Binary Search Trees

Reflecting on Dictionaries

Cost

- Complexity of various implementations of dictionaries
  - assuming it contains *n* entries

	Unsorted array	***Array sorted by key***	***Linked list***	***Hash Table***
lookup	*O(n)*	*O(log n)*	*O(n)*	*O(1)* *average*
insert	*O(1)* *amortized*	*O(n)*	*O(1)*	*O(1)* *average and amortized*

- Hash dictionaries are clearly the best implementation
  - O(1) lookup and insertion are hard to beat!

Cost

- *Hash dictionaries are clearly the best implementation*
  - *O(1) lookup and insertion are hard to beat!*

or are they?

- It's O(1) **average**
  - we could be (very) unlucky and incur an O(n) cost
    - e.g., if we use a poor hash function
- It's O(1) **amortized**
  - from time to time, we need to resize the table
    - then the operation costs O(n)

Always read the fine prints!

- Operations like finding the entry with the smallest key cost O(n)
  - we have to check every entry

Using hash dictionaries is too risky or not good enough for applications that require a guaranteed (short) response time

But they are great for applications that don't have such constraints

Goal

- Develop a data structure that has **guaranteed** O(log n) worst-case complexity for lookup, insert and find_min
  - **always!**
  - O(1) would be great but we can't get that

Returns the entry with the smallest key

	Unsorted array	*Array sorted by key*	*Linked list*	*Hash Table*	
lookup	*O(n)*	*O(log n)*	*O(n)*	*O(1)* *average*	*O(log n)*
insert	*O(1)* *amortized*	*O(n)*	*O(1)*	*O(1)* *average and amortized*	*O(log n)*
find_min	*O(n)*	*O(1)*	*O(n)*	*O(n)*	*O(log n)*

Exercise

Getting Started

- The only O(log n) so far is lookup in sorted arrays

	Unsorted array	*Array sorted by key*	*Linked list*	*Hash Table*	
lookup	*O(n)*	*O(log n)*	*O(n)*	*O(1)* *average*	*O(log n)*
insert	*O(1)* *amortized*	*O(n)*	*O(1)*	*O(1)* *average and amortized*	*O(log n)*
find_min	*O(n)*	*O(1)*	*O(n)*	*O(n)*	*O(log n)*

- That's binary search
  - Let's start there

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

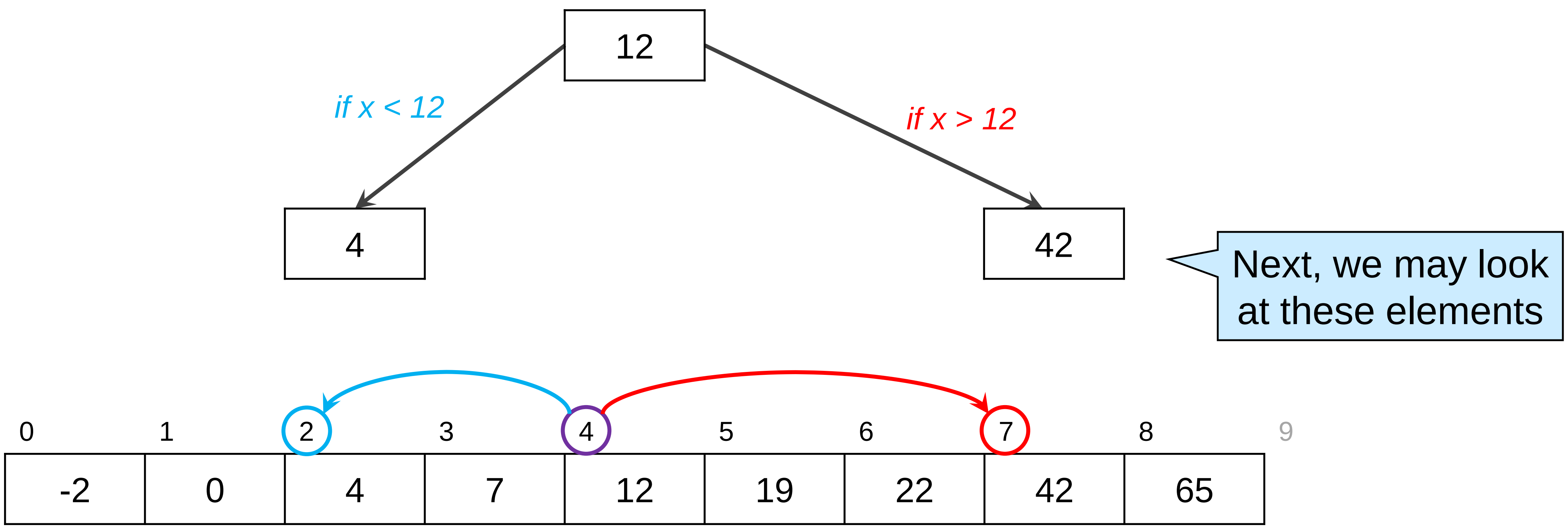
12

We always look at this element

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

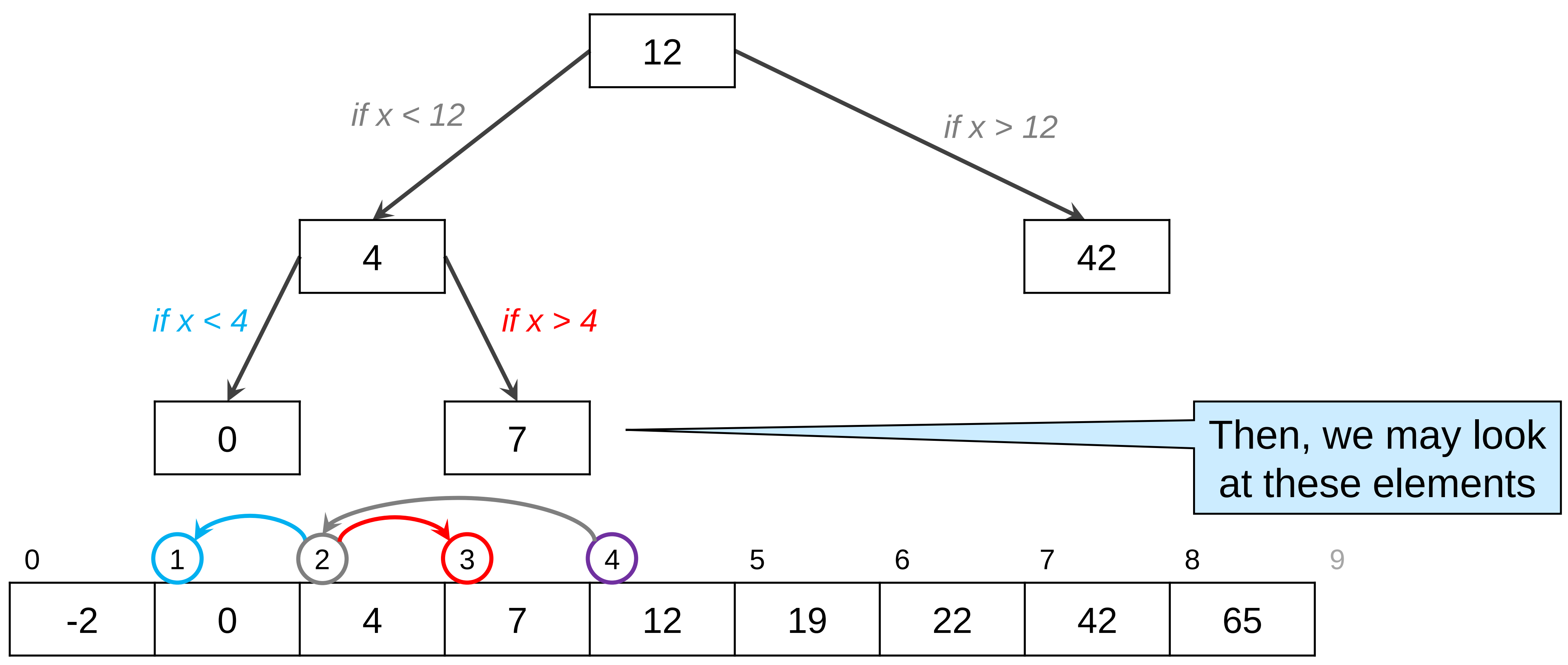
Searching for a Number

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



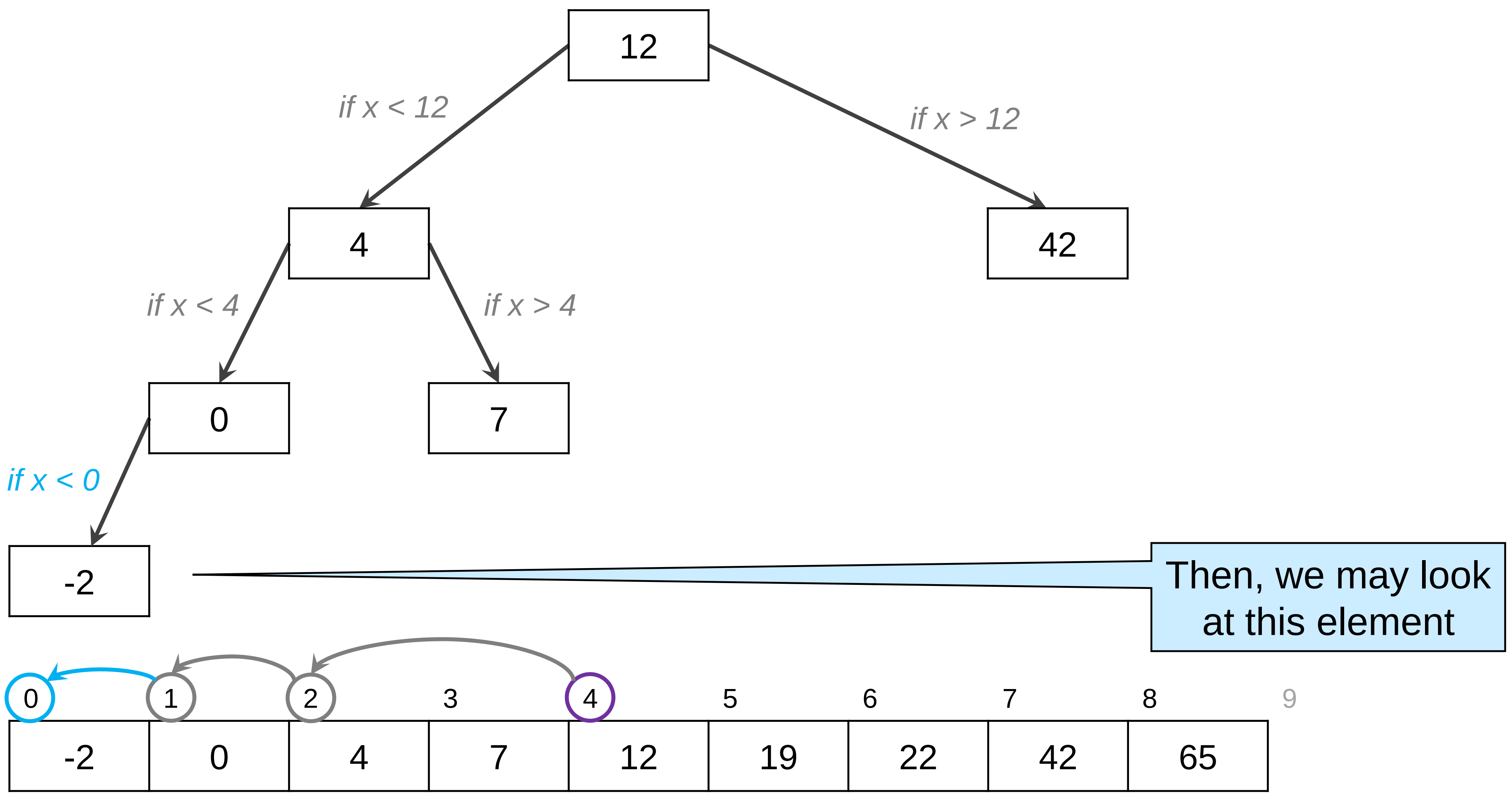
Searching for a Number

- 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



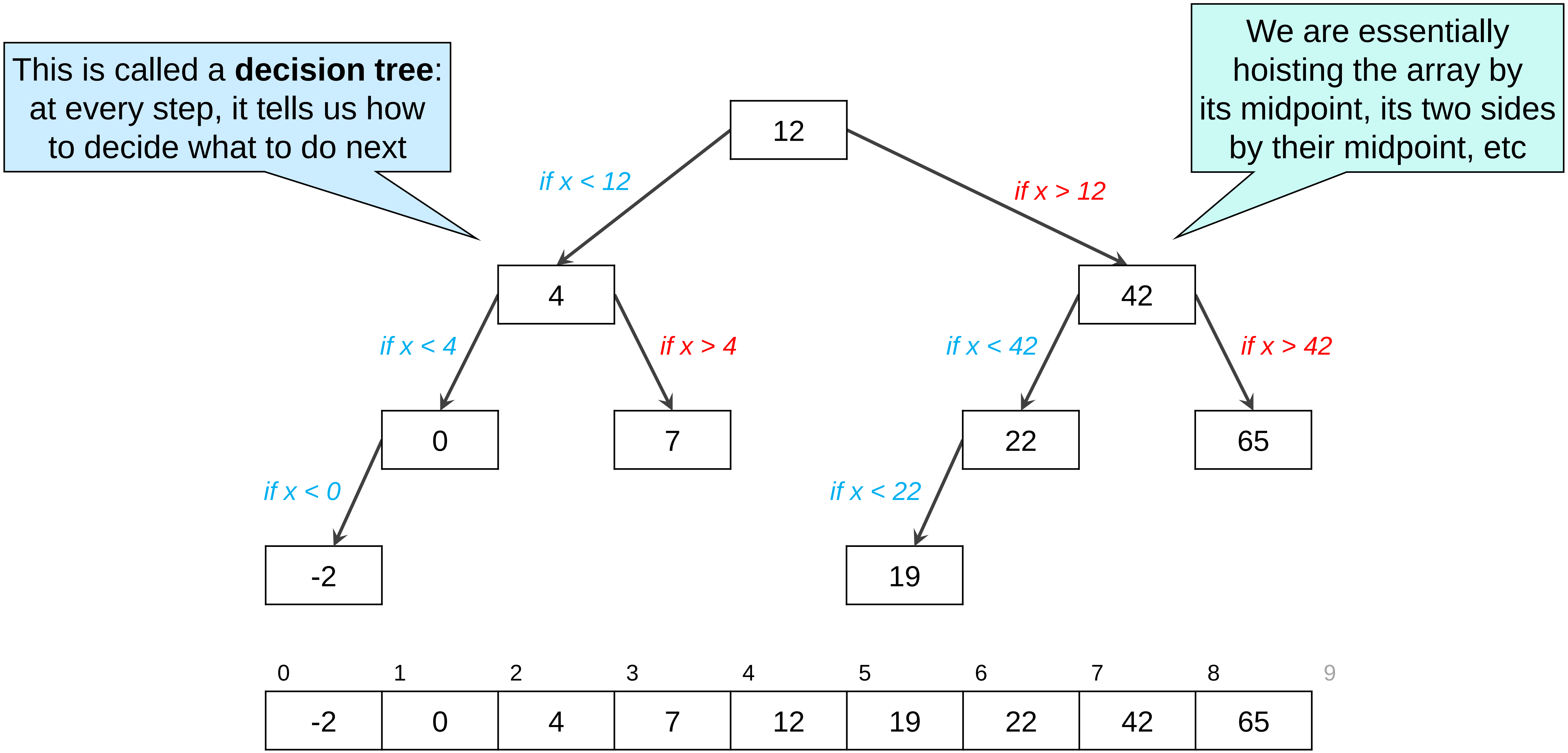
Searching for a Number

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



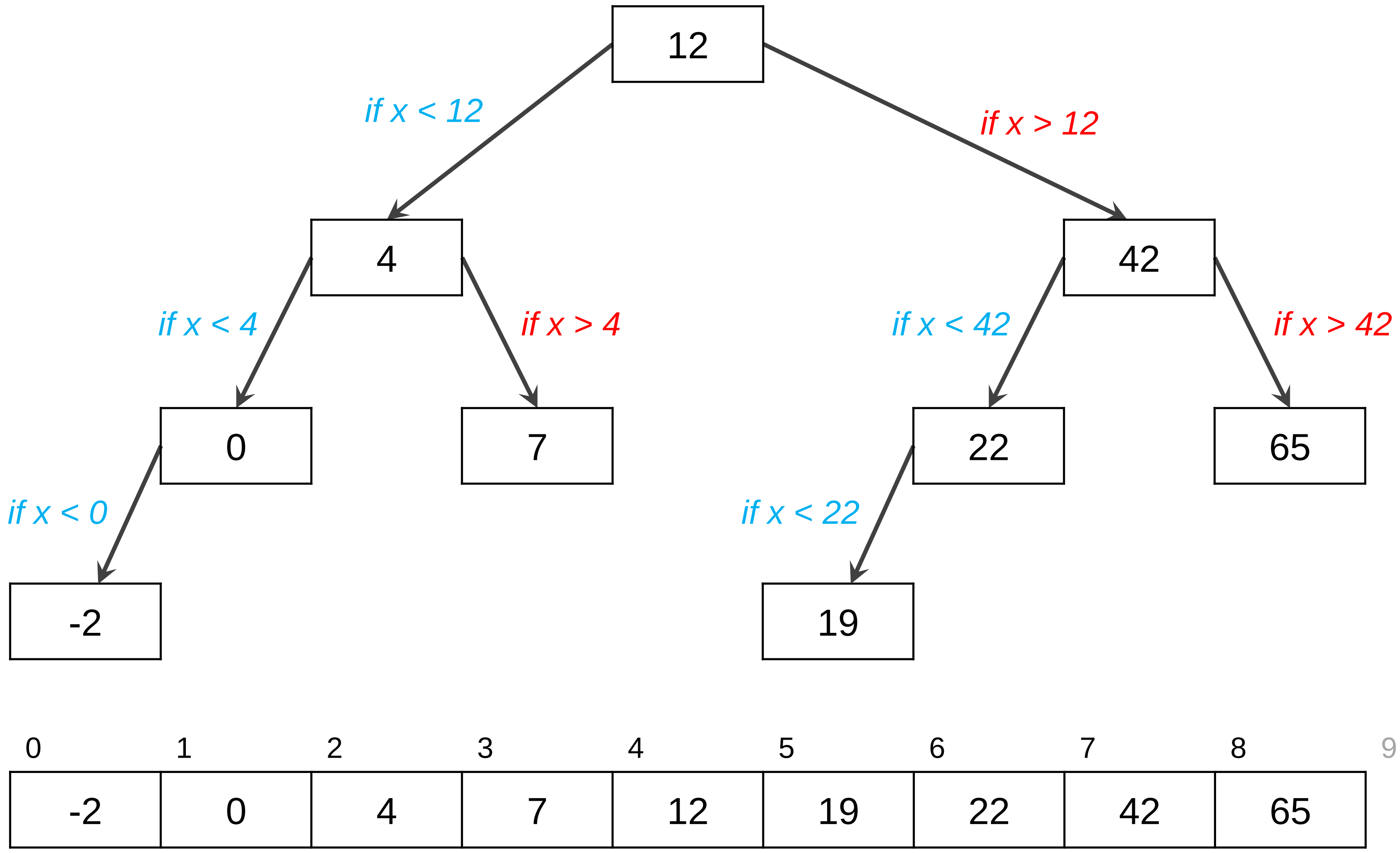
Searching for a Number

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



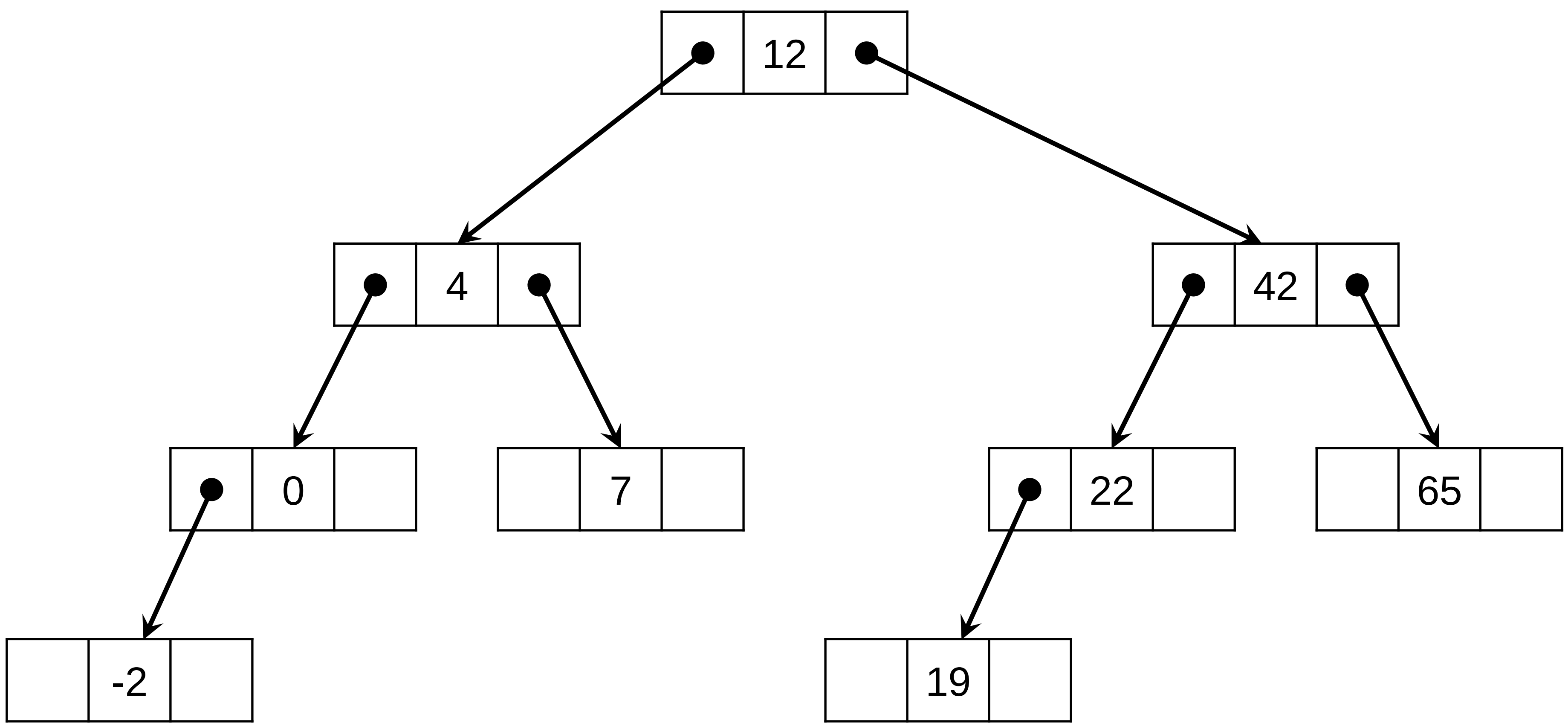
Searching for a Number

- 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

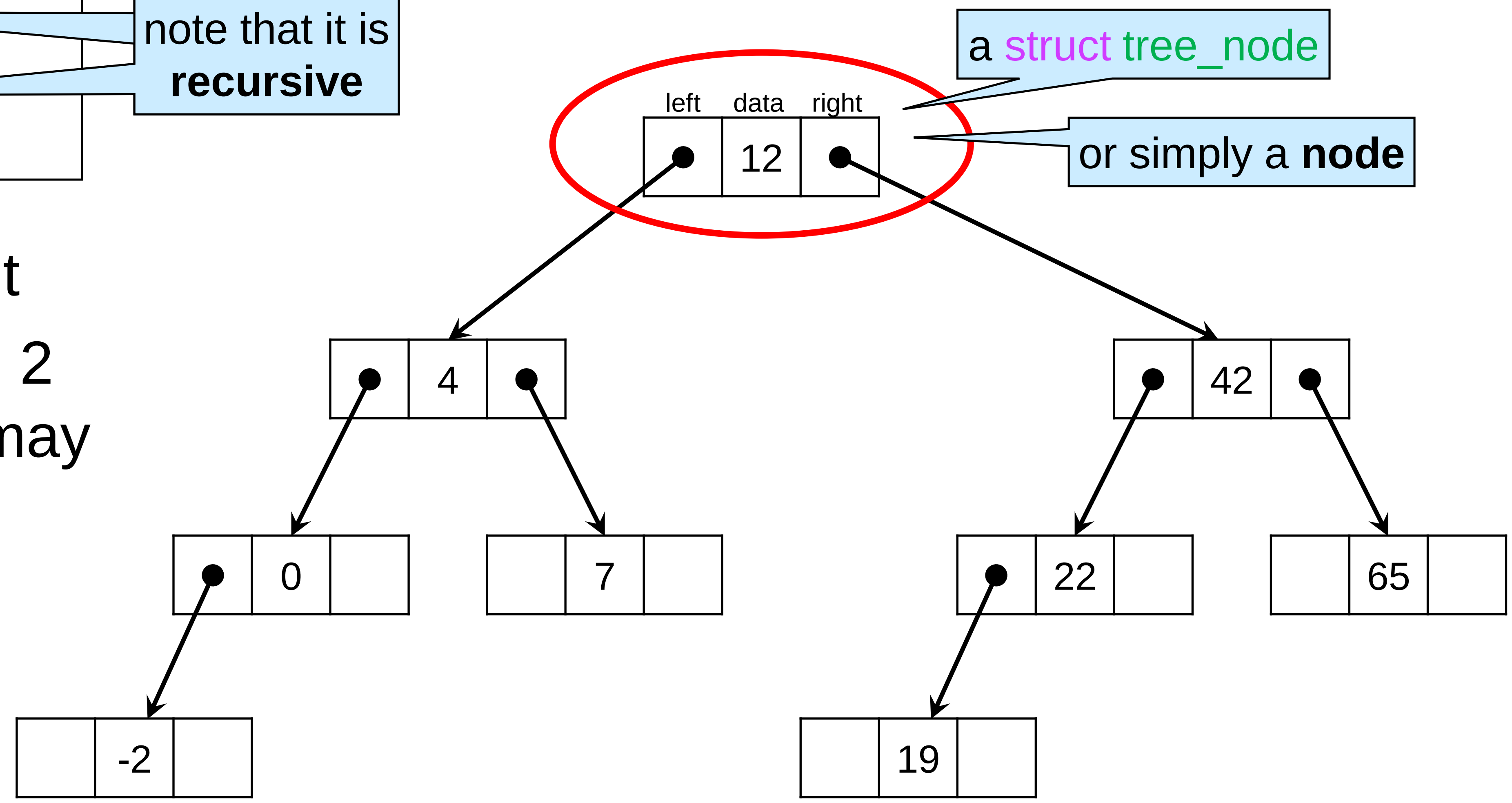
Arrays gave us more power than needed

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**



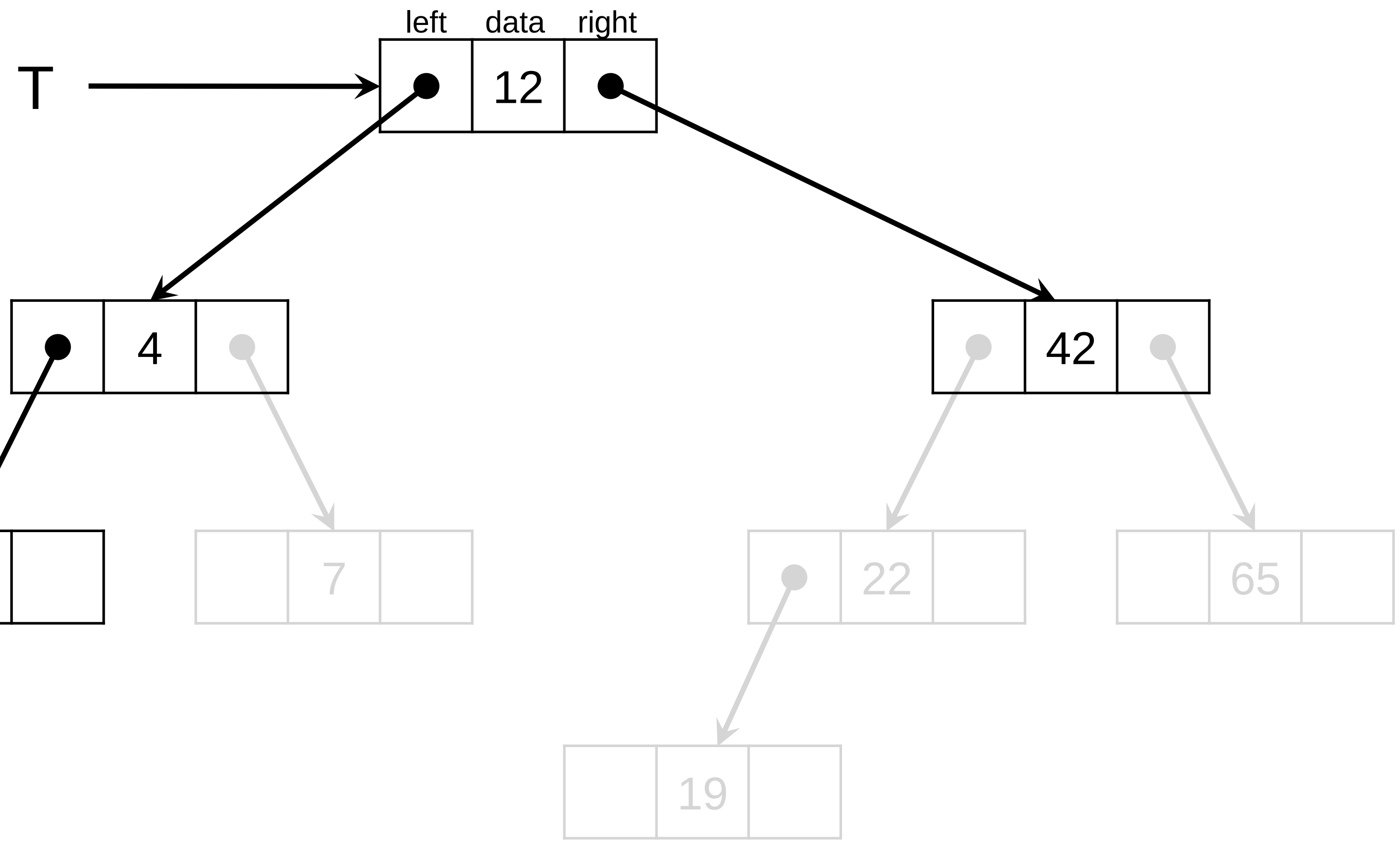
  - a data element
  - pointers to the 2 elements we may look at next

- 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;
};
```



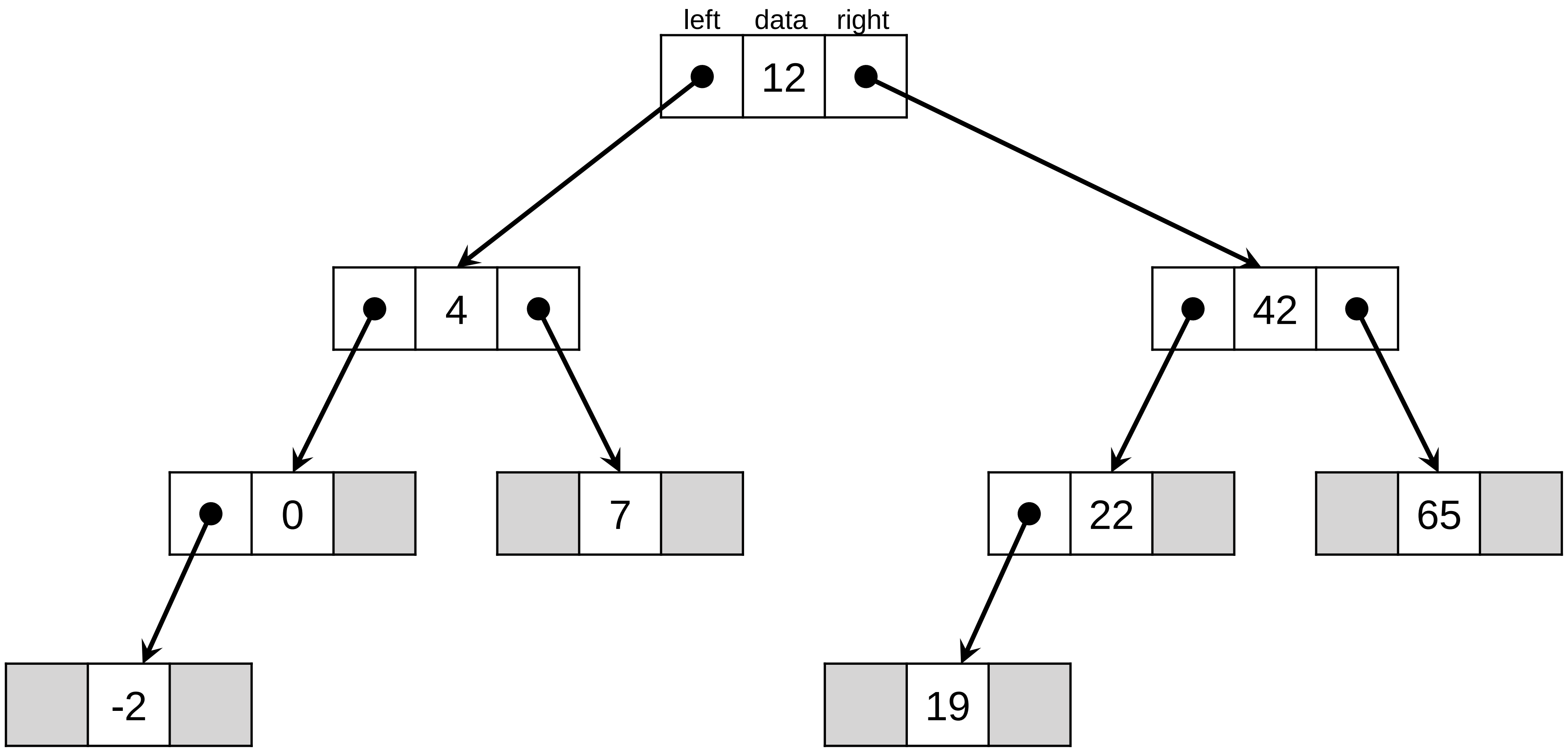
- Let's build the first few nodes of this example

```
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?



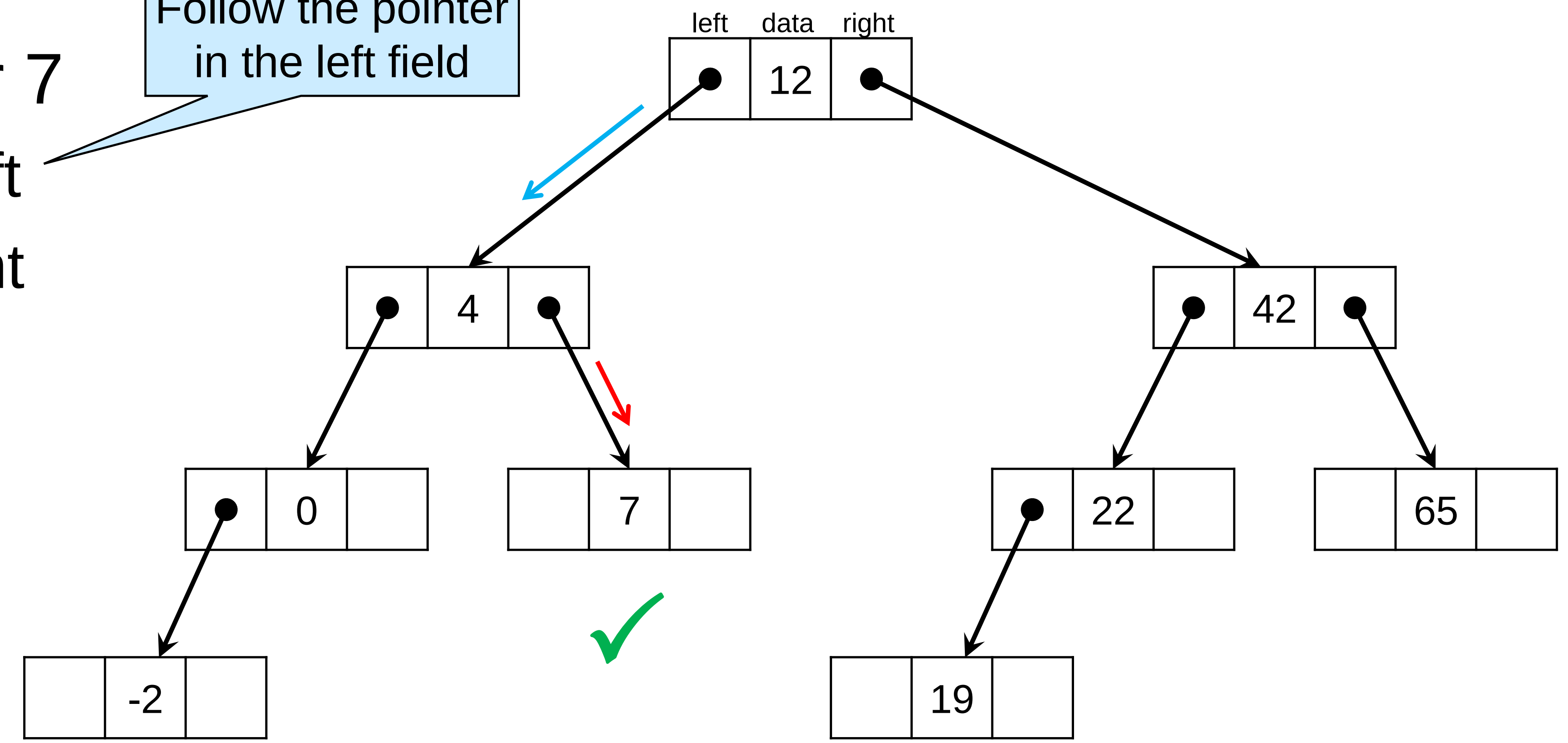
  - NULL
    - each sequence of left/right pointers works like a NULL-terminated list
  - a dummy node
    - not very useful here

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

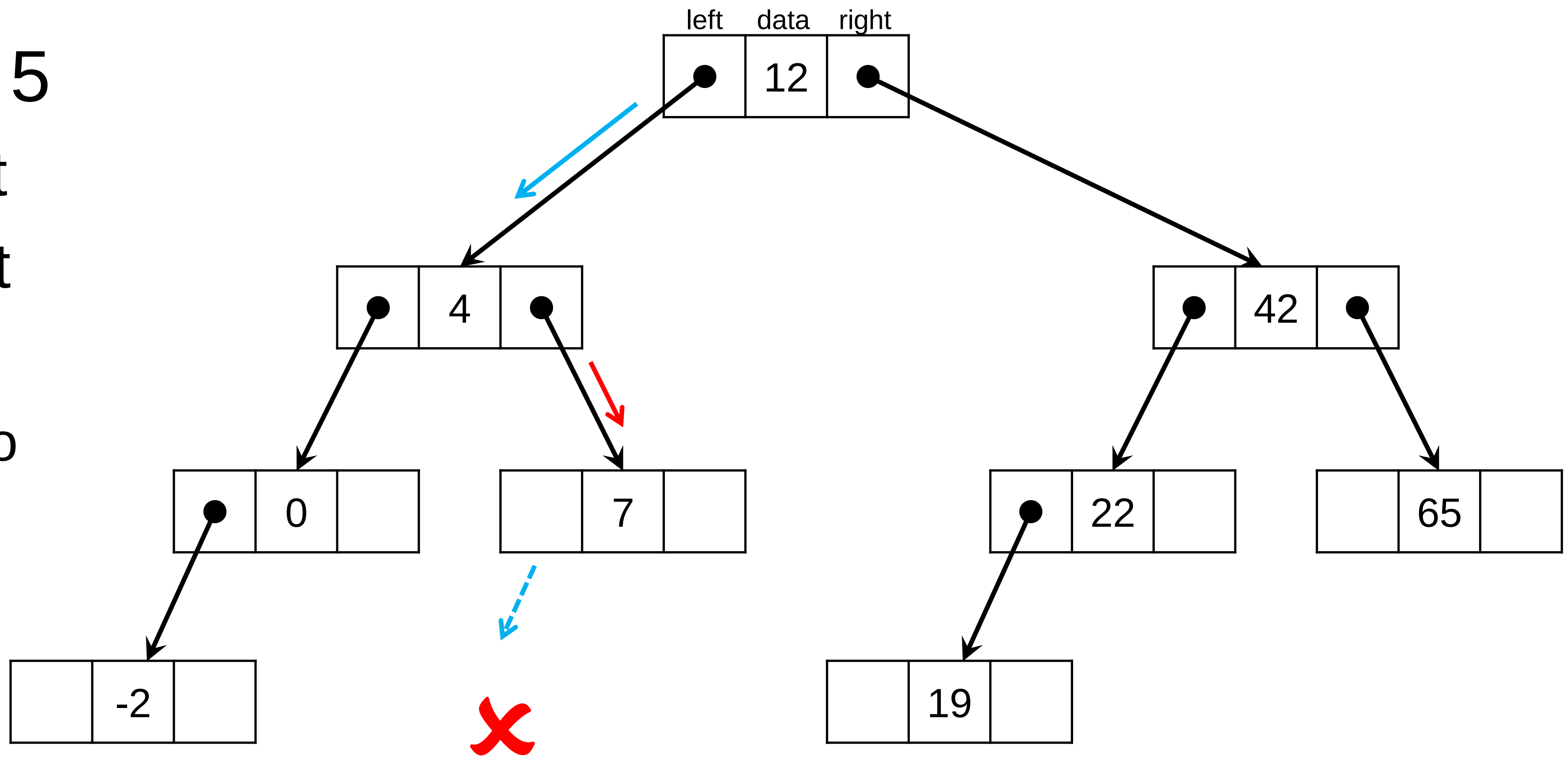


- We are doing the **same steps** as binary search
- Starting from an n-element array, the cost is O(log n)

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

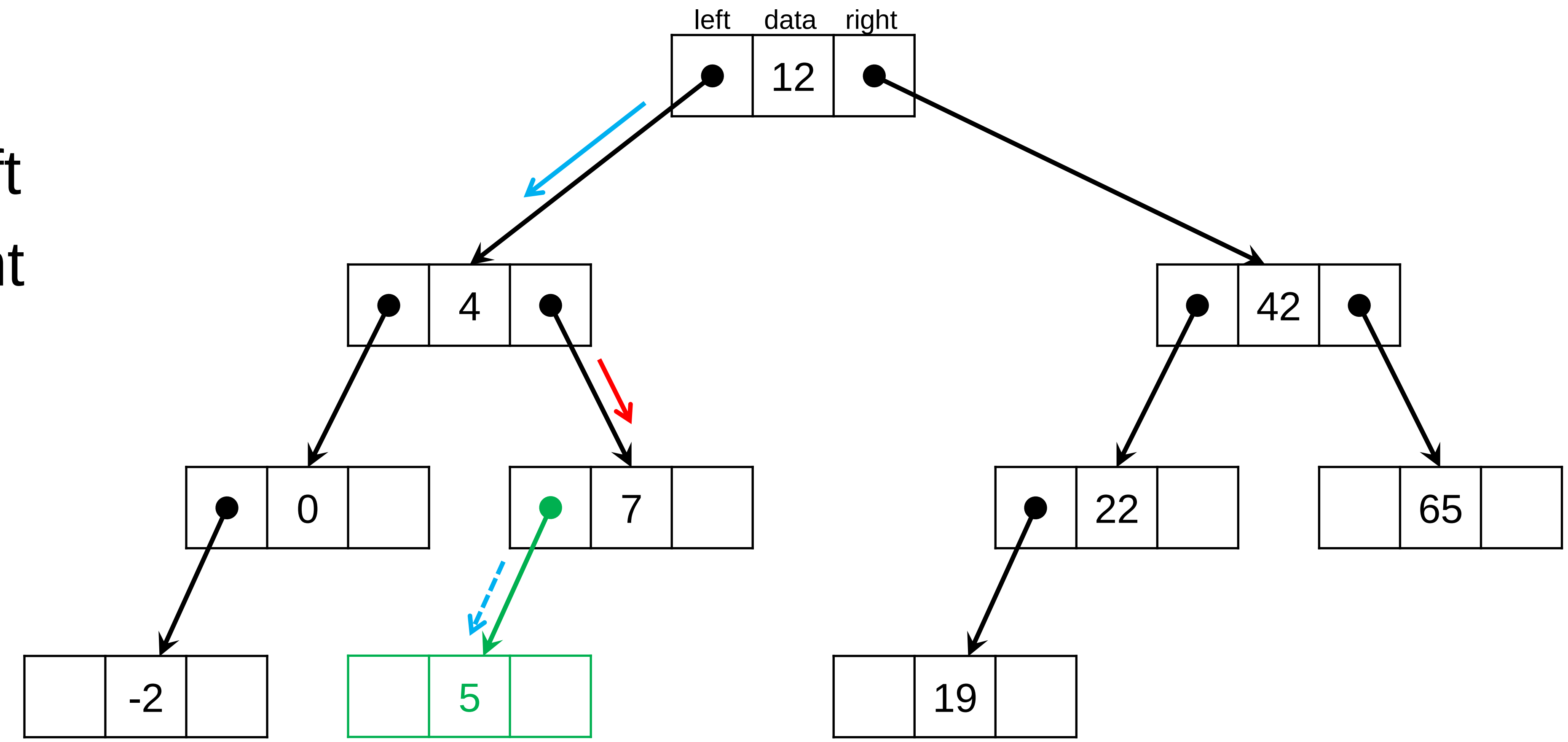
	Target data structure
lookup	*O(log n)*
insert	*O(log n)*
find_min	*O(log n)*

✓

○ What about insert and find_min?

Insertion

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



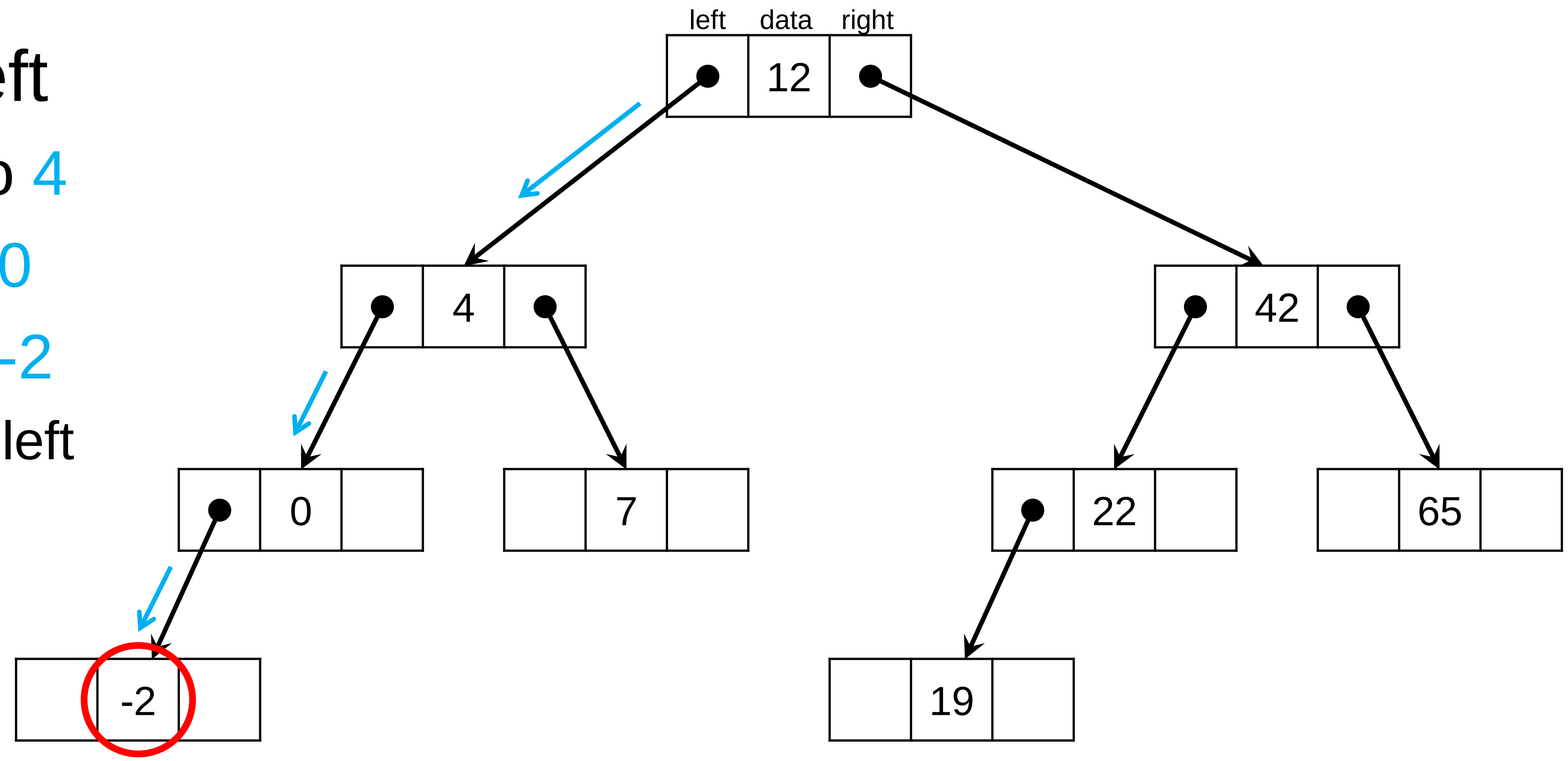
- We are doing the **same steps** we would do to search for it, and then put it where it should have been
  - so that we find it when searching for it next time
- For an n-element array, this costs O(log n)

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)

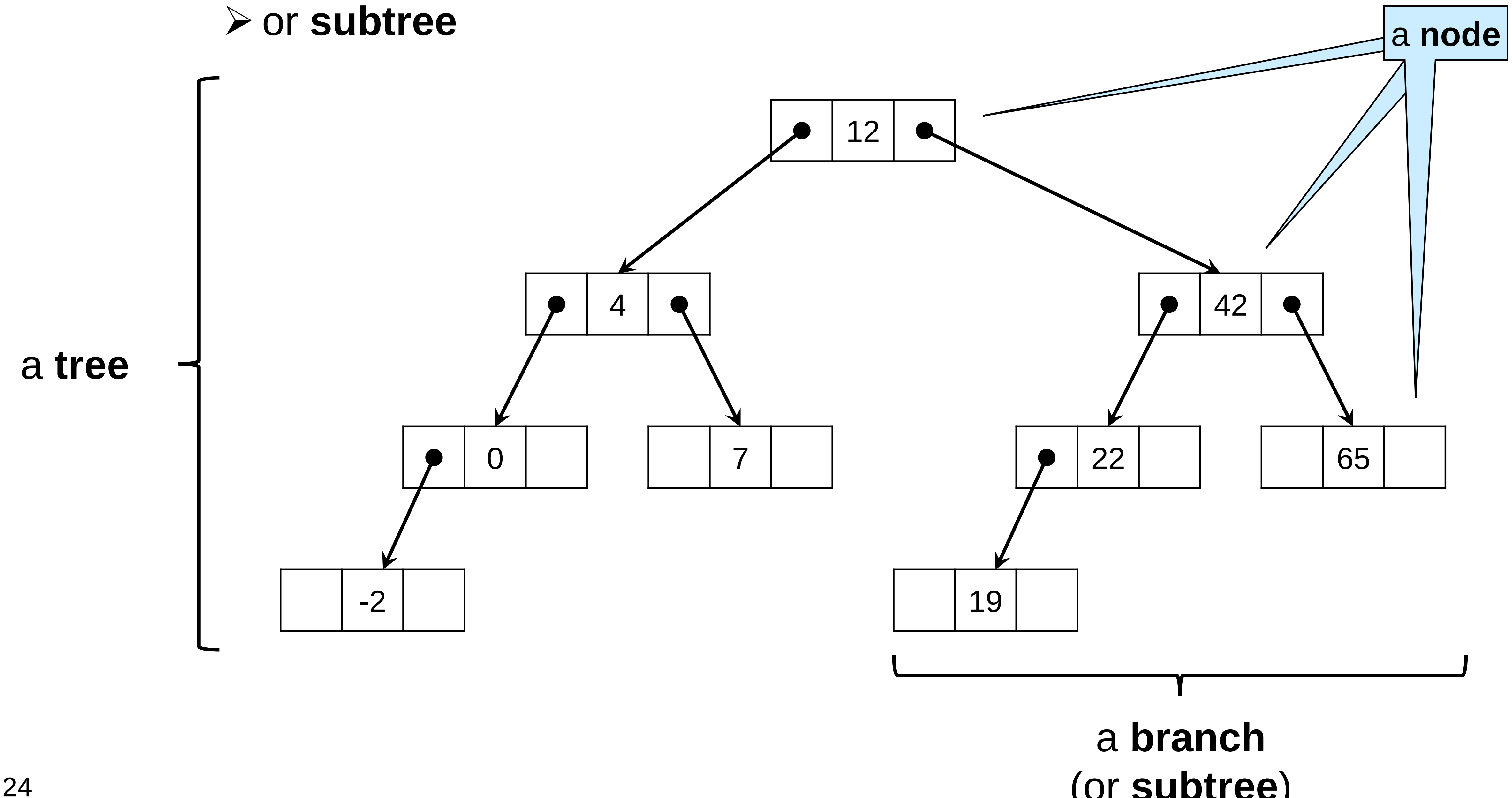
in *this* setup

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

Trees

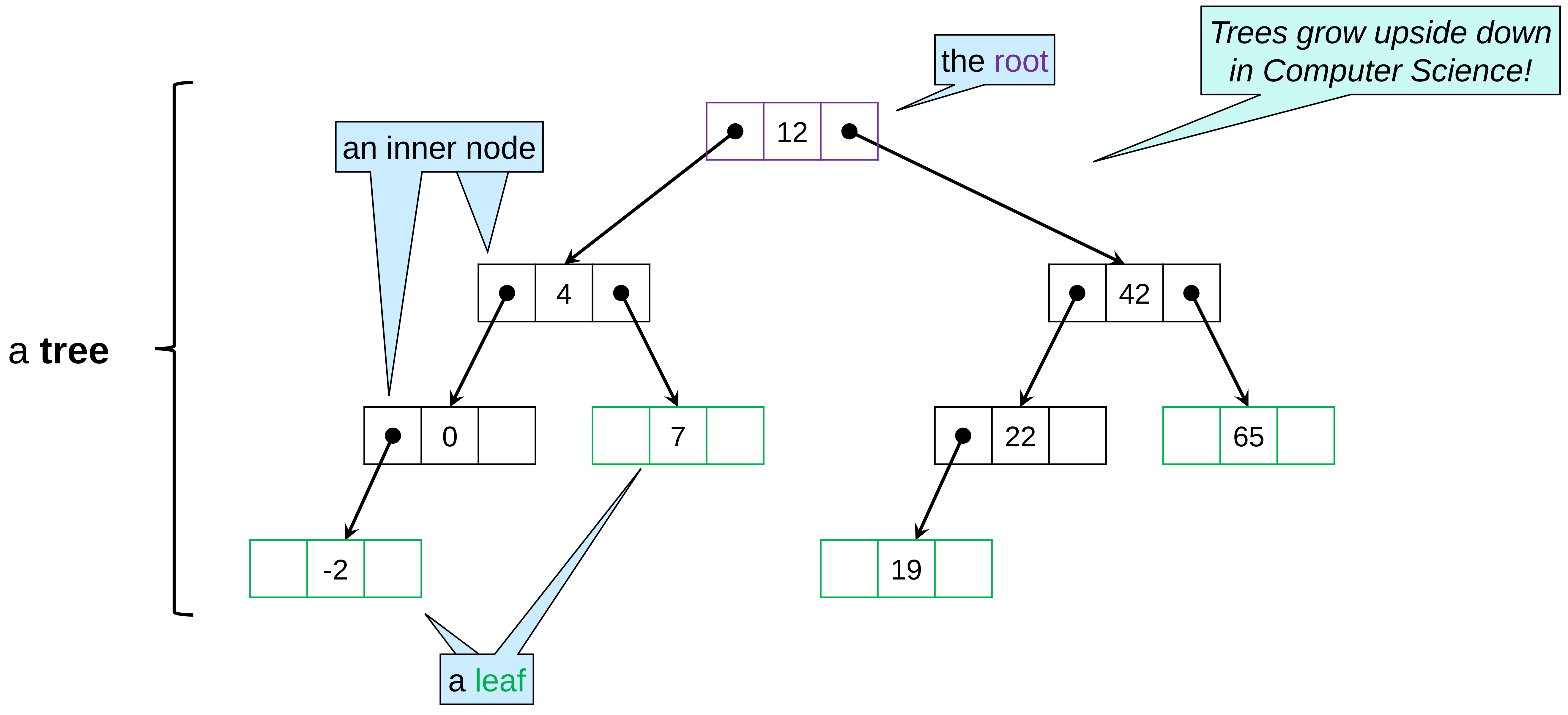
Terminology

- 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

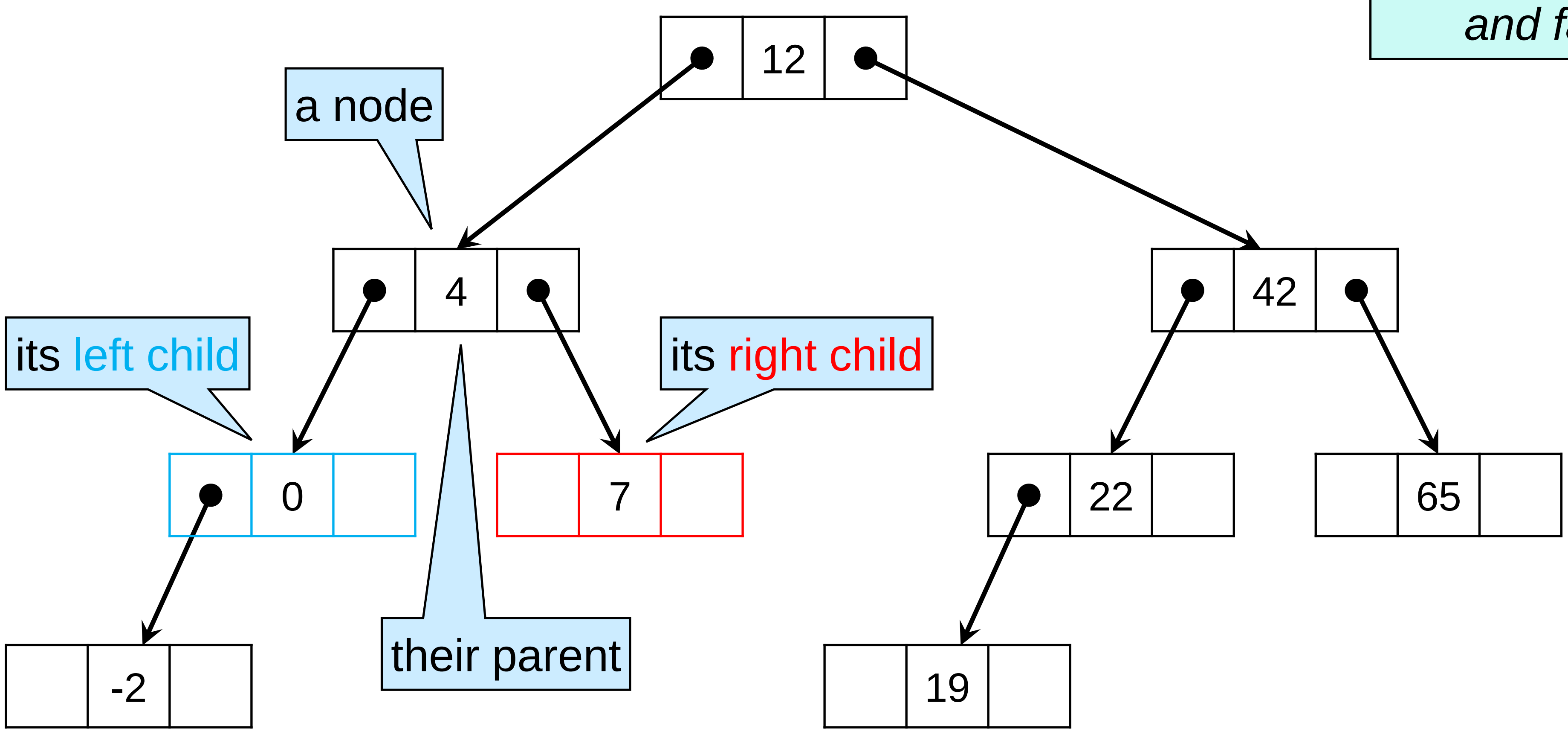
- The node at the top is called the **root** of the tree
  - the nodes at the bottom are the **leaves** of the tree
  - the other nodes are called **inner nodes**



Terminology

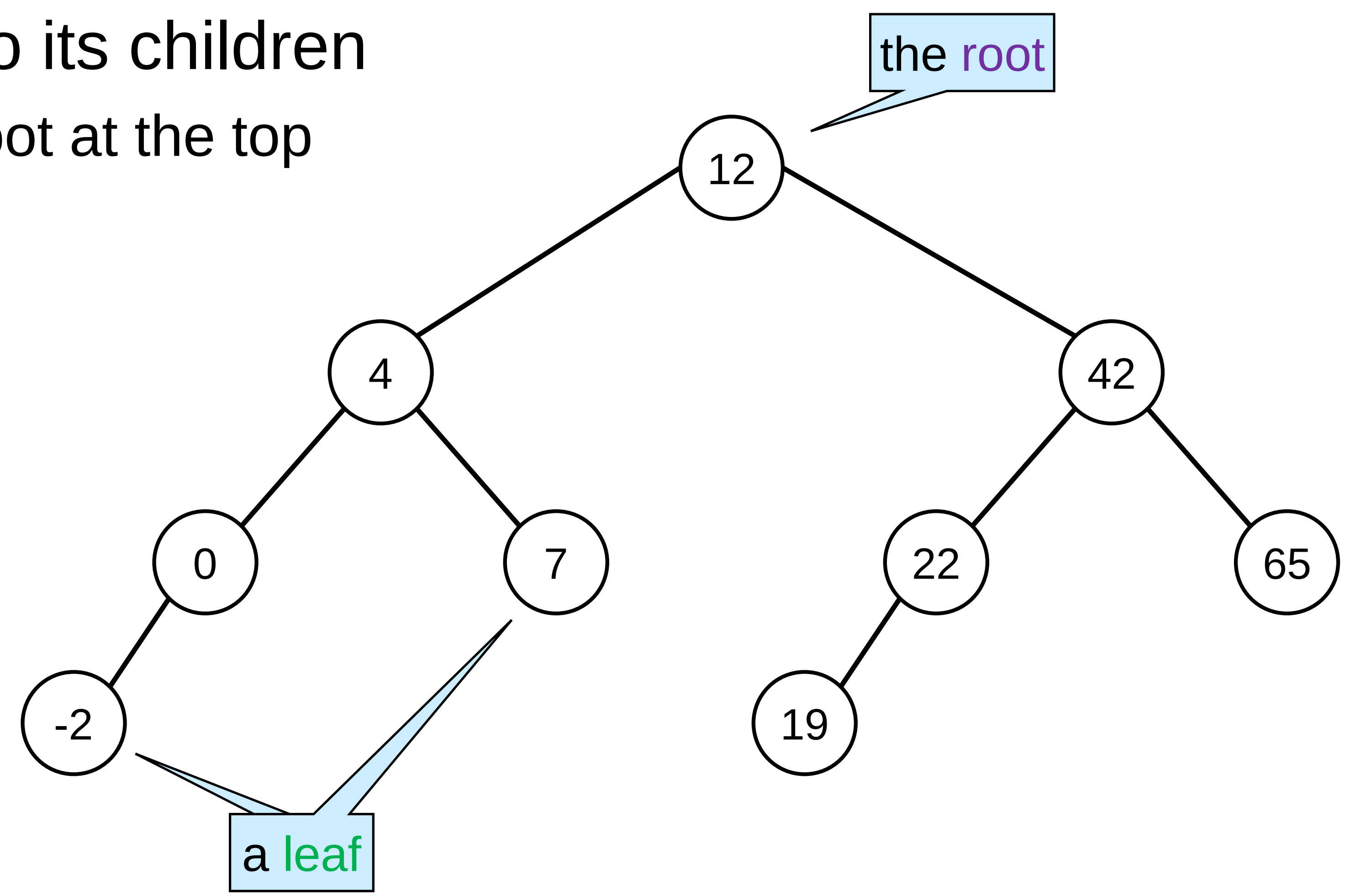
- Given any node
  - the node to its left is its **left child**
  - the node to its right is its **right child**
  - the node above it is its **parent**

.. *and Computer Science mixes botanical trees and family trees!*



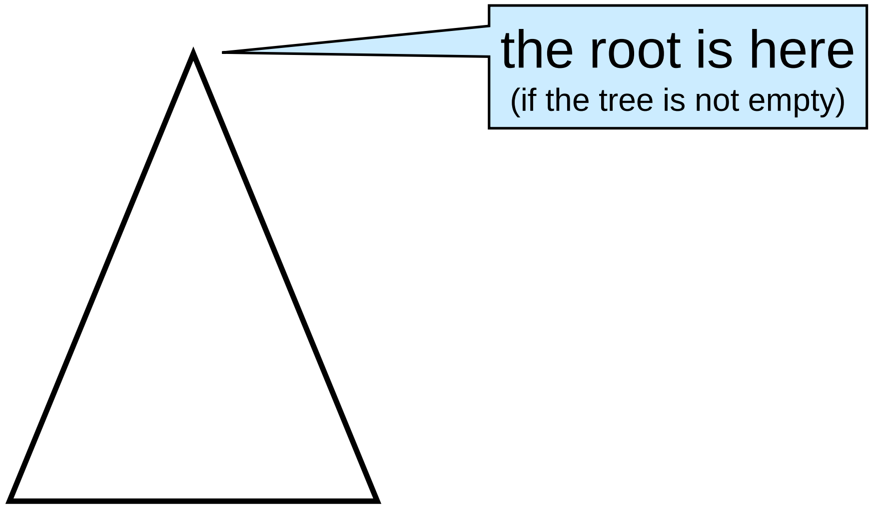
Concrete Tree Diagrams

- 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

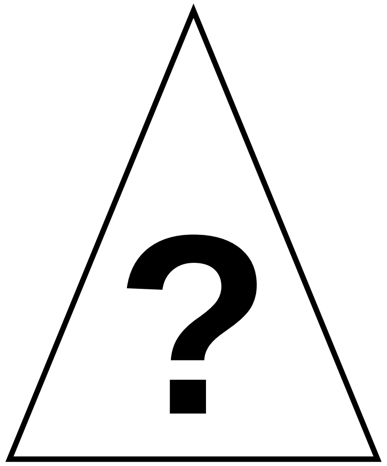
- We will often reason about trees that are arbitrary
  - their actual content is unimportant, so we abstract it away
  - We draw a generic tree as a triangle



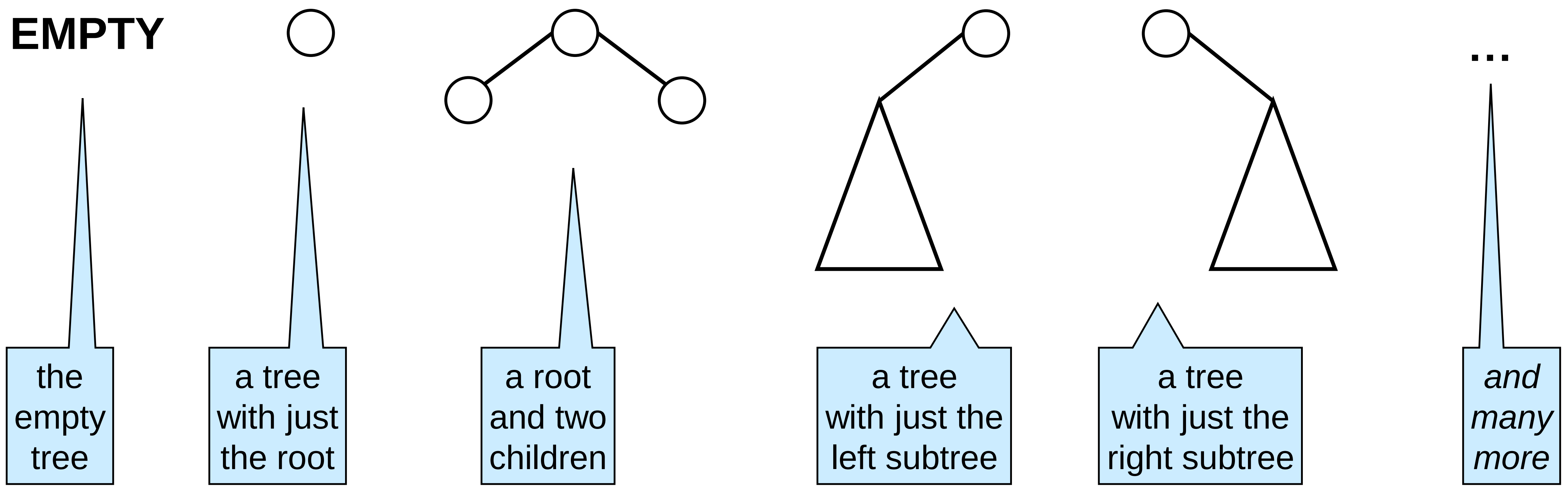
  - We represent the empty tree by simply writing "Empty"

Empty

What do Trees Look Like?



- Abstract trees come in many shapes

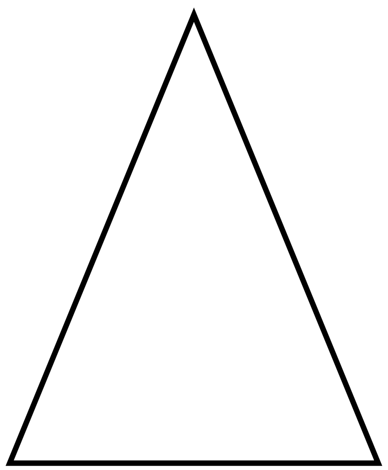


- When working with trees, we need to account for all these possibilities
  - we will forget some

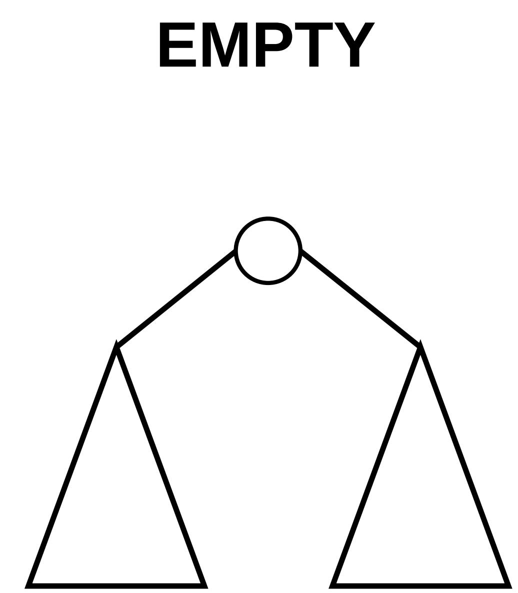
- *Is there a simpler description?*

What Trees Look Like

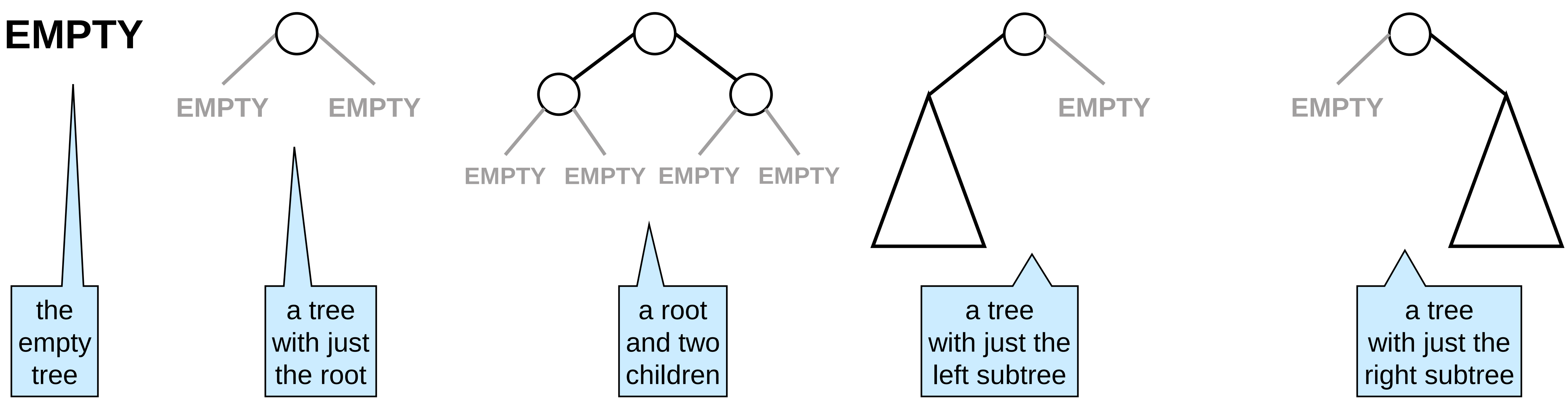
- A tree can be



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

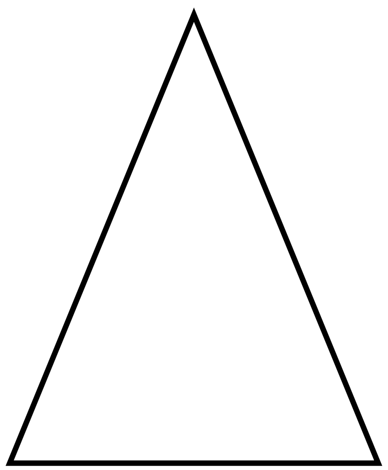


- **Every tree** reduces to these two cases



What Trees Look Like

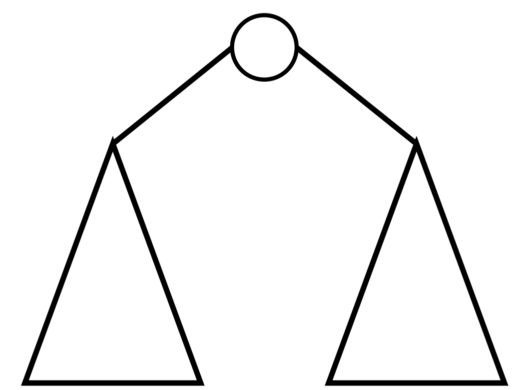
- *A tree can be*



  - *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

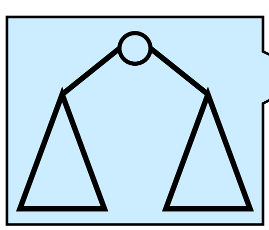
A Minimal Tree Invariant

- *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



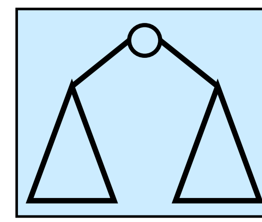
```
bool is_tree(tree* T) {
  // Code for empty tree
  if (T == NULL) return true;

  // 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

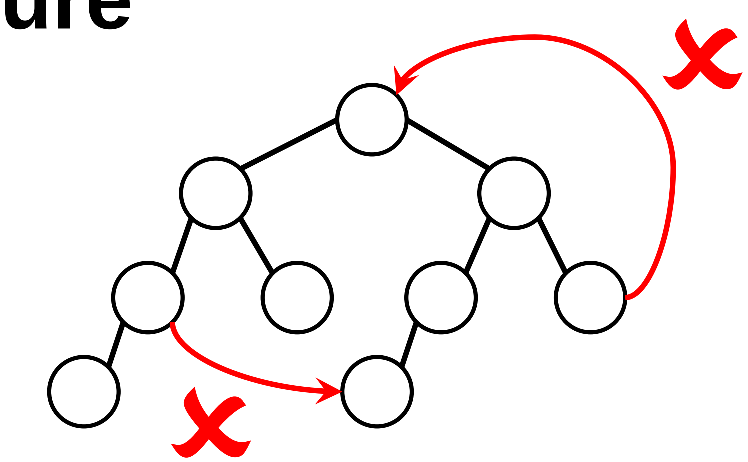
EMPTY



```
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



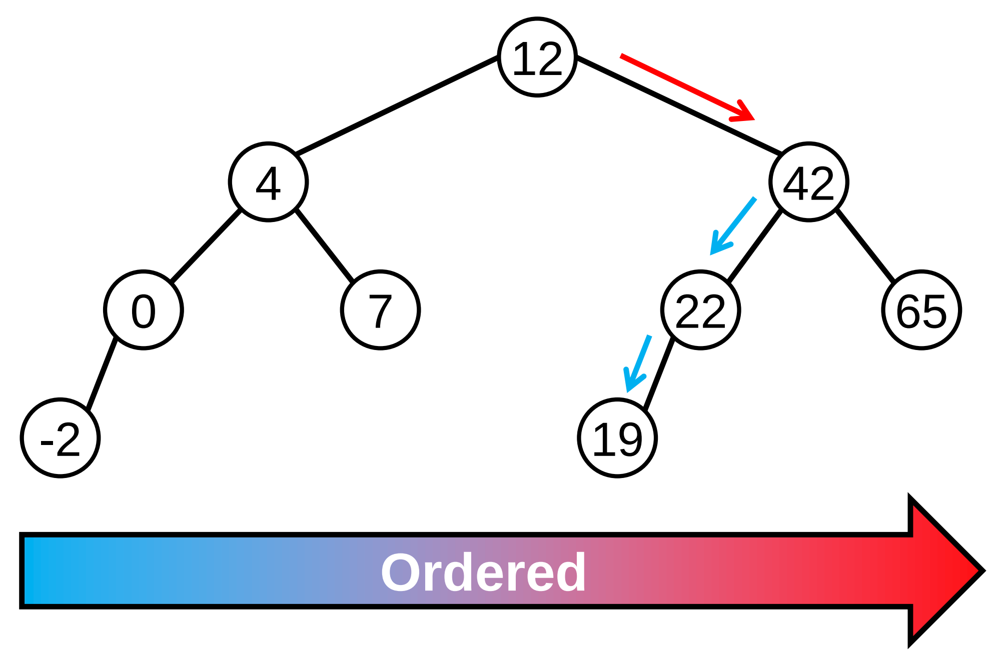
- *What additional constraints on contents do we need to use trees to implement **dictionaries**?*

Binary Search Trees

Binary Search Trees

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

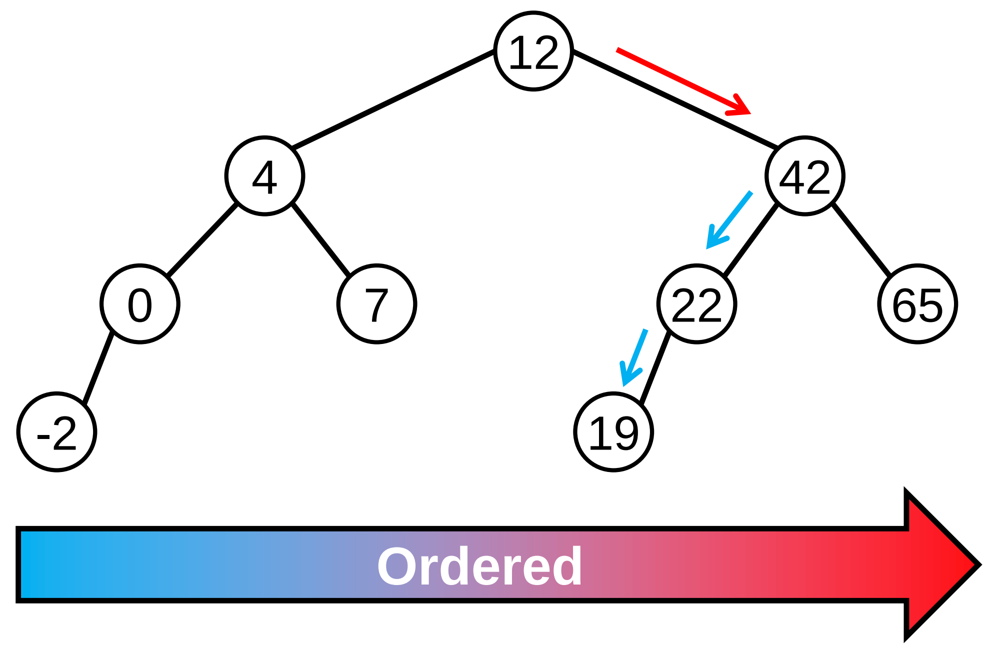
- Because lookup emulates binary search, the data in the tree need to be **ordered**
  - smaller values on the left
  - bigger values on the right



- 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 …

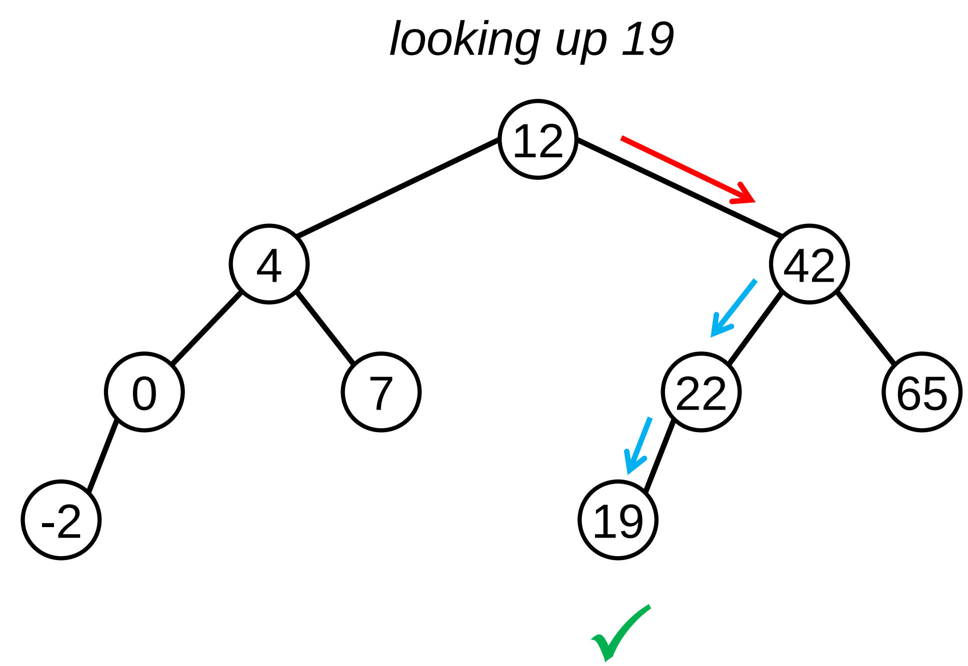
… whose nodes are ordered

We will see later
how to implement this

Looking Up Keys

Implementing lookup

- Leverage the structure of the tree!
  - empty: the key is not found
  - non-empty:
    - if root contains the key, found
    - if key is smaller than the root's go left
    - if key is bigger than the root's go right

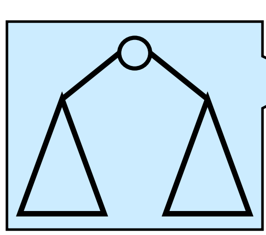


- In code:

```
entry bst_lookup(tree* T, key k)
//@requires is_bst(T);
{
  // Code for empty tree
  if (T == NULL) return NULL;

  // Code for non-empty tree
  if (k == T->data) return T->data;
  if (k < T->data) return bst_lookup(T->left, k);
  //@assert k > T->data;
  return bst_lookup(T->right, k);
}
```

lookup works only on BST's

EMPTY

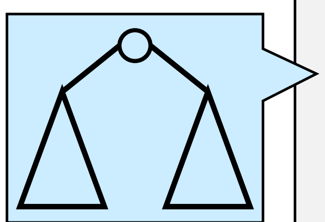


we go left by recursing on the left subtree

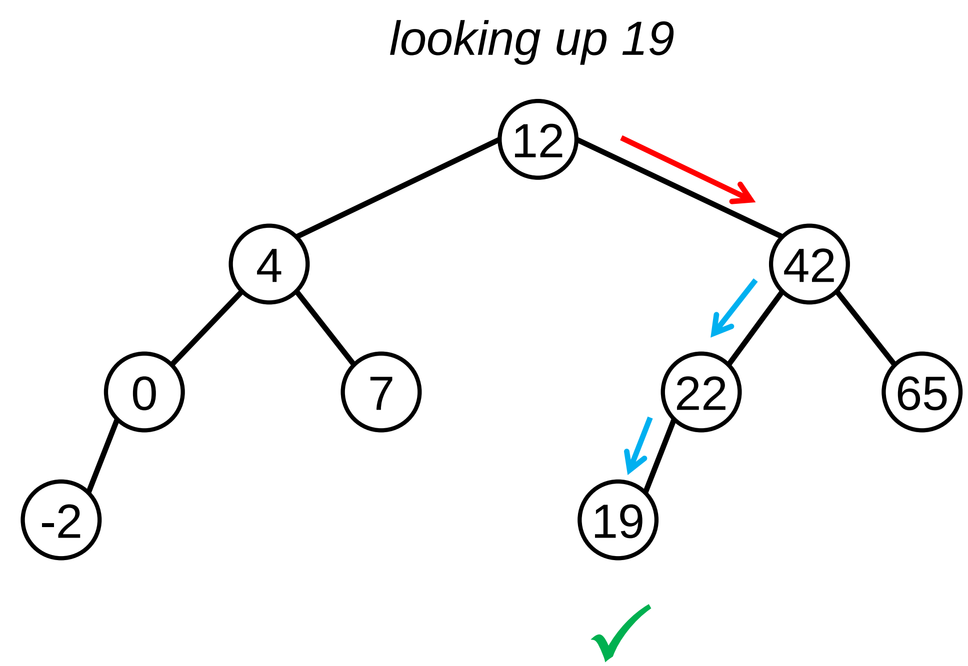
we go right by recursing on the right subtree

Implementing lookup

EMPTY



```
entry bst_lookup(tree* T, key k)
//@requires is_bst(T);
{
  // Code for empty tree
  if (T == NULL) return NULL;

  // Code for non-empty tree
  if (k == T->data) return T->data;
  if (k < T->data) return bst_lookup(T->left, k);
  //@assert k > T->data;
  return bst_lookup(T->right, k);
}
```



- *But < and > work only for integers!*
  - *also, keys and entries are not the same thing in general*
- We want a dictionary that uses trees
  - to store **entries of any type**
  - and look them up using **keys of any type**

Implementing lookup

- *But < and > work only for integers!*
  - *also, keys and entries are not the same thing in general*
- *We want a dictionary that uses trees*
  - *to store **entries of any type*** entry
  - *and look them up using **keys of any type*** key

just like for hash dictionaries

- We need functions that
  - extract the key from an entry: entry_key
  - compare two keys: key_compare
- It is for the client to decide on the type of keys and entries
  - So the client shall provide these functions

A Client Interface

- The BST dictionary needs a **client interface** that
  - requests the client to provide types entry and key
  - declares a function to extract the key of an entry
  - declares a function to compare two keys

Client Interface

```
// typedef ______* entry;
// typedef ______ key;

key entry_key(entry e)
  /*@requires e != NULL;                    @*/ ;

int key_compare(key k1, key k2) ;
```

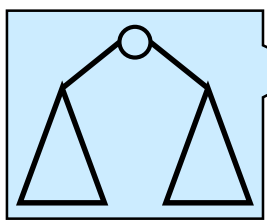
returns
- **< 0** if k1 is smaller than k2
- **0** if k1 and k2 are the same
- **> 0** if k1 is larger than k2

- *We could make it fully generic*
  - *but let's keep things simple*

Implementing lookup

- With it, we can write a general implementation

EMPTY



```
entry bst_lookup(tree* T, key k)
//@requires is_bst(T);
//@ensures \result == NULL
           || key_compare(entry_key(\result), k) == 0;
{
  // Code for empty tree
  if (T == NULL) return NULL;

  // Code for non-empty tree
  int cmp = key_compare(k, entry_key(T->data));
  if (cmp == 0) return T->data;
  if (cmp < 0) return bst_lookup(T->left, k);
  //@assert cmp > 0;
  return bst_lookup(T->right, k);
}
```

We save the outcome of the comparison in the variable cmp

- **< 0** if k1 is smaller than k2
- **0** if k1 and k2 are the same
- **> 0** if k1 is larger than k2

found!

go left

go right

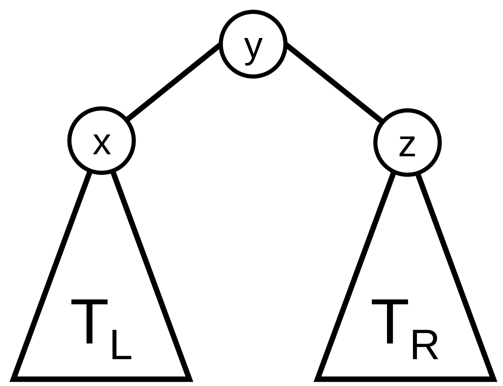
- We can now even provide a useful postcondition
  - either lookup returns NULL
    - no entry in T has key k
  - or the key of the returned entry is the same as k

just like for hash dictionaries

Checking Ordering

Ordered Trees – I

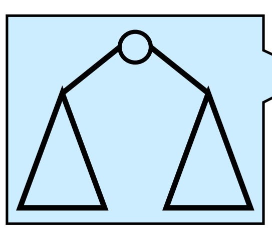
- The data in every node must be
  - bigger than its left child's
  - smaller than its right child



x < y < z

- In code:

EMPTY



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

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

The empty tree is ordered

If T has a left child, it must be smaller

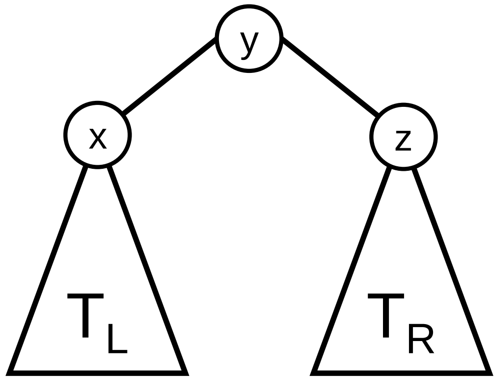
The left subtree must be ordered

and similarly on the right

For simplicity, assume int data

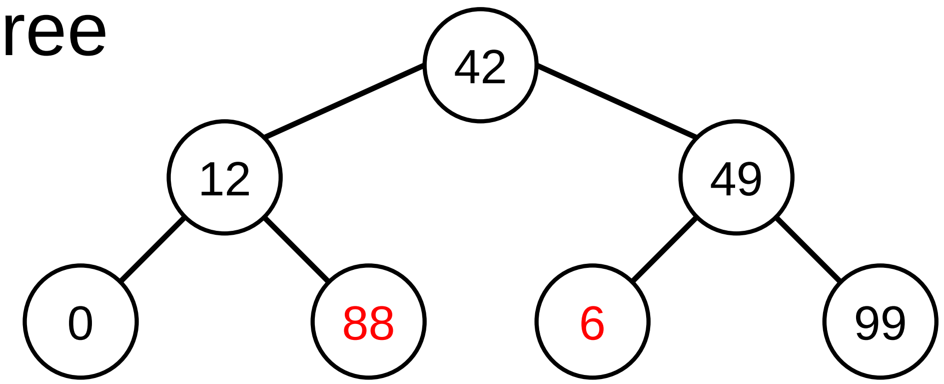
Ordered Trees – I

- *The data in every node must be*
  - *bigger than its left child's*
  - *smaller than its right child*



$x < y < z$

- *Is this enough?*
  - This is true of this tree



lookup cannot find 88 and 6

  - but it is **not** ordered

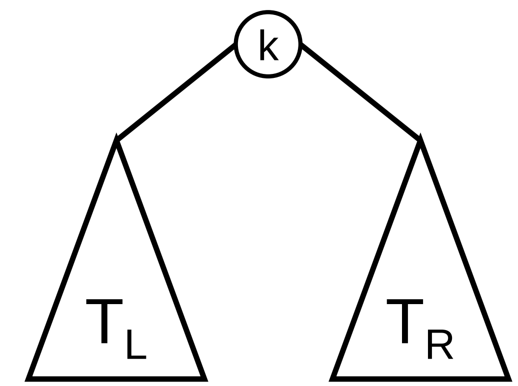
- To be ordered, we want $T_L < y < T_R$
  - not $x < y < z$

This is a **global** constraint:
we need to check the whole subtrees

This is a ***local*** constraint:
it only checks the children of each node

Ordered Trees – II

- The data in every node must be
  - bigger than **everything in its left subtree**
  - smaller than **everything in its right subtree**

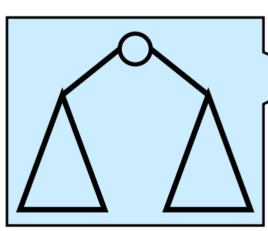


$T_L < k < T_R$

- We need two helper functions
  - gt_tree that checks $k > T_L$ (i.e., $T_L < k$)
  - lt_tree that checks $k < T_R$

```
bool gt_tree(key k, tree* T)    // checks that T < k
//@requires is_tree(T);
{
  // Code for empty tree
  if (T == NULL) return true;

  // Code for non-empty tree
  return key_compare (k, entry_key(T->data)) > 0
      && gt_tree(k, T->left)
      && gt_tree(k, T->right);
}
```

EMPTY

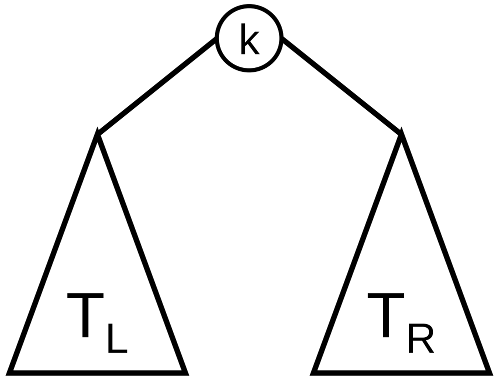


- gt_tree has cost O(n)
  - if T contains n nodes
  - because it compares k with every node in T

lt_tree is similar

Ordered Trees – II

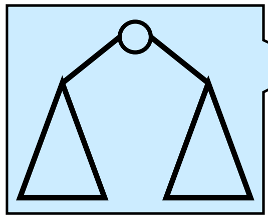
- *The data in every node must be*
  - *bigger than everything in its left subtree*
  - *smaller than everything in its right subtree*



$T_L < k < T_R$

- In code:

EMPTY



```
bool gt_tree(key k, tree* T) {…}   // O(n)
bool lt_tree(key k, tree* T) {…}   // O(n)

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

  // Code for non-empty tree
  key k = entry_key(T->data);
  return is_ordered(T->left)   && gt_tree(k, T->left)
      && is_ordered(T->right) && lt_tree(k, T->right);
}
```

- is_ordered costs $O(n^2)$
  - if T contains n nodes
  - because it calls gt_tree and lt_tree on each node

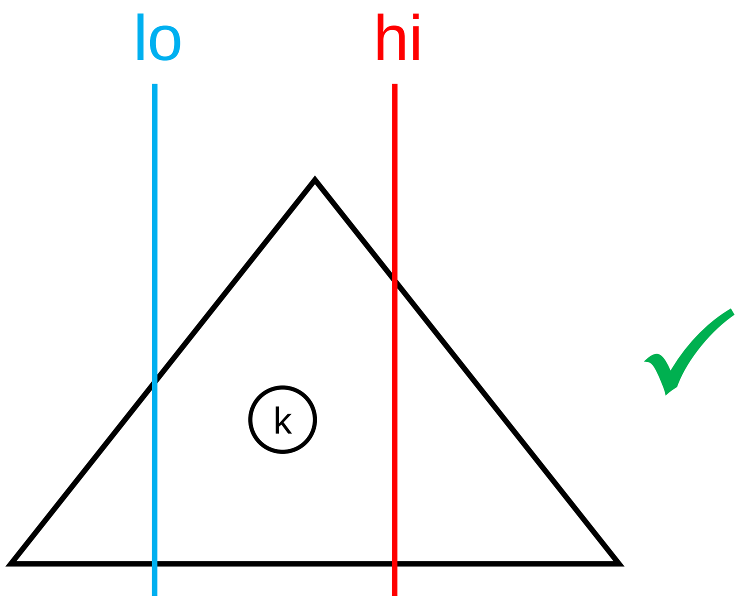
Ordered Trees – III

- *Can we do better than $O(n^2)$?*

Even though we typically don't care about the cost of specification functions

- As we examine each key k, keep track of its **allowed range**
  - if lo < k < hi, then
    - lo < k_L < k for the key k_L of its left child (if any)
    - k < k_R < hi for the key k_R of its right child (if any)
  - if k is the root, then –∞ < k < ∞

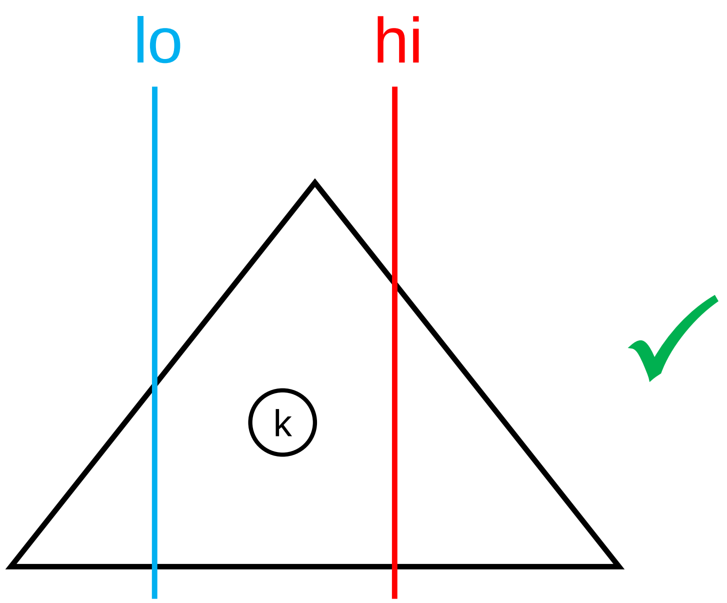
This assumes integer keys



- For arbitrary keys,
  - use *entries* as the bounds and entry_key to extract their key
  - use key_compare to compare k with another key
  - use NULL as –∞ and ∞

NULL is a value of type entry that is not a valid entry

Ordered Trees – III

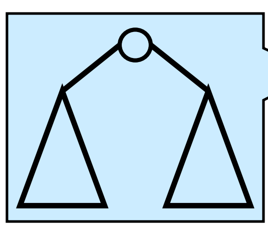


- *As we examine each key k, keep track of its allowed range*

- In code:

We carry around the range (lo, hi) as additional parameters

EMPTY



```
bool is_ordered(tree* T, entry lo, entry hi)
//@requires is_tree(T);
{
  // Code for empty tree
  if (T == NULL) return true;

  // Code for non-empty tree
  key k = entry_key(T->data);
  return (lo == NULL || key_compare(entry_key(lo), k) < 0)
      && (hi == NULL || key_compare(k, entry_key(hi)) < 0)
      && is_ordered(T->left, lo, T->data)
      && is_ordered(T->right, T->data, hi);
}
```

Check that $lo < k < hi$

Check that $lo < T_L < k$

Check that $k < T_R < hi$

- Complexity: O(n)
  - if T contains n nodes
  - we test every node in the tree

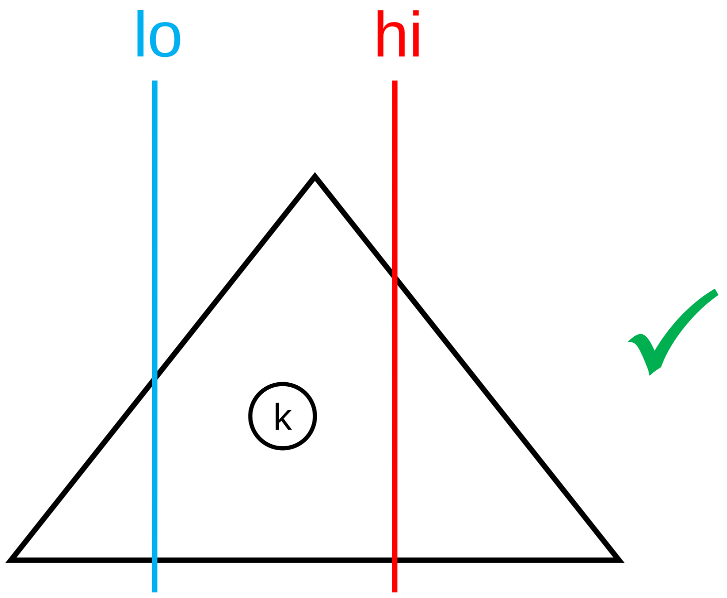
Ordered Trees – III

- We need to update is_bst slightly

```
bool is_ordered(tree* T, entry lo, entry hi) { … }

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

Initially
lo = $-\infty$

Initially
hi = ∞

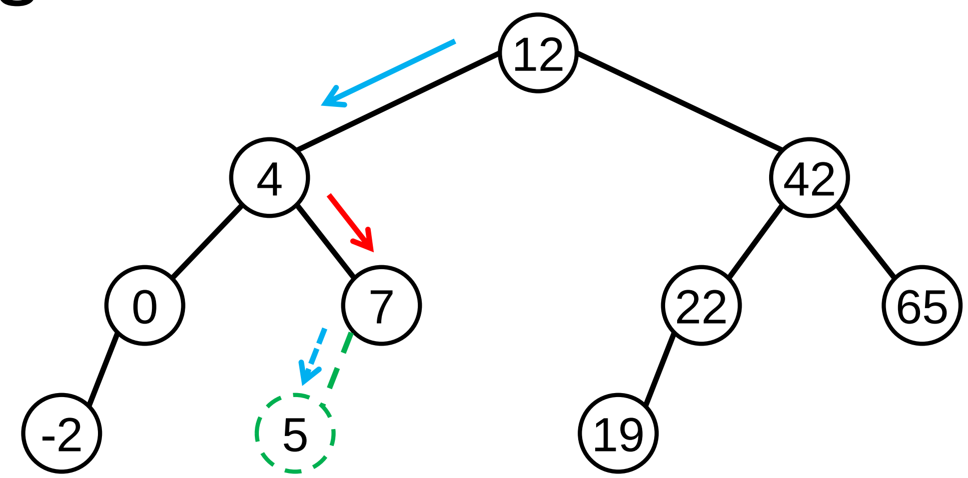


Inserting Entries

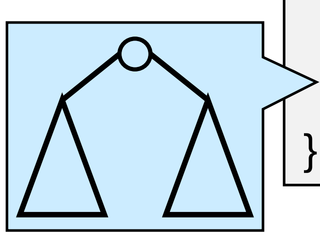
Inserting into a BST

- *Do the same steps we would do to search for this entry, and then put it where it should have been*
- The code follows the possible shapes of the tree

inserting 5



EMPTY



```
void bst_insert(tree* T, entry e)
//@requires is_bst(T) && e != NULL;
{
  // Code for empty tree
  …

  // Code for non-empty tree
   …
}
```

Inserting into an Empty BST

- We simply create a node for the new entry

inserting 5

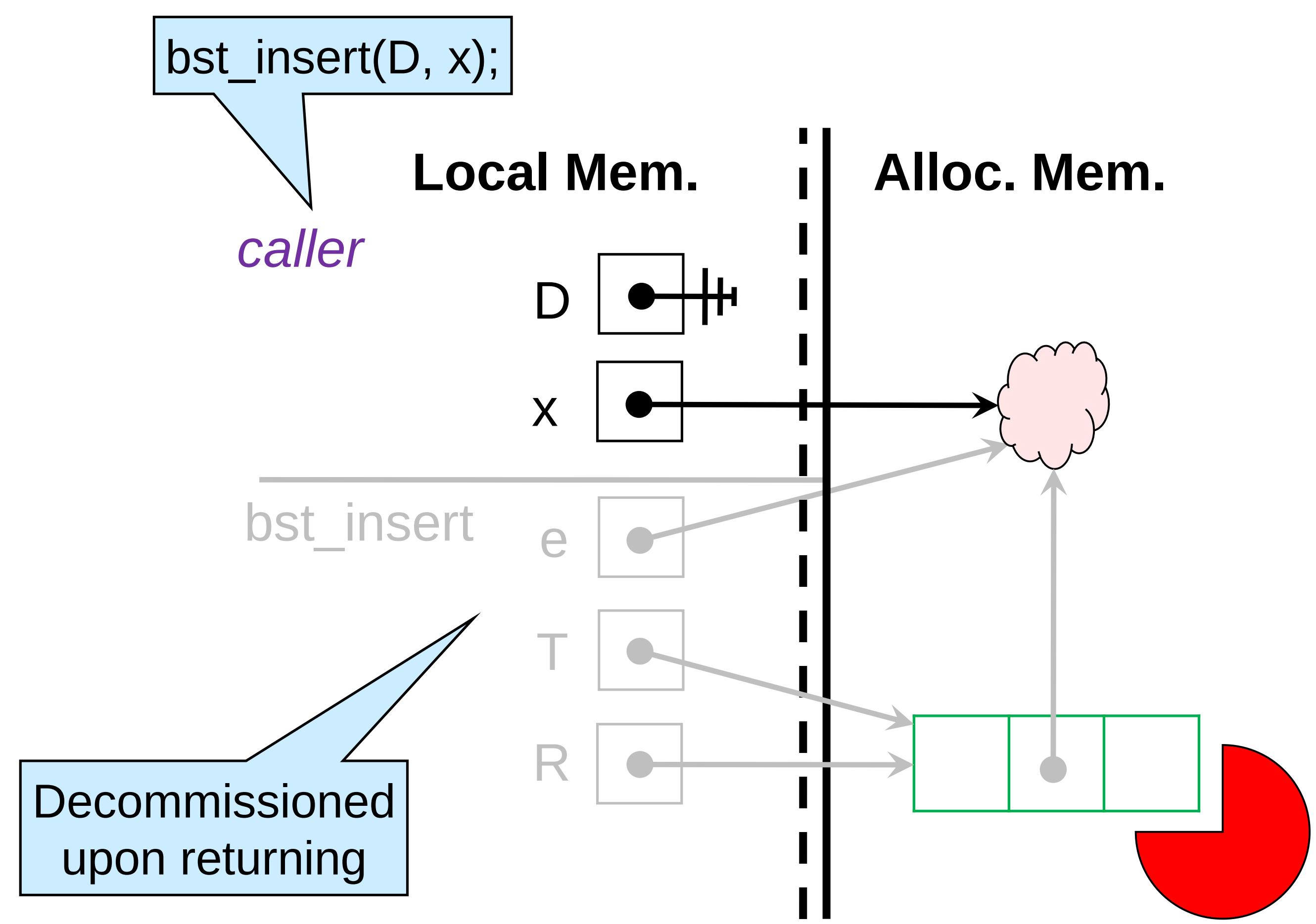
5

```
void bst_insert(tree* T, entry e)
//@requires is_bst(T) && e != NULL;
{
  // Code for empty tree
  if (T == NULL) {
    tree* R = alloc(tree);
    R->data = e;
    T = R;
  }

  // Code for non-empty tree
   …
}
```

EMPTY

Sets left and right to NULL

- Does this achieve what we want?
  - No: T is a **copy** of the caller's tree
    - changing T does not change the original



- We need to ***return*** the new node to the caller
  - bst_insert must return a tree

Inserting into an Empty BST

- We simply create a node for the new entry **and return it**

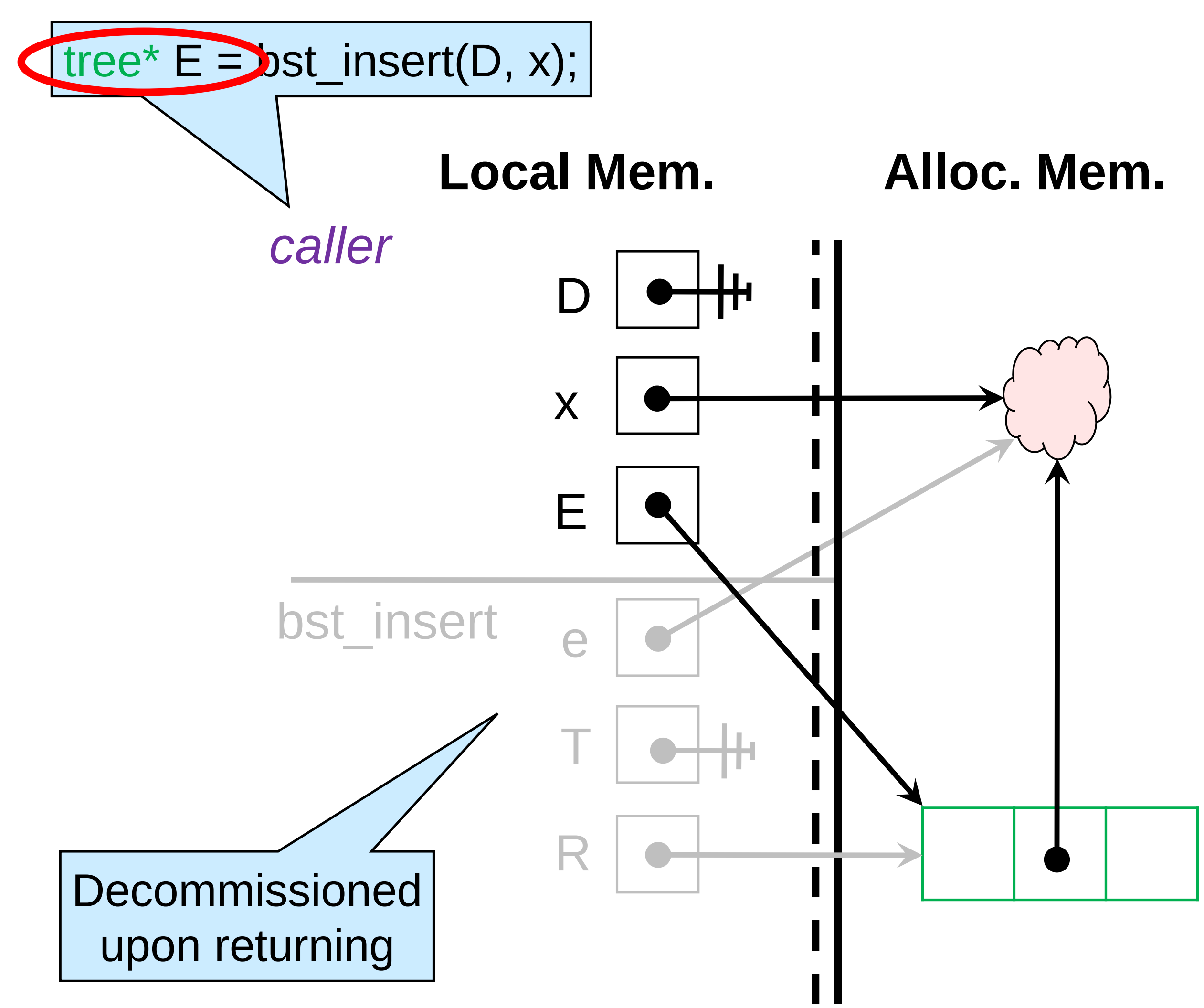
inserting 5

5

```
tree* bst_insert(tree* T, entry e)
//@requires is_bst(T) && e != NULL;
//@ensures is_bst(\result) && \result != NULL;
{
  // Code for empty tree
  if (T == NULL) {
    tree* R = alloc(tree);
    R->data = e;
    return R;
  }

  // Code for non-empty tree
   …
}
```

EMPTY

Sets left and right to NULL



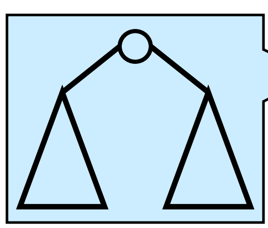
- The returned tree must be a valid BST

Inserting in a Non-empty BST

- If an entry with the same key is present, we overwrite it

```
tree* bst_insert(tree* T, entry e)
//@requires is_bst(T) && e != NULL;
//@ensures is_bst(\result) && \result != NULL;
//@ensures bst_lookup(\result, entry_key(e)) == e;
{
  // Code for empty tree
  if (T == NULL) {
    tree* R = alloc(tree);
    R->data = e;
    return R;
  }

  // Code for non-empty tree
  int cmp = key_compare(entry_key(e), entry_key(T->data));
  if (cmp == 0) T->data = e;
  else if (cmp < 0) T->left = bst_insert(T->left, e);
  else { //@assert cmp > 0;
    T->right = bst_insert(T->right, e);
  }
  return T;
}
```

EMPTY



Additional postcondition

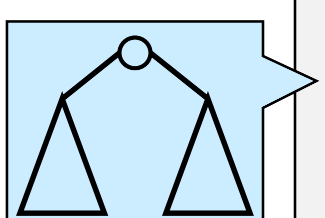
- **< 0** if k1 is smaller than k2
- **0** if k1 and k2 are the same
- **> 0** if k1 is larger than k2

We save the outcome of the comparison in the variable cmp

- When inserting in the left subtree, we **reattach** the tree returned by the recursive call
  - the pointer is the same except if it was NULL
- and similarly on the right

Inserting into a BST

EMPTY



```
tree* leaf(entry e)
//@requires e != NULL;
//@ensures is_bst(\result) && \result != NULL;
{
  tree* T = alloc(tree);
  T->data = e;
  T->left = NULL;     // not necessary
  T->right = NULL;    // not necessary
  return T;
}

tree* bst_insert(tree* T, entry e)
//@requires is_bst(T) && e != NULL;
//@ensures is_bst(\result) && \result != NULL;
//@ensures bst_lookup(\result, entry_key(e)) == e;
{
  // Code for empty tree
  if (T == NULL) return leaf(e);

  // Code for non-empty tree
  int cmp = key_compare(entry_key(e), entry_key(T->data));
  if (cmp == 0) T->data = e;
  else if (cmp < 0) T->left = bst_insert(T->left, e);
  else { //@assert cmp > 0;
    T->right = bst_insert(T->right, e);
  }
  return T;
}
```

- We make bst_insert more readable by
  - moving the code that creates a new leaf into a helper function
  - explicitly setting its children to NULL

Refactoring code to make it more readable is important for **maintainability**

BST Dictionaries

Are we There Yet?

- Our target dictionary interface is

Like hash dictionaries

Library Interface

```
// typedef ______* dict_t;

dict_t  dict_new()
  /*@ensures \result != NULL;                              @*/ ;

entry dict_lookup(dict_t D, key k)
  /*@requires D != NULL;                                   @*/
  /*@ensures \result == NULL
          || key_compare(entry_key(\result), k) == 0;      @*/ ;

void dict_insert(dict_t D, entry e)
  /*@requires D != NULL && e != NULL;                      @*/
  /*@ensures dict_lookup(D, entry_key(e)) == e;            @*/ ;

entry dict_min(dict_t D)
  /*@requires D != NULL;                                   @*/ ;
```

… plus find_min

with this client interface

Client Interface

```
// typedef ______* entry;
// typedef ______ key;

key entry_key(entry  e)
  /*@requires e != NULL;                 @*/ ;

int key_compare(key k1, key k2) ;
```

- So far, we have implemented lookup and insertion

Are we There Yet?

```
entry bst_lookup(tree* T, key k);
tree* bst_insert(tree* T, entry e);
```

?

Library Interface

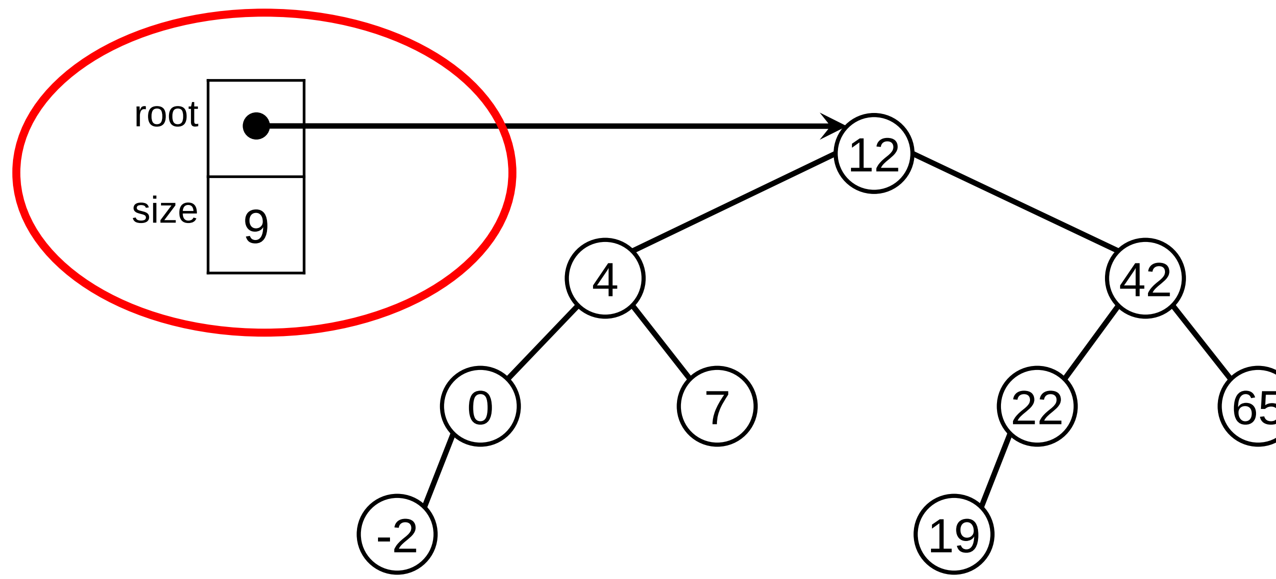
```
// typedef ______* dict_t;

dict_t dict_new()
  /*@ensures \result != NULL;                                   @*/ ;

entry dict_lookup(dict_t D, key k)
  /*@requires D != NULL;                                        @*/
  /*@ensures \result == NULL
             || key_compare(entry_key(\result), k) == 0;        @*/ ;

void dict_insert(dict_t D, entry e)
  /*@requires D != NULL && e != NULL;                           @*/
  /*@ensures dict_lookup(D, entry_key(e)) == e;                 @*/ ;

entry dict_min(dict_t D)
  /*@requires D != NULL;                                        @*/ ;
```

- They do not match!
  - bst_insert returns a tree* but dict_insert does not return anything
  - NULL is a valid BST but not a valid dictionary

Implementing BST Dictionaries

- We can define a **header** that contains a pointer to a tree
  - and possibly other data

```
struct dict_header {
  tree* root;
  int size;  // example of other data
};
typedef struct dict_header dict;
```



- and **wrappers** around the BST functions
  - ➢ they mediate between trees and dicts
  - Here's the specification function for BST dictionaries

```
bool is_dict(dict* D) {
  return D != NULL
      && is_bst(D->root);
}
```

ignoring other data

- the dictionary itself can't be NULL
  - ➢ this satisfies the dictionary interface
- but the underlying BST can
  - ➢ that's how we represent the empty dictionary

Implementing BST Dictionaries

```
struct dict_header {
  tree* root;
  int size;  // example of other data
};
typedef struct dict_header dict;
```

- We define **wrappers** around the BST functions
  - they mediate between the trees and dicts

Lookup

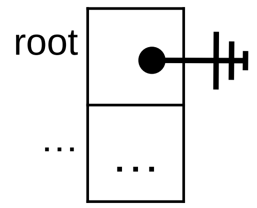
```
entry dict_lookup(dict* D, key k)
//@requires is_dict(D);
//@ensures \result == NULL
               || key_compare(entry_key(\result), k) == 0;
{
  return bst_lookup(D->root, k);
}
```

Insertion

```
void dict_insert(dict* D, entry e)
//@requires is_dict(D) && e != NULL;
//@ensures dict_lookup(D, entry_key(e)) == e;
//@ensures is_dict(D);
{
  D->root = bst_insert(D->root, e);
}
```

- Creating a dictionary
  - allocates a header and
  - sets the root to the empty BST

```
dict* dict_new()
//@ensures is_dict(\result);
{
  dict* D = alloc(dict);
  D->root = NULL;
  return D;
}
```



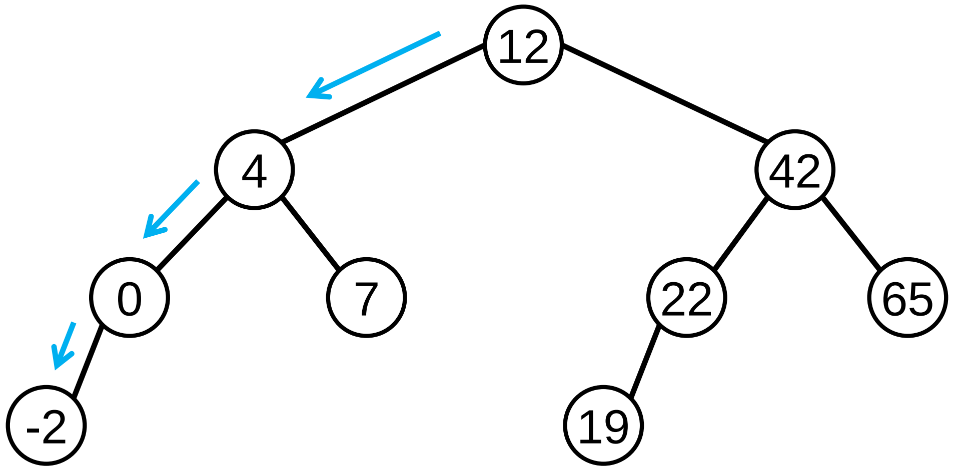
dict_new creates the empty dictionary

Implementing BST Dictionaries

```
struct dict_header {
  tree* root;
  int size;  // example of other data
};
typedef struct dict_header dict;
```

- We are only left with implementing find_min

```
entry dict_min(dict* D)
//@requires is_dict(D);
{
  if (D->root == NULL) return NULL;
  tree* T = D->root;
  while (T->left != NULL)
    T = T->left;
  return T->data;
}
```



- The abstract client dict_t is just dict*

```
typedef dict* dict_t;
```

- That's it! ✓

The BST Dictionary Library

Implementation

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

// Representation invariant
bool is_bst (tree* T)  { … }

// BST auxiliary functions
entry bst_lookup(tree* T, key k)
//@requires is_bst(T);
//@ensures \result == NULL
              || key_compare(entry_key(\result), k) == 0;
{ … }


tree* bst_insert(tree* T, entry e)
//@requires is_bst(T) && e != NULL;
//@ensures is_bst(\result) && \result != NULL;
//@ensures bst_lookup(\result, entry_key(e)) == e;
{ … }

// Implementing the dictionary
// Concrete type
struct dict_header {
  tree* root;
};
 typedef struct dict_header dict;

// Representation invariant
bool is_dict (dict* D) {
  return D != NULL && is_bst(D->root);
}
```

```
// Implementation of interface functions
dict* dict_new()
//@ensures is_dict(\result);
{
  dict* D = alloc(dict);
  D->root = NULL;
  return D;
}

entry dict_lookup(dict* D, key k)
//@requires is_dict(D);
//@ensures \result == NULL
              || key_compare(entry_key(\result), k) == 0;
{
  return bst_lookup(D->root, k);
}

void dict_insert(dict* D, entry e)
//@requires is_dict(D) && e != NULL;
//@ensures dict_lookup(D, entry_key(e)) == e;
//@ensures is_dict(D);
{
  D->root = bst_insert(D->root, e);
}

entry dict_min(dict* D)
//@requires is_dict(D);
{
  if (D->root == NULL) return NULL;
  tree* T = D->root;
  while (T->left != NULL)
    T = T->left;
  return T->data;
}

// Client type
typedef dict* dict_t;
```

Client Interface

```
// typedef ______* entry;
// typedef ______ key;

key entry_key(entry  e)
  /*@requires e != NULL;                @*/ ;

int key_compare(key k1, key k2) ;
```

Library Interface

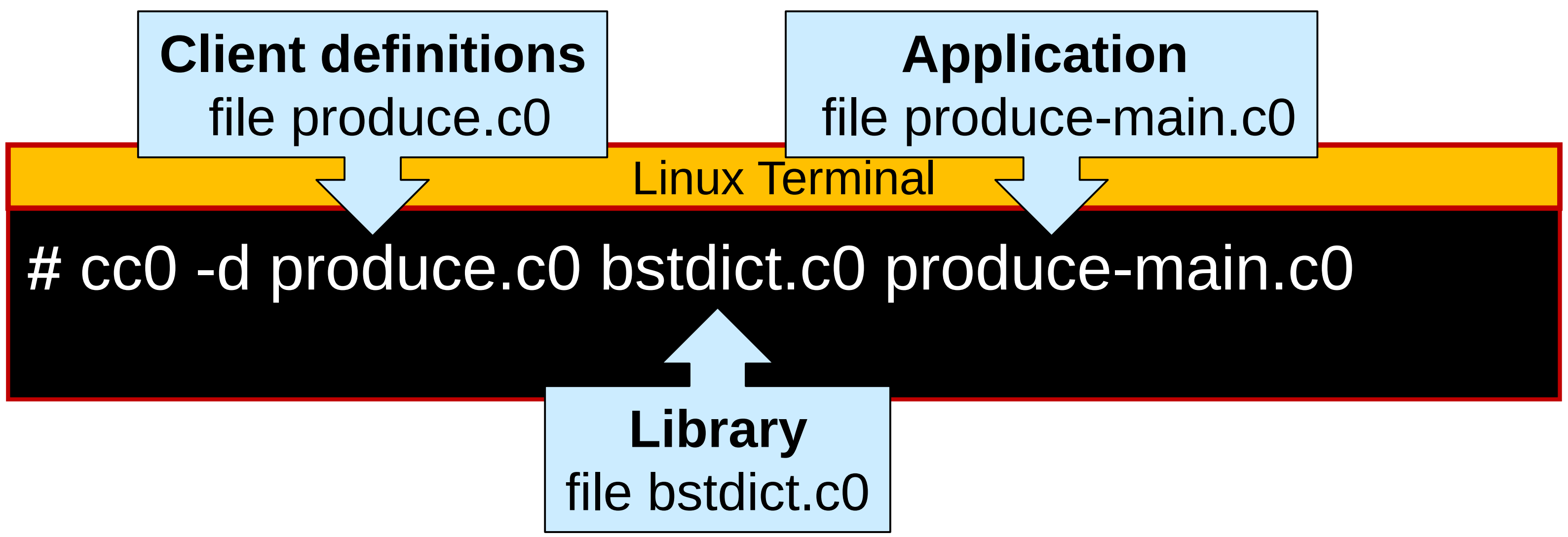
```
// typedef ______* dict_t;

dict_t  dict_new()
  /*@ensures \result != NULL;                        @*/ ;

entry dict_lookup(dict_t D, key k)
  /*@requires D != NULL;                              @*/
  /*@ensures \result == NULL
            || key_compare(entry_key(\result), k) == 0;  @*/ ;

void dict_insert(dict_t D, entry e)
  /*@requires D != NULL && e != NULL;                 @*/
  /*@ensures hdict_lookup(D, entry_key(e)) == e;      @*/ ;

entry dict_min(dict_t D)
  /*@requires D != NULL;                              @*/ ;
```

How

What

Using BST Dictionaries

- We can now use this new implementation of dictionaries for our application
  - once we write an appropriate client definition file



- We could easily make this library fully generic

Recall our Goal

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

- We have succeeded

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

○ *or have we …*