Data Structures and Object-Oriented Design VII

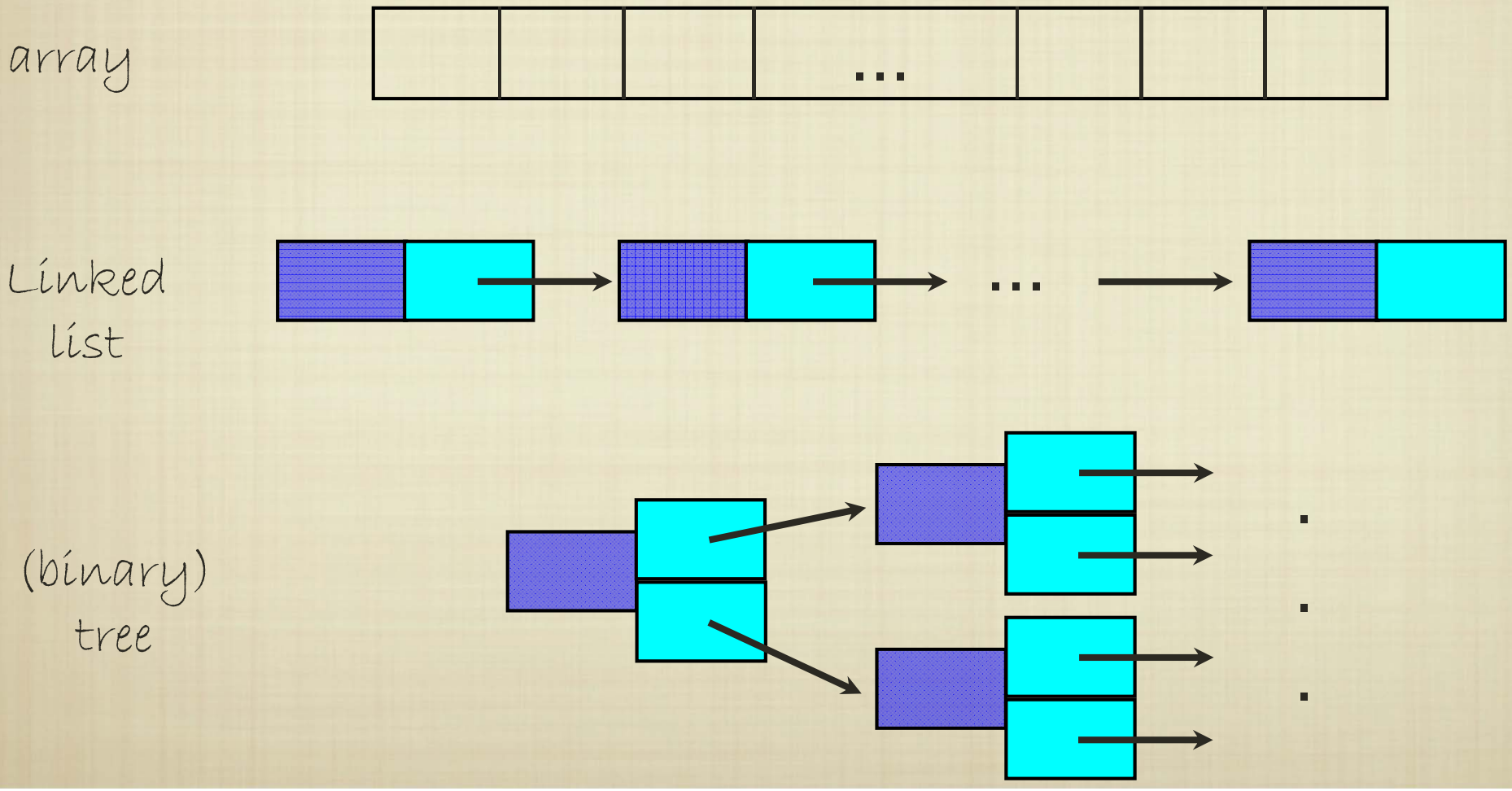
Spring 2014

Carola Wenk

Data Structures We Know

- We've seen arrays and linked structures. All queue and stack operations take constant time, but what about if we want to add, remove and find from the items being stored?

`Collection` interface



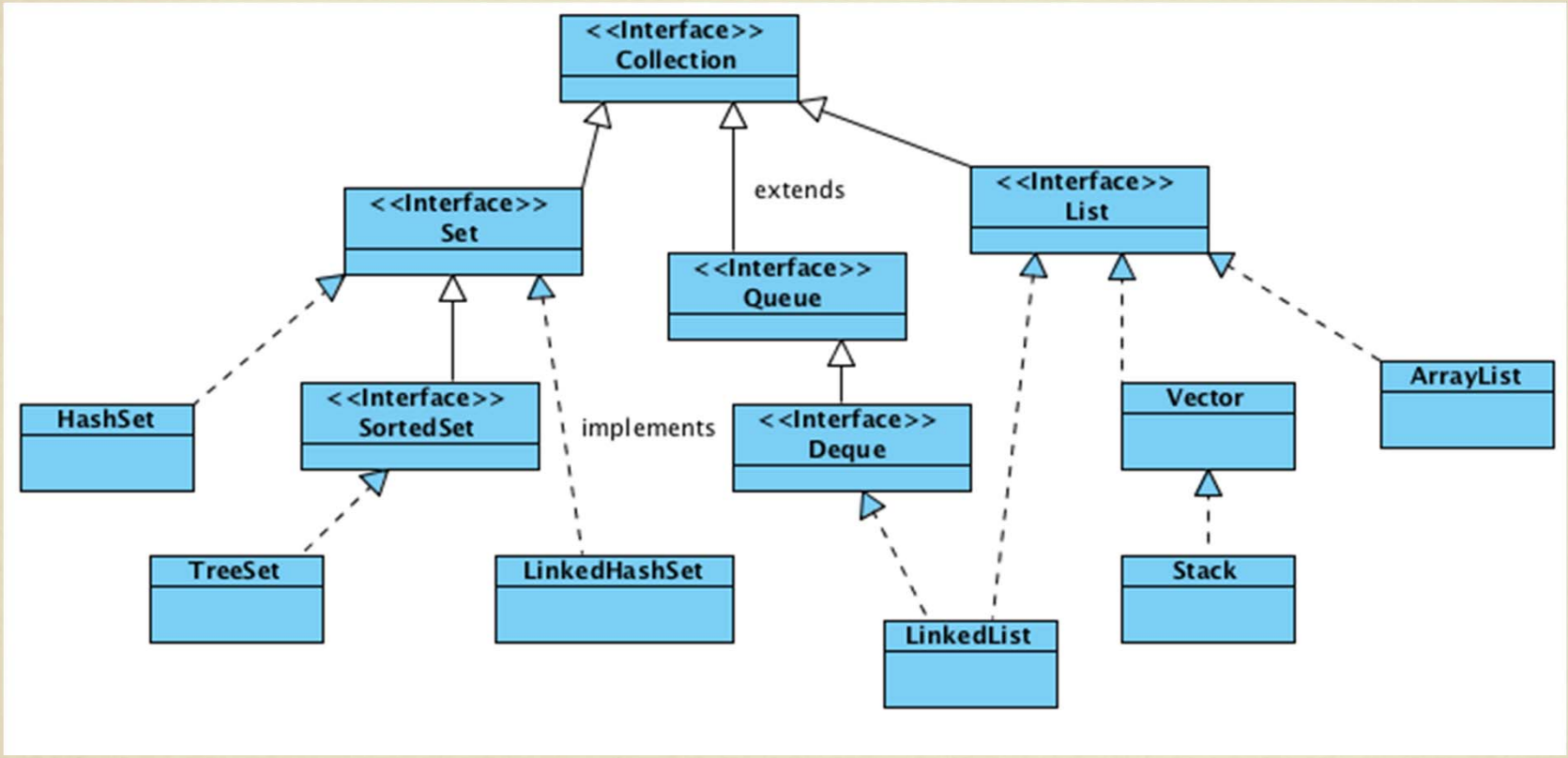
Collection Interface

```
public interface Collection<T>{
    public void add(T item);
    public void remove(T item);
    public boolean contains(T item);
}
```

or **find**

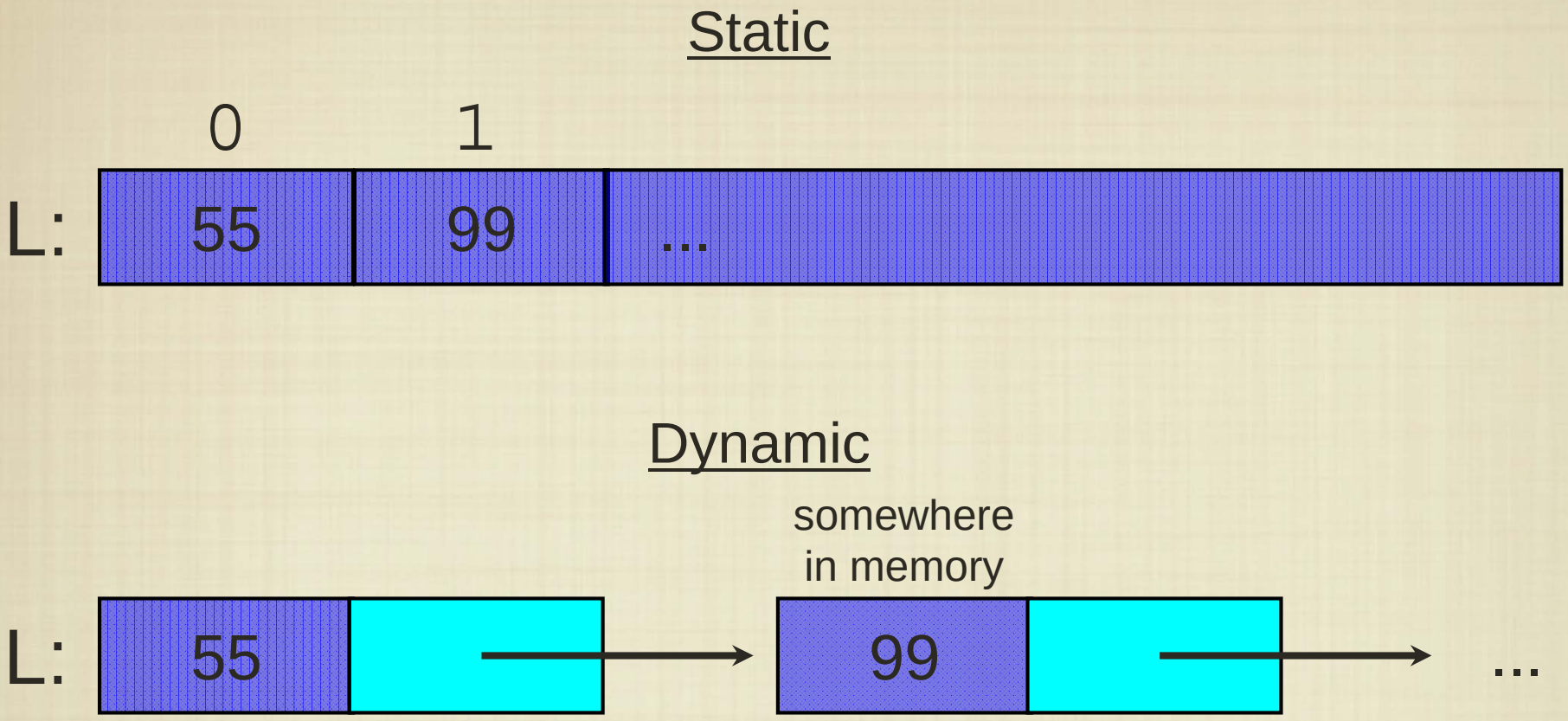
Collections

- Java uses “growable” arrays and linked lists to implement various interfaces derived from `Collection`.



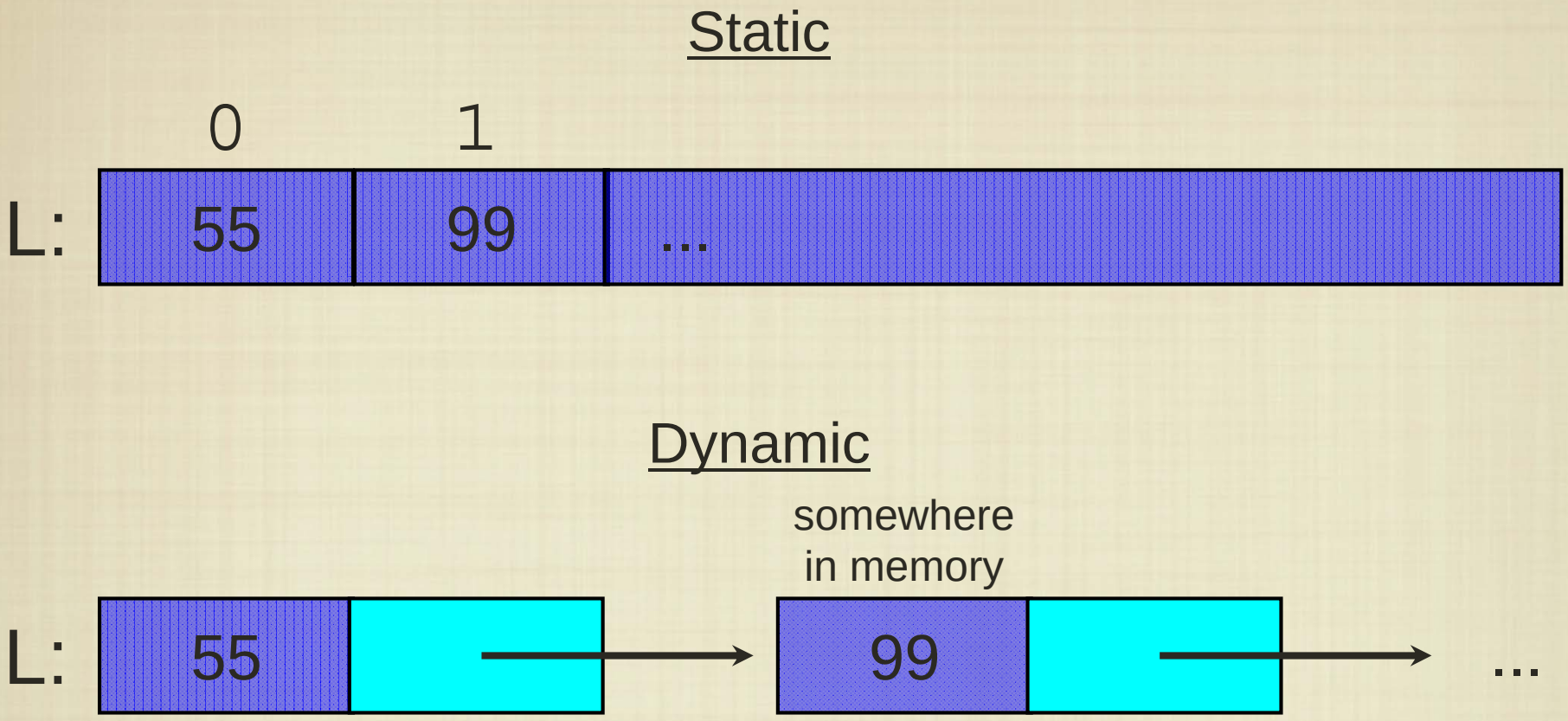
Some of these collections require ordered elements, others do not. What is a “hash” table?

Dynamic Lists



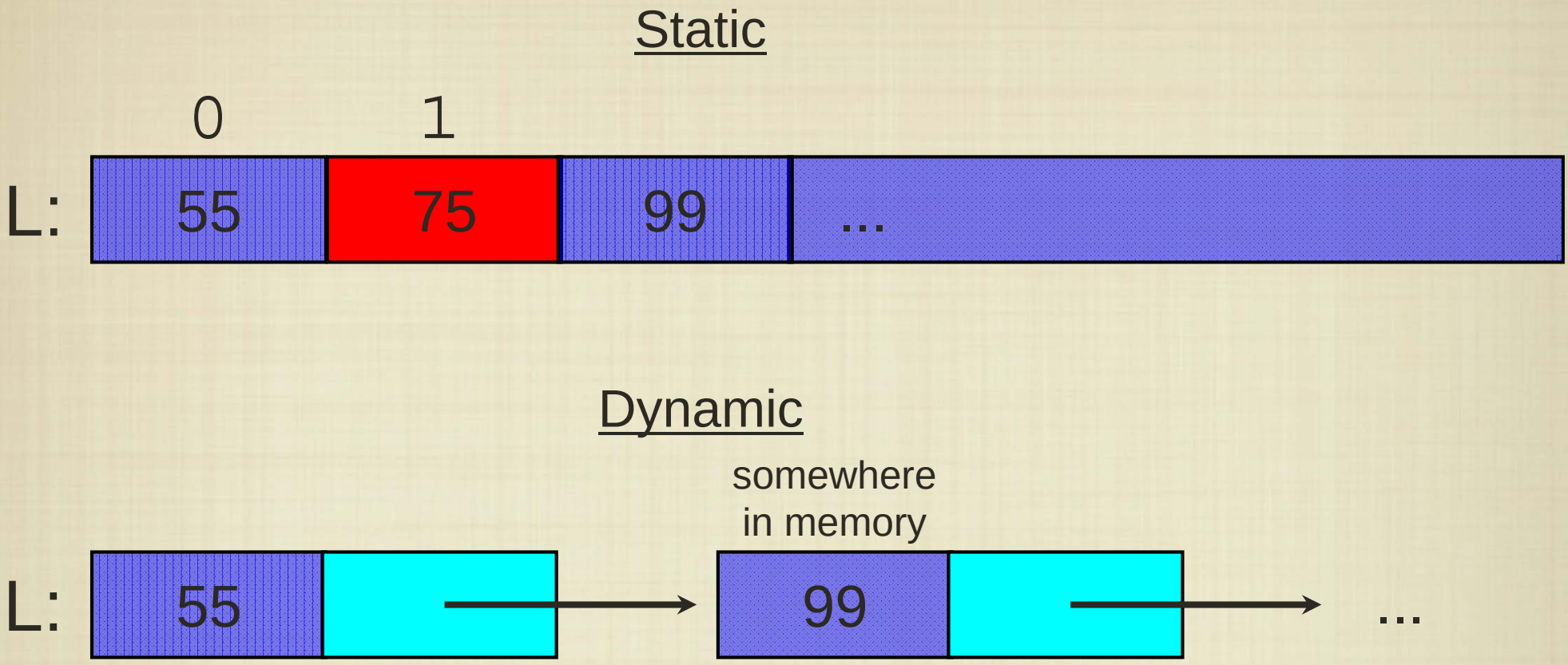
In a dynamic list each element is indirectly adjacent to its neighbor.

Dynamic Lists



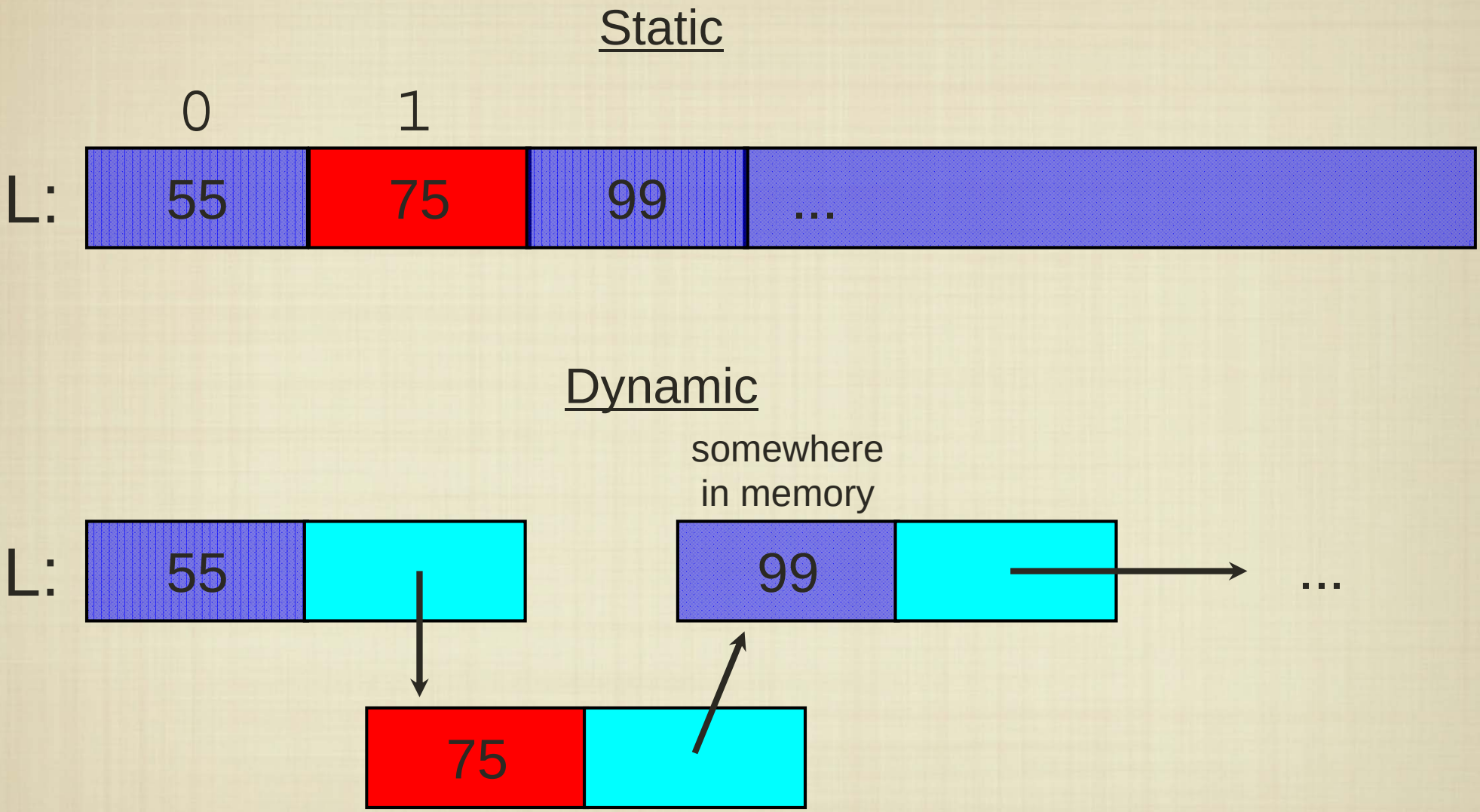
How do we add an item to the dynamic list?

Dynamic Lists



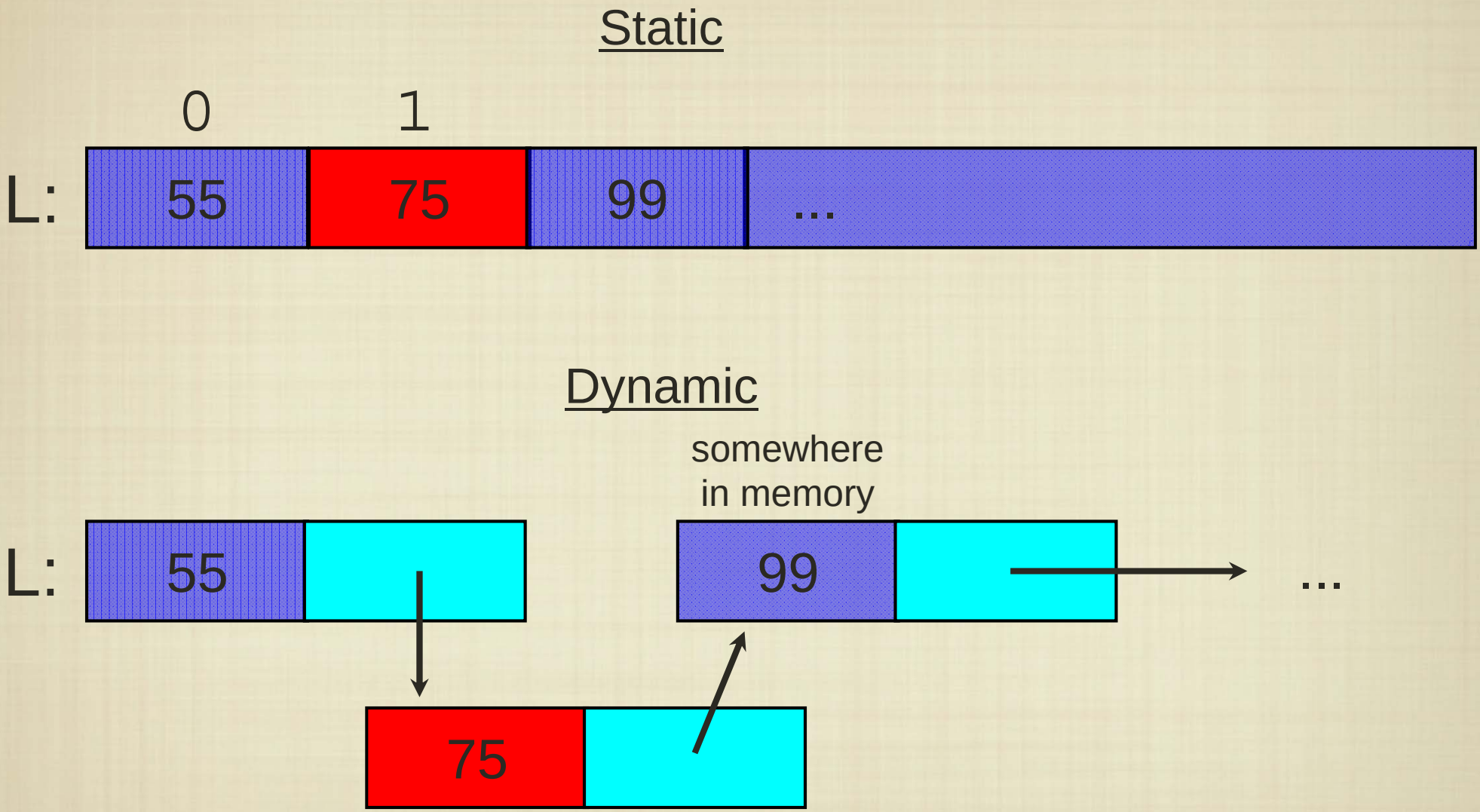
How do we add an item to the dynamic list?

Dynamic Lists



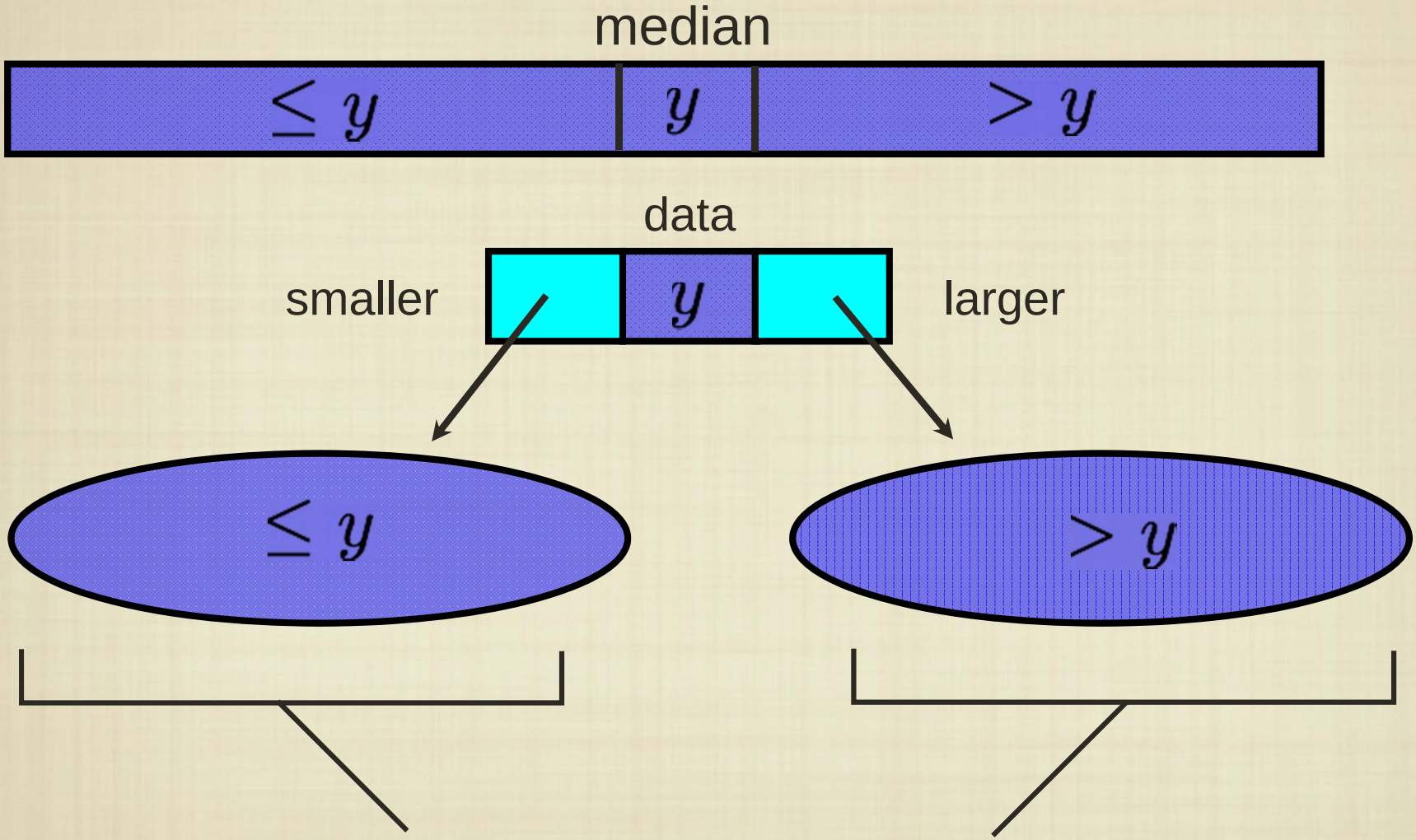
Because it is indirectly defined, to add an element to the dynamic list, we just need to reassign neighbor relationships.

Dynamic Lists



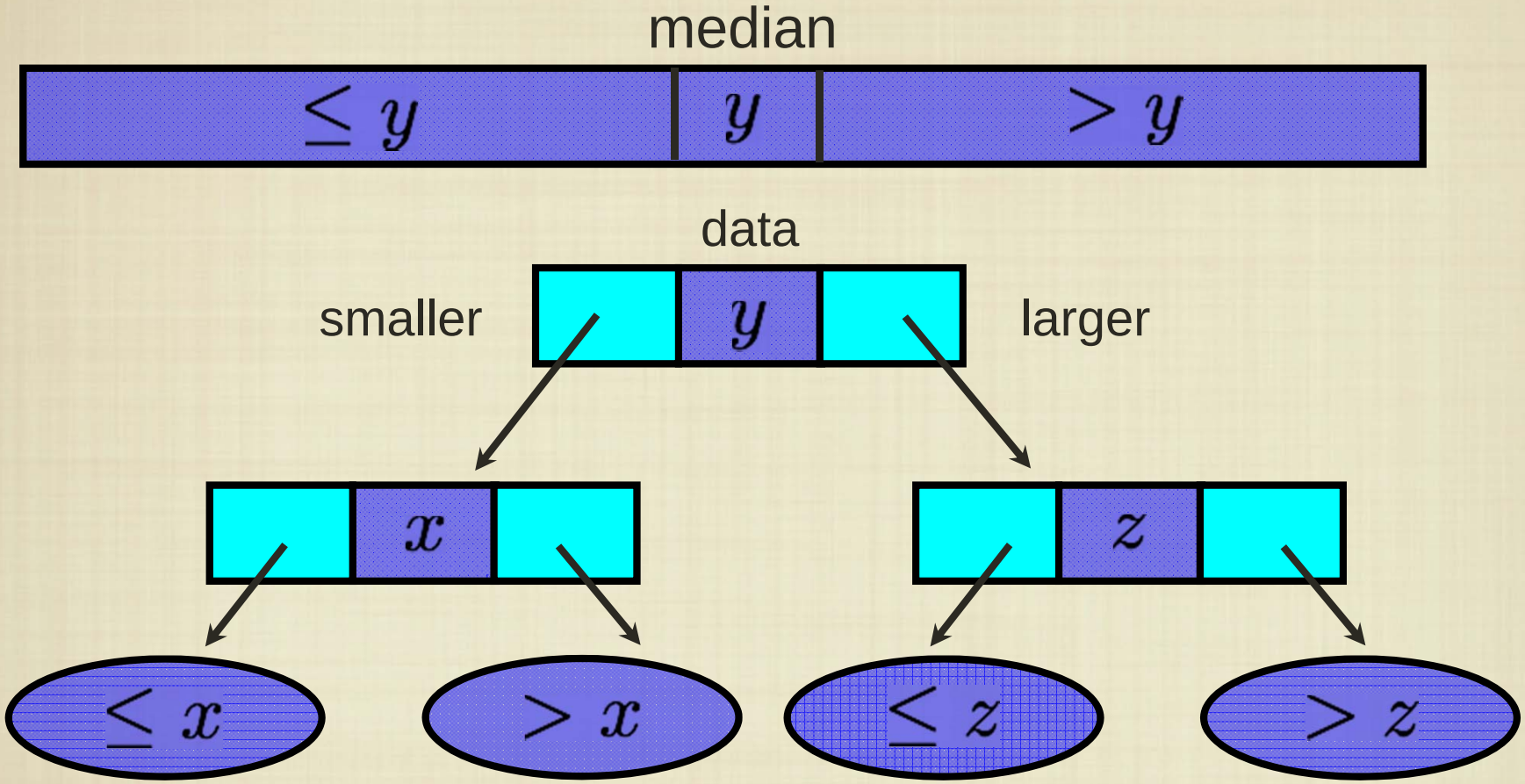
In contrast to the static list, we only need to perform a constant amount of work to add an item to the dynamic list.

Remember Binary Search?



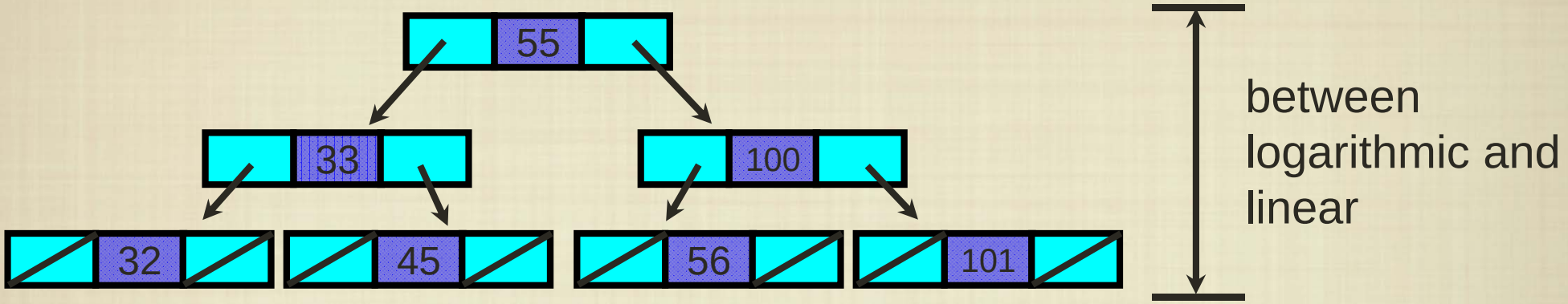
The two halves of a binary search tree can be defined recursively.

Binary Search Trees



- How do we define this type of structure in Java?

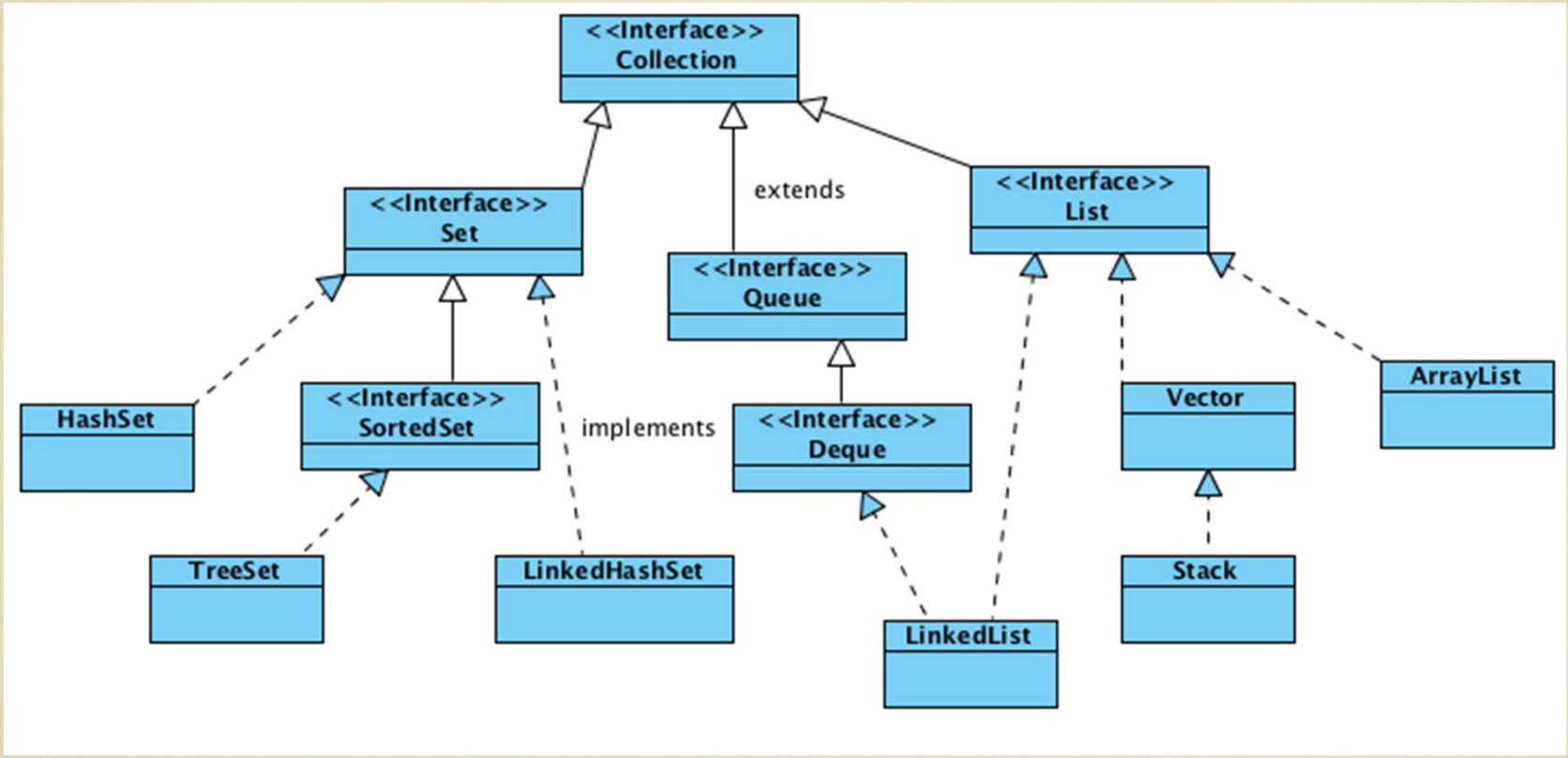
Summary of Binary Search Trees



- The time to perform operations in binary search trees is highly dependent on how they are built.
- The best-case depth of a binary tree is logarithmic in the number of elements; there are sophisticated techniques (AVL, red-black) for ensuring this depth in the worst-case.

Collections

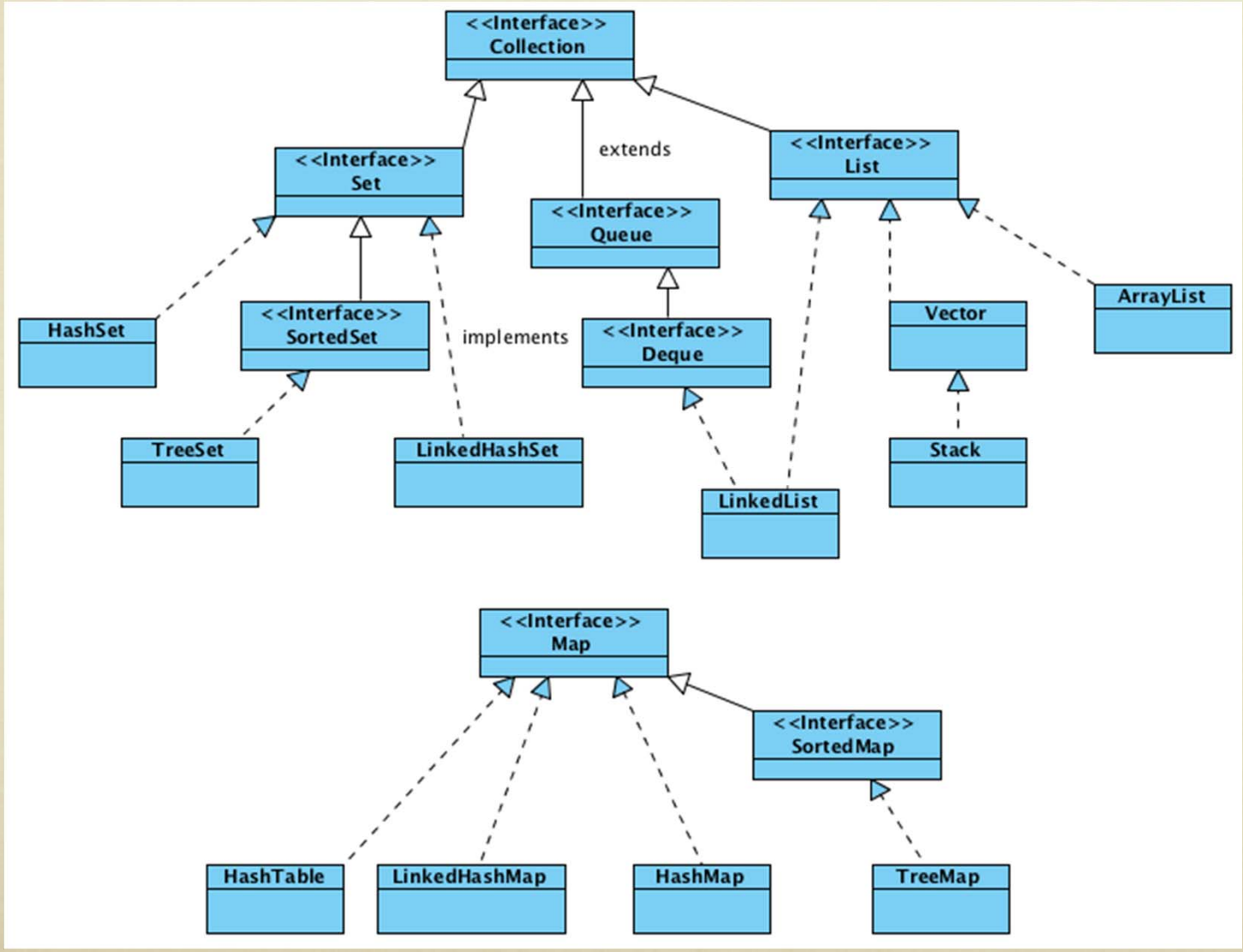
- Java uses “growable” arrays and linked lists to implement various interfaces derived from `Collection`.



Some of these collections require ordered elements, others do not. What is a “hash” table?

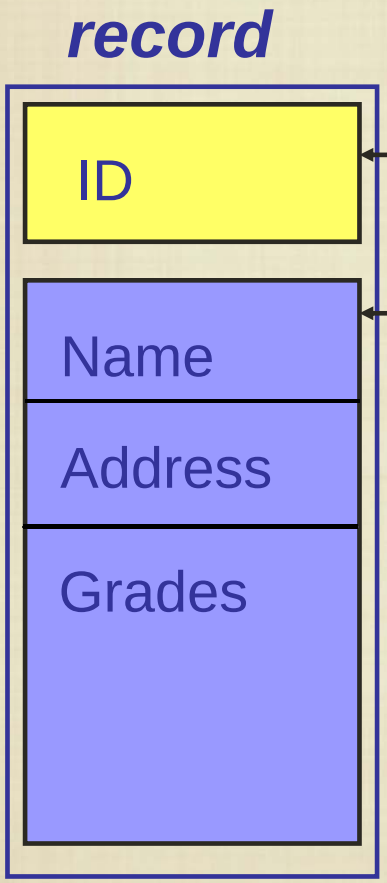
Collections and Maps

- The `Collection` interface is for storage and access, while a `Map` interface is geared towards associating keys with objects.



Student database problem

Tulane's student database D stores *n* ***records***:



key

value

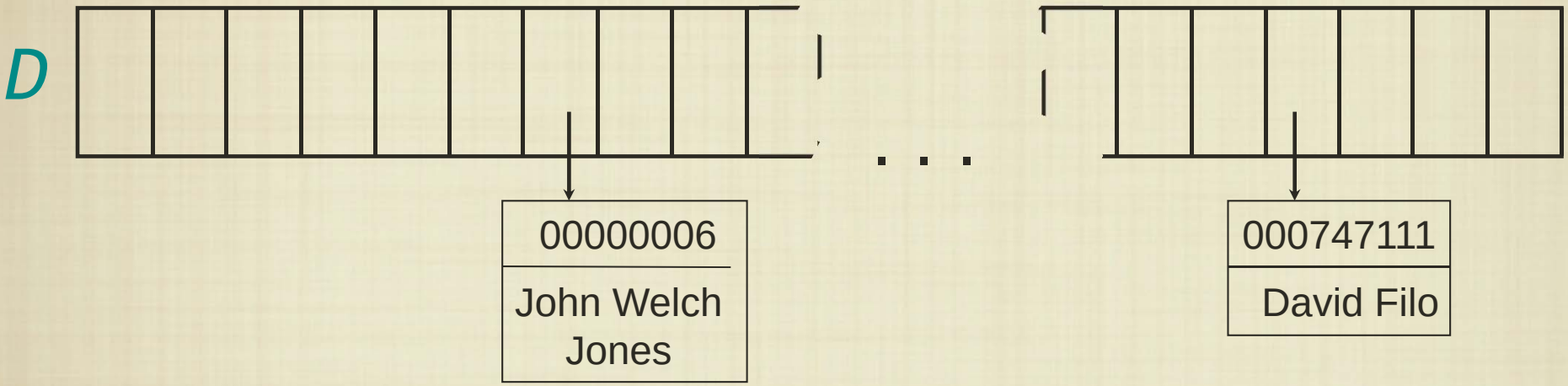
Operations on D:

- "add" D.put(*key*,*value*)
- "find" D.get(*key*)
- D.remove(*key*)

How should the data structure D be organized?

Direct-Access Table (array)

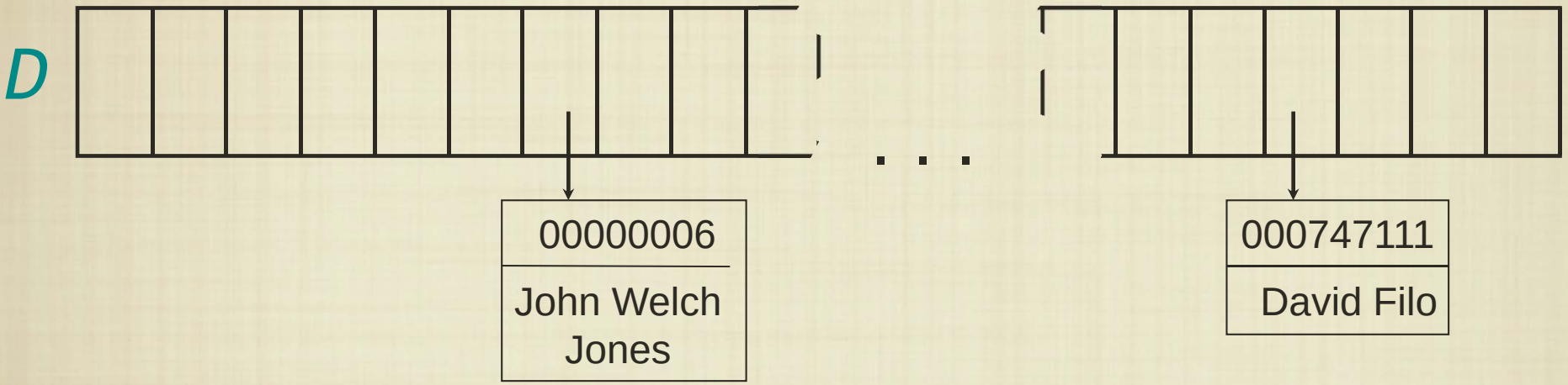
- Suppose every key is a different number: $K \subseteq \{0, 1, \ldots, m-1\}$
- Set up an array $D[0 \ldots m-1]$ such that `D[`*key*`]` = *value* for every record, and `D[`*key*`]`=*null* for keys without records.



`put, get, remove` take $O(1)$ time.

Direct-Access Table (array)

- Suppose every key is a different number: $K \subseteq \{0, 1, \ldots, m-1\}$
- Set up an array $D[0 \ldots m-1]$ such that $D[key] = value$ for every record, and $D[key]=null$ for keys without records.



Problem: The range of keys can be large:

- 64-bit numbers (which represent 18,446,744,073,709,551,616 different keys),
- Character strings (even more!).