Programming Languages and Techniques (CIS120)

Lecture 7

Jan 26, 2012

Binary Search Trees

Announcements

- Homework 2 is due Monday.
  - On-time due date: Jan 30 at 11:59:59pm

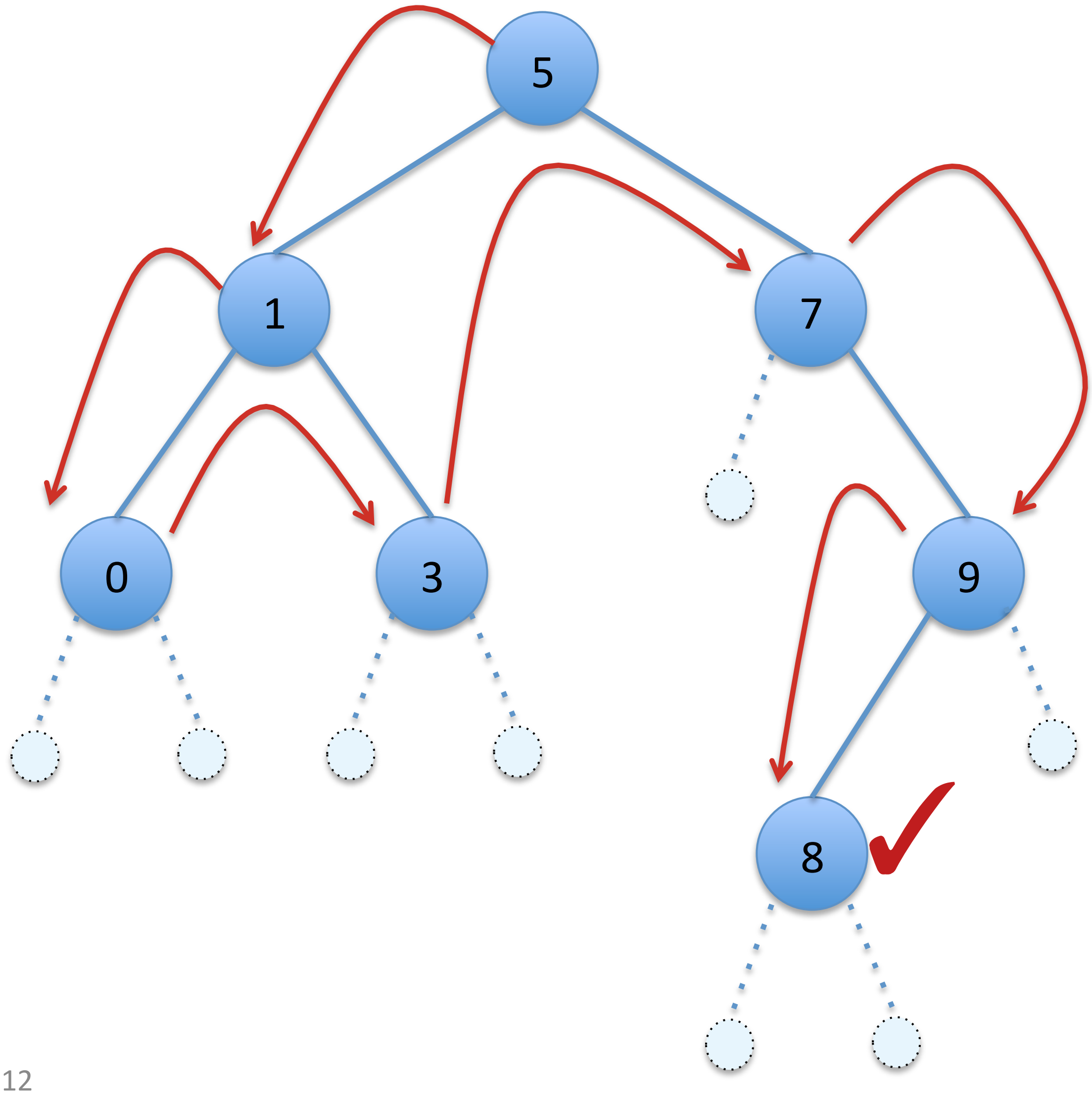
- Updated Lecture Notes available online
  - Covers datatypes and trees

Trees as Containers

- Like lists, trees aggregate data
- Like lists, we can determine whether the data structure *contains* a particular element

- CHALLENGE: can we use the tree structure to make this process faster?

Search during `(contains t 8)`



Searching for Data in a Tree

- Recall the contains function:

```
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
```

- It searches through the tree, looking for n
  - In this case, the search is a *pre-order* traversal of the tree
  - Other traversal strategies would work equally well
- In the worst case, it might search through the entire tree
- Can we do better?

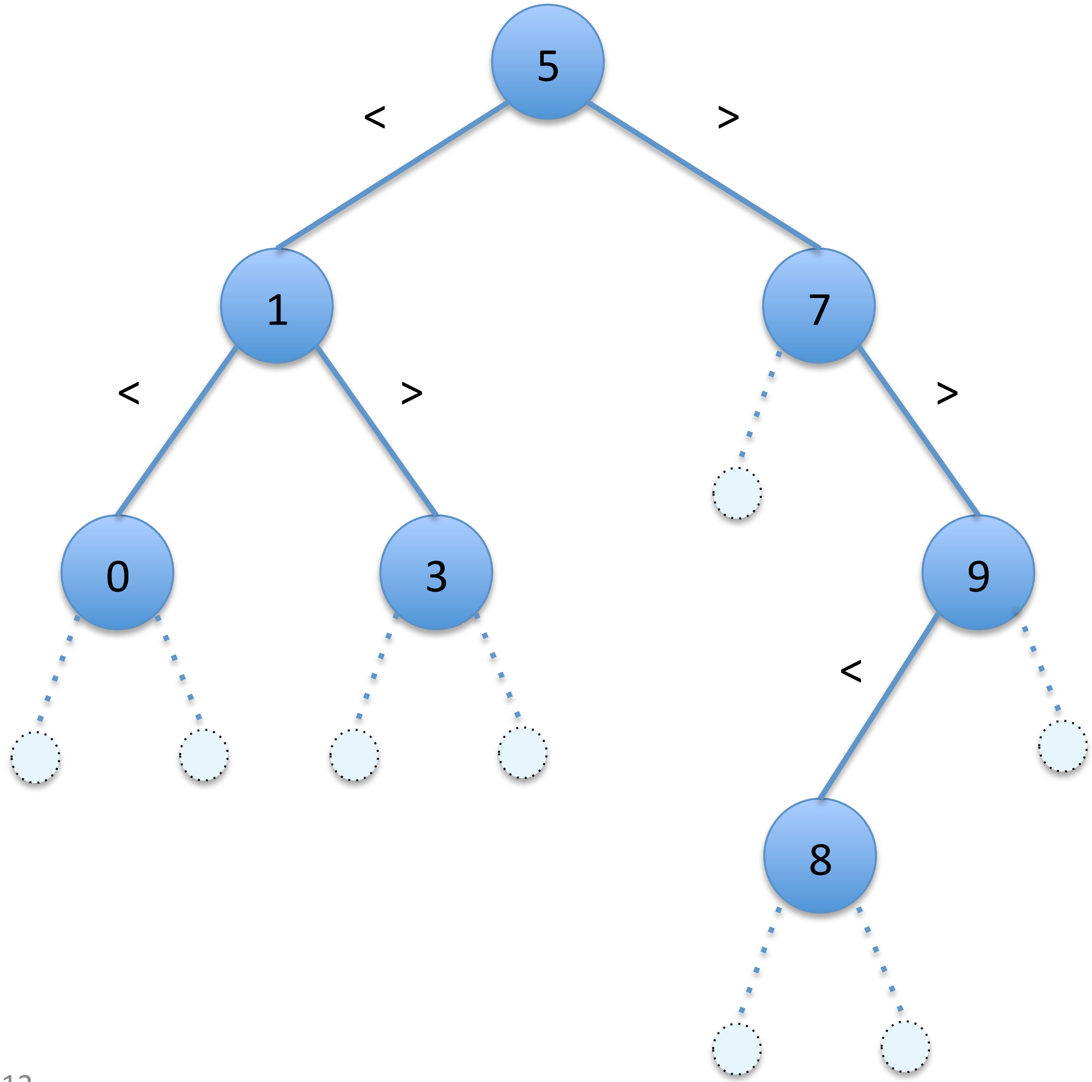
Binary Search Trees (BST)

- Key insight:
  - We can use an *ordering* on the data to cut down the search space
  - This is why telephone books are arranged alphabetically

- A BST is a binary tree with additional *invariants:*

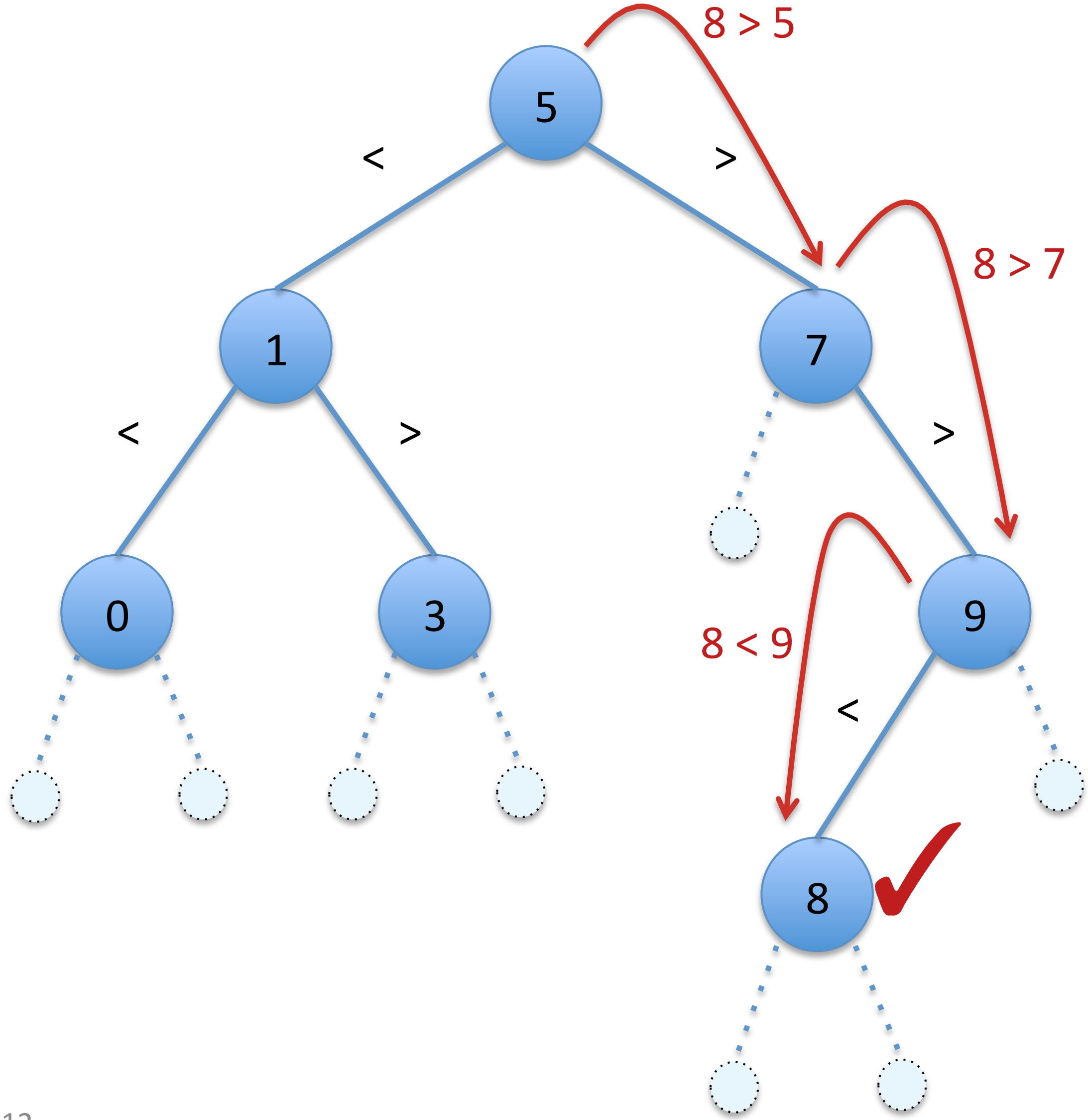
- `Empty` is a BST
- `Node(lt,x,rt)` is a BST if
  - `lt` and `rt` are both BSTs
  - all nodes of `lt` are `<` `x`
  - all nodes of `rt` are `>` `x`

An Example Binary Search Tree



Note that the BST invariants hold for this tree.

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 fail (i.e. return an incorrect answer) if the input is *not* a BST.

How to we construct a BST?

- Option 1:
  - Write a function to check whether an arbitrary tree satisfies the BST invariant.
  - Call the check whenever we need to know about a given tree.
- Option 2:
  - Create functions that *preserve* the BST invariant
  - Starting from some trivial BST (e.g. `Empty`), we can apply such functions to get other BSTs
  - Examples: `insert` and `delete`

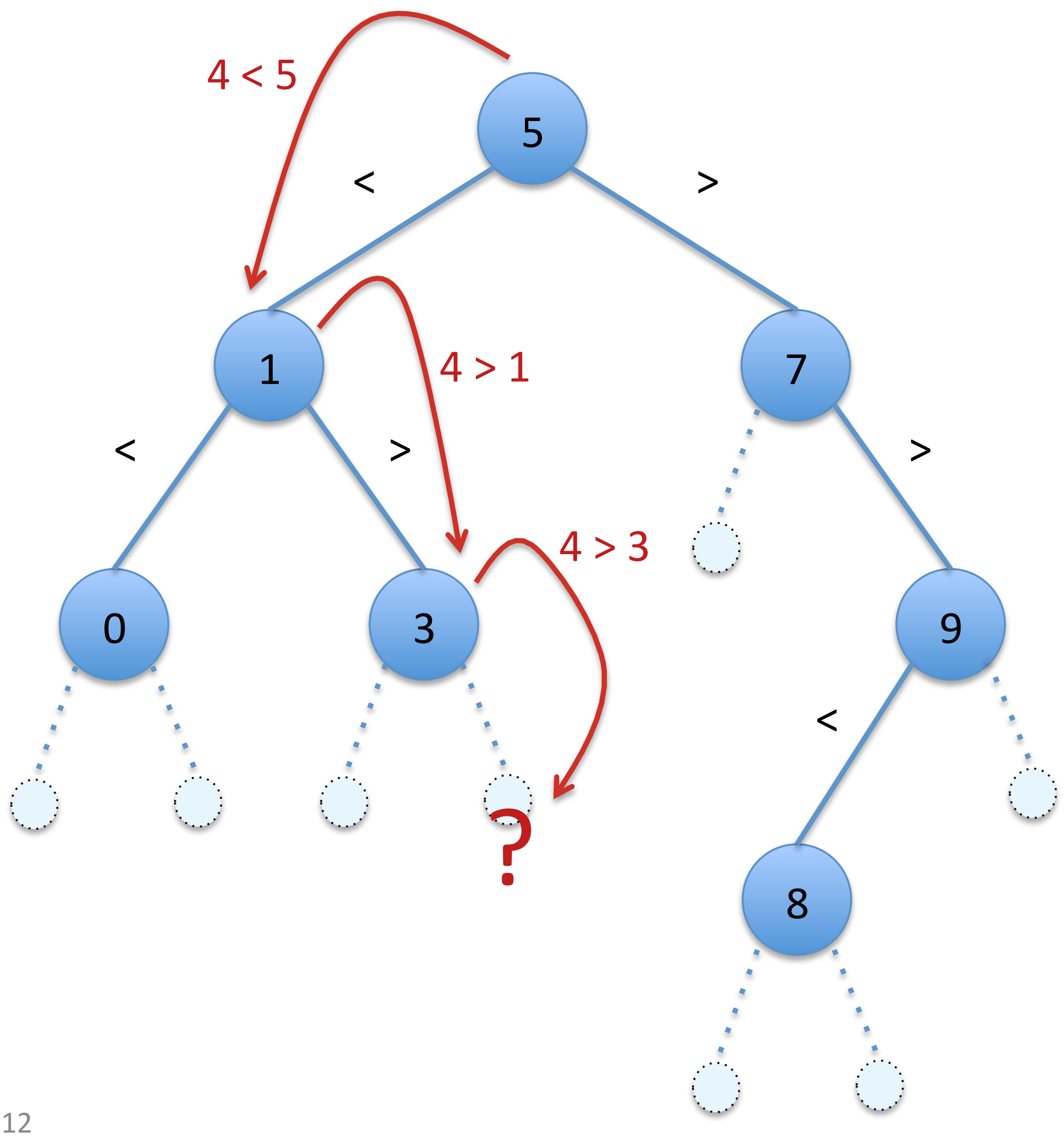
Checking the BST Invariants

```
(* Check whether all nodes of t are < n *)
let rec tree_less (t:tree) (n:int) : bool =
  begin match t with
  | Empty -> true
  | Node(lt,x,rt) ->
      x < n && (tree_less lt n) && (tree_less rt n)
  end
```

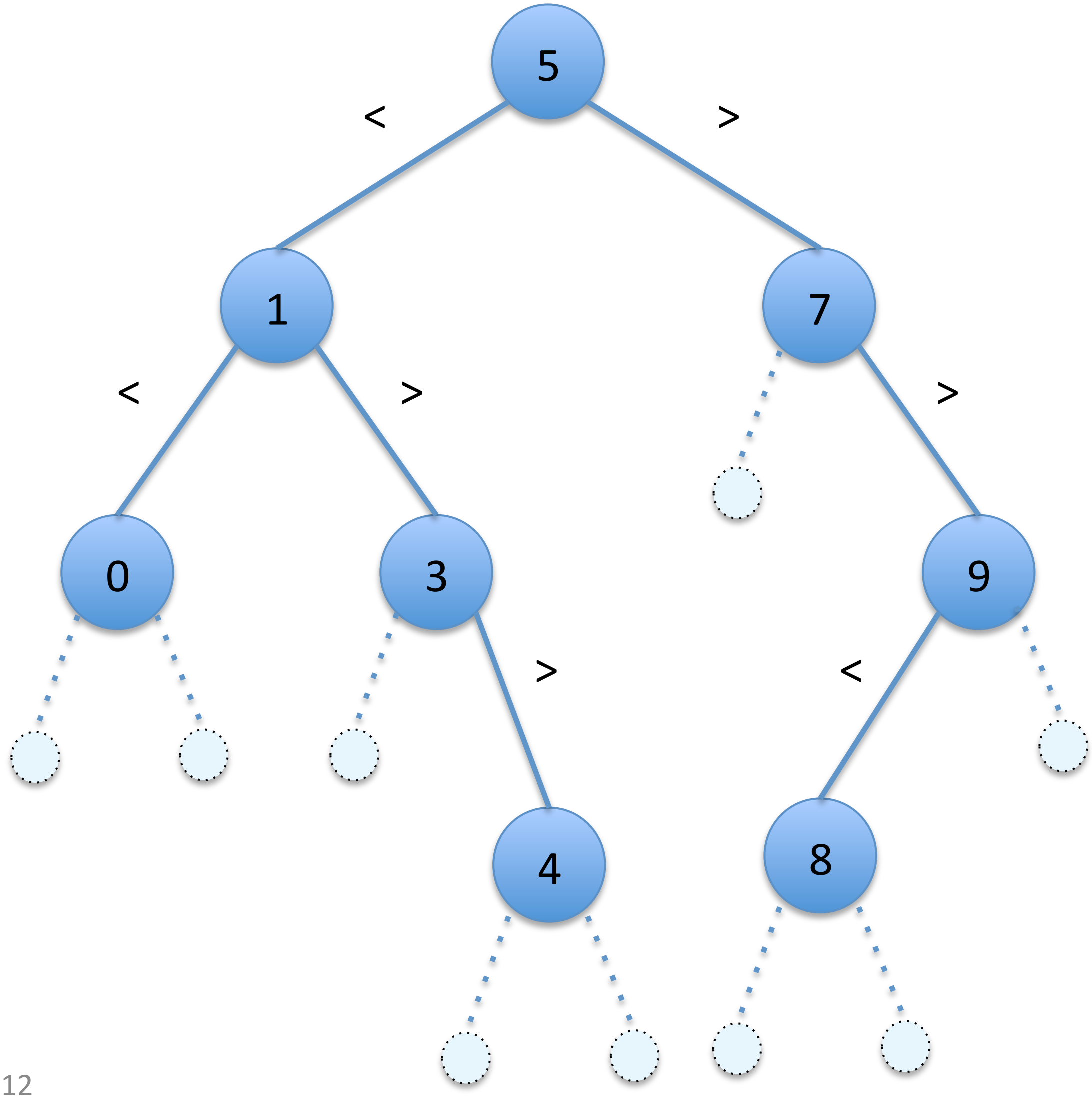
```
(* Determines whether t is a BST *)
let rec is_bst (t:tree) : bool =
  begin match t with
  | Empty -> true
  | Node(lt,x,rt) ->
      is_bst lt && is_bst rt &&
      (tree_less lt x) && (tree_gtr rt x)
  end
```

*Definition of tree_gtr omitted (it's similar to tree_less)

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



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

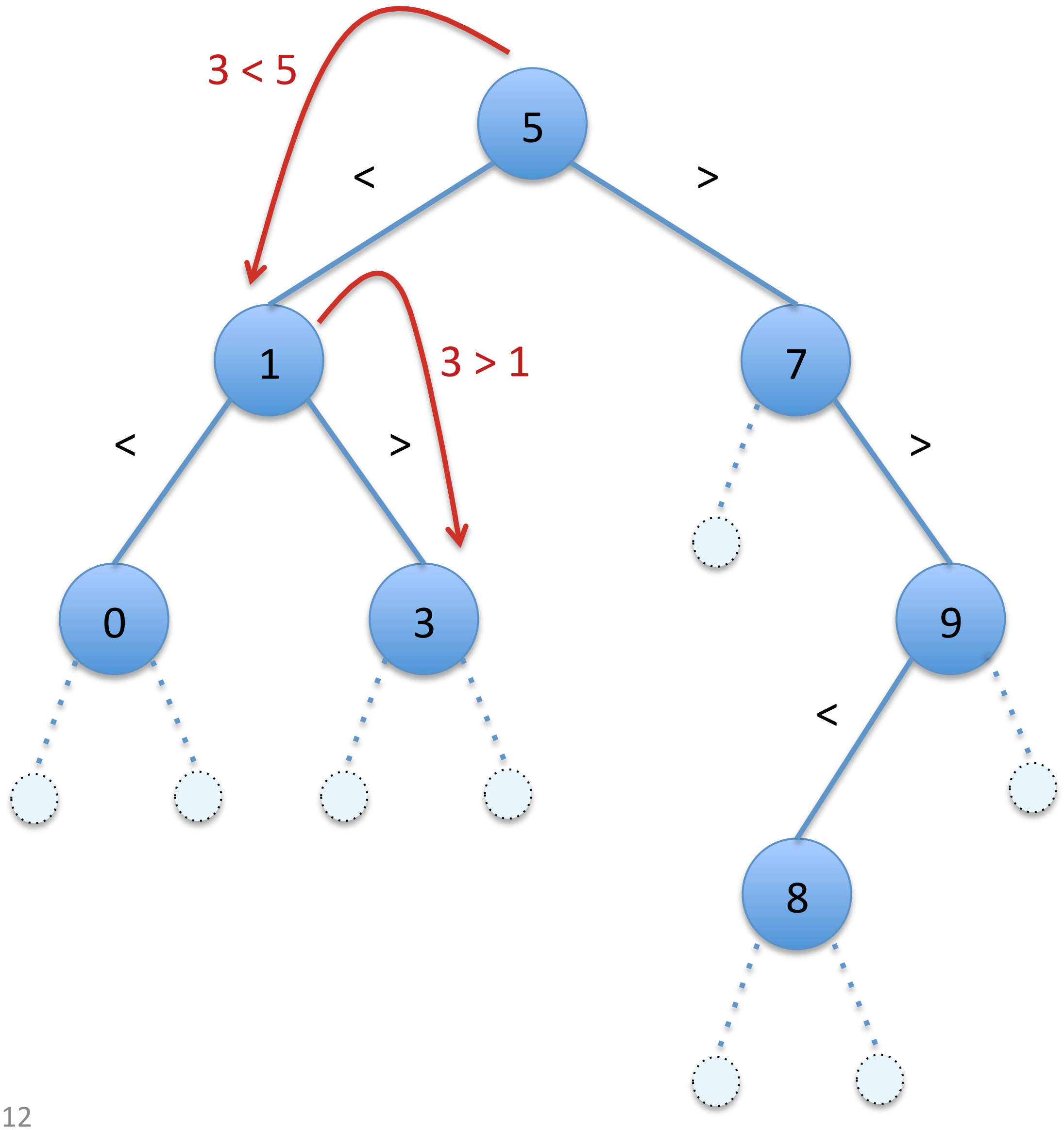


Inserting Into a BST

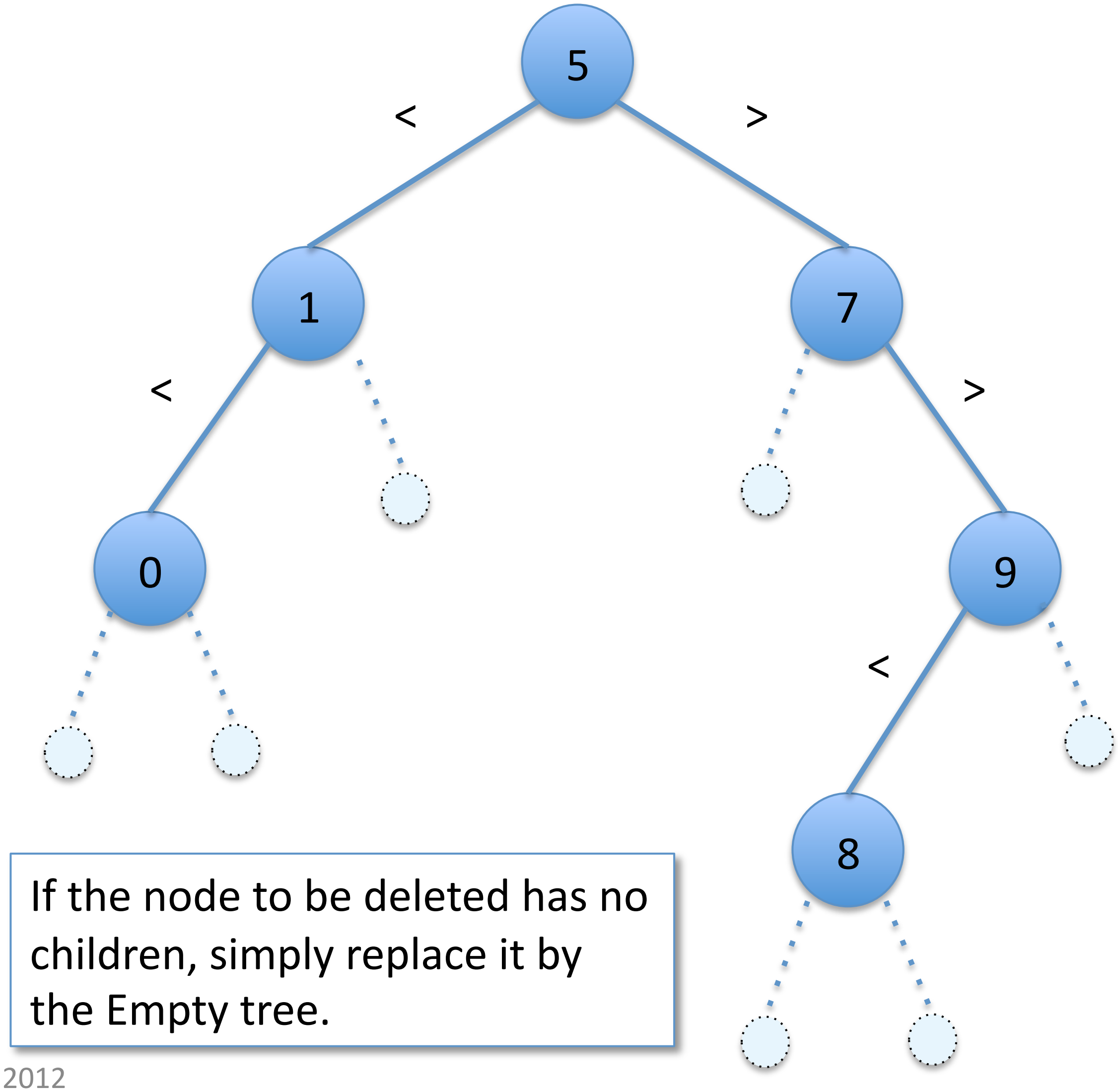
```
(* Inserts n into the BST t *)
let rec insert (t:tree) (n:int) : tree =
  begin match t with
  | Empty -> Node(Empty,n,Empty)
  | Node(lt,x,rt) ->
       if x = n then t
       else if n < x then Node(insert lt n, x, rt)
       else Node(lt, x, insert rt n)
  end
```

- Note the similarity to searching the tree.
- Assuming that t is a BST, the result is also a BST. Why?

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

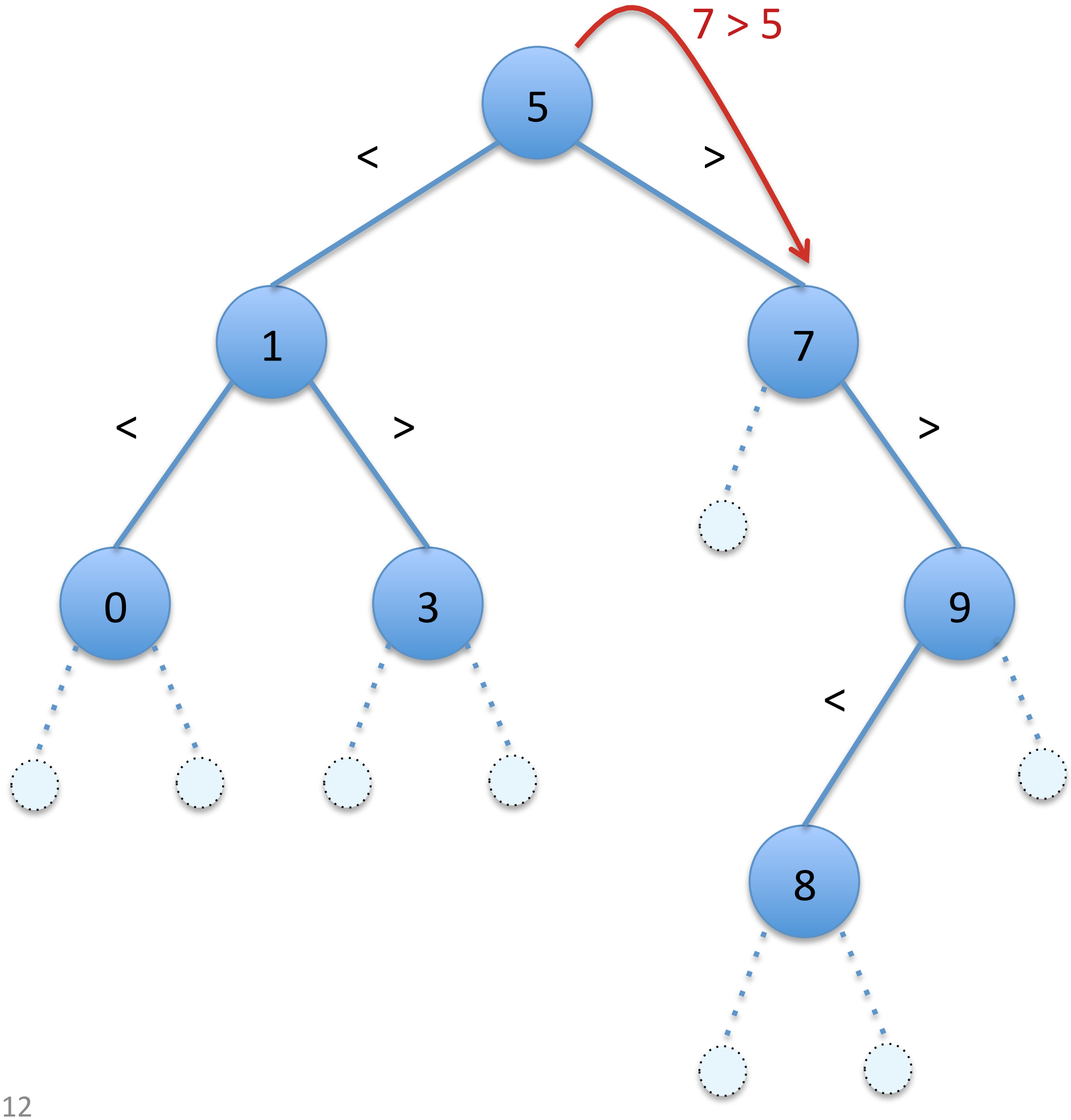


Deletion – No Children: `(delete t 3)`



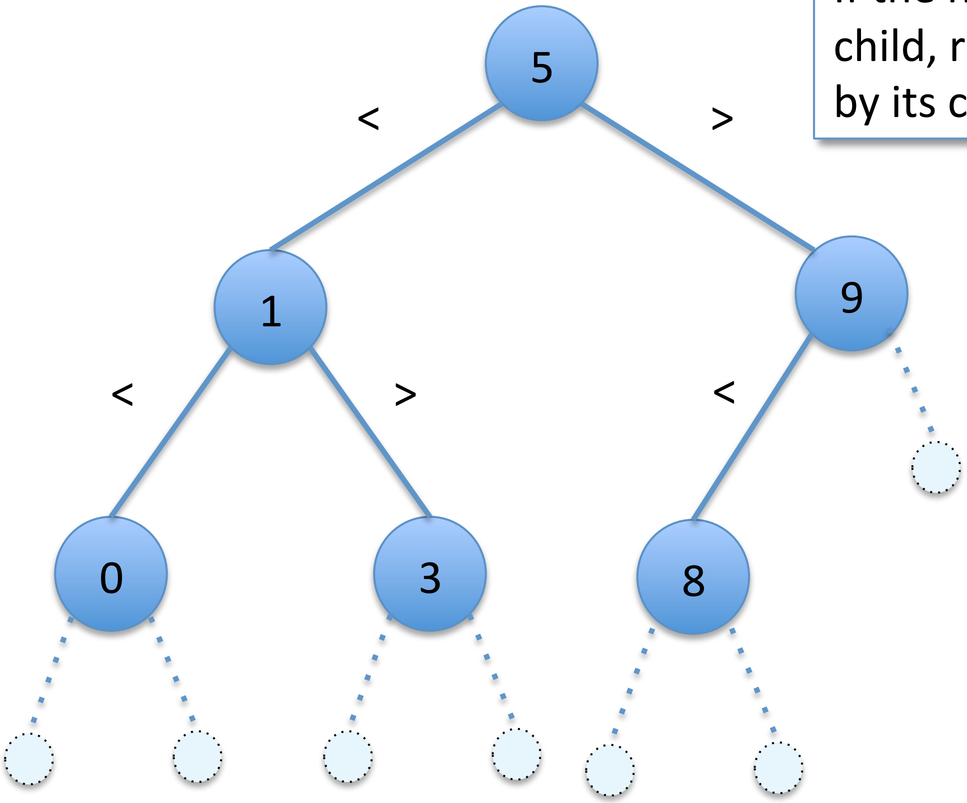
If the node to be deleted has no children, simply replace it by the Empty tree.

Deletion – One Child: `(delete t 7)`

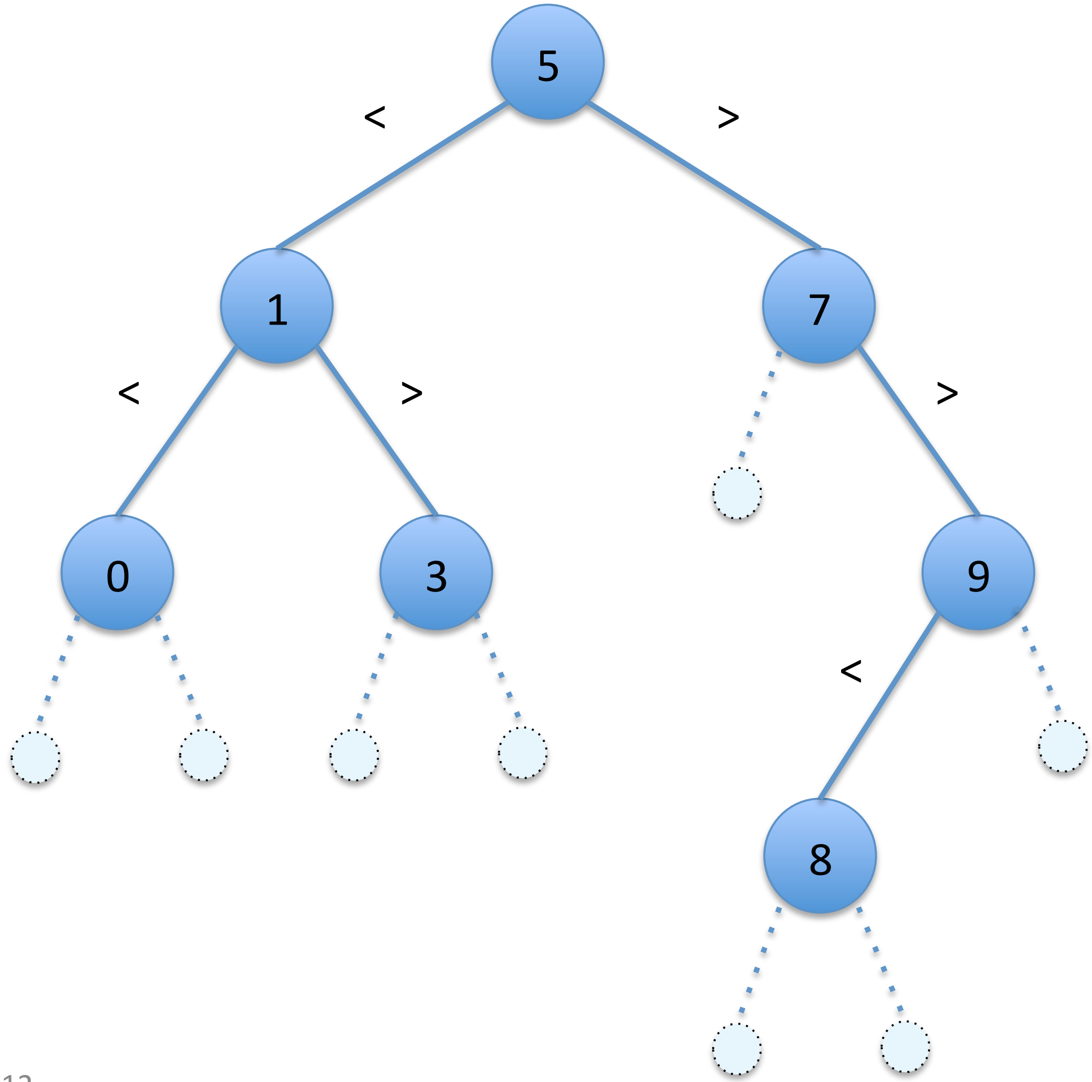


Deletion – One Child: `(delete t 7)`

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

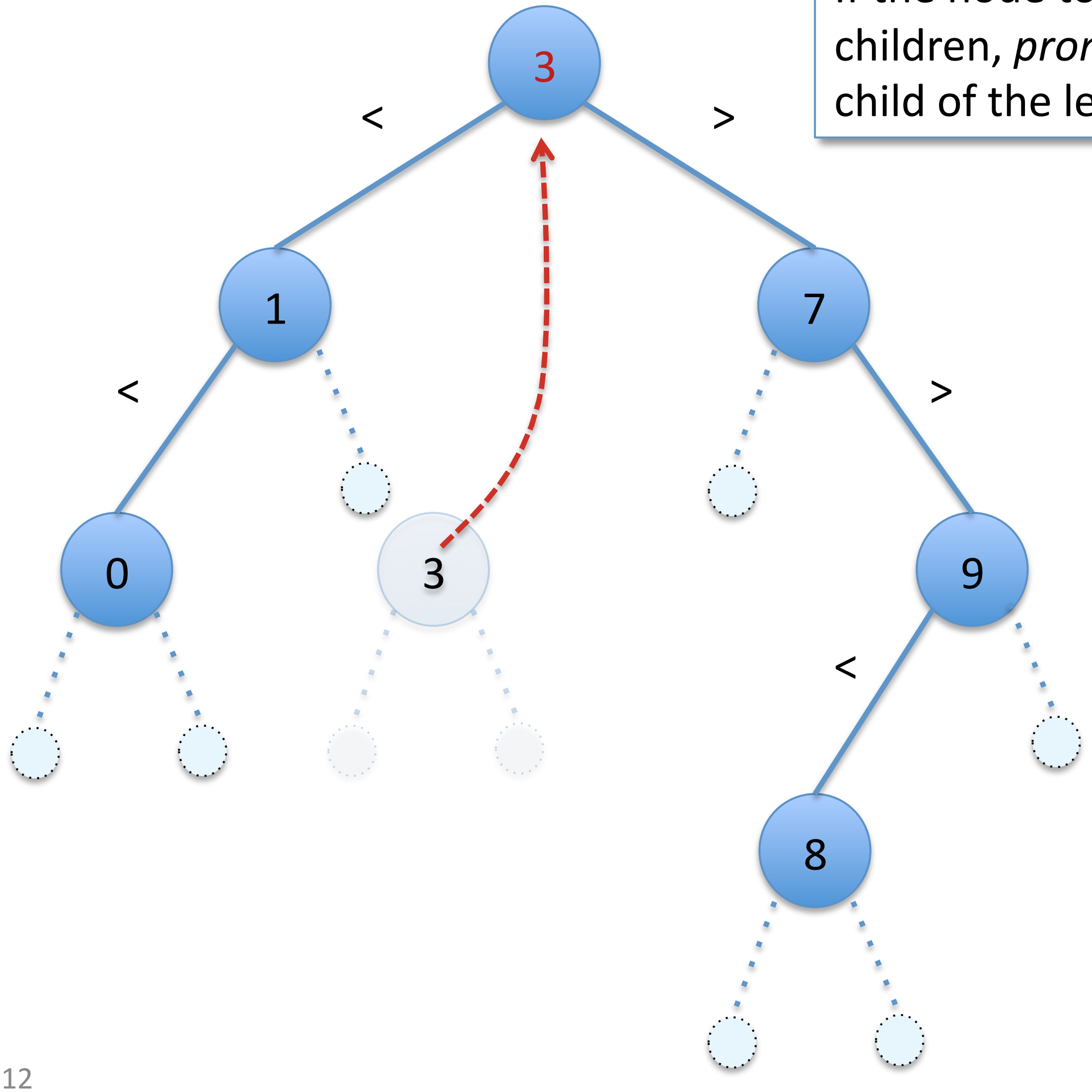


Deletion – Two Children: `(delete t 5)`



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

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



Subtleties of the Two-Child Case

- Suppose Node(lt,x,rt) is to be deleted and lt and rt are both themselves nonempty trees.
- Then:
  - There exists a maximum element, m, of lt (why?)
  - m is smaller than every element of rt (why?)
- To promote m we replace the deleted node by:
  Node(delete lt m, m, rt)
  - i.e. we recursively delete m from lt
  - Note the resulting tree satisfies the BST invariants
- Question: will this always work?

`tree_max`: A *partial* function

```
let rec tree_max (t:tree) : int =
  begin match t with
  | Empty -> ????
  | Node(lt,x,rt) -> …
  end
```

- Problem: `tree_max` isn't defined for *all* binary trees.
  - In particular, it isn't defined for the *empty* binary tree
  - Technically, `tree_max` is a *partial function*
- What to do?

Solutions to Partiality: Option 1

- Return a *default or error value*
  - e.g. define `tree_max Empty` to be `-1`
  - Error codes used often in C programs; null used often in Java
- But…
  - What if -1 (or whatever default you choose) really *is* the maximum value?
  - Can lead to many bugs if the default or error value isn't handled properly by the callers.

- Defaults should be avoided if possible

Solutions to Partiality: Option 2*

- Abort the program:
  - In OCaml: `failwith “an error message”`

- Whenever it is called, `failwith` aborts the program and reports the error message it is given.
- This solution to partiality is appropriate whenever you *know* that a certain case is impossible.
  - Often happens when there is an invariant on a datastructure
  - The compiler isn’t smart enough to figure out that the case is impossible…
  - `failwith` is also useful to “stub out” unimplemented parts of your program.

*There are a few other ways to deal with partiality (*using datatypes* or *exceptions*) that we’ll see later in the course

BST Invariants and `tree_max`

- For delete, 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.
- So: we can write tree_max *assuming* that the input tree is a nonempty BST:

```
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
```

- Note: BST invariant is used because it guarantees that the maximum valued node is farthest to the right

Deleting From a BST

```
(* returns a binary search tree that has the same set of
   nodes as t except with n removed (if it's there) *)
let rec delete (t:tree) (n:int) : tree =
  begin match t with
    | Empty -> Empty
    | Node(lt,x,rt) ->
     if x = n then
       begin match (lt,rt) with
         | (Empty, Empty) -> Empty
         | (Node _, Empty) -> lt
         | (Empty, Node _) -> rt
         |  _ -> let m = tree_max lt in
        Node(delete lt m, m, rt)
       end
     else if n < x then Node(delete lt n, x, rt)
     else Node(lt, x, delete rt n)
  end
```