Unit 9, Part 4

Hash Tables

Computer Science S-111
Harvard University

David G. Sullivan, Ph.D.

Data Dictionary Revisited

- We've considered several data structures that allow us to store and search for data items using their key fields:

data structure	***searching for an item***	***inserting an item***
a list implemented using an array	$O(\log n)$ using binary search	$O(n)$
a list implemented using a linked list	$O(n)$ using linear search	$O(n)$
binary search tree		
balanced search trees (2-3 tree, B-tree, others)		

- We'll now look at hash tables, which can do better than $O(\log n)$.

Ideal Case: Searching = Indexing

- We would achieve optimal efficiency if we could treat the key as an index into an array.
- Example: storing data about members of a sports team
  - key = jersey number (some value from 0-99).
  - class for an individual player's record:

```
public class Player {
    private int jerseyNum;
    private String firstName;
    …
}
```

  - store the player records in an array:

```
Player[] teamRecords = new Player[100];
```

- In such cases, search and insertion are *O*(1):

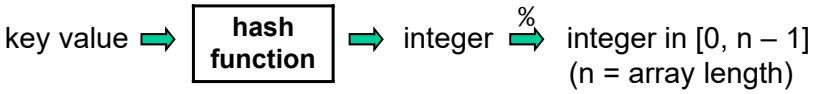
```
public Player search(int jerseyNum) {
    return teamRecords[jerseyNum];
}
```

Hashing: Turning Keys into Array Indices

- In most real-world problems, indexing is not as simple as the sports-team example. Why?
  -
  -
  -
- To handle these problems, we perform *hashing*:
  - use a *hash function* to convert the keys into array indices
    `"Sullivan"` → `18`
  - use techniques to handle cases in which multiple keys are assigned the same hash value
- The resulting data structure is known as a *hash table*.

Hash Functions

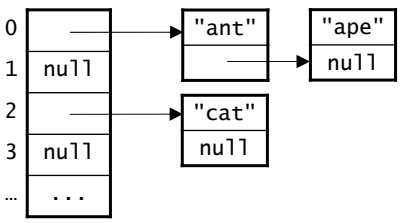
- A hash function defines a mapping from keys to integers.
- We then use the modulus operator to get a valid array index.



- Here's a very simple hash function for keys of lower-case letters:
  h(key) = ASCII value of first char – ASCII value of 'a'
  - examples:
    h("ant") = ASCII for 'a' – ASCII for 'a' = 0
    h("cat") = ASCII for 'c' – ASCII for 'a' = 2
- h(key) is known as the key's *hash code.*
- A *collision* occurs when items with different keys are assigned the same hash code.

Dealing with Collisions I: Separate Chaining

- Each position in the hash table serves as a *bucket* that can store multiple data items.
- Two options:
  1. each bucket is itself an array
     - need to preallocate, and a bucket may become full
  2. each bucket is a linked list
     - items with the same hash code are "chained" together
     - each "chain" can grow as needed



Dealing with Collisions II: Open Addressing

- When the position assigned by the hash function is occupied, find another open position.
- Example: "wasp" has a hash code of 22, but it ends up in position 23 because position 22 is occupied.
- We'll consider three ways of finding an open position – a process known as *probing*.
- We also perform probing when searching.
  - example: search for "wasp"
    - look in position 22
    - then look in position 23
  - need to figure out when to safely stop searching (more on this soon!)

0	"ant"
1	
2	"cat"
3	
4	"emu"
5	
6	
7	
…	...
22	"wolf"
23	**"wasp"**
24	"yak"
25	"zebra"

Linear Probing

- Probe sequence: h(key), h(key) + 1, h(key) + 2, …, wrapping around as necessary.
- Examples:
  - "ape" (h = 0) would be placed in position 1, because position 0 is already full.
  - "bear" (h = 1): try 1, 1 + 1, 1 + 2 – open!
  - where would "zebu" end up?
- Advantage: if there is an open cell, linear probing will eventually find it.
- Disadvantage: get "clusters" of occupied cells that lead to longer subsequent probes.
  - probe length = the number of positions considered during a probe

0	"ant"
1	"ape"
2	"cat"
3	"bear"
4	"emu"
5	
6	
7	
…	...
22	"wolf"
23	"wasp"
24	"yak"
25	"zebra"

Quadratic Probing

- Probe sequence: h(key), h(key) + 1^2, h(key) + 2^2, h(key) + 3^2, …, wrapping around as necessary.
- Examples:
  - "ape" (h = 0): try 0, 0 + 1 – open!
  - "bear" (h = 1): try 1, 1 + 1, 1 + 4 – open!
  - "zebu"?
- Advantage: smaller clusters of occupied cells
- Disadvantage: may fail to find an existing open position. For example:

table size = 10
x = occupied

trying to insert a key with h(key) = 0

offsets of the probe sequence in italics

0	x		5	x	*25*
1	x	*1 81*	6	x	*16 36*
2			7		
3			8		
4	x	*4 64*	9	x	*9 49*

0	"ant"
1	"ape"
2	"cat"
3	
4	"emu"
5	"bear"
6	
7	
…	...
22	"wolf"
23	"wasp"
24	"yak"
25	"zebra"

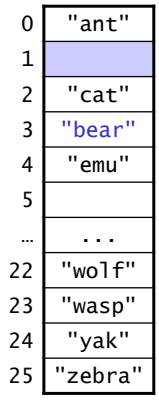
Double Hashing

- Use two hash functions:
  - h1 computes the hash code
  - h2 computes the increment for probing
  - probe sequence: h1, h1 + h2, h1 + 2*h2, …
- Examples:
  - h1 = our previous h
  - h2 = number of characters in the string
  - "ape" (h1 = 0, h2 = 3): try 0, 0 + 3 – open!
  - "bear" (h1 = 1, h2 = 4): try 1 – open!
  - "zebu"?
- Combines good features of linear and quadratic:
  - reduces clustering
  - will find an open position if there is one, provided the table size is a prime number

0	"ant"
1	"bear"
2	"cat"
3	"ape"
4	"emu"
5	
6	
7	
…	...
22	"wolf"
23	"wasp"
24	"yak"
25	"zebra"

Removing Items Under Open Addressing

- Problematic example (using linear probing):
  - insert "ape" (h = 0): try 0, 0 + 1 – open!
  - insert "bear" (h = 1): try 1, 1 + 1, 1 + 2 – open!
  - remove "ape"
  - search for "ape": try 0, 0 + 1 – conclude not in table
  - search for "bear": **try 1 – conclude not in table, but "bear" is further down in the table!**



- To fix this problem, distinguish between:
  - *removed positions* that previously held an item
  - *empty positions* that have never held an item
- During probing, we *don't* stop if we see a removed position.
  ex: search for "bear": try 1 (removed), 1 + 1, **1 + 2 – found!**
- We can insert items in either empty or removed positions.

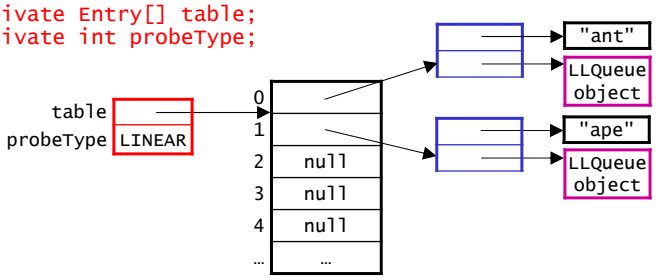
An Interface For Hash Tables

```
public interface HashTable {
    boolean insert(Object key, Object value);
    Queue<Object> search(Object key);
    Queue<Object> remove(Object key);
}
```

- `insert()` takes a key-value pair and returns:
  - `true` if the key-value pair can be added
  - `false` if it cannot be added (referred to as *overflow*)
- `search()` and `remove()` both take a key, and return a queue containing all of the values associated with that key.
  - example: an index for a book
    - key = word
    - values = the pages on which that word appears
  - return `null` if the key is not found

An Implementation Using Open Addressing

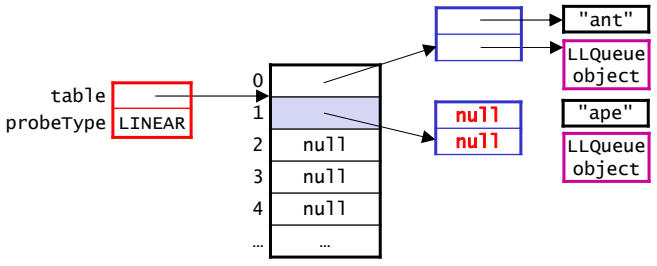
```
public class OpenHashTable implements HashTable {
    private class Entry {
        private Object key;
        private LLQueue<Object> values;
        …
    }
    …
    private Entry[] table;
    private int probeType;
}
```



- We use a private inner class for the entries in the hash table.
- We use an `LLQueue` for the values associated with a given key.

Empty vs. Removed

- When we remove a key and its values, we:
  - leave the `Entry` object in the table
  - set the `Entry` object's `key` and `values` fields to `null`
  - example: after `remove("ape")`:



- Note the difference:
  - a truly empty position has a value of `null` in the table (example: positions 2, 3 and 4 above)
  - a removed position refers to an `Entry` object whose `key` and `values` fields are `null` (example: position 1 above)

Probing Using Double Hashing

```
private int probe(Object key) {
    int i = h1(key);    // first hash function
    int h2 = h2(key);   // second hash function

    // keep probing until we get an empty position or match
    while (table[i] != null && !key.equals(table[i].key)) {
        i = (i + h2) % table.length;
    }

    return i;
}
```

- It is essential that we:
  - check for `table[i] != null` first. why?
  - call the `equals` method on `key`, not `table[i].key`. why?

Avoiding an Infinite Loop

- The while loop in our probe method could lead to an infinite loop.

```
while (table[i] != null && !key.equals(table[i].key)) {
    i = (i + h2) % table.length;
}
```

- When would this happen?
- We can stop probing after checking n positions (n = table size), because the probe sequence will just repeat after that point.
  - for quadratic probing:
    $(h1 + n^2) \% n = h1 \% n$
    $(h1 + (n+1)^2) \% n = (h1 + n^2 + 2n + 1) \% n = (h1 + 1) \% n$
  - for double hashing:
    $(h1 + n*h2) \% n = h1 \% n$
    $(h1 + (n+1)*h2) \% n = (h1 + n*h2 + h2) \% n = (h1 + h2) \% n$

Avoiding an Infinite Loop (cont.)

```
private int probe(Object key) {
    int i = h1(key);    // first hash function
    int h2 = h2(key);   // second hash function
    int numChecked = 1;

    // keep probing until we get an empty position or a match
    while (table[i] != null && !key.equals(table[i].key)) {
        if (numChecked == table.length) {
            return -1;
        }
        i = (i + h2) % table.length;
        numChecked++;
    }

    return i;
}
```

Search and Removal

```
public LLQueue<Object> search(Object key) {
    // throw an exception if key == null
    int i = probe(key);
    if (i == -1 || table[i] == null) {
        return null;
    } else {
        return table[i].values;
    }
}

public LLQueue<Object> remove(Object key) {
    // throw an exception if key == null
    int i = probe(key);
    if (i == -1 || table[i] == null) {
        return null;
    }

    LLQueue<Object> removedVals = table[i].values;
    table[i].key = null;
    table[i].values = null;
    return removedVals;
}
```

Insertion

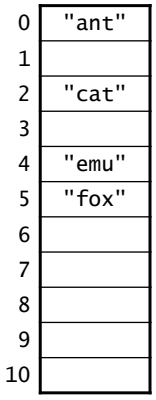
- We begin by probing for the key.
- Several cases:
  1. the key is already in the table (we're inserting a duplicate)
     → add the value to the values in the key's `Entry`
  2. the key is not in the table: three subcases:
     a. encountered 1 or more removed positions while probing
        → put the (key, value) pair in the *first* removed position seen during probing. why?

     b. no removed position; reached an empty position
        → put the (key, value) pair in the empty position

     c. no removed position or empty position
        → overflow; return `false`

Tracing Through Some Examples

- Start with the hash table at right with:
  - double hashing
  - our earlier hash functions h1 and h2
- Perform the following operations:
  - insert "bear" (h1 = 1, h2 = 4):
  - insert "bison" (h1 = 1, h2 = 5):
  - insert "cow" (h1 = 2, h2 = 3):
  - delete "emu" (h1 = 4, h2 = 3):
  - search "eel" (h1 = 4, h2 = 3):
  - insert "bee" (h1 = ___, h2 = ____):



Dealing with Overflow

- Overflow = can't find a position for an item
- When does it occur?
  - linear probing:
  - quadratic probing:
    -
    -
  - double hashing:
    - if the table size is a prime number: same as linear
    - if the table size is not a prime number: same as quadratic
- To avoid overflow (and reduce search times), grow the hash table when the % of occupied positions gets too big.
  - problem: we need to rehash **all** of the existing items. why?

Implementing the Hash Function

- Characteristics of a good hash function:
  1) efficient to compute
  2) uses the entire key
     - changing any char/digit/etc. should change the hash code
  3) distributes the keys more or less uniformly across the table
  4) must be a function!
     - a key must always get the same hash code
- In Java, every object has a `hashCode()` method.
  - the version inherited from `Object` returns a value based on an object's memory location
  - classes can override this version with their own

Hash Functions for Strings: version 1

- h_a = the sum of the characters' ASCII values
  - example: h_a("eat") = 101 + 97 + 116 = 314
- All permutations of a given set of characters get the same code.
  - example: h_a("tea") = h_a("eat")
  - could be useful in a Scrabble game
    - allow you to look up all words that can be formed from a given set of characters
- The range of possible hash codes is very limited.
  - example: hashing keys composed of 1-5 lower-case char's (padded with spaces)
  - 26*27*27*27*27 = over 13 million possible keys
  - smallest code = h_a("a ") = 97 + 4*32 = 225
    largest code = h_a("zzzzz") = 5*122 = 610
    } 610 – 225 = 385 codes

Hash Functions for Strings: version 2

- Compute a *weighted* sum of the ASCII values:

$$h_b = a_0 b^{n-1} + a_1 b^{n-2} + \ldots + a_{n-2} b + a_{n-1}$$

where a_i = ASCII value of the ith character
b = a constant
n = the number of characters

- Multiplying by powers of b allows the *positions* of the characters to affect the hash code.
  - different permutations get different codes
- We may get arithmetic overflow, and thus the code may be negative. We adjust it when this happens.
- Java uses this hash function with b = 31 in the `hashCode()` method of the `String` class.

Hash Table Efficiency

- In the best case, search and insertion are *O*(1).
- In the worst case, search and insertion are linear.
  - open addressing: *O*(m), where m = the size of the hash table
  - separate chaining: *O*(n), where n = the number of keys
- With good choices of hash function and table size, complexity is generally better than *O*(log n) and approaches *O*(1).
- *load factor* = # keys in table / size of the table.
  To prevent performance degradation:
  - open addressing: try to keep the load factor < 1/2
  - separate chaining: try to keep the load factor < 1
- Time-space tradeoff: bigger tables have better performance, but they use up more memory.

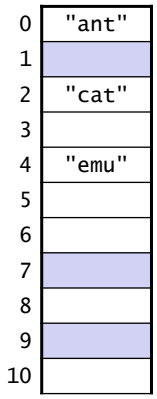
Hash Table Limitations

- It can be hard to come up with a good hash function for a particular data set.
- The items are not ordered by key. As a result, we can't easily:
  - print the contents in sorted order
  - perform a range search (find all values between v1 and v2)
  - perform a rank search – get the kth largest item

  We *can* do all of these things with a search tree.

Extra Practice

- Start with the hash table at right with:
  - double hashing
  - h1(key) = ASCII of first letter – ASCII of 'a'
  - h2(key) = key.length()
  - shaded cells are removed cells
- **What is the probe sequence for "baboon"?**
  (the sequence of positions seen during probing)

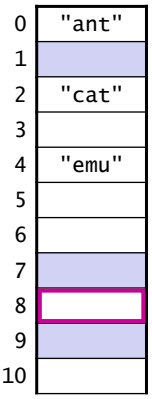


A. 1, 2, 5
B. 1, 6
C. 1, 7, 2
D. 1, 7, 3
E. 1, 7, 2, 8

Extra Practice

- Start with the hash table at right with:
  - double hashing
  - h1(key) = ASCII of first letter – ASCII of 'a'
  - h2(key) = key.length()
  - shaded cells are removed cells
- **What is the probe sequence for "baboon"?**
  (h1 = 1, h2 = 6) try: 1 % 11 = **1**
  (1 + 6) % 11 = **7**
  (1 + 2*6) % 11 = **2**
  (1 + 3*6) % 11 = **8**
  empty cell, so stop probing

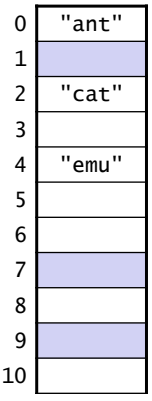
A. 1, 2, 5
B. 1, 6
C. 1, 7, 2
D. 1, 7, 3
E. **1, 7, 2, 8**



Extra Practice

- Start with the hash table at right with:
  - double hashing
  - h1(key) = ASCII of first letter – ASCII of 'a'
  - h2(key) = key.length()
  - shaded cells are removed cells
- What is the probe sequence for "baboon"?
  (h1 = 1, h2 = 6) try: 1 % 11 = 1
  (1 + 6) % 11 = 7
  (1 + 2*6) % 11 = 2
  (1 + 3*6) % 11 = 8
- **If we insert "baboon", in what position will it go?**

A. 1 B. 7 C. 2 D. 8



Extra Practice

- Start with the hash table at right with:
  - double hashing
  - h1(key) = ASCII of first letter – ASCII of 'a'
  - h2(key) = key.length()
  - shaded cells are removed cells

0	"ant"
1	"baboon"
2	"cat"
3	
4	"emu"
5	
6	
7	
8	
9	
10	

- What is the probe sequence for "baboon"?
  (h1 = 1, h2 = 6) try: 1 % 11 = 1
  (1 + 6) % 11 = 7
  (1 + 2*6) % 11 = 2
  (1 + 3*6) % 11 = 8

- **If we insert "baboon", in what position will it go?**

  A. **1** B. 7 C. 2 D. 8

  ↖ the first *removed* position seen while probing