

Hash Tables (cont.); Comparison Sorts

CSE 332 Spring 2020

Instructor: Hannah C. Tang

Teaching Assistants:

Annie Mao

Howard Xiao

Lucy Jiang

Chris Choi

Jade Watkins

Nathan Lipiarski

Ethan Knutson

Khushi Chaudhari

Richard Jiang

Announcements

- ❖ Guest lecture on Friday! Richard Jiang will cover non-comparison sorts
- ❖ Lecture questions: pollev.com/cse332

Learning Objectives

- ❖ Implement Dictionary ADT operations for a separate-chaining hash table and an open-addressing linear-probing hash table
- ❖ Be able to implement InsertionSort, SelectionSort, In-place HeapSort, and MergeSort
 - As well as discuss their characteristics, such as stability, space usage, and more

Lecture Outline

❖ Hash Tables

- Collision Resolution: Open Addressing
 - **Quadratic Probing**
 - Double Hashing
- Collision Avoidance: Rehashing
- Java-specific Hash Table Concerns
- Conclusion

❖ Comparison Sorting

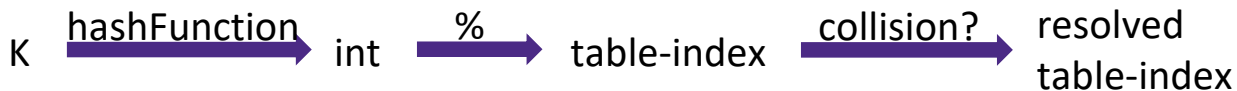
- Simple Algorithms: InsertionSort and SelectionSort
- Fancier Algorithms: HeapSort

Hash Table Components

```
HashTable h;  
h.add("cat", 100);  
h.add("bee", 50);
```

0	-
1	-
2	100
3	50
4	-

- ❖ Implementing a hash table requires the following components:



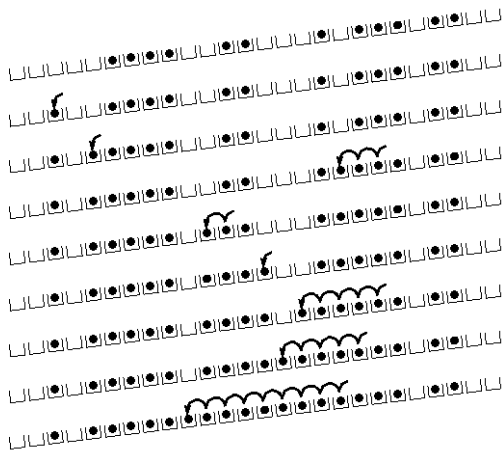
```
hashFunction("cat") == 2;  
2 % 5 == 2  
hashFunction("bee") == 2525393088;  
2525393088 % 5 == 3
```

Linear Probing: Primary Clustering

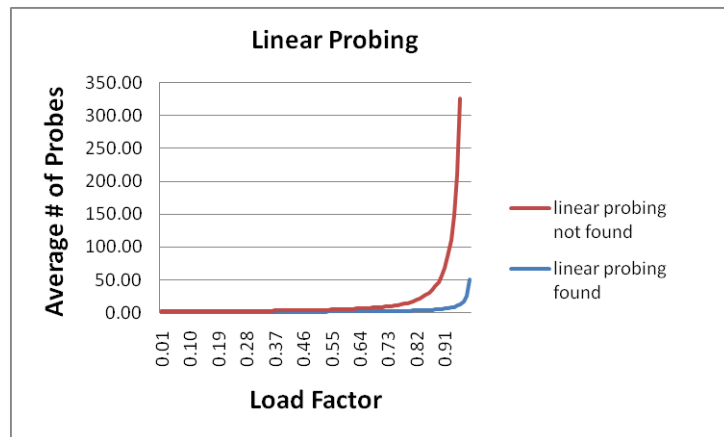
❖ Linear probing uses the following probe function:

■ i^{th} probe: $(h(\text{key}) + i) \% \text{TableSize}$

❖ Which tends to produce long probe sequences due to **primary clustering**



[R. Sedgwick]



Quadratic Probing

❖ Avoid primary clustering by changing the probe function:

■ i^{th} probe: $(h(\text{key}) + i^2) \% \text{TableSize}$

■ Probe sequence becomes:

• 0th probe: $h(\text{key}) \% \text{TableSize}$

• 1st probe: $(h(\text{key}) + 1) \% \text{TableSize}$

• 2nd probe: $(h(\text{key}) + 4) \% \text{TableSize}$

• 3rd probe: $(h(\text{key}) + 9) \% \text{TableSize}$

❖ Intuition: Probes quickly “leave the neighborhood”

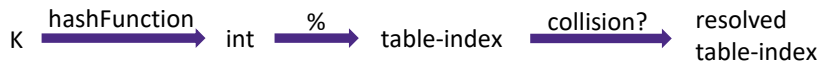
Reminder: a dictionary maps *keys* to *values*;
an *item* or *data* refers to the (key, value) pair

Quadratic Probing: Add Example

❖ Example: add 89, 18, 49, 58, 79

- Let $\text{hashFunction}(x) = x$ ← Bad hash fn
- Let $\text{TableSize} = 10$ ← Bad (non-prime) table size

0	49
1	-
2	58
3	79
4	-
5	-
6	-
7	-
8	18
9	89



Quadratic Probing: Another Add Example

❖ Example: add 76, 40, 48, 5, 55, 47

- Let $\text{hashFunction}(x) = x$ ← Still a bad hash fn
- Let $\text{TableSize} = 7$

0

48

1

-

2

5

3

55

4

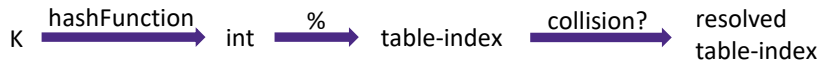
-

5

40

6

76



Quadratic Probing: Another Add Example

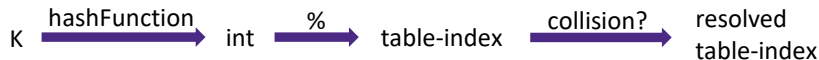
❖ Example: add ~~76, 40, 48, 5, 55, 47~~

- Let `hashFunction(x) = x`
- Let `TableSize = 7`

- $(47 + 1) \% 7 = 6$ collision!
- $(47 + 4) \% 7 = 2$ collision!
- $(47 + 9) \% 7 = 0$ collision!
- $(47 + 16) \% 7 = 0$ collision!
- $(47 + 25) \% 7 = 2$ collision!

0	48
1	-
2	5
3	55
4	-
5	40
6	76

- **Will we ever get a 1 or 4?!?**



Quadratic Probing: Another Add Example

❖ Example: add 76, 40, 48, 5, 55, 47

❖ Will we ever get a 1 or 4?!?

- add(47) will *always* fail here. Why?

- For all i , $(5 + i^2) \% 7$ is 0, 2, 5, or 6

- Proof uses induction and

- $(5 + i^2) \% 7 = (5 + (i - 7)^2) \% 7$

❖ In fact, for all c and k ,

- $(c + i^2) \% k = (c + (i - k)^2) \% k$

0

48

1

-

2

5

3

55

4

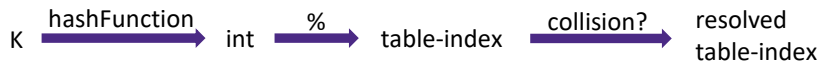
-

5

40

6

76



Quadratic Probing: Bad News / Good News

- ❖ Bad News:
 - After `TableSize` probes, we cycle through the same indices
- ❖ Good News:
 - If **TableSize is prime and $\lambda < \frac{1}{2}$** , then quadratic probing will find an empty slot in at most `TableSize/2` probes
 - So: If you keep $\lambda < \frac{1}{2}$ and `TableSize` is prime, no need to detect cycles
- ❖ Proof posted online after lecture
 - Textbook also has proof, but it's slightly less detailed

Quadratic Probing: Success Guarantee (1 of 2)

If `TableSize` is prime and $\lambda < \frac{1}{2}$, then quadratic probing will find an empty bucket in `TableSize/2` probes or fewer

- ❖ Intuition: if the table is less than half full, then probing `TableSize/2` distinct buckets must find an empty one
 - Therefore, prove the first `TableSize/2` probes are distinct

Any i^{th} and any j^{th} probe results in a distinct bucket

- ❖ Theorem: for all $0 \leq i, j \leq \text{TableSize}/2$, and $i \neq j$
 $(h(x) + i^2) \% \text{TableSize} \neq (h(x) + j^2) \% \text{TableSize}$

Quadratic Probing: Success Guarantee (2 of 2)

- ❖ Proof, by contradiction: suppose that for some $i \neq j$:

$$(h(x) + i^2) \% \text{TableSize} = (h(x) + j^2) \% \text{TableSize}$$

$$\Rightarrow i^2 \% \text{TableSize} = j^2 \% \text{TableSize}$$

$$\Rightarrow (i^2 - j^2) \% \text{TableSize} = 0$$

$$\Rightarrow [(i + j)(i - j)] \% \text{TableSize} = 0$$

$$\Rightarrow [(i + j)(i - j)] = k * \text{TableSize} \text{ for some } k \geq 1$$

or

$$[(i + j)(i - j)] = 0$$

CONTRADICTION!

- ❖ How can $i+j = 0$ or $i+j = k * \text{TableSize}$ when:
 $0 \leq i, j$ and $i \neq j$ and $i, j \leq \text{TableSize}/2$?
- ❖ How can $i-j = 0$ or $i-j = k * \text{TableSize}$ when
 $i \neq j$ and $i, j \leq \text{TableSize}/2$?

Quadratic Probing: Secondary Clustering

- ❖ Quadratic probing does not suffer from primary clustering!
 - We don't grow "big blobs" by adding to the end of a cluster

Working as designed!
- ❖ Quadratic probing does not resolve collisions between different keys that hash to *the same index*
 - These keys **have the same series of moves** looking for an empty spot
 - Called **secondary clustering** ☹️ *Unanticipated trade off!*
- ❖ Since the problem occurs when we have the different keys hashing to the same initial index, can we avoid secondary clustering with *a probe function that also incorporates the key?*
 - Known as **double hashing**

Lecture Outline

❖ Hash Tables

- Collision Resolution: Open Addressing
 - Quadratic Probing
 - **Double Hashing**
- Collision Avoidance: Rehashing
- Java-specific Hash Table Concerns
- Conclusion

❖ Comparison Sorting

- Simple Algorithms: InsertionSort and SelectionSort
- Fancier Algorithms: HeapSort

Double Hashing

❖ Double hashing:

- i^{th} probe: $(h(\text{key}) + i * g(\text{key})) \% \text{TableSize}$

- Probe sequence becomes:

- 0th probe: $h(\text{key}) \% \text{TableSize}$

- 1st probe: $(h(\text{key}) + g(\text{key})) \% \text{TableSize}$

- 2nd probe: $(h(\text{key}) + 2 * g(\text{key})) \% \text{TableSize}$

- ...

❖ Idea:

- $g(\text{key})$ lets us “go different places from initial collisions”

- It is very unlikely that for some key, $h(\text{key}) == g(\text{key})$

- (assuming good hash functions h and g)

- $i * g(\text{key})$ lets us “leave the neighborhood”

❖ Detail: Ensure $g(\text{key})$ can't generate 0

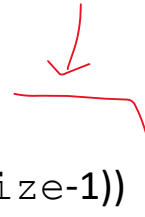
Double Hashing: Add Example

❖ Example: add 13, 28, 33, 147, 43

- Let $\text{hashFunction}(x) = x \% \text{TableSize}$
- Let $g(x) = 1 + ((x / \text{TableSize}) \% (\text{TableSize} - 1))$
- Let $\text{TableSize} = 10$

← Bad (non-prime) table size

Still bad hash fns



0	-
1	-
2	-
3	13
4	33
5	-
6	147
7	-
8	28
9	-



Double Hashing: Add Example

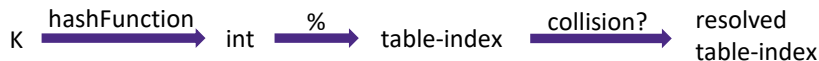
❖ Example: add ~~13, 28, 33, 147~~, 43

- Let $\text{hashFunction}(x) = x \% \text{TableSize}$
- Let $g(x) = 1 + ((x / \text{TableSize}) \% (\text{TableSize} - 1))$
- Let $\text{TableSize} = 10$

- $h(43) = 3$
- $g(43) = 1 + (4 \% 9) = 5$
- $3 + 0 * 5 = 3$ collision!
- $3 + 1 * 5 = 8$ collision!
- $3 + 2 * 5 = 13$ collision

▪ **Will we ever get anything else!?!?**

0	-
1	-
2	-
3	13
4	33
5	-
6	147
7	-
8	28
9	-

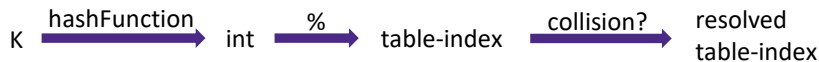


Double Hashing: Add Example

- ❖ Example: add ~~13, 28, 33, 147~~, 43
 - Let $\text{hashFunction}(x) = x \% \text{TableSize}$
 - Let $g(x) = 1 + ((x / \text{TableSize}) \% (\text{TableSize} - 1))$
 - Let $\text{TableSize} = 10$

 - Will we ever get anything else?!?
 - $\text{add}(43)$ will always fail here. Why?

0	-
1	-
2	-
3	13
4	33
5	-
6	147
7	-
8	28
9	-



Double Hashing: Considerations (1 of 2)

- ❖ Our example implies the possibility of infinite probe sequences 😞
 - But we can avoid infinite probes if our functions are:
 - $h(\text{key}) = \text{hash1}(\text{key}) \% p$
 - $g(\text{key}) = q - (\text{hash2}(\text{key}) \% q)$
 - And p and q are primes, with $2 < q < p$

Double Hashing: Considerations (2 of 2)

❖ Double hashing:

- i^{th} probe: $(h(\text{key}) + i * g(\text{key})) \% \text{TableSize}$

❖ Assume $g(\text{key})$ divides TableSize

- That is, there exists some integer x such that $x * g(\text{key}) = \text{TableSize}$
- Therefore: after x probes, we'll "loop through" the same indices as before

▪ Example:

- $\text{TableSize} = 50$
- $g(\text{key}) = 25$
- Probe sequence:
 - $i=0: h(\text{key})$
 - $i=1: h(\text{key}) + 25$
 - $i=2: h(\text{key}) + 50 = h(\text{key})$
 - $i=3: h(\text{key}) + 75 = h(\text{key}) + 25$
 - ...

❖ Bottom line: don't let $g(\text{key})$ divide TableSize

- That is, choose a prime TableSize when using double hashing

Double Hashing: Performance

- ❖ Assume $g()$ distributes its keys uniformly over its range
 - That is: probability of $g(\text{key1}) \% p == g(\text{key2}) \% p$ is $1/p$
- ❖ We won't prove the following:
 - Average # of probes (in the limit as TableSize $\rightarrow \infty$), **unsuccessful** find:

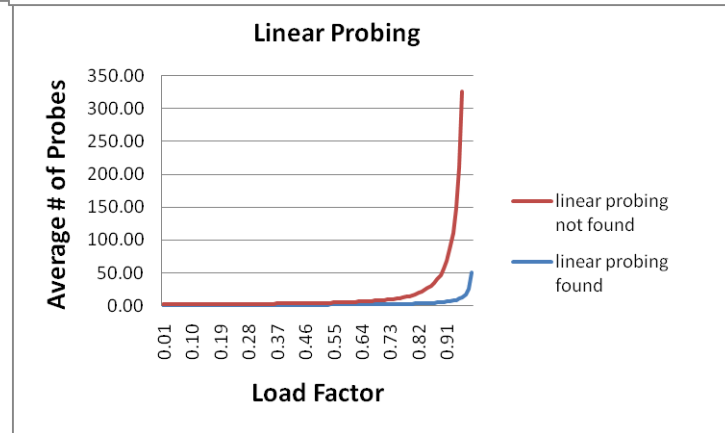
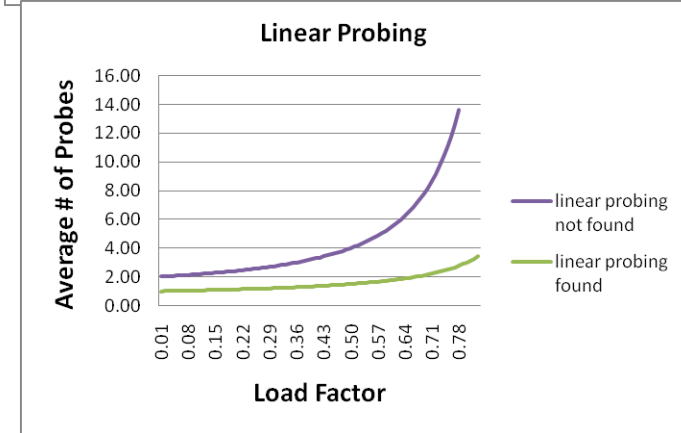
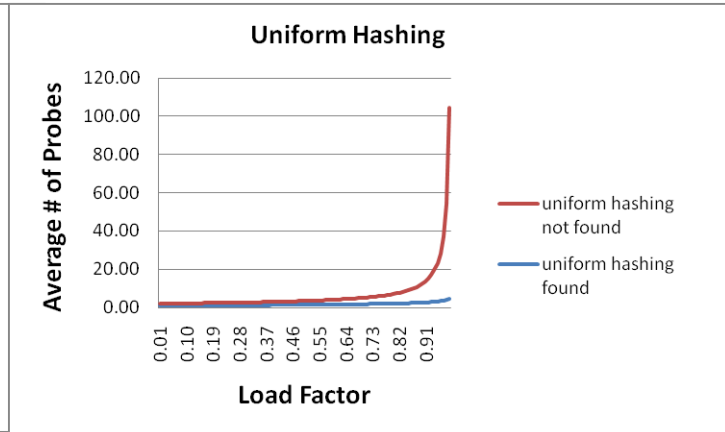
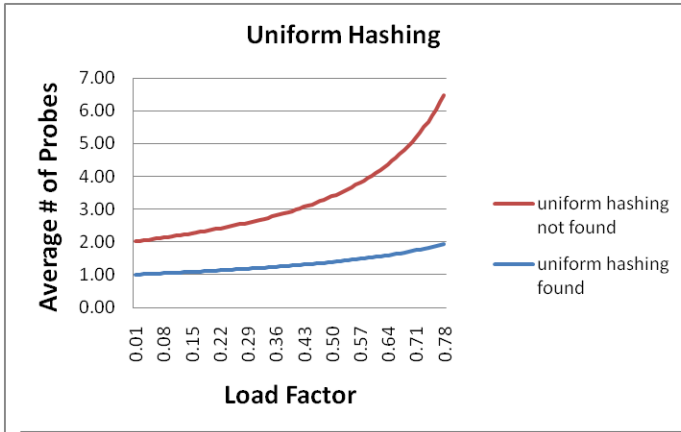
$$\frac{1}{1-\lambda}$$

- Average # of probes (in the limit as TableSize $\rightarrow \infty$), **successful** find:

$$\frac{1}{\lambda} \log_e \left(\frac{1}{1-\lambda} \right)$$

- ❖ Bottom line:
 - Performance of unsuccessful finds degrades with λ (but not as quickly as linear probing degrades)
 - Performance of successful finds degrades not nearly as quickly

Double Hashing vs Linear Probing Performance



Lecture Outline

❖ Hash Tables

- Collision Resolution: Open Addressing
 - Quadratic Probing
 - Double Hashing
- **Collision Avoidance: Rehashing**
- Java-specific Hash Table Concerns
- Conclusion

❖ Comparison Sorting

- Simple Algorithms: InsertionSort and SelectionSort
- Fancier Algorithms: HeapSort

Separate Chaining vs Open Addressing

❖ Separate Chaining

- **find, add, remove** proportional to λ if using unsorted LL
- If using another data structure for buckets (e.g. AVL tree), runtime is proportional to runtime for that structure

❖ Open addressing: has clustering issues as table fills ($\lambda > 1/2$)

- Why use it:
 - Some runtime for allocating nodes; open addressing could be faster?
 - Easier data representation?

Rehashing (1 of 3)

- ❖ As with array-based stacks/queues/lists, if table gets too full, create a bigger table and “copy” everything over
- ❖ With separate chaining, we decide what “too full” means
 - Keep load factor reasonable (e.g., < 1)?
 - Consider average or max size of non-empty chains?
- ❖ For open addressing, half-full is a good rule of thumb

Rehashing (2 of 3)

- ❖ Can't actually copy to the same indices in the new table
 - We'd calculated the index based on **TableSize**
- ❖ For each key/value in old table, must add into new table
 - Iterate over old table: $O(n)$
 - n calls to the hash function: $n \cdot O(1) = O(n)$
- ❖ Can we avoid all those hash function calls?
 - Space/time tradeoff: Could store $h(\text{key})$ with each item
 - Iterating over the table is still $O(n)$; saving $h(\text{key})$ only helps by a constant factor

Rehashing (3 of 3)

- ❖ New table size
 - Twice-as-big is a good idea, except ... ummm ... that won't be prime!
 - So go *about* twice-as-big
 - Hard-coded list of primes (you probably won't grow more than 20-30 times)
 - Calculate primes after that

Lecture Outline

❖ Hash Tables

- Collision Resolution: Open Addressing
 - Quadratic Probing
 - Double Hashing
- Collision Avoidance: Rehashing
- **Java-specific Hash Table Concerns**
- Conclusion

❖ Comparison Sorting

- Simple Algorithms: InsertionSort and SelectionSort
- Fancier Algorithms: HeapSort

Hashing and Equality Testing

- ❖ Our examples use an `int` key, which overlooks a critical detail:
 - We hash \mathbb{K} to get a table index
 - While chaining or probing, we need to test whether the current \mathbb{K}' is equal to the \mathbb{K} we're looking for
- ❖ So a Java hash table needs a hash *and* an equality function
 - Fortunately, in Java every object defines an **`equals`** and a **`hashCode`** method

```
class Object {
    boolean equals(Object o) {...}
    int hashCode() {...}
    ...
}
```

Overriding equals()? Override hashCode() too

- ❖ The Java library (and your project's hash table) make a very important assumption that *all* clients must satisfy:
 - Object-oriented way of saying it:
If `a.equals(b)`, then `a.hashCode() == b.hashCode()`
 - Functor way of saying it:
If `c.compare(a,b) == 0`, then
`h.hashCode(a) == h.hashCode(b)`
- ❖ In other words, if you ever override equals:
 - You must also override hashCode() in a consistent way
 - See [Core Java](#) book, Ch. 5, for other "gotchas" with equals()

compareTo() rules

- ❖ Java also makes assumptions about `compareTo()` that affect:
 - All our dictionaries
 - Sorting (next major topic)

- ❖ Comparison must impose a consistent, total ordering:
 - For all **a**, **b**, and **c**,
 - If `a.compareTo(b) < 0`, then `b.compareTo(a) > 0`
 - If `a.compareTo(b) == 0`, then `b.compareTo(a) == 0`
 - If `a.compareTo(b) < 0` and `b.compareTo(c) < 0`, then `a.compareTo(c) < 0`

A Generally-Good hashCode()

```
int result = 17; // start at a prime

foreach field f
    int fieldHashCode =
        boolean: (f ? 1: 0)
        byte, char, short, int: (int) f
        long: (int) (f ^ (f >>> 32))
        float: Float.floatToIntBits(f)
        double: Double.doubleToLongBits(f),
            then above conversion to int
        Object: object.hashCode()
    result = 31 * result + fieldHashCode;

return result;
```



Lecture Outline

❖ Hash Tables

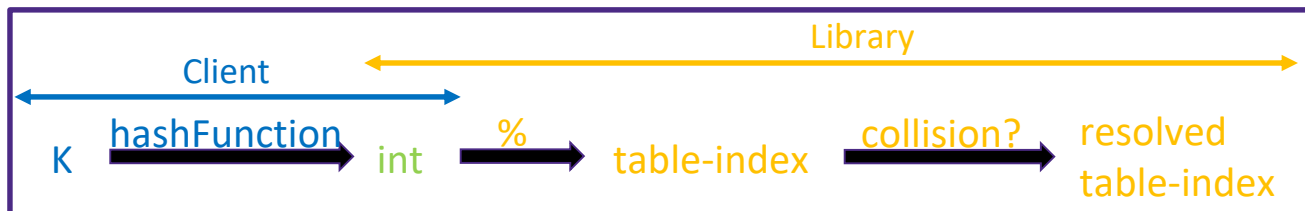
- Collision Resolution: Open Addressing
 - Quadratic Probing
 - Double Hashing
- Collision Avoidance: Rehashing
- Java-specific Hash Table Concerns
- **Conclusion**

❖ Comparison Sorting

- Simple Algorithms: InsertionSort and SelectionSort
- Fancier Algorithms: HeapSort

Who Hashes What?

- ❖ When used as a library, hash tables generally have two roles: client vs library



- ❖ We learned both, but you'll spend more time as clients
 - Both roles must contribute to minimizing collisions
 - Client should aim for different ints for expected keys
 - Avoid "wasting" any part of K or the `int`'s bits
 - Library should aim for putting "similar" ints in different indices
 - Conversion to index is almost always "mod table-size"
 - Using prime numbers for table-size is common

Summary: Hash Tables vs. Balanced Trees

- ❖ In terms of a Dictionary ADT for just **add**, **find**, **remove**, hash tables and balanced trees are just different data structures
 - Hash tables $O(1)$ on average (assuming few collisions)
 - Balanced trees $O(\log n)$ worst-case
- ❖ Constant-time is better, right?
 - Yes, but you need “hashing to behave” (must avoid collisions)
 - Yes, but what if we want to **findMin**, **findMax**, **predecessor**, and **successor**, **printSorted**?
 - Hash tables are not designed to efficiently implement these operations
 - Your textbook considers hash tables to be a different ADT
 - Not so important to argue over the definitions

Summary: Hash Tables

- ❖ The hash table is one of the most important data structures
 - Useful in many, many real-world applications
 - Popular topic for job interview questions
- ❖ **Crucial** to use a good hash function
 - Hash tables rely on good distribution
 - Not overly expensive to calculate (bit shifts good!)
- ❖ Important to keep hash table at a good size
 - Prime table size
 - “Preferred λ ” depends on type of table
- ❖ What we skipped:
 - Perfect hashing, universal hash functions, hopscotch hashing, cuckoo hashing

Lecture Outline

❖ Hash Tables

- Collision Resolution: Open Addressing
 - Quadratic Probing
 - Double Hashing
- Collision Avoidance: Rehashing
- Java-specific Hash Table Concerns
- Conclusion

❖ Comparison Sorting

- Simple Algorithms: InsertionSort and SelectionSort
- Fancier Algorithms: HeapSort

Introduction to Sorting (1 of 2)

- ❖ Stacks, queues, priority queues, and dictionaries/sets all provide one element at a time
- ❖ But often we want “all the items” in some order
 - Alphabetical list of people
 - Population list of countries
 - Search engine results by relevance
- ❖ Different sorting algorithms have different asymptotic and constant-factor trade-offs
 - Knowing one way to sort just isn't enough; no single “best sort”
 - **Sorting is an excellent case-study in making trade-offs!**

Introduction to Sorting (2 of 2)

- ❖ *Preprocessing* (e.g. sorting) data to make subsequent operations faster is a general technique in computing!
 - Example: Sort the items so that you can:
 - Find the k^{th} largest in constant time for any k
 - Perform binary search to find an item in logarithmic time
 - Whether preprocessing is beneficial depends on
 - How often the items will change
 - How many items there are

- ❖ Preprocessing's benefits depend on
 - How often the items will change and how many items there are
 - **Sorting is an excellent case-study in making trade-offs!**

Comparison Sorting: Definition

- ❖ Problem: We have n comparable items in an array, and we want to rearrange them to be in increasing order

- ❖ Input:
 - An array A of (key, value) pairs
 - A comparison function (consistent and total)
 - Given keys a & b , what is their relative ordering? $<$, $=$, $>$?
 - Ex: keys that implement Comparable or have a Comparator

- ❖ Output/Side-Effect:
 - Reorganize the elements of A such that for any index i and j ,
if $i < j$ then $A[i] \leq A[j]$
 - [Usually unspoken] A must have all the same items it started with
 - Could also sort in reverse order, of course

Comparison Sort: Variations (1 of 2)

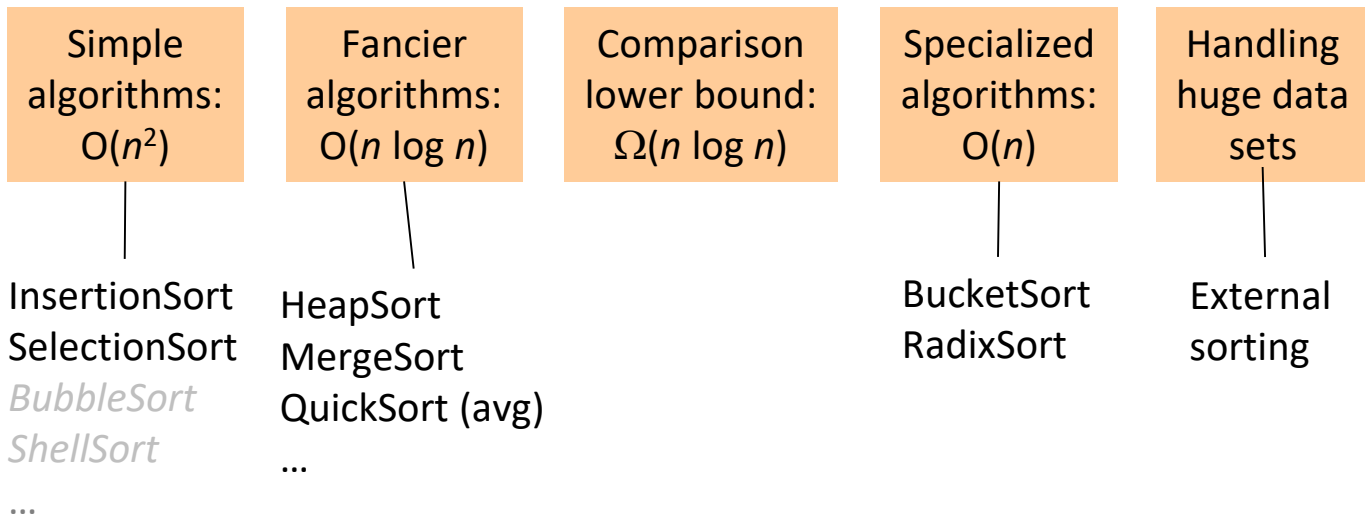
1. Maybe elements are in a linked list
 - Could convert to array and back in linear time, but some algorithms can still “work” on linked lists
2. Maybe if there are ties we should preserve the original ordering
 - Sorts that do this naturally are called **stable sorts**
3. Maybe we must not use more than $O(1)$ “auxiliary space”
 - These are called **in-place sorts**
 - Not allowed to allocate memory proportional to input (i.e., $O(n)$), but can allocate $O(1)$ # of variables
 - Work is done by swapping around in the array

Comparison Sort: Variations (2 of 2)

4. Maybe we can do more with elements than just compare
 - Comparison sorts assume a binary 'compare' operator
 - In special cases we can sometimes get faster algorithms

5. Maybe we have too many items to fit in memory
 - Use an **external sorting** algorithm

Sorting: The Big Picture



Lecture Outline

❖ Hash Tables

- Collision Resolution: Open Addressing
 - Quadratic Probing
 - Double Hashing
- Collision Avoidance: Rehashing
- Java-specific Hash Table Concerns
- Conclusion

❖ Comparison Sorting

- **Simple Algorithms: InsertionSort and SelectionSort**
- Fancier Algorithms: HeapSort

InsertionSort

- ❖ Idea: At step k , insert the k^{th} element in the correct position
 - Sort first two elements
 - Now insert 3rd element in order
 - ...

- ❖ Loop invariant (“when loop index is i ”):
 - First i elements are sorted

- ❖ Time:

Best-case: _____ Worst-case: _____ “Average” case: _____

- ❖ Characteristics:

Stable: _____ In-place: _____

Demo:

https://docs.google.com/presentation/d/10b9aRqpGJu8pUk8OpfqUIEEem8ou-zmmC7b_BE5wgNg0/present

SelectionSort

- ❖ Idea: At step k , select the smallest elt and put it at k^{th} position
 - Find smallest element, put it 1st
 - Find next smallest element, put it 2nd
 - ...
- ❖ Loop invariant (“when loop index is i ”):
 - First i elements are the i smallest elements in sorted order
- ❖ Time:
Best-case: _____ Worst-case: _____ “Average” case: _____
- ❖ Characteristics:
Stable: _____ In-place: _____

Demo:

<https://docs.google.com/presentation/d/1p6g3r9BpwTARjUyIA0V0yspP2temzHNJEJjCG41I4r0/edit>

InsertionSort vs. SelectionSort

- ❖ Different algorithms, same problem
- ❖ Same worst-case and “average”-case asymptotic complexity
 - InsertionSort has better best-case complexity; preferable when input is “mostly sorted”
- ❖ Other algorithms are more efficient *for non-small arrays that are not already almost sorted*
 - InsertionSort may do well on small arrays (empirically: $N < \sim 15$)

Lecture Outline

❖ Hash Tables

- Collision Resolution: Open Addressing
 - Quadratic Probing
 - Double Hashing
- Collision Avoidance: Rehashing
- Java-specific Hash Table Concerns
- Conclusion

❖ Comparison Sorting

- Simple Algorithms: InsertionSort and SelectionSort
- **Fancier Algorithms: HeapSort**

Naïve HeapSort

- ❖ Idea: Put everything into a **MIN** heap. At step **k**, remove the **kth**-min element
 - insert each `arr[i]` –OR– better yet, use `buildHeap`
 - ```
for(i=0; i < arr.length; i++)
 arr[i] = deleteMin();
```
- ❖ Loop invariant (“when loop index is **i**”):
  - First **i** elements are the **i** smallest elements in sorted order
- ❖ Time:  
Best-case: \_\_\_\_\_ Worst-case: \_\_\_\_\_ “Average” case: \_\_\_\_\_

Demo: <https://goo.gl/EZWwSJ>

# In-place HeapSort

- ❖ Idea: Put everything into a **MAX** heap. At step **k**, remove the **k<sup>th</sup>**-max element
  - insert each `arr[i]` –OR– better yet, use `buildHeap`
  - `for(i=0; i < arr.length; i++)`  
`arr[arr.length - i] = deleteMax();`
- ❖ Loop invariant (“when loop index is **i**”): same as naïve version
- ❖ Time:  
Best-case: \_\_\_\_\_ Worst-case: \_\_\_\_\_ “Average” case: \_\_\_\_\_
- ❖ Characteristics:  
Stable: \_\_\_\_\_ In-place: \_\_\_\_\_

Demo:

<https://docs.google.com/presentation/d/1SzcQC48OB9agStD0dFRgccU-tyjD6m3esrSC-GLxmNc/present>

## Aside: “AVL sort” and “data structure sorts”

- ❖ We can also use a balanced tree to:
  - **add** each element: total time  $O(n \log n)$
  - Do an in-order traversal  $O(n)$
- ❖ Cannot be made in-place and constants worse than HeapSort
  - Both are  $O(n \log n)$  in worst, best, and average case; neither parallelizes well
- ❖ Don't even think about trying to sort with a hash table ...