CSC 427: Data Structures and Algorithm Analysis

Fall 2011

Space vs. time

- space/time tradeoffs
- hashing
  - hash table, hash function
  - linear probing vs. chaining
  - HashSet & HashMap implementations

1

Space vs. time

for many applications, it is possible to trade memory for speed

- i.e., additional storage can allow for a more efficient algorithm

example: heap sort

- can perform an efficient sort of a list by building a heap, then extracting in order

example: Boggle game

- build separate lists of all words found on the board
- only search board once, then efficient list accesses/updates

example: string matching

- need to find pattern P in larger string S
- recall brute force solution: O(|S|*|P|)
- Boyer-Moore algorithm builds 2 pattern-specific tables & uses them to reduce the search cost to at most 3*|S| (but often < |S|)

2

Hash tables

recall: `TreeSet` & `TreeMap` use an underlying binary search tree (actually, a red-black tree) to store values

- as a result, add/put, contains/get, and remove are O(log N) operations
- iteration over the Set/Map can be done in O(N)

the other implementations of the `Set` & `Map` interfaces, `HashSet` & `HashMap`, use a "magic" data structure to provide O(1) operations*

**legal disclaimer:* performance can degrade to O(N) under bad/unlikely conditions
however, careful setup and maintenance can ensure O(1) in practice

the underlying data structure is known as a *Hash Table*

- "magic" performance is bought with lots of additional memory

3

Hash tables

a hash table is a data structure that supports *constant time* insertion, deletion, and search *on average*

- degenerative performance is possible, but unlikely
- it may waste considerable storage
- iteration order is not defined (and may even change over time)

idea: data items are stored in a table, based on a key

- the key is mapped to an index in the table, where the data is stored/accessed

simple example: want to know which letters appear in a file

- have an array of 26 Booleans, map each letter to an index
- when read a letter, store `true` in its corresponding index

key	index	value
"A"	→ 0	true
"B"	→ 1	false
"C"	→ 2	true
		. . .
"Z"	→ 25	false

4

Mapping examples

extension: storing words

- must map entire words to indices, e.g.,

"A" → 0 "AA" → 26 "BA" → 52 . . .
"B" → 1 "AB" → 27 "BB" → 53 . . .
. . . .
"Z" → 25 "AZ" → 51 "BZ" → 77. . .

- PROBLEM?

mapping each potential item to a unique index is generally not practical

of 1 letter words = 26
# of 2 letter words = 26^2 = 676
# of 3 letter words = 26^3 = 17,576
. . .

- even if you limit words to at most 8 characters, need a table of size 217,180,147,158
- for any given file, the table will be mostly empty!

5

Table size < data range

since the actual number of items stored is generally MUCH smaller than the number of potential values/keys:

- can have a smaller, more manageable table

e.g., table size = 1000
possible mapping: add ASCII values of letters, mod by 1000

"AB" → 65 + 66 = 131

"BANANA" → 66 + 65 + 78 + 65 + 78 + 65 = 417

"BANANABANANABANANA" → 417 + 417 + 417 = 1251 % 1000 = 251

- POTENTIAL PROBLEMS?

6

Collisions

the mapping from a key to an index is called a *hash function*

- the hash function can be written independent of the table size
- if it maps to an index > table size, simply wrap-around (i.e., index % tableSize)

since `|range(hash function)| < |domain(hash function)|`, can have multiple items map to the same index (i.e., a *collision*)

"ACT" → 67 + 65 + 84 = 216 "CAT" → 67 + 65 + 84 = 216

the hash function should do everything possible to avoid collisions

- in particular, it should distribute the keys evenly
- e.g., "sum of ASCII codes" hash function
  OK if table size is 1000
  BAD if table size is 10,000
  most words are <= 8 letters, so max sum of ASCII codes = 1,016
  so most entries are mapped to first 1/10th of table

7

Better hash function

a good hash function should

- produce an even spread, regardless of table size
- take order of letters into account (to handle anagrams)
- the hash function used by `java.util.String` multiplies the ASCII code for each character by a power of 31

$$hashCode() = char_0*31^{(len-1)} + char_1*31^{(len-2)} + char_2*31^{(len-3)} + \ldots + char_{(len-1)}$$

where $len = this.length()$, $char_i = this.charAt(i)$:

```
/**
 * Hash code for java.util.String class
 *   @return an int used as the hash index for this string
 */
private int hashCode() {
    int hashIndex = 0;

    for (int i = 0; i < this.length(); i++) {
        hashIndex = (hashIndex*31 + this.charAt(i));
    }
    return hashIndex;
}
```

8

Collision handling

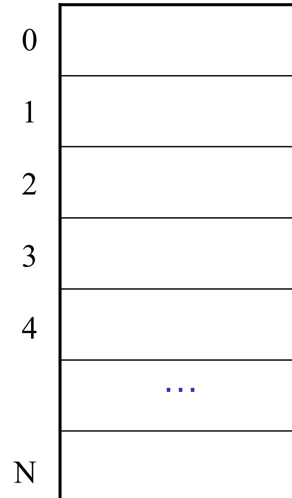
even if the hash function distributes keys evenly, collisions are inevitable

early approach: linear probing

- as you store values,
  map to an index using the hash function
  if the spot is empty, insert the value
  otherwise, store in next available spot
  (wrap if necessary)

- example: suppose hash(word) = word.length()
  add "CAT", "A", "DOG", …

- when search for an entry, can't look just at the hash index – have to continue sequentially until find it or an open spot
- when you delete an entry, must mark its spot in case the gap messes up a later search



9

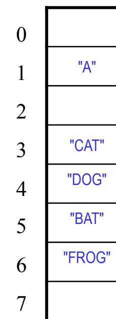
Linear probing analysis

advantage: the table size is fixed

disadvantage: primary clustering

- each collision causes an entry to steal a spot
- because it steals sequentially, a cluster forms

here, 5 checks to find that "ANT" is not stored
similarly, 4-6 letter words are affected



THEOREM: Define load factor λ to be the fraction of the table that is full. Assuming a reasonably large table, the average number of locations examined per insertion is roughly $(1 + 1/(1-\lambda)^2)/2$

empty table	$(1 + 1/(1 - 0)^2)/2 = 1$
half full	$(1 + 1/(1 – .5)^2)/2 = 2.5$
3/4 full	$(1 + 1/(1 - .75)^2)/2 = 8.5$
9/10 full	$(1 + 1/(1 - .9)^2)/2 = 50.5$

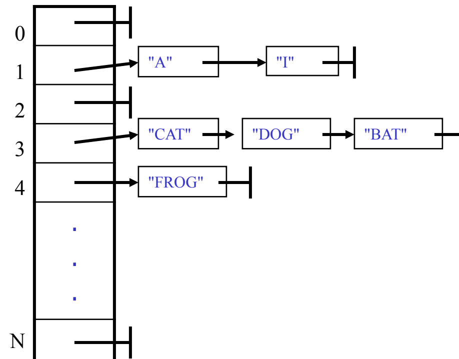
for linear probing to be practical, must expand the table whenever it gets close to full & then rehash entries to new spots

10

Chaining

to avoid clustering, modern implementations use *chaining*

- each entry in the hash table is a linked list
- when you insert, hash to the correct index then add to list
- when you search, hash to the correct index then search the list sequentially
- when you delete, hash to the correct index & remove from the list



advantages:

- insertion is always O(1)
- search requires traversing all entries with same hash function – no clustering effect since never steal another hash value's spot
- deletion is immediate – don't need to mark the spot

disadvantage:

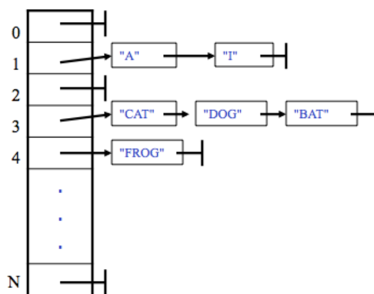
- requires even more memory, size is O(tableSize + numEntries)

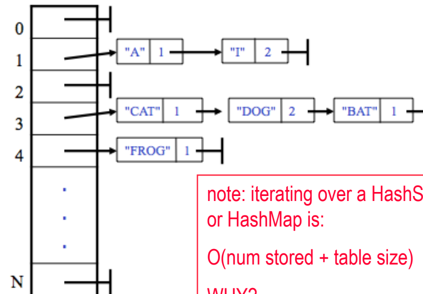
11

HashSet & HashMap

`java.util.HashSet` and `java.util.HashMap` use chaining

- e.g., `HashSet<String>` `HashMap<String, Integer>`





note: iterating over a HashSet or HashMap is:

O(num stored + table size)

WHY?

- defaults: table size = 16, max capacity before rehash = 75%
  can override these defaults in the HashSet/HashMap constructor call
- a default hash function is defined for every `Object` (based on its address)
- a class can define its own hash function by overriding `hashCode`
  must ensure that `obj1.equals(obj2)` → `obj1.hashCode() == obj2.hashCode()`

12