Quicker Quicksort
Sorting Stability
Hashing and Maps

CS206
April 22

MergeSort & QuickSort

- Inefficient on small lists
  - lots of recursive calls (on very small pieces) eat time
- Idea:
  - Rather than make recursive calls down to size 1 cut off recursion earlier and use insertion sort

Hybrid MergeSort

put insertion sort into the base case and make the base case bigger

Empirically, 10-15 works best

```
    private void doMergeSort3i(int lowerIndex, int higherIndex) {

        if (lowerIndex > (higherIndex-12)) {
            iSort.insertionSortIP(array, lowerIndex, higherIndex);
        } else if (lowerIndex < higherIndex) {
            int middle = lowerIndex + (higherIndex - lowerIndex) / 2;
            // Below step sorts the left side of the array
            doMergeSort3i(lowerIndex, middle);
            // Below step sorts the right side of the array
            doMergeSort3i(middle + 1, higherIndex);
            // Now merge both sides
            mergeParts3(lowerIndex, middle, higherIndex);
        }
    }
```

Hybrid Quicksort

For quicksort, it is quicker to cut off early (as with mergesort) but run insertion sort once on at the end.

```
    public int[] qs4i(int inputArr[]) {
        doQS4i(inputArr, 0, inputArr.length-1);
        new Insertion().insertionSort2(inputArr);
        return inputArr;
    }

    private void doQS4i(int arr[], int begin, int end)
    {
        if ((end-begin) < 15 ) {
            // just let it drop
        } else {
            int partitionIndex = partition(arr, begin, end);
            doQS4i(arr, begin, partitionIndex-1);
            doQS4i(arr, partitionIndex+1, end);
        }
    }
```

Empirically, 10-15 works best

Stability

- Suppose you have multiple things on which to sort. Eg spreadsheet columns
  - Ties in column B should be sorted by column A
- Can do this with two sorting passes if the sort is “stable”.
- Mergesort is stable
- Quicksort is not

The student class

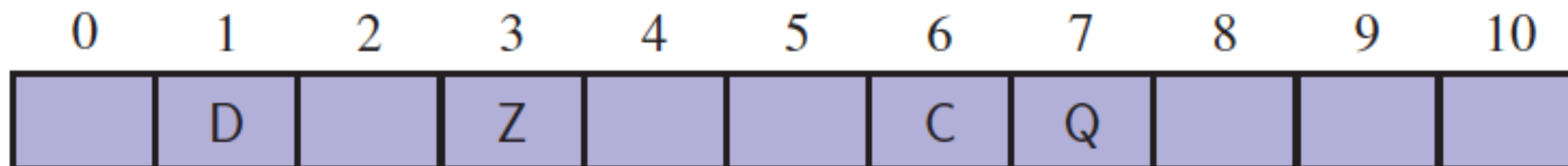
- Comparators for name and age
- Static methods
  - Are not always evil
    - Are reasonable when the return value of the method is dependent ONLY on arguments to the method
      - Should be used carefully!!!!!!!!!!
    - switch to VSC for student, mergeOb and qOb

Map

- A searchable collection of key-value pairs
- Multiple entries with the same key are not allowed
- Also known as dictionary (python), associative array (perl)

Notion of a Map

- Intuitively, a map `M` supports the abstraction of using keys as indices with a syntax such as `M[k]`.
- Simplest setting is a map with n items using keys that are known to be integers from 0 to $N-1$, for some $N \geq n$.

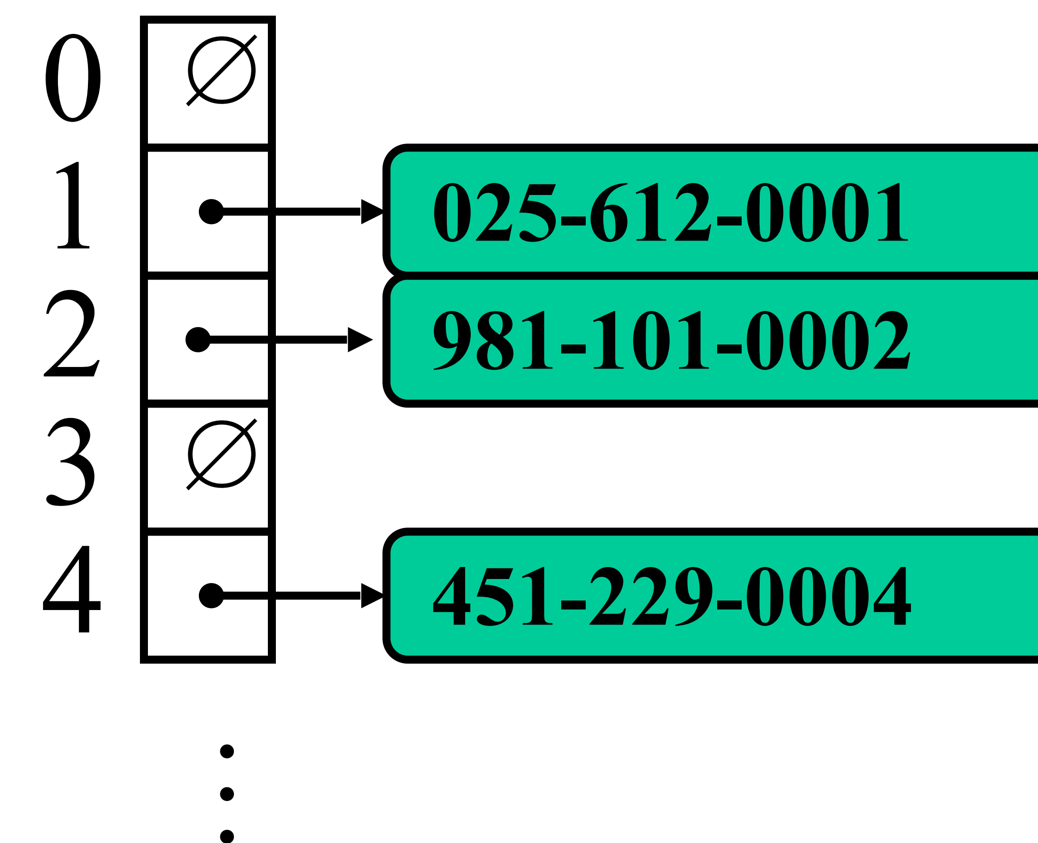


Improving Maps

- Can we tradeoff time and space
  - UnsortedMap implementation
    - efficient spacewise
    - not great timewise
    - So if storing lots of info but accessing rarely, OK
  - But what if storing less and access often?
  - Can we get O(1) time for get/set/remove at a cost of space?

More General Keys

- Earlier: motivated Maps with discussion of keys as integers. What if our keys are not integers in range 0 to $N-1$?
- Use a function to map keys to integers into the right range
- Example: Rather than entire SSN, use only last 4 digits



Hash Functions and Tables

- A hash function h maps a key to integers in a fixed interval $[0, N - 1]$
- $h(x) = x \% N$ is such a function for integers
- A hash table is an array of size N
  - associated hash function h
  - item (k, v) is stored at index $h(k)$

Java Hash classes

- HashMap & Hashtable
  - HashMap is quicker (25% in my tests)
  - HashMap is NOT thread safe
- takes a key - value pair (a la priority queue)
  - applies a hash function to the key and stores the object
  - You do not know the hash function
- O(1) time for store and access

Mini-Homework (part 1)
What is the output of main?

// Using the same student class as earlier in lecture

```
public static void main(String args[])
 {
    HashMap<Integer, Student> hm=new HashMap<>();
     for (Student st : Student.getStudents())
         hm.put(st.getYear(), st);
    for (Map.Entry m:hm.entrySet()) {
         System.out.println(m.getKey()+”--"+m.getValue());
     }
 }
```

```
public class Student {
    public static Student[] getStudents()
    {
        Student[] sss = new Student[12];
        sss[0]=new Student("Lisa", 23);
        sss[1]=new Student("Rosie", 22);
        sss[2]=new Student("Charlotte", 22);
        sss[3]=new Student("Synthia", 20);
        sss[4]=new Student("AnnaSophia", 23);
        sss[5]=new Student("Flora", 21);
        sss[6]=new Student("Libby", 21);
        sss[7]=new Student("Rachel", 22);
        sss[8]=new Student("Catherine", 23);
        sss[9]=new Student("Erin", 22);
        sss[10]=new Student("Xinran", 23);
        sss[11]=new Student("Ashley", 23);
        return sss;
    }
}
```

Mini Homework (part 2)

insertion sort for quicksort

1,2,3,4,5,6,9,7,8,10,11,14,13,12,15,16,17,18,19,22,20,21,25,24,22,23

For the data above, how many compare and move operations are required to sort using insertion sort. If the "average" case time for insertion sort is n*n/4 how much faster is it in this, mostly sorted, case