CS 112

Introduction to Data Structures

WEEK 11

Binary Search Trees and the STL set / map

Eric Araújo

Calvin University · Fall 2026

This Week

1. Binary search as a shape
2. The BST property
3. insert, contains, traversals
4. Shape decides everything **KEY**
5. remove: the three cases
6. The STL set and map
7. Choosing a container

Seven Guesses

I'm thinking of a number between 1 and 128. You get seven guesses.

- Guessing at random: under a 6% chance
- Trying 1, 2, 3, … up to 128 guesses
- Guessing the **middle** each time: always wins, in 7

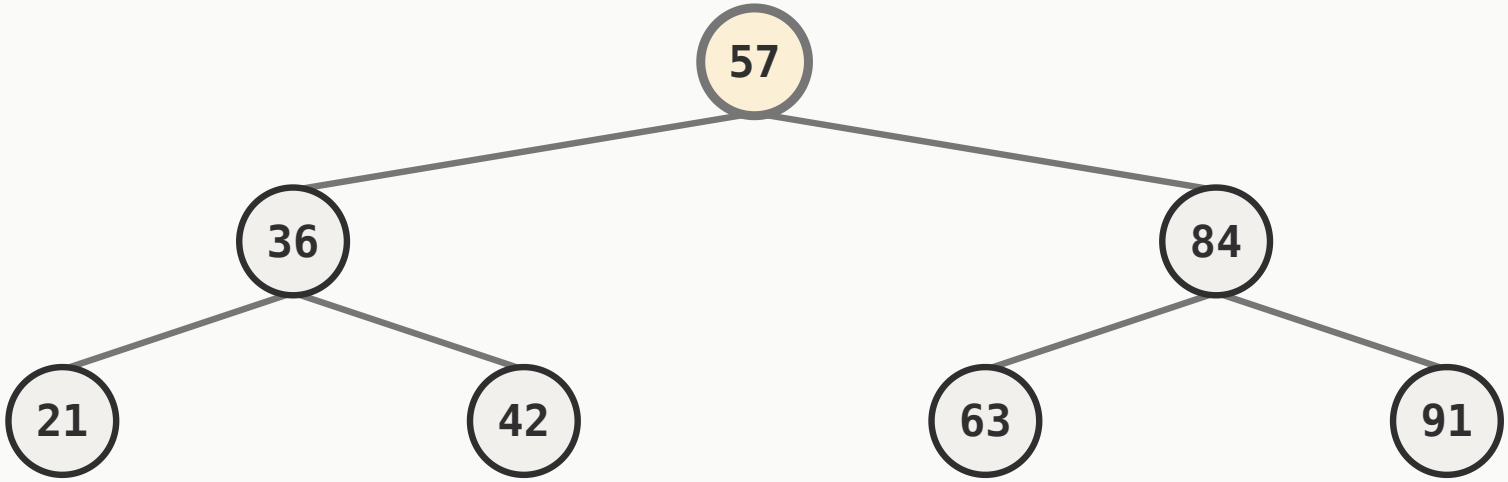
> Halving the search space each time is O(lg n): $\log_2 128 = 7$. The catch is that binary search needs the data **sorted** and **indexable**: an array, not a linked list.

What If the Structure Did the Halving?

A sorted array gives fast search but slow insertion. A linked list gives fast insertion but no halving.

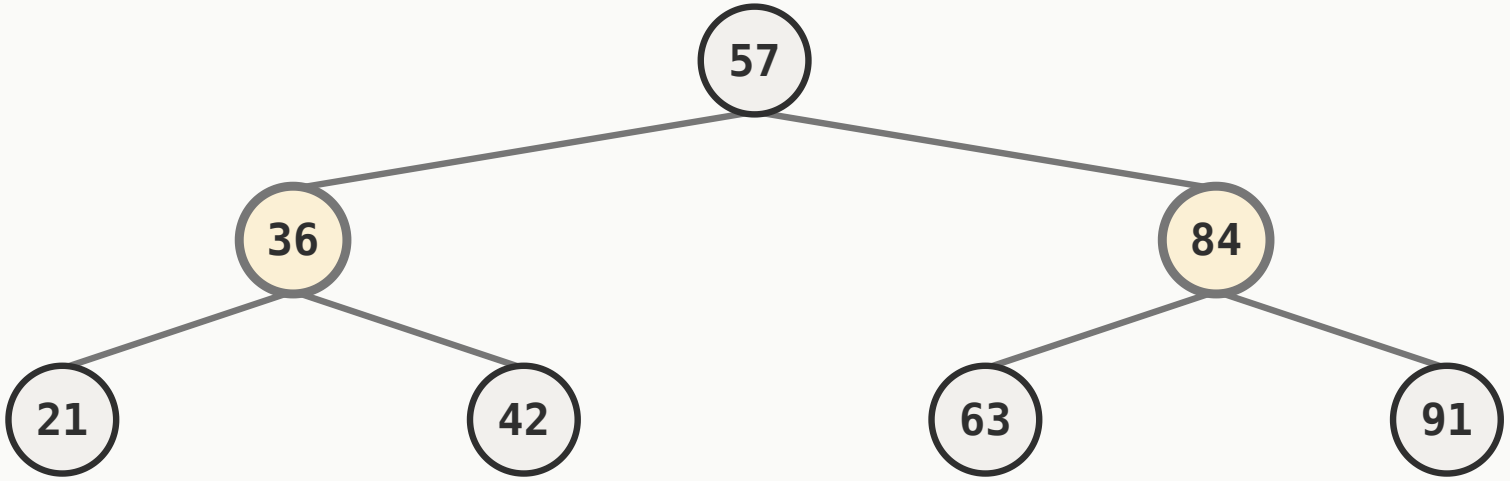
> We want both: **every step should cut the remaining possibilities in half**, and inserting shouldn't shift anything. That structure is a binary search tree.

Tree Vocabulary



- **root**: the top node, no parent (57)
- **leaf**: a node with no children (21, 42, 63, 91)
- **interior**: has 1 or 2 children (36, 84)
- **height**: links from the root to the deepest node (2 here)

The BST Property



For **every** node: everything in its left subtree is smaller, everything in its right subtree is larger. Not just its children, *everything below it*.

Two Classes Again

```
class Node {
  Item  myItem;
  Node *myLeft;
  Node *myRight;
};
```

```
class BST {
 public:
  void insert(Item it);
  bool contains(Item it);
 private:
  Node *myRoot;
  unsigned mySize;
};
```

> Same shape as the linked list: a public container hiding a private node type. A node now has **two** pointers instead of one; that's the entire structural difference.

insert

```
void Node::insert(Item it) {
    if (it < myItem) {
        if (myLeft == nullptr) { myLeft = new Node(it); }
        else                   { myLeft->insert(it); }
    } else {
        if (myRight == nullptr) { myRight = new Node(it); }
        else                    { myRight->insert(it); }
    }
}
```

> Compare, go left or right, recurse. New nodes always arrive as **leaves**: which is why the Node constructor sets both children to `nullptr`.

TALK TO YOUR NEIGHBOR · TTYN

Which insertion order builds the tree on the earlier slide (57 at the root)?

- **A.** 21, 36, 42, 57, 63, 84, 91
- **B.** 57, 36, 84, 21, 42, 63, 91
- **C.** 91, 84, 63, 57, 42, 36, 21
- **D.** 36, 21, 42, 57, 84, 63, 91

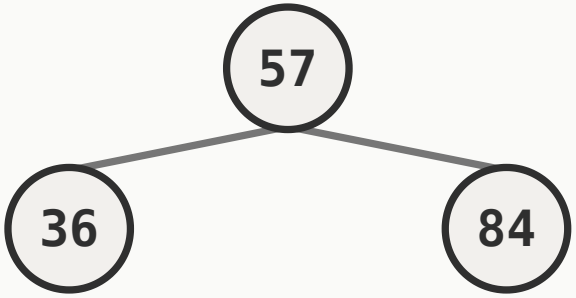
find / contains

```
bool Node::contains(Item it) const {
    if (it == myItem) { return true; }
    if (it < myItem) {
        return myLeft  != nullptr && myLeft->contains(it);
    }
    return myRight != nullptr && myRight->contains(it);
}
```

> One comparison eliminates an entire subtree, the guessing game, in code. Every step down discards roughly half of what's left.

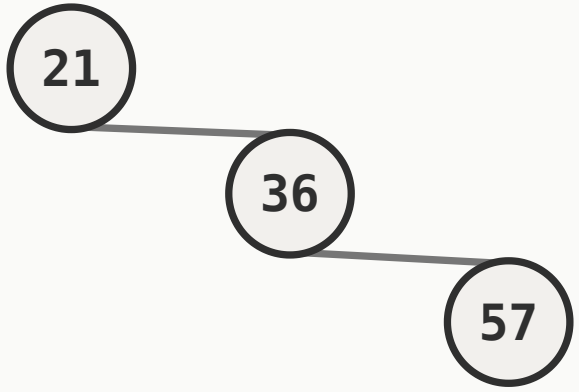
Shape Decides Everything

Balanced



height ≈ lg n → **O(lg n)**

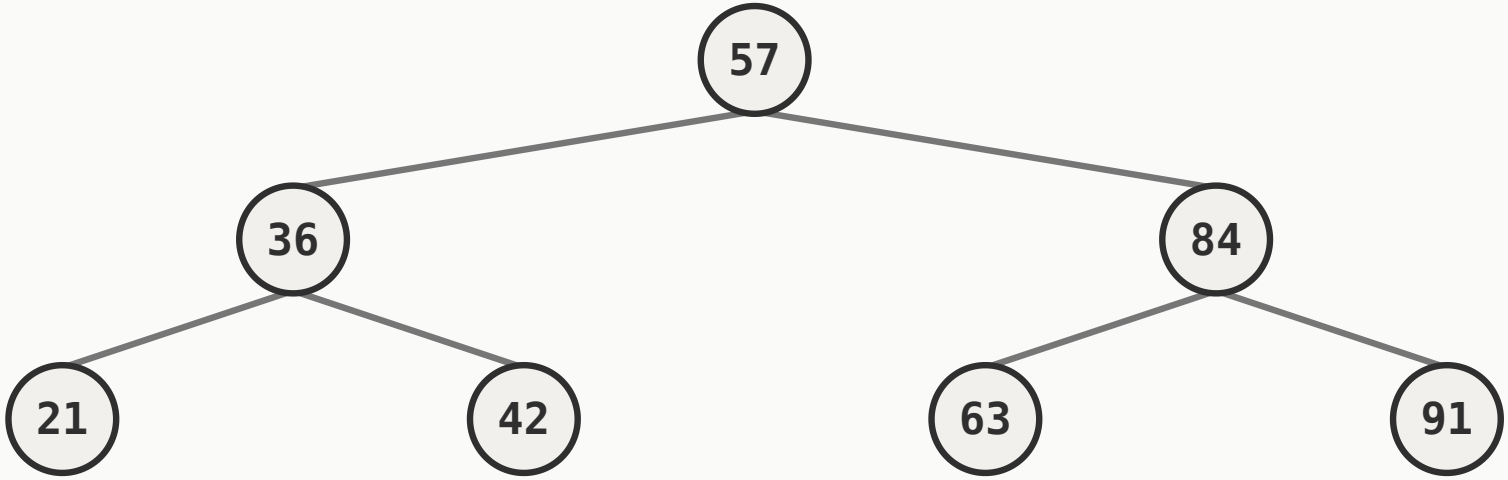
Degenerate



height = n − 1 → **O(n)**

> Insert sorted data and every node goes right: you have built a linked list with extra steps. Same code, same values, the *order they arrived* ruined it. Fixing that is next week.

Traversals



Order	Rule	Result
Pre-order	visit, left, right	57 36 21 42 84 63 91
In-order	left, visit, right	**21 36 42 57 63 84 91**
Post-order	left, right, visit	21 42 36 63 91 84 57

> In-order comes out **sorted**: that falls straight out of the BST property.

TALK TO YOUR NEIGHBOR · TTYN

Which traversal must a BST destructor use?

- **A.** Pre-order, delete the root first
- **B.** In-order, delete in sorted order
- **C.** Post-order, delete both children before the node
- **D.** Any of them

Height

```
int Node::height() const {
    int leftH  = (myLeft  == nullptr) ? -1 : myLeft->height();
    int rightH = (myRight == nullptr) ? -1 : myRight->height();
    return 1 + max(leftH, rightH);
}
```

- An empty tree has height −1; a single node, 0
- Computing it costs O(n): you have to visit everything
- A **full** tree of height h holds 2^(h+1) − 1 nodes

> That last line is the payoff: a million nodes fit in a balanced tree of height 20. Twenty comparisons to find anything.

remove: Three Cases

- **A leaf**: unlink it from its parent and delete. Easy.
- **One child**: hand the child up to the parent. Also easy.
- **Two children**: you can't just remove it; something must take its place.

> Which value can replace it without breaking the BST property? It must be larger than everything on the left and smaller than everything on the right.

TALK TO YOUR NEIGHBOR · TTYN

Removing the root 57 (children 36 and 84): which value replaces it?

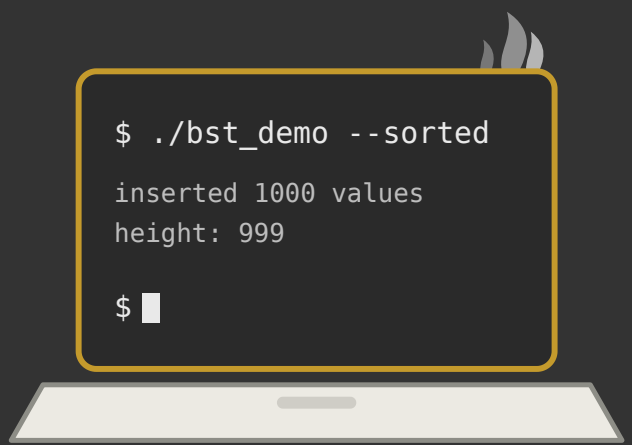
A. 36, the left child

B. 84, the right child

C. 42, the largest value in the left subtree

D. 21, the smallest value in the tree

Demo



Inserting sorted data into a BST.

(congratulations, you have reinvented the linked list)

Where BSTs Stand

Operation	Balanced BST	Degenerate BST	Sorted array
find	O(lg n)	O(n)	O(lg n)
insert	O(lg n)	O(n)	O(n)
remove	O(lg n)	O(n)	O(n)
in-order listing	O(n)	O(n)	O(n)

> A balanced BST matches the sorted array on lookups and beats it badly on changes. Everything depends on that word **balanced**: which is exactly what AVL trees guarantee.

You Built These by Hand

Dynamic array, linked list, stack, queue, binary search tree, self-balancing tree. The STL ships all of them.

> Knowing what's inside is the point of the last twelve weeks. From here on, use the library, but now you know *why* `set` is fast, and what it costs.

`set`: Unique, Sorted, Fast

```
#include <set>

set<string> enrolled;

enrolled.insert("ada");
enrolled.insert("grace");
enrolled.insert("ada");          // ignored: already there

cout << enrolled.size();         // 2
cout << enrolled.count("ada"); // 1  (0 or 1, never more)
```

> Two guarantees: **no duplicates**, and iteration comes out **in sorted order**. Both fall out of the balanced tree underneath.

What's Underneath

Operation	Complexity	Why
`insert`	O(lg n)	walk down the tree, rebalance
`count` / `find`	O(lg n)	the guessing game again
`erase`	O(lg n)	remove, then rebalance
iteration	O(n) total	in-order traversal

> A balanced BST with a friendly interface. Every cost on this slide is one you derived yourself in weeks 11 and 12.

Iterating

```
for (const string &name : enrolled) {      // sorted order
    cout << name << endl;
}

set<string>::iterator it = enrolled.find("ada");
if (it != enrolled.end()) {
    cout << *it << " is enrolled" << endl;
}
```

`find` returns `end()` when there's no match, comparing against `end()` is how you ask "was it there?". Never dereference `end()`; it points past the last element, not at one.

TALK TO YOUR NEIGHBOR · TTYN

What does this print?

```
set<int> s;
s.insert(7);
s.insert(3);
s.insert(7);
s.insert(9);
cout << s.size();
```

A. 3

B. 4

C. 2

D. It won't compile

`map`: Keys to Values

```
#include <map>

map<string, int> scores;

scores["ada"]   = 95;
scores["grace"] = 98;

cout << scores["ada"];          // 95
cout << scores.size();          // 2
```

> Python's dictionary, with two differences: the keys are kept **sorted**, and lookup is O(lg n) rather than average O(1).

Walking a `map`

```
for (const auto &entry : scores) {
    cout << entry.first << ": " << entry.second << endl;
}

// ada: 95
// grace: 98        ← always in key order
```

Each element is a `pair`: `.first` is the key, `.second` is the value. The keys come out sorted because a map is the same balanced tree, keyed on `.first`.

TALK TO YOUR NEIGHBOR · TTYN

What does `scores.size()` report?

```
map<string, int> scores;
scores["ada"] = 95;

if (scores["bob"] > 90) {     // just a lookup?
    cout << "bob did well";
}
cout << scores.size();
```

A. 1

B. 2

C. 0

D. It throws an exception

The `[]` Trap

Inserts if absent

```
m["missing"]     // creates it!
```

Asks without inserting

```
m.count("missing")
m.find("missing") != m.end()
```

> This bites everyone once: a loop that only *reads* a map quietly doubles its size. The behaviour is deliberate, `m[k]++` as a word counter depends on it, but it means `[]` is not a read-only operation.

A Word Counter

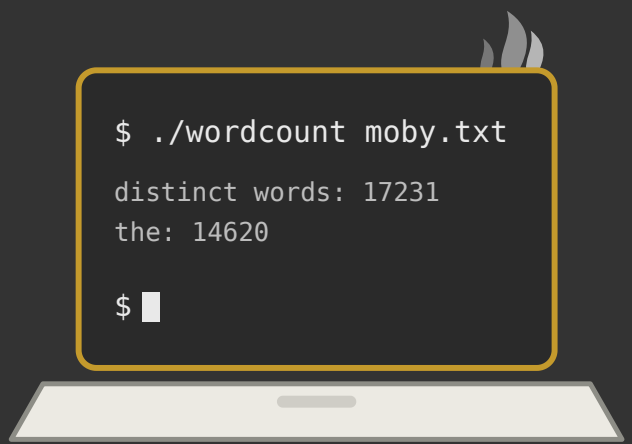
```
map<string, int> counts;
string word;

while (fin >> word) {
    ++counts[word];        // creates it at 0, then increments
}

for (const auto &entry : counts) {
    cout << entry.first << " " << entry.second << endl;
}
```

> Eight lines, alphabetised output, O(lg n) per word. Try writing that with an array of structs and you'll appreciate the container.

Demo



Counting words in a book with eight lines of code.

(spoiler: 'the' wins, and it is not close)

Ordered or Unordered?

	`set` / `map`	`unordered_set` / `unordered_map`
Underneath	balanced tree	hash table
find / insert	O(lg n)	O(1) average, O(n) worst
Iteration order	**sorted**	unspecified
Needs	`<` on the key	a hash function

> If you need sorted output or range queries, take the tree. If you only ever ask "is it there?", the hash table is faster, and that's next week.

Choosing a Container

- Index by position, mostly appending → `vector`
- Constant insert/remove at the ends → `deque`, `stack`, `queue`
- Membership, no duplicates, sorted → `set`
- Key to value, sorted keys → `map`
- Membership or lookup as fast as possible, order irrelevant → `unordered_*`

> The question is never "which container is best?"; it's "which operations will I do a million times?"

Week 11 Recap

- A BST puts smaller values left, larger right, **for every node**
- insert and find are the guessing game: compare, then discard half
- New nodes always arrive as **leaves**
- In-order traversal comes out **sorted**; the destructor must be **post-order**
- Balanced gives O(lg n); sorted input gives a list and O(n)
- Removing a two-child node: promote the max of the left subtree

Next: a14 →

This deck stands on earlier CS112 materials by
Joel Adams and **Victor Norman** · adapted and extended by **Eric Araújo**