# Programming Languages and Techniques (CIS120)

## Lecture 6

## Binary Search Trees

(Lecture notes Chapter 7)

# Announcements

- Homework 2: Computing Human Evolution
  - available now
  - due Tuesday, September 17th
- Reading: Chapter 7
- Please Complete the Entry Survey
  - See the link on Piazza (soon to be posted)

# Recap: Binary Trees

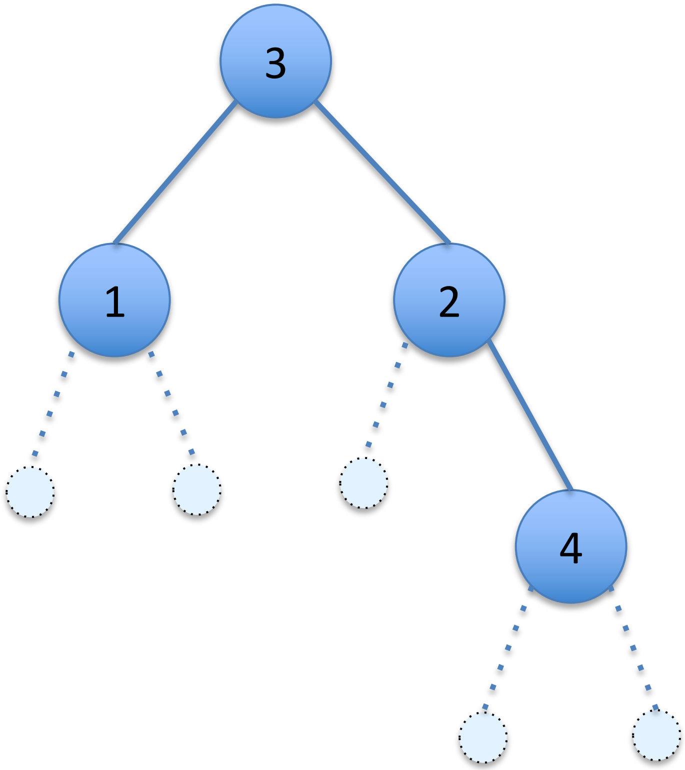
trees with (at most) two branches

# Binary Trees in OCaml

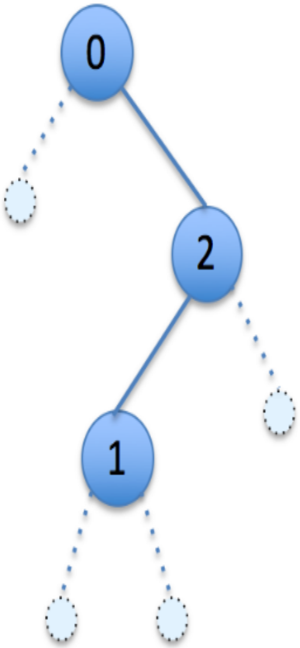
```
type tree =
| Empty
| Node of tree * int * tree
```

```
let t : tree =
  Node (Node (Empty, 1, Empty),
    3,
    Node (Empty, 2,
      Node (Empty, 4, Empty)))
```

=



# What code constructs the pictured tree?



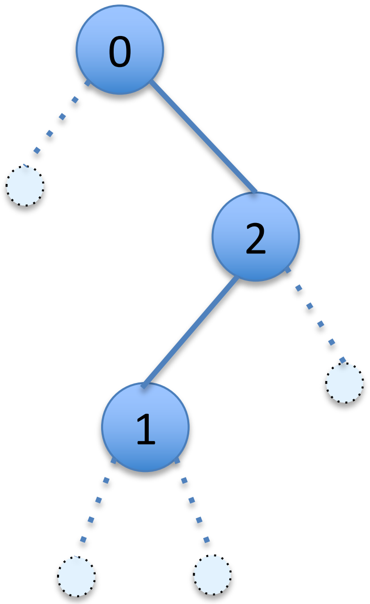
1. let t : tree = Node (Empty, 2, Node (Node (Empty, 0, Empty), 1, Empty))

2. let t : tree = Node (Empty, 2, Node (Empty, 1, Node (Empty, 0, Empty)))

3. let t : tree = Node (Empty, 0, Node (Empty, 2, Node (Empty, 1, Empty)))

4. let t : tree = Node (Empty, 0, Node (Node (Empty, 1, Empty), 2, Empty))

# Which definition constructs the pictured tree?



1

```
let t : tree =
  Node (Empty, 2, Node
        (Node (Empty, 0, Empty), 1, Empty))
```

2

```
let t : tree =
  Node (Empty, 2, Node
        (Empty, 1, Node (Empty, 0, Empty)))
```

3

```
let t : tree =
  Node (Empty, 0, Node
        (Empty, 2, Node (Empty, 1, Empty)))
```

4

```
let t : tree =
  Node (Empty, 0, Node
        (Node (Empty, 1, Empty), 2, Empty))
```

Answer: 4

# Tree Programming Examples

examples: height, size, etc.

See tree.ml and treeExamples.ml

# Trees as Containers

# 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

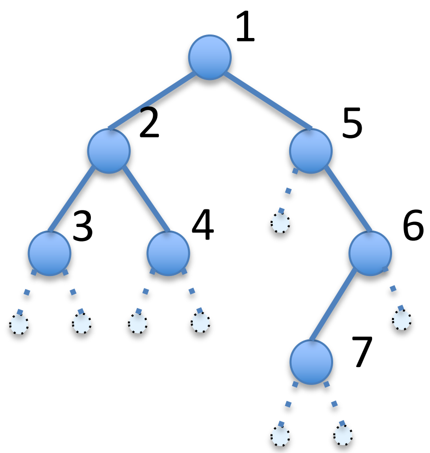
```
type tree =
| Empty
| Node of tree * int * tree
```

# Searching for Data in a Tree

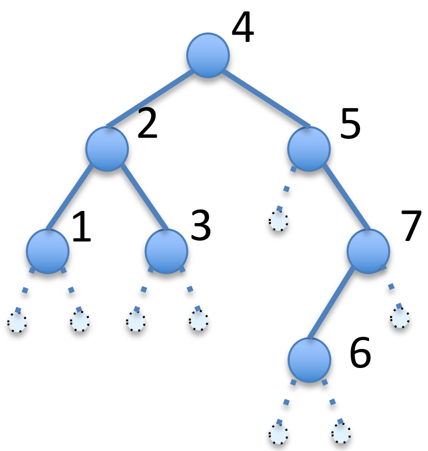
```
let rec contains (t:tree) (n:int) : bool =
  begin match t with
  | Empty -> false
  | Node(lt,x,rt) ->
        x = n
     || contains lt n
     || contains rt n
  end
```

- This function searches through the tree, looking for n
- In the worst case, it might have to traverse the *entire tree*

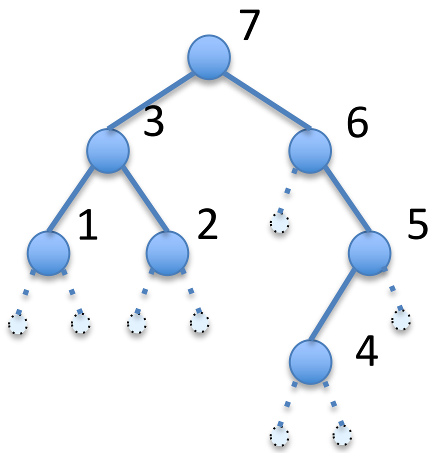
# Recursive Tree Traversals



Pre-Order
Root – Left – Right



In Order
Left – Root – Right

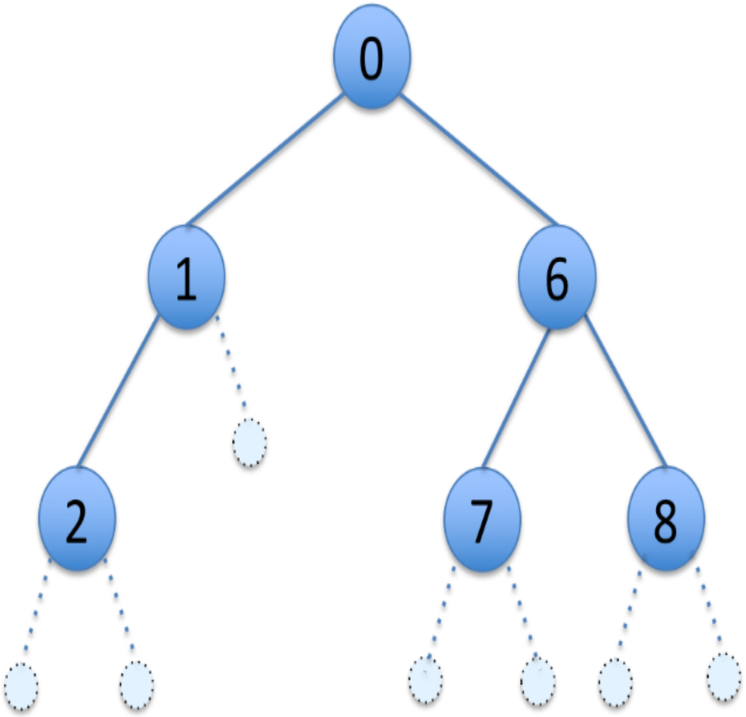


Post-Order
Left – Right – Root

```
(* Code for Pre-Order Traversal *)
let rec f (t:tree) : … =
  begin match t with
  | Empty -> …
  | Node(l, x, r) ->
    let root = … x … in (* process root *)
    let left =  f l  in (* recursively process left subtree *)
    let right = f r  in (* recursively process right subtree *)
    combine root left right
  end
```

Other traversals
vary the order
in which these
are computed…

# In what sequence will the nodes of this tree be visited by a post-order traversal?



Post-Order
Left – Right – Root

[0;1;6;2;7;8]

[0;1;2;6;7;8]

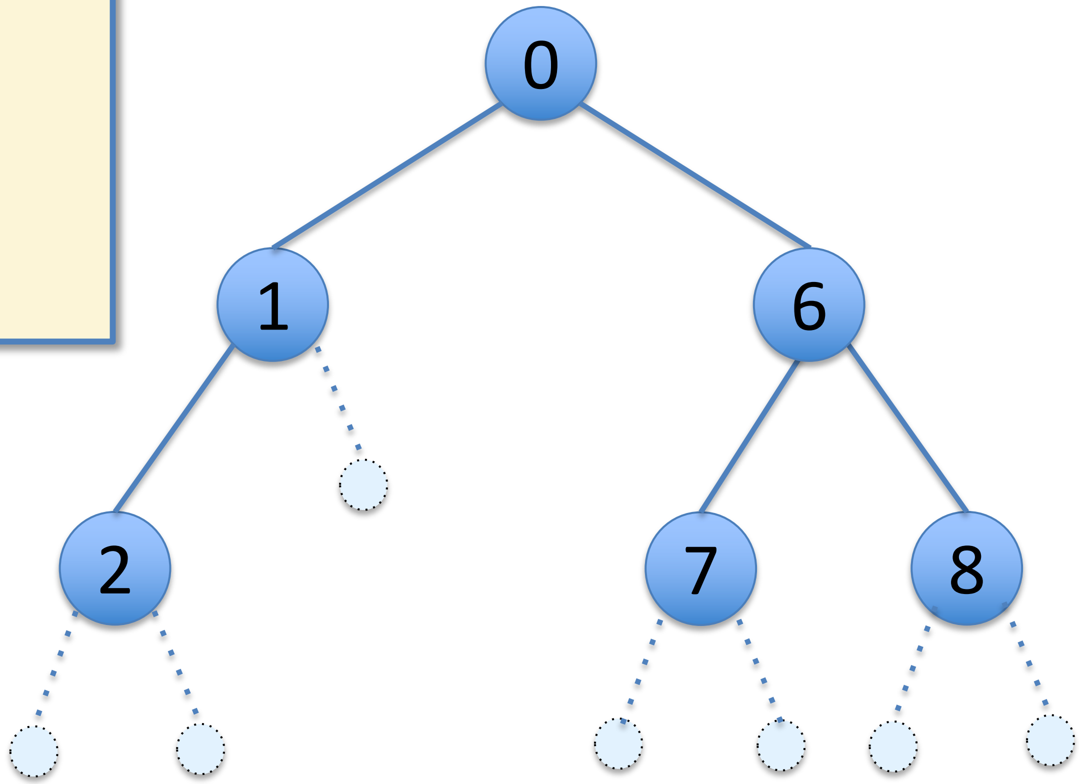
[2;1;0;7;6;8]

[7;8;6;2;1;0]

[2;1;7;8;6;0]

In what sequence will the nodes of this tree be visited by a post-order traversal?

1. [0;1;6;2;7;8]
2. [0;1;2;6;7;8]
3. [2;1;0;7;6;8]
4. [7;8;6;2;1;0]
5. [2;1;7;8;6;0]

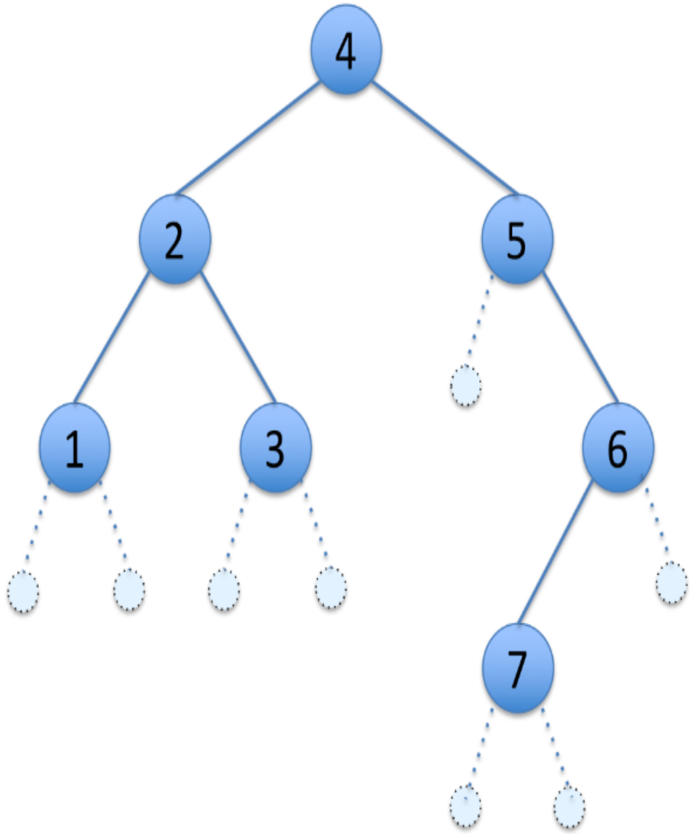


Post-Order
Left – Right – Root

Answer: 5

# What is the result of applying this function on this tree?

```
let rec inorder (t:tree) : int list =
  begin match t with
        | Empty -> []
        | Node (left, x, right) ->
               inorder left @ (x :: inorder 
right)
end
```



[]

[1;2;3;4;5;6;7]

[1;2;3;4;5;7;6]

[4;2;1;3;5;6;7]

[4]

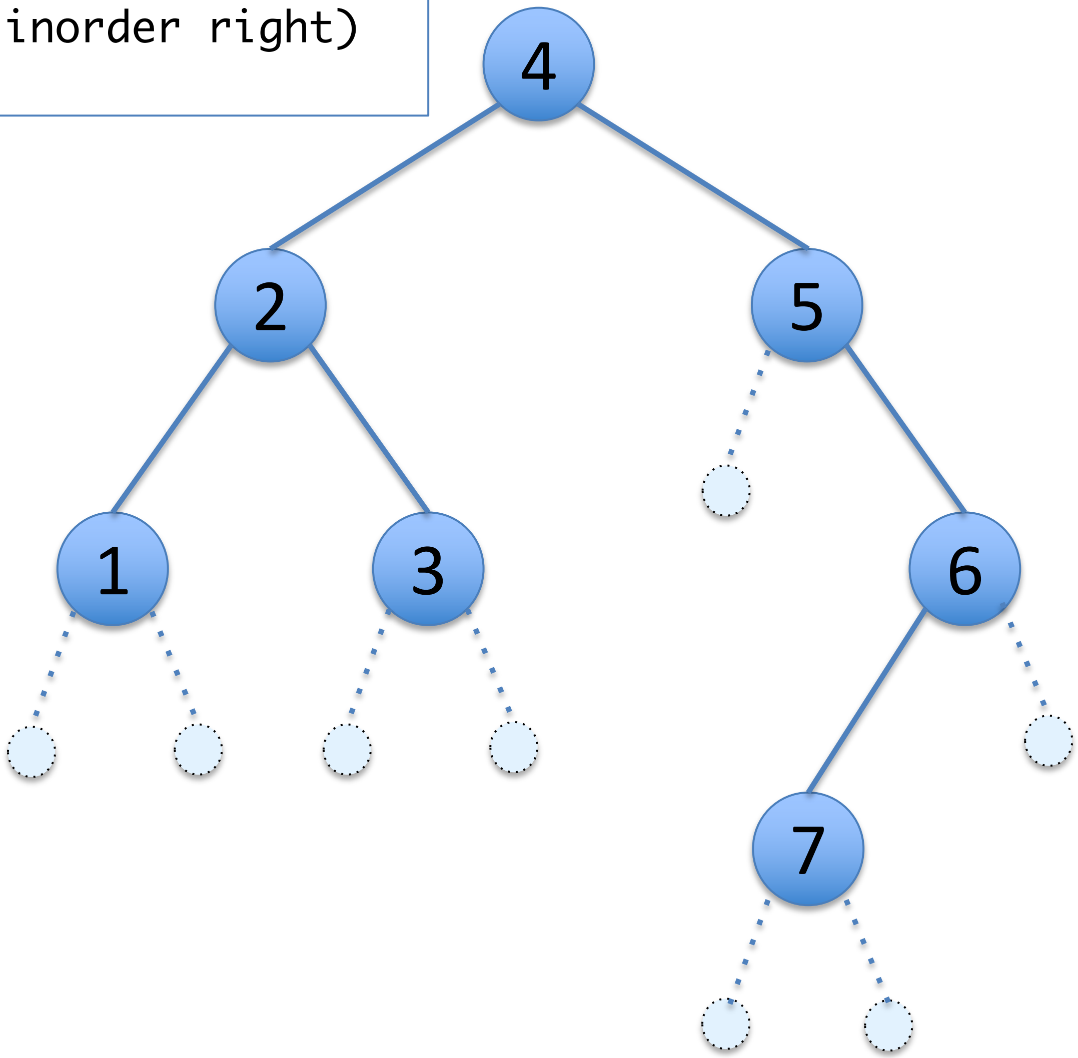
[1;1;1;1;1;1;1]

none of the above

```
let rec inorder (t:tree) : int list =
  begin match t with
    | Empty -> []
    | Node (left, x, right) ->
        inorder left @ (x :: inorder right)
end
```

What is the result of applying this function on this tree?

1. []
2. [1;2;3;4;5;6;7]
3. [1;2;3;4;5;7;6]
4. [4;2;1;3;5;6;7]
5. [4]
6. [1;1;1;1;1;1;1]
7. none of the above



Answer: 3

# 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
- But 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**:

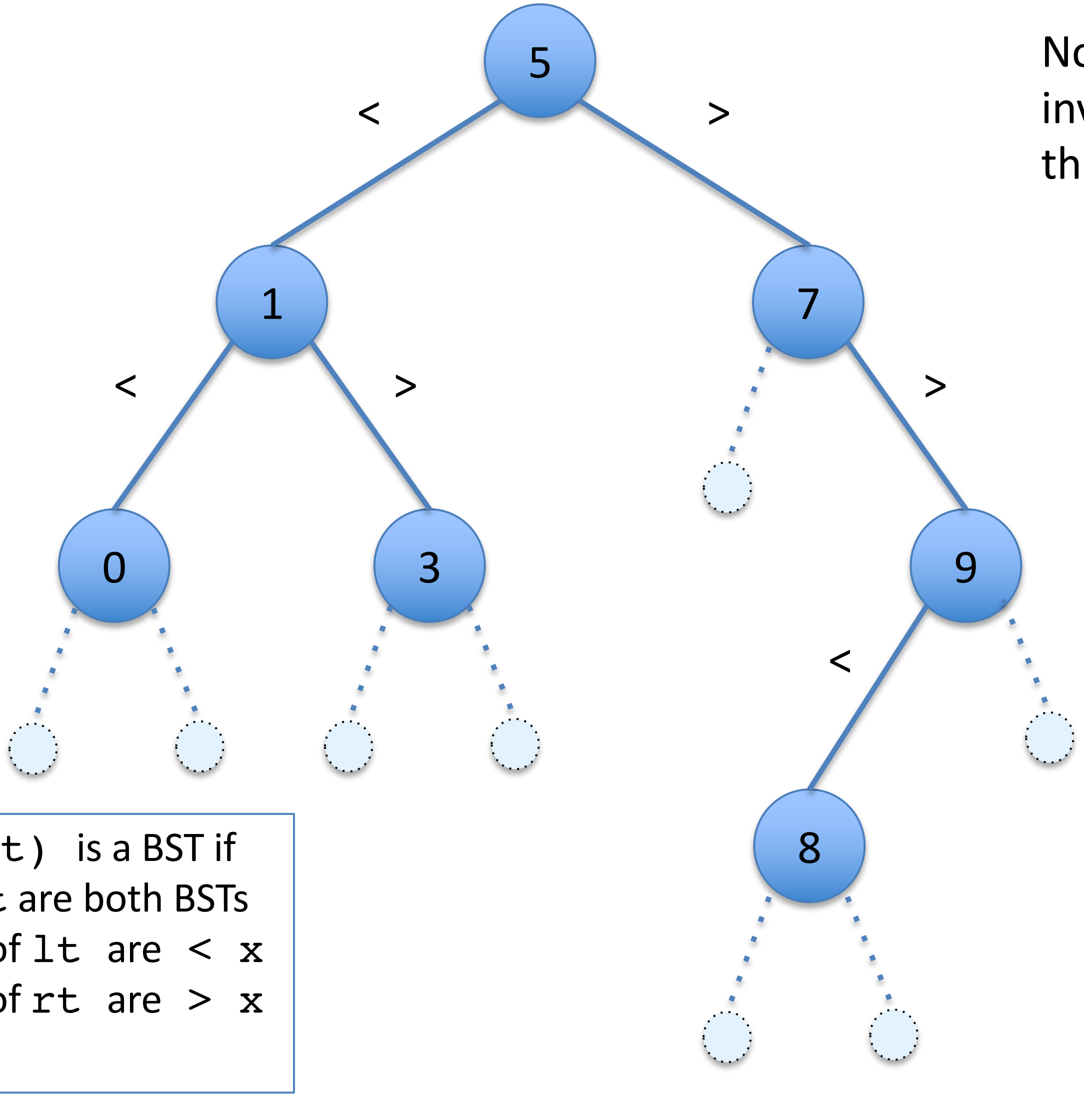
  - `Node(lt,x,rt)` is a BST if
    - `lt` and `rt` are both BSTs
    - all nodes of `lt` are `< x`
    - all nodes of `rt` are `> x`
  - `Empty` is a BST

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

---

*An 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.

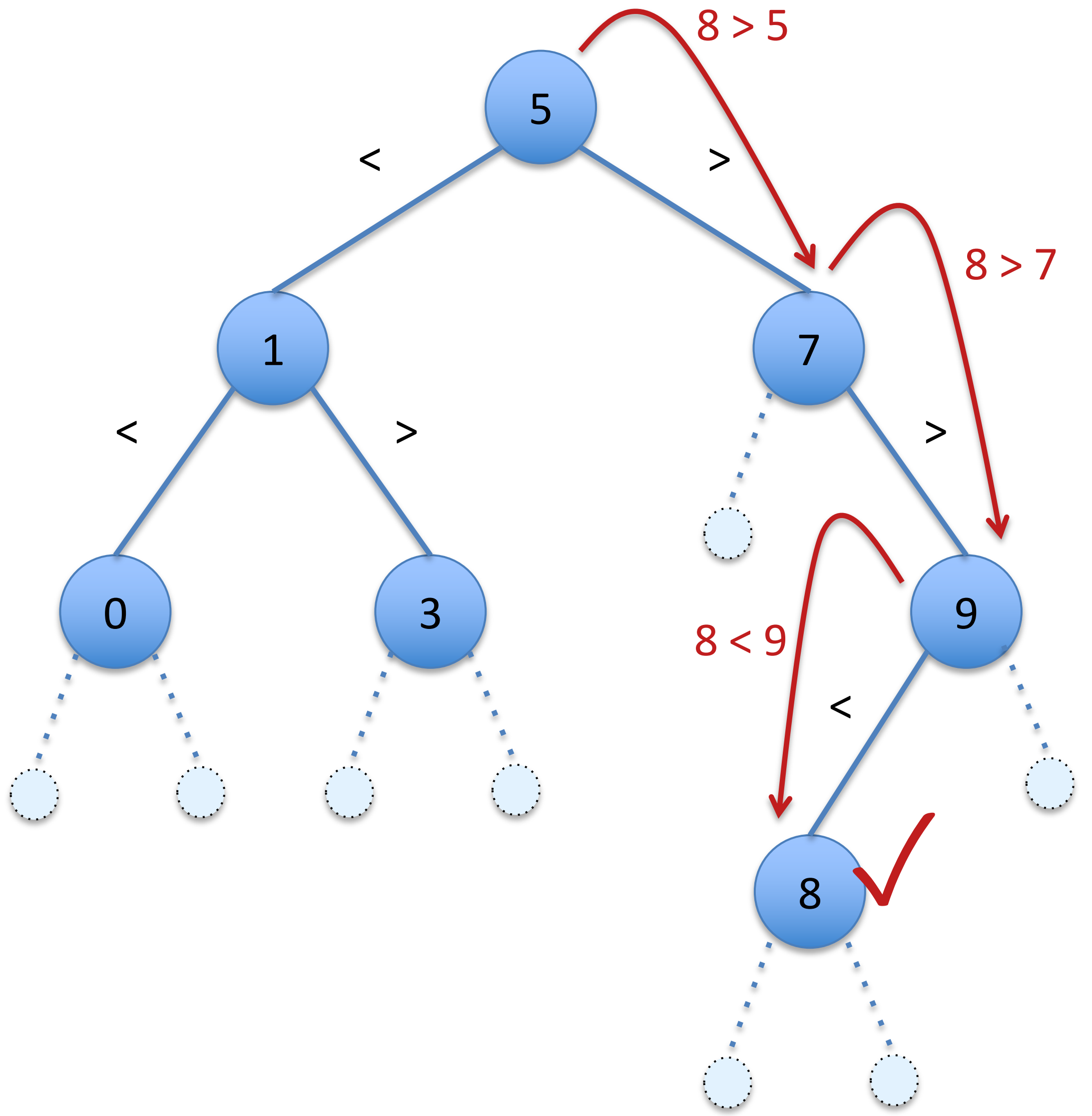
# An Example Binary Search Tree



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

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



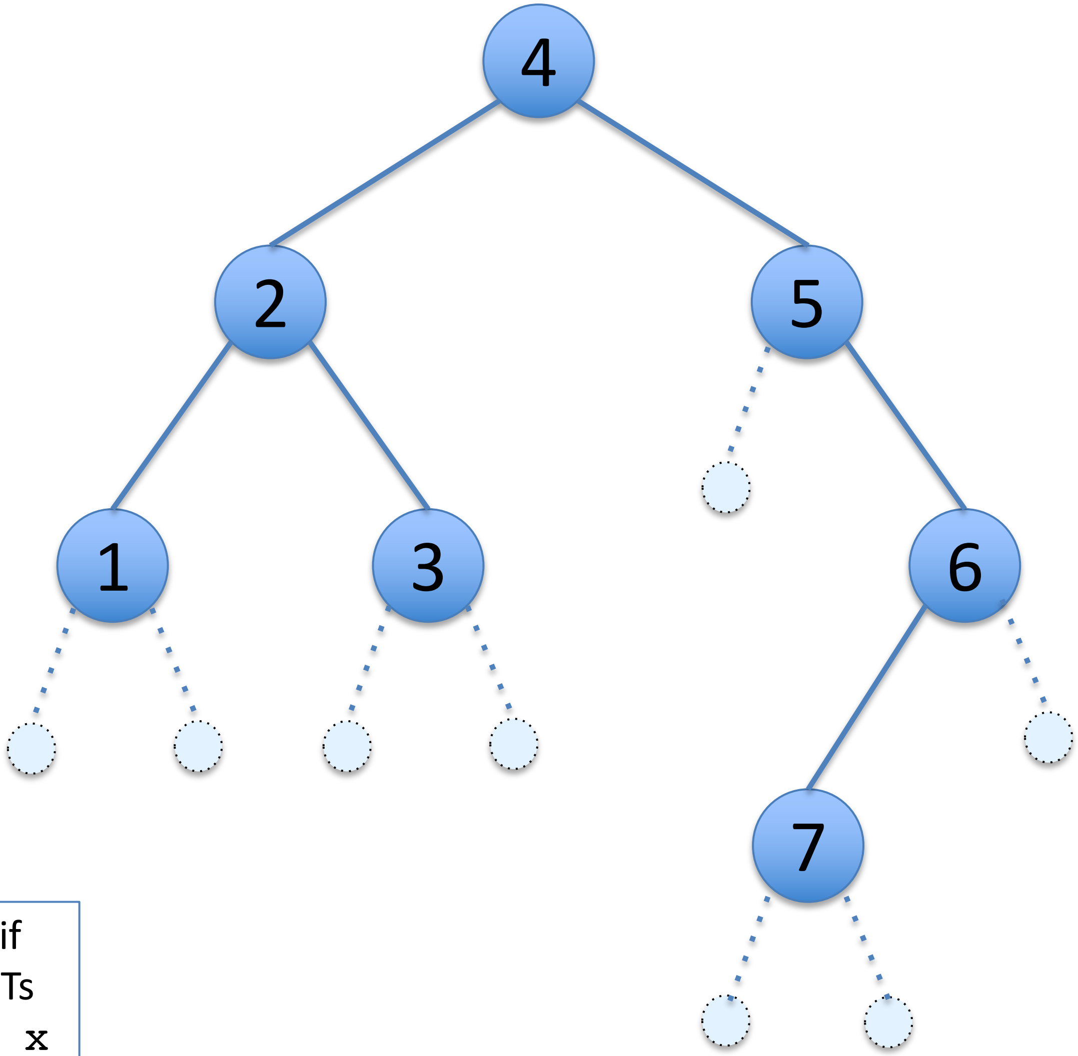
# Searching a BST

```
(* 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.

Is this a BST??

1. yes
2. no

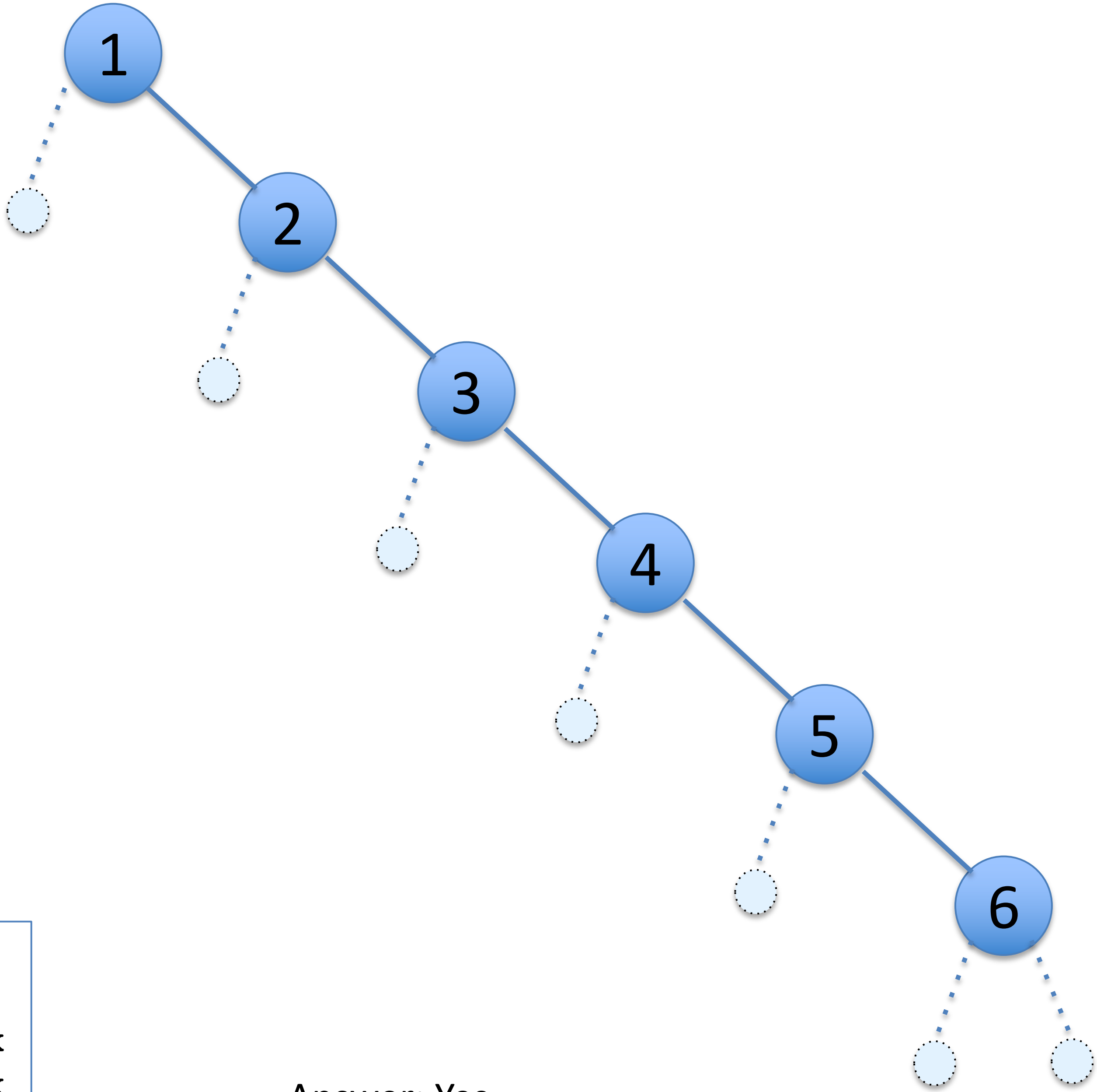


- `Node(lt,x,rt)` is a BST if
  - `lt` and `rt` are both BSTs
  - all nodes of `lt` are `<` `x`
  - all nodes of `rt` are `>` `x`
- `Empty` is a BST

Answer: no, 7 to the left of 6

Is this a BST??

1. yes
2. no

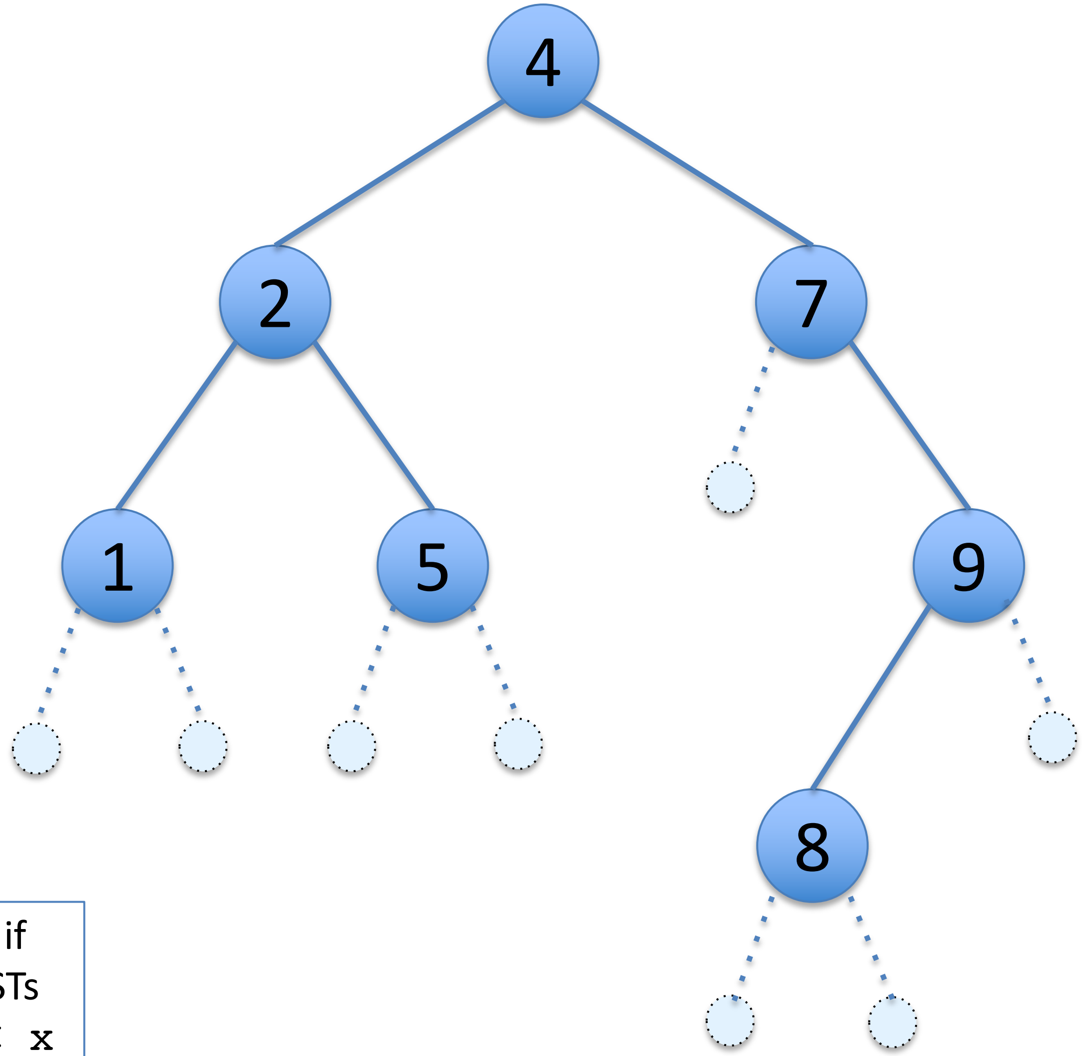


- `Node(lt,x,rt)` is a BST if
  - `lt` and `rt` are both BSTs
  - all nodes of `lt` are < `x`
  - all nodes of `rt` are > `x`
- `Empty` is a BST

Answer: Yes

Is this a BST??

1. yes
2. no

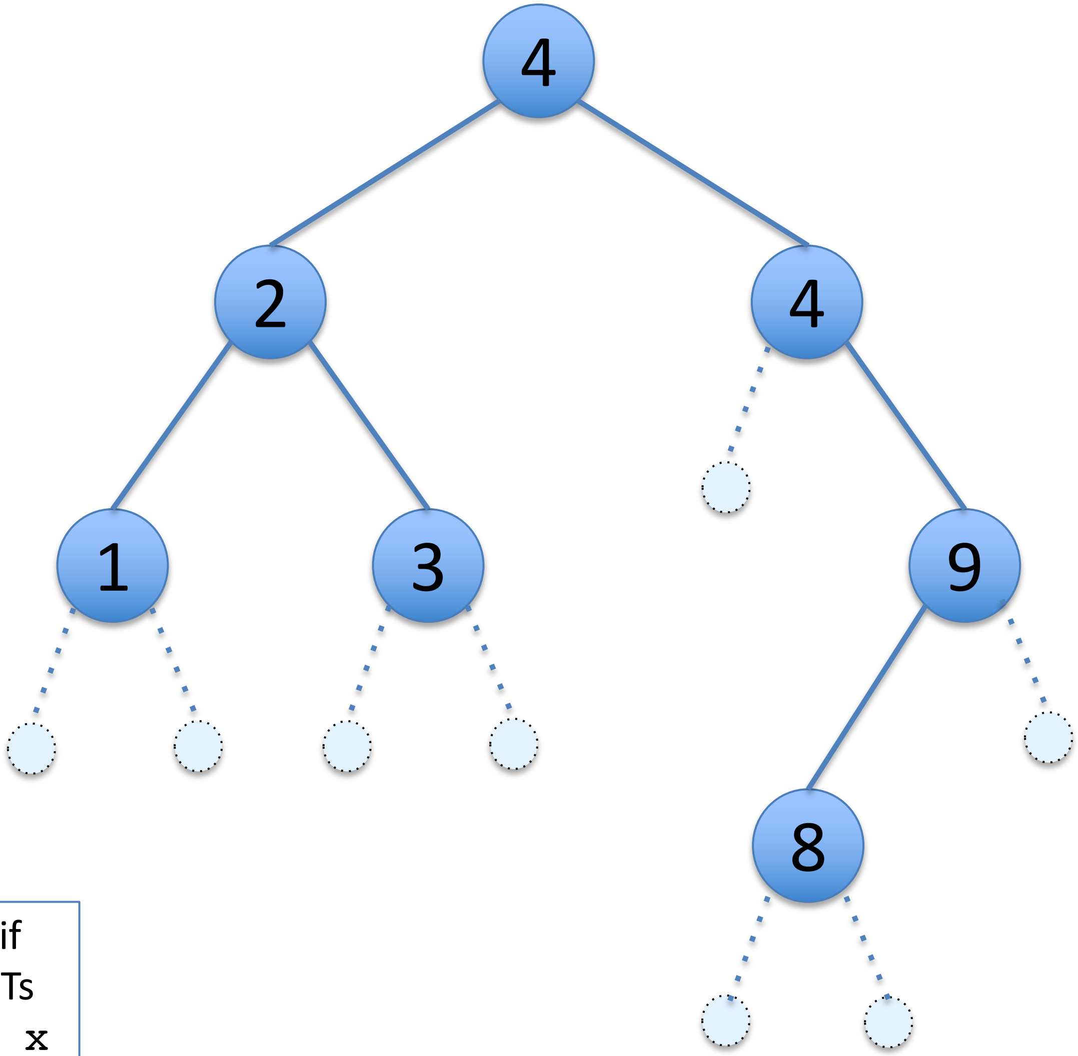


- `Node(lt,x,rt)` is a BST if
  - `lt` and `rt` are both BSTs
  - all nodes of `lt` are `<` `x`
  - all nodes of `rt` are `>` `x`
- `Empty` is a BST

Answer: no, 5 to the left of 4

Is this a BST??

1. yes
2. no

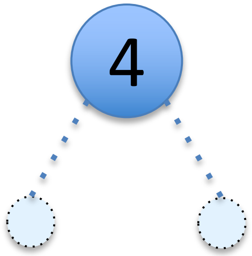


- `Node(lt,x,rt)` is a BST if
  - `lt` and `rt` are both BSTs
  - all nodes of `lt` are `<` `x`
  - all nodes of `rt` are `>` `x`
- `Empty` is a BST

Answer: no, 4 to the right of 4

Is this a BST??

1. yes
2. no

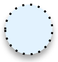


- `Node(lt,x,rt)` is a BST if
    - `lt` and `rt` are both BSTs
    - all nodes of `lt` are `<` `x`
    - all nodes of `rt` are `>` `x`
- `Empty` is a BST

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

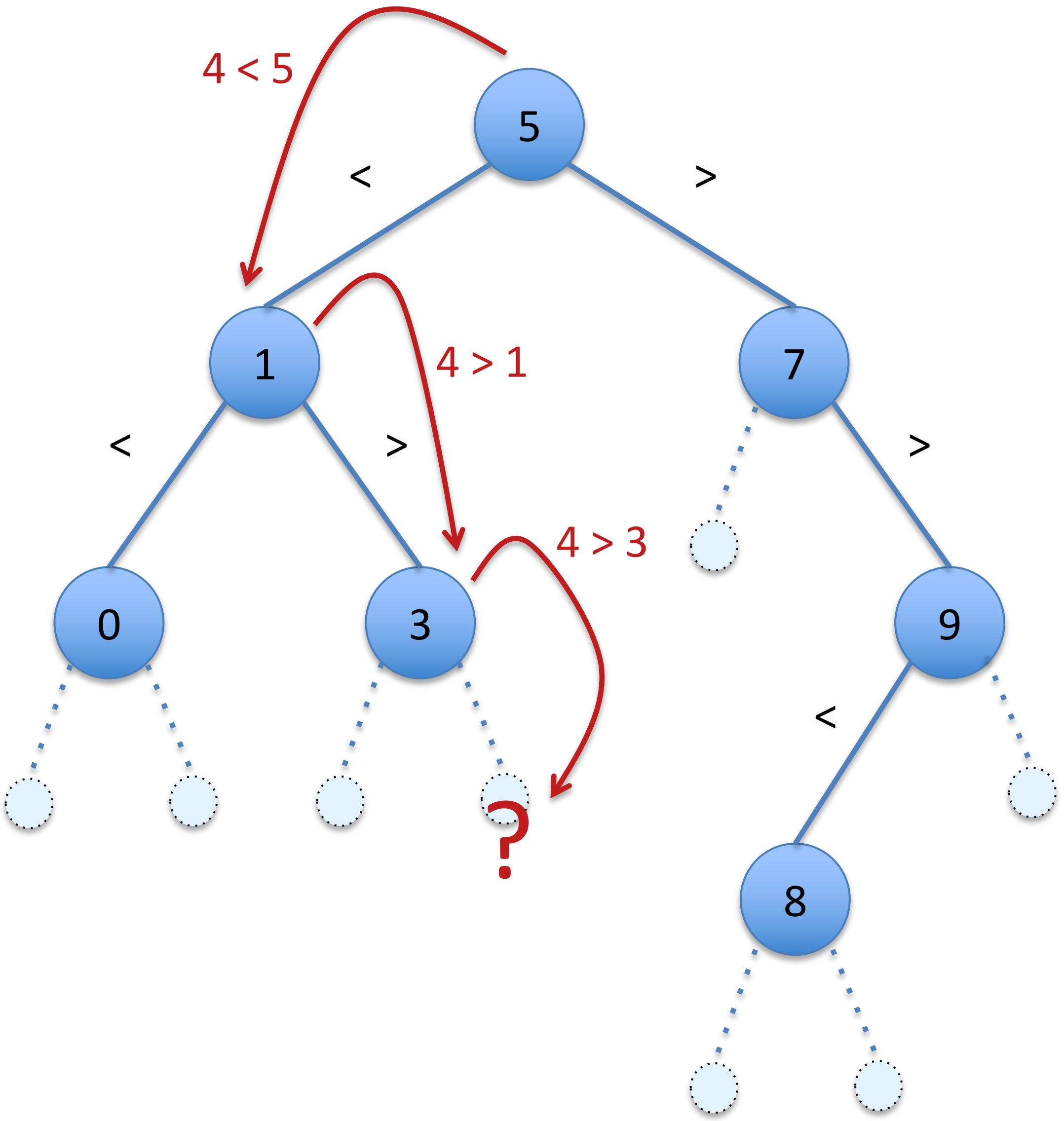
# Inserting into a BST

- Suppose we have a BST `t` and a new element `n`, and we wish to compute a new BST `t'` containing all the elements of `t` together with `n`
  - Need to make sure the tree we build is really a BST – i.e., make sure to put n in the right place!

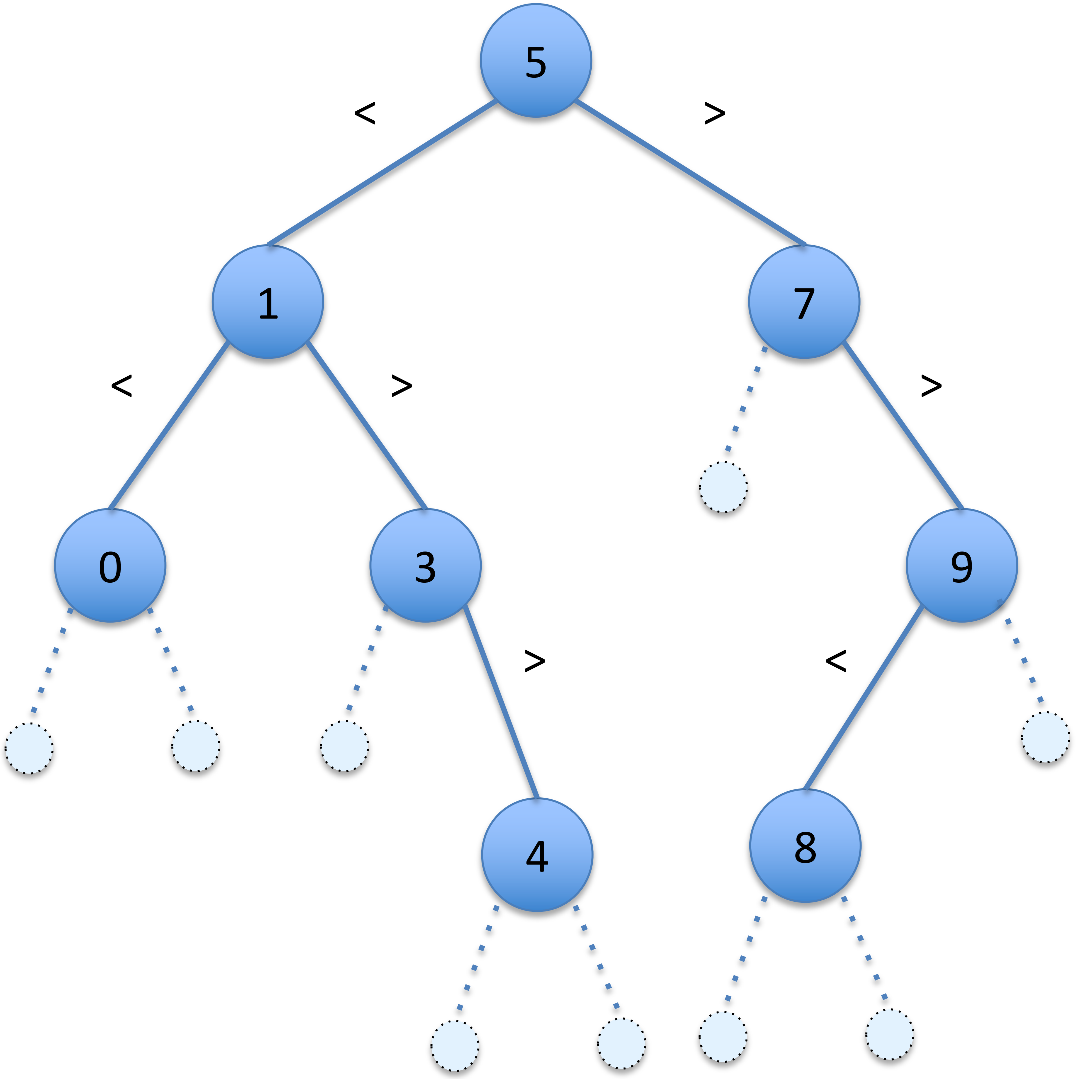
- This gives us a way to build up a BST containing any set of elements we like:
  - Starting from the `Empty` BST, apply this function repeatedly to get the BST we want
  - If insertion *preserves* the BST invariants, then any tree we get from it will be a BST *by construction*
    - No need to check!
  - Later: we can also "rebalance" the tree to make lookup efficient (NOT in CIS 120; see CIS 121)

*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 the similarity to searching the tree.
- Assuming that t is a BST, the result is also a BST. (Why?)
- Note that the result is a *new* tree with (possibly) one more Node; the original tree is unchanged

Critical point!