Programming Languages and Techniques (CIS120)

Lecture 7

Feb 3, 2014

BSTs

Generic Types

Announcements

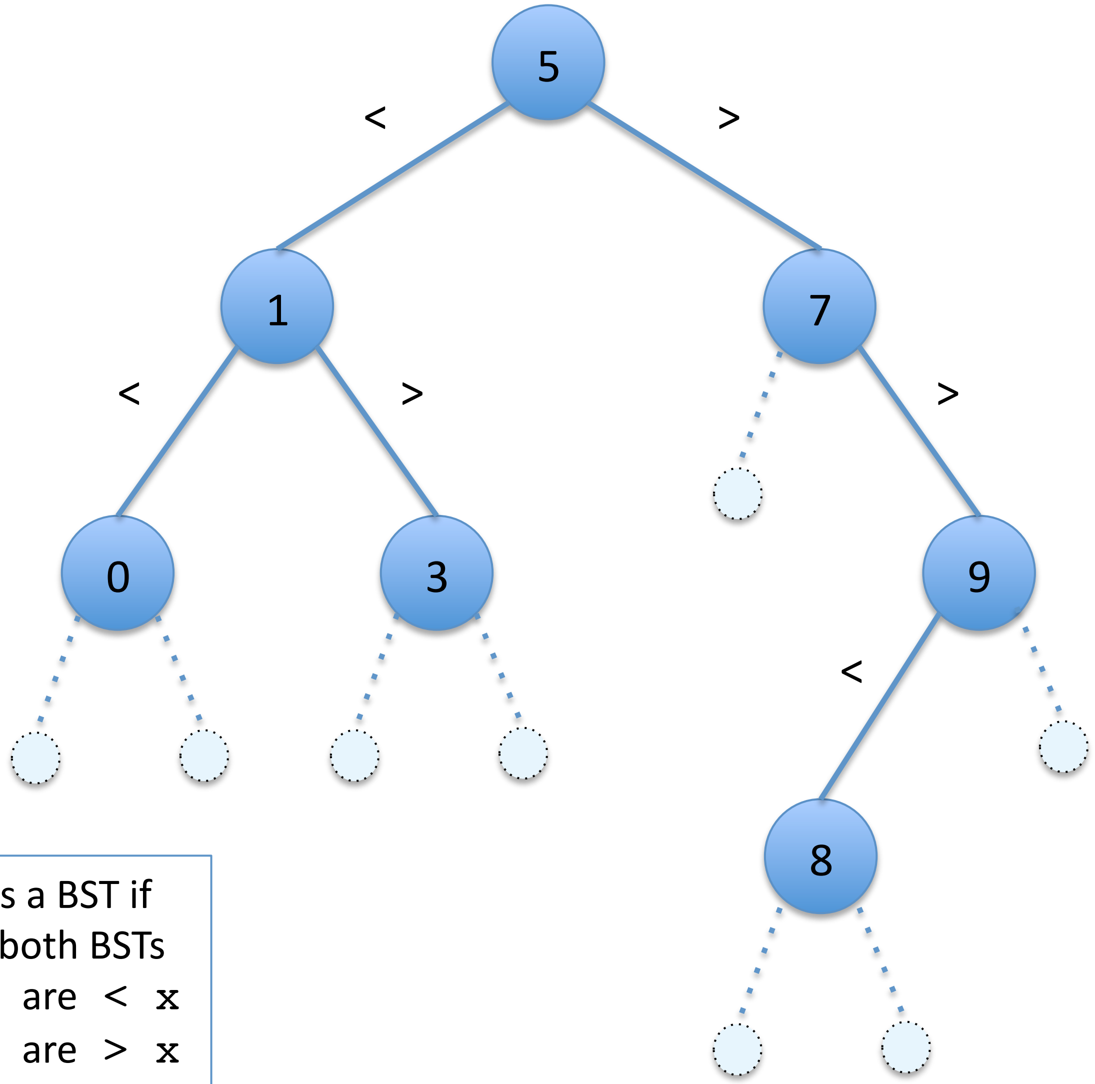
- Homework 2 due Tuesday

- Read Chapters 7 & 8 in the lecture notes

Trees as containers

Big idea: find things faster by searching less

A 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

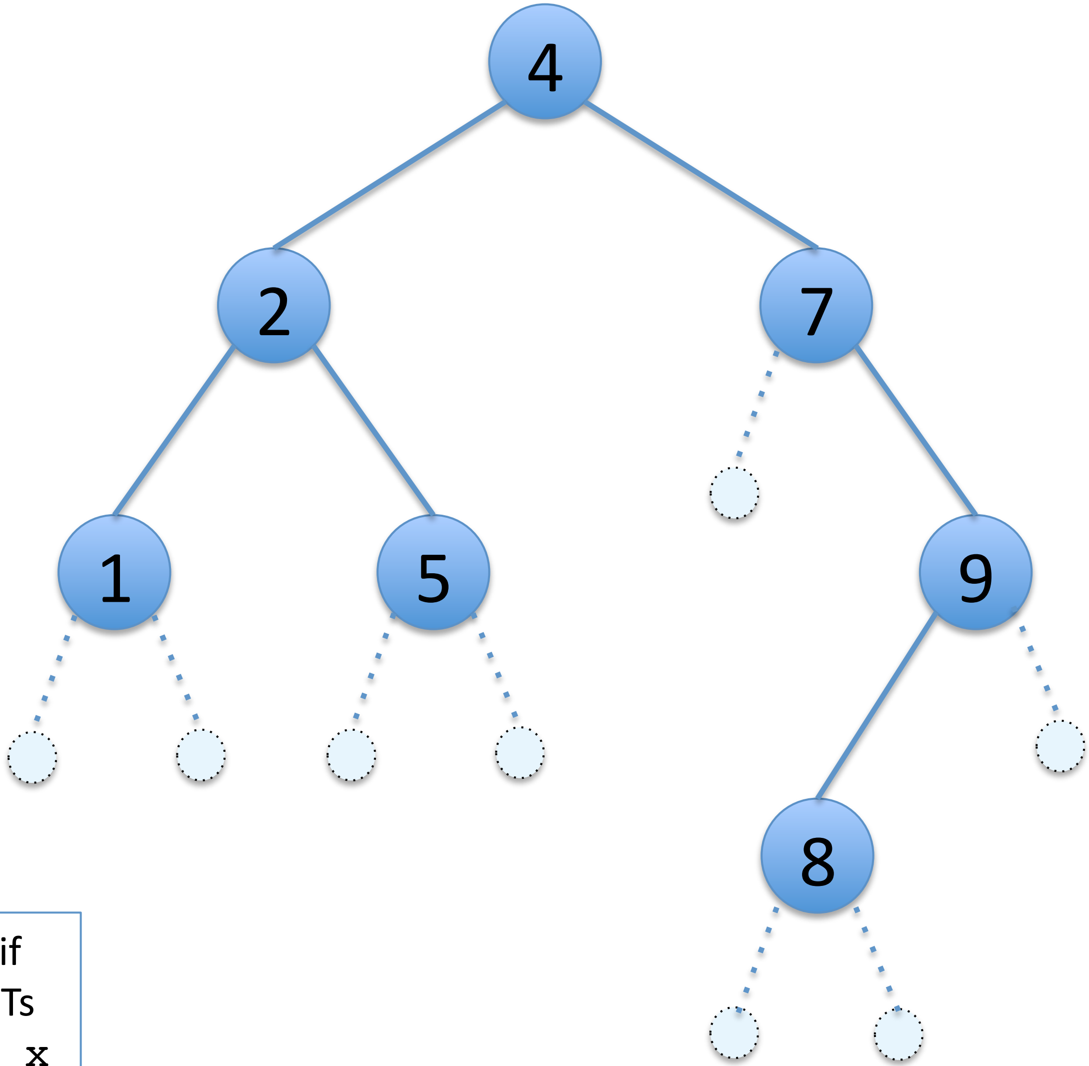
Searching a BST

```
(* 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!

Is this a BST??

1. yes
2. no

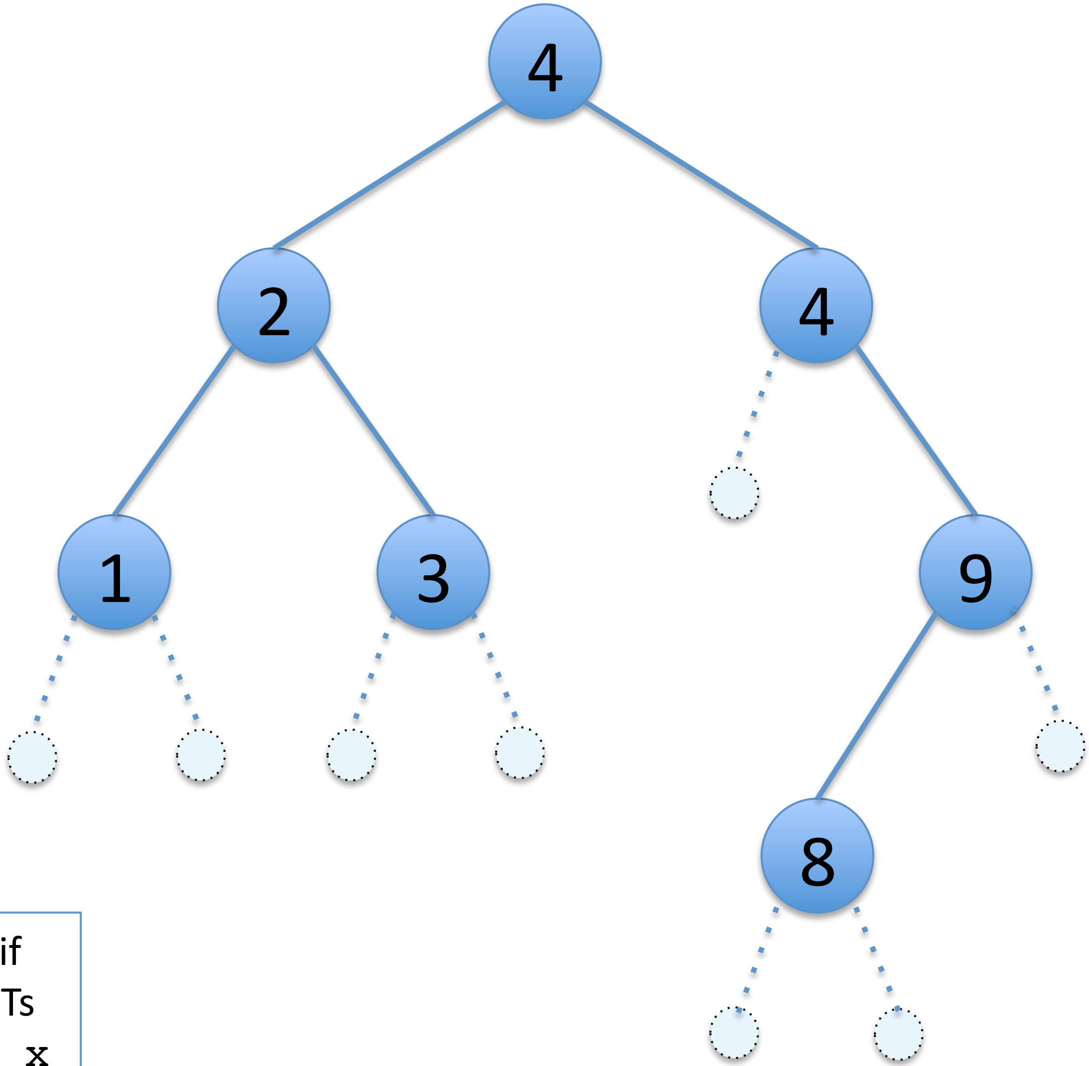


- `Node(lt,x,rt)` is a BST if
  - `lt` and `rt` are both BSTs
  - all nodes of `lt` are `<` `x`
  - all nodes of `rt` are `>` `x`
- `Empty` is a BST

Answer: no, 5 to the left of 4

Is this a BST??

1. yes
2. no

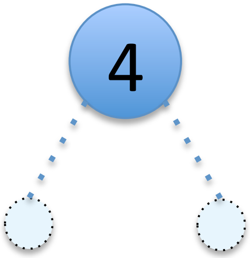


- `Node(lt,x,rt)` is a BST if
  - `lt` and `rt` are both BSTs
  - all nodes of `lt` are `<` `x`
  - all nodes of `rt` are `>` `x`
- `Empty` is a BST

Answer: no, two 4s

Is this a BST??

1. yes
2. no

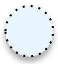


- `Node(lt,x,rt)` is a BST if
  - `lt` and `rt` are both BSTs
  - all nodes of `lt` are `<` `x`
  - all nodes of `rt` are `>` `x`
- `Empty` is a BST

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

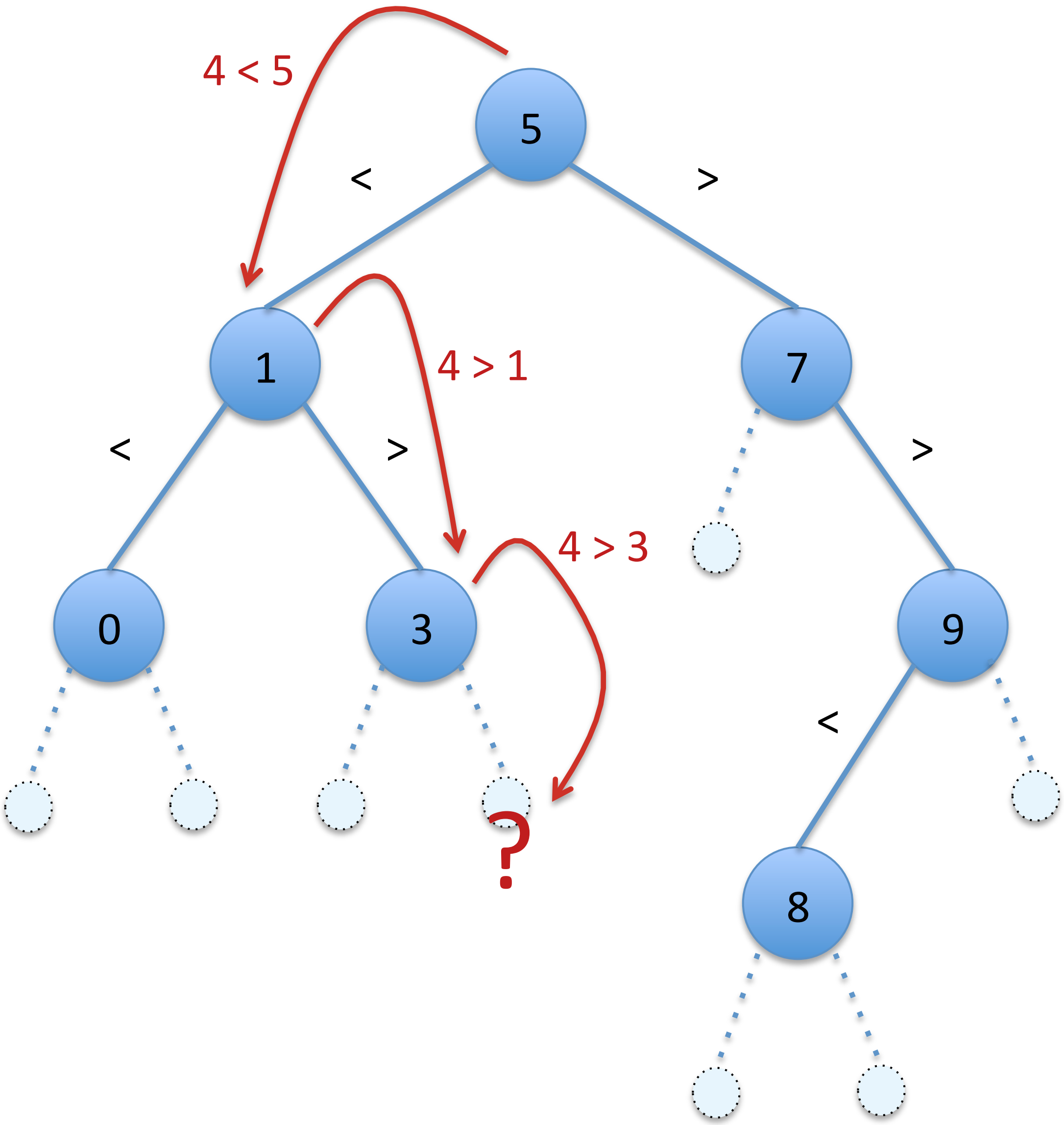
Answer: yes

How do we construct a BST?

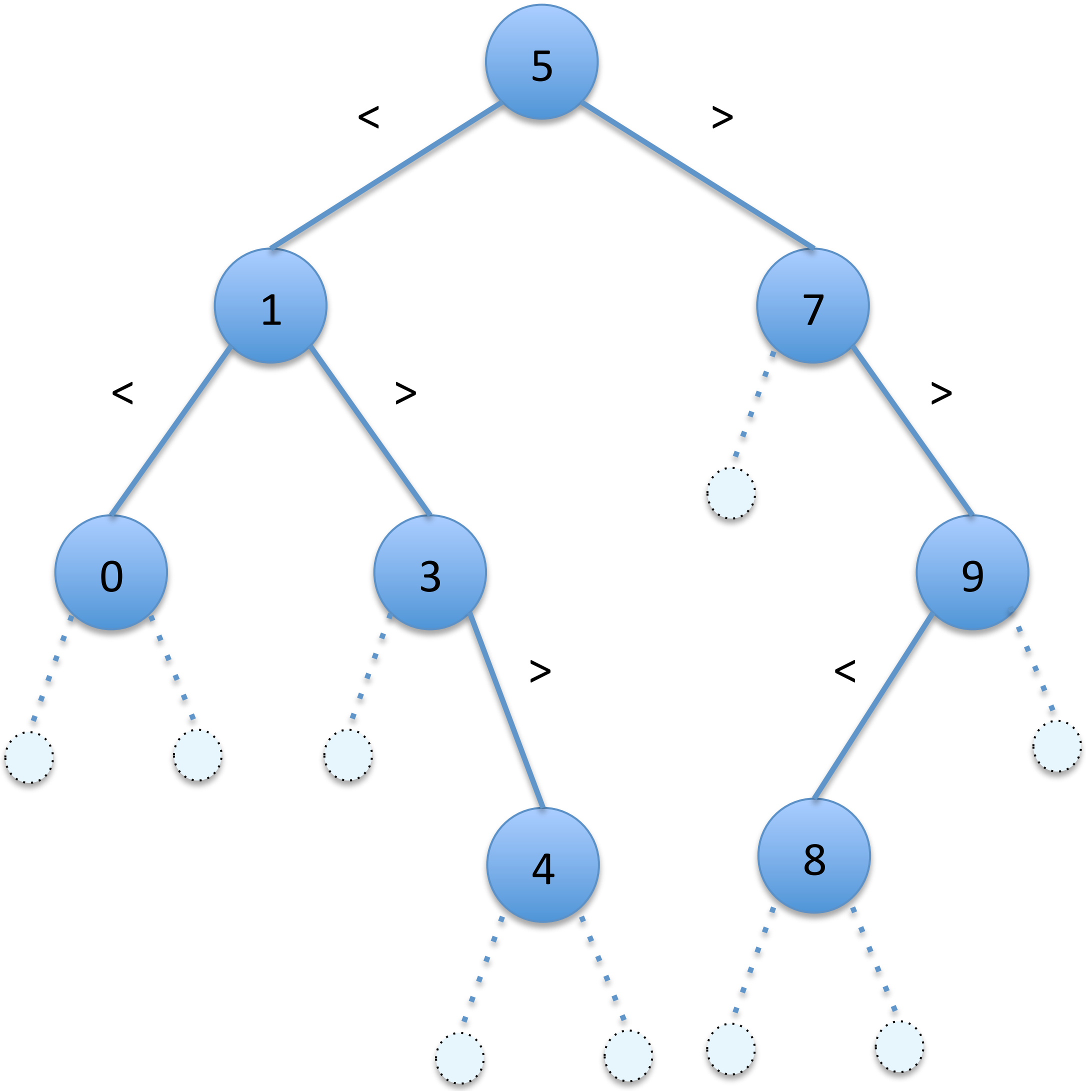
- Option 1:
  - Build a tree
  - Check that the BST invariants hold

- 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!

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



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

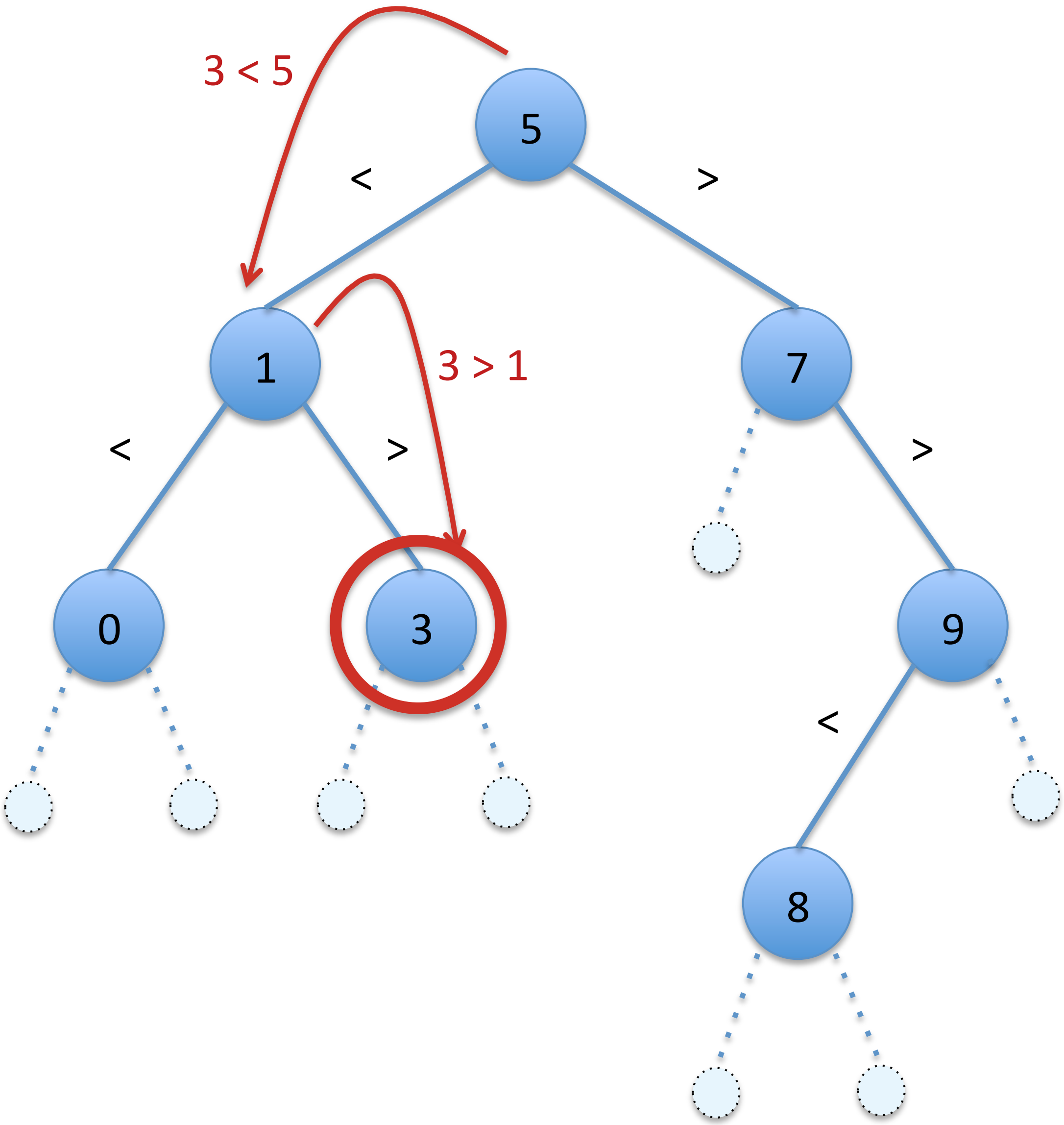


Inserting Into a BST

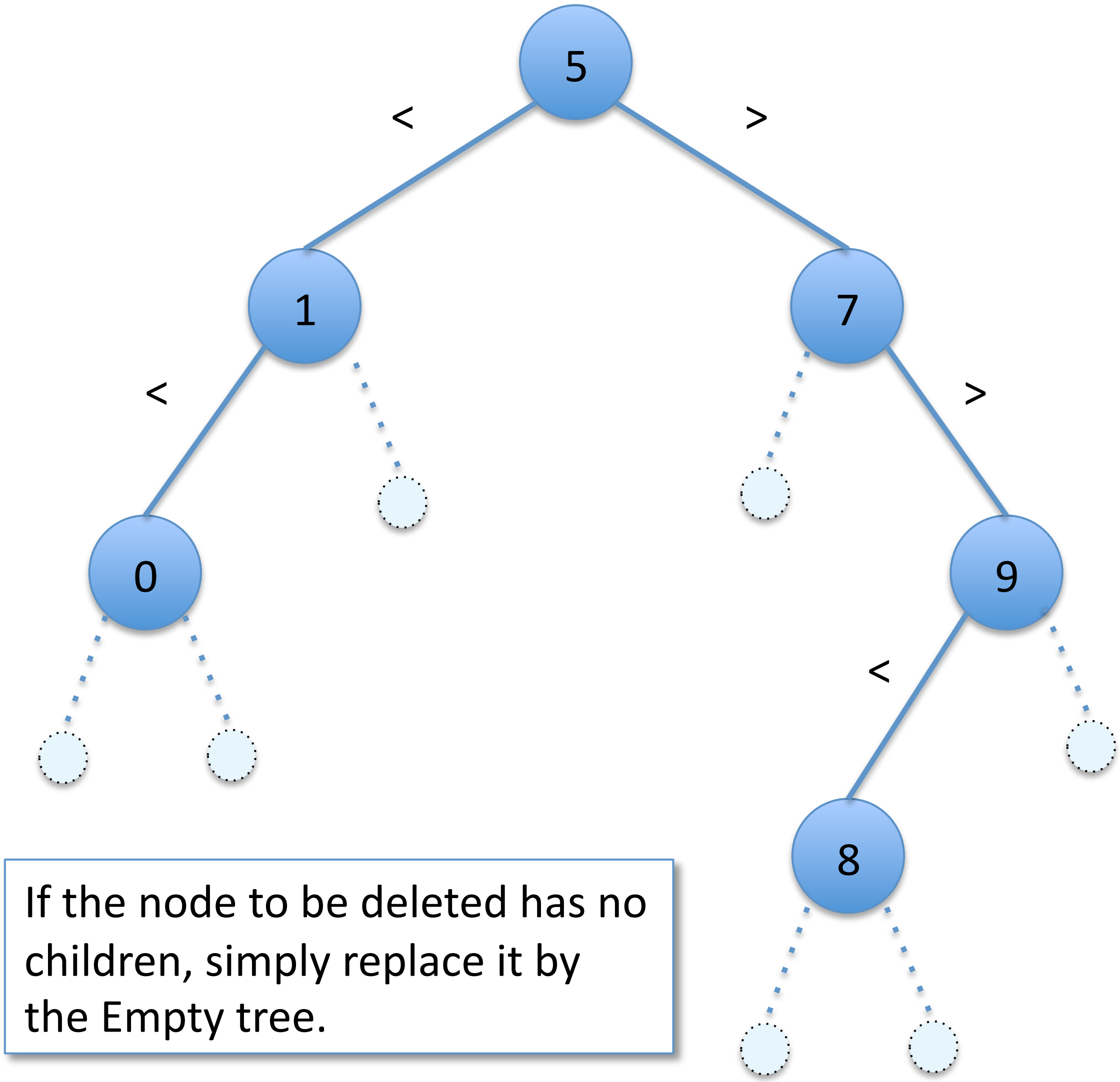
```
(* Insert n into the BST t *)
let rec insert (t:tree) (n:int) : tree =
  begin match t with
  | Empty -> Node(Empty,n,Empty)
  | Node(lt,x,rt) ->
       if x = n then t
       else if n < x then Node(insert lt n, x, rt)
       else Node(lt, x, insert rt n)
  end
```

- 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?)

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

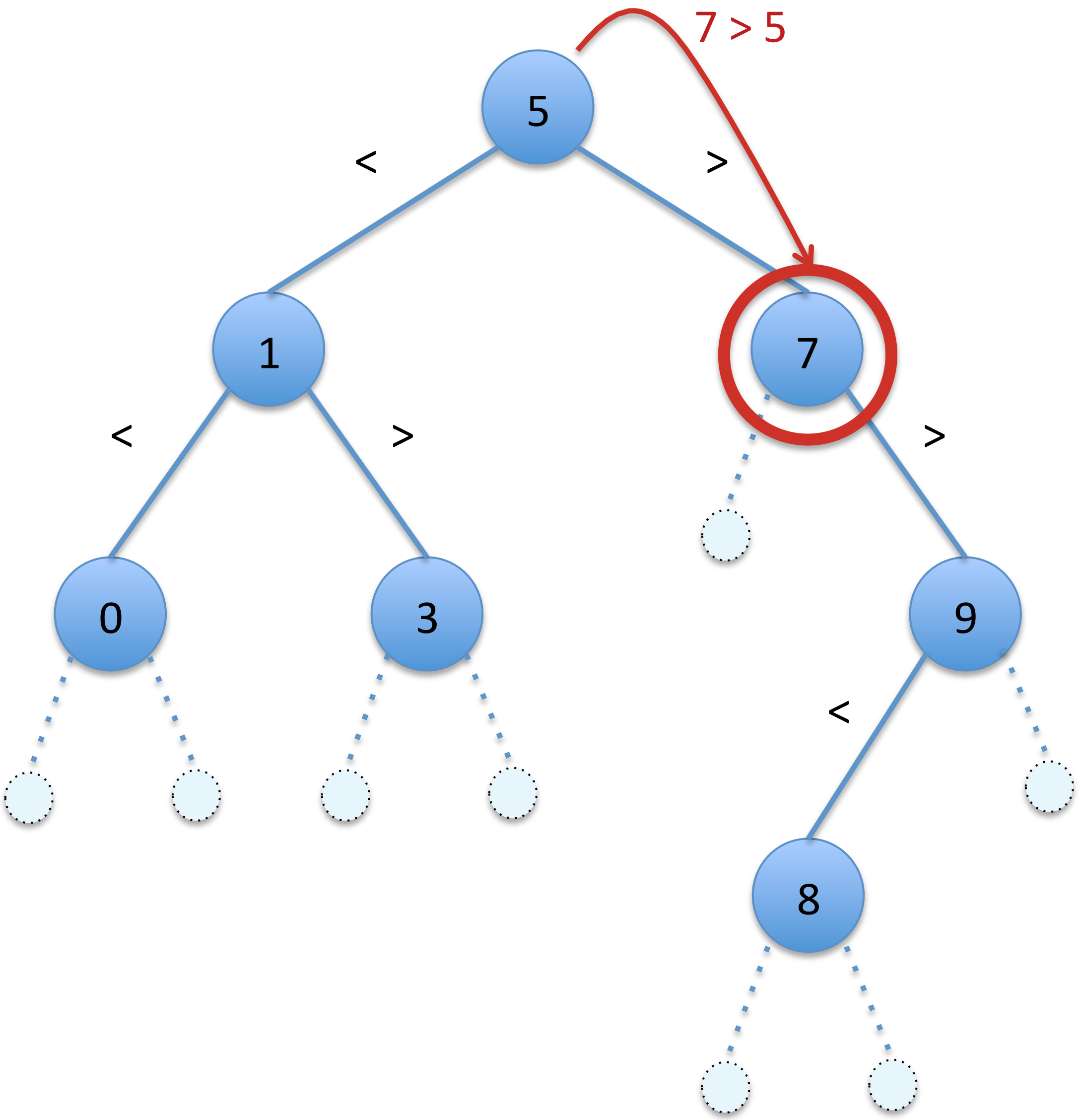


Deletion – No Children: `(delete t 3)`



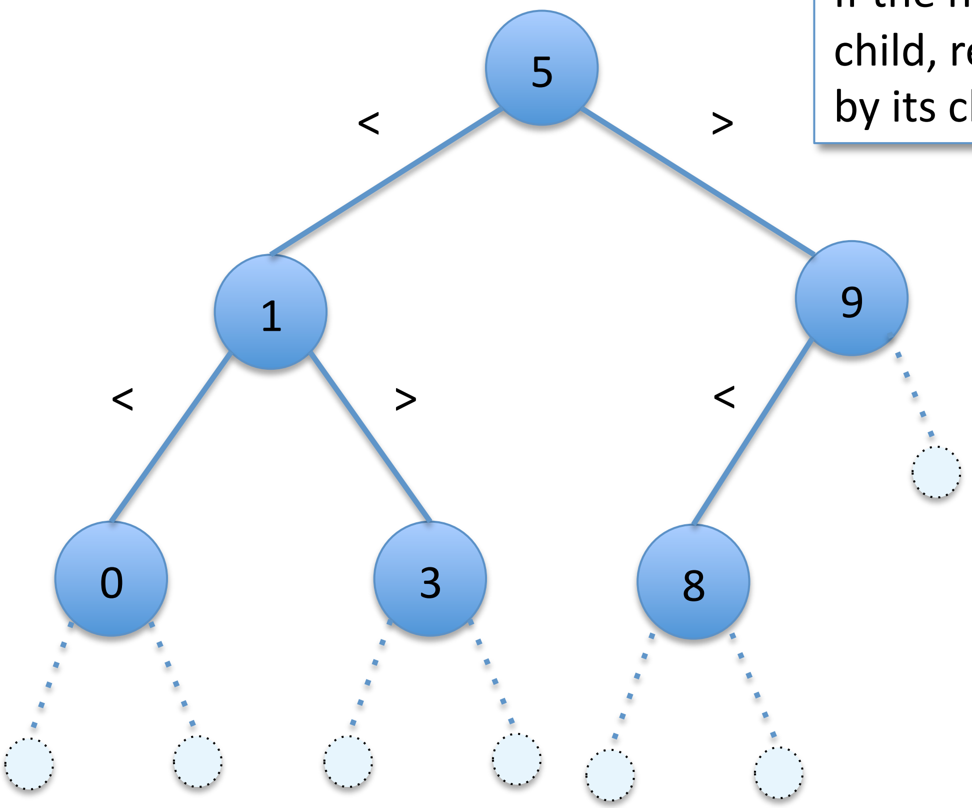
If the node to be deleted has no children, simply replace it by the Empty tree.

Deletion – One Child: `(delete t 7)`

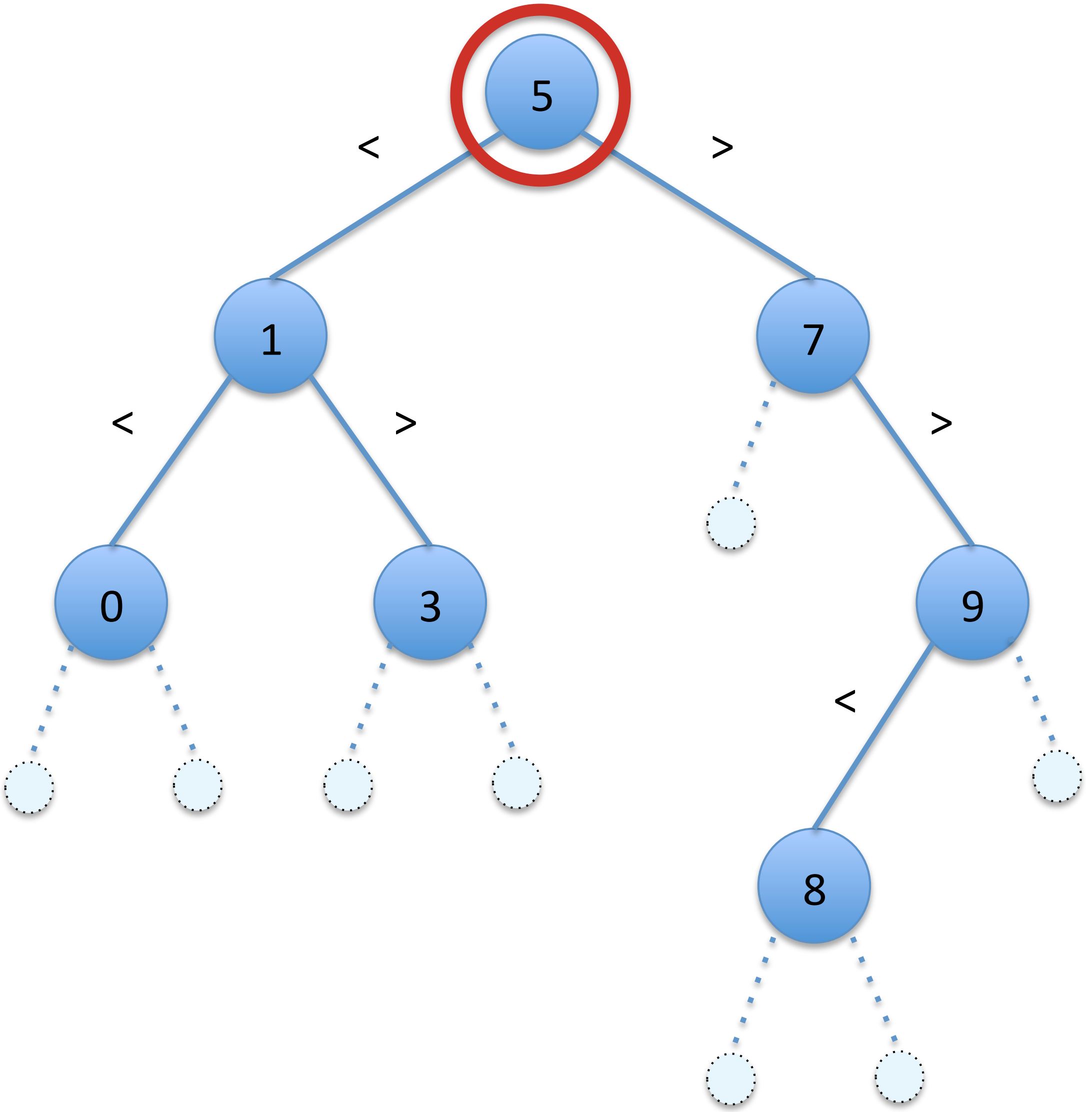


Deletion – One Child: `(delete t 7)`

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

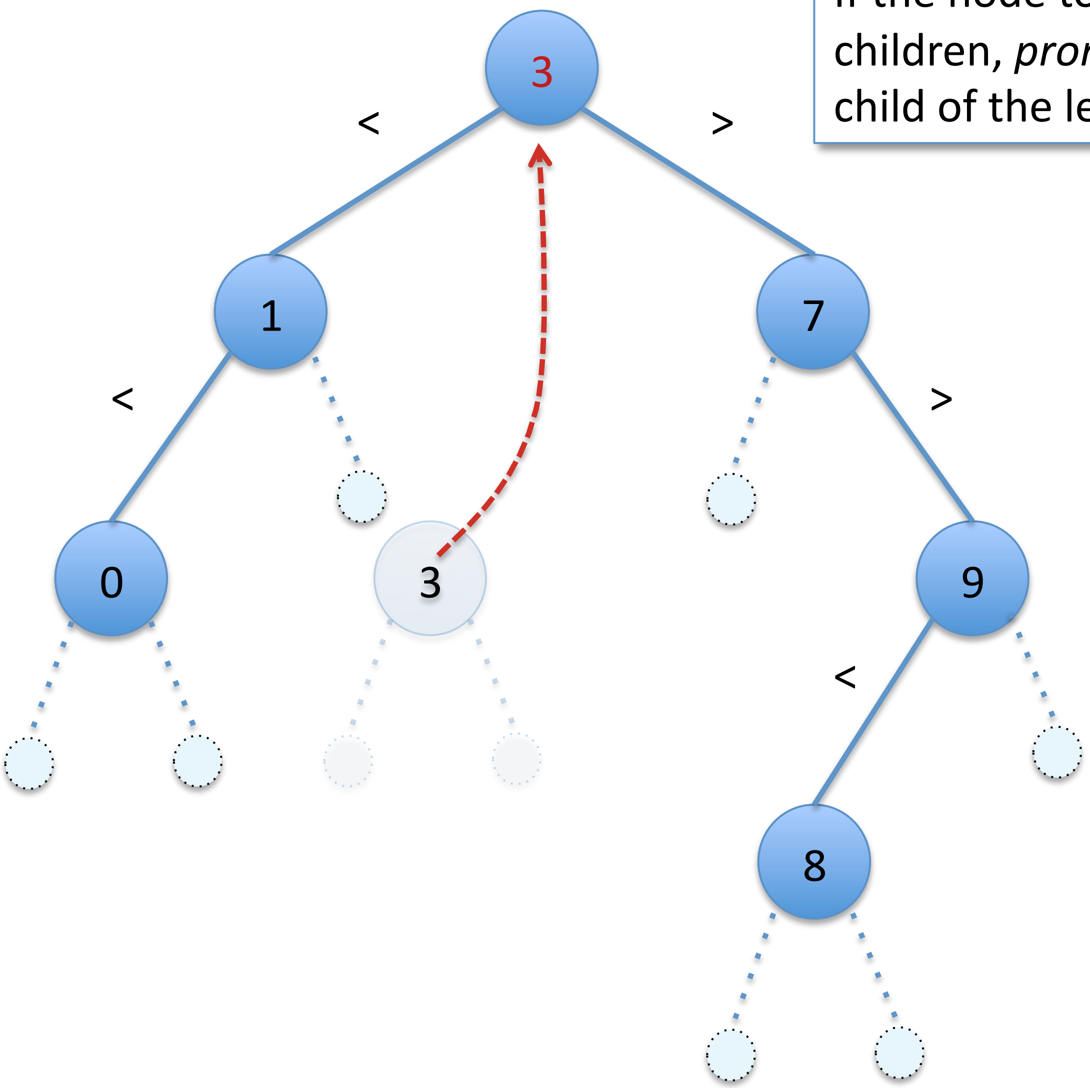


Deletion – Two Children: `(delete t 5)`



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

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:
  1. There exists a maximum element, m, of lt (Why?)
  2. Every element of rt is greater than m (Why?)

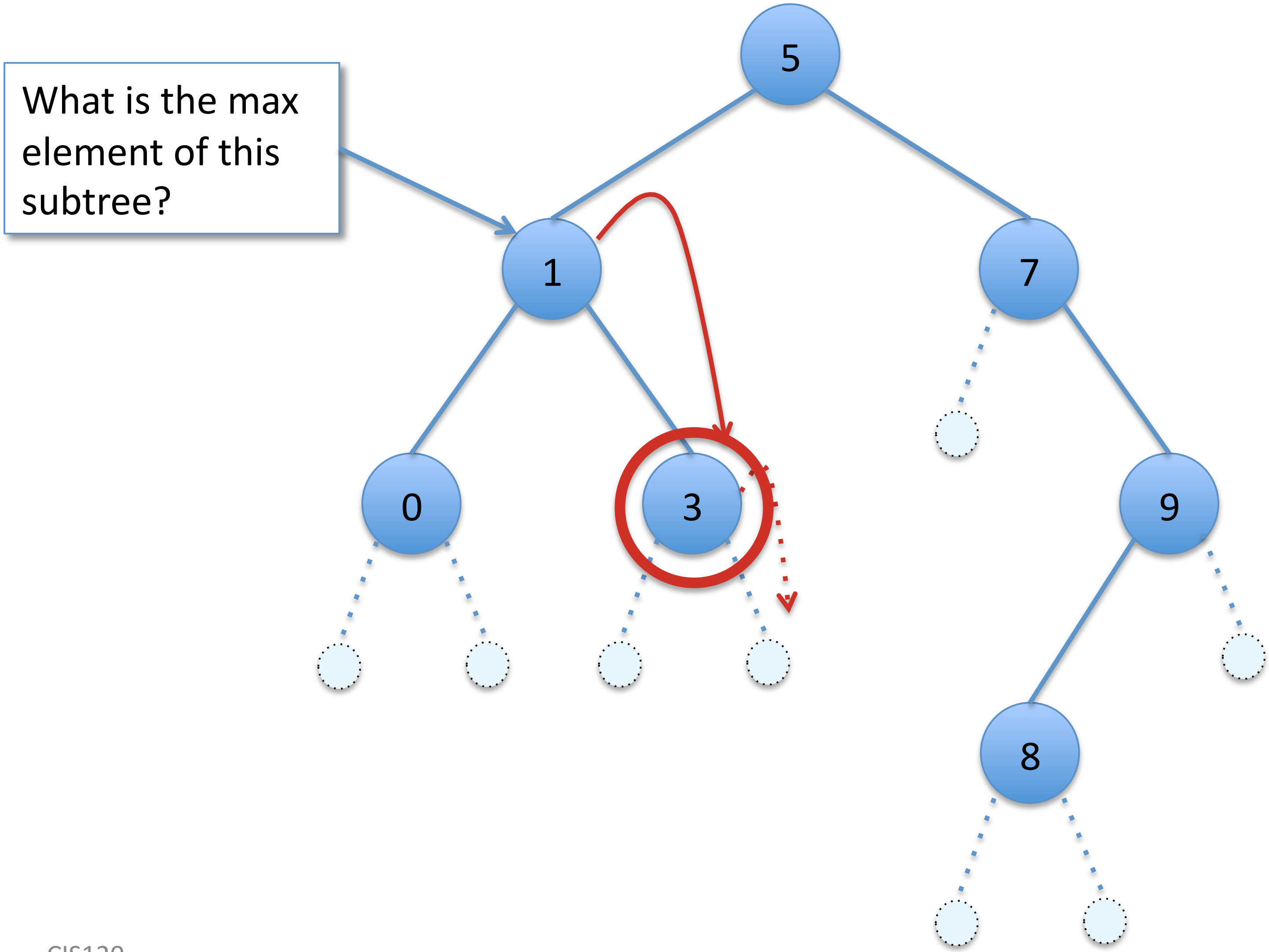
- To promote m we replace the deleted node by:

  Node(delete lt m, m, rt)

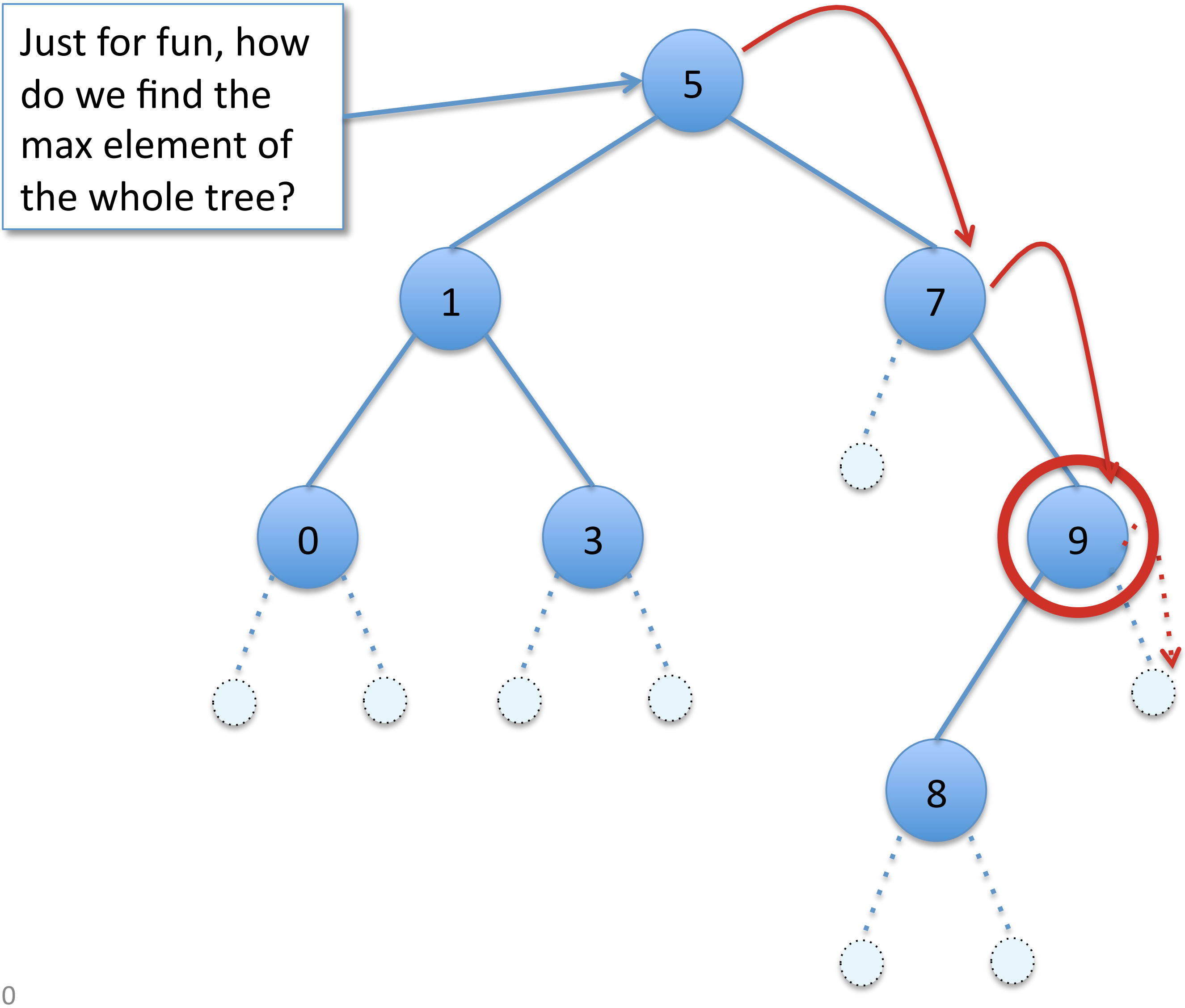
  – I.e. we recursively delete m from lt and relabel the root node m

  – The resulting tree satisfies the BST invariants

How to Find the Maximum Element?


What is the max element of this subtree?

How to Find the Maximum Element?



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

```
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:

- We never call tree_max on an empty tree
  - This is a consequence of the BST invariants and the case analysis done by the delete function
- BST invariant guarantees that the maximum-value node is farthest to the right

Deleting From a BST

```
(* return 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
```

If we insert a label n into a BST and then immediately delete n, do we always get back a tree of exactly the same shape?

1. yes
2. no

Answer: no, what if the node is in the tree

If we insert a label n into a BST *that does not already contain n* and then immediately delete n, do we always get back a tree of exactly the same shape?

1. yes
2. no

Answer: yes

If we delete n from a BST (containing n) and then immediately insert n again, do we always get back a tree of exactly the same shape?

1. yes
2. no

Answer: no, what if we delete the root?

Generic Functions and Data

Wow, that took quite a bit of typing... Do we have to repeat it all again if we want to use BSTs containing strings, or characters, or floats?

Structurally Identical Functions

- Observe: many functions on lists, trees, and other datatypes don’t depend on the contents, only on the structure.
- Compare: `length` for “`int list`” vs. “`string list`”

```
let rec length (l: int list) : int =
  begin match l with
  | [] -> 0
  | _::tl -> 1 + length tl
  end
```

```
let rec length (l: string list) : int =
  begin match l with
  | [] -> 0
  | _::tl -> 1 + length tl
  end
```

The functions are *identical*, except for the type annotation.

Notation for Generic Types

- OCaml provides syntax for functions with *generic* types

```
let rec length (l:'a list) : int =
  begin match l with
  | [] -> 0
  | _::tl -> 1 + (length tl)
  end
```

- Notation: `‘a` is a *type variable*; the function `length` can be used on a `t list` for *any* type `t`.
- Examples:
  - `length [1;2;3]` use length on an int list
  - `length [“a”;”b”;”c”]` use length on a string list

Generic List Append

Note that the two input lists must have the *same* type of elements.

The return type is the same as the inputs.

```
let rec append (l1:'a list) (l2:'a list) : 'a list =
  begin match l1 with
  | [] -> l2
  | h::tl -> h::(append tl l2)
  end
```

Pattern matching works over generic types!

In the body of the branch:

- `h` has type `‘a`
- `tl` has type `‘a list`

Generic Zip

Functions can operate over *multiple* generic types.

```
let rec zip (l1:'a list) (l2:'b list) : ('a*'b) list =
  begin match (l1,l2) with
  | (h1::t1, h2::t2) -> (h1,h2)::(zip t1 t2)
  | _ -> []
  end
```

- Distinct type variables can be instantiated differently:
  `zip [1;2;3] [“a”;”b”;”c”]`
- Here, `‘a` is instantiated to `int`, `‘b` to `string`
- Result is

  `[(1,”a”);(2,”b”);(3,”c”)]`

  of type `(int * string) list`

User-Defined Generic Datatypes

- Recall our integer tree type:

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

- We can define a generic version by adding a type parameter, like this:

```
type 'a tree =
| Empty
| Node of 'a tree * 'a * 'a tree
```

Parameter ‘a used here

Note that the recursive uses also mention ‘a

User-Defined Generic Datatypes

- BST operations can be generic too; only change is to type annotation

```
(* Insert n into the BST t *)

let rec insert (t:'a tree) (n:'a) : ’a 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
```

Equality and comparison work for any type of data