Programming Languages and Techniques (CIS120)

Lecture 8

Jan 28, 2013

BSTs II and Generic Types

Announcements

- Homework 2 due tomorrow

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:*

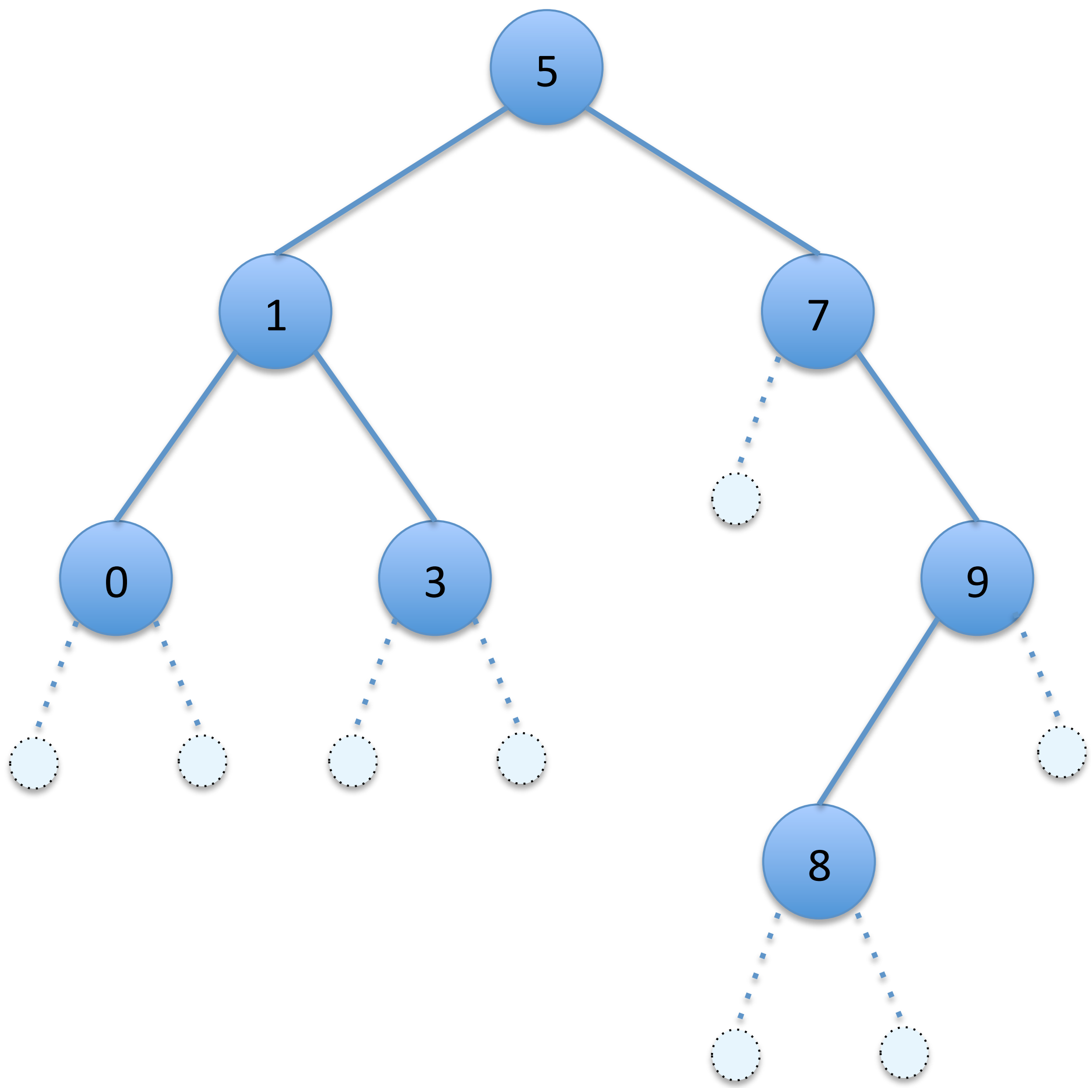
> - `Empty` is a BST
> - `Node(lt,x,rt)` is a BST if
>   - `lt` and `rt` are both BSTs
>   - all nodes of `lt` are `<` `x`
>   - all nodes of `rt` are `>` `x`

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?

Demo: bst.ml



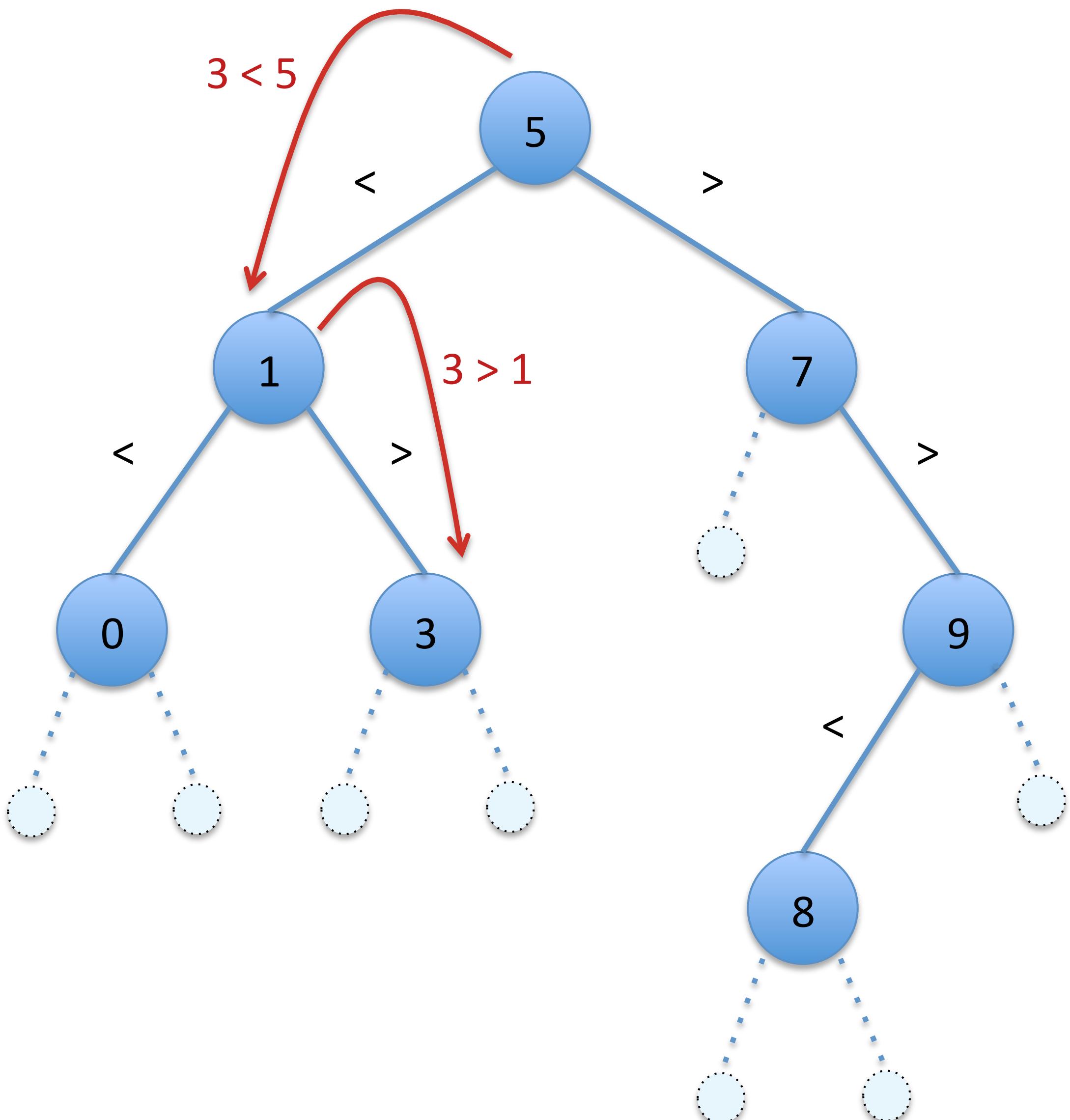
Checking the BST Invariants

```
(* 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
```

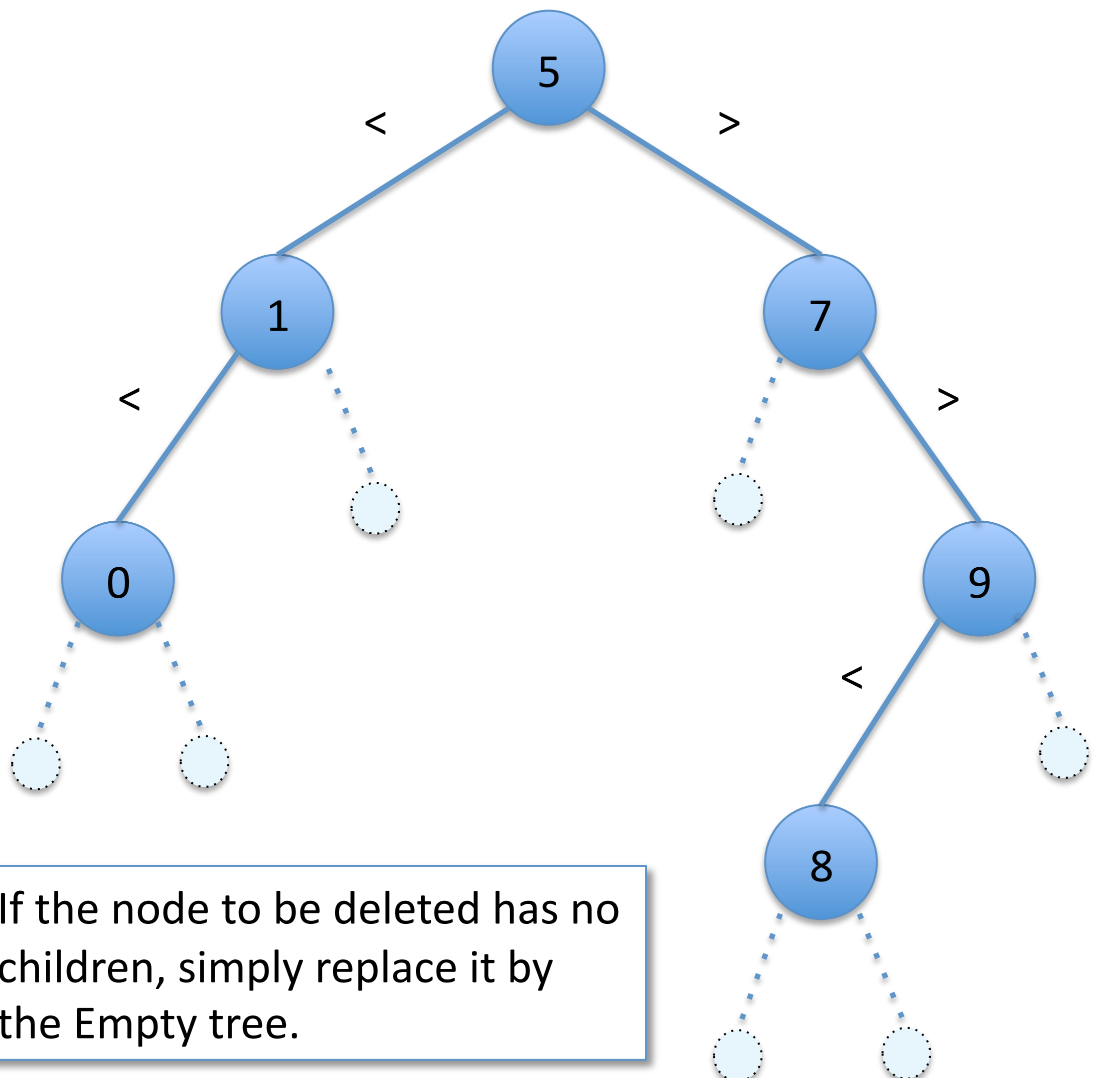
```
(* Determines whether t is a BST *)
let rec is_bst (t:tree) : bool =
  begin match t with
  | Empty -> true
  | Node(lt,x,rt) ->
      is_bst lt && is_bst rt &&
      (tree_less lt x) && (tree_gtr rt x)
  end
```

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

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

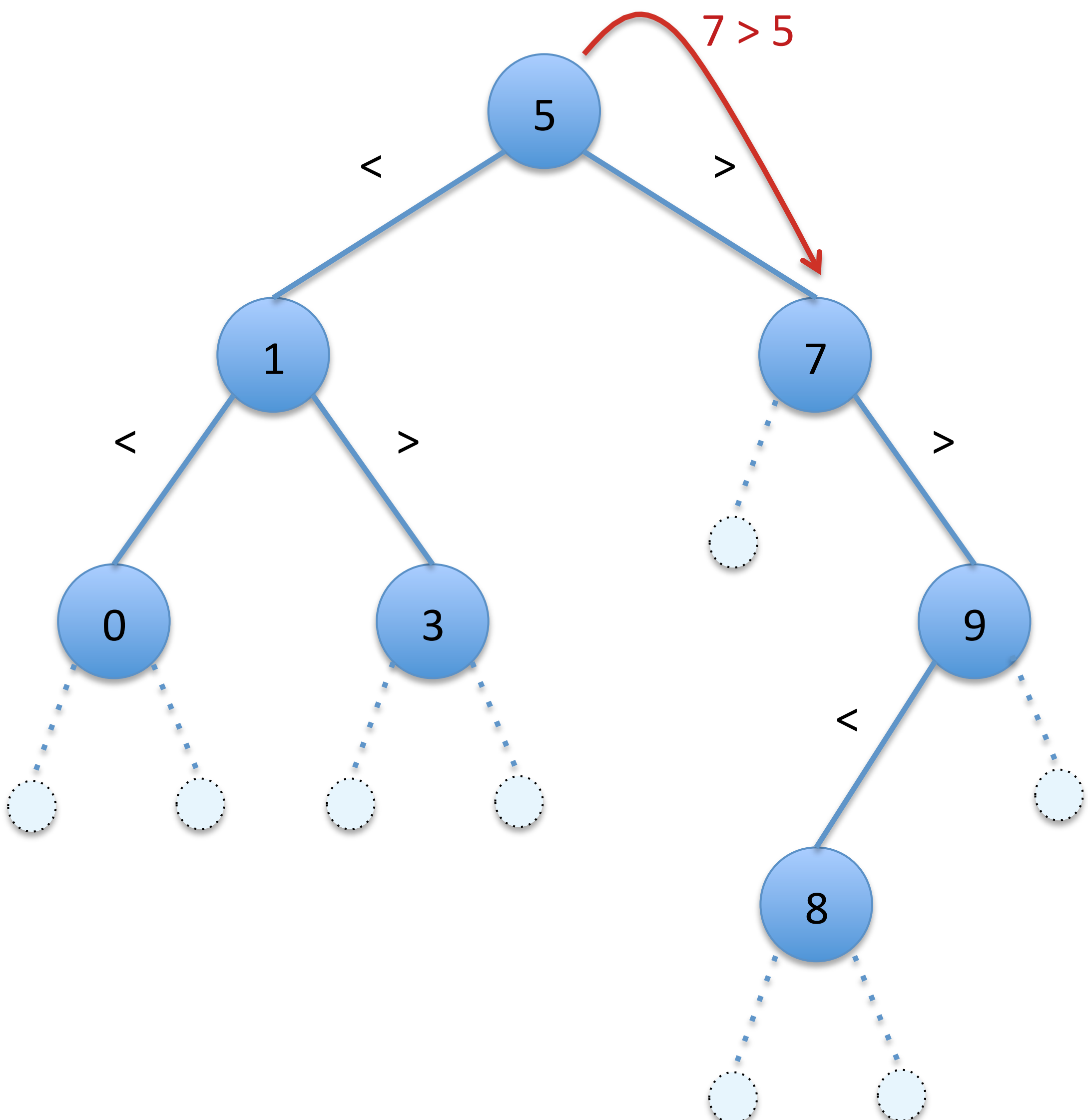


Deletion – No Children: `(delete t 3)`



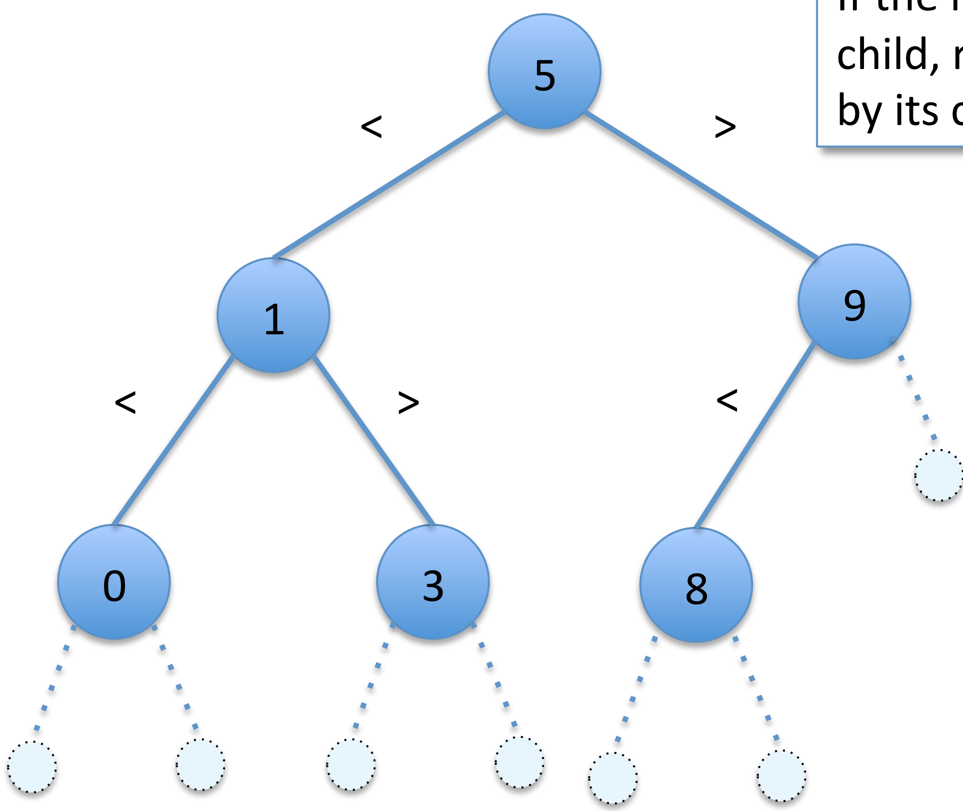
If the node to be deleted has no children, simply replace it by the Empty tree.

Deletion – One Child: `(delete t 7)`

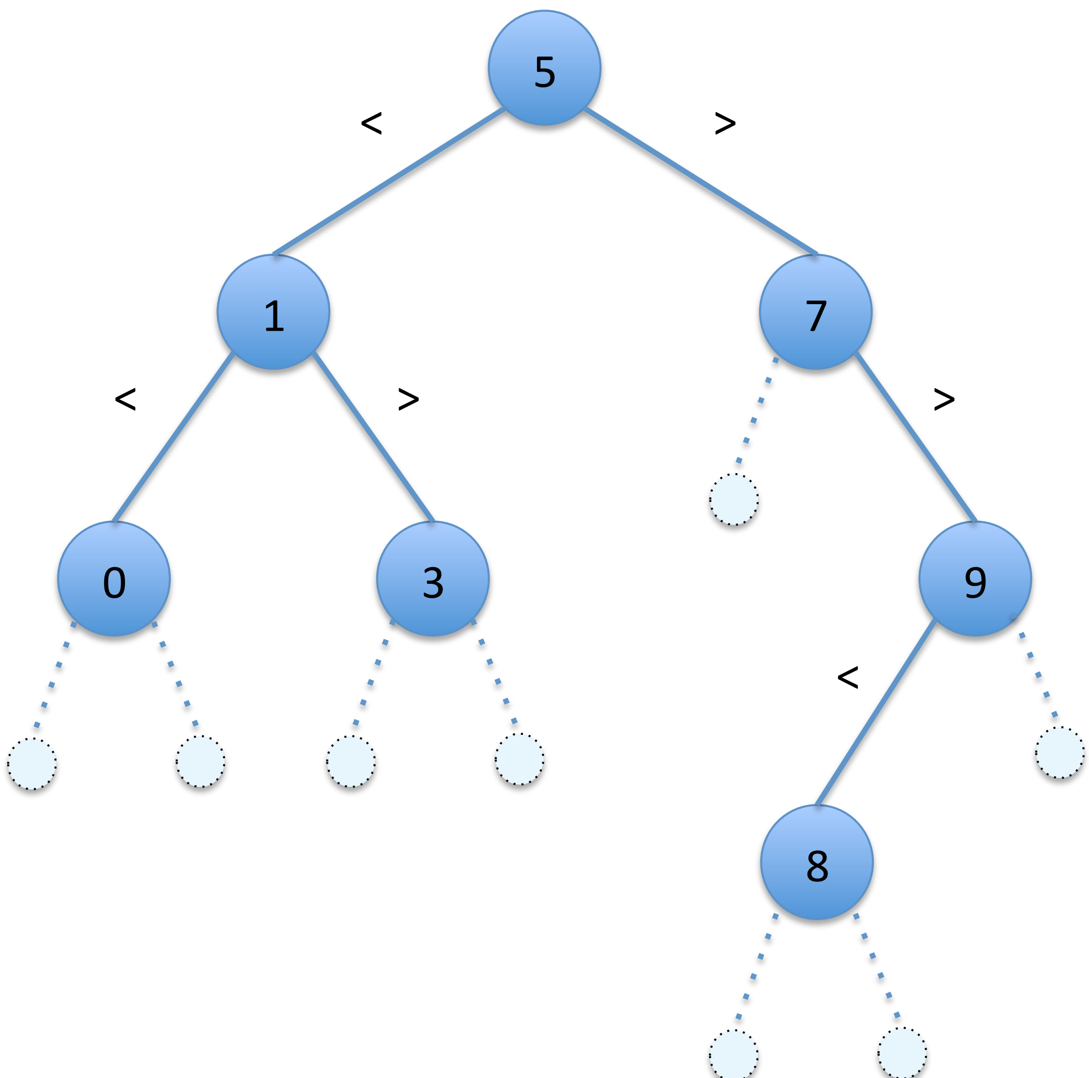


Deletion – One Child: `(delete t 7)`

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

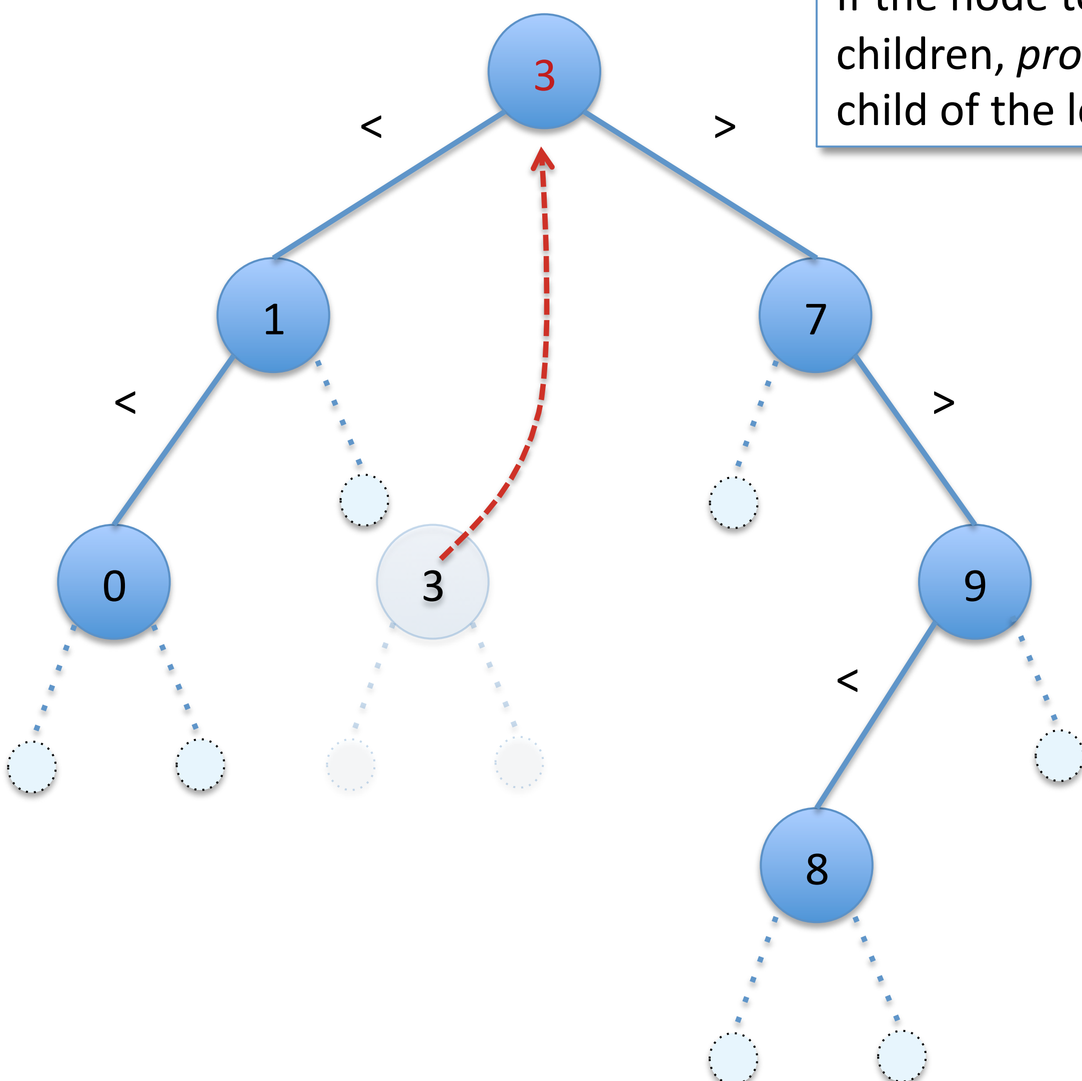


Deletion – Two Children: `(delete t 5)`



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
```

Generic Functions and Data

Wow, that was a lot of work. What about 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 length1 (l:int list) : int =
  begin match l with
  | [] -> 0
  | _::tl -> 1 + length1 tl
  end
```

```
let rec length2 (l:string list) : int =
  begin match l with
  | [] -> 0
  | _::tl -> 1 + length2 tl
  end
```

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

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 can also be generic – in this case the result is of the same type 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` instantiated to `int`, `‘b` to `string`
- Result is the `(int * string) list`:

  `[(1,”a”);(2,”b”);(3,”c”)]`

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:

Parameter ‘a used here

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

Note that the recursive uses also mention ‘a

User-defined Generic Datatypes

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

```
(* Inserts 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