Programming Languages and Techniques (CIS1200)

Lecture 6

Binary Trees and Binary Search Trees

(Lecture notes Chapters 6 and 7)

CIS 1200 Announcements

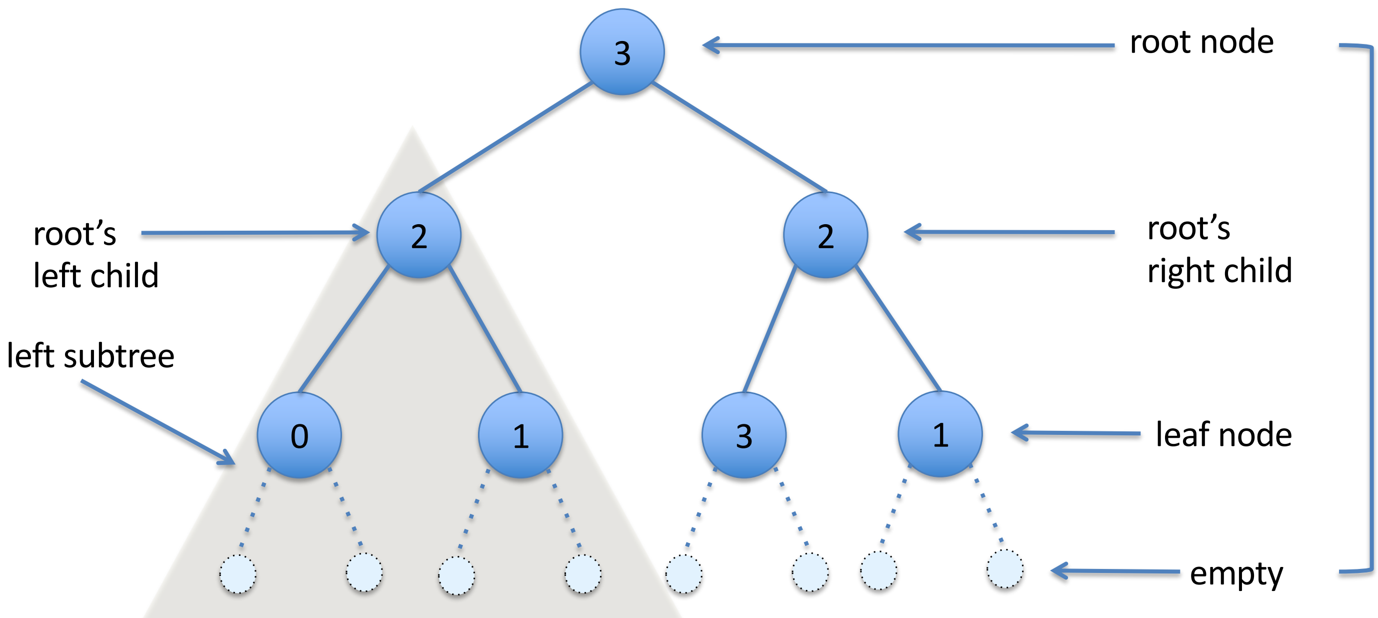
- HW02 available today, due next Tuesday at 11.59pm

- Please fill out the intro survey (coming soon, details on Ed)

Binary Trees

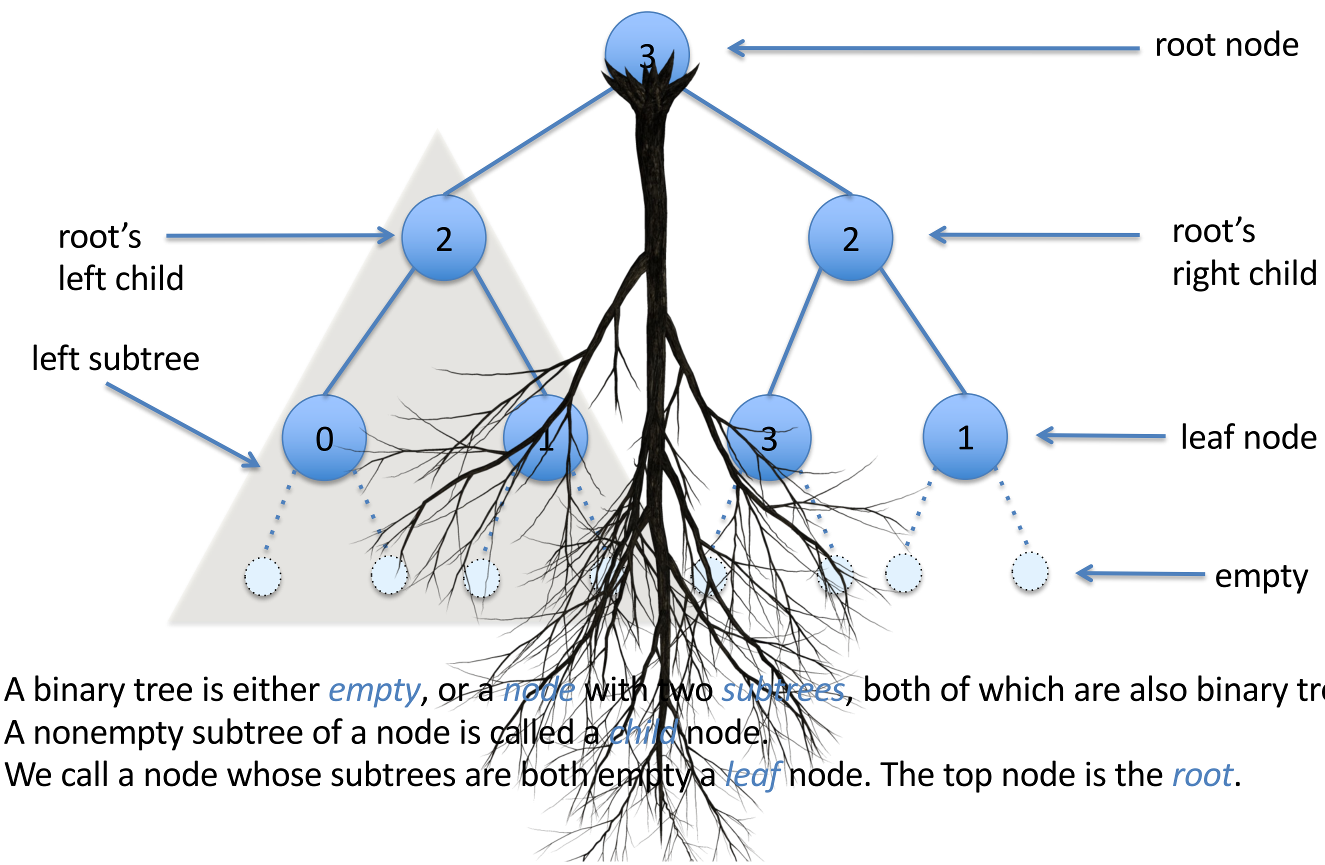
A particular form of tree-structured data

Binary Trees



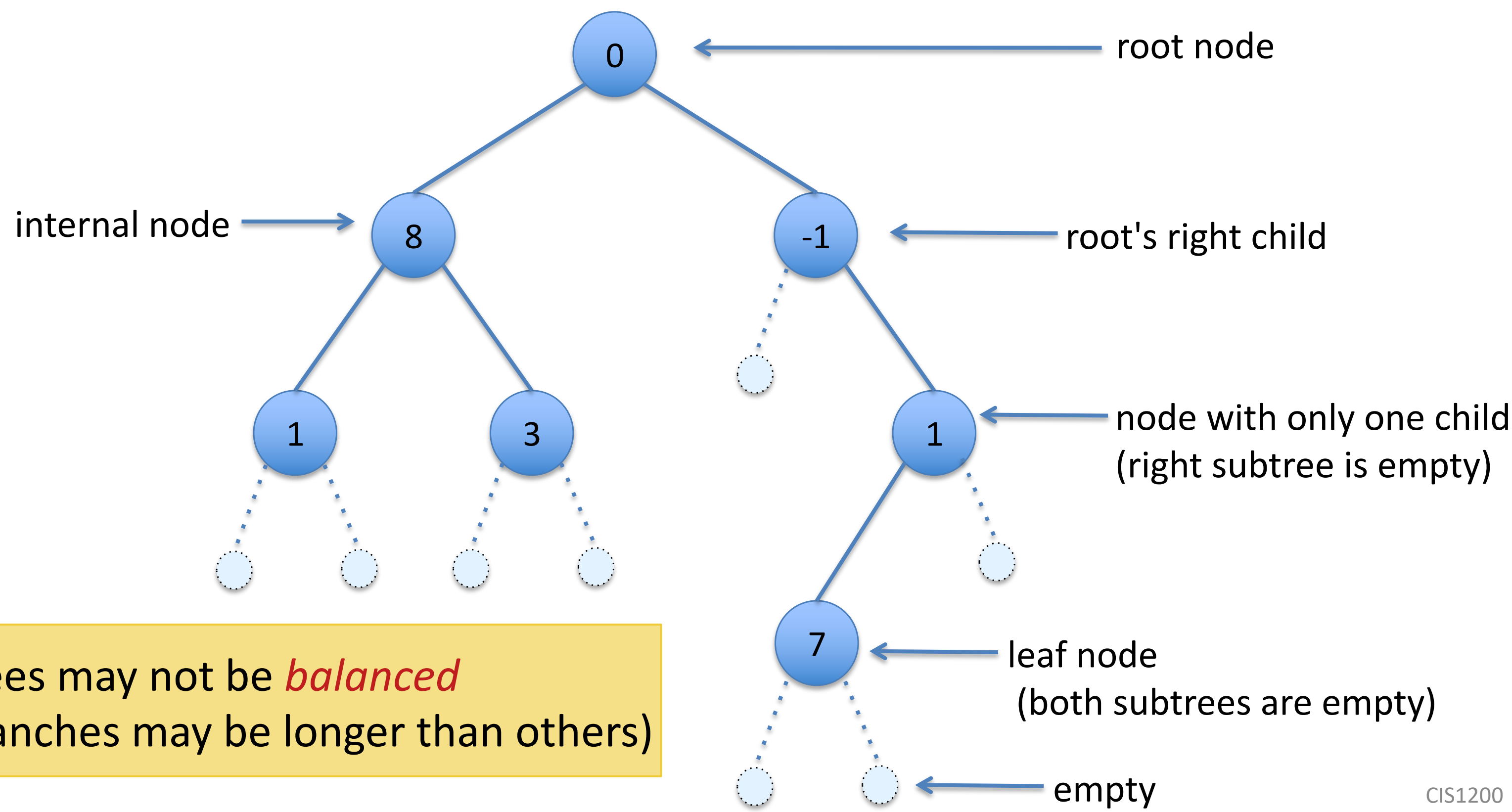
A binary tree is either *empty*, or a *node* with two *subtrees*, both of which are also binary trees.
A nonempty subtree of a node is called a *child* node.
We call a node whose subtrees are both empty a *leaf* node. The top node is the *root*.

Trees are drawn upside-down



A binary tree is either *empty*, or a *node* with two *subtrees*, both of which are also binary trees.
A nonempty subtree of a node is called a *child* node.
We call a node whose subtrees are both empty a *leaf* node. The top node is the *root*.

Another Binary Tree



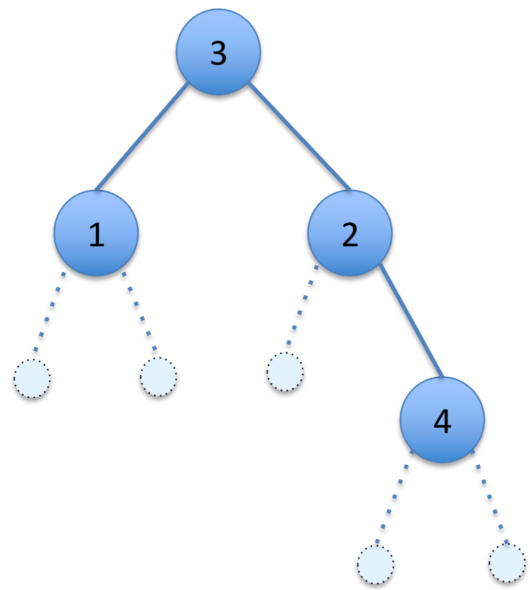
Binary trees may not be *balanced*
(some branches may be longer than others)

Binary Trees in OCaml

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

```
let t : tree =
  Node (Node (Empty, 1, Empty),
    3,
    Node (Empty, 2,
      Node (Empty, 4, Empty)))
```

=



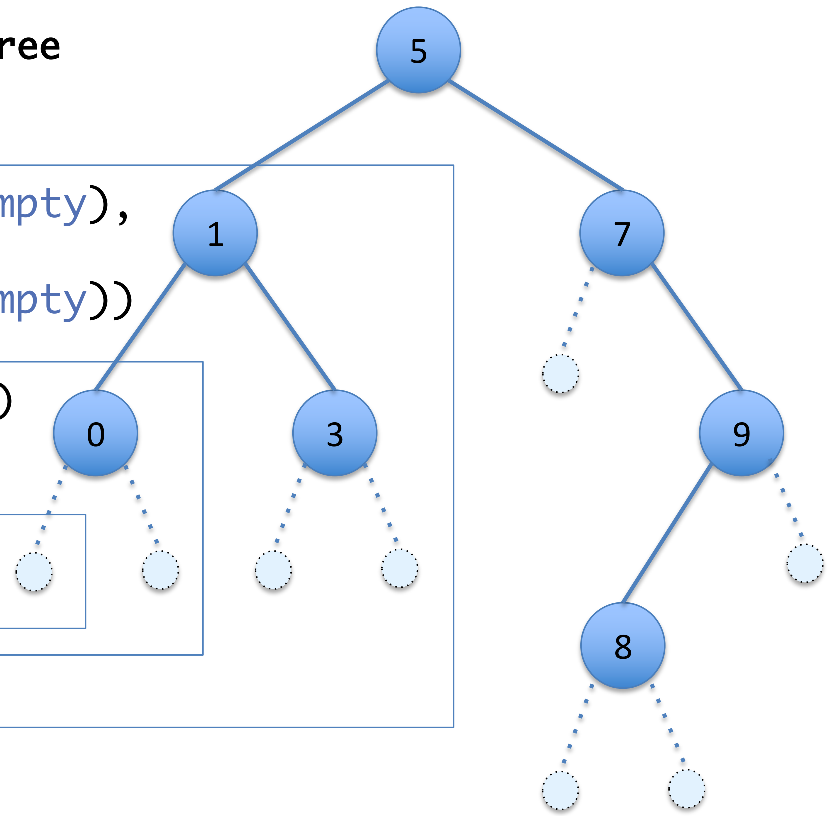
Representing trees

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

```
Node (Node (Empty, 0, Empty),
      1,
      Node (Empty, 3, Empty))
```

```
Node (Empty, 0, Empty)
```

```
Empty
```



Working with binary trees

see tree.ml

treeExamples.ml

Some functions on trees

```
(* counts the number of nodes in the tree *)
let rec size (t:tree) : int =
  begin match t with
  | Empty -> 0
  | Node(l,_,r) -> 1 + (size l) + (size r)
  end

(* length of longest path from the root to a leaf *)
let rec height (t:tree) : int =
  begin match t with
  | Empty -> 0
  | Node(l,_,r) -> 1 + max (height l) (height r)
  end
```

Structural Recursion Over *Trees*

Structural recursion builds an answer from smaller components:

```
let rec f (t : tree) … : … =
  begin match t with
  | Empty -> …
  | Node(l,x,r) -> … (f l …) … x … (f r …) …
  end
```

The branch for `Empty` calculates the value (`f Empty`) directly.

– this is the *base case* of the recursion

The branch for `Node(l,x,r)` calculates
`(f (Node(l,x,r))` given `x` and `(f l)` and `(f r)`.

– this is the *inductive case* of the recursion

Tree vs. List Recursion

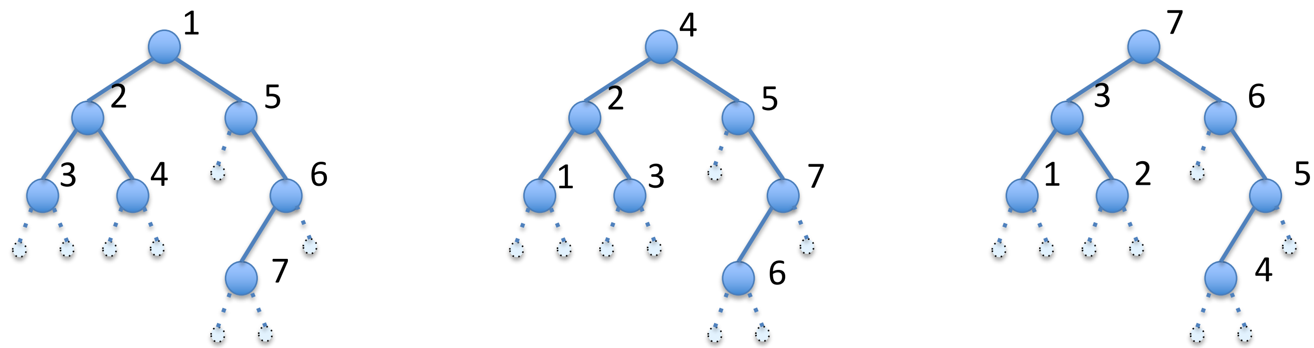
```
let rec f (t : tree) … : … =
  begin match t with
  | Empty -> …
  | Node(l,x,r) -> … (f l …) … x … (f r …) …
  end
```

Two recursive calls, for left and right sub trees, versus *one* for lists.

```
let rec f (l : … list) … : … =
  begin match l with
  | [] -> …
  | ( hd :: rest ) -> … hd … (f rest …) …
  end
```

Tree Traversals

Recursive Tree Traversals



Pre-Order
`root @ left @ right`

In Order
`left @ root @ right`

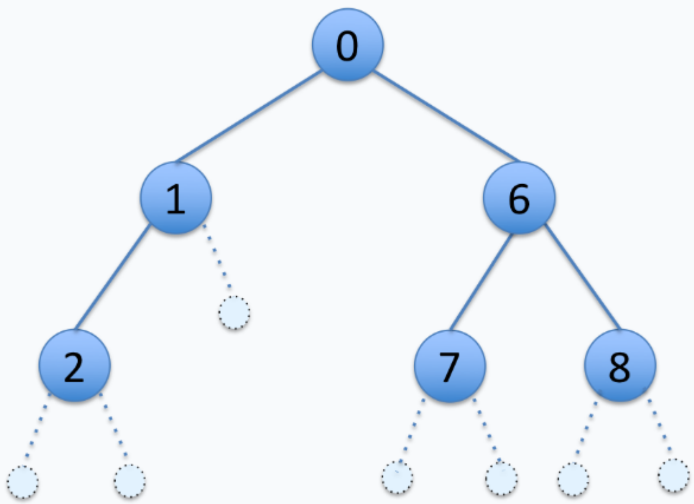
Post-Order
`left @ right @ root`

```
let rec f (t:tree) : int list =
  begin match t with
  | Empty -> []
  | Node(l, x, r) ->
    let root  = [ x ] in (* process root *)
    let left  = f l   in (* recursive call left subtree *)
    let right = f r   in (* recursive call right subtree *)
    … combine root, left, and right …
  end
```

Traversals vary the order in which these are combined…

6: In what sequence will the nodes of this tree be visited by a post-order traversal?

0



Post-Order
Left – Right – Root

[0;1;6;2;7;8] 0%

[0;1;2;6;7;8] 0%

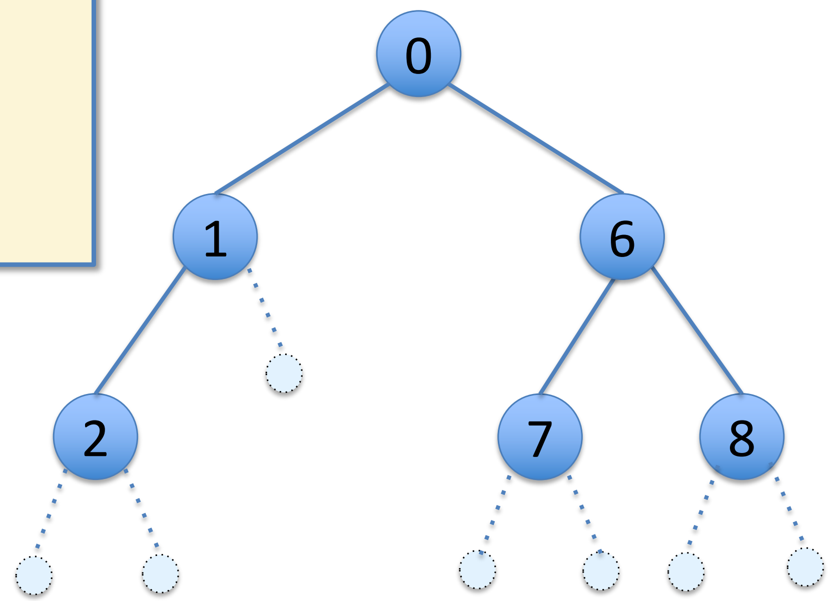
[2;1;0;7;6;8] 0%

[7;8;6;2;1;0] 0%

[2;1;7;8;6;0] 0%

In what sequence will the nodes of this tree be visited by a post-order traversal?

1. [0;1;6;2;7;8]
2. [0;1;2;6;7;8]
3. [2;1;0;7;6;8]
4. [7;8;6;2;1;0]
5. [2;1;7;8;6;0]



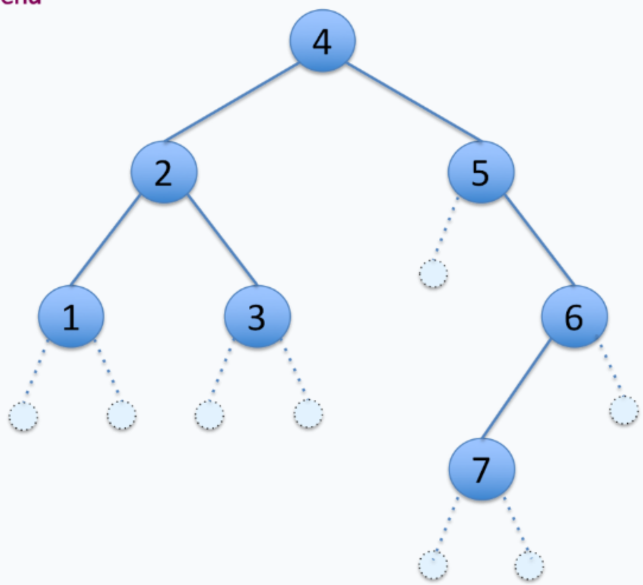
Post-Order
left @ right @ root

Answer: 5

6: What is the result of applying this function on this tree?

0

```
let rec inorder (t:tree) : int list =
  begin match t with
        | Empty -> []
        | Node (left, x, right) ->
                inorder left @ (x :: inorder
right)
end
```



[]

0%

[1;2;3;4;5;6;7]

0%

[1;2;3;4;5;7;6]

0%

[4;2;1;3;5;6;7]

0%

[4]

0%

[1;1;1;1;1;1;1]

0%

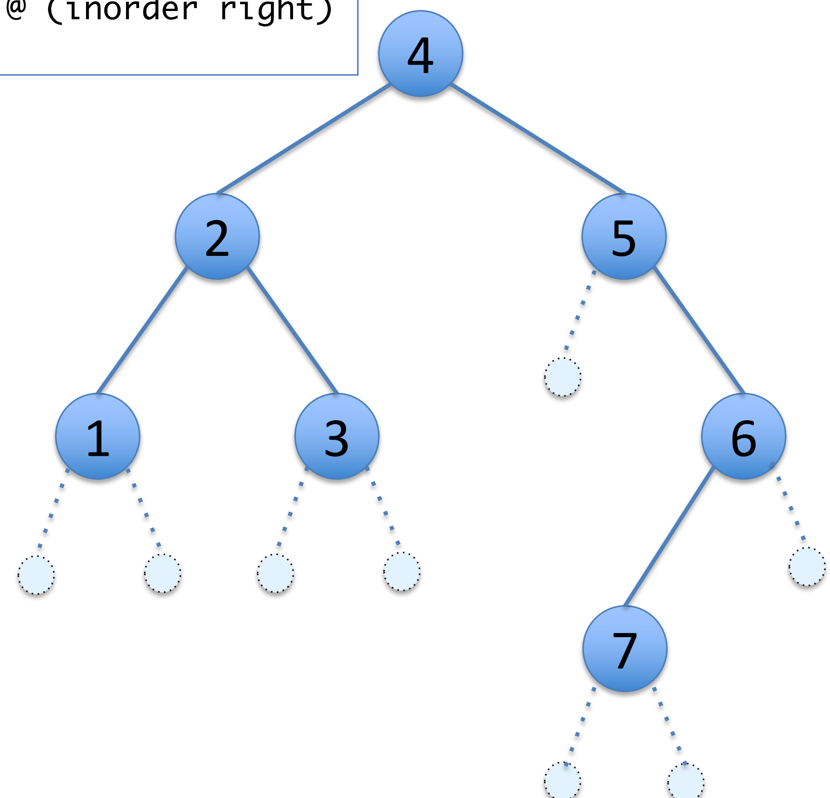
none of the above

0%

```
let rec inorder (t:tree) : int list =
  begin match t with
    | Empty -> []
    | Node (left, x, right) ->
        (inorder left) @ [x] @ (inorder right)
end
```

What is the result of applying this function on this tree?

1. []
2. [1;2;3;4;5;6;7]
3. [1;2;3;4;5;7;6]
4. [4;2;1;3;5;6;7]
5. [4]
6. [1;1;1;1;1;1;1]
7. none of the above



Answer: 3

Trees as Containers

See tree.ml and treeExamples.ml

Trees as Containers

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

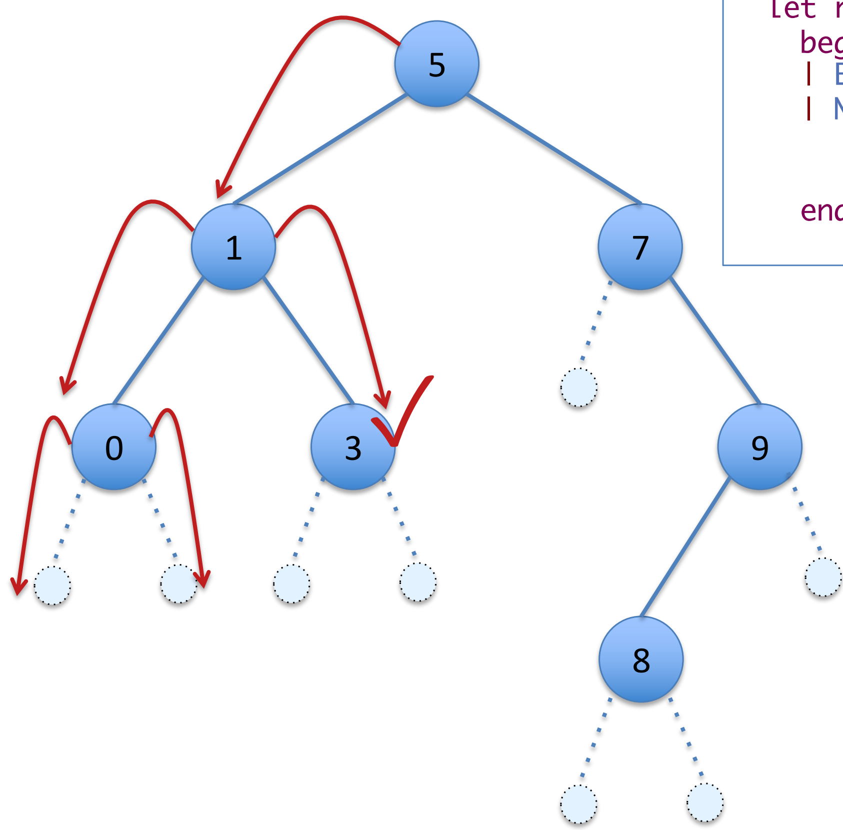
```
type tree =
| Empty
| Node of tree * int * tree
```

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
```

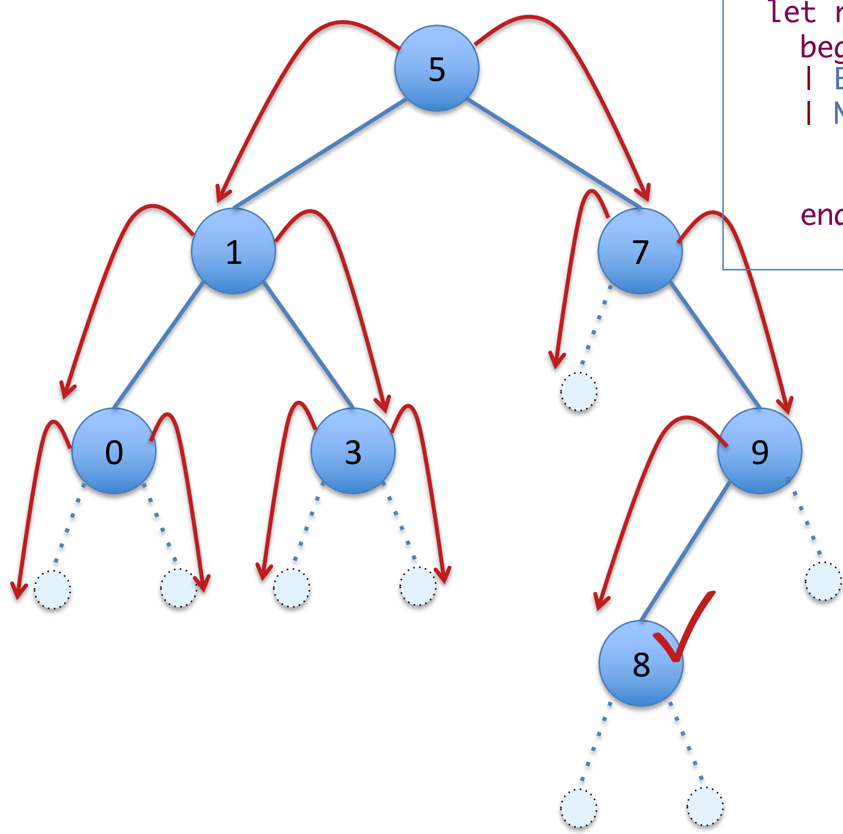
- This function searches through the tree `t`, looking for a number n
- The || operator is a *short-circuiting "or"*
  - When computing b||c, if b simplifies to `true`, then c is ignored
  - This can save time if simplifying c is expensive
- Even so, `contains` might have to traverse the *entire tree*

Search during `(contains t 3)`




```
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
```

Search during (contains t 8)




```
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
```

Ordered Trees

Big idea: find things faster by searching less

Key Insight:

Ordered data can be searched more quickly

- This is why dictionaries are arranged alphabetically
- But it requires the ability to focus on (roughly) *half* of the current data

Binary Search Trees

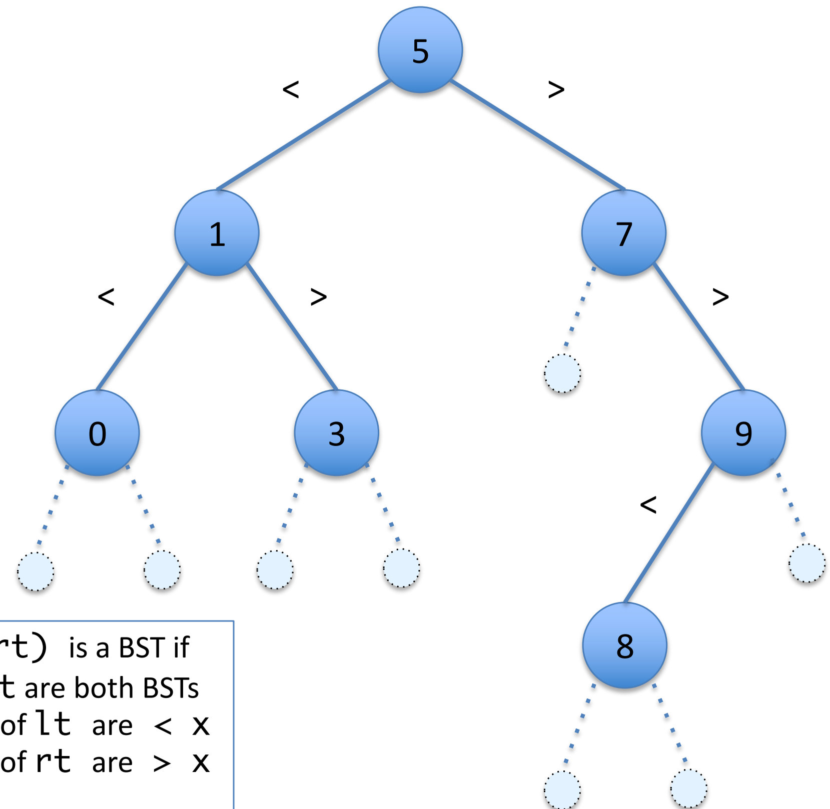
- A *binary search tree* (BST) is a binary tree with an additional *invariant*:*

> - `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

- *The BST invariant means that container functions can take time proportional to the **height** instead of the **size** of the tree.*

*A 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

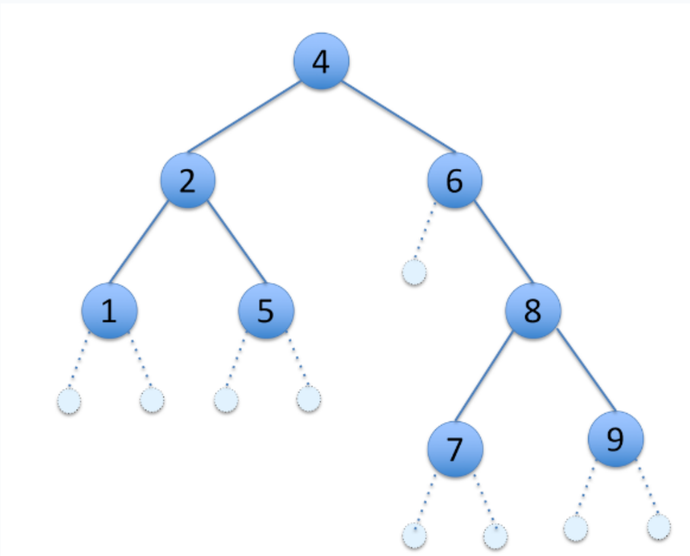


- `Node(lt,x,rt)` is a BST if
  - `lt` and `rt` are both BSTs
  - all nodes of `lt` are < x
  - all nodes of `rt` are > x
- `Empty` is a BST

Note that the BST invariants hold for this tree!

6: Is this a BST?

0



No

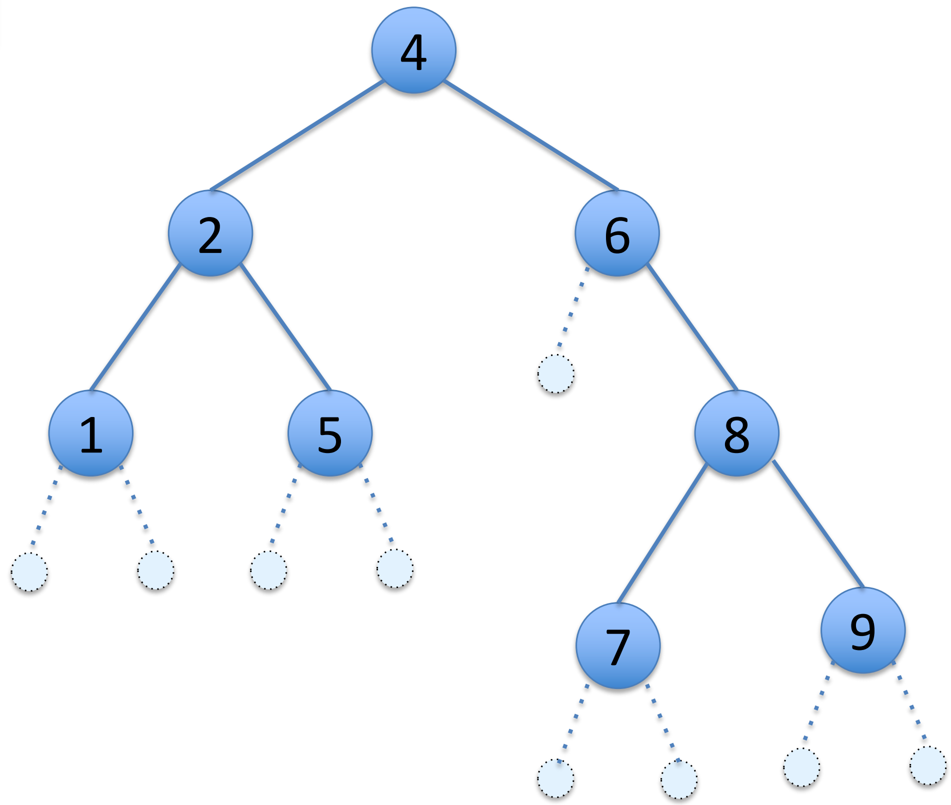
0%

Yes

0%

Is this a BST??

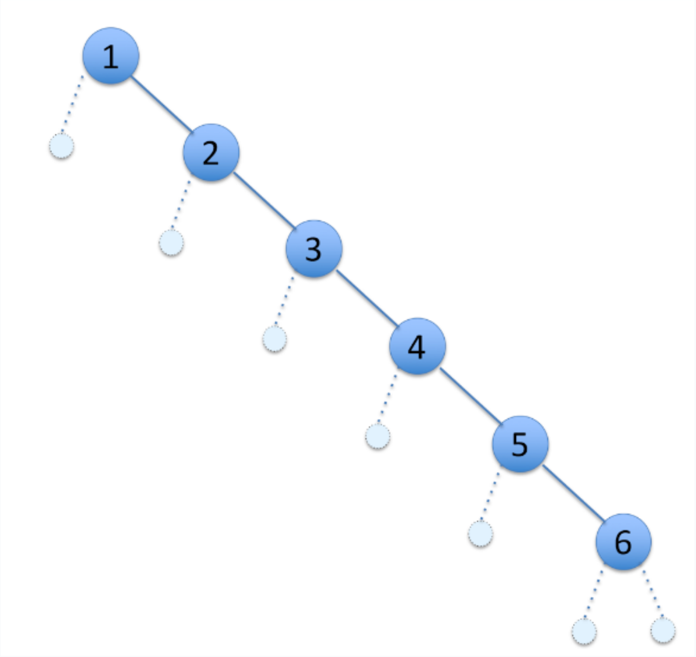
1. yes
2. no



Answer: no, 5 to the left of 4

6: Is this a BST?

✓ 0



No

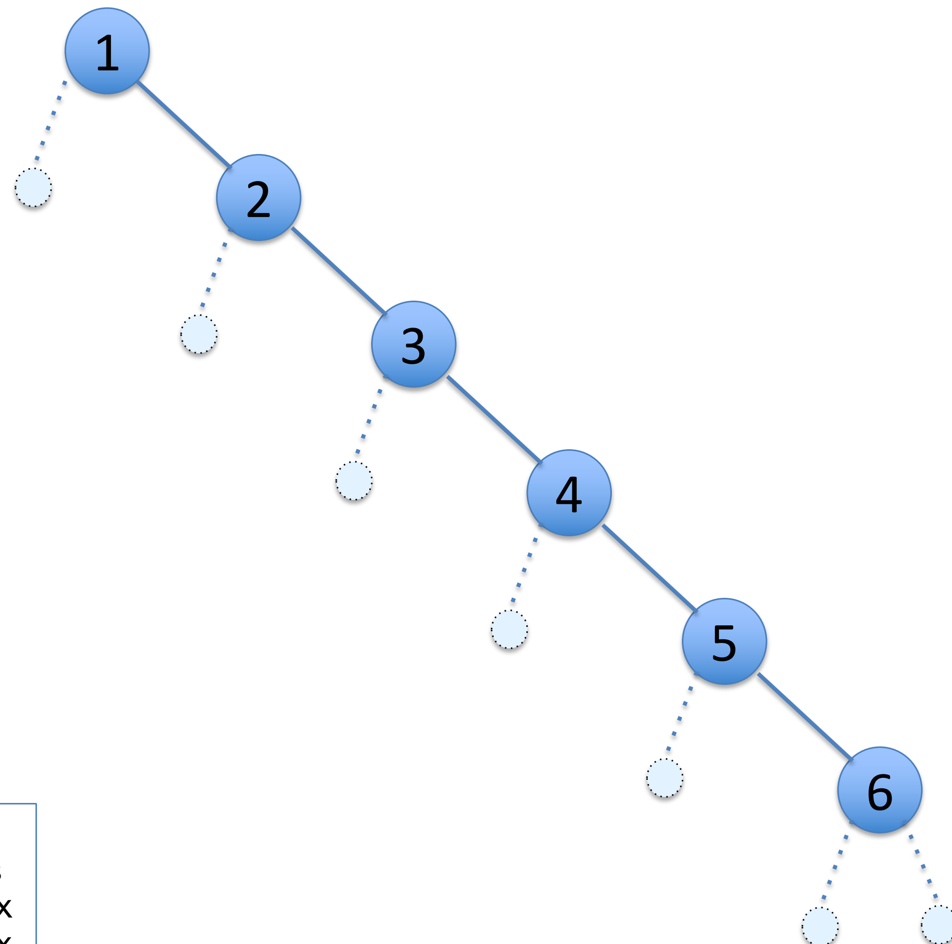
0%

Yes

0%

Is this a BST??

1. yes
2. no

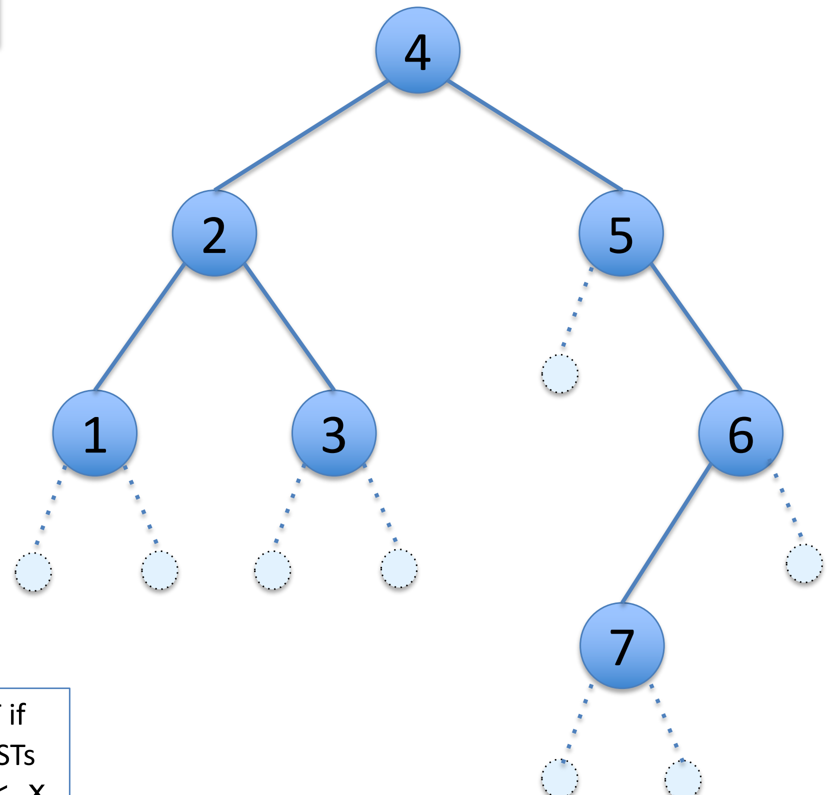


- Node(lt,x,rt) is a BST if
  - lt and rt are both BSTs
  - all nodes of lt are < x
  - all nodes of rt are > x
- Empty is a BST

Answer: Yes

Is this a BST??

1. yes
2. no

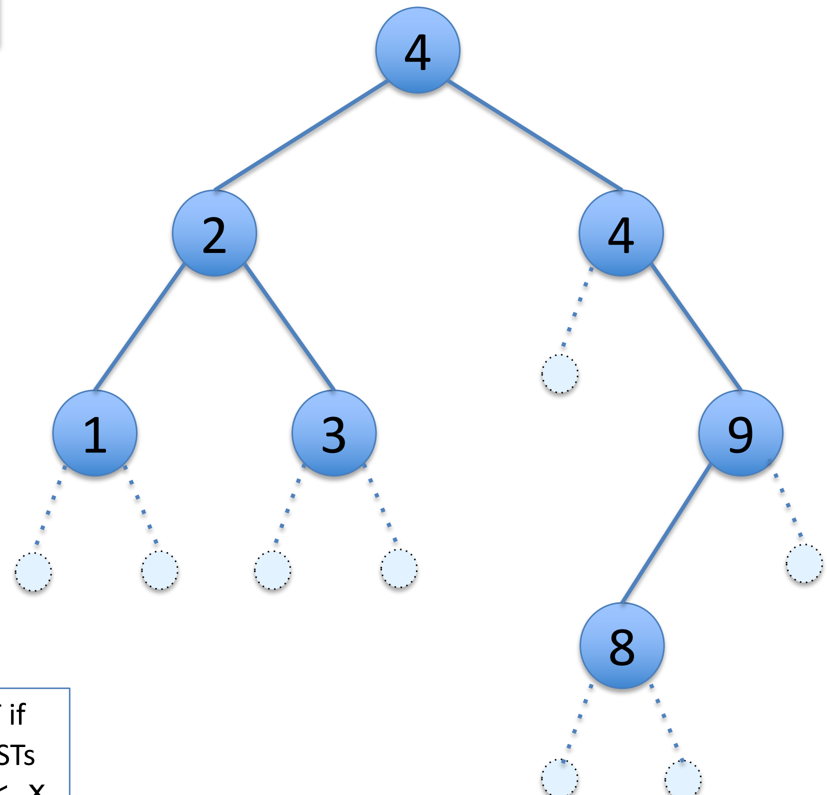


- Node(lt,x,rt) is a BST if
  - lt and rt are both BSTs
  - all nodes of lt are < x
  - all nodes of rt are > x
- Empty is a BST

Answer: no, 7 to the left of 6

Is this a BST??

1. yes
2. no

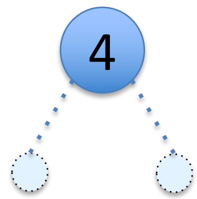


- `Node(lt,x,rt)` is a BST if
  - `lt` and `rt` are both BSTs
  - all nodes of `lt` are < `x`
  - all nodes of `rt` are > `x`
- `Empty` is a BST

Answer: 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

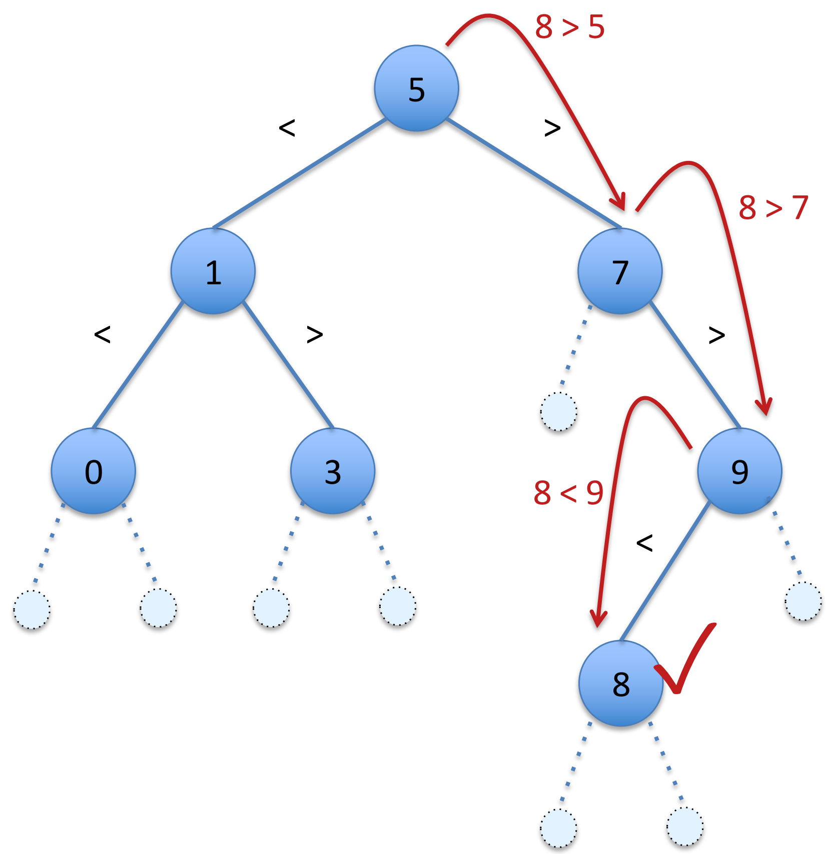
Is this a BST??

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

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
```

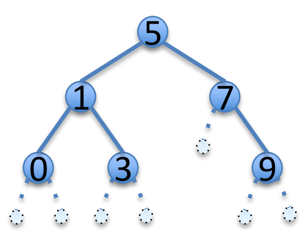
- The BST invariants guide the search.
- Note that lookup may return an incorrect answer if the input is *not* a BST!
  - This function *assumes* that the BST invariants hold of t.

Demo

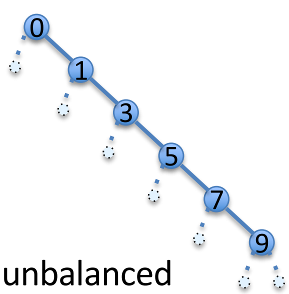
bst.ml – compare contains and lookup

BST Performance

- `lookup` takes time proportional to the *height* of the tree.
  - not the *size* of the tree (as we saw for `contains` on unordered trees)
- 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

Manipulating BSTs

Inserting an element

insert : tree -> int -> tree

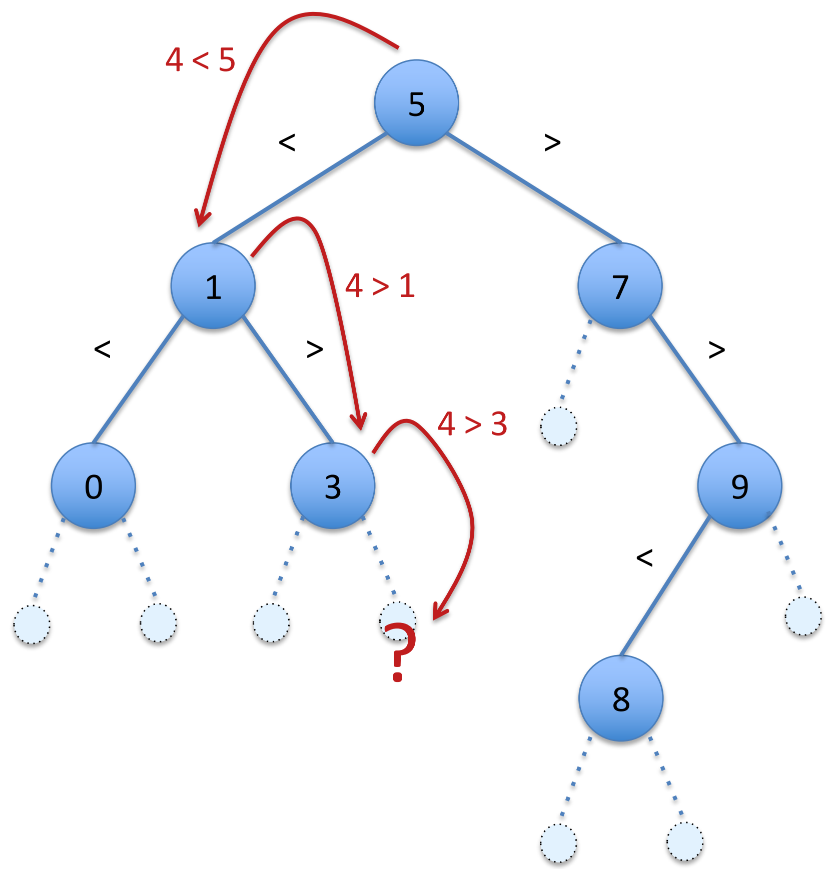
Inserting into a BST

- Suppose we have a BST `t` and a new element n, and we wish to compute a new BST containing all the elements of `t` together with n
  - Need to make sure the tree we build is really a BST – i.e., make sure to put n in the right place!

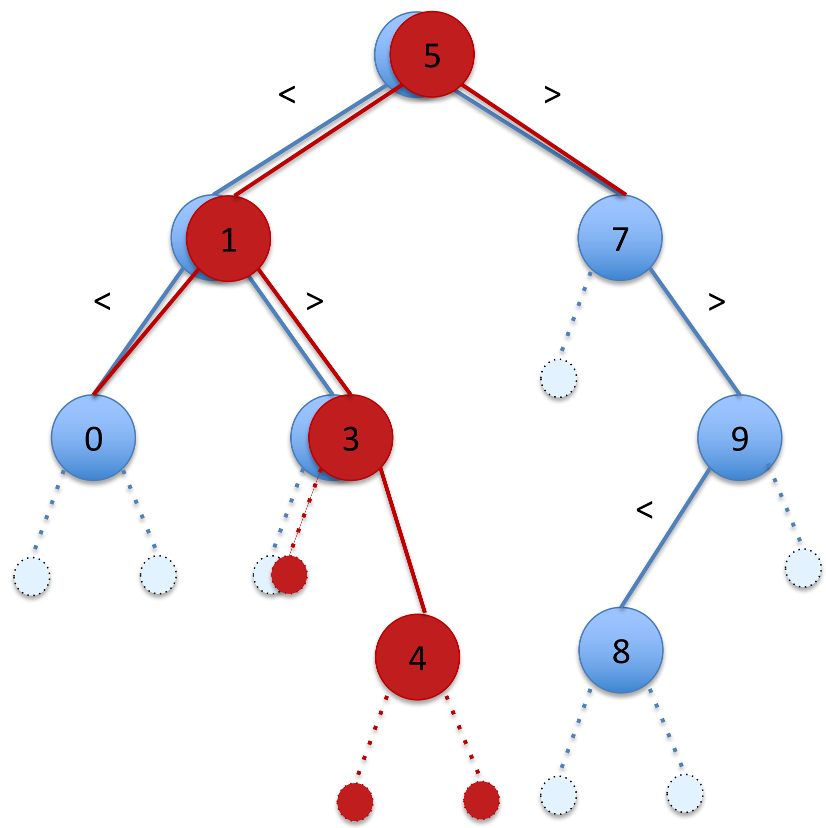
- This way we can build a BST containing any set of elements we like:
  - Starting from the `Empty` BST, apply this function repeatedly to get the BST we want
  - If insertion *preserves* the BST invariants, then any tree we get from it will be a BST *by construction*
    - No need to check!
  - Later: we can also “rebalance” the tree to make lookup even more efficient
    - (NOT in CIS 120; see CIS 121)

First step: find the right place…

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
```

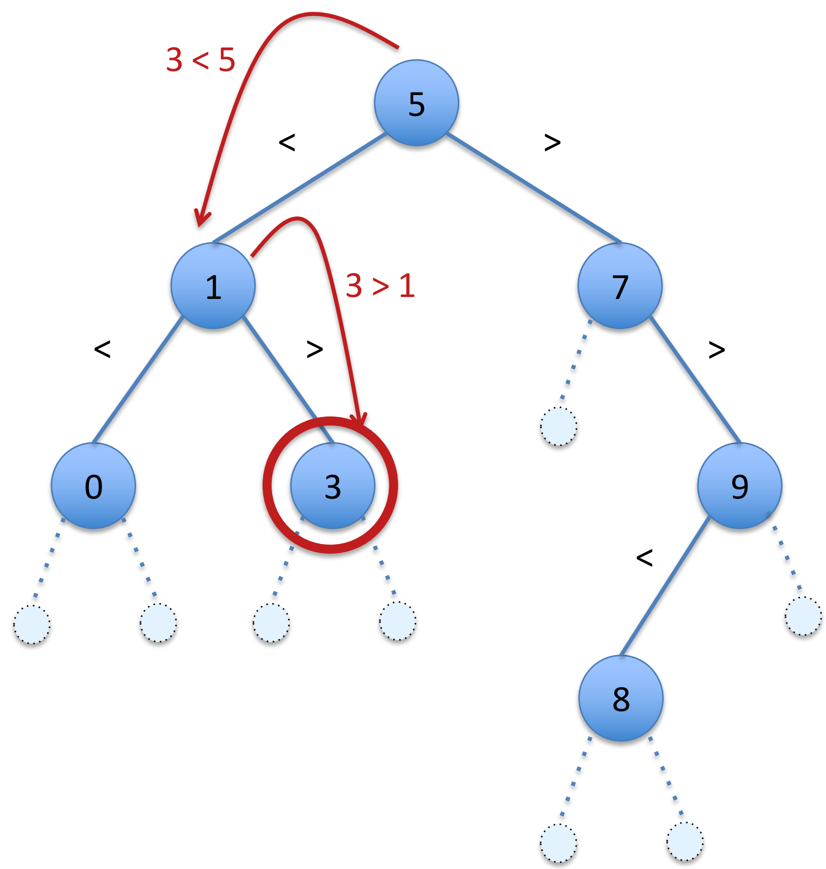
- Note similarity to searching the tree
- If t is a BST, the result is also a BST (why?)
- The result is a *new* tree with (possibly) one more Node; the original tree is unchanged

Critical point!

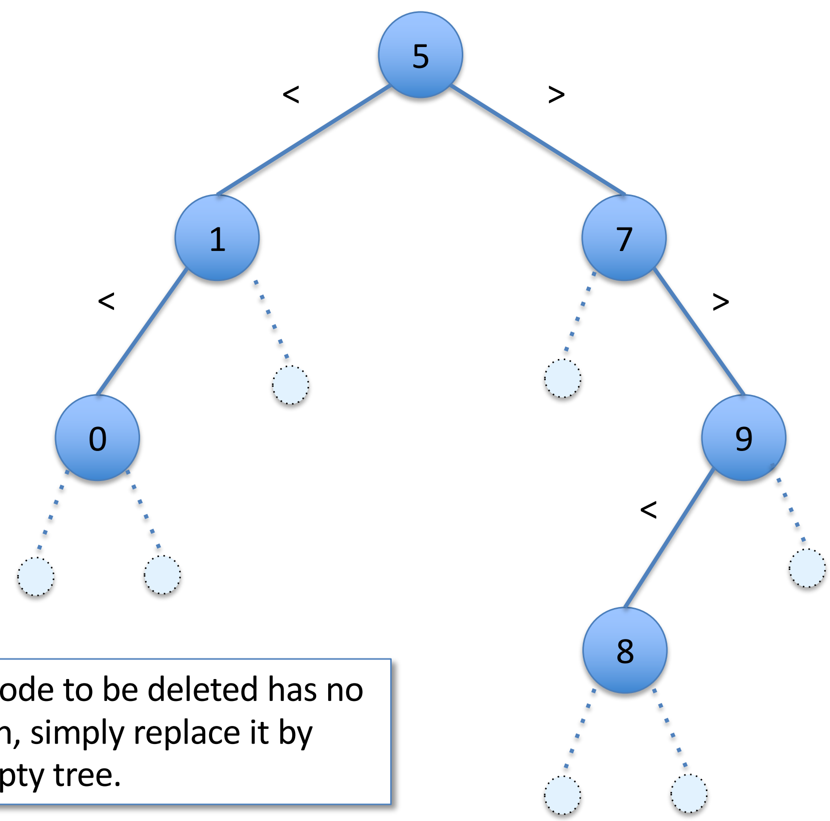
Deleting an Element from a BST

delete : tree -> int -> tree

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

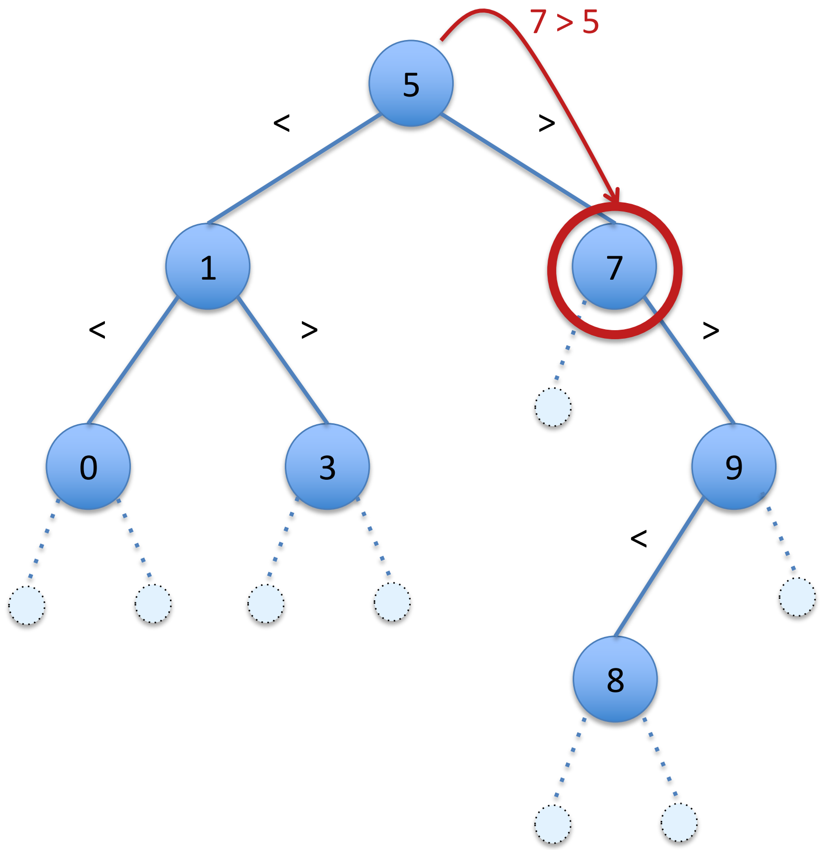


Deletion – No Children: `(delete t 3)`

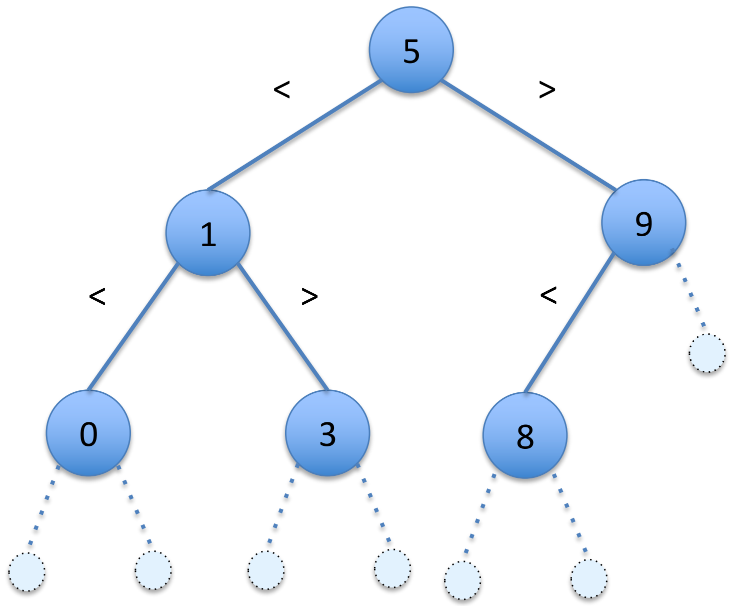


If the node to be deleted has no children, simply replace it by the Empty tree.

Deletion – One Child: `(delete t 7)`

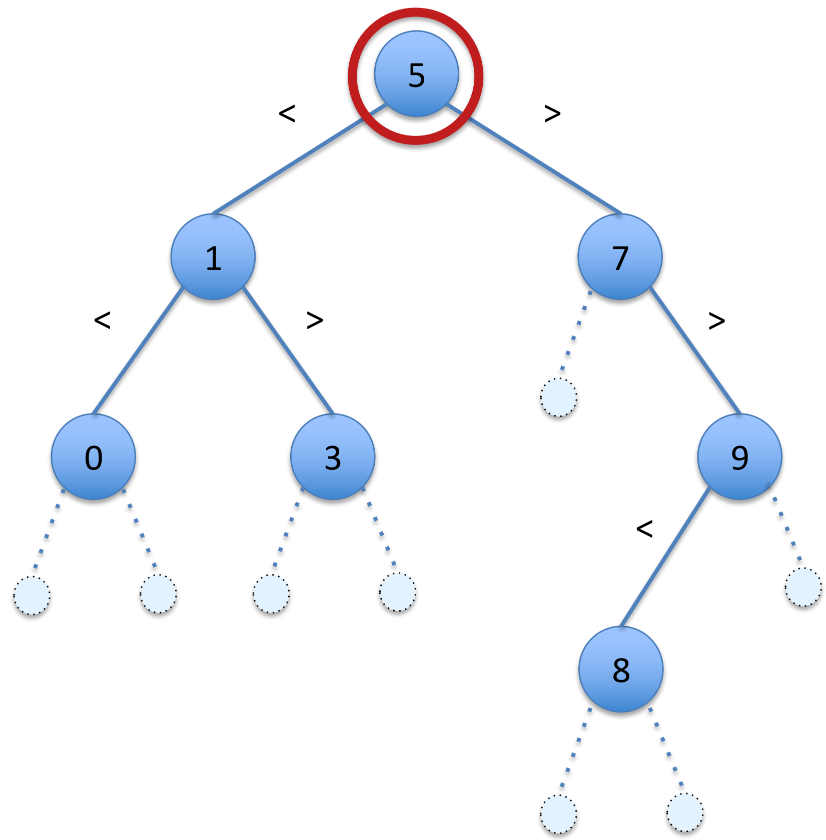


Deletion – One Child: `(delete t 7)`



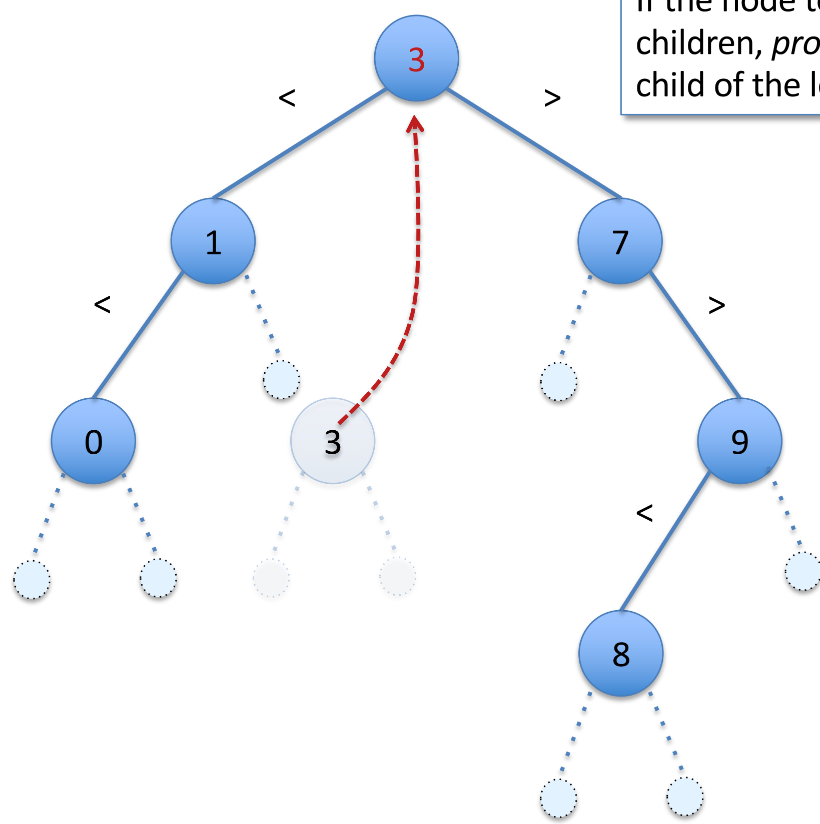
If the node to be delete has one child, replace the deleted node by the child.

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.

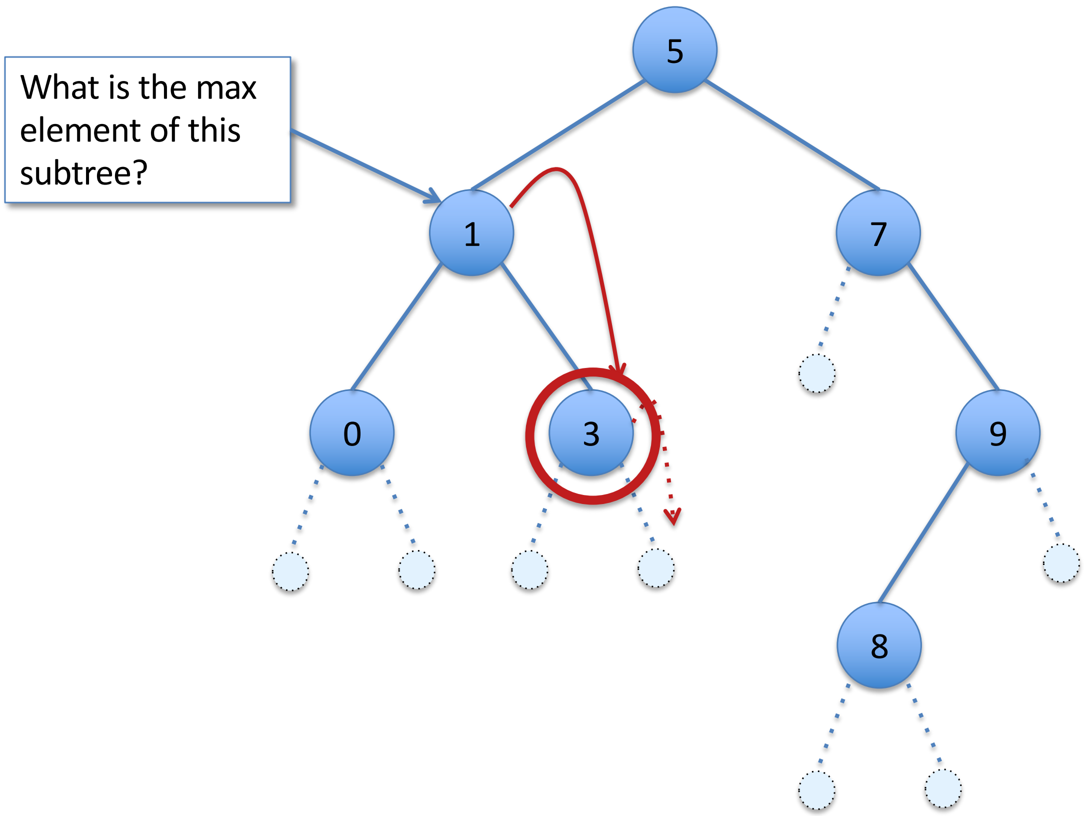


Subtleties of the Two-Child Case

- Suppose Node(lt,x,rt) is to be deleted and lt and rt are both themselves nonempty trees.
  - Suppose m is the **maximum** element of lt
  - Then *every* element of rt is greater than m !
    - (Why?)

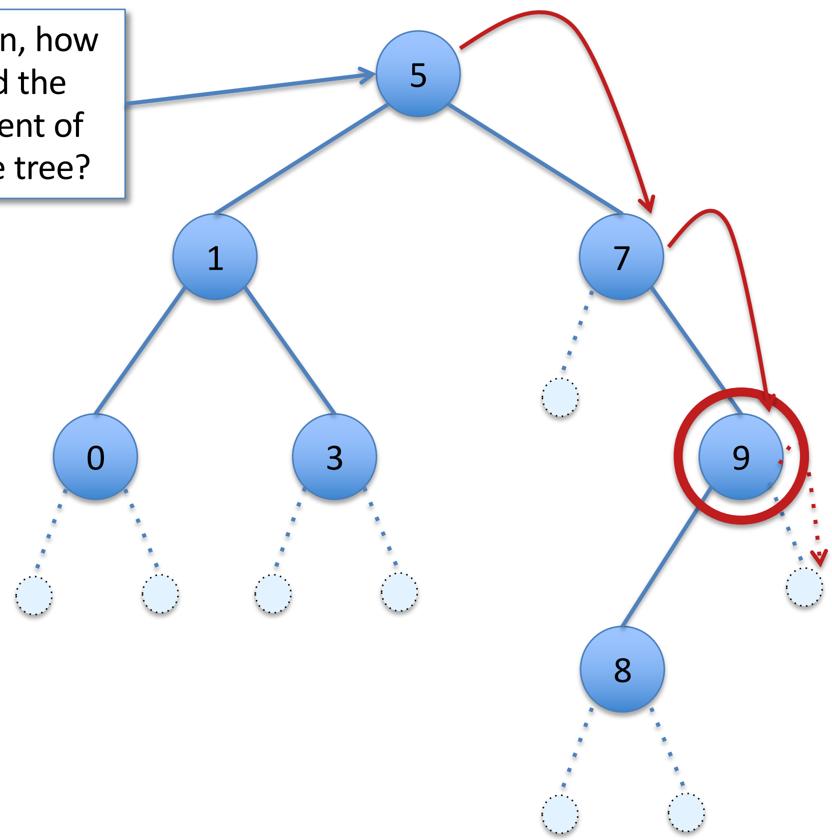
- To *promote* m, we replace the deleted node by:
  Node(delete lt m, m, rt)
  - I.e. we (recursively) delete m from lt and relabel the root node m
  - The resulting tree satisfies the BST invariants

How to Find the Maximum Element?



How to Find the Maximum Element?

Just for fun, how do we find the max element of this whole tree?



Tree Max

```
let rec tree_max (t:tree) : int =
  begin match t with
  | Node(_,x,Empty) -> x
  | Node(_,_,rt) -> tree_max rt
  | _ -> failwith “tree_max called on Empty”
  end
```

- BST invariant guarantees that the maximum-value node is farthest to the right
- Note that `tree_max` is a *partial** function
  – Fails when called with an empty tree
- Fortunately, we never need to call tree_max on an empty tree!
  – This is a consequence of the BST invariants and the case analysis done by the delete function

* Partial, in this context, means “not defined for all inputs”.