Data Dictionary Revisited

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

<table>
<thead>
<tr>
<th>data structure</th>
<th>searching for an item</th>
<th>inserting an item</th>
</tr>
</thead>
<tbody>
<tr>
<td>a list implemented using an array</td>
<td>$O(\log n)$ using binary search</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>a list implemented using a linked list</td>
<td>$O(n)$ using linear search</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>binary search tree</td>
<td></td>
<td></td>
</tr>
<tr>
<td>balanced search trees (2-3 tree, B-tree, others)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Today, we'll look at hash tables, which allow us to do better than $O(\log n)$. 
Ideal Case: Searching = Indexing

- The optimal search and insertion performance is achieved when we can 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:
    ```java
    public class Player {
        private int jerseyNum;
        private String firstName;
    }
    ```
  - store the player records in an array:
    ```java
    Player[] teamRecords = new Player[100];
    ```
  - In such cases, we can perform both search and insertion in $O(1)$ time. For example:
    ```java
    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 it is in the sports-team example. Why?
  - 
  - 
  - 

- To handle these problems, we perform hashing:
  - use a hash function to convert the keys into array indices
    - "Sullivan" $\rightarrow$ 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 the set of possible keys to the set of integers.
- We then use the modulus operator to get a valid array index.

\[ \text{key value} \rightarrow \text{hash function} \rightarrow \text{integer} \equiv \text{integer in } [0, n - 1] \quad (n = \text{array length}) \]

- Here's a very simple hash function for keys of lower-case letters:
  \[ h(\text{key}) = \text{ASCII value of first char} - \text{ASCII value of 'a'} \]
- examples:
  \[ h(\text{'ant'}) = \text{ASCII for 'a'} - \text{ASCII for 'a'} = 0 \]
  \[ h(\text{'cat'}) = \text{ASCII for 'c'} - \text{ASCII for 'a'} = 2 \]
- \( h(\text{key}) \) 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

- If multiple items are assigned the same hash code, we "chain" them together.
- Each position in the hash table serves as a bucket that is able to store multiple data items.
- Two implementations:
  1. each bucket is itself an array
     - disadvantages:
       - large buckets can waste memory
       - a bucket may become full; overflow occurs when we try to add an item to a full bucket
  2. each bucket is a linked list
     - disadvantage:
       - the references in the nodes use additional memory
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 will consider three ways of finding an open position – a process known as probing.

- The hash table also performs probing to search for an item.
  - example: when searching for "wasp", we look in position 22 and then look in position 23
  - we can only stop a search when we reach an empty position

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 position, linear probing will eventually find it.

- Disadvantage: "clusters" of occupied positions develop, which tends to increase the lengths of subsequent probes.
  - probe length = the number of positions considered during a probe
Quadratic Probing

- Probe sequence: \( h(key), h(key) + 1, h(key) + 4, h(key) + 9, \ldots \), wrapping around as necessary.
  - the offsets are perfect squares: \( h + 1^2, h + 2^2, h + 3^2, \ldots \)

- Examples:
  - "ape" (\( h = 0 \)): try 0, 0 + 1 – open!
  - "bear" (\( h = 1 \)): try 1, 1 + 1, 1 + 4 – open!
  - "zebu"?

- Advantage: reduces clustering

- Disadvantage: it may fail to find an existing open position. For example:

<table>
<thead>
<tr>
<th>Table Size</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = occupied</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>25</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trying to insert a key with ( h(key) = 0 )</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>16</td>
<td>36</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offsets of the probe sequence in italics</td>
<td>2</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>49</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Double Hashing

- Use two hash functions:
  - \( h_1 \) computes the hash code
  - \( h_2 \) computes the increment for probing
  - probe sequence: \( h_1, h_1 + h_2, h_1 + 2* h_2, \ldots \)

- Examples:
  - \( h_1 = \) our previous \( h \)
  - \( h_2 = \) number of characters in the string
  - "ape" (\( h_1 = 0, h_2 = 3 \)): try 0, 0 + 3 – open!
  - "bear" (\( h_1 = 1, h_2 = 4 \)): try 1 – open!
  - "zebu"?

- Combines the good features of linear and quadratic probing:
  - reduces clustering
  - will find an open position if there is one, provided the table size is a prime number

<table>
<thead>
<tr>
<th>Table Size</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = occupied</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>25</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trying to insert a key with ( h(key) = 0 )</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>16</td>
<td>36</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offsets of the probe sequence in italics</td>
<td>2</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>49</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Removing Items Under Open Addressing

- Consider the following scenario:
  - 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!

- When we remove an item, we leave something behind to indicate an item was removed.

- Three types of positions: occupied, empty, removed.

- We stop probing when we encounter an empty position, but not when we encounter 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.

<table>
<thead>
<tr>
<th></th>
<th>&quot;ant&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&quot;cat&quot;</td>
</tr>
<tr>
<td>2</td>
<td>&quot;bear&quot;</td>
</tr>
<tr>
<td>3</td>
<td>&quot;emu&quot;</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>&quot;...&quot;</td>
</tr>
<tr>
<td>8</td>
<td>&quot;wolf&quot;</td>
</tr>
<tr>
<td>9</td>
<td>&quot;wasp&quot;</td>
</tr>
<tr>
<td>10</td>
<td>&quot;yak&quot;</td>
</tr>
<tr>
<td>11</td>
<td>&quot;zebra&quot;</td>
</tr>
</tbody>
</table>

An Interface For Hash Tables

```java
class 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 there is overflow and the pair cannot be added

- `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)
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.

  • 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:
      \[(h1 + n^2) \mod n = h1 \mod n\]
      \[(h1 + (n+1)^2) \mod n = (h1 + n^2 + 2n + 1) \mod n = (h1 + 1) \mod n\]
    • for double hashing:
      \[(h1 + nh2) \mod n = h1 \mod n\]
      \[(h1 + (n+1)h2) \mod n = (h1 + nh2 + h2) \mod n = (h1 + h2) \mod n\]
Avoiding an Infinite Loop (cont.)

```java
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

- Both of these methods begin by probing for the key.

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

public LLQueue<Object> remove(Object key) {
    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)
    - $26^5 \cdot 27^5 = 27^6 > 13 \text{ million possible keys}$
    - smallest code = $h_a(“a    ”) = 97 + 4 \cdot 32 = 225$
    - largest code = $h_a(“zzzzz”) = 5 \cdot 122 = 610$
    - $610 - 225 = 385 \text{ codes}$

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 $i$th 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
  - perform a rank search – get the kth largest item
We can do all of these things with a search tree.
Dictionaries in Java's Class Library

- Java provides a generic interface for dictionaries:
  
  ```java
  public interface Map<K, V> {
      ...
  }
  ```

  - K is the type of the keys
  - V is the type of the values

- It differs somewhat from our dictionary implementations:
  - `insert()` is called `put()`
  - `search()` is called `get()`
  - it does `not` support duplicates
    - to have multiple values for a given key, the client can use a list as the key's value

- The implementations of `Map<K, V>` include:
  - `TreeMap<K, V>` - uses a balanced search tree
  - `HashMap<K, V>` - uses a hash table with separate chaining