Programming Languages and Techniques (CIS1200)

Lecture 7

Binary Search Trees

(Chapters 7 & 8)

Announcements

- Wellness committee!
- 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, February 14th
  - 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

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

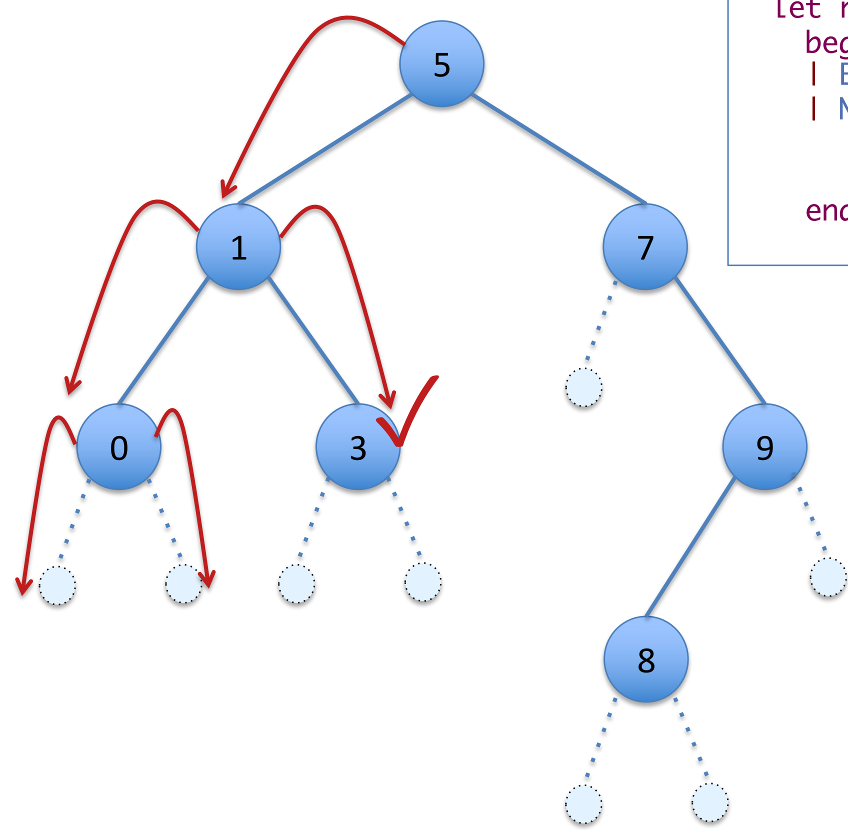
```
type tree =
| Empty
| Node of tree * int * tree
```

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
```

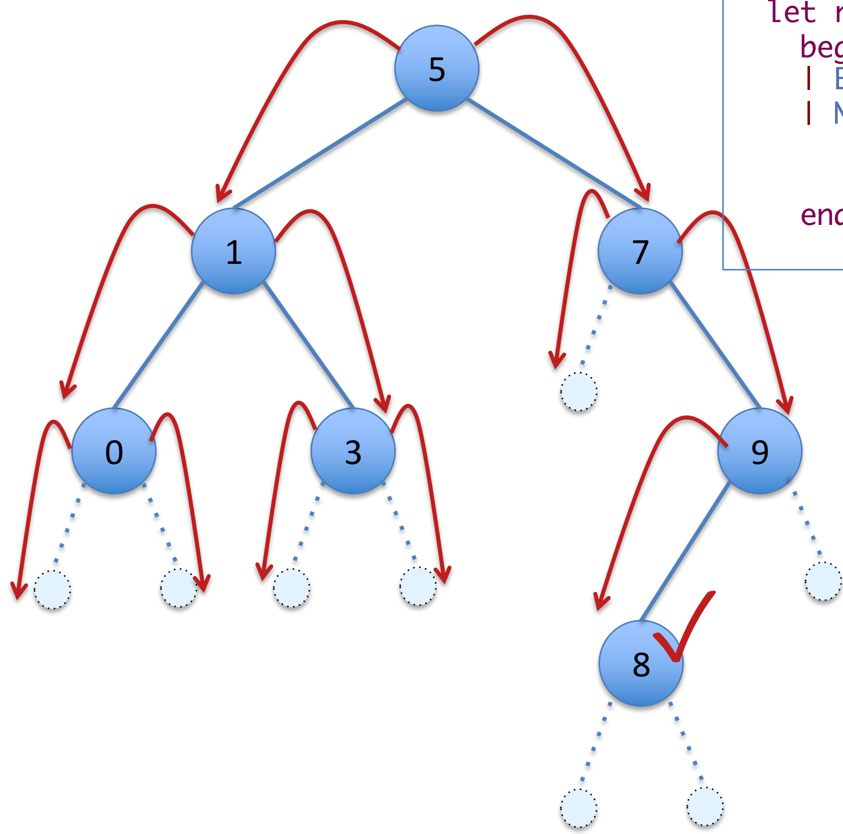
- This function searches through the tree `t`, looking for a number n
- The || operator is a *short-circuiting "or"*
  - When computing b||c, if b simplifies to `true`, then c is ignored
  - This can save time if simplifying c is expensive
- Even so, `contains` might have to traverse the *entire tree*

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

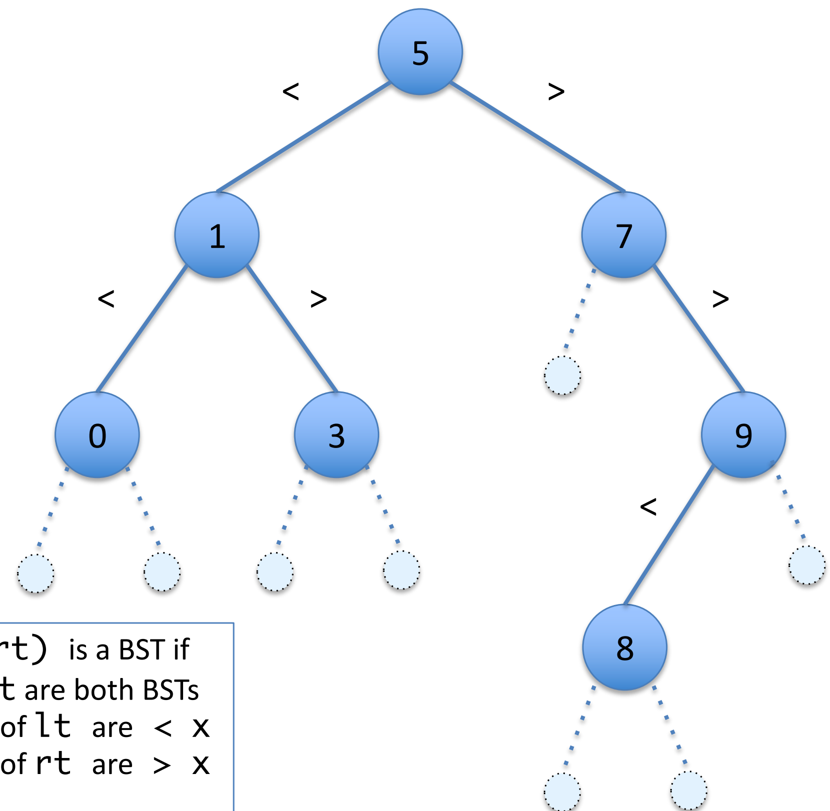
- A *binary search tree* (BST) is a binary tree with an additional *invariant**:

> - `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.

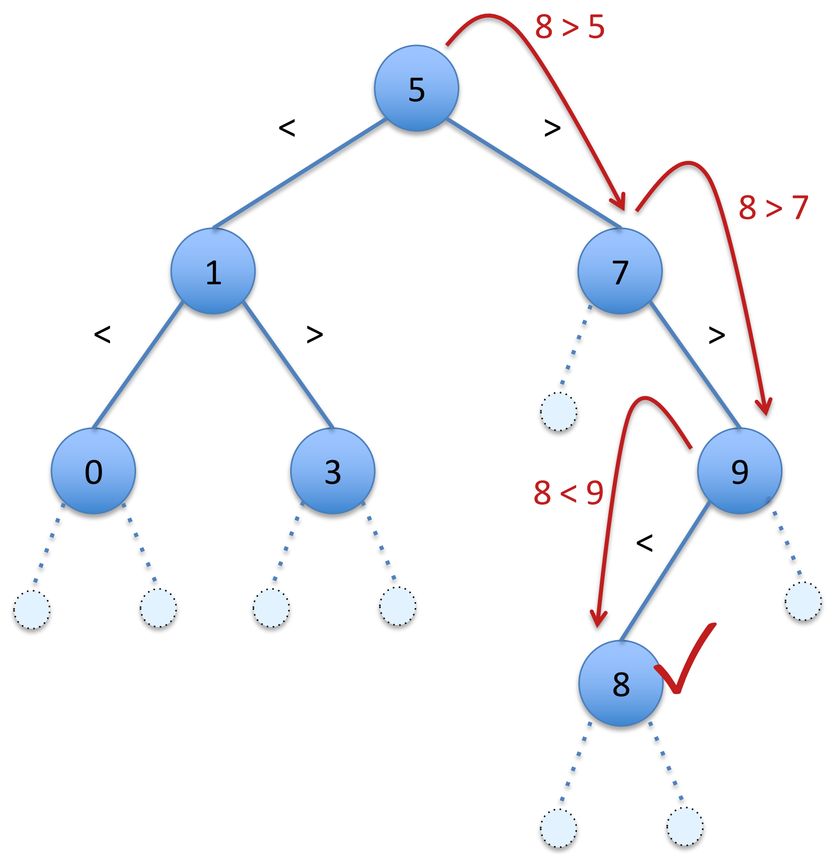
An Example Binary Search Tree



- 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!

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



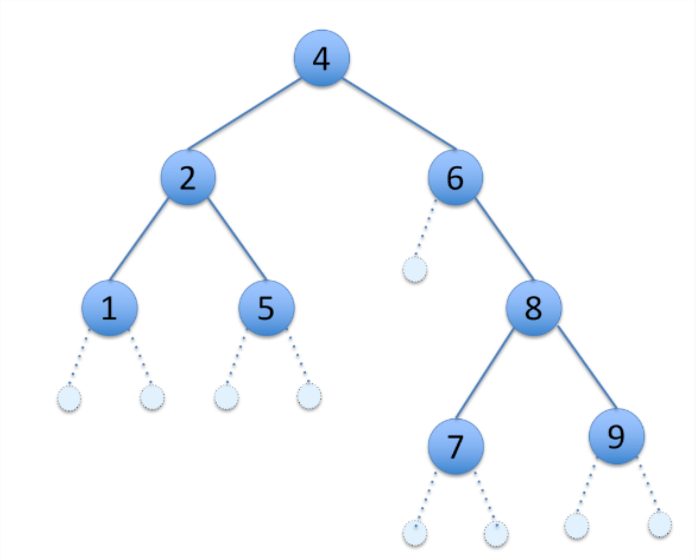
Searching a BST

```
(* 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.

6: Is this a BST?

0



No

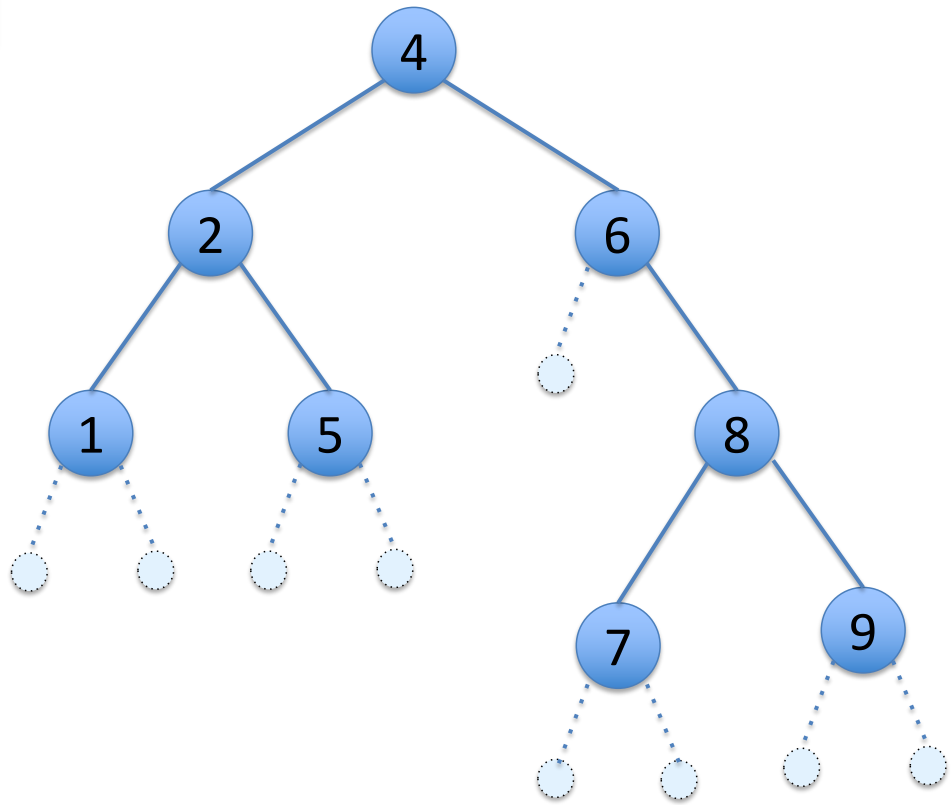
0%

Yes

0%

Is this a BST??

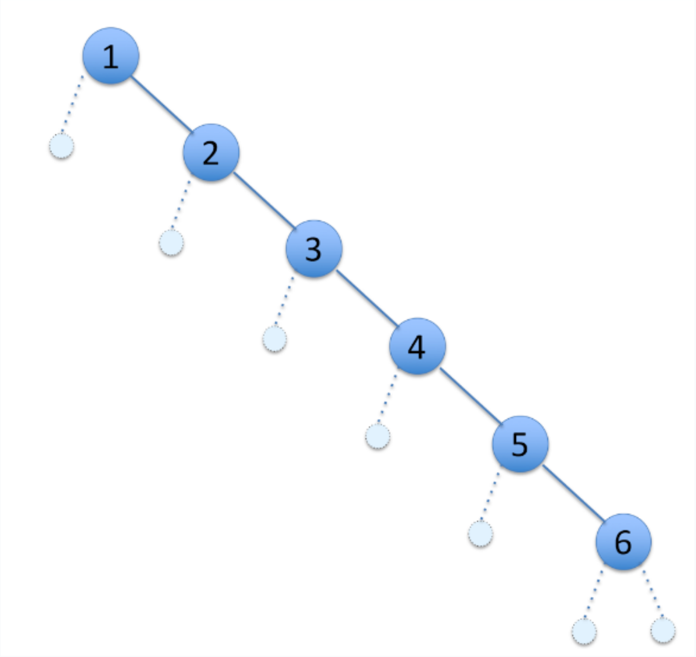
1. yes
2. no



Answer: no, 5 to the left of 4

6: Is this a BST?

0



No

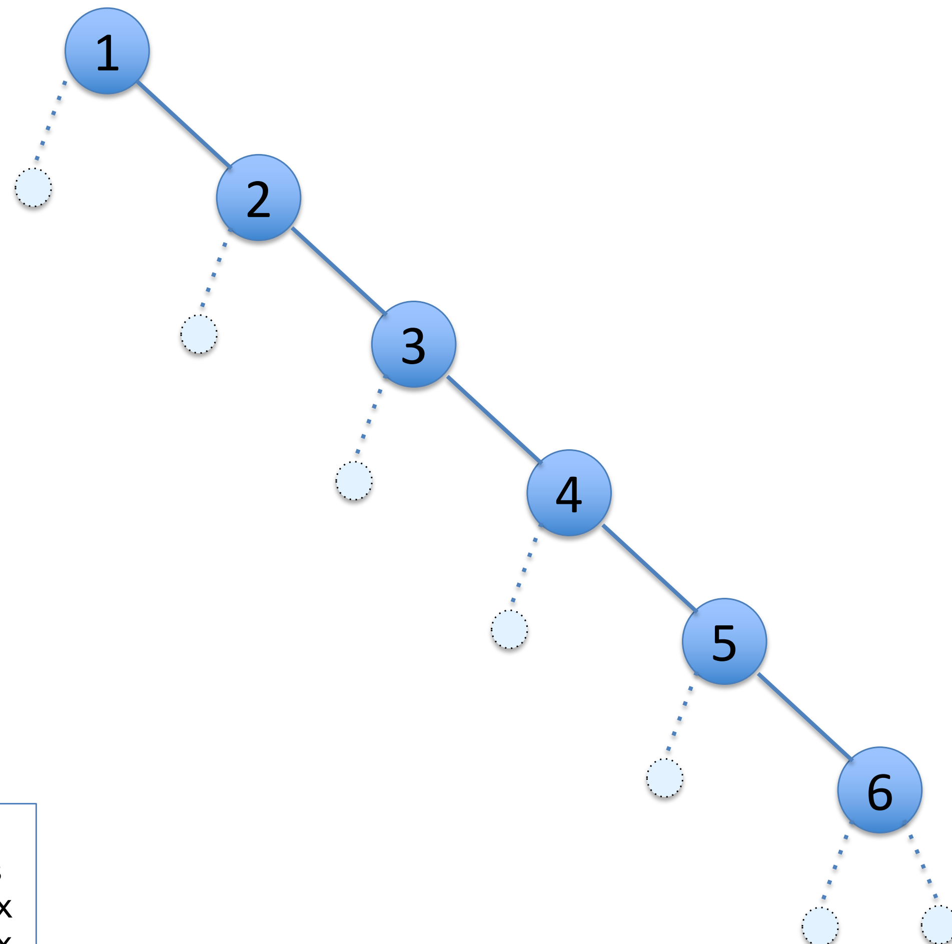
0%

Yes

0%

Is this a BST??

1. yes
2. no

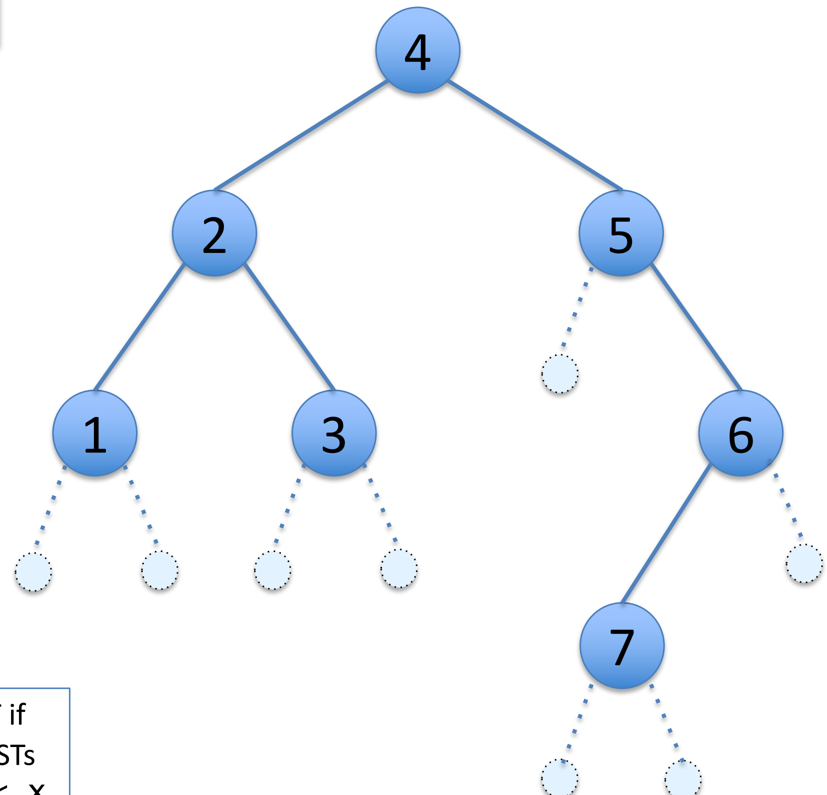


- `Node(lt,x,rt)` is a BST if
  - `lt` and `rt` are both BSTs
  - all nodes of `lt` are `<` `x`
  - all nodes of `rt` are `>` `x`
- `Empty` is a BST

Answer: Yes

Is this a BST??

1. yes
2. no

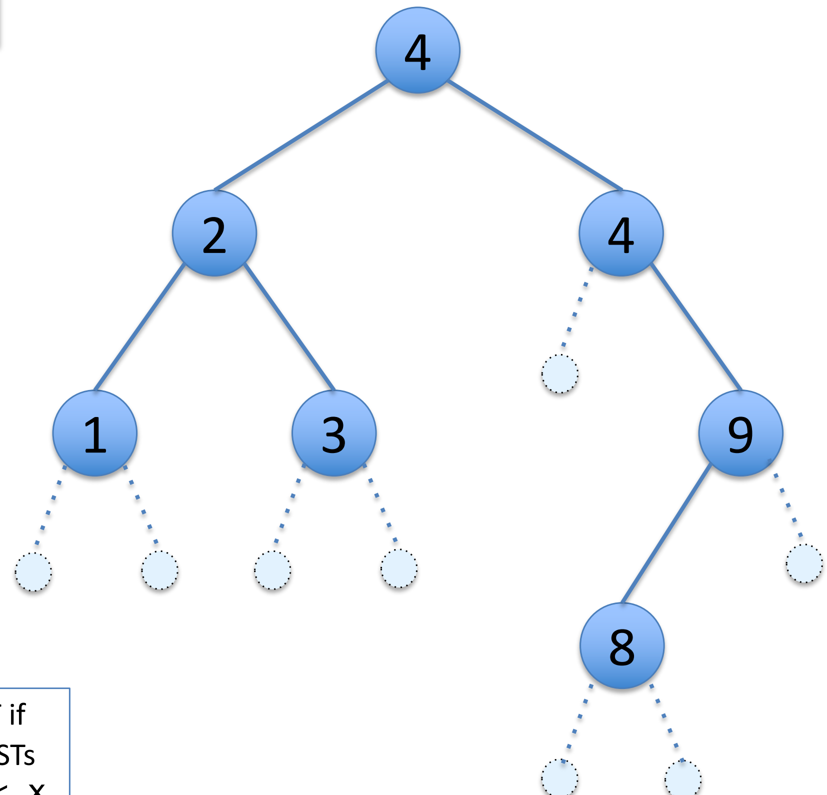


- Node(lt,x,rt) is a BST if
  - lt and rt are both BSTs
  - all nodes of lt are < x
  - all nodes of rt are > x
- Empty is a BST

Answer: no, 7 to the left of 6

Is this a BST??

1. yes
2. no

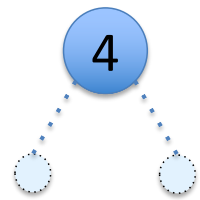


- `Node(lt,x,rt)` is a BST if
  - `lt` and `rt` are both BSTs
  - all nodes of `lt` are < `x`
  - all nodes of `rt` are > `x`
- `Empty` is a BST

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

Manipulating BSTs

Inserting an element

`insert : tree -> int -> tree`

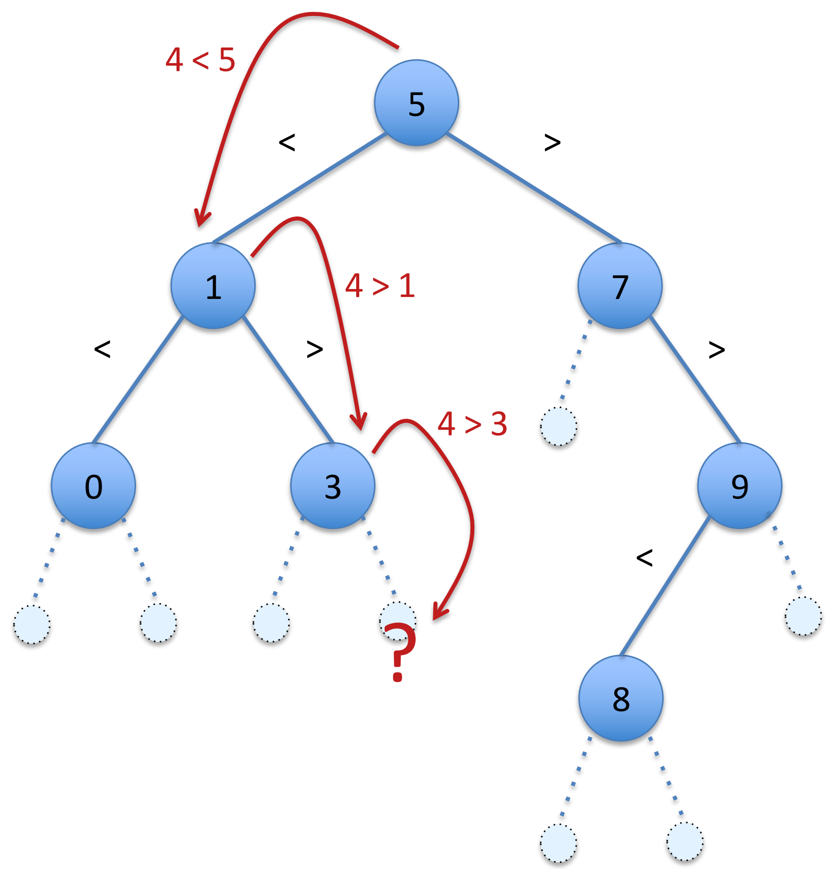
"`insert t x`" returns a new tree containing x and all of the elements of t

Inserting into a BST

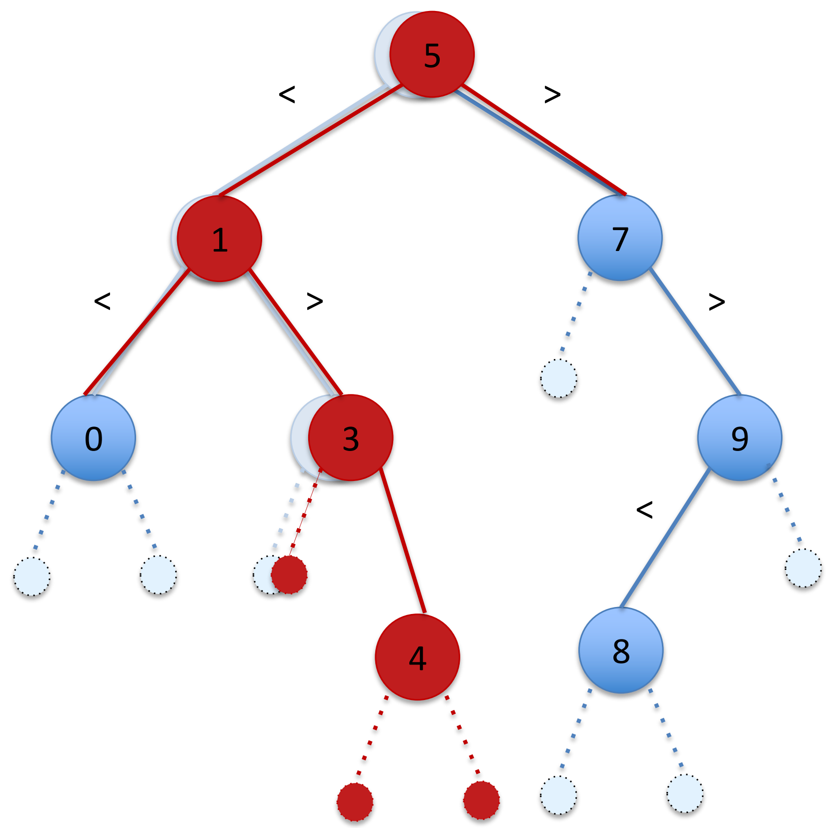
- Challenge: can we make sure that the result of insert really is a BST?
  - i.e., the new element needs to be in the right place!
- Payoff: we can build a BST containing any set of elements
  - Starting with `Empty`, apply insert repeatedly
  - If insert *preserves* the BST invariants, then any tree 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

Critical point!

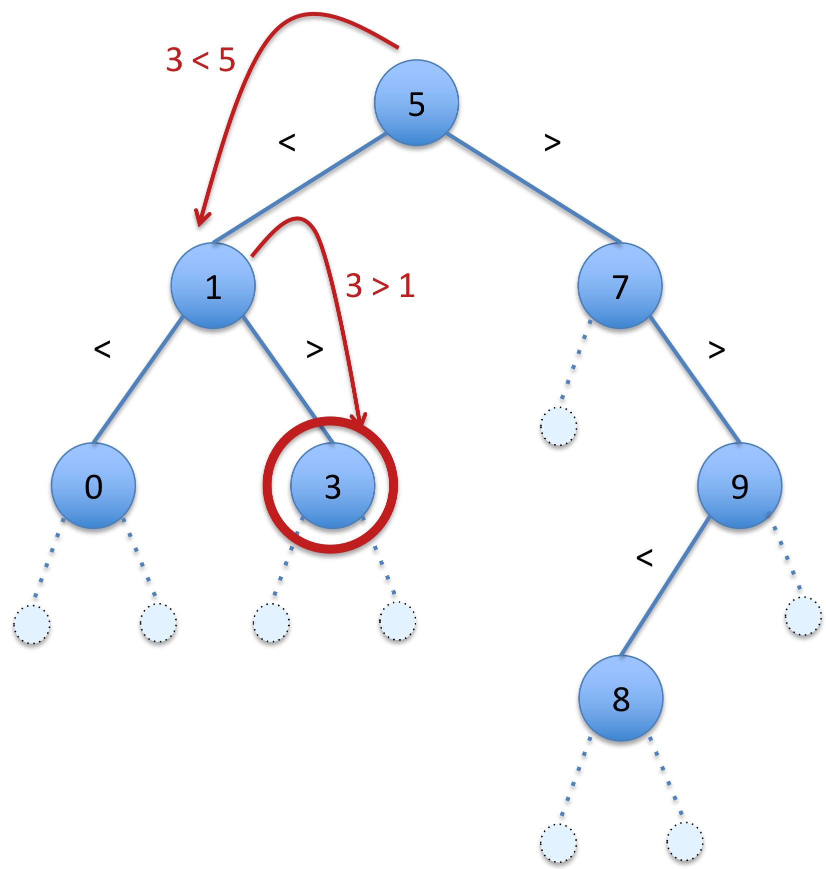
Manipulating BSTs

Deleting an element

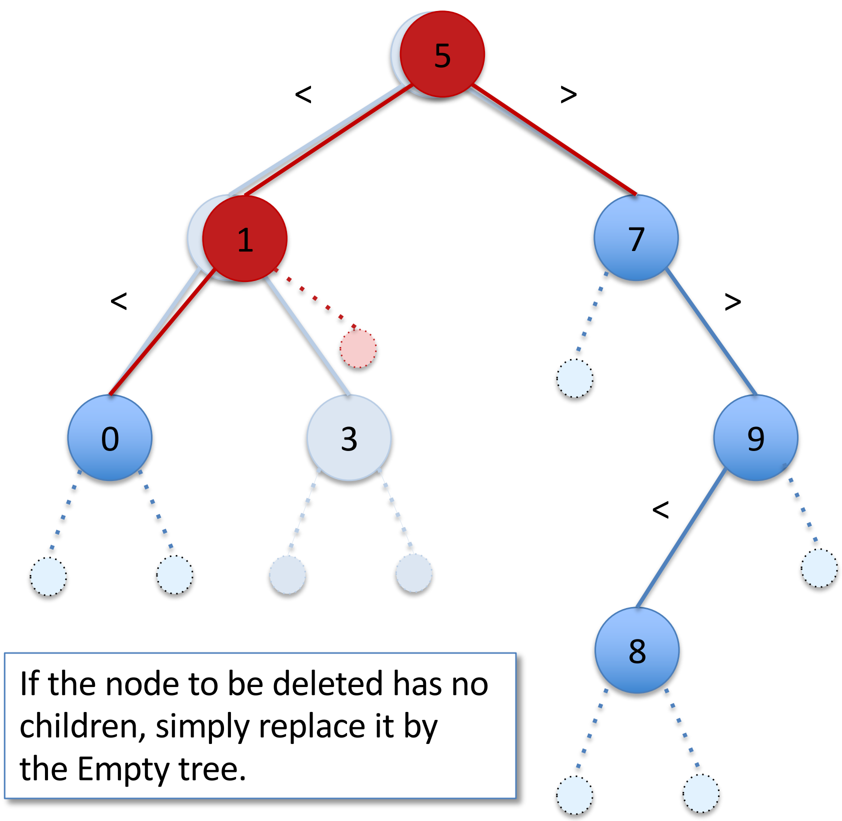
```
delete : tree -> int -> tree
```

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

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

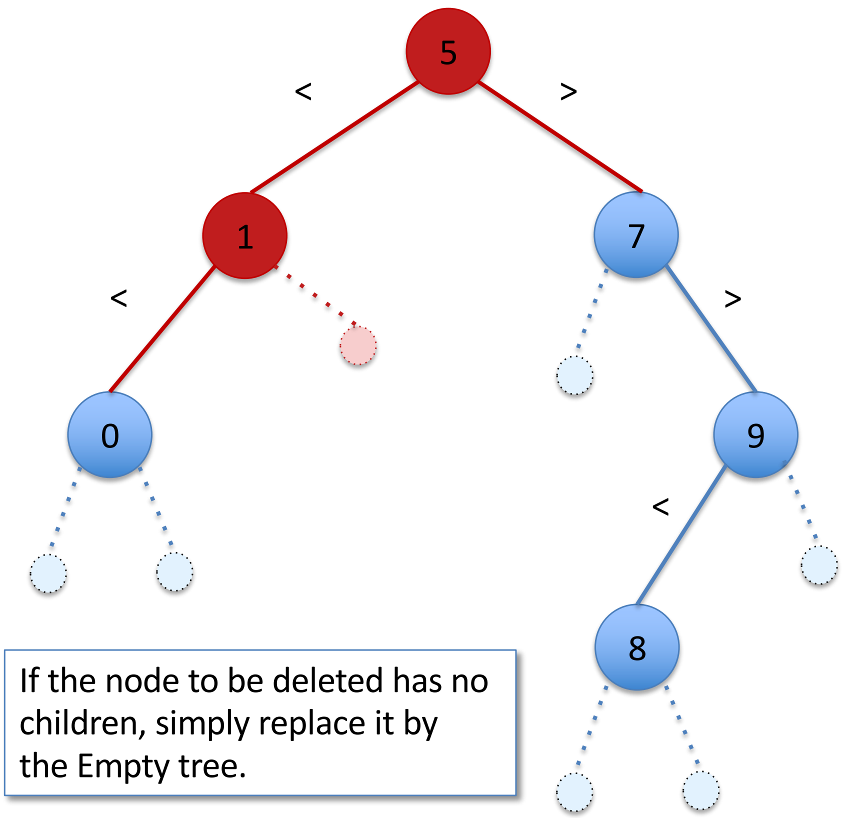


Deletion – No Children: `(delete t 3)`



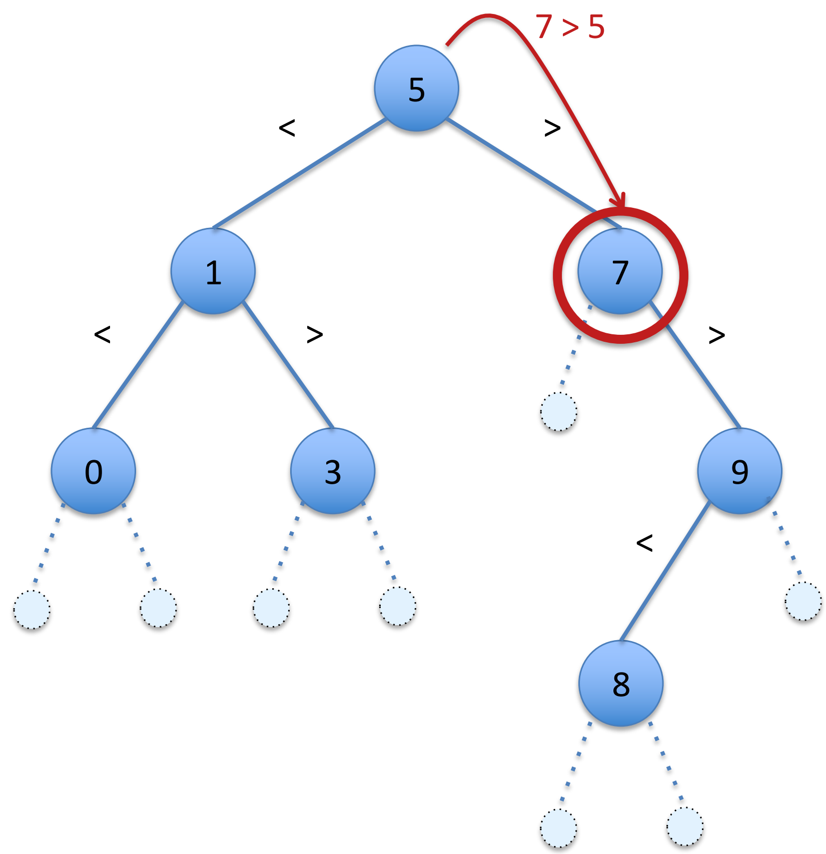
If the node to be deleted has no children, simply replace it by the Empty tree.

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

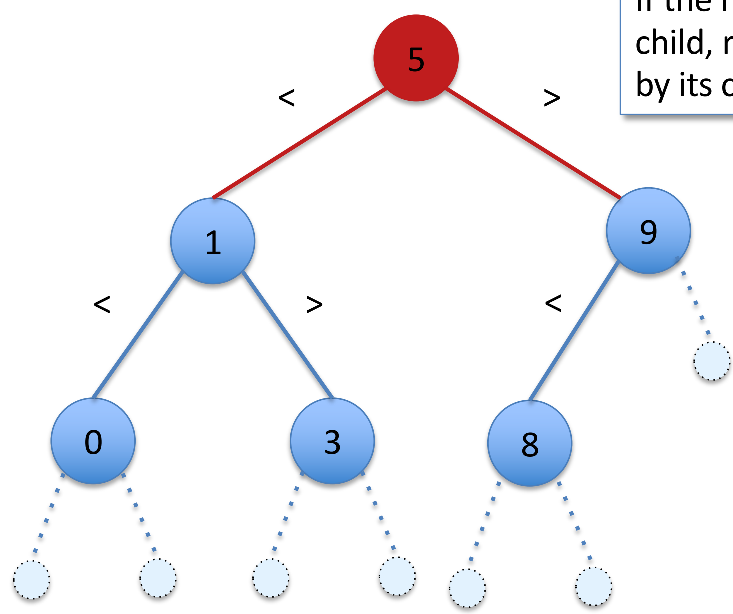


If the node to be deleted has no children, simply replace it by the Empty tree.

Deletion – One Child: `(delete t 7)`

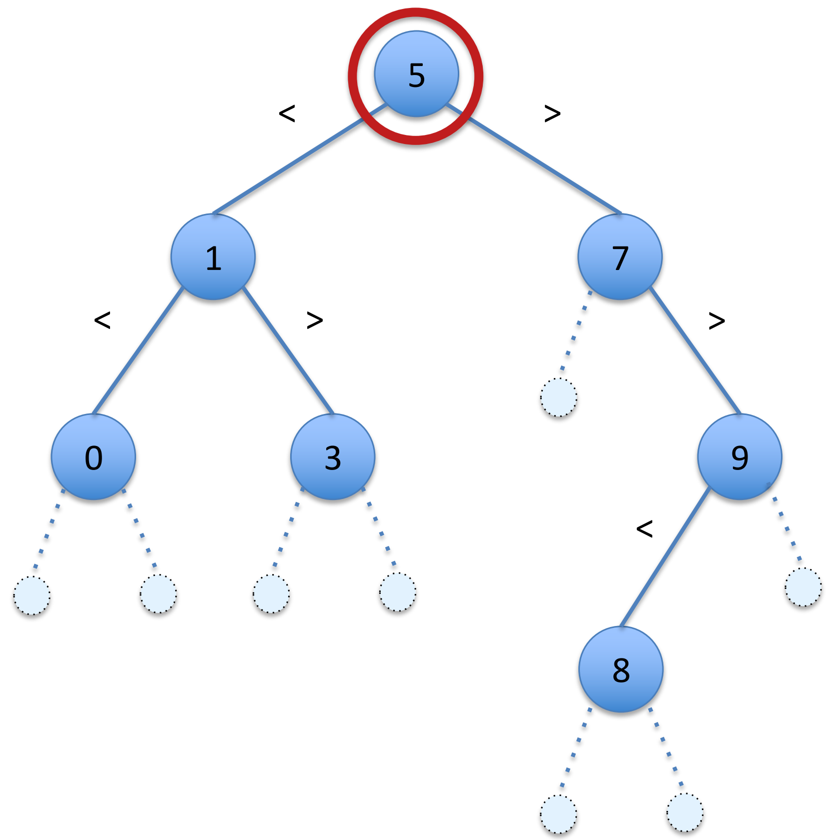


Deletion – One Child: `(delete t 7)`



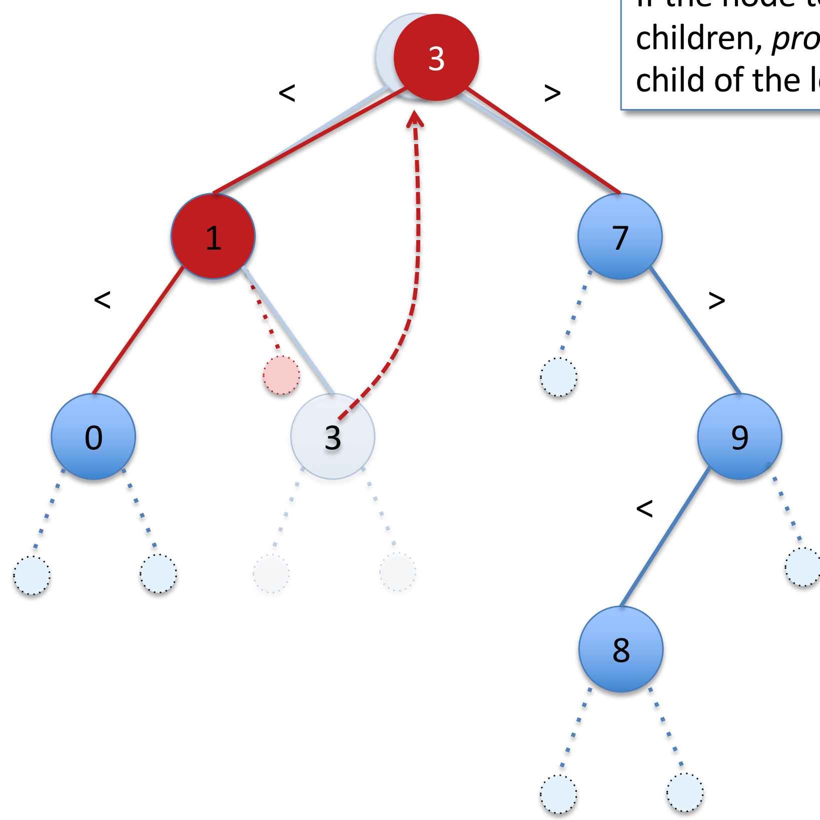
If the node to be delete has one child, replace the deleted node by its child.

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

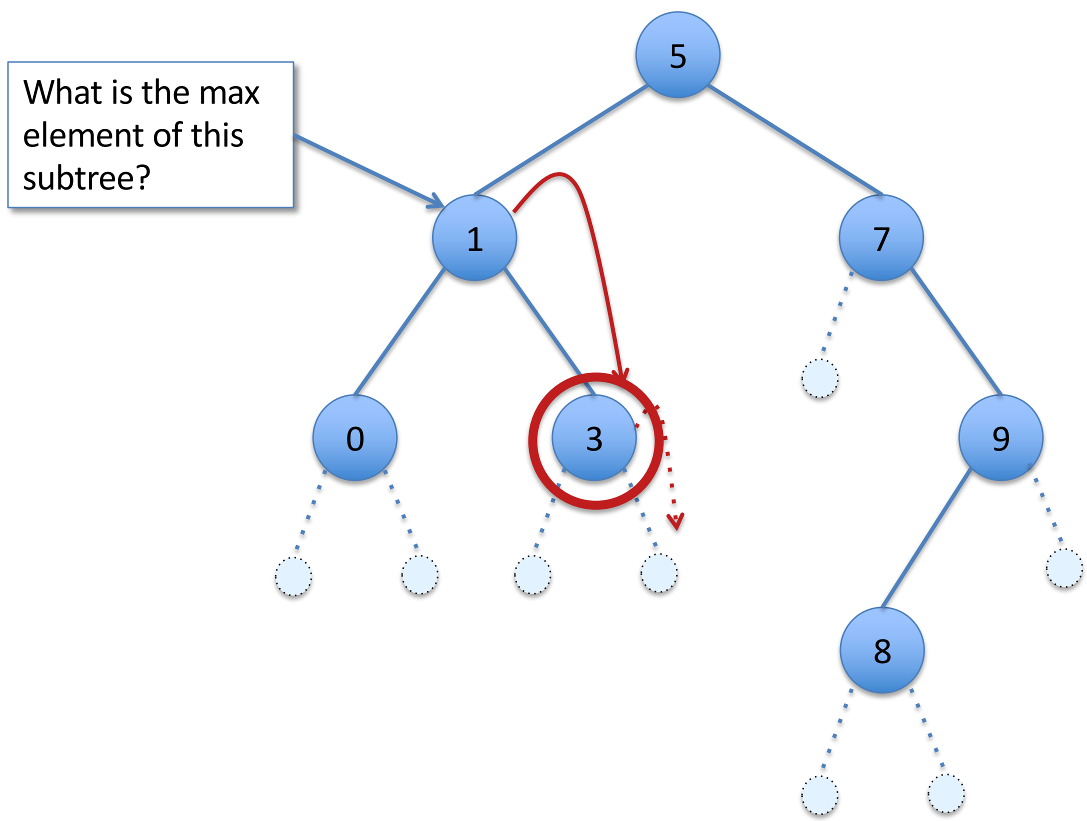


Deletion – Two Children: `(delete t 5)`

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

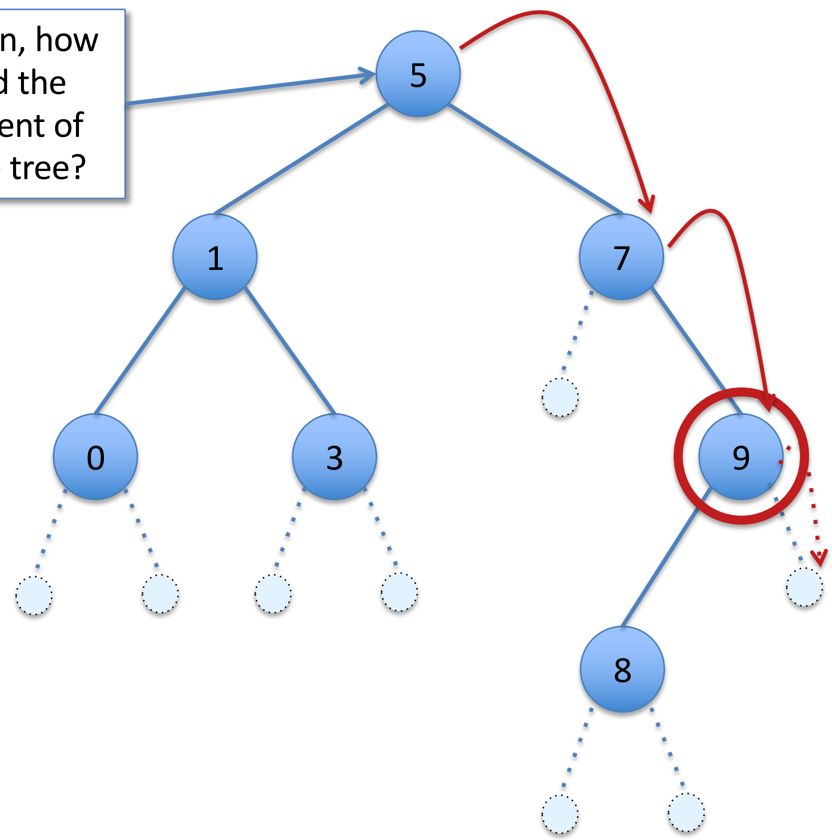


How to Find the Maximum Element?



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%

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%

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

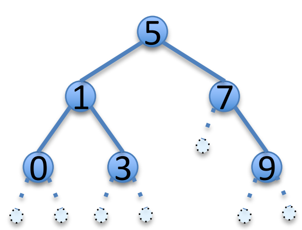
0%

no

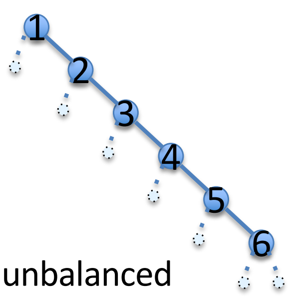
0%

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 work... 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.