# Programming Languages and Techniques (CIS1200)

Lecture 7

Binary Search Trees

(Chapters 7 & 8)

# Announcements

- Check out the entry survey on Ed
  - Help us get to know you!
- HW2 due *Tuesday* at 11.59pm
- Read Chapters 7 & 8
  - Binary Search Trees
- Midterm 1: Friday, September 27th
  - Details will be posted on Ed and announced in class
  - Look for announcements about review session, etc.
  - Content: HW 1 – 3, Chapters 1-10 of lecture notes
  - Contact cis1200@seas.upenn.edu with concerns

# Recap: Ordered Trees

Big idea: find things faster by searching less

*Key Insight:*

*Ordered data can be searched more quickly*

- This is why telephone books are arranged alphabetically
- Requires the ability to focus on (roughly) *half* of the current data

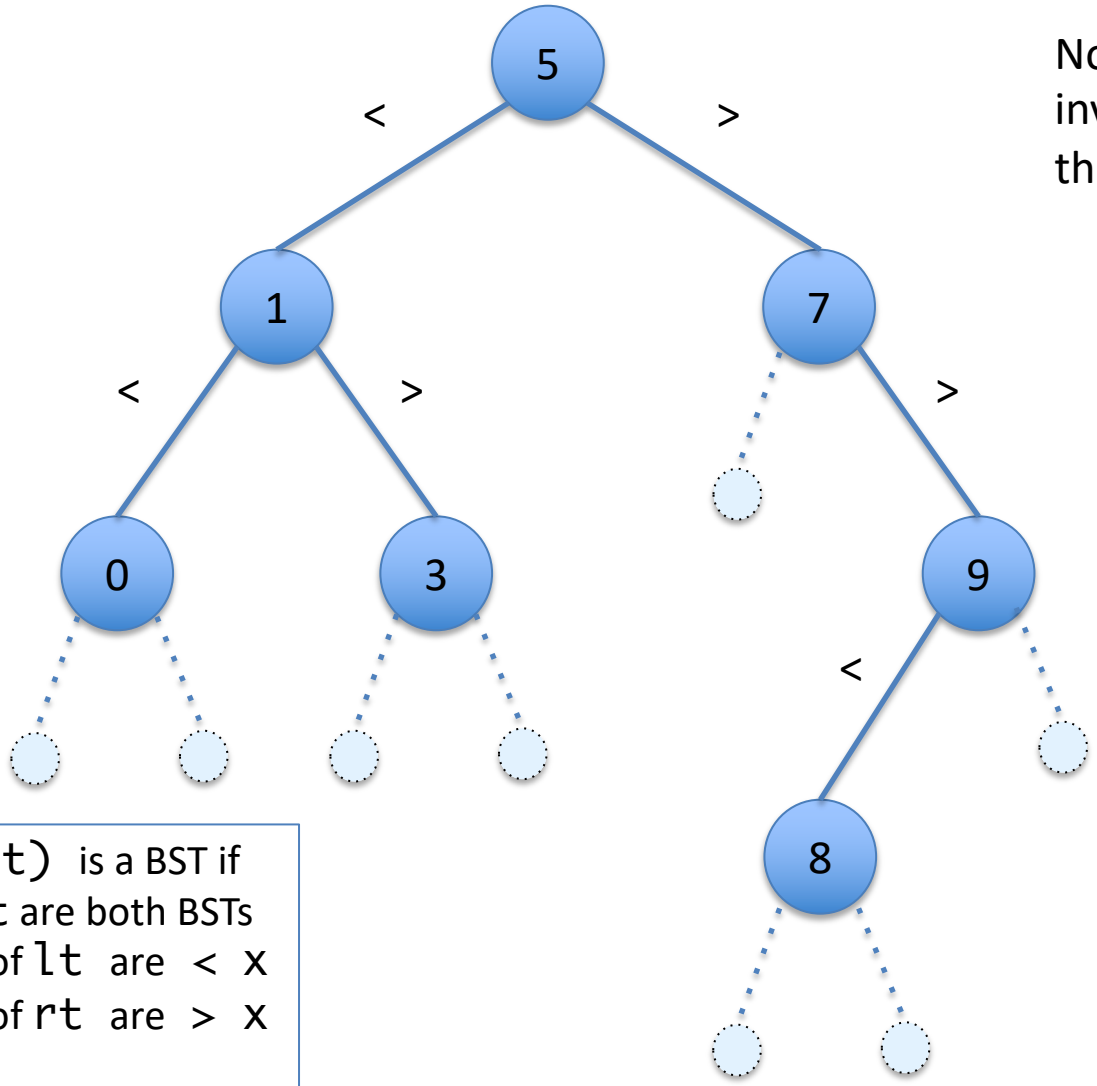
# Binary Search Trees

- A *binary search tree* (BST) is a binary tree with 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

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

# An Example Binary Search Tree




Note that the BST invariants hold for this tree.

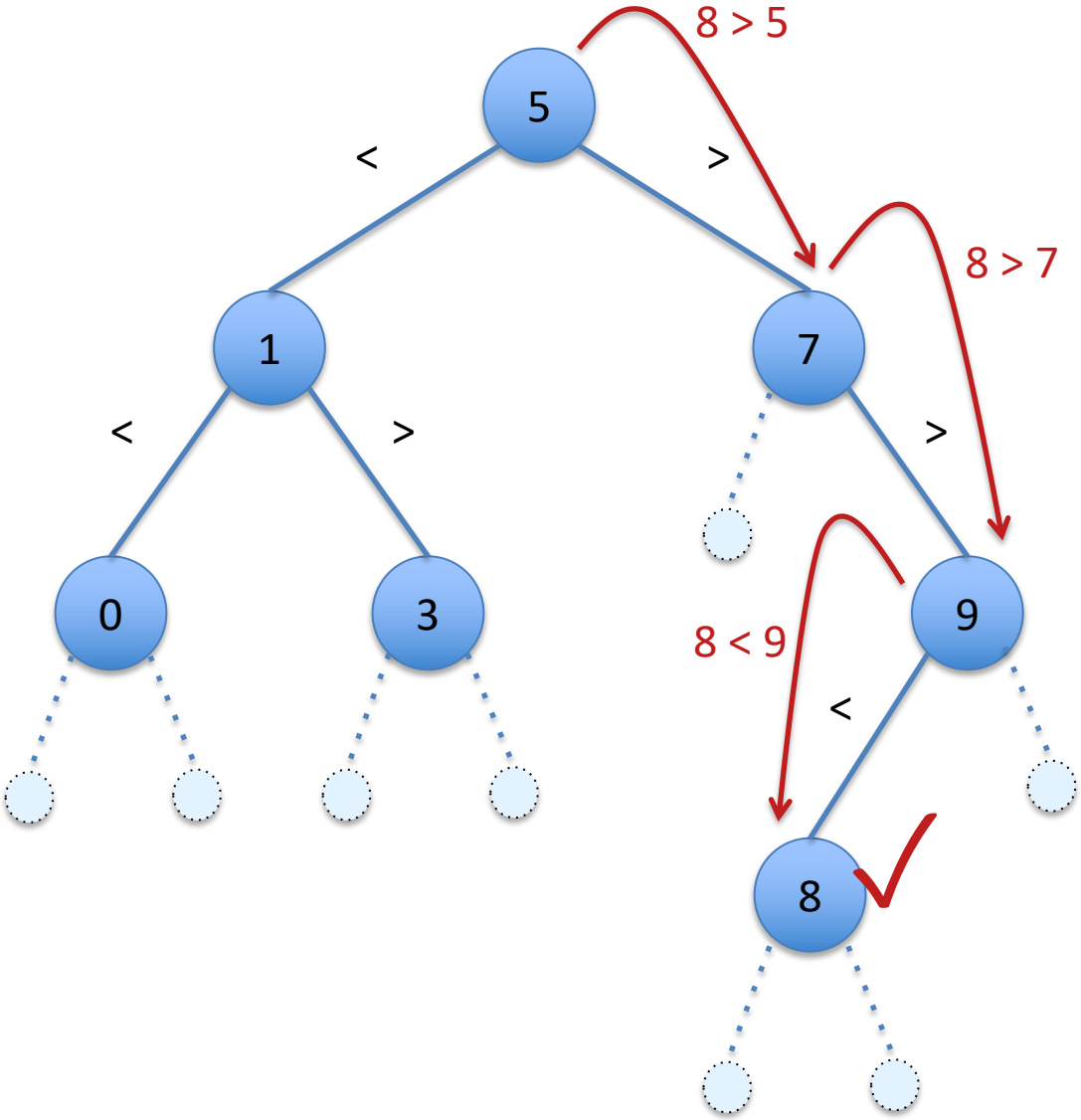
- `Node(lt,x,rt)` is a BST if
  - `lt` and `rt` are both BSTs
  - all nodes of `lt` are `<` `x`
  - all nodes of `rt` are `>` `x`
- `Empty` is a BST

# 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.

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



# Manipulating BSTs

Inserting an element

```
insert : tree -> int -> tree
```

"`insert t x`" returns a new tree containing x and all of the elements of t

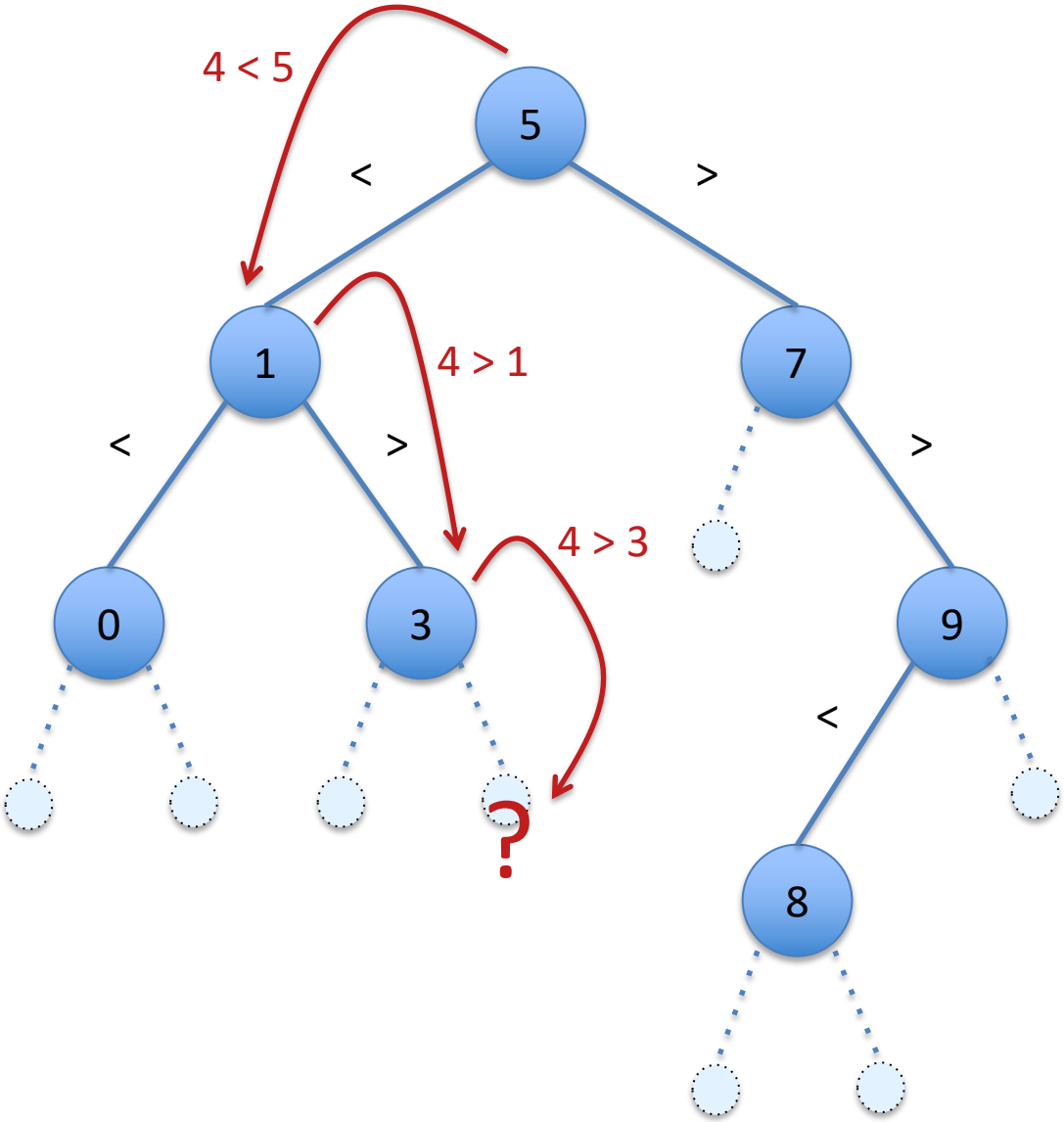
# Inserting into a BST

- Challenge: can we make sure that the result of insert really is a BST?
  - i.e., the new element needs to be in the right place!

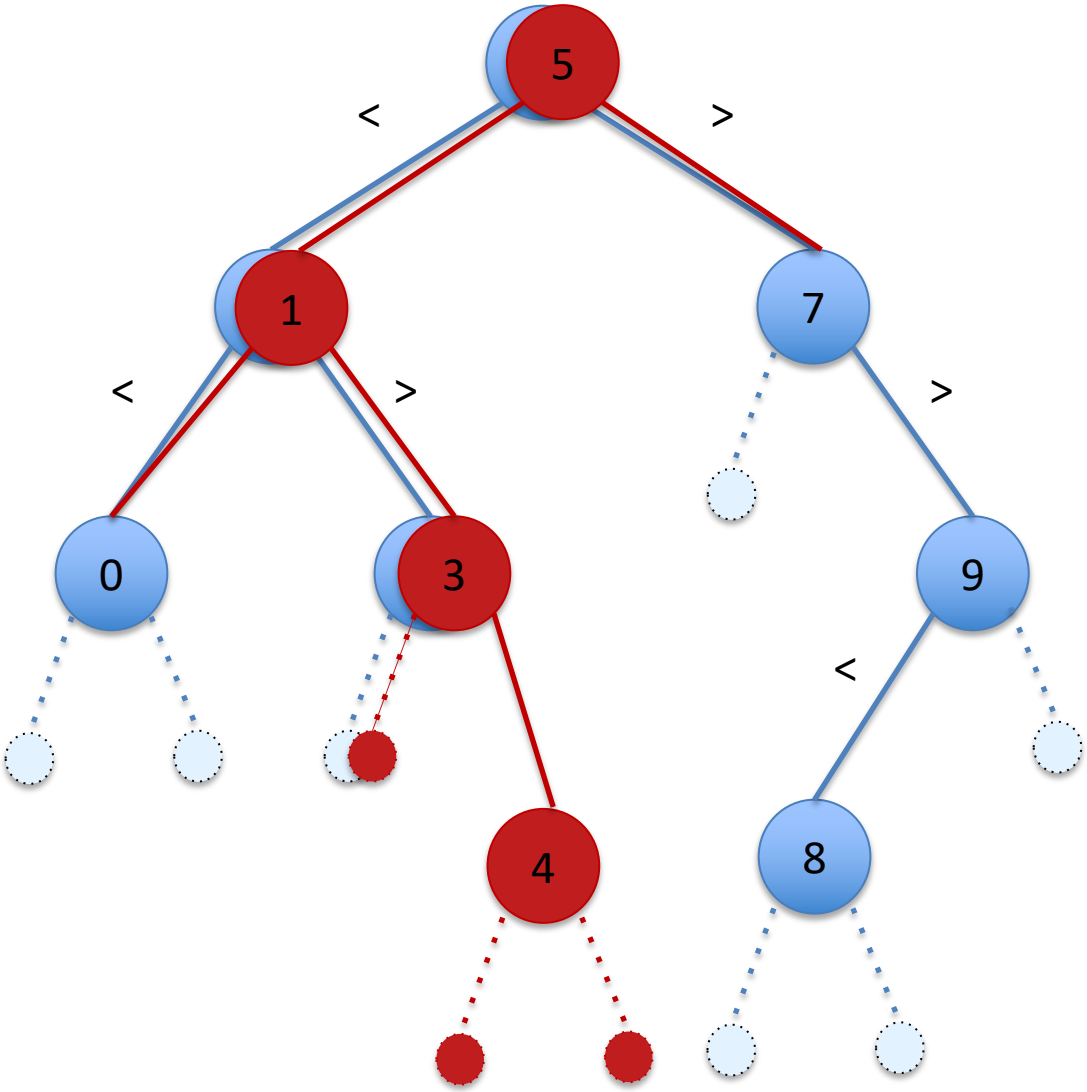
- Payoff: we can build a BST containing any set of elements
  - Starting with `Empty`, apply insert repeatedly
  - If insert *preserves* the BST invariants, then any tree we get from it will be a BST *by construction*
    - No need to check!
  - Later: we can also "rebalance" the tree to make lookup efficient (NOT in CIS 1200; see CIS 1210)

*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

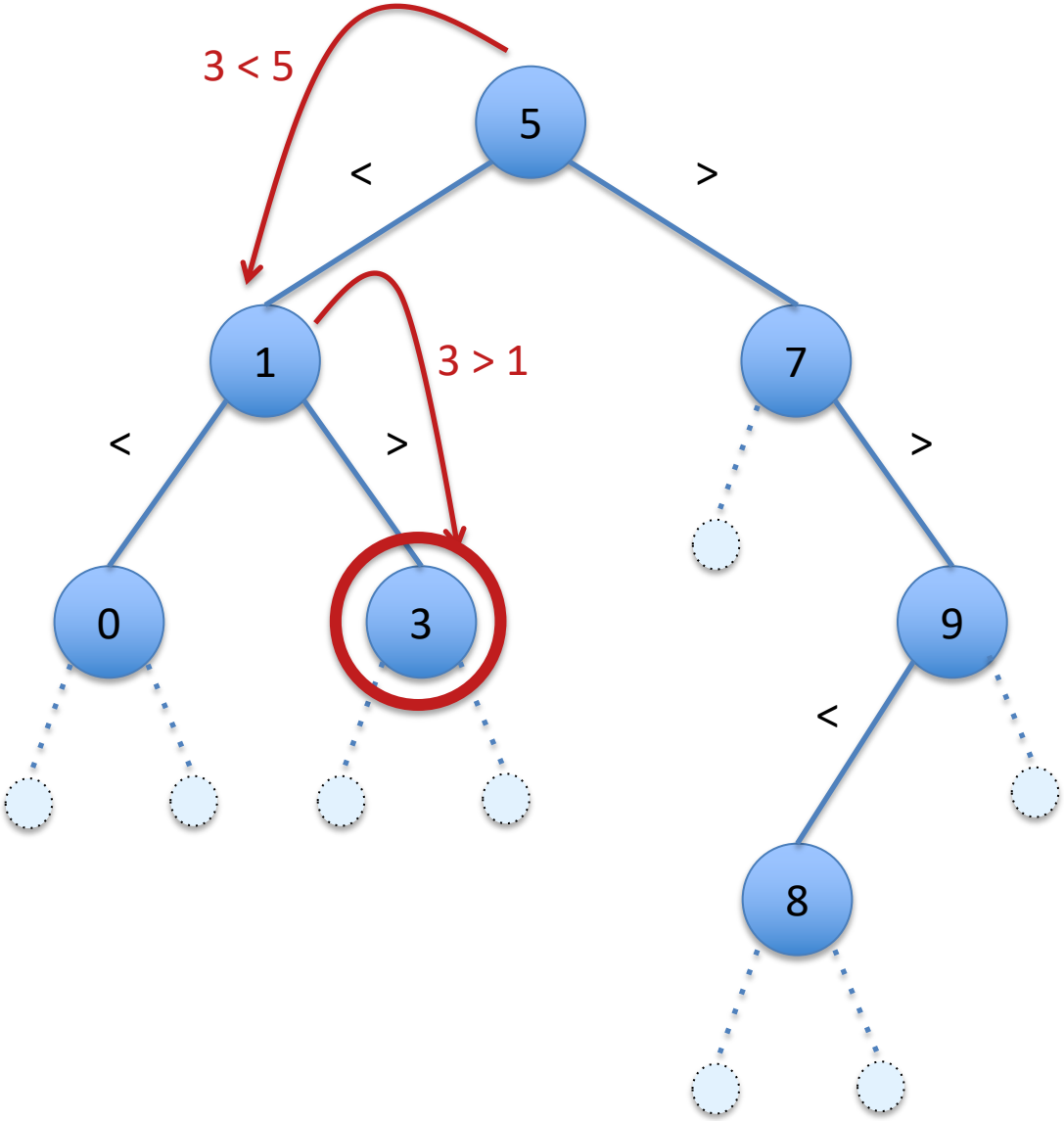
# Manipulating BSTs

Deleting an element

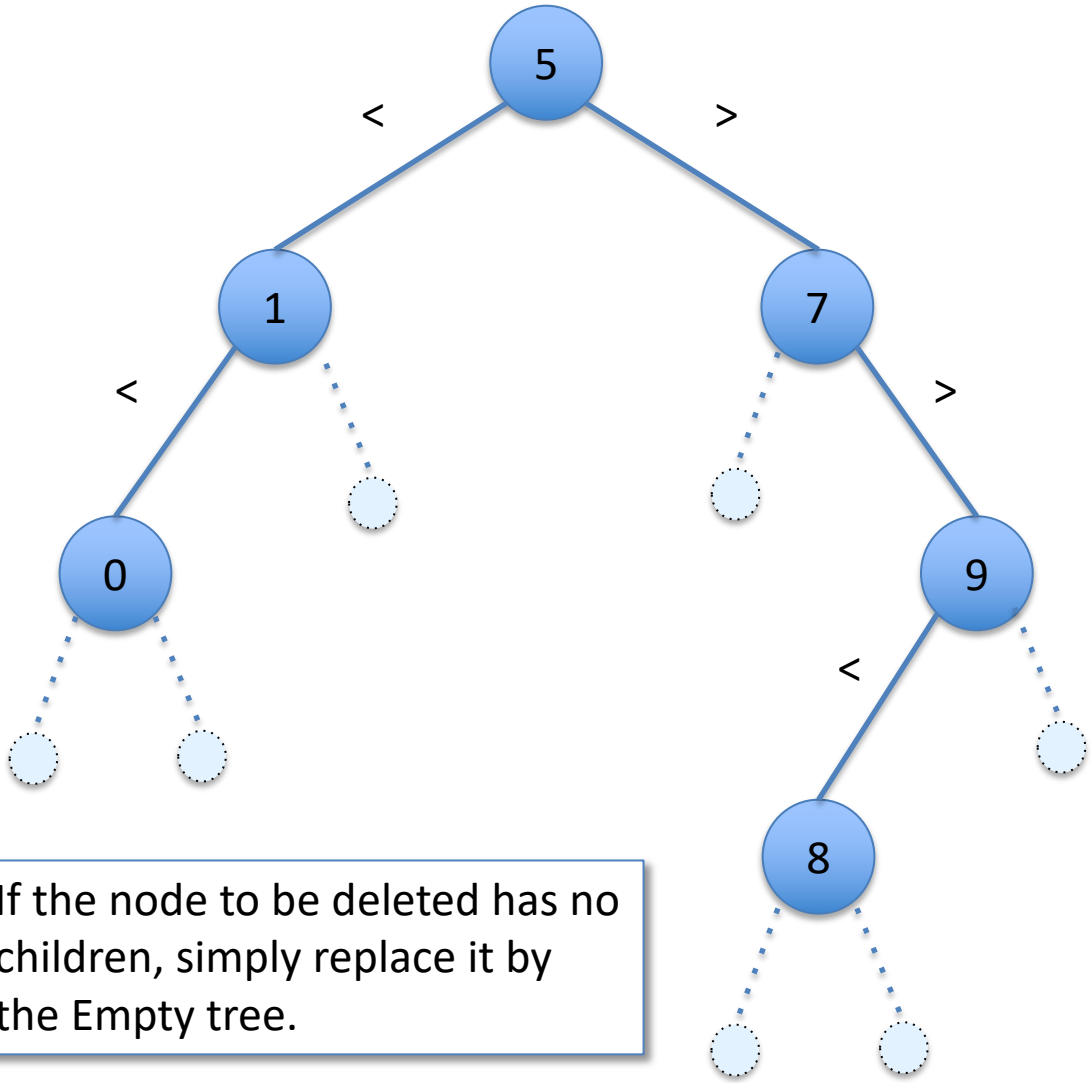
```
delete : tree -> int -> tree
```

"`delete t x`" returns a tree containing all of the elements of t except for x

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

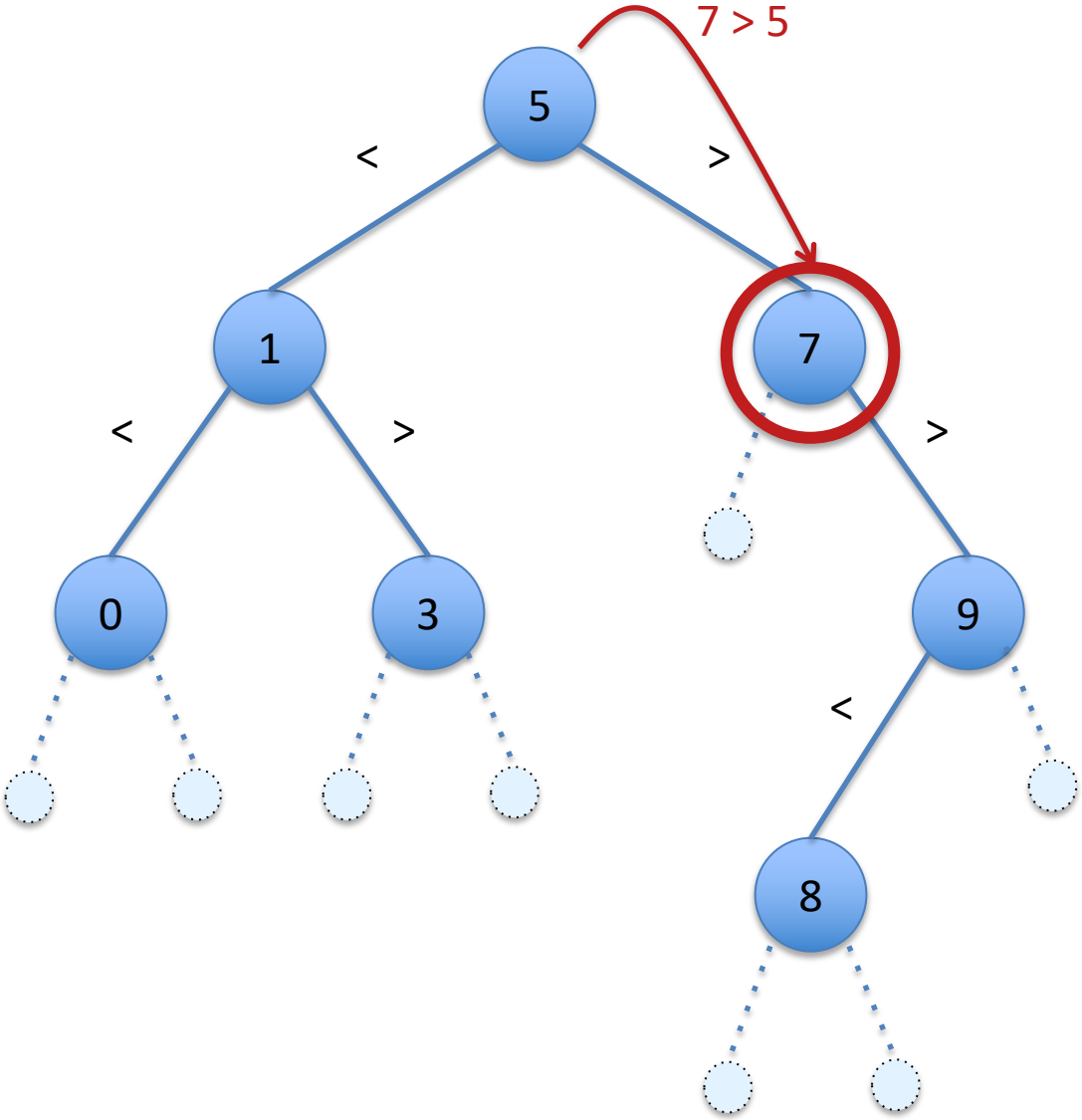


# Deletion – No Children: `(delete t 3)`



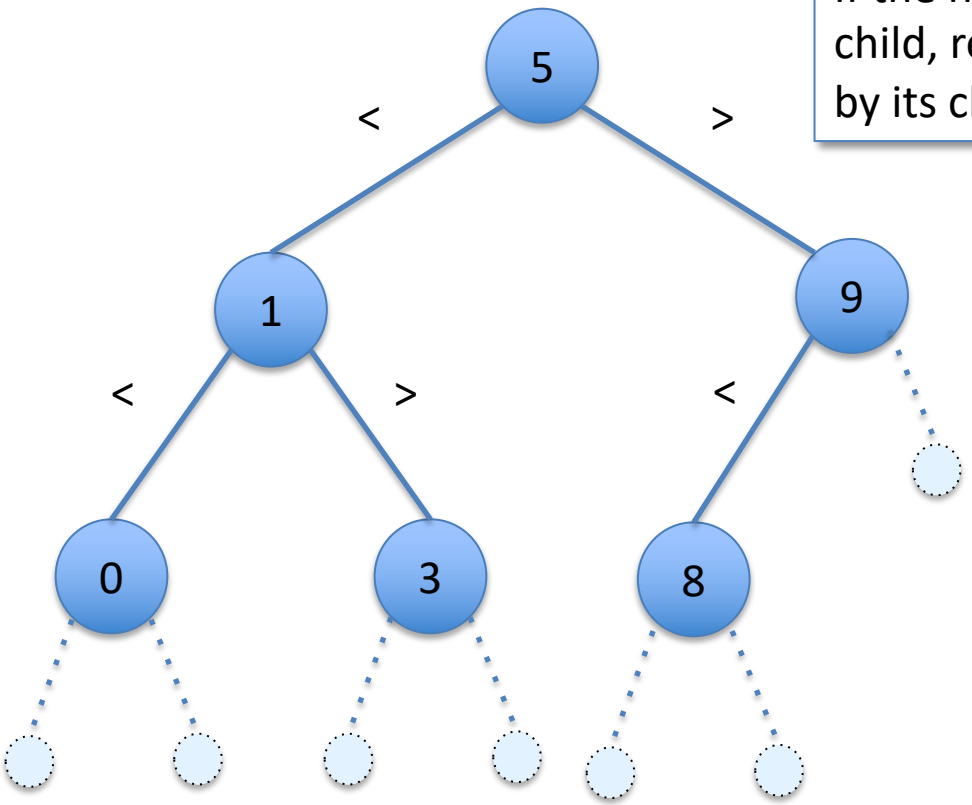

If the node to be deleted has no children, simply replace it by the Empty tree.

# Deletion – One Child: `(delete t 7)`

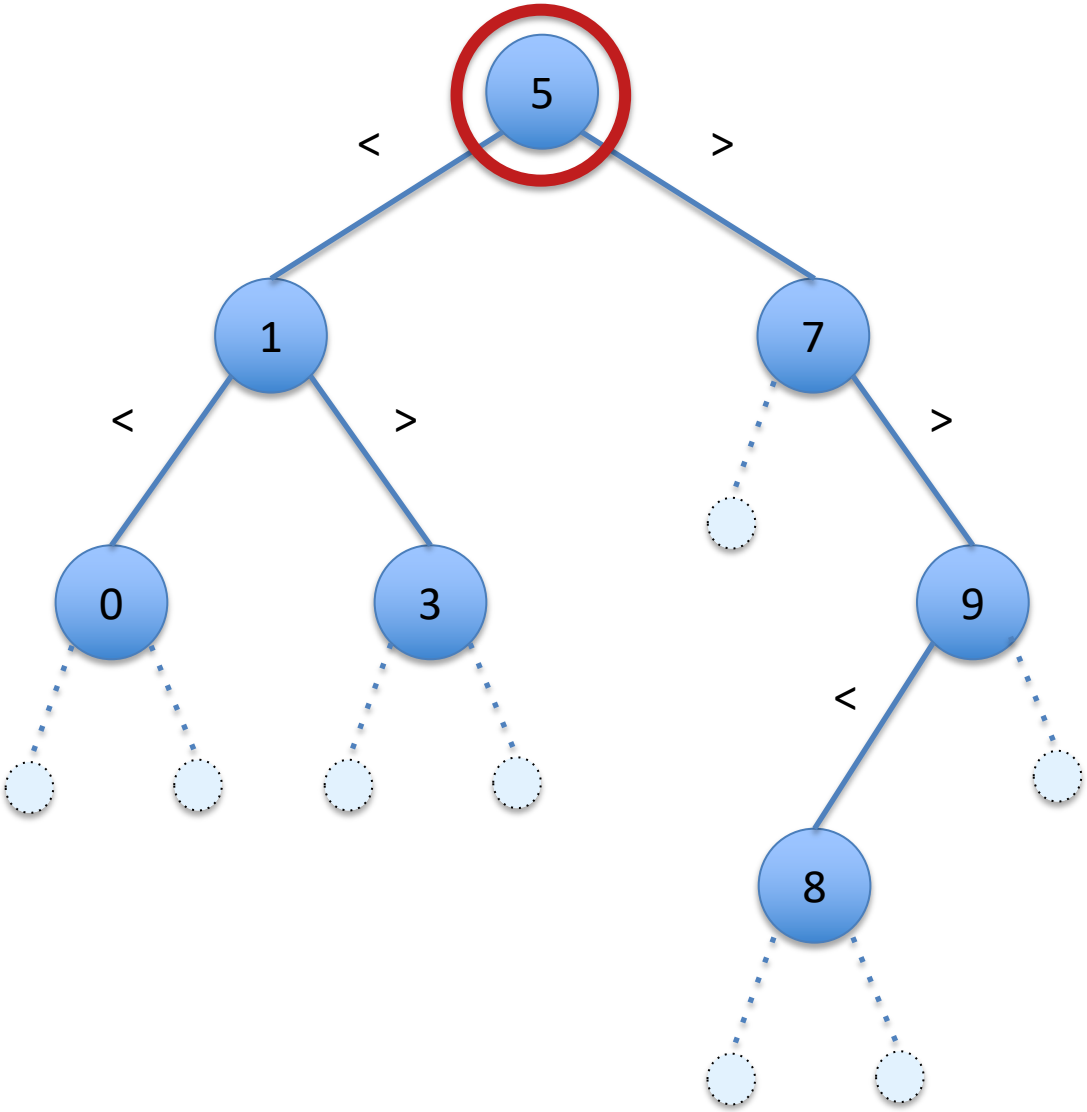


# Deletion – One Child: `(delete t 7)`

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

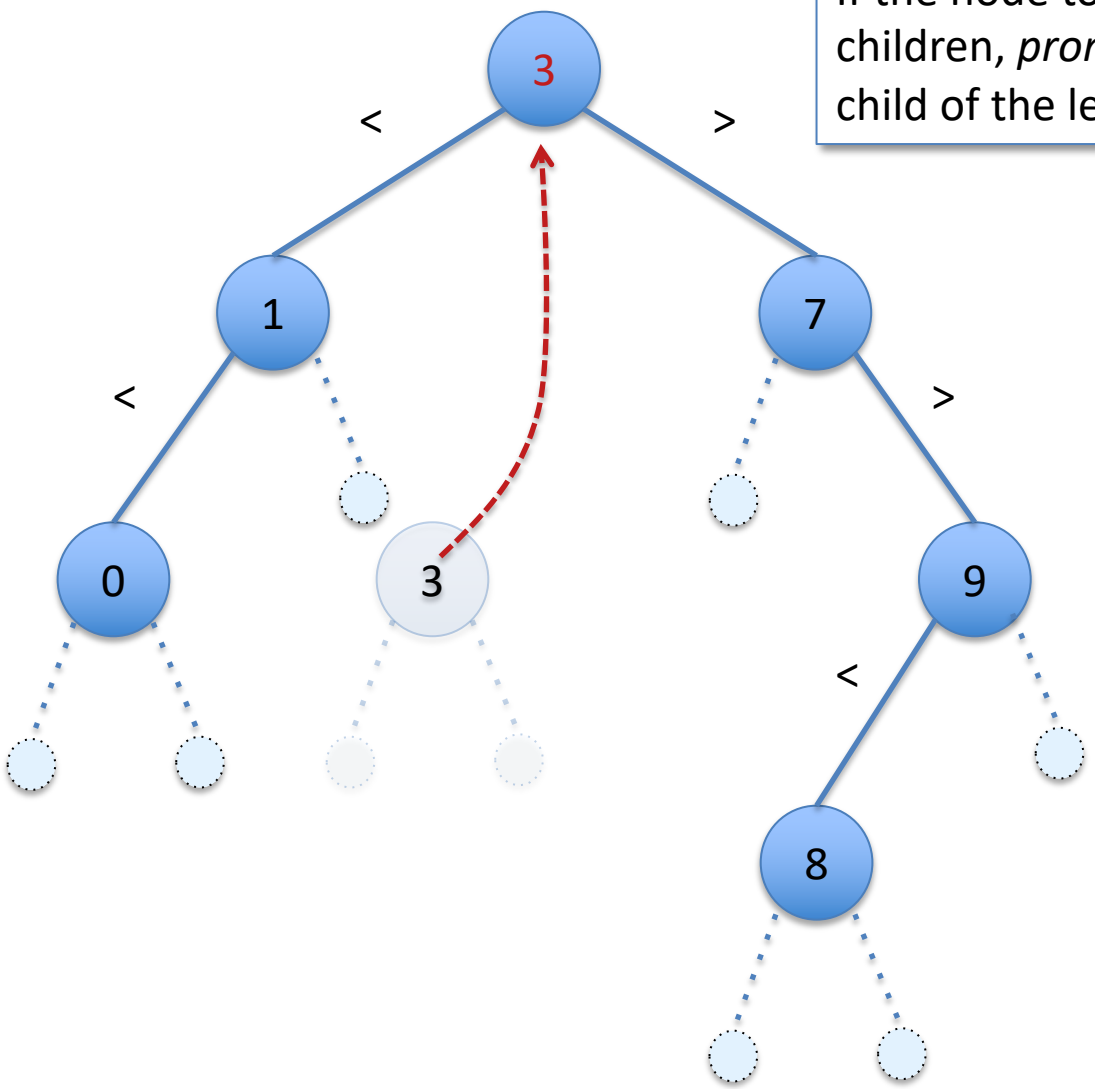


# Deletion – Two Children: `(delete t 5)`

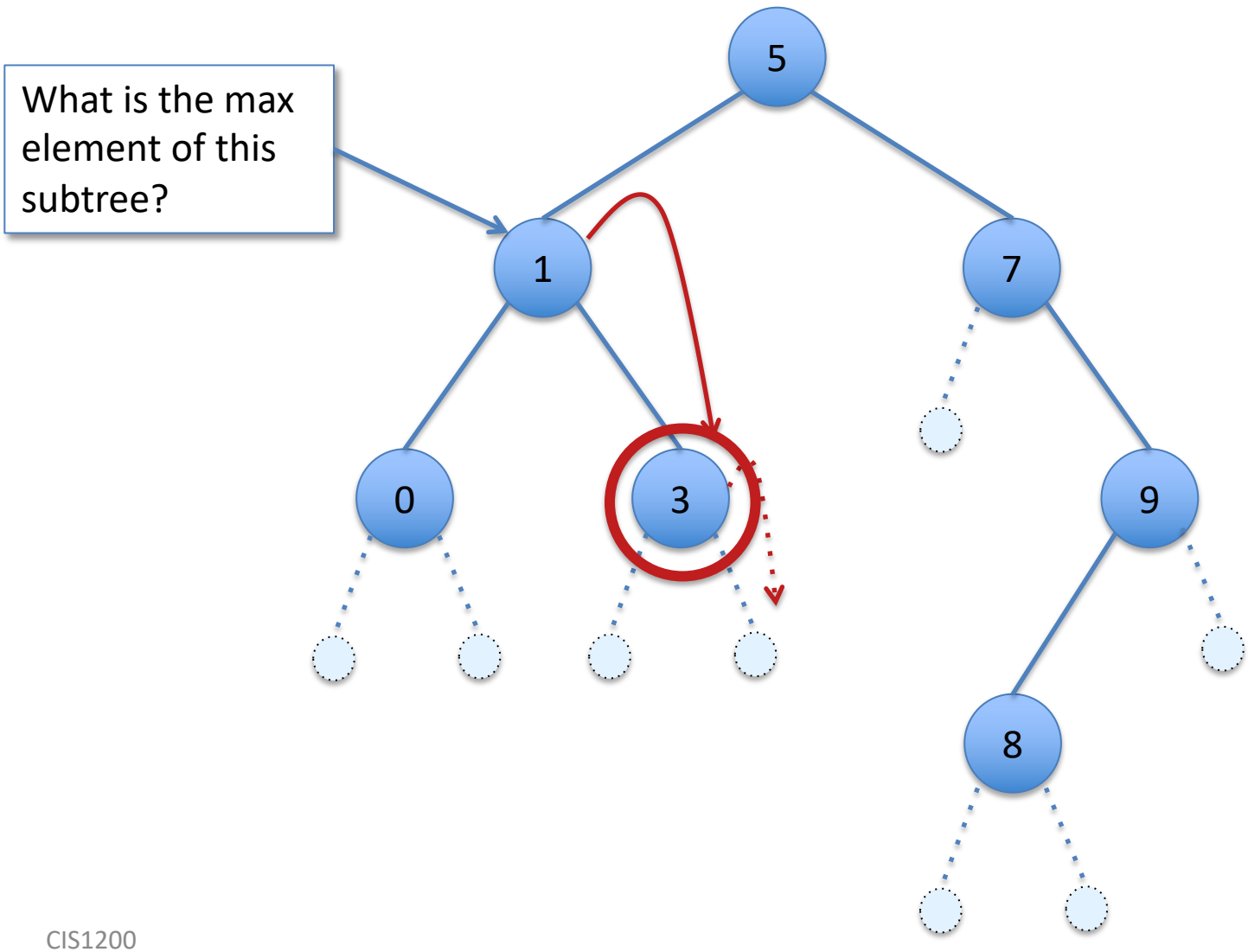


# Deletion – Two Children: `(delete t 5)`

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

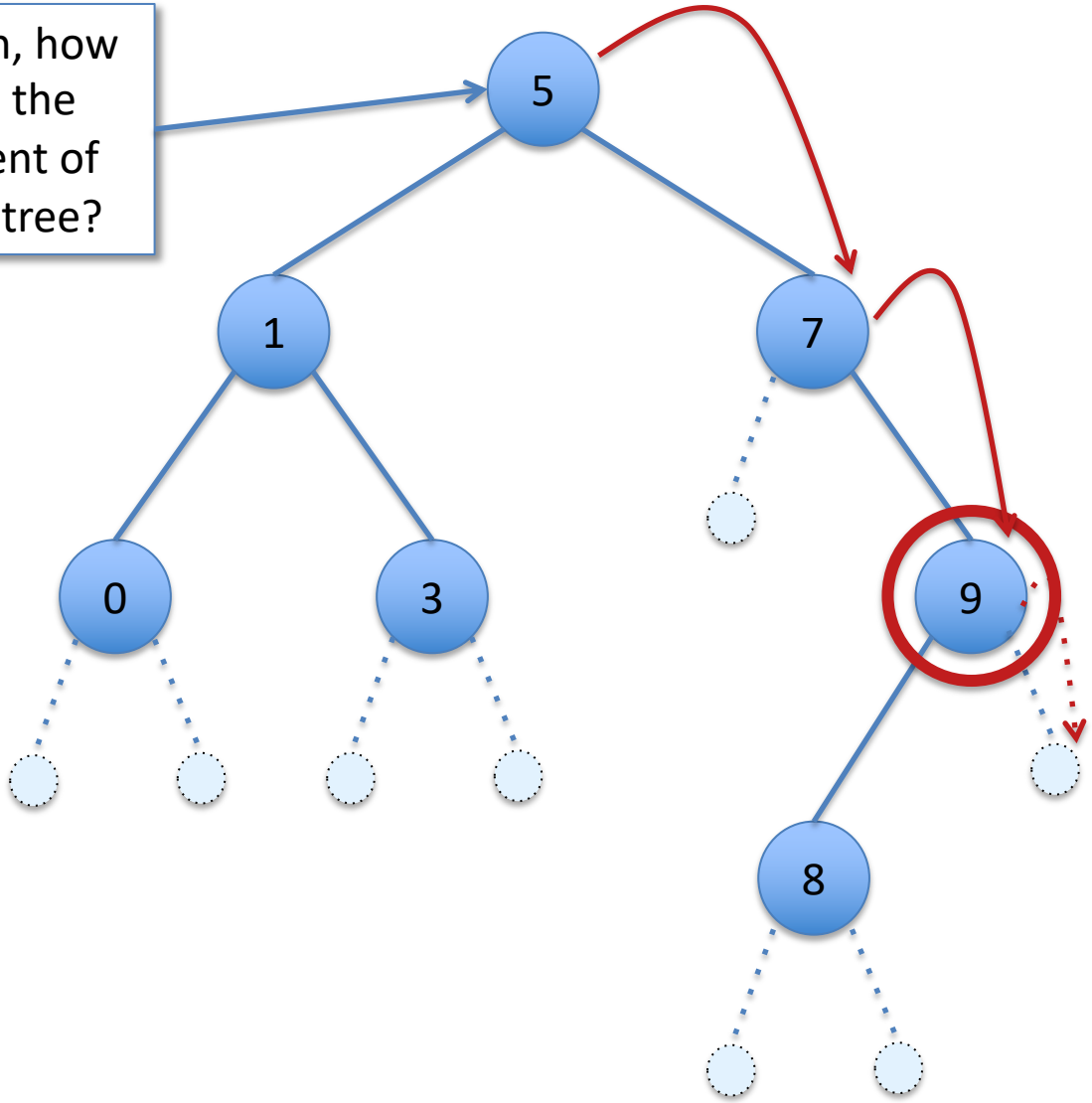


# How to Find the Maximum Element?



# How to Find the Maximum Element?

Just for fun, how do we find the max element of the 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"

# Code for BST delete

bst.ml

# Deleting From a BST

```
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
```

See bst.ml

# 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

## 7: 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?

0

yes

0%

no

0%

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 was in the tree to begin with?)

7: If we insert a value 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?

0

yes

0%

no

0%

If we insert a value 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

## 7: 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?

0

yes

0%

no

0%

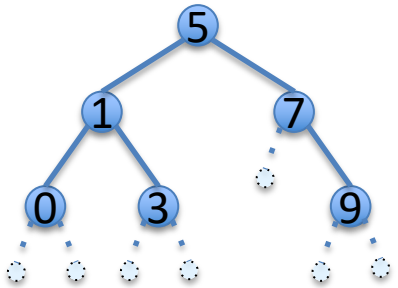
If we delete n from a BST (containing n) and then immediately insert n again, do we always get back a tree of exactly the same shape?

1. yes
2. no

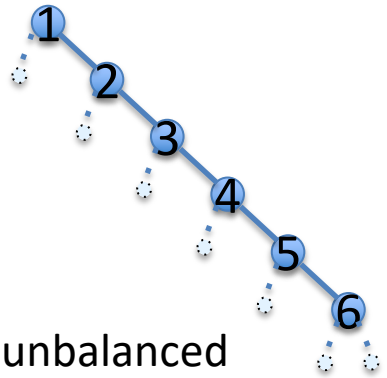
Answer: no (e.g., what if we delete the item at the root node?)

# BST Performance

- `lookup` takes time proportional to the *height* of the tree.
  - not the *size* of the tree (as it did with `contains` for 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

# Demo

bst.ml – compare contains and lookup

# Generic Functions and Data

Wow, implementing BSTs took quite a bit of typing... Do we have to do it all again if we want to use BSTs containing strings, and again for characters, and again for floats, and…?

or

*How not to repeat yourself, Part I.*

# Structurally Identical Functions

- Observe: many functions on lists, trees, and other datatypes don't depend on the contents, only on the structure.
- Compare:

```
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 allows defining 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,* indicating that 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
- Idea: OCaml fills in `'a` whenever length is used

# 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`

# Zip function

- Combine two lists into one list
  - ignore elements from longer list if they are not the same length

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

```
zip [1;2;3] ["a";"b";"c"]
  ⟼ [(1,"a"); (2,"b"); (3,"c")]
```

- Does it matter what type of lists these are?

# 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`

Intuition: OCaml tracks instantiations of type variables ('a and 'b) and makes sure they are used consistently

# 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 *do not need to be* instantiated differently:

```
zip [1;2;3] [4;5;6]
```

- Here, `'a` is instantiated to `int`, `'b` to `int`
- Result is

```
[(1,4);(2,5);(3,6)]
```

of type `(int * int) list`

Intuition: OCaml tracks instantiations of type variables ('a and 'b) and makes sure they are used consistently

# 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 of `tree` also mention `'a`

# User-Defined Generic Datatypes

- BST operations can be generic too; the only change is to the 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 are generic — they work for *any* type of data.

Does the following function typecheck?

```
let f (l : 'a list) : 'b list =
begin match l with
| [] -> true::l
| _::rest -> 1::l
end
```

1. yes
2. no

Answer: no: even though the return type is generic, the two branches must agree (so that ‘b can be consistently instantiated).

Does the following code typecheck?

```
let f (x : 'a) : 'a =
  x + 1

;; print_endline (f “hello”)
```

1. yes
2. no

Answer: no, the type annotations and uses of f aren’t consistent.

However it is a bit subtle: without the use (f "hello") the code *would* be correct – so long as all uses of f provide only 'int' the code is consistent! Despite the "generic" type annotation, f really has type int -> int.