CMPT 225: Data Structures & Programming – Unit 31 – Exam Review

Dr. Jack Thomas
Simon Fraser University
Spring 2021

The April 19th Exam

- **Monday, April 19th, at 12:00pm to 3:00pm** (that's noon, not midnight!)
- One attempt, three hours, MUST be completed during this time!
- Completed on Canvas, under the Quiz tab the same way the Midterm was.
- If this doesn't work for you, **NOTIFY ME ASAP!**

Format

- Roughly twice as long as the midterm.
- The same three types of questions:
  - **Very Short Answer Questions**: Answers should be a sentence or two.
  - **Short Answer Questions**: A paragraph (or equivalent).
  - **Code Questions**: Questions that involve coding. Highly recommend you open the IDE to a blank project before you begin the midterm so you can code there and then copy-paste your answer over.

Academic Integrity

- The midterm is **open book** again, meaning you're free to consult your notes, course material, or even the open internet.
- You **may NOT cooperate with anyone** to complete your midterm, especially other students.
- Any source you use outside of course material **must be cited** – looking code up is fine, lifting code directly will be treated as plagiarism.
- **WE HAD CASES OF THIS ON THE MIDTERM**, so double check!

Content

- The exam is **cumulative**, meaning there will be questions from both before and after the midterm.
- **At least half** of the exam will be based on material we covered **after the midterm**.
- We **won’t be taking questions directly** from the assignments, labs, or textbooks, but they may be similar.

How to Study for the Exam

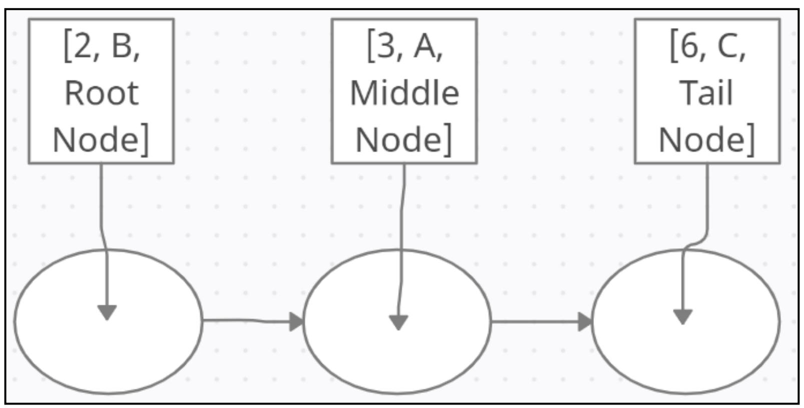
1. Attend this review (good job!)
2. Also, review the Midterm Review!
3. Consult your notes.
4. Check the slides
5. Watch the recordings.
6. Go through your code and the sample solutions.

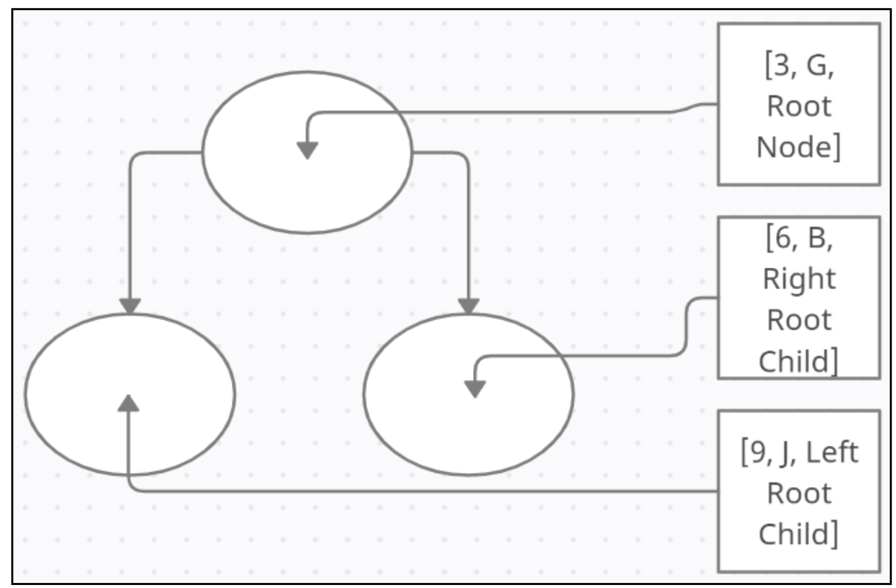
Adaptable Priority Queues

- A **variation on Priority Queues** that makes it possible to remove entries other than the next highest priority one, or swap the key or value of a given entry.
- The **Java PriorityQueue** is **already adaptable**.
- Efficiency impact varies by PQ implementation: the remove and replace functions are constant for the **unsorted list**, O(log n) for the **heap**, and constant for remove/O(n) for replace for **sorted lists**.

Adaptable Priority Queues

- Achieved through **location-aware entries**, which introduced us to the difference between positions and entries.





The Adaptable Priority Queue ADT

- An **extension of the Priority Queue** data structure that allows for removing and editing arbitrary entries, not just the highest priority.
- Standard methods include all of the PQ ones, as well as:
  - **Remove**: Removes a given entry from the PQ, while ensuring it remains ordered.
  - **replaceKey**: Swaps the key of a given entry, then adjusting the ordering as needed.
  - **replaceValue**: Swaps the value of a given entry, which probably also requires re-checking the ordering.

Maps

- A **key-based data structure** where **all keys are unique**.
- Sometimes called **associative stores**, since multiple entries with the same key might be stored in the same spot.
- This leads to the idea of **keys as indexes** leading to **addresses**.
- Available as an **interface**, not a standard class, but with some implementations like **HashMap**.

The Map ADT

- A unique-key-based data structure, storing a set of key-value pairs called entries.
- Standard methods include:
  - **Get**: Return the value associated with the given key.
  - **Put**: If a given key doesn’t exist in the map yet, add it and the given value, otherwise replace the existing value of the given key with the given value.
  - **Remove**: Removes and returns the entry associated with a given key.
  - **keySet**: Returns a collection of all the keys stored in the entries.
  - **Values**: Returns a collection of all the values stored in the entries.
  - **entrySet**: Returns a collection of all entries.

Hash Tables

- A form of **Map** made from a **Bucket Array** and **Hash Function**.
- Hash Functions turn keys into **Hash Codes**, using methods like **Polynomial Hash Codes**.
- **Compression Functions** include the **Division Method** and the **MAD Method**.
- **Collision Handling** includes **Separate Chaining** and **Open Addressing**.

Hash Tables

- The **Load Factor** is the current proportion of full buckets, and expanding the bucket array when it’s too full is called **Rehashing**.
- Efficient with constant-time access for adding and retrieving, like an array, but **only when collisions are kept low**.
- Useful for counting collisions, caches, and other quick-access memory solutions.
- **Java** has a **Hashtable** class built in.

Ordered Maps

- Maps, but can also **retrieve subsets of entries** whose keys fall between bounds.
- Can be implemented with **the Ordered Search Table**, using an **ArrayList** as the underlying data structure to implement its methods.
- Also where we introduced **Binary Search**.

Ordered Map ADT

- A key-based data structure that can return entries based on the relative ordering of its entries' keys.
- Standard methods include the ones for any Map, as well as:
  - **firstEntry**: Returns the entry with the smallest key.
  - **lastEntry**: Returns the entry with the largest key.
  - **ceilingEntry**: Returns the entry with the smallest key greater than or equal to a given key.
  - **higherEntry**: As above, but only greater than.
  - **floorEntry**: Returns the entry with the largest key less than or equal to a given key.
  - **lowerEntry**: As above, but only less than.

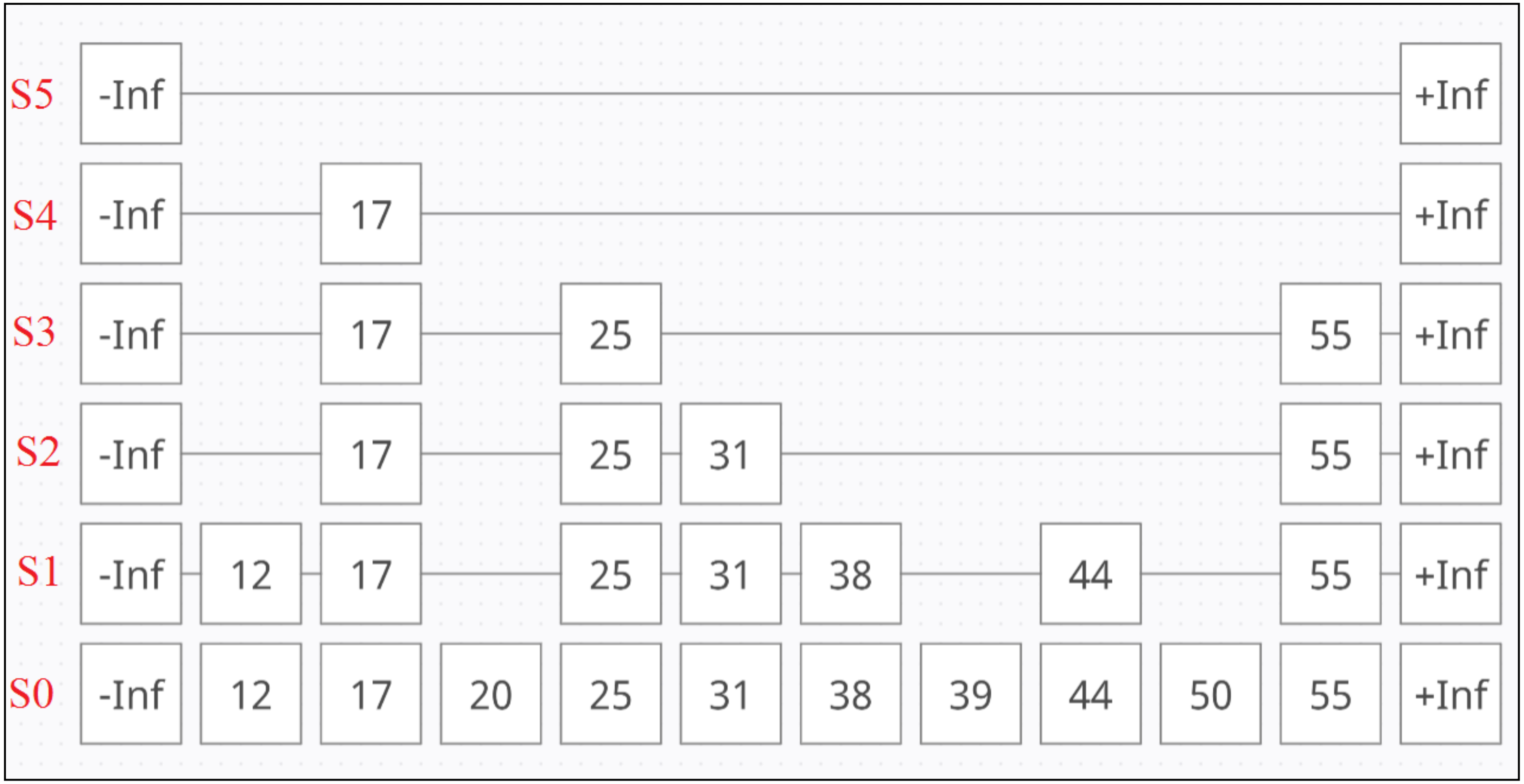
Map Implementation Choices

- There's a recap at this point discussing some of the **different options for implementing Maps** and their efficiency in different situations.

Map Method	List	HashTable	Ordered Search Table
size, isEmpty	O(1)	O(1)	O(1)
entrySet	O(n)	O(n)	O(n)
get	O(n)	O(1)/O(n)	O(log n)
put	O(1)	O(1)	O(n)
remove	O(n)	O(1)/O(n)	O(n)

Skip Lists

- A data structure suitable for **implementing Ordered Maps**.



Skip List: The ADT

- A data structure that implements and extends the Ordered Map ADT.
- Improves the average time of search and update operations to O(log n) through random arrangements.
- Standard methods include those of the Ordered Map ADT, along with:
  - **Next**: The position following a given position on a level.
  - **Prev**: The position preceding a given position on a level.
  - **Below**: The position below a given position in a tower.
  - **Above**: The position above a given position in a tower.

Dictionaries

- A Map-like data structure with **non-unique keys**, which can store multiple entries with the same key under one location.
- Remember that the **Java Dictionary isn't the same as this one**.
- Can also be implemented with an unordered list, ordered search table, hash table using separate chaining, or skip list.

The Dictionary: The ADT

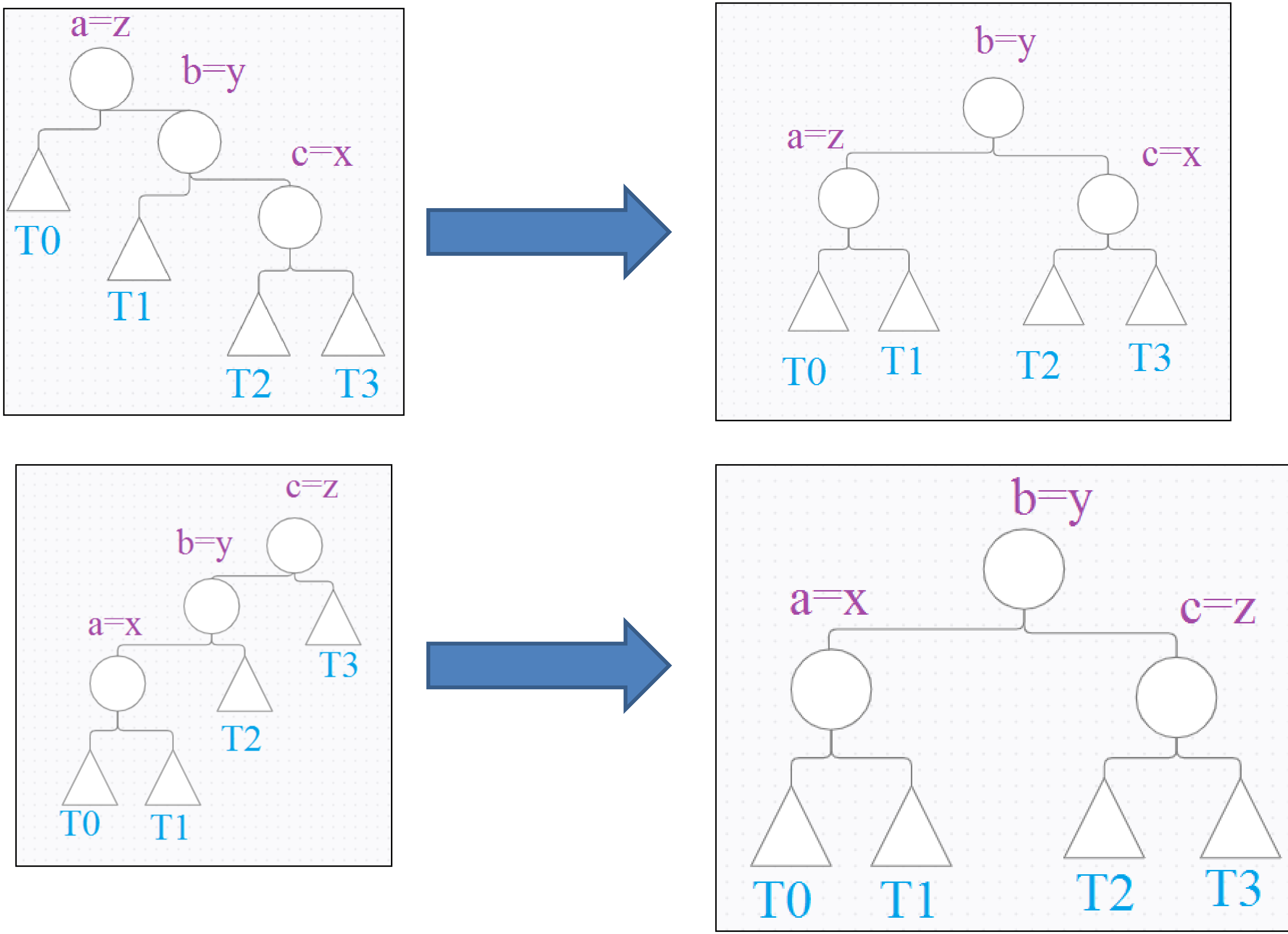
- A **non-unique-key-based data structure** for storing entries made of key-value pairs.
- Includes the following standard methods:
  - **Get**: Returns an entry with a given key.
  - **getAll**: Returns a collection of all entries with agiven key.
  - **Put**: Creates and adds a new entry with a given key and value into the dictionary.
  - **Remove**: Removes a given entry from the dictionary, and returns it as proof.
  - **entrySet**: Returns a collection of all entries.
  - **isEmpty**: Return whether the dictionary is empty.
  - **Size**: Return the number of entries.

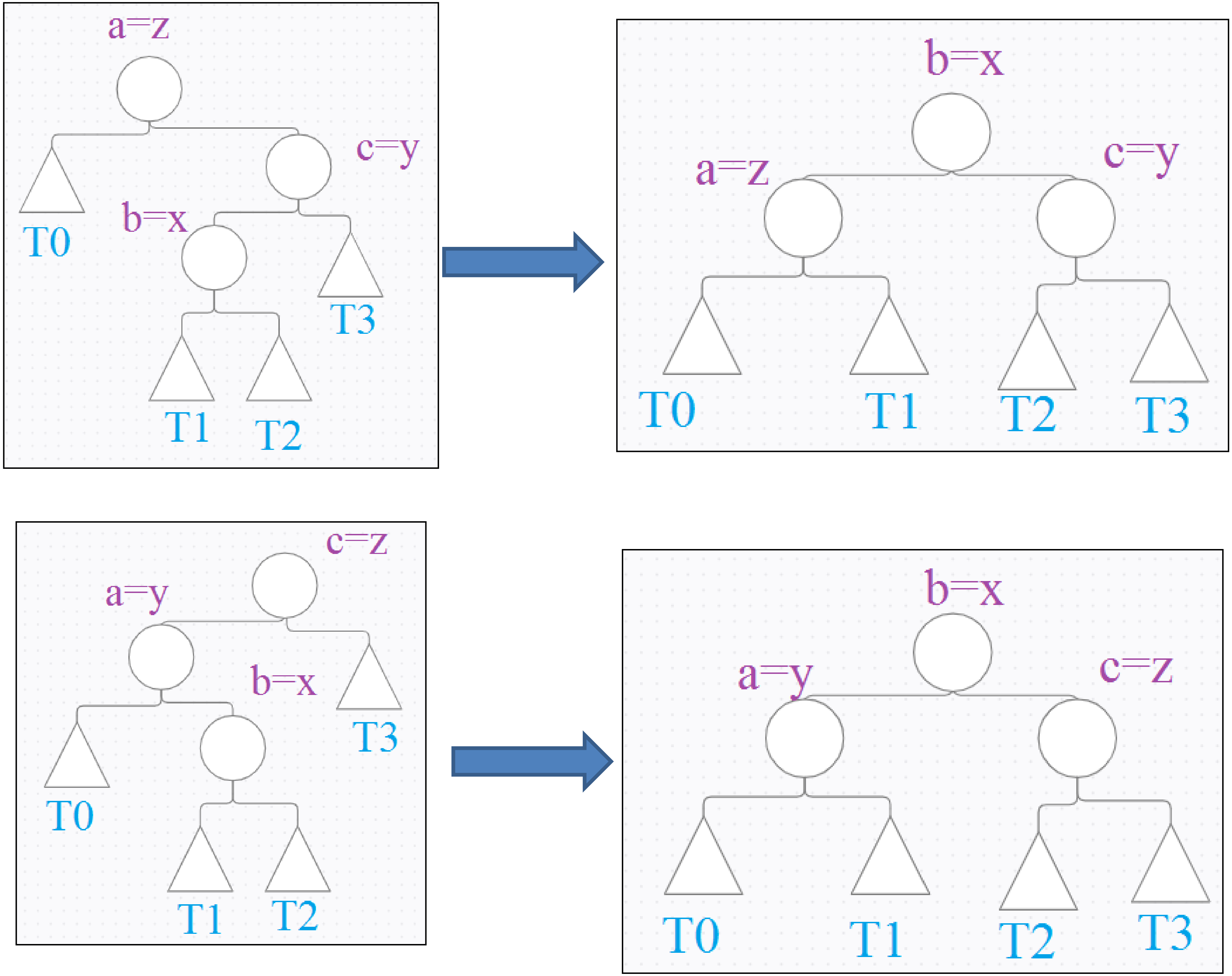
Binary Search Trees

- **Binary** means each node in the Tree can have 0, 1, or 2 children.
- **Search** means the left children's keys are less than the parent, while the right children's keys are greater.
- Implements the Binary Search insight, making them suitable for Ordered Maps and Dictionaries through the use of **Tree Search**, an **in-order traversal**.
- Don't forget the **blank external nodes**.

AVL Trees

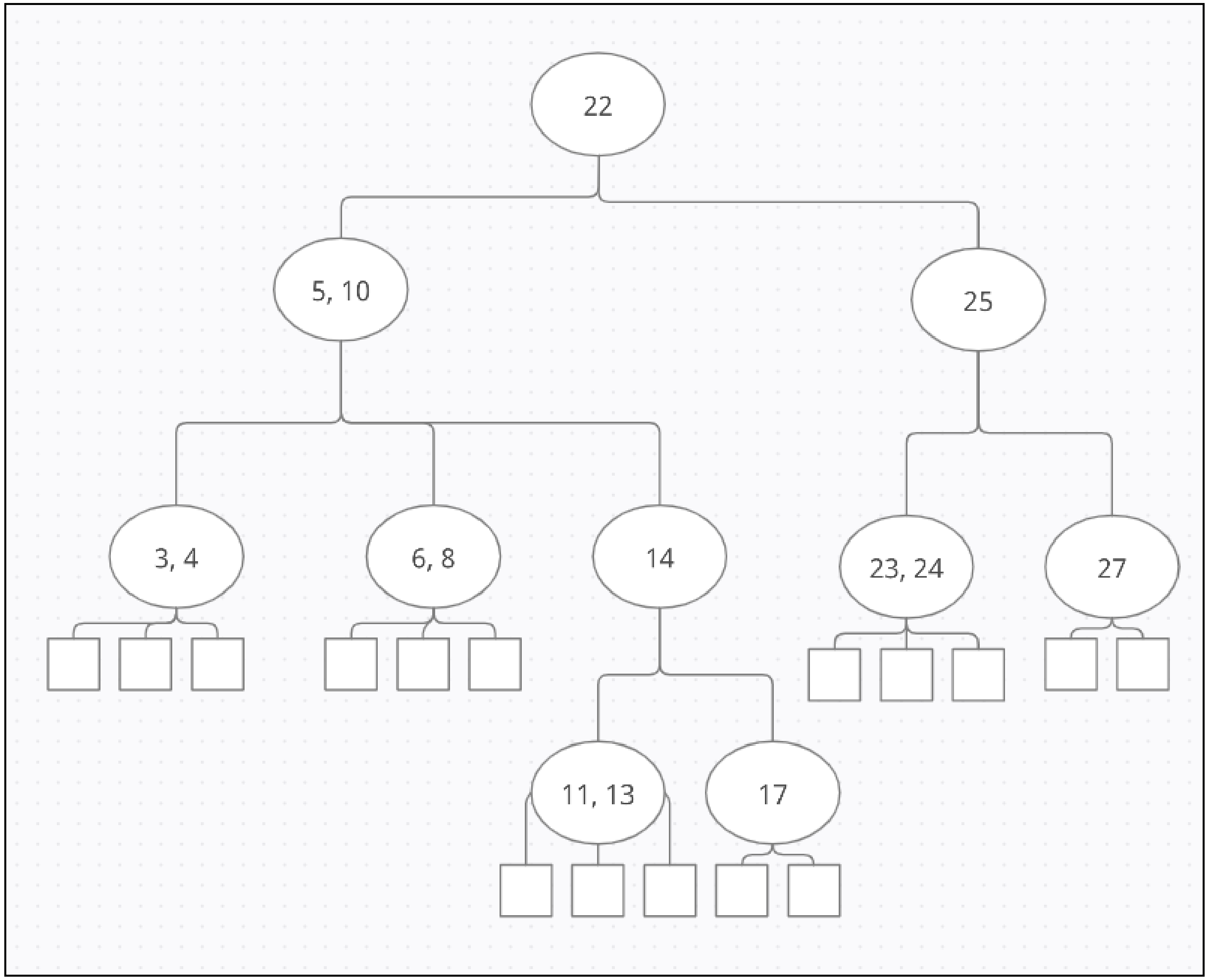
- A **self-balancing** extension of Binary Search Trees.
- Solves the issue that while Tree Search **bounds search times to the height of the Tree**, nothing was keeping the height of the Tree in check.
- Adds the **Height-Balancing Property** to **Rebalance** the Tree after actions that might unbalance it.


a=z
b=y
c=x
T0
T1
T2
T3
b=y
a=z
c=x
T0
T1
T2
T3
c=z
b=y
a=x
T3
T2
T0
T1
b=y
a=x
c=z
T0
T1
T2
T3


a=z
c=y
b=x
T0
T3
T1
T2
b=x
a=z
c=y
T0
T1
T2
T3
c=z
a=y
b=x
T3
T0
T1
T2
b=x
a=y
c=z
T0
T1
T2
T3

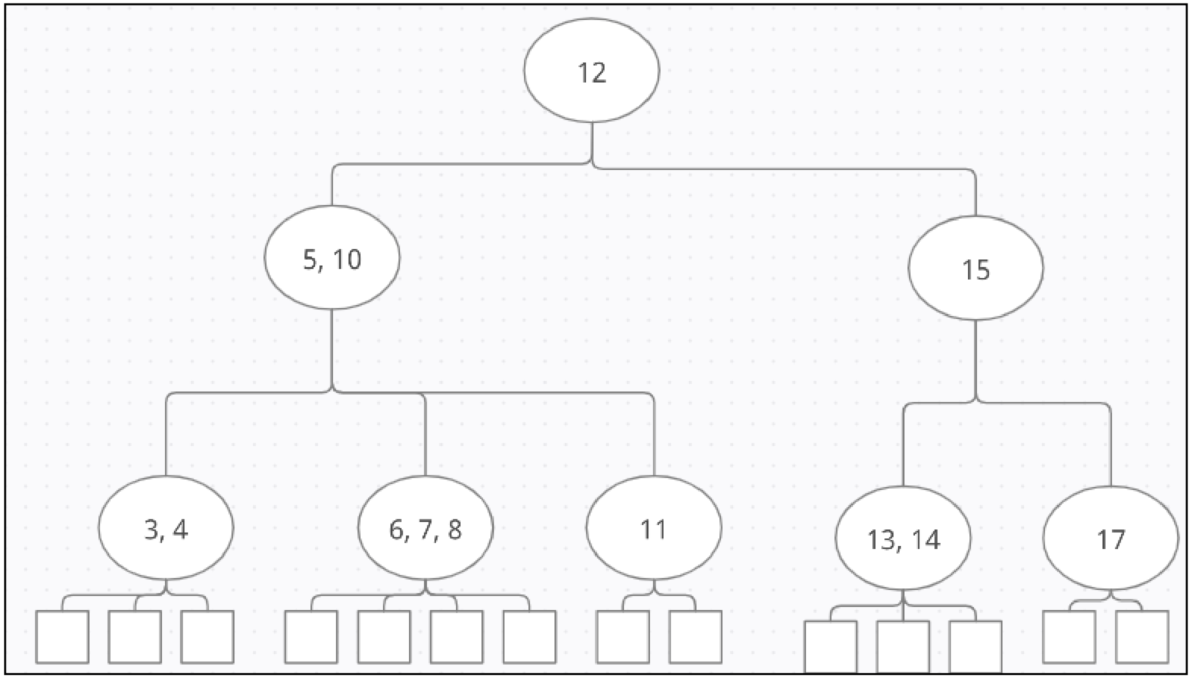
Multi-Way Trees

- Trees that can have **more than one entry per node**, and **many children**.
- By enforcing an ordering on the entries and children, we create the **Multi-Way Search Tree**.

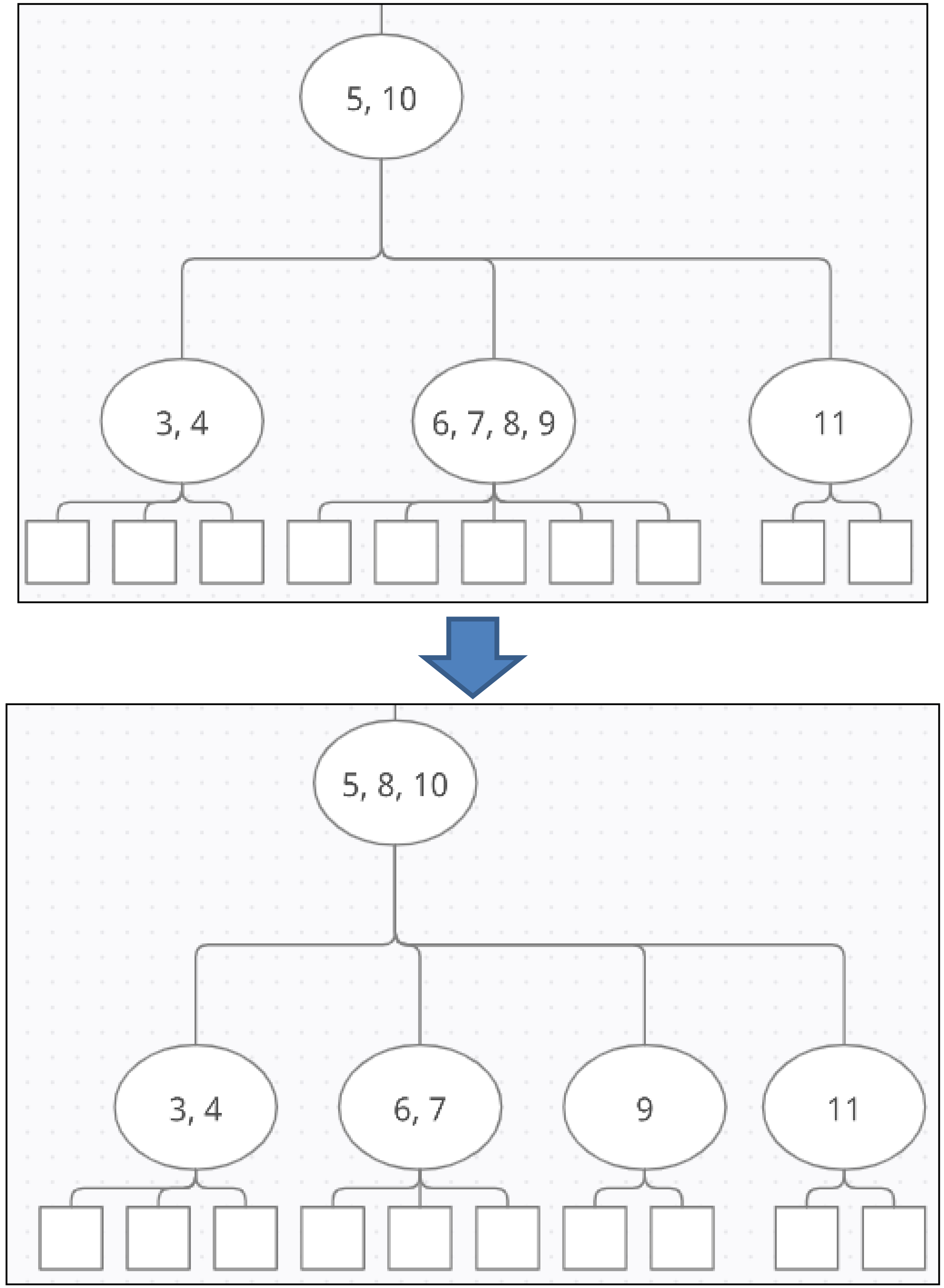


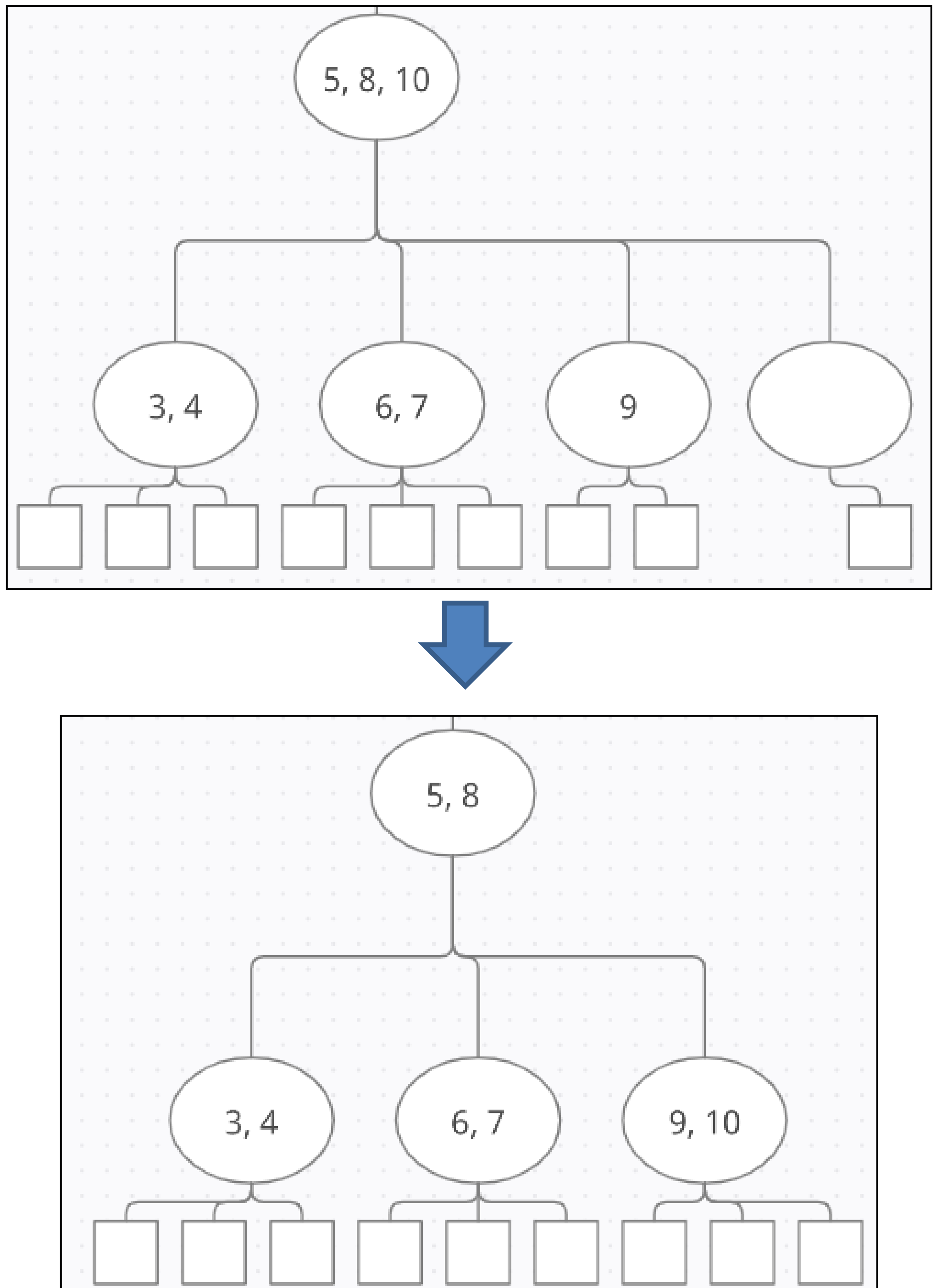
2-4 Trees

- A self-balancing variant of Multi-Way Search Trees that **limit the number of children a node may have to 2, 3, or 4**, and requires **all external nodes** to have the **same depth**.

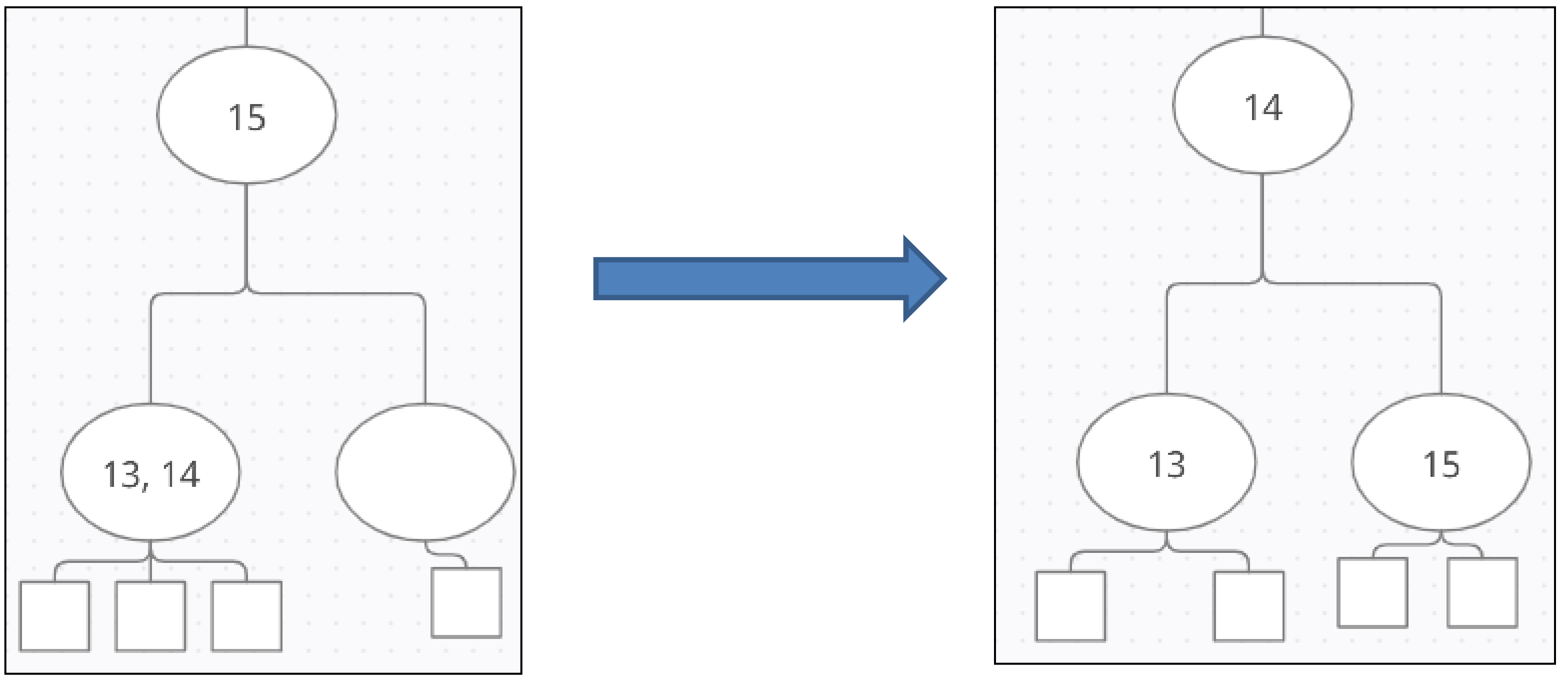


- This has the same effect of **restraining the height of the Tree to log n** that AVLs have on BSTs.
- Adding and removing now has to rebalance the Tree in the case of **overflows** and **underflows**.


5, 10
3, 4
6, 7, 8, 9
11
5, 8, 10
3, 4
6, 7
9
11



5, 8, 10
3, 4
6, 7
9
5, 8
3, 4
6, 7
9, 10


15
13, 14
14
13
15

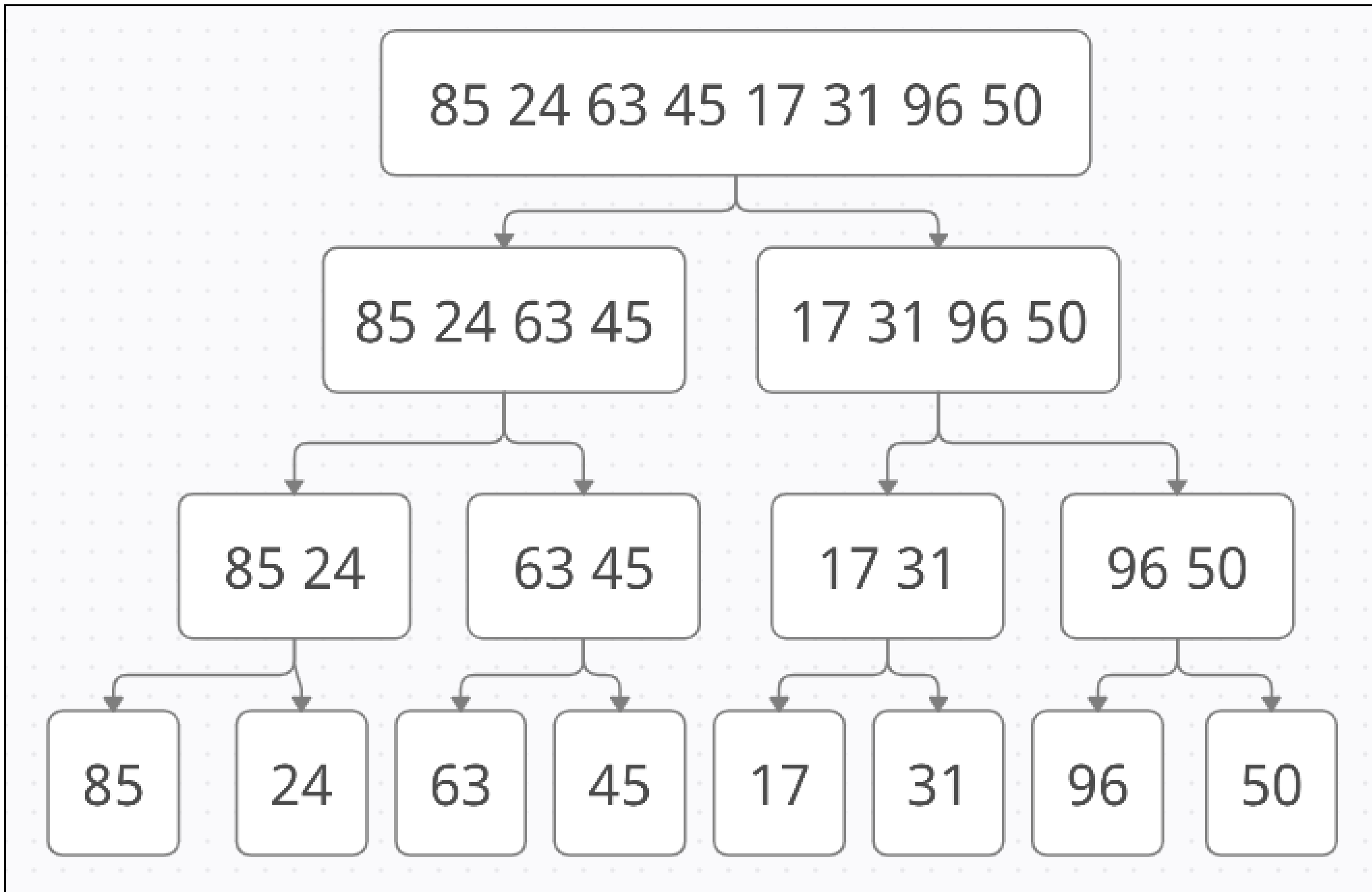
Interlude on Advanced Trees

- **Self-Balancing Trees** are good candidates for implementing some Maps, like Ordered Maps.
- Java includes **TreeMap** and **TreeSet**, for a Map (and later, a Set) based on a **Red-Black Tree**, which is another type of self-balancing Tree we didn't discuss (so you won't be tested on).

Merge-Sort

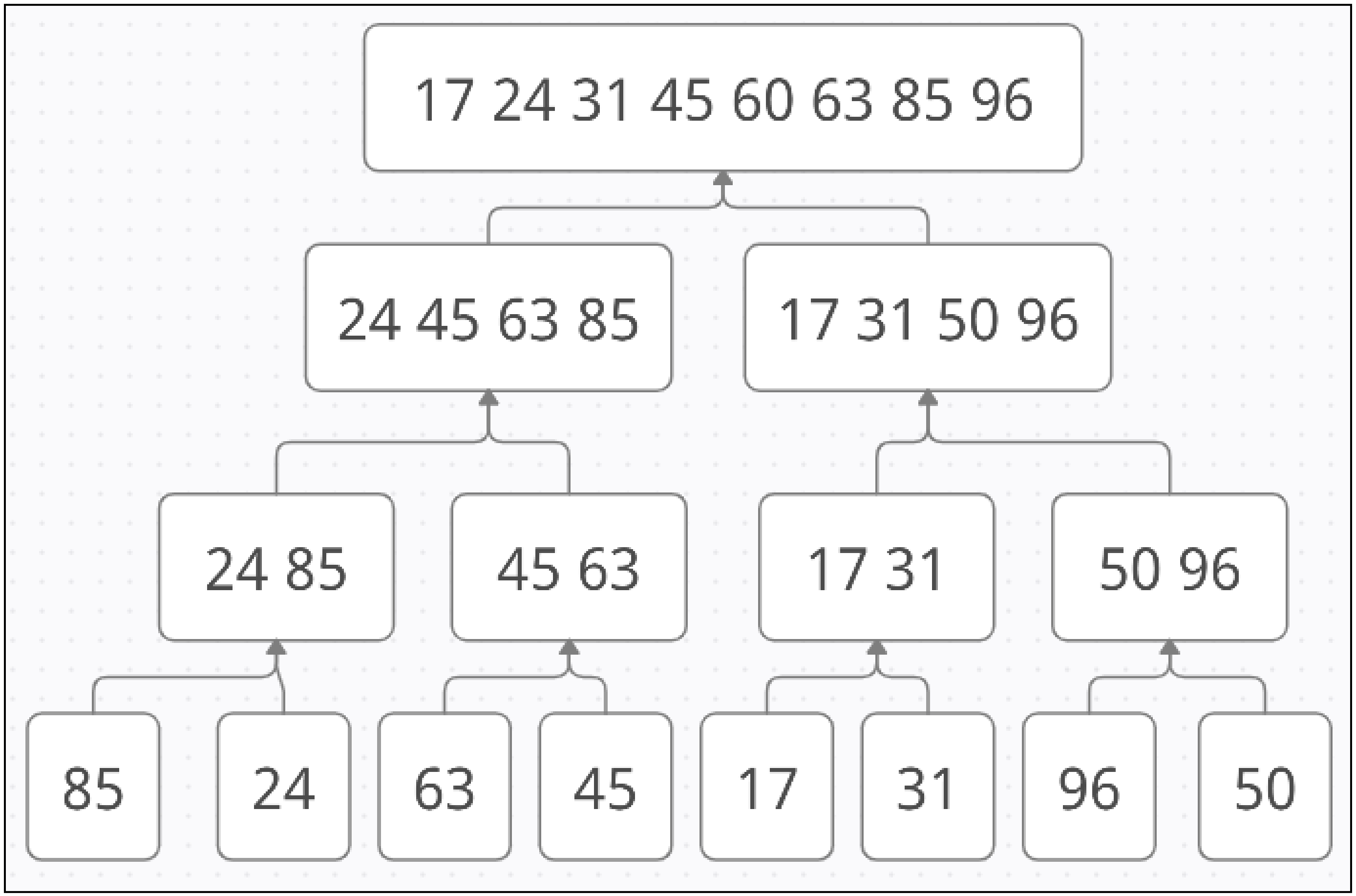
- A **comparison-based**, **Divide-and-Conquer** sorting solution.
- **Divides** the input into halves and **recurses** until every element is alone, then **sorts** while putting them back together again.
- **O(n log n)**, because each “layer” of the **Merge-Sort Decision Tree** takes O(n) and the height of the Tree is O(log n)

Divide...



85 24 63 45 17 31 96 50
85 24 63 45
17 31 96 50
85 24
63 45
17 31
96 50
85
24
63
45
17
31
96
50

…and Conquer


17 24 31 45 60 63 85 96
24 45 63 85
17 31 50 96
24 85
45 63
17 31
50 96
85
24
63
45
17
31
96
50

You Know It's Time For This Guy

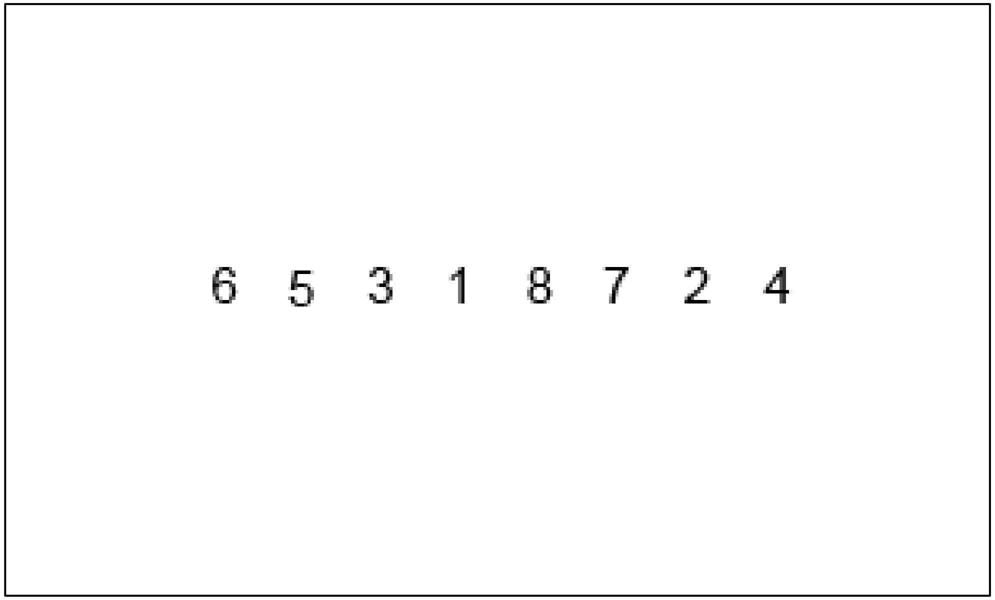
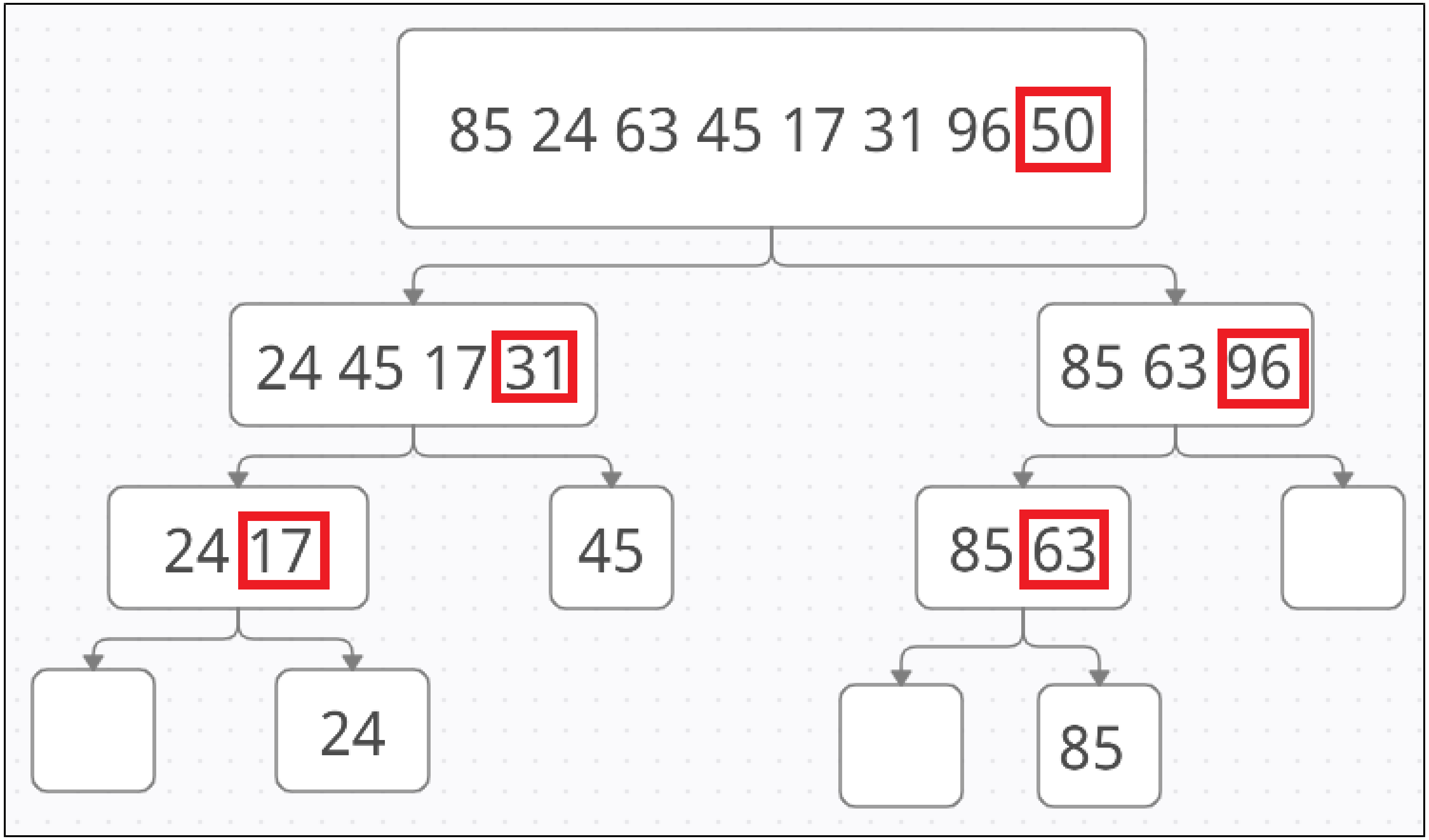


Image credit: https://en.wikipedia.org/wiki/File:Merge-sort-example-300px.gif

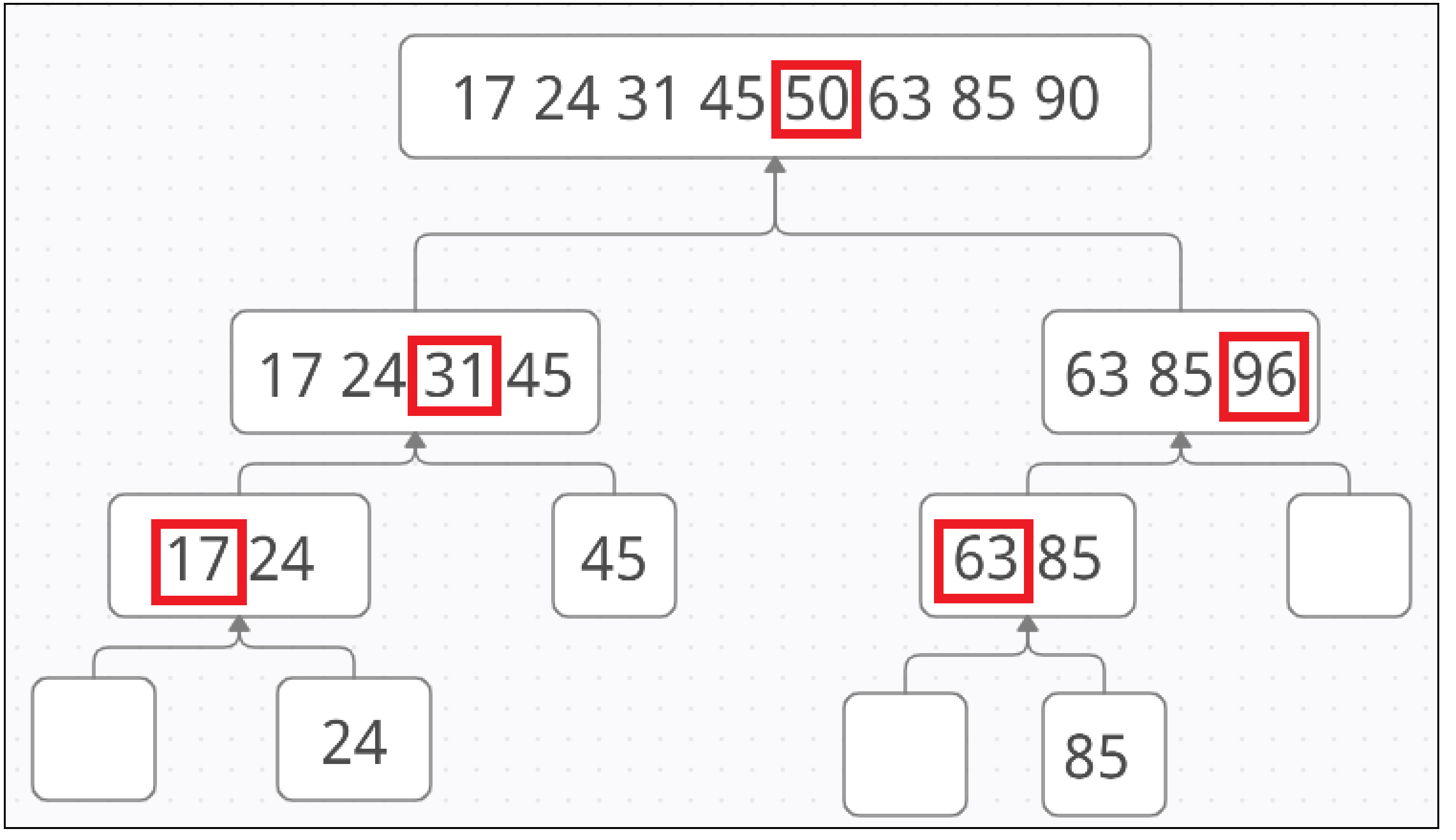
Quick-Sort

- A **comparison-based**, **Divide-and-Conquer** sorting solution.
- **Divides** the input according to a randomly chosen pivot value and then **recurses**, so that when it's time to start putting things back together again they're **already in order**.
- **O(n log n), probably**, if the **randomly-chosen pivot** works out.

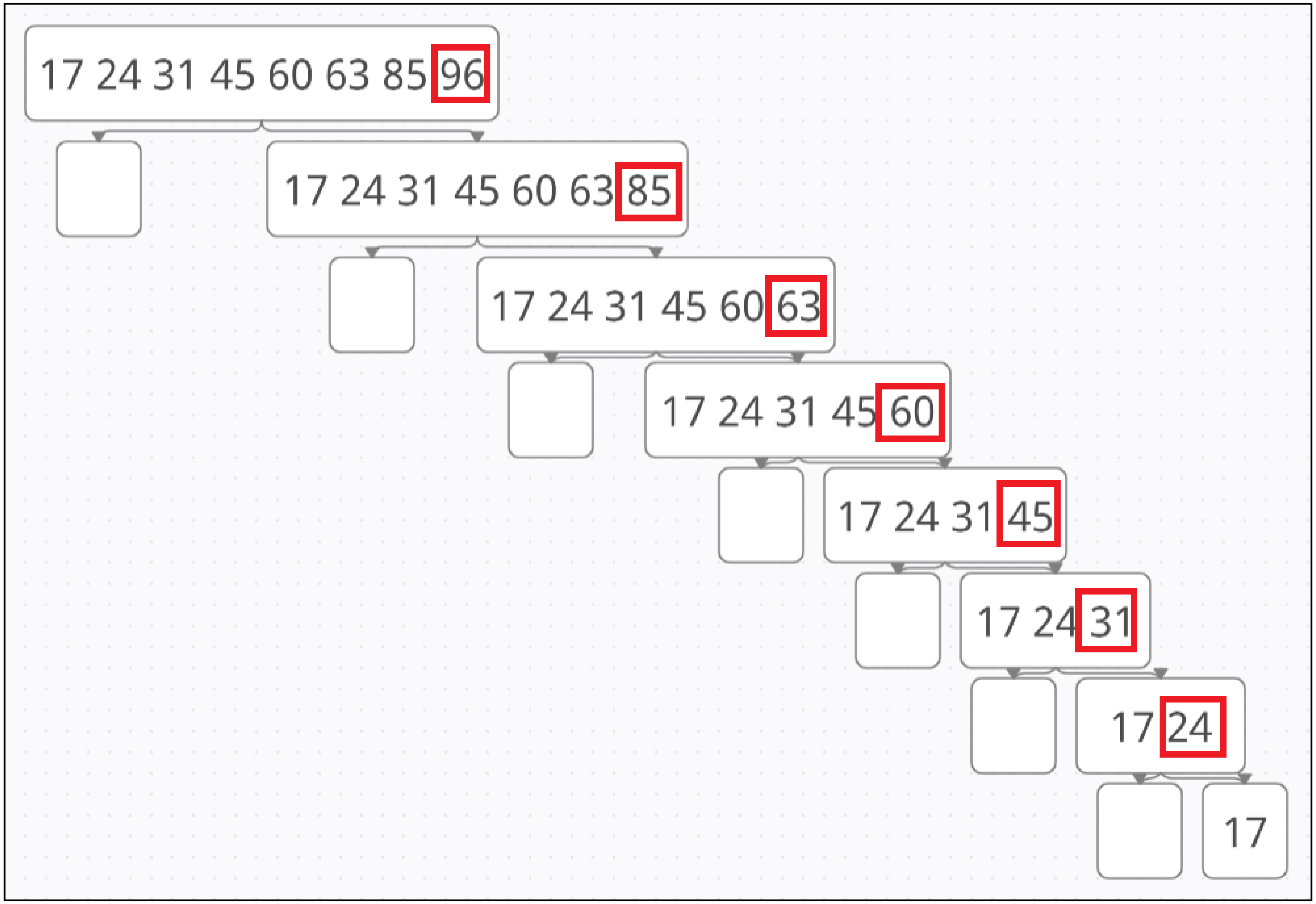
Divide…



...and Conquer



Consider a Worst-Case



Maybe By Now You Can Tell What's Going On Here

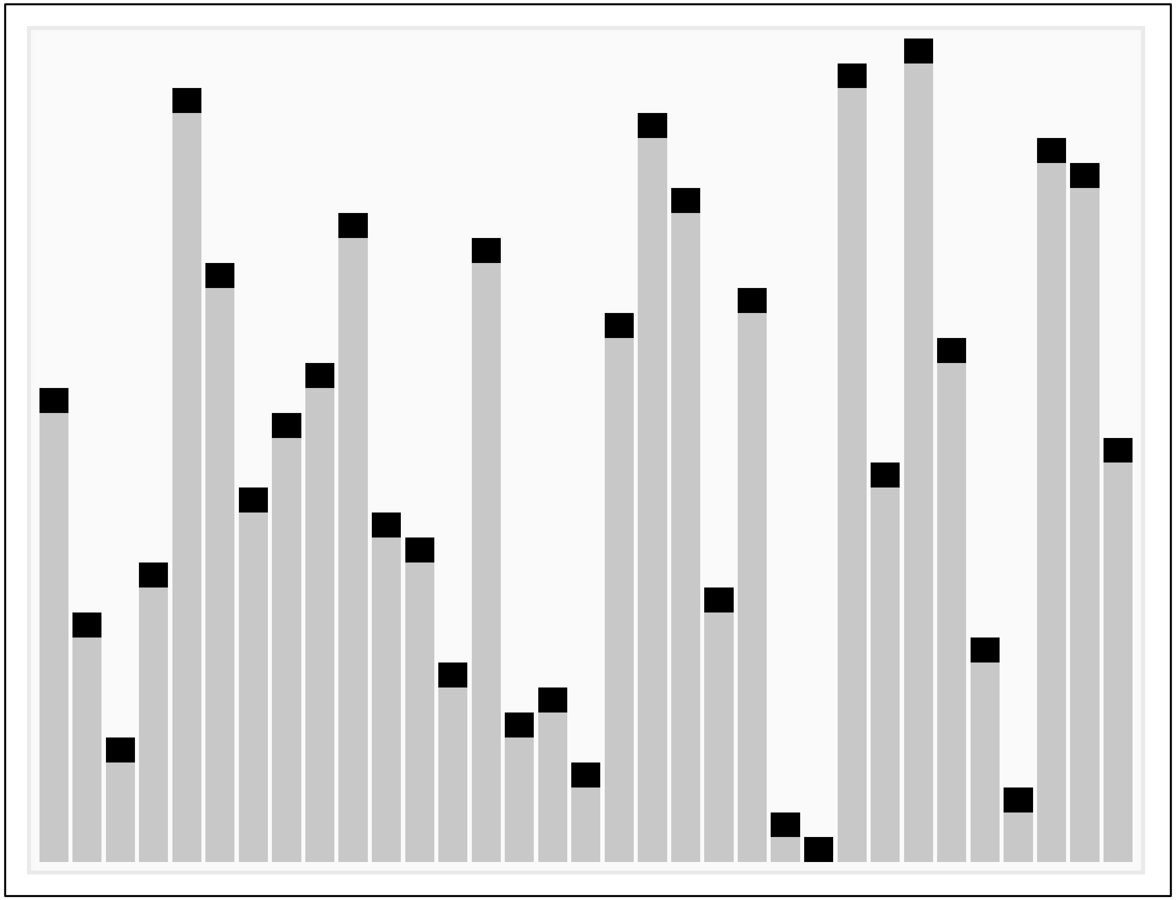


Image credit:
https://upload.wikimedia.org/wikipedia/commons/6/6a/Sorting_quicksort_anim.gif

Bucket & Radix-Sort

- If you **limit the input range**, you can use **Bucket-Sort** to match an input directly to a bucket, **without requiring comparisons**.
- **Radix-Sort** allows for sorting according to multiple terms by layering Bucket- Sorts in reverse order (e.g. alphabetical order).
- This can get sorting times down to **O(n).**

Sorting Generally

- Don’t forget **Insertion-Sort** (check Unit 4) and **Heap-Sort** (implicitly what a Heap does).
- Sorting **Stability** is the property of whether a particular Sort will keep the relative positions of equal elements (i.e. will an already-sorted sequence come out in the same order every time).
- Sorting **In-Place** affects how much memory space needs to be available for a Sort to work itself out, since those that can be implemented in-place don’t require any additional space.

Examples for Sorting In-Place

- **Insertion-Sort** can be done in-place, simply by moving the entries around within the sequence being sorted. This is how the version we introduced in section 4 works.

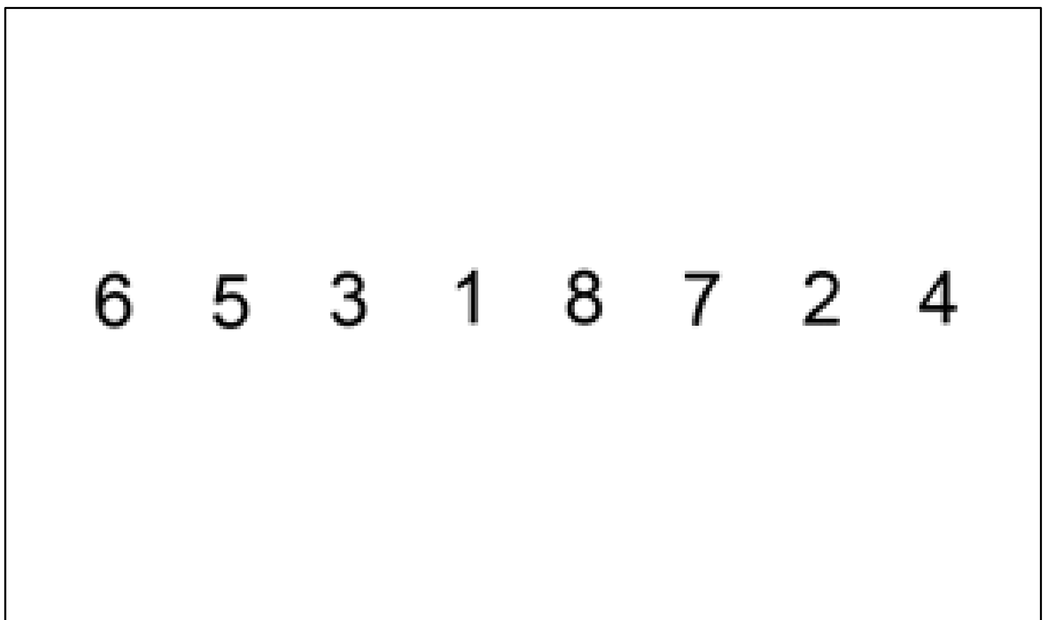


Image credit: https://upload.wikimedia.org/wikipedia/commons/0/0f/Insertion-sort-example-300px.gif

Summary of Sorting Algorithms

- Let's do a quick run-down of our options:
  1. **Insertion-Sort** ($O(n^2)$, stable, in-place)
  2. **Merge-Sort** (O(n log n), stable, not in-place)
  3. **Quick-Sort** (O(n log n)*, unstable, in-place)
  4. **Heap-Sort** (O(n log n), unstable, in-place)
  5. **Bucket-Sort/Radix-Sort**(O(n+N)/O(d(n+N)), stable, not in-place).

Sets

- A **high-level data structure** that is simply made up of elements and doesn't care how they're stored, added, removed, etc.
- Mainly concerned about whether or not an element is a **member** of the Set or not.
- All elements in a Set are **unique**.
- **Ordered Sets** let you establish some relative order between elements, while **Mergable Sets** let you combine different Sets through Unions, Intersections, and Subtractions.
- A **Partition** is a collection of Sets with no elements in common with each other.

The Set ADT

- A collection of distinct objects.
- Extremely general – no explicit notion of keys or even an order, elements are just part of the set or not part of the set.
- Include the following standard methods:
  - **Add**: Adds a given element to the set.
  - **Remove**: Removes (but does not return) a given element from the set.
  - **Contains**: Tells you whether a given element is in a set or not.
  - **Iterator**: Returns a collection of the elements in the set.

The Ordered Set ADT

- An extension of Set that includes support for a total ordering of the elements.
- The standard methods are expanded:
  - **pollFirst**: Return and remove the smallest element.
  - **pollLast**: Return and remove the largest element.
  - **Ceiling**: Return the smallest element that's greater than or equal to a given element.
  - **Floor**: Return the largest element that's less than or equal to a given element.
  - **Lower**: Returns the greatest element less than a given element.
  - **Higher**: Returns the smallest element greater than a given element.

The Mergable Set ADT

- A Set which supports being combined with other Sets.
- Adds the following methods:
  - **Union**: Replaces the set with the union of itself and a given set.
  - **Intersect**: Replaces the set with the intersection of itself and a given set.
  - **Subtract**: Replaces itself with the difference of itself and a given set.

The Partition ADT

- A collection of disjoint Sets.
- Reintroduces the notion of positions to Sets,
- Includes the following standard methods:
  - **makeSet**: Create a single-element set out of a given element and returns the position storing it.
  - **Union**: Returns the union of two given sets, while removing them.
  - **Find**: Returns the set containing the element in a given position.

Union-Find Structures

- A data structure equivalent to (but distinct from) a Partition of Sets.
- Uses a combination of Union and Find operations to build a data structure up from individual elements to the desired number of disjoint Sets.

Selection

- Finding an element in a Set according to its **rank** (smallest, largest, median, etc) is called **Order Statistics**.
- The generalized form is called the **Selection Problem**.
- **Prune-and-Search** is a design pattern we can apply to this problem to create an algorithm, **Quick-Select**, that solves the problem.
- Quick-Select is essentially Quick-Sort for finding an element by its rank.

Recap – The End of the Course, but Not the End of the World

- The Midterm is on **Monday at 12:00pm** and must be submitted by 3:00pm – three hours!
- It's on **Canvas**, with a mix of theory and coding questions, so **open up your IDE**.
- It is **cumulative**, covering material both before and after the midterm.
- It's **open book**, but **no cooperating with others** or lifting solutions directly from the internet. ***<u>Cite any sources used</u>***.
- I'll be available on **Discord** and in the **virtual lecture room** if you need me!