Searching Sorted Data

Searching for a Number

- Consider the following sorted array

0	1	2	3	4	5	6	7	8	9
-2	0	4	7	12	19	22	42	65	

- When searching for a number x using binary search, we **always** start by looking at the midpoint, index 4

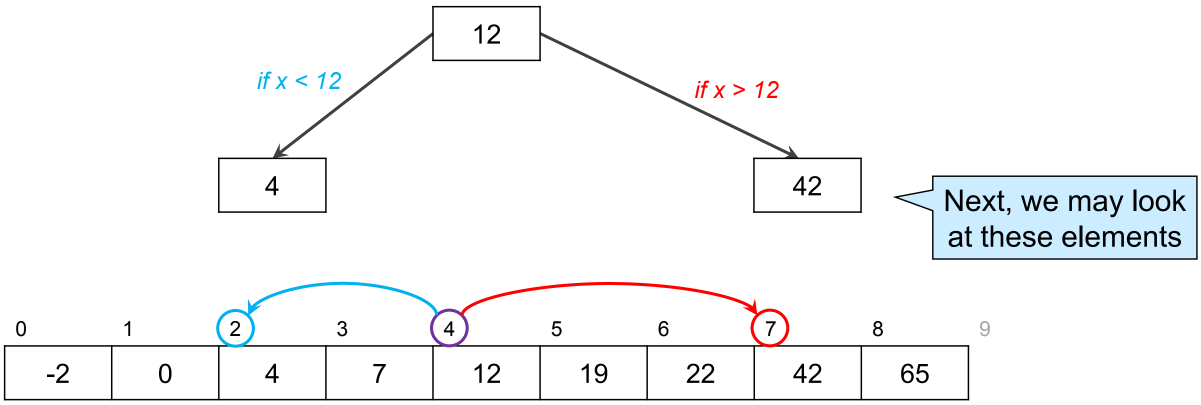
12

We always look at this element

- Then, 3 things can happen
  - $x = 12$ (and we are done)
  - $x < 12$
  - $x > 12$

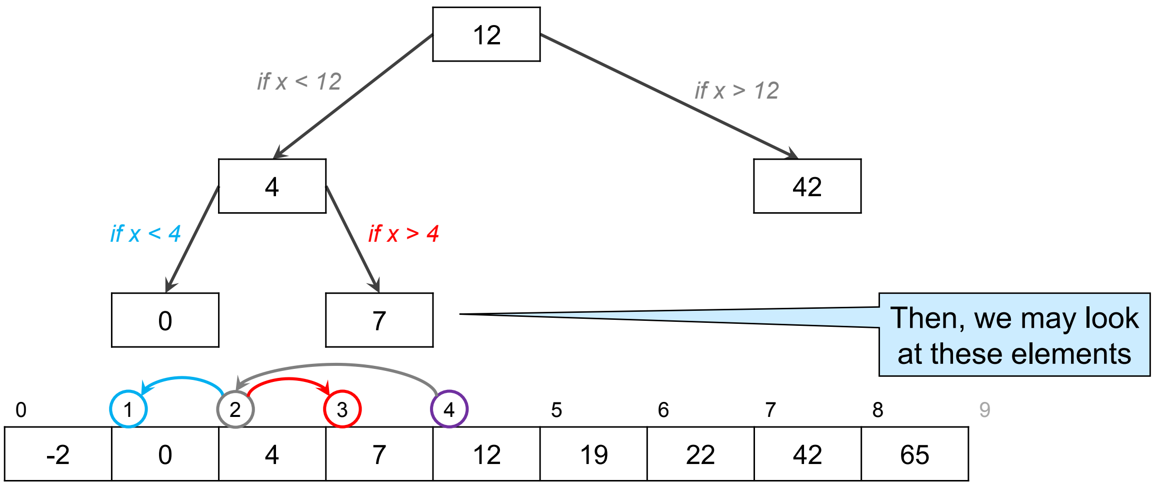
Searching for a Number

- If x < 12, the next index we look at is **necessarily** 2
- If x > 12, the next index we look at is **necessarily** 7



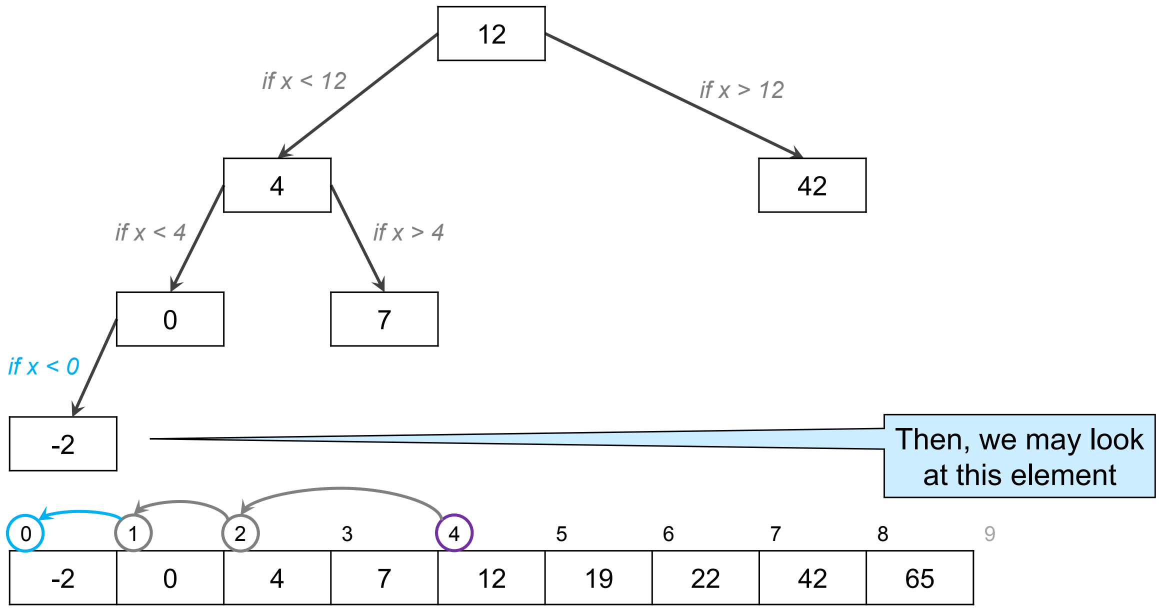
Searching for a Number

- Assume x < 12, so we look at 4
  - if x = 4, we are done
  - if x < 4, we **necessarily** look at 0
  - if x > 4, we **necessarily** look at 7



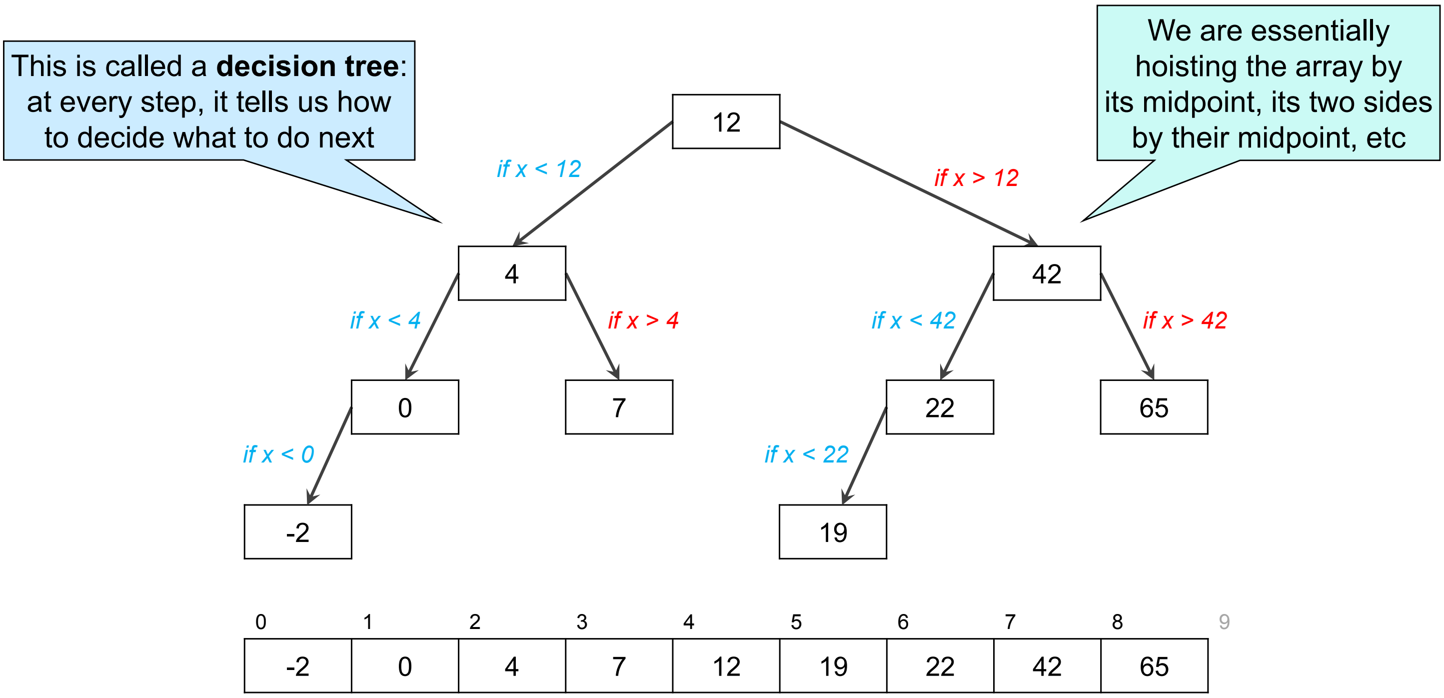
Searching for a Number

- Assume x < 4, so we look at 0
  - if x = 0, we are done
  - if x < 0, we **necessarily** look at -2



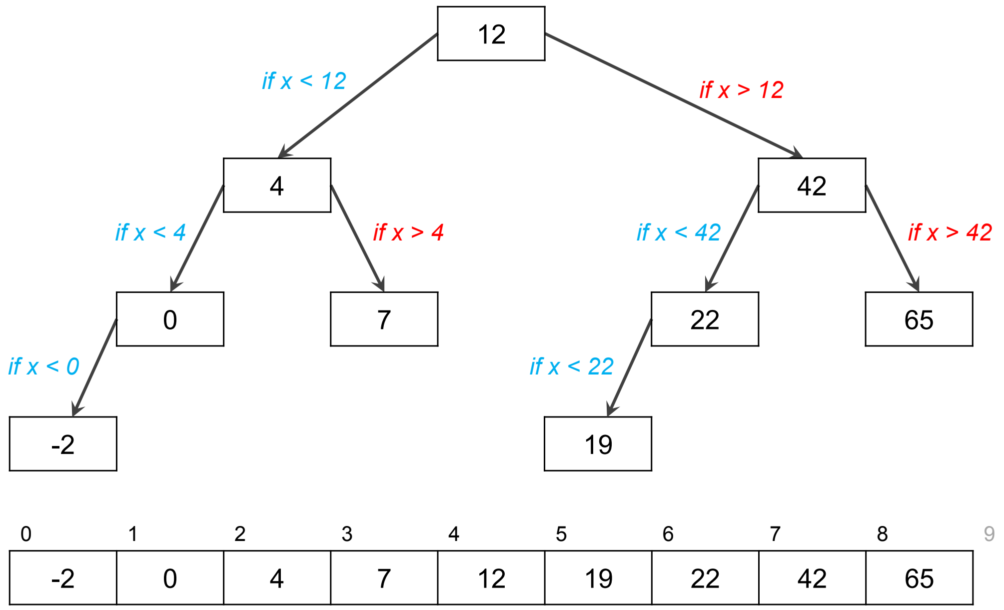
Searching for a Number

- We can map out all possible sequences of elements binary search may examine, for any x



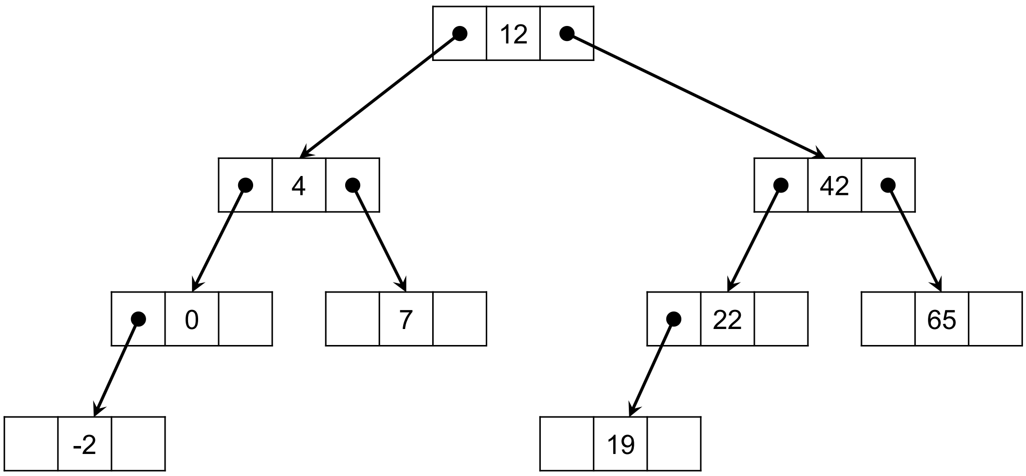
Searching for a Number

- An array provides direct access to all elements
  - This is overkill for binary search
  - At any point, it needs direct access to **at most two** elements



Searching for a Number

- We can achieve the same access pattern by pairing up each element with **two pointers**
  - one to each of the two elements that may be examined next



- We are losing direct access to arbitrary elements,
  - but it retains access to the elements that matter to binary search

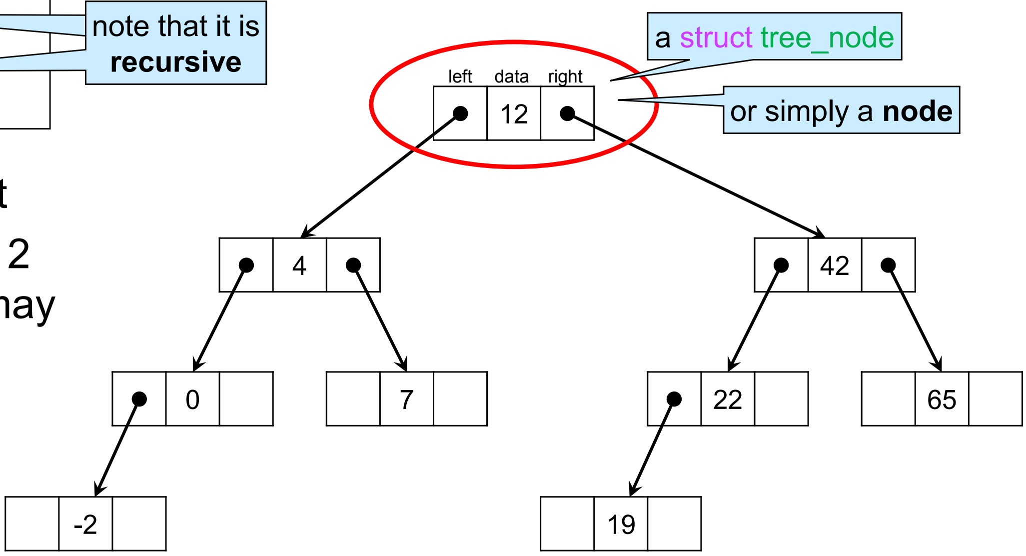
Arrays gave us more power than needed

Towards an Implementation

- We can capture this idea with this **type declaration**:

```
typedef struct tree_node tree;
struct tree_node {
  tree* left;
  int data;
  tree* right;
};
```

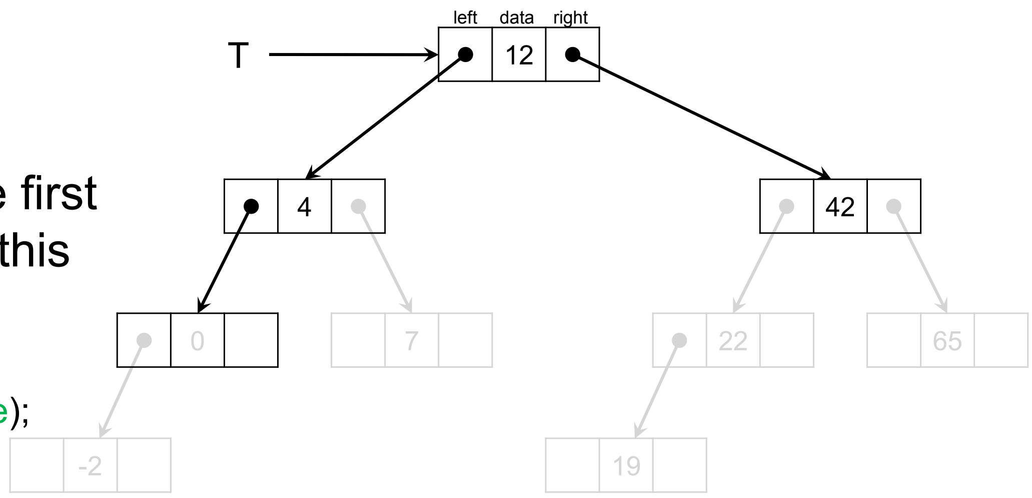
note that it is **recursive**



  - a data element
  - pointers to the 2 elements we may look at next
- This is called a **node**
- This arrangement of data in memory is called a **tree**

Constructing this Tree

```
typedef struct tree_node tree;
struct tree_node {
  tree* left;
  int data;
  tree* right;
};
```



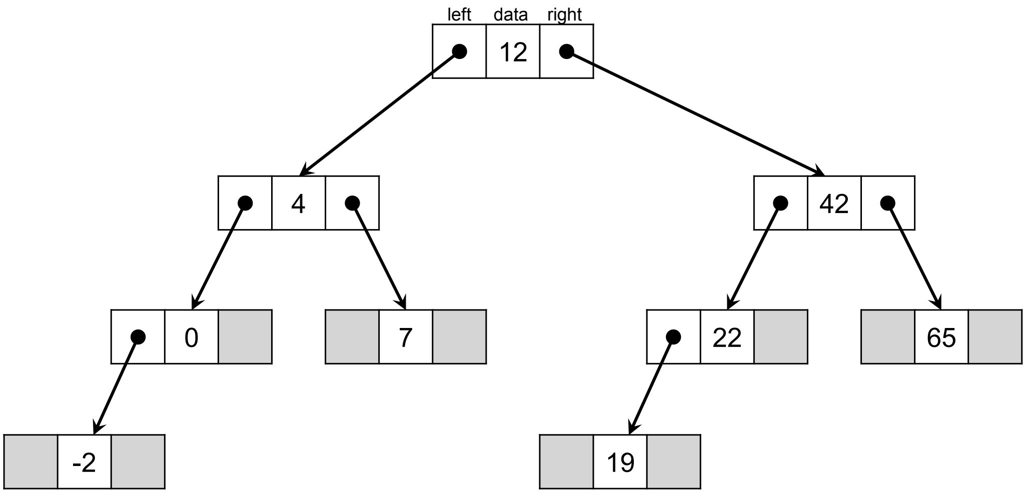
- Let's build the first few nodes of this example

```
tree* T = alloc(tree);
T->data = 12;
T->left = alloc(tree);
T->left->data = 4;
T->right= alloc(tree);
T->right->data = 42;
T->left->left = alloc(tree);
…
```

The End of the Line

```
typedef struct tree_node tree;
struct tree_node {
  tree* left;
  int data;
  tree* right;
};
```

- What should the blank left/right fields point to?



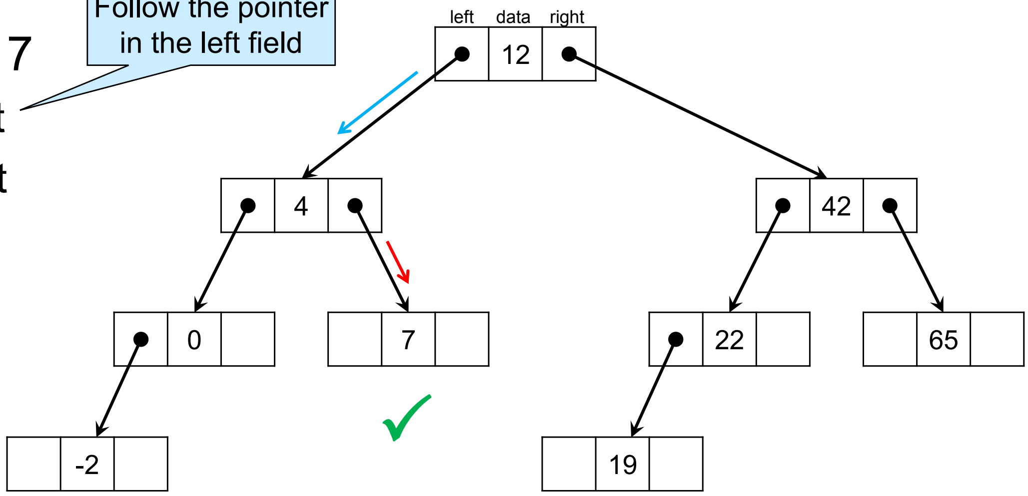
  - NULL ✓
    - each sequence of left/right pointers works like a NULL-terminated list
  - a dummy node ✗
    - not very useful here

We used dummy nodes to get direct access to the end of a list

Searching

- Searching for 7
  - 7 < 12: go left
  - 7 > 4: go right
  - 7 = 7: **found**

Follow the pointer in the left field

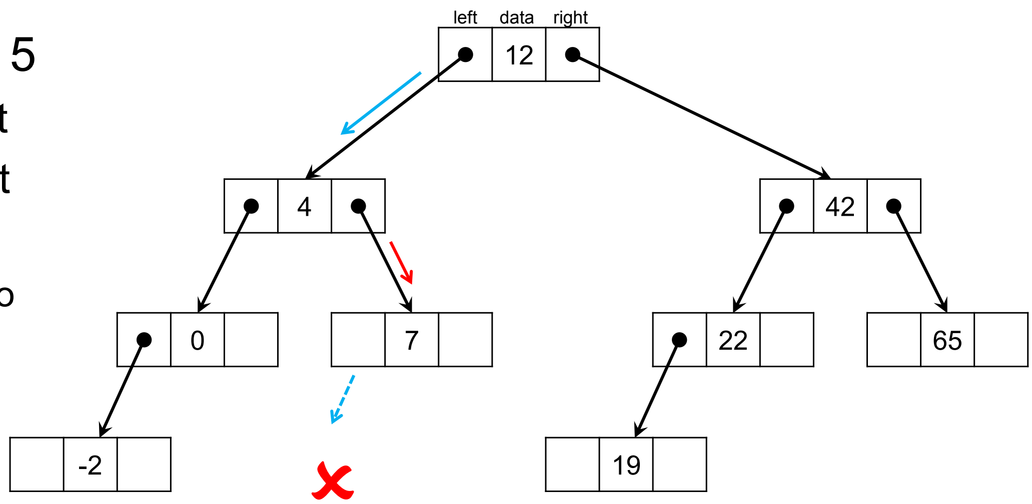


- We are doing the **same steps** as binary search
- Starting from an n-element array, the cost is O(log n)

If the tree is obtained as in this example

Searching

- Searching for 5
  - 5 < 12: go left
  - 5 > 4: go right
  - 5 < 7: go left
    - nowhere to go
  - **not there**



- We are doing the **same steps** as binary search
- Starting from an n-element array, the cost is O(log n)

If the tree is obtained as in this example

Recall our Goal

- Develop a data structure that has **guaranteed** O(log n) worst-case complexity for lookup, insert and find_min
  - **always!**

○ lookup has cost O(log n)

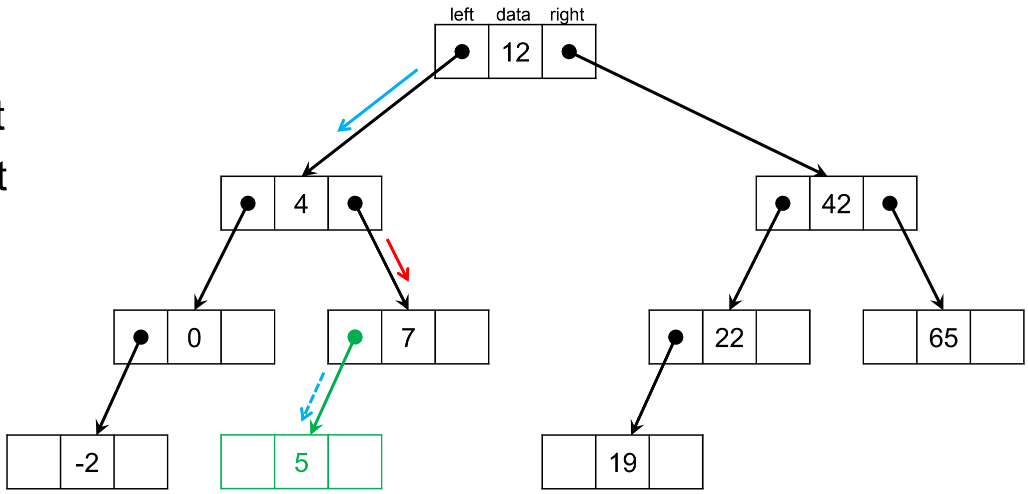
in *this* setup

	Target data structure
lookup	*O(log n)* ✓
insert	*O(log n)*
find_min	*O(log n)*

○ What about insert and find_min?

Insertion

- Inserting 5
  - 5 < 12: go left
  - 5 > 4: go right
  - 5 < 7: go left
    - **put it there**



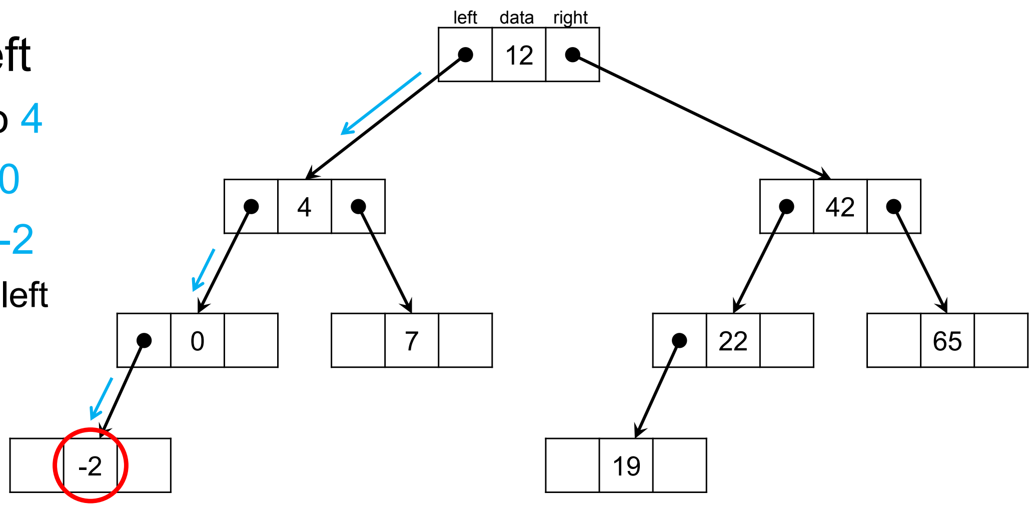
- We are doing the **same steps** we would do to search for it, and then put it where it should have been
  - so that we find it when searching for it next time
- For an n-element array, this costs O(log n)

We couldn’t get this with sorted arrays

If the tree is obtained as in this example

Finding the Smallest Key

- Keep going left
  - left from 12 to 4
  - left from 4 to 0
  - left from 0 to -2
    - nothing to its left
  - **the smallest key is -2**



- Starting from an n-element array, we can go left at most O(log n) times
  - The cost is O(log n)

If the tree is obtained as in this example

Recall our Goal

- Develop a data structure that has **guaranteed** O(log n) worst-case complexity for lookup, insert and find_min
  - **always!**

  - lookup, insert and find_min all have cost O(log n)

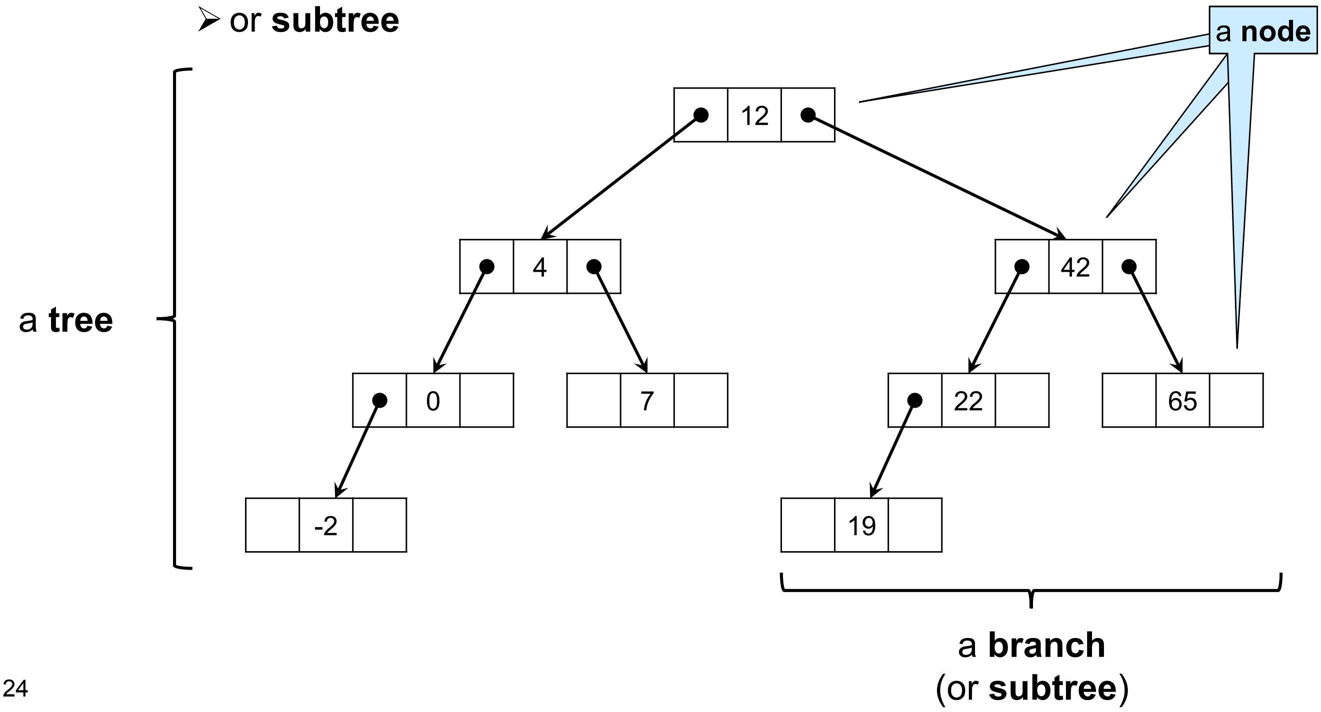
in *this* setup

	Target data structure
lookup	*O(log n)* ✓
insert	*O(log n)* ✓
find_min	*O(log n)* ✓

Trees

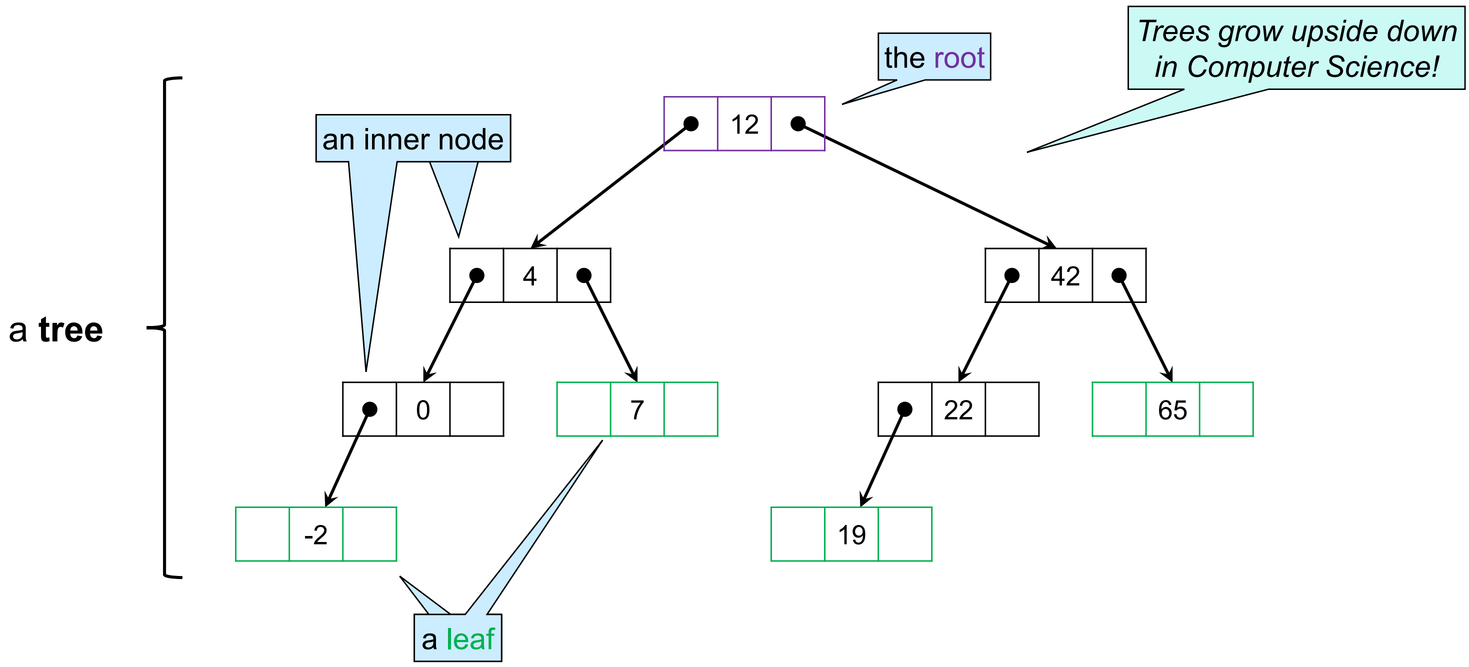
Terminology

- This arrangement of data is called a (binary) **tree**
  - each item in it is called a **node**
  - the part of a tree hanging from a node is called a **branch**
    - or **subtree**



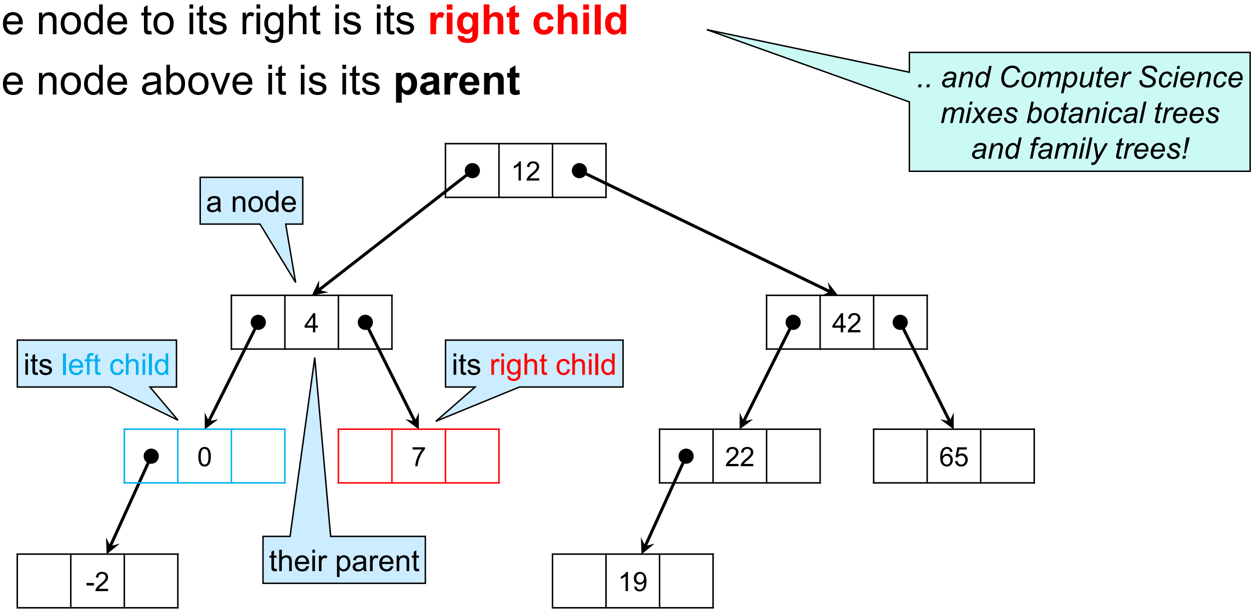
Terminology

- The node at the top is called the **root** of the tree
  - the nodes at the bottom are the **leaves** of the tree
  - the other nodes are called **inner nodes**



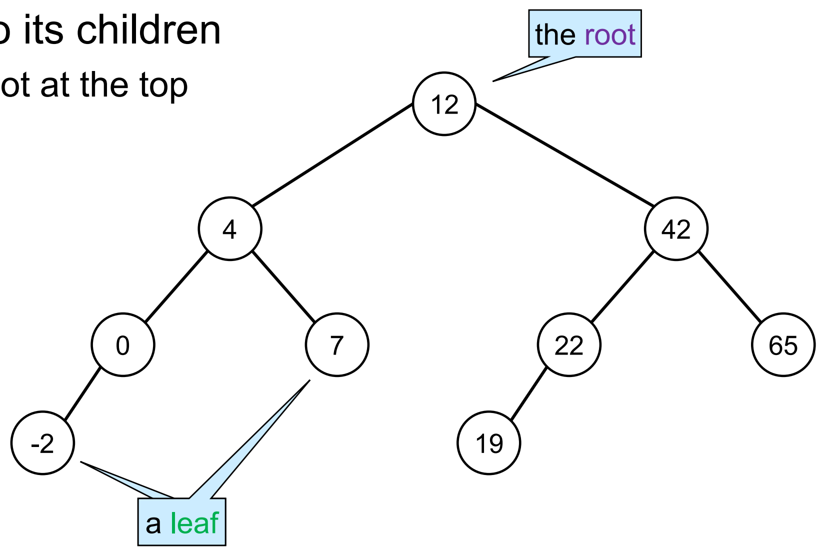
Terminology

- Given any node
  - the node to its left is its **left child**
  - the node to its right is its **right child**
  - the node above it is its **parent**



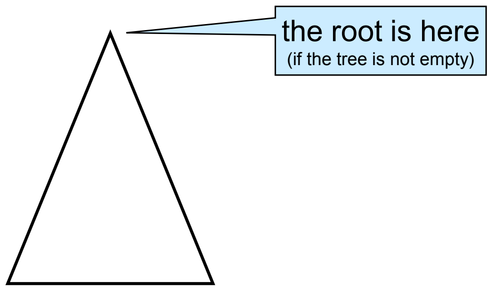
Concrete Tree Diagrams

- When drawing trees, we generally omit the details of the memory diagrams
  - draw just the data in a node
    - not the pointer fields
  - and the connection to its children
    - we always draw the root at the top



Pictorial Abstraction

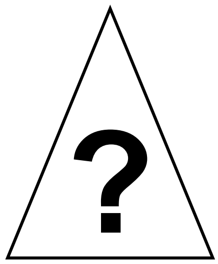
- We will often reason about trees that are arbitrary
  - their actual content is unimportant, so we abstract it away
  - We draw a generic tree as a triangle



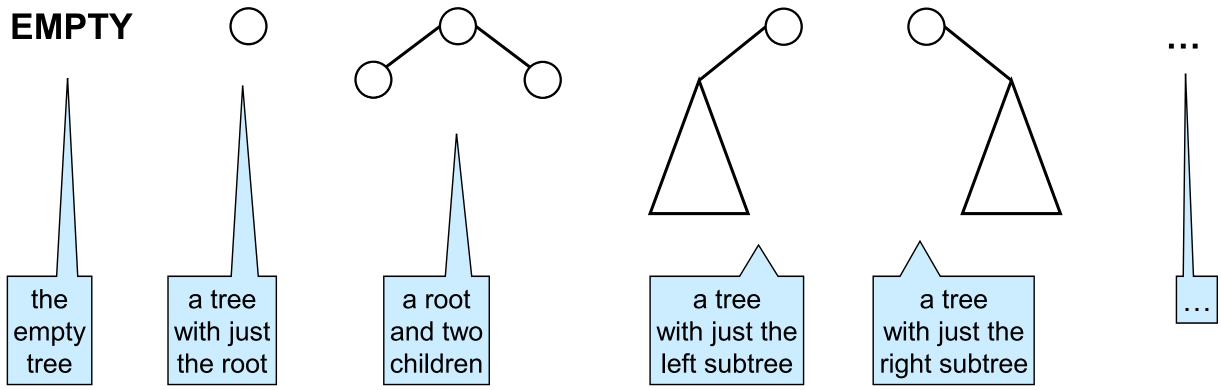
  - We represent the empty tree by simply writing "Empty"

Empty

What do Trees Look Like?



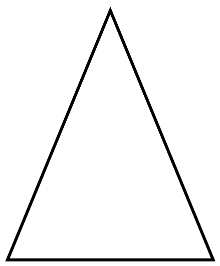
- Abstract trees come in many shapes

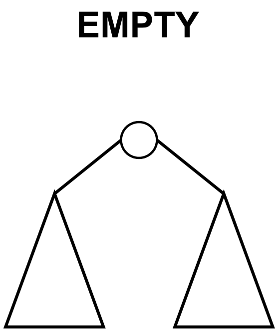


- When working with trees, we need to account for all these possibilities
  - we will forget some
- *Is there a simpler description?*

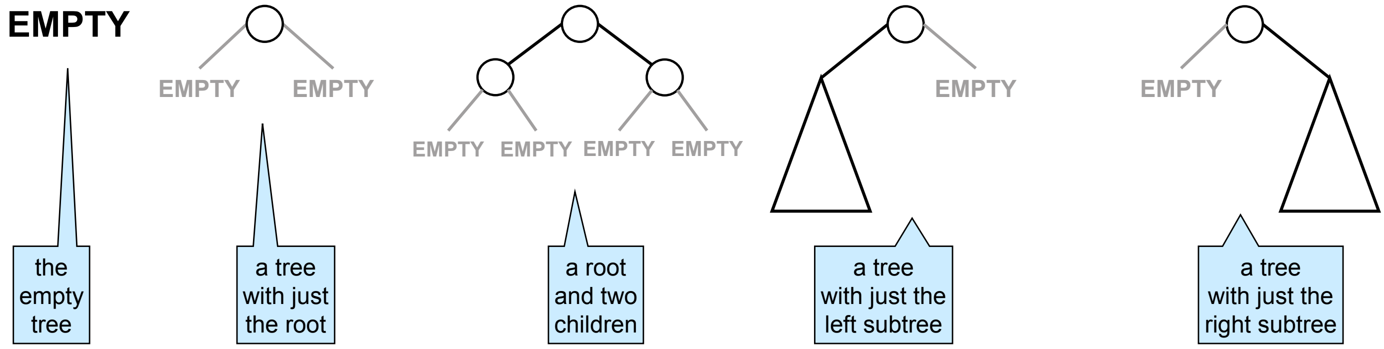
What Trees Look Like

- A tree can be
  - either empty
  - or a root with
    a tree on its left and
    a tree on its right



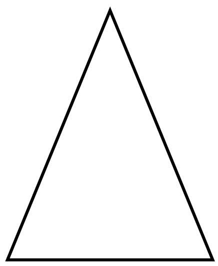


- **Every tree** reduces to these two cases



What Trees Look Like

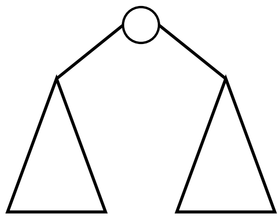
- A tree can be



  - either empty

EMPTY

  - or a root with
    a tree on its left and
    a tree on its right



- We only need to consider these two cases when
  - writing code about trees
  - reasoning about trees

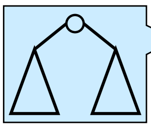
A Minimal Tree Invariant

- We only need to consider these two cases when writing code about trees
- Let's apply this to write a basic invariant about trees of *entries*
  - Just check that the data field is never NULL

```
typedef struct tree_node tree;
struct tree_node {
  tree* left;
  entry data;   // != NULL
  tree* right;
};
```

Recall we are using trees to implement dictionaries:
- we store entries in nodes
- valid entries are non-NULL

EMPTY



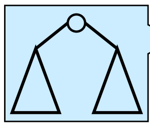
```
bool is_tree(tree* T) {
  // Code for empty tree
  if (T == NULL) return true;

  // Code for non-empty tree
  return is_tree(T->left)
      && T->data != NULL
      && is_tree(T->right);
}
```

- This is a **recursive** function
  - the **base case** is about the empty tree
  - the **recursive case** is about every tree that is not empty
    - with a root
    - and two subtrees

A Minimal Tree Invariant

- We just check that the data field is never NULL

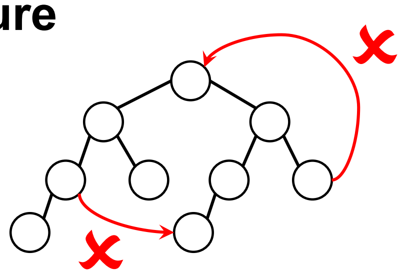
EMPTY



```
bool is_tree(tree* T) {
  // Code for empty tree
  if (T == NULL) return true;

  // Code for non-empty tree
  return is_tree(T->left)
      && T->data != NULL
      && is_tree(T->right);
}
```

- But trees have constraints on their **structure**
  - a node does not point to an ancestor
  - a node has at most one parent

How to check them is left as an exercise



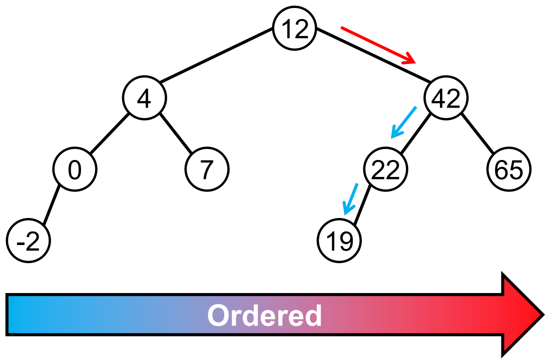
- *What additional constraints on contents do we need to use trees to implement dictionaries?*

Binary Search Trees

Binary Search Trees

- *What additional constraints on the contents do we need to use trees to implement dictionaries?*

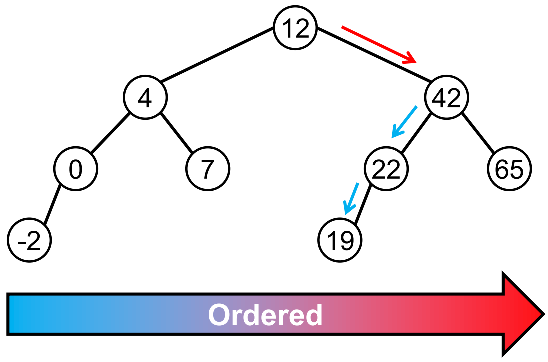
- Because lookup emulates binary search, the data in the tree need to be **ordered**
  - smaller values on the left
  - bigger values on the right



- A tree whose nodes are ordered is called a **binary search tree**

The BST Invariant

- A tree whose nodes are ordered is called a **binary search tree**



- We can write a specification function that check BSTs

```
bool is_bst(tree* T) {
  return is_tree(T)
      && is_ordered(T);
}
```

A BST is a valid tree …

… whose nodes are ordered

We will see later how to implement this