CMPT 225: Data Structures & Programming – Unit 19 – Ordered Maps

Dr. Jack Thomas

Simon Fraser University

Spring 2021

Today's Topics

- Reintroducing Order to Keys
- Retrieving Subsets
- Ordered Maps ADT
- Ordered Maps in Java
- Binary Search

Keys and Ordering

- At first, the move **from position-based to key-based data structures appeared to remove ordering**, as the relative position of each entry in the structure is no longer meaningful.
- **Priority Queues re-introduced the idea at least partially**, since the entry at the front of the queue must be the highest-priority one according to their key.
- However, neither Priority Queues, Maps, nor Hash Tables so far have **supported a full ordering of their entries by their keys**.

Retrieving Subsets

- Many position-based data structures let us retrieve not just one entry, but **a whole subset of neighbours** – ArrayList.subList(2, 5) for example.
- If those structures are sorted, like an array of numbers from smallest to largest or an alphabetical list of names, **these subsets can also be useful** – all products under $100, or all students whose names start with B.
- This may be something we'd like to do with a Map as well!

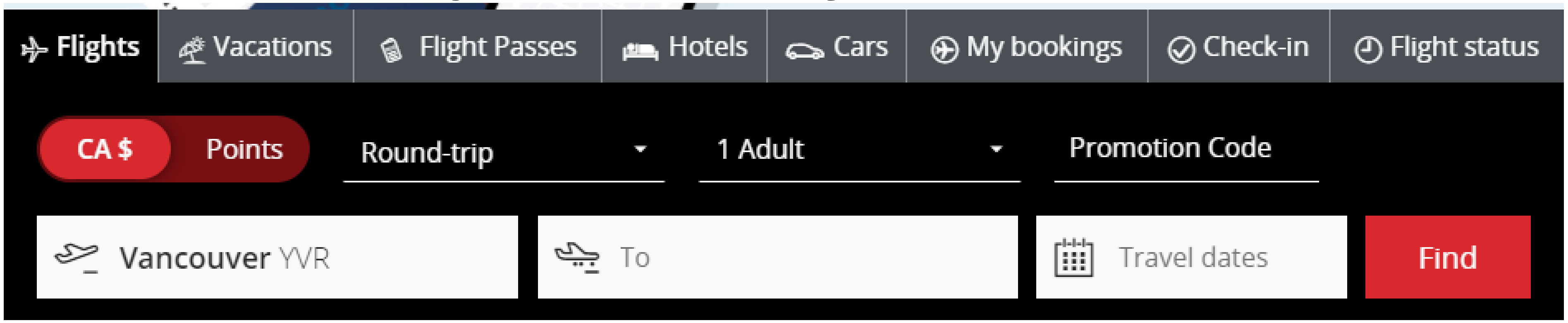
It’s Airports Again

- I think I just don’t want to find a new picture.
- Still, imagine you’re looking to buy a ticket to fly somewhere.

Image credit: https://www.internationalairportreview.com/article/81738/smarter-way-cut-queue/

Retrieving All Flights That Fit Your Criteria

- If you want to **buy a ticket for a particular flight**, it's easy to imagine that as a key-based system – a combination of the departure city, arrival city, and time, could be a unique key.
- What if you don't want a particular flight, but rather **every flight between two particular cities on a particular day**?



Can We Do This Already?

- A **Priority Queue isn't well-adapted** for this, and a **Hash Table** would already need to know which individual flights fall within the criteria to **retrieve each of them one at a time**.
- What we need is a way to **order the flights** chronologically according to our departure-arrival-time key and **return the subset between two given keys** (midnight to midnight)

Enter the Ordered Map

- The **Ordered Map** is a sub-type of Map that can not only return an entry with a given key, but also a **set of entries whose keys fall between two keys**, given some way to order the keys.
- As such, the chosen key for an Ordered Map must produce a **total ordering**. This may be the result of a **natural ordering** (such as an integer key going smallest to largest) or the result of a **comparator** (a rule for how to compare two objects, like the BirthdayEntry from our Priority Queue lab).

Where Ordered Maps Fit

- Ordered Maps are a **variety of Map** (and typically implements Map as an interface), but they are **not a full implementation** the way a Hash Table is.
- In fact, a Hash Table or an un-ordered list would be an **unsuitable way to implement an Ordered Map**.
- We'll discuss some data structures that **implement and build on Ordered Maps** later.

Ordered Map ADT

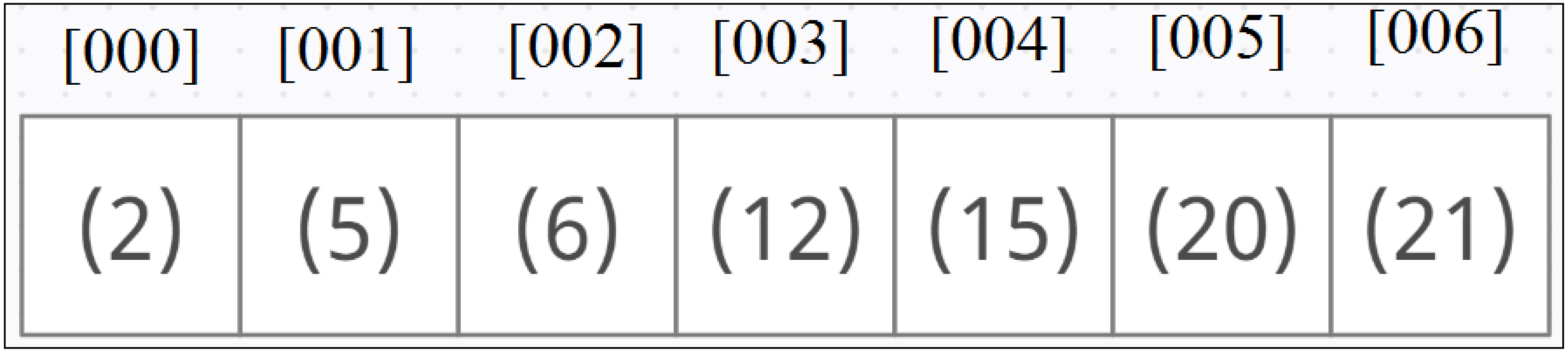
- A key-based data structure that can return entries based on the relative ordering of its entries' keys.
- Standard methods include the ones for any Map, as well as:
  - **firstEntry**: Returns the entry with the smallest key.
  - **lastEntry**: Returns the entry with the largest key.
  - **ceilingEntry**: 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.

Ordered Maps in Java

- Unfortunately, **Java doesn’t have a standard Ordered Map** class or interface.
- There are a few classes that fulfill similar purposes, like the **SortedMap** interface or the **LinkedHashMap** class, but these have slight differences owing to the fact that they’re not direct implementations of the Ordered Map ADT.

Ordered Search Table

- Technically, much like Maps and Priority Queues, Ordered Maps **don’t lay out how the data within is actually stored**.
- If we have a total order of keys, however, then we could use an **ArrayList** as the underlying data structure, since it will enable fast searching for keys between certain criteria.
- These are called **Ordered Search Tables**, and yes, we’ve sort of come full circle.



Finding Our Keys

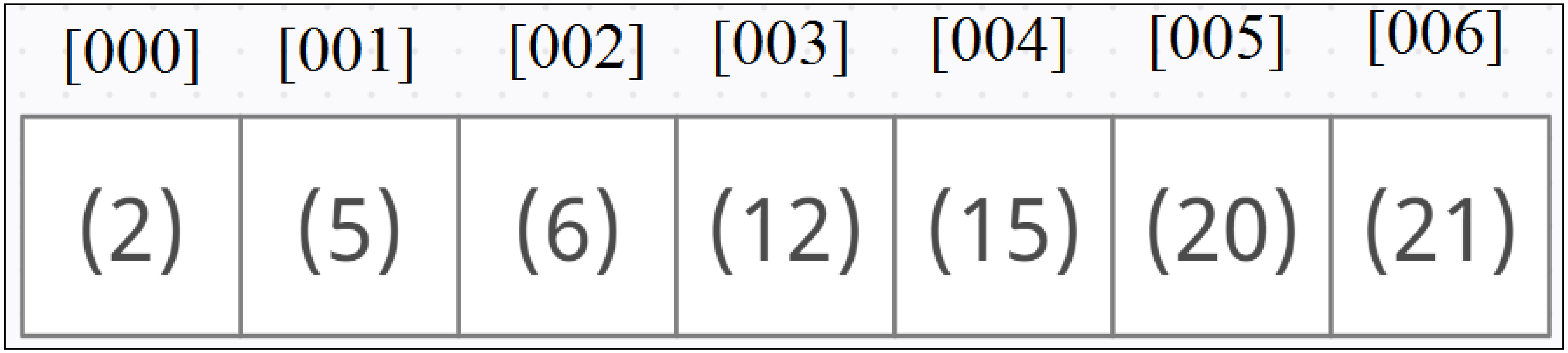
- An Ordered Search Table makes accessing any one key fast, but now we need to find multiple keys – in the worst case, all n keys, if the bounds of our subset cover all our entries.
- Thankfully, **once we find either our upper or lower bound**, we just need to package together and return every neighbour on our way to the other bound, which is way more efficient than tracking down each key across an unsorted set.
- We do need to **find that first bound**, however, and can we do better than O(n) on that?

Binary Search

- A straightforward way to search for a key in a sorted set that lets you access any position directly (like our ArrayList-based Ordered Search Table).
  1. Jump to the middle element. Is it what you're looking for? Great! You're done.
  2. If if's larger than what you're looking for, jump to the middle element between it and the front.
  3. If it's smaller than what you're looking for, jump to the middle element between it and the back.
  4. Repeat until the key is found or you've found where it would've been if it were in the set.

Binary Search Example

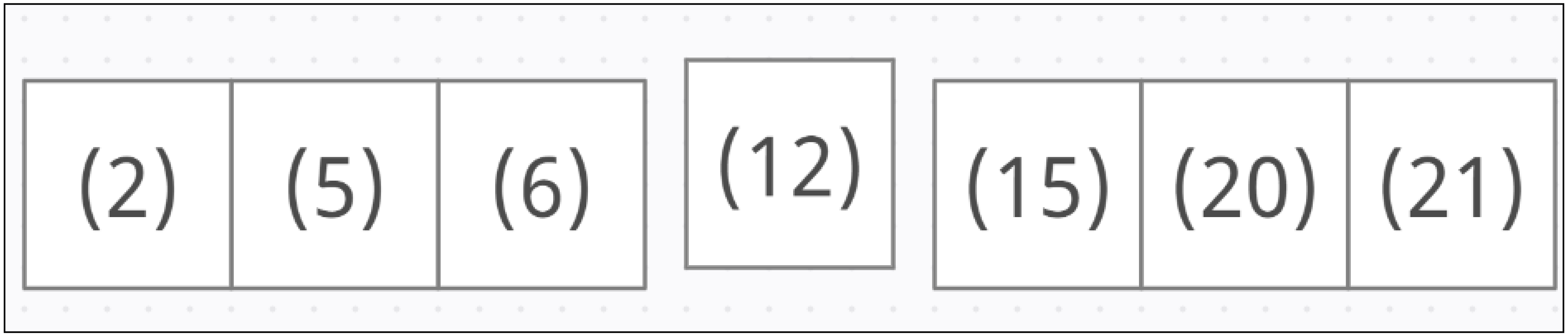
- Say we’re given the following Ordered Search Table and are looking for an entry with a key of 20.



[000]	[001]	[002]	[003]	[004]	[005]	[006]
(2)	(5)	(6)	(12)	(15)	(20)	(21)

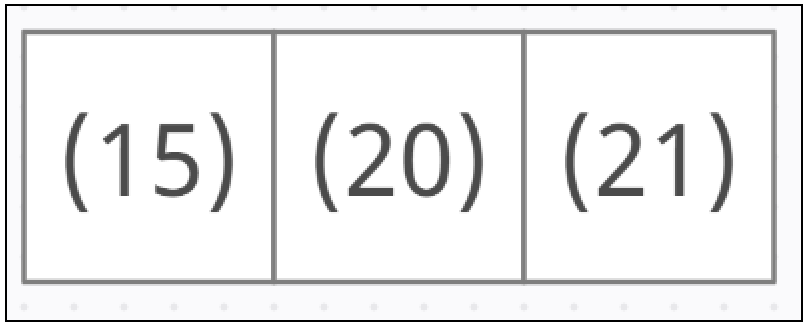
Binary Search Example

- First let’s look at the key in the middle of the array – it’s smaller than what we want, so we know we should look somewhere between it and the last entry.

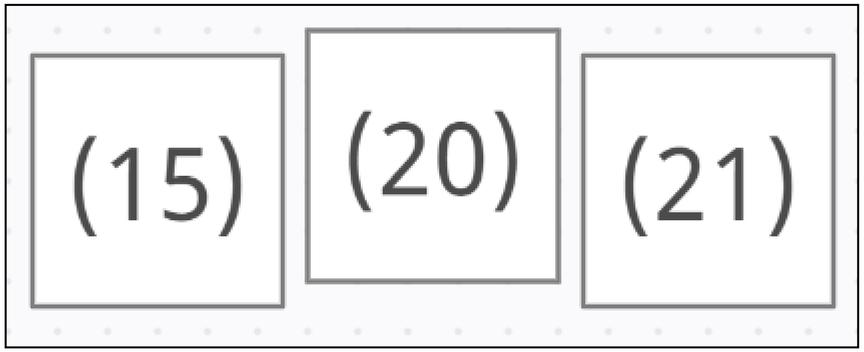


Binary Search Example

- One we've separated out our subset (maybe a recursive call? A loop?), we repeat the process.



- We look at the middle entry again, and presto!



```
Algorithm BinarySearch(S,k,low,high):
    Input: An ordered array list S storing n entries and
    integers low and high.
    Output: An entry of S with key equal to k and index
    between low and high, if such an entry exists, and
    otherwise null.
    if low > high then
        return null
    else
        mid <- floor((low + high)/2)
        e <- S.get(mid)
        if k = e.getKey() then
            return e
        else if k < e.getKey() then
            return BinarySearch(S,k,low,mid-1)
        else
            return BinarySearch(S,k,mid+1, high)
```

Asymtotic Analysis Update

- Now that we've seen list-based Maps, Hash Tables, and Ordered Search Tables, let's remind ourselves of their relative efficiencies.

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)

Recap – An Ordered Summary

- **Ordered Maps** are a version of Map that allows for returning ordered subsets of entries according to their keys.
- Java has **no Ordered Map interface or class**, but its ideas are represented in other interfaces and classes.
- **Ordered Search Tables** are an ArrayList-based version of Ordered Maps.
- **Binary Search** is a classic algorithm for finding a particular value in a sorted set.