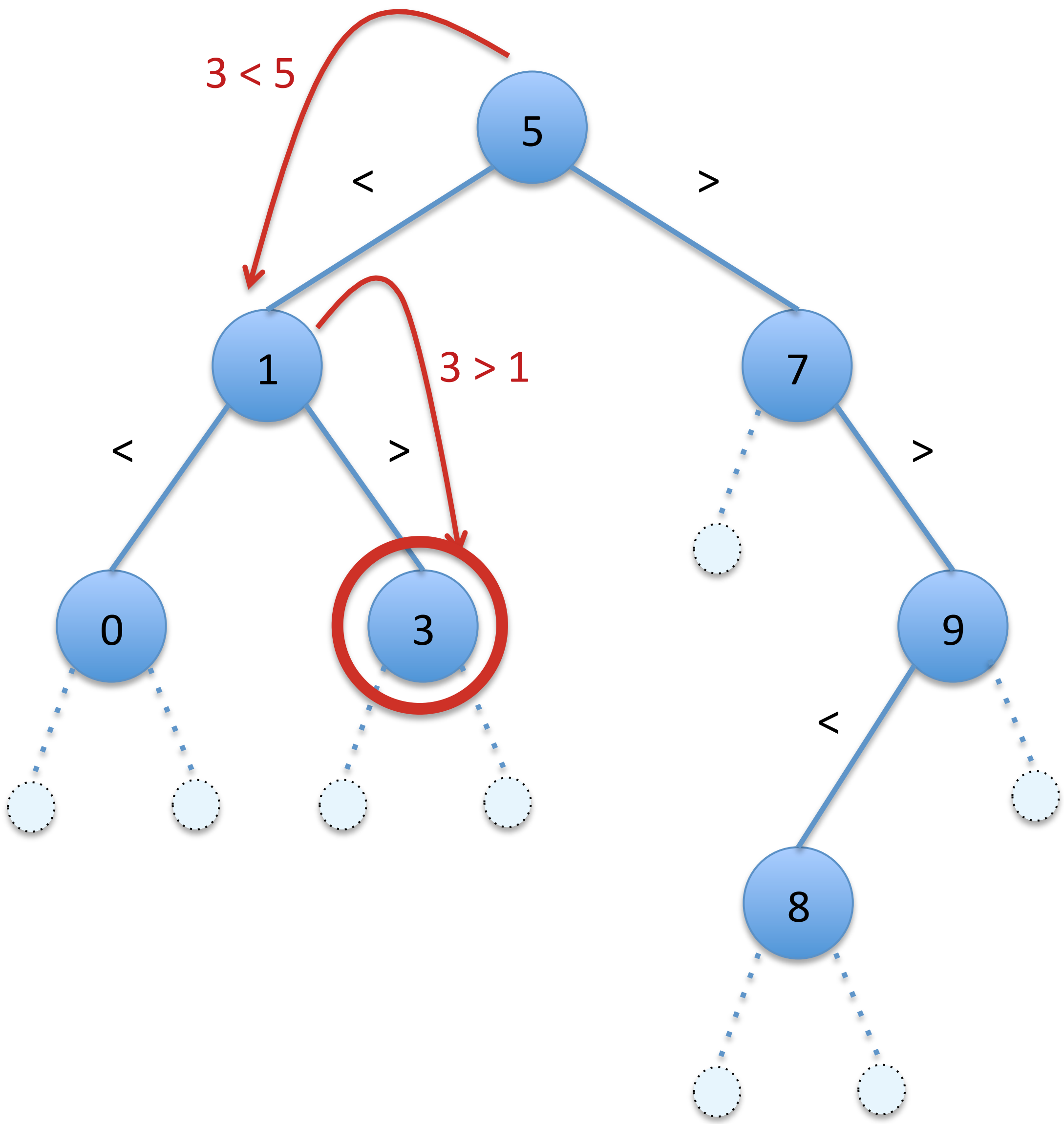
```
(* Insert n into the BST t *)
let rec insert (t:tree) (n:int) : tree =
  begin match t with
  | Empty -> Node(Empty,n,Empty)
  | Node(lt,x,rt) ->
     if x = n then t
    else if n < x then Node(insert lt n, x, rt)
    else Node(lt, x, insert rt n)
  end
```

- Note the similarity to searching the tree.
- Note that the result is a *new* tree with one more Node; the original tree is unchanged
- Assuming that t is a BST, the result is also a BST. (Why?)

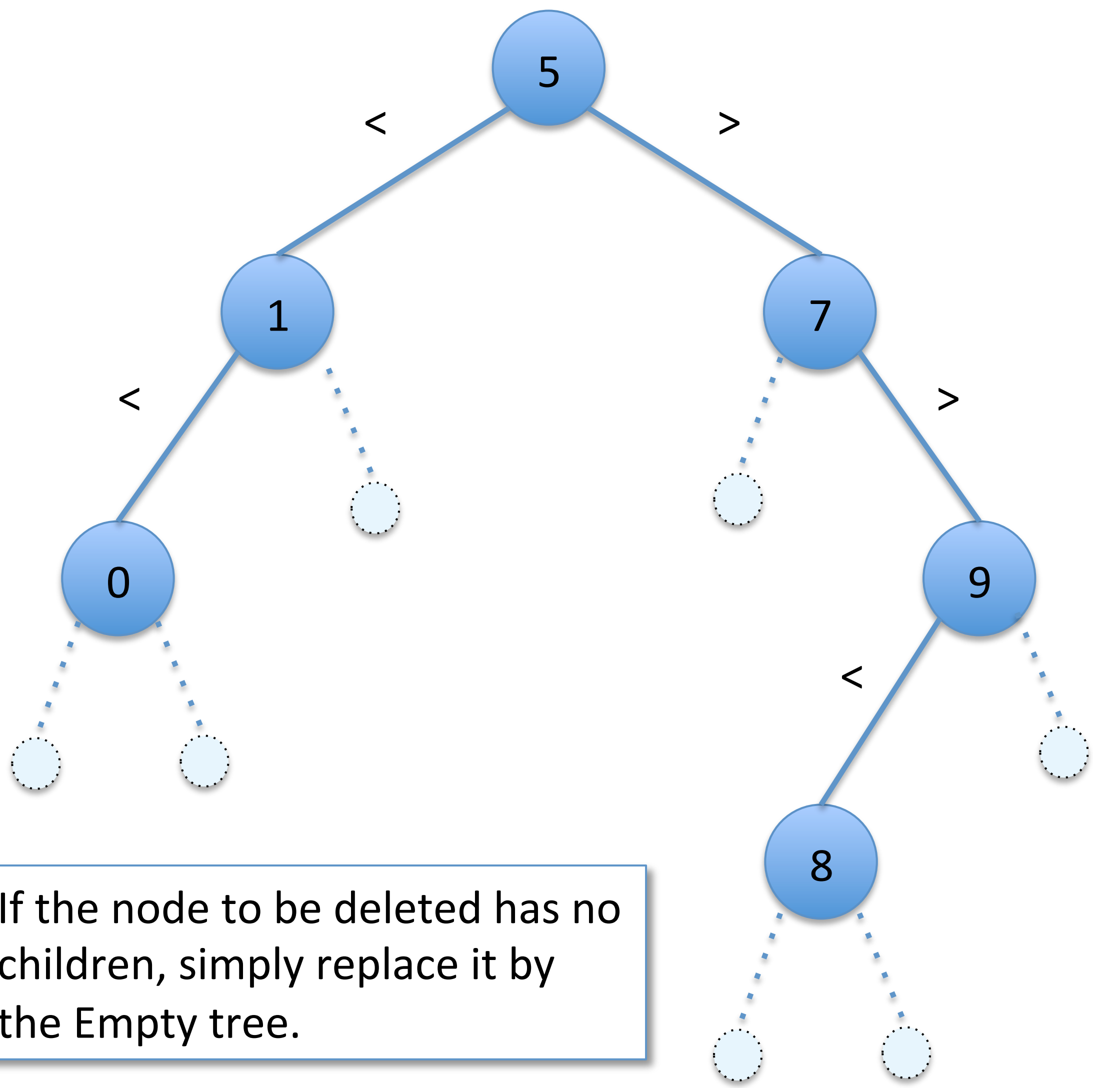
Constructing BSTs

Deleting an element

Deletion – No Children: `(delete t 3)`

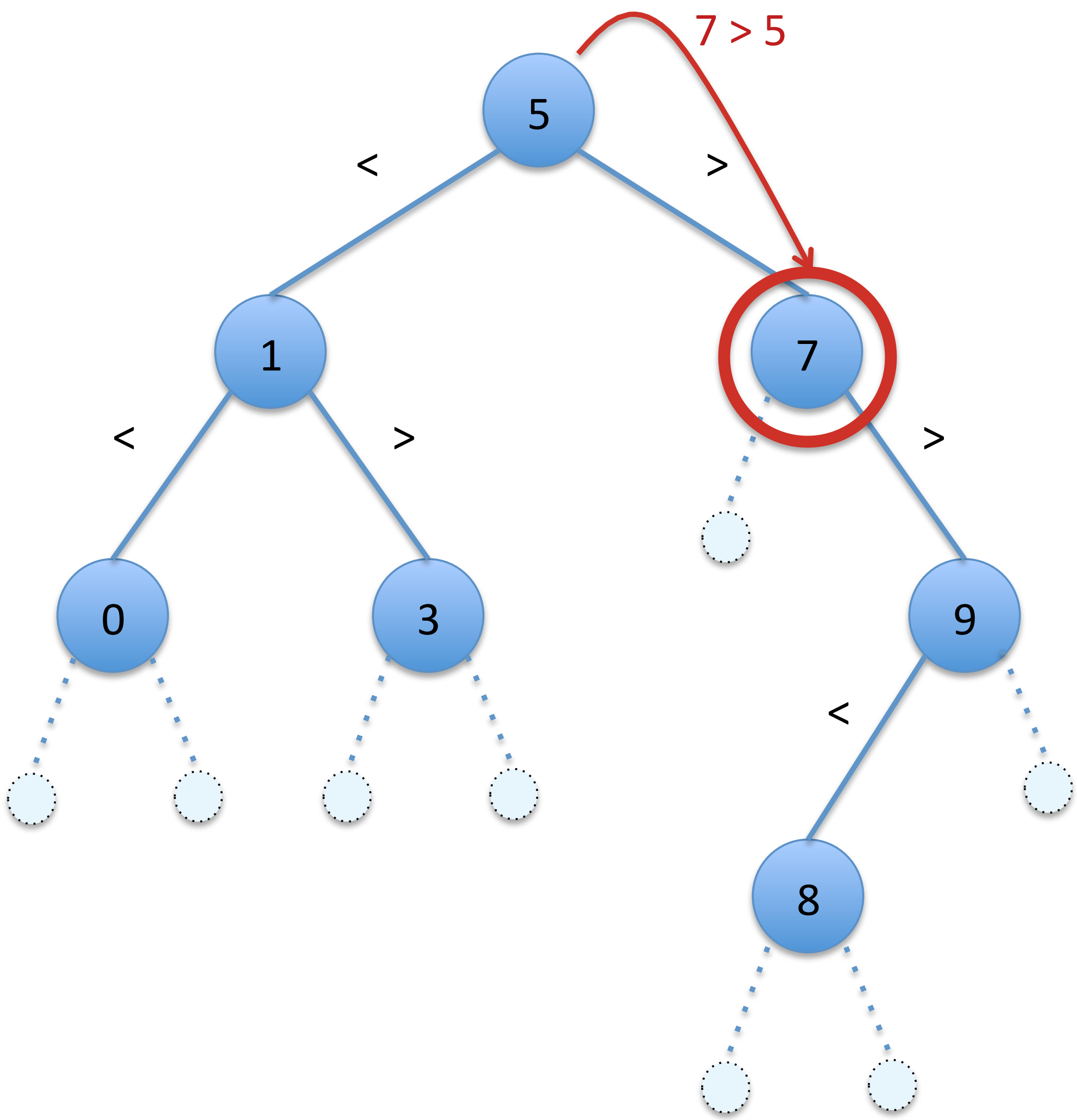


Deletion – No Children: `(delete t 3)`



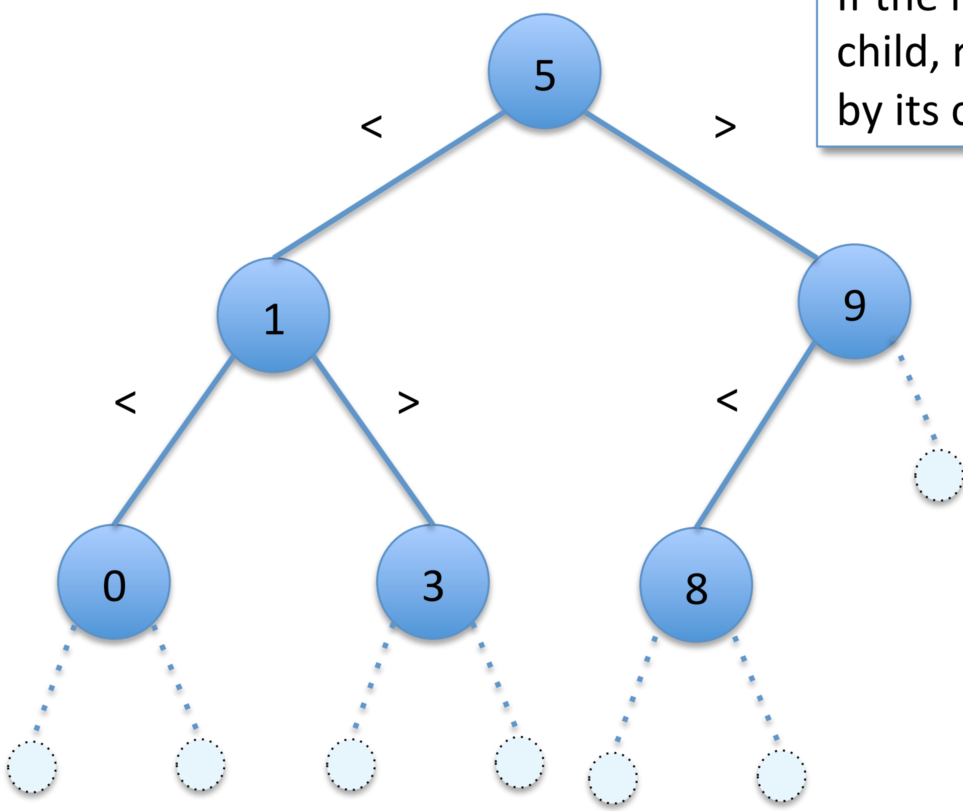
If the node to be deleted has no children, simply replace it by the Empty tree.

Deletion – One Child: `(delete t 7)`

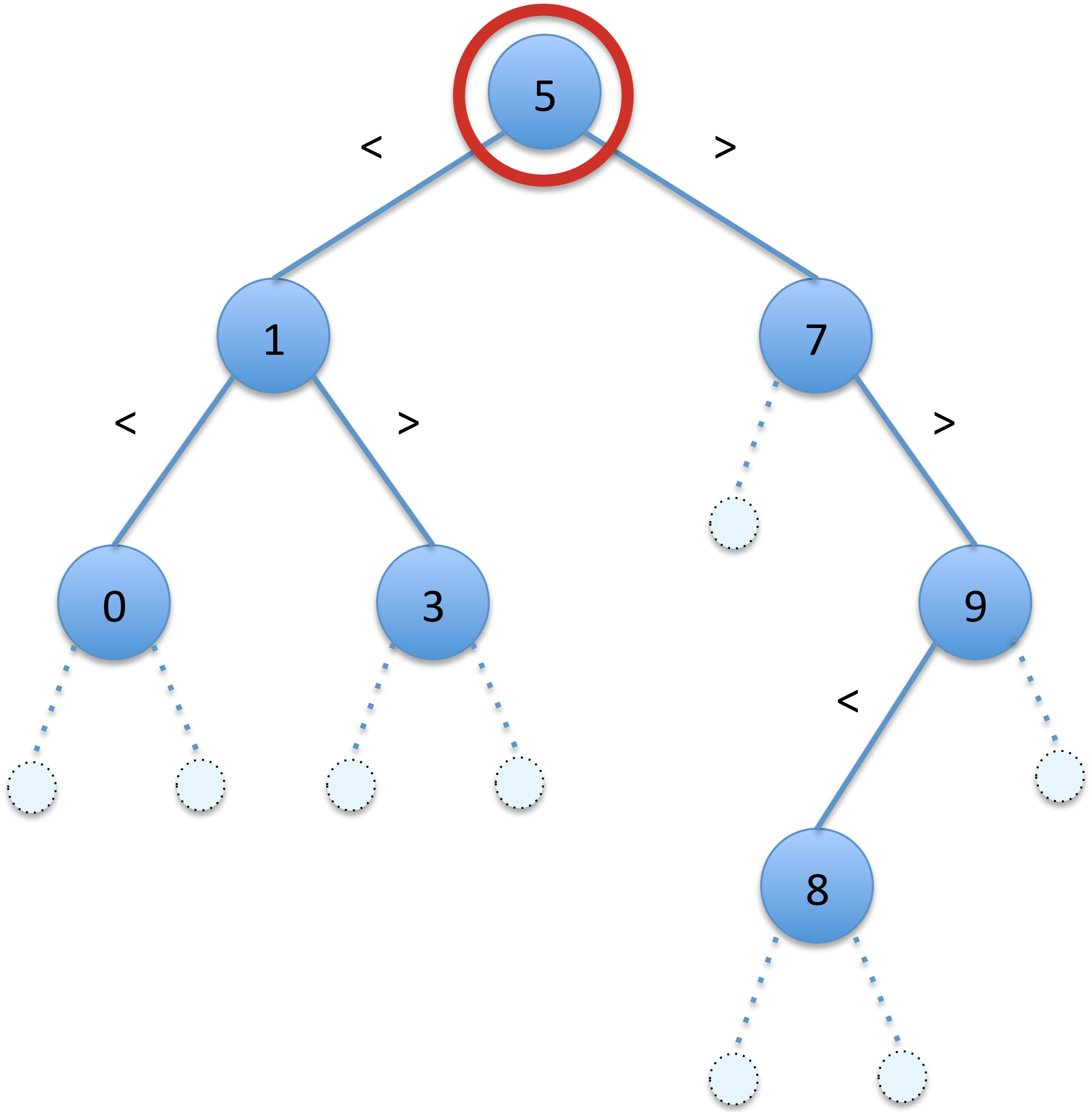


Deletion – One Child: `(delete t 7)`

If the node to be delete has one child, replace the deleted node by its child.

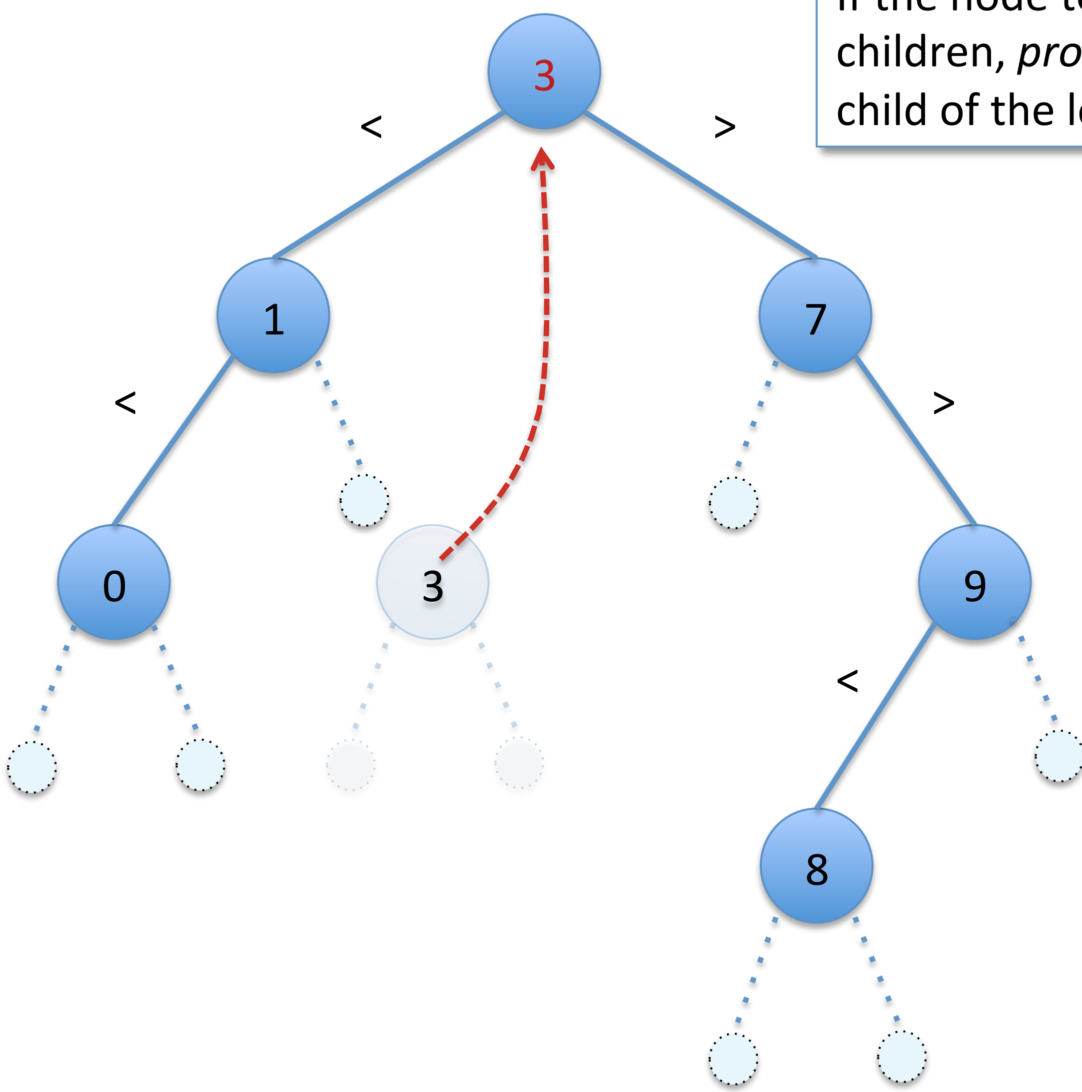


Deletion – Two Children: `(delete t 5)`



Deletion – Two Children: `(delete t 5)`

If the node to be delete has two children, *promote* the maximum child of the left tree.



Would it also work to move the *smallest* label from the *right-hand* subtree?

1. yes
2. no

Answer: yes