Combining Data Structures

Choosing a Data Structure

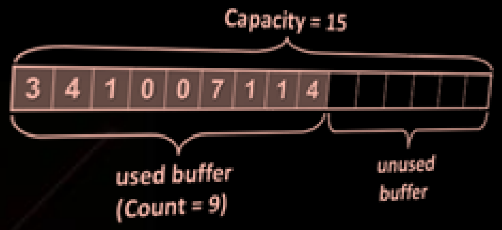
SoftUni Team

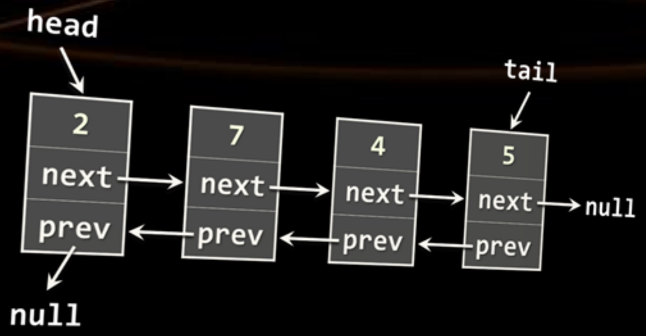
Technical Trainers

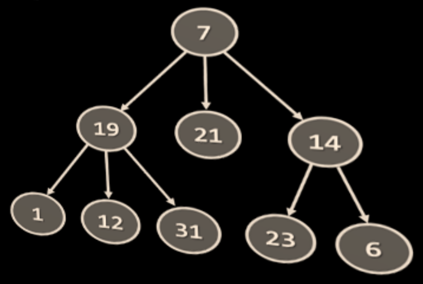
Software University

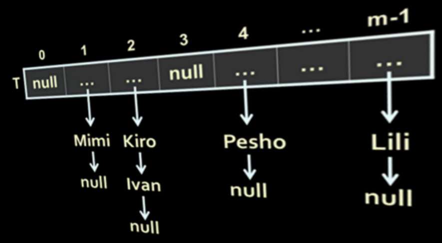
http://softuni.bg

Table of Contents









Choosing the Right DS

Lists vs. Hash Tables vs. Balanced Trees

Choosing a Collection - Array

- Array (**T[ ]**)
  - Use when **fixed number of elements** need processing **by index**
  - No resize → for fixed number of elements only
  - **Add** / **delete** needs creating a new array + move **O(n)** elements
  - Compact and lightweight

Data Structure	Add	Find	Delete	Get-by-index
Static array: T[ ]	O(n)	O(n)	O(n)	O(1)

Choosing a Collection – Array Based List

SOFTWARE UNIVERSITY FOUNDATION

- Resizable array-based list (**`List<T>`**)
  - Use when elements should be fast **added** and processed **by index**
  - Add (append to the end) has **O(1)** amortized complexity
  - The most-often used collection in programming

Data Structure	Add	Find	Delete	Get-by-index
Auto-resizable array-based list: `List<T>`	O(1)	O(n)	O(n)	O(1)

Choosing a Collection – Linked List

- Doubly-linked list (**`LinkedList<T>`**)
  - Use when elements should be **added at the both sides** of the list
  - Use when you need to **remove by a node reference**
  - Otherwise use resizable array-based list (**`List<T>`**)

Data Structure	Add	Find	Delete	Get-by-index
Double-linked list: `LinkedList<T>`	O(1)	O(n)	O(n)	O(n)

Choosing a Collection – Stack

- Stack (`Stack<T>`)
  - Use to implement **LIFO** (last-in-first-out) behavior
  - `List<T>` could also work well

Data Structure	Add	Find	Delete	Get-by-index
Stack: `Stack<T>`	O(1)	-	O(1)	-

Choosing a Collection – Queue

- Queue (`Queue<T>`)
  - Use to implement **FIFO** (first-in-first-out) behavior
  - `LinkedList<T>` could also work well

Data Structure	Add	Find	Delete	Get-by-index
Queue: `Queue<T>`	O(1)	-	O(1)	-

Choosing a Collection – Map

- Hash-table-based map (**`Dictionary<K,V>`**)
  - Fast **add key-value pairs** + fast **search by key** – O(1)
  - Keys have **no particular order**
  - Keys should implement **`GetHashCode(…)`** and **`Equals(…)`**

Data Structure	Add	Find	Delete	Get-by-index
Hash-table: `Dictionary<K,V>`	O(1)	O(1)	O(1)	-

Choosing a Collection – Tree Map

- Balanced tree-based map (**`OrderedDictionary<K,V>`**)
  - Elements are **ordered** by key
  - Fast **add key-value pairs** + fast **search by key** + fast **sub-range**
  - Keys should be **`IComparable<T>`**
  - Balanced trees → slower than hash-tables: **O(log n)** vs. **O(1)**

Data Structure	Add	Find	Delete	Get-by-index
Balanced tree-based dictionary: **`SortedDictionary<K,V>`**	O(log n)	O(log n)	O(log n)	-

Choosing a Collection – Multi Map

- Hash-table-based multi-dictionary (**`MultiDictionary<K,V>`**)
  - Fast **add key-value** + fast **search by key** + **multiple values** by key
  - Add by existing key **appends a new value** for the same key
  - Keys have **no particular order**

Data Structure	Add	Find	Delete	Get-by-index
Hash-table-based multi-dictionary: `MultiDictionary<K,V>`	O(1)	O(1)	O(1)	-

Choosing a Collection – Tree Multi Map

- Tree-based multi-dictionary (**`OrderedMultiDictionary<K,V>`**)
  - Keys are **ordered** by key
  - Fast **add key-value** + fast **search by key** + fast **sub-range**
  - Add by existing key appends a new value for the same key

Data Structure	Add	Find	Delete	Get-by-index
Tree-based multi-dictionary: **`SortedDictionary<K,V>`**	O(log n)	O(log n)	O(log n)	-

Choosing a Collection – Hash Set

- Hash-table-based set (**`HashSet<T>`**)
  - **Unique** values + fast **add** + fast **contains**
  - Elements have **no particular order**
  - Elements should implement **`GetHashCode(…)`** and **`Equals(…)`**

Data Structure	Add	Find	Delete	Get-by-index
Hash-table-based set: `HashSet<T>`	O(1)	O(1)	O(1)	-

Choosing a Collection – Tree Set

- Balanced tree-based set (**`SortedSet<T>`**)
  - **Unique** values + **sorted order**
  - Fast **add** + fast **contains** + fast **sub-range**
  - Elements should be **`IComparable<T>`**

Data Structure	Add	Find	Delete	Get-by-index
Balanced tree-based set: `SortedSet<T>`	O(log n)	O(log n)	O(log n)	-

Choosing a Collection – Hash Bag

- Hash-table-based bag (**Bag<T>**)
  - Bags allow **duplicates**
  - Fast **add** + fast **find** + fast **contains**
  - Elements have **no particular order**

Data Structure	Add	Find	Delete	Get-by-index
Hash-table-based bag: **Bag<T>**	O(1)	O(1)	O(1)	-

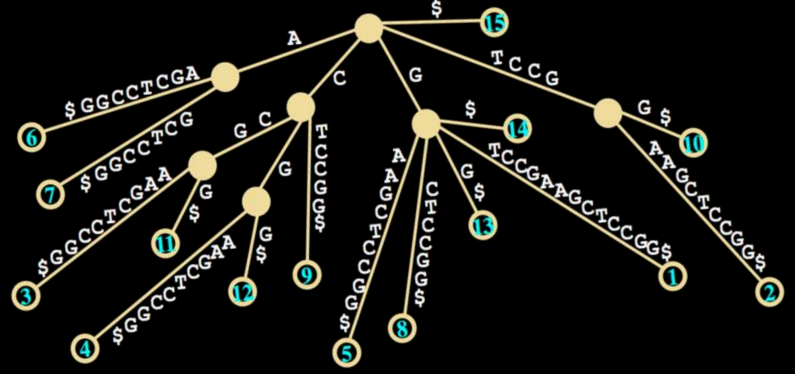
Choosing a Collection – Tree Bag

- Balanced tree-based bag (**`SortedBag<T>`**)
  - Allow **duplicates**, **sorted order**
  - Fast **add** + fast **find** + fast **contains**
  - Access by **sorted index** + extract **sub-range**

Data Structure	Add	Find	Delete	Get-by-index
Balanced tree-based bag: `OrderedBag<T>`	O(log n)	O(log n)	O(log n)	-

Choosing a Collection – Special DS

- Priority Queue (**Heap**) – **fast max/min** element
- **Rope** – fast **add/remove** by **index**
- **Prefix** tree (Trie) – fast **prefix search**
- **Suffix** tree – fast **suffix search**
- **Interval** tree – fast **interval search**
- **K-d** trees, **Quad** trees – fast **geometric distance search**



Data Structure Efficiency – Comparison

Data Structure	Add	Find	Delete	Get-by-index
Static array: T[]	O(n)	O(n)	O(n)	O(1)
Double-linked list: LinkedList<T>	O(1)	O(n)	O(n)	O(n)
Auto-resizable array-based list: List<T>	O(1)	O(n)	O(n)	O(1)
Stack: Stack<T>	O(1)	-	O(1)	-
Queue: Queue<T>	O(1)	-	O(1)	-

Data Structure Efficiency – Comparison (2)

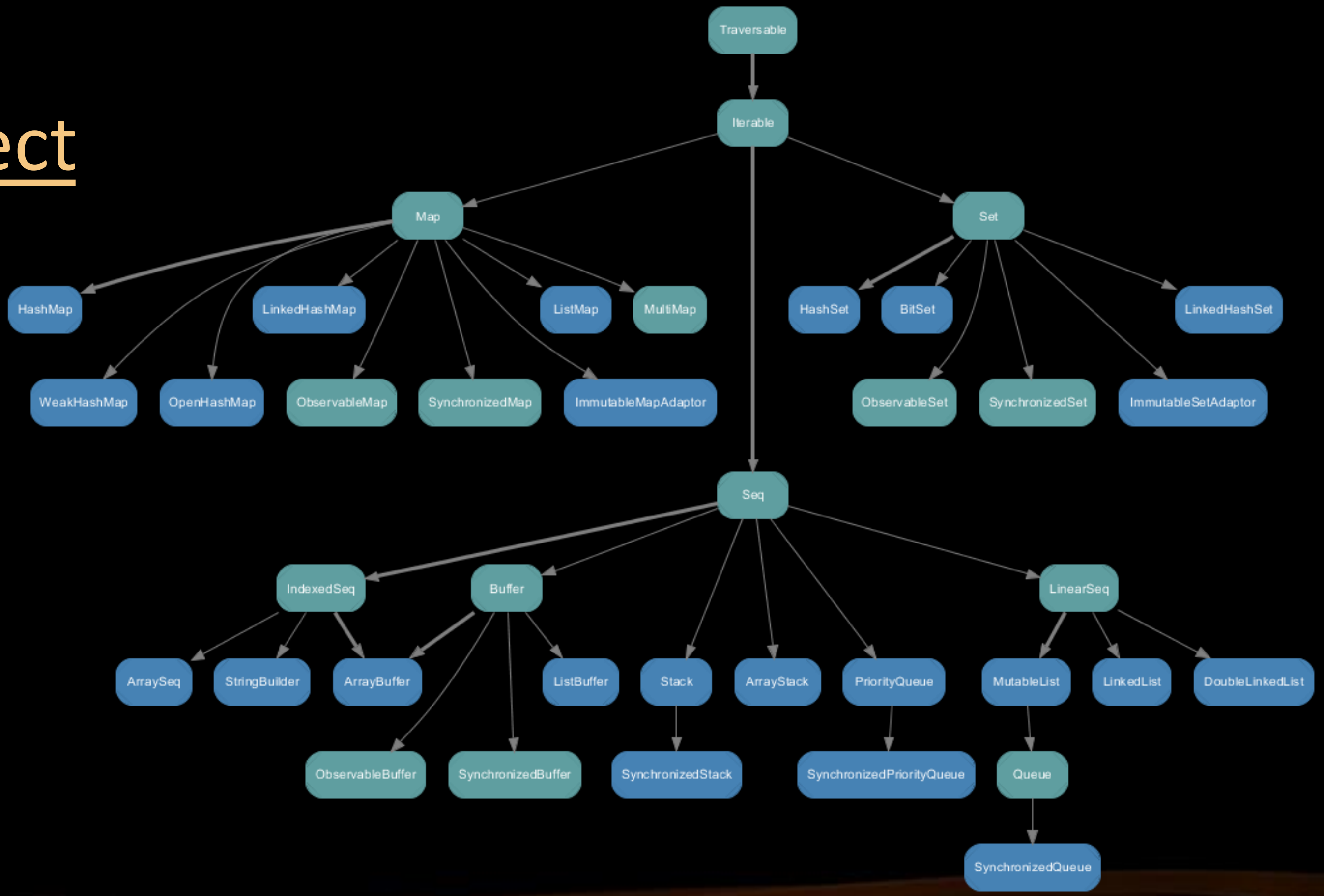
Data Structure	Add	Find	Delete	Get-by-index
Hash-table: `Dictionary<K,V>`	O(1)	O(1)	O(1)	-
Balanced tree-based dictionary: `SortedDictionary<K,V>`	O(log n)	O(log n)	O(log n)	-
Hash-table-based set: `HashSet<T>`	O(1)	O(1)	O(1)	-
Balanced tree-based set: `SortedSet<T>`	O(log n)	O(log n)	O(log n)	-

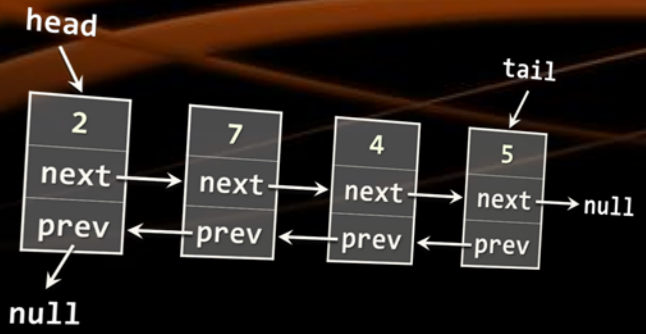
Data Structure Efficiency – Comparison (3)

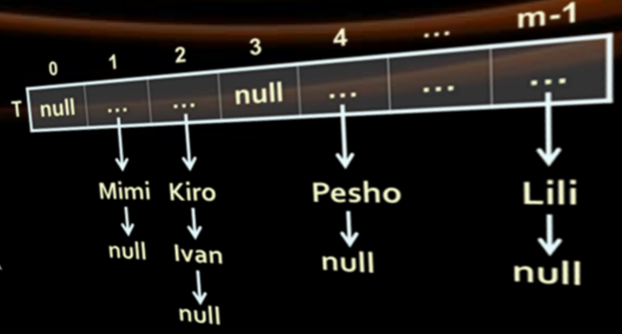
Data Structure	Add	Find	Delete	Get-by-index
Hash-table-based multi-dictionary: `MultiDictionary<K,V>`	O(1)	O(1)	O(1)	-
Tree-based multi-dictionary: `SortedDictionary<K,V>`	O(log n)	O(log n)	O(log n)	-
Hash-table-based bag: `Bag<T>`	O(1)	O(1)	O(1)	-
Balanced tree-based bag: `OrderedBag<T>`	O(log n)	O(log n)	O(log n)	-

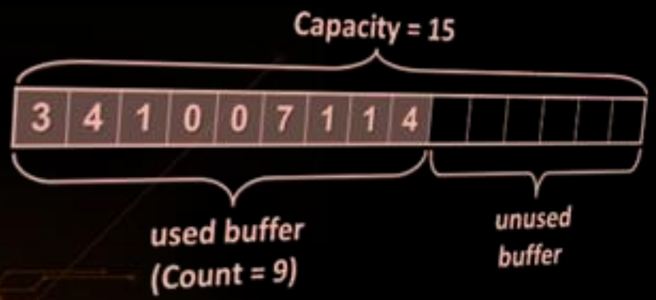
Java – Collections/Guava APIs

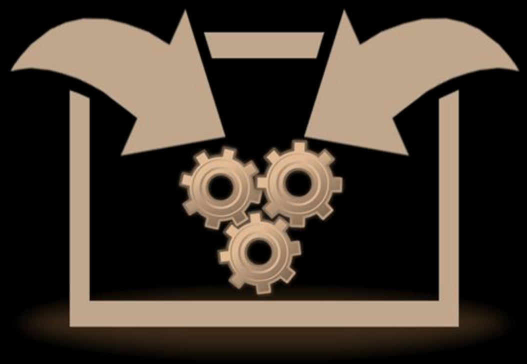
- All commonly used collections
  - [java.util.Collections]
  - [com.google.common.collect]

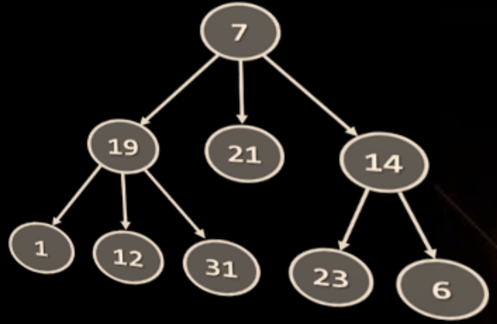












SOFTWARE UNIVERSITY FOUNDATION

Combining Data Structures

Combining Data Structures

- Many scenarios → combine several DS
  - No ideal DS → choose between space and time
- For example, we can combine:
  - A hash-table for fast search by key_1 (e.g. *name*)
  - A hash-table for fast search by {key_2 + key_3} (e.g. *name* + *town*)
  - A balanced search tree for fast extract-range(start_key … end_key)
  - A rope for fast access-by-index
  - A balanced search tree for fast access-by-sorted-index

Problem: Collection of Persons

- Design a data structure that efficiently implements:

Operation	Return Type
Add(email, name, age, town)	**bool** – unique email
Find(email)	**Person** or **null**
Delete(email)	**bool**
Find-All(email_domain)	**IEnumerable<P>** – sorted by email
Find-All(name, town)	**IEnumerable<P>** – sorted by email
Find-All(start_age, end_age)	**Ienumerable<P>** – sorted by age, email
Find-All(start_age, end_age, town)	**Ienumerable<P>** – sorted by age, email

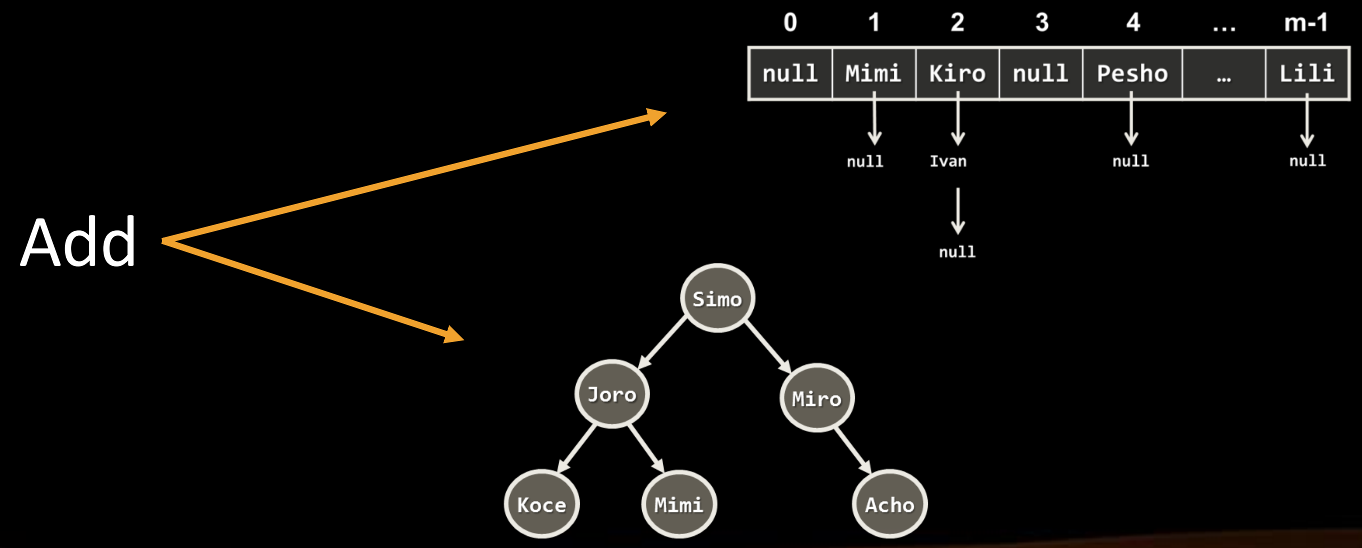
List Based Solution

- List based solution – **single list** for all operations
  - **Easy** to **implement**
  - Easy to achieve **correct behavior**
  - Useful for **creating unit tests**

0	1	2	…	n-1
Mimi	Jana	Acho	…	Lili

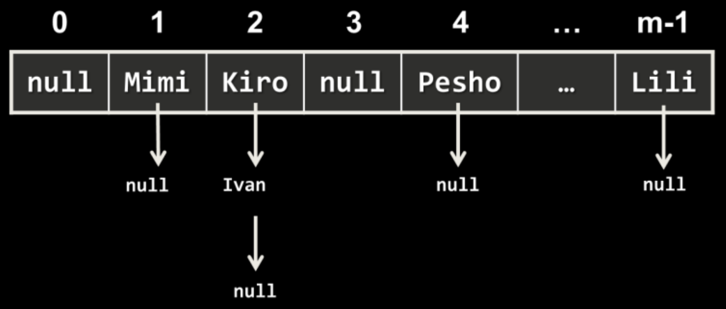
Solution: Add Person

- bool **Add**(email, name, age, town)
  - Create a **`Person`** object to hold { *email* + *name* + *age* + *town* }
  - Add the new *person* to all underlying data structures



Solution: Find by Email

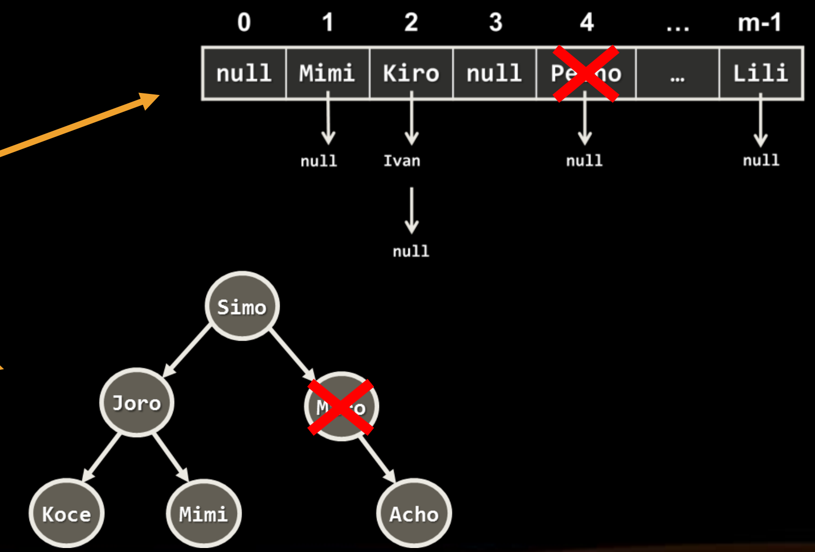
- Person **Find**(email)
  - Use a hash-table to map { *email* → *person* }
  - Complexity – O(1)



Solution: Delete

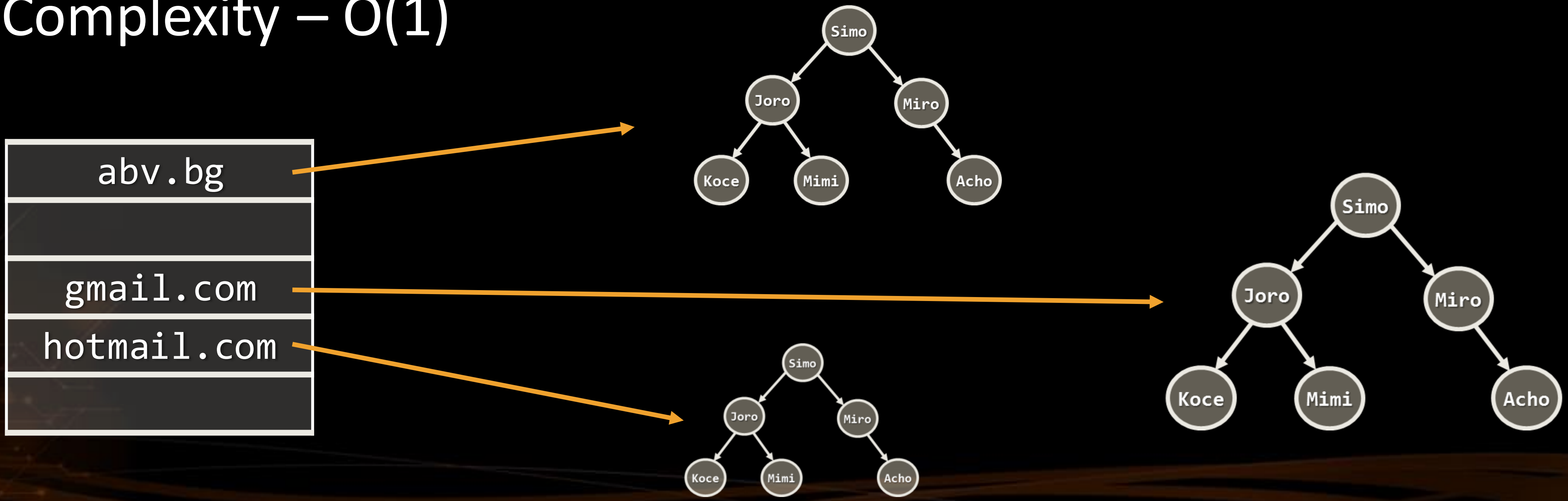
- bool **Delete**(email)
  1. Find the *person* by *email* in the underlying hash-table
  2. Delete the *person* from all underlying data structures
  3. Complexity – O(log n)

Delete



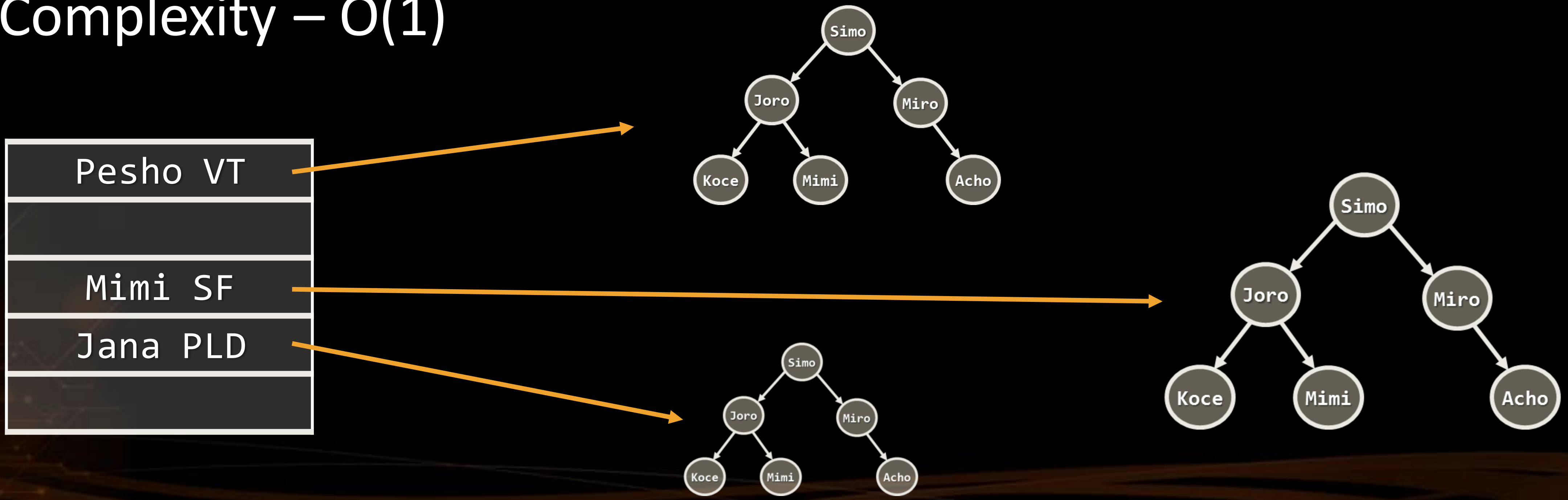
Solution: Find by Domain

- IEnumerable<Person> **Find-All**(*email_domain*)
  - Use a hash-table to map {*email_domain* → *SortedSet<Person>*}
  - Get *email_domain* by the *email* when adding persons
  - Complexity – O(1)



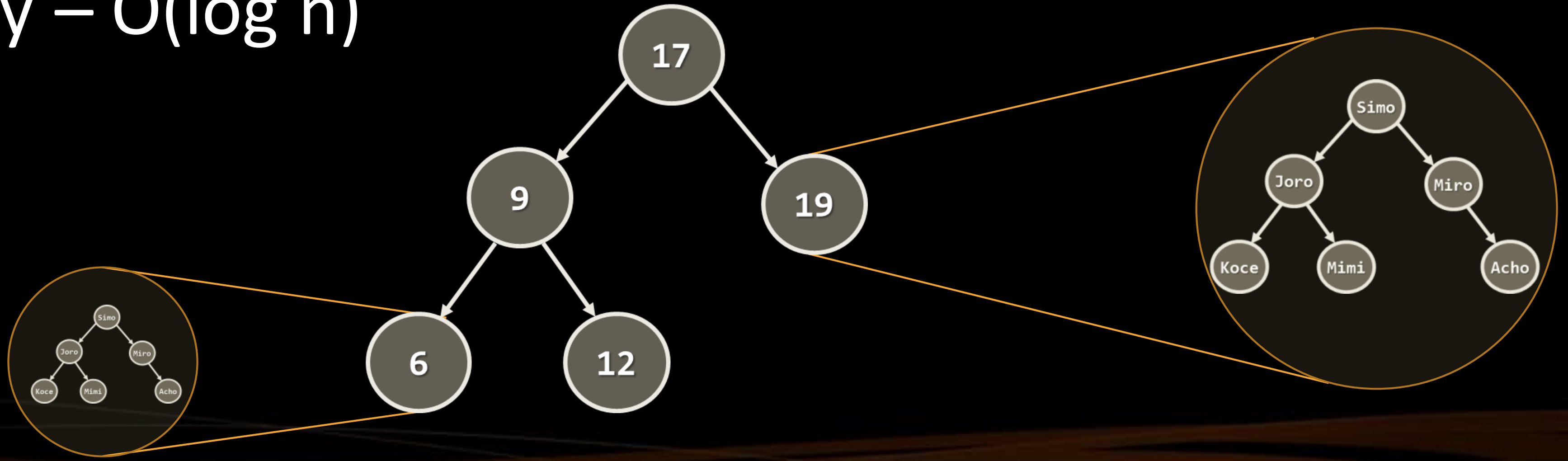
Solution: Find by Name + Town

- IEnumerable<Person> **Find-All**(*name*, *town*)
  - Combine the keys {*name* + *town*} into a single string *name_town*
  - Use a hash-table to map {*name_town* → *SortedSet<Person>*}
  - Complexity – O(1)



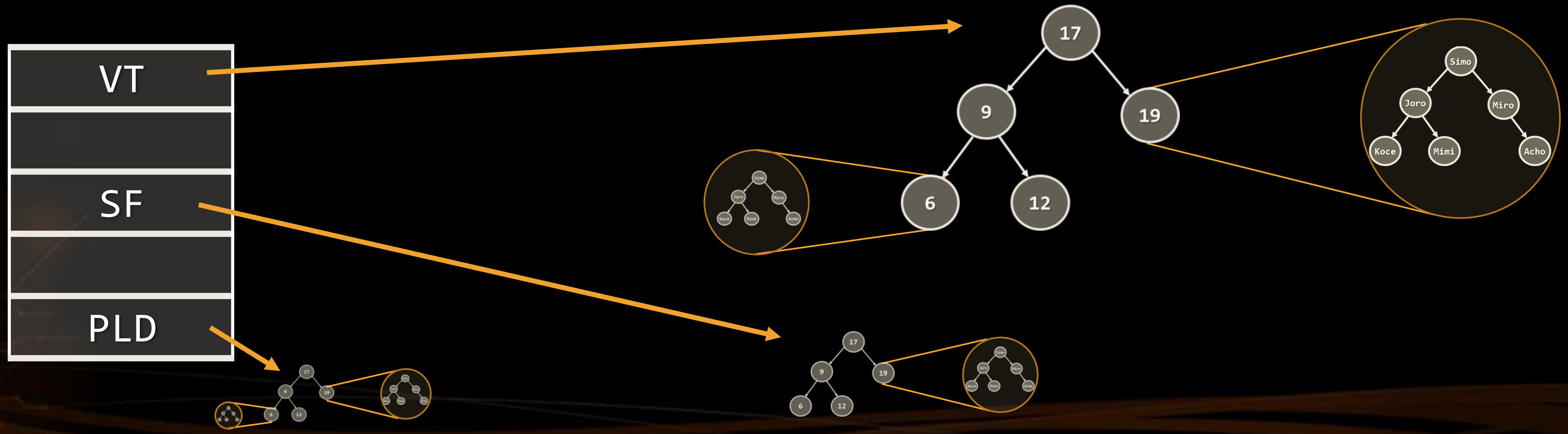
Problem: Collection of Persons

- IEnumerable<Person> **Find-All**(*start_age*, *end_age*)
  - Use a **balanced search tree** to keep all *persons* ordered by *age*:
    - `OrderedDictionary<age, SortedSet<Person>>`
  - Use the **Range**(*start_age, end_age*) operation in the tree
  - Complexity – O(log n)



Problem: Collection of Persons

- Ienumerable<Person> **Find-All**(*start_age*, *end_age*, *town*)
- Use a hash-table to map {*town* → *persons_by_ages*}
  - *People_by_ages* can be stored as balanced search tree:
    - **`OrderedDictionary<age, SortedSet<Person>>`**



Lab Exercise

Collection of People

Summary

- Different data structures have different efficiency for their operations
  - List-based collections provide fast append and access-by-index, but slow find and delete
  - The fastest add / find / delete structure is the hash table – `O(1)` for all operations
  - Balanced trees are ordered – O(log n) for add / find / delete + range(start, end)
- Combining data structures is often essential
  - E.g. combine multiple hash-tables to find by different keys