Lecture 12: Hash Open Indexing
Linear Probing

Insert the following values into the Hash Table using a hashFunction of % table size and linear probing to resolve collisions
38, 19, 8, 109, 10

Problem:
- Linear probing causes clustering
- Clustering causes more looping when probing

Primary Clustering
When probing causes long chains of occupied slots within a hash table
HW 4 Partner Form due tonight 11:59pm

HW 3 code modeling got more details

Midterm Friday
- Debugging
- More office hours today
- Review session on Wednesday 4pm – 5:50pm  PAA A102
Runtime

When is runtime good?
Empty table

When is runtime bad?
Table nearly full
When we hit a “cluster”

Maximum Load Factor?
$\lambda$ at most 1.0

When do we resize the array?
$\lambda \approx \frac{1}{2}$
Clusters are caused by picking new space near natural index

**Solution 2: Open Addressing**

**Type 2: Quadratic Probing**

If we collide instead try the next \( i^2 \) space

```java
public int hashFunction(String s)
    int naturalHash = this.getHash(s);
    if(natural hash in use) {
        int i = 1;
        while (index in use) {
            try (naturalHash + i * i);
            i++;
    }
Quadratic Probing

Insert the following values into the Hash Table using a hashFunction of % table size and quadratic probing to resolve collisions
89, 18, 49, 58, 79

<table>
<thead>
<tr>
<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>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>58</td>
<td>79</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>18</td>
<td>49</td>
<td></td>
</tr>
</tbody>
</table>

(49 % 10 + 0 * 0) % 10 = 9
(49 % 10 + 1 * 1) % 10 = 0
(58 % 10 + 0 * 0) % 10 = 8
(58 % 10 + 1 * 1) % 10 = 9
(58 % 10 + 2 * 2) % 10 = 2
(79 % 10 + 0 * 0) % 10 = 9
(79 % 10 + 1 * 1) % 10 = 0
(79 % 10 + 2 * 2) % 10 = 3

Problems:
- If $\lambda \geq \frac{1}{2}$ we might never find an empty spot
- Infinite loop!
- Can still get clusters
Secondary Clustering

Insert the following values into the Hash Table using a hashFunction of % table size and quadratic probing to resolve collisions
19, 39, 29, 9

When using quadratic probing sometimes need to probe the same sequence of table cells, not necessarily next to one another
Probing

- $h(k) = \text{the natural hash}$
- $h'(k, i) = \text{resulting hash after probing}$
- $i = \text{iteration of the probe}$
- $T = \text{table size}$

**Linear Probing:**

$h'(k, i) = (h(k) + i) \% T$

**Quadratic Probing**

$h'(k, i) = (h(k) + i^2) \% T$

For both types there are only $O(T)$ probes available
- Can we do better?
Double Hashing

Probing causes us to check the same indices over and over- can we check different ones instead?

Use a second hash function!

$$h'(k, i) = (h(k) + i \cdot g(k)) \mod T$$

<- Most effective if $g(k)$ returns value prime to table size

```java
public int hashFunction(String s)
    int naturalHash = this.getHash(s);
    if(natural hash in use) {
        int i = 1;
        while (index in use) {
            try (naturalHash + i * jump_Hash(key));
            i++;
    ```
Second Hash Function

Effective if $g(k)$ returns a value that is \textit{relatively prime} to table size
- If $T$ is a power of 2, make $g(k)$ return an odd integer
- If $T$ is a prime, make $g(k)$ return any smaller, non-zero integer
  - $g(k) = 1 + (k \% T(-1))$

How many different probes are there?
- $T$ different starting positions
- $T - 1$ jump intervals
- $O(T^2)$ different probe sequences
  - Linear and quadratic only offer $O(T)$ sequences
Resizing

How do we resize?
- Remake the table
- Evaluate the hash function over again.
- Re-insert.

When to resize?
- Depending on our load factor $\lambda$
- Heuristic:
  - for separate chaining $\lambda$ between 1 and 3 is a good time to resize.
  - For open addressing $\lambda$ between 0.5 and 1 is a good time to resize.
What are the running times for:

* **insert**
  - Best: $O(1)$
  - Worst: $O(n)$ (if insertions are always at the end of the linked list)

* **find**
  - Best: $O(1)$
  - Worst: $O(n)$

* **delete**
  - Best: $O(1)$
  - Worst: $O(n)$
Linear probing: Average-case insert

If $\lambda < 1$ we’ll find a spot eventually.

What’s the average running time?

**Uniform Hashing Assumption**

for any pair of elements $x, y$

the probability that $h(x) = h(y)$ is $\frac{1}{\text{TableSize}}$

If find is unsuccessful: $\frac{1}{2} \left(1 + \frac{1}{(1-\lambda)^2}\right)$

If find is successful: $\frac{1}{2} \left(1 + \frac{1}{(1-\lambda)}\right)$

We won’t ask you to prove these
1. Pick a hash function to:
   - Avoid collisions
   - Uniformly distribute data
   - Reduce hash computational costs

2. Pick a collision strategy
   - Chaining
     - LinkedList
     - AVL Tree
   - Probing
     - Linear
     - Quadratic
   - Double Hashing

   **No clustering**
   Potentially more “compact” ($\lambda$ can be higher)

   **Managing clustering can be tricky**
   Less compact ($\lambda < \frac{1}{2}$)
   Array lookups tend to be a constant factor faster than traversing pointers
Summary

Separate Chaining
- Easy to implement
- Running times $O(1 + \lambda)$

Open Addressing
- Uses less memory.
- Various schemes:
  - Linear Probing – easiest, but need to resize most frequently
  - Quadratic Probing – middle ground
  - Double Hashing – need a whole new hash function, but low chance of clustering.

Which you use depends on your application and what you’re worried about.
Other applications of hashing

- Cryptographic hash functions: Hash functions with some additional properties
  - Commonly used in practice: SHA-1, SHA-265
  - To verify file integrity. When you share a large file with someone, how do you know that the other person got the exact same file? Just compare hash of the file on both ends. Used by file sharing services (Google Drive, Dropbox)
  - For password verification: Storing passwords in plaintext is insecure. So your passwords are stored as a hash.
  - For Digital signature
  - Lots of other crypto applications

- Finding similar records: Records with similar but not identical keys
  - Spelling suggestion/corrector applications
  - Audio/video fingerprinting
  - Clustering

- Finding similar substrings in a large collection of strings
  - Genomic databases
  - Detecting plagiarism

- Geometric hashing: Widely used in computer graphics and computational geometry
Wrap Up

Hash Tables:
- Efficient find, insert, delete on average, under some assumptions
- Items not in sorted order
- Tons of real world uses
- ...and really popular in tech interview questions.

Need to pick a good hash function.
- Have someone else do this if possible.
- Balance getting a good distribution and speed of calculation.

Resizing:
- Always make the table size a prime number.
- $\lambda$ determines when to resize, but depends on collision resolution strategy.
List ADT

**ArrayList**

- **state**
  - data[]
  - size
- **behavior**
  - get(index) return data[index]
  - set(index, value) update data[index]
  - append(value) add value at end
  - insert(index, value) add value at index
  - delete(index) remove value at index
  - size() return size

**LinkedList**

- **state**
  - Node front
  - size
- **behavior**
  - get(index) return node's value
  - set(index, value) update node's value
  - append(value) create new node, add to end
  - insert(index, value) create new node, add at index
  - delete(index) remove node
  - size() return size

---

**ArrayList**

- Set of ordered items
- Count of items

**behavior**

- `get(index)` return item at index
- `set(index, item)` replace item at index
- `append(item)` add item to end
- `insert(index, item)` add item at index
- `delete(index)` delete item at index
- `size()` count of items

**LinkedList**

- Node front
- Size

**behavior**

- `get(index)` return node's value
- `set(index, value)` update node's value
- `append(value)` create new node, add to end
- `insert(index, value)` create new node, add at index
- `delete(index)` remove node
- `size()` return size

**Examples**

**ArrayList**

```
0 1 2 3 4 5 6 7
88.6 26.1 94.4 0 0
```

**LinkedList**

```
88.6 → 26.1 → 94.4
```
Stack ADT

**stack**: A collection based on the principle of adding elements and retrieving them in the opposite order.
- Last-In, First-Out ("LIFO")
- Elements are stored in order of insertion.
  - We do not think of them as having indexes.
- Client can only add/remove/examine the last element added (the "top").

**Stack ADT**

<table>
<thead>
<tr>
<th>state</th>
<th>behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set of ordered items</td>
<td>push(item) add item to top</td>
</tr>
<tr>
<td>Number of items</td>
<td>pop() return and remove item at top</td>
</tr>
<tr>
<td></td>
<td>peek() look at item at top</td>
</tr>
<tr>
<td></td>
<td>size() count of items</td>
</tr>
<tr>
<td></td>
<td>isEmpty() count of items is 0?</td>
</tr>
</tbody>
</table>

**ArrayStack<E>**

- state
  - data[]
  - size
- behavior
  - push data[size] = value, if out of room grow data
  - pop return data[size - 1], size-1
  - peek return data[size - 1]
  - size return size
  - isEmpty return size == 0

**LinkedStack<E>**

- state
  - Node top
  - size
- behavior
  - push add new node at top
  - pop return and remove node at top
  - peek return node at top
  - size return size
  - isEmpty return size == 0
queue: Retrieves elements in the order they were added.
- First-In, First-Out ("FIFO")
- Elements are stored in order of insertion but don't have indexes.
- Client can only add to the end of the queue, and can only examine/remove the front of the queue.

ArrayQueue<E>

state
- data[]
- Size
- front index
- back index

behavior
- add(item) add item to back
- remove() remove and return item at front
- peek() return item at front
- size() count of items
- isEmpty() count of items is 0?

LinkedQueue<E>

state
- Node front
- Node back
- size

behavior
- add add node to back
- remove remove and return node at front
- peek return node at front
- size return size
- isEmpty return size == 0
Map/Dictionary ADT

dictionary: Holds a set of unique keys and a collection of values, where each key is associated with one value.
- a.k.a. "dictionary", "associative array", "hash"

### ArrayDictionary<K, V>

**state**
Pair<K, V>[] data

**behavior**
- put(key, item) add item to collection indexed with key
- get(key) return item associated with key
- containsKey(key) return if key already in use
- remove(key) remove item and associated key
- size() return count of items

### LinkedDictionary<K, V>

**state**
- front
- size

**behavior**
- put if key is unused, create new with pair, add to front of list, else replace with new value
- get scan all pairs looking for given key, return associated item if found
- containsKey scan all pairs, return if key is found
- remove scan all pairs, skip pair to be removed
- size return count of items in dictionary

```
<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;at&quot;</td>
<td>43</td>
</tr>
<tr>
<td>&quot;in&quot;</td>
<td>37</td>
</tr>
<tr>
<td>&quot;you&quot;</td>
<td>22</td>
</tr>
<tr>
<td>&quot;the&quot;</td>
<td>56</td>
</tr>
</tbody>
</table>
```
Binary Search Trees

A **binary search tree** is a binary tree that contains comparable items such that for every node, all children to the left contain smaller data and all children to the right contain larger data.

Binary Search Trees allow us to:
- quickly find what we’re looking for
- add and remove values easily

Dictionary Operations:
Runtime in terms of height, “h”
get() – O(h)
put() – O(h)
remove() – O(h)

What do you replace the node with?
Largest in left sub tree or smallest in right sub tree

TreeDictionary\(<K, V>\)

- **state**
  - overallRoot
  - size
- **behavior**
  - put if key is unused, create new pair, place in BST order, rotate to maintain balance
  - get traverse through tree using BST property, return item if found
  - containsKey traverse through tree using BST property, return if key is found
  - remove traverse through tree using BST property, replace or nullify as appropriate
  - size return count of items in dictionary
Meet AVL Trees

**AVL Trees** must satisfy the following properties:
- **binary trees**: all nodes must have between 0 and 2 children
- **binary search tree**: for all nodes, all keys in the left subtree must be smaller and all keys in the right subtree must be larger than the root node
- **balanced**: for all nodes, there can be no more than a difference of 1 in the height of the left subtree from the right.
  \[ \text{Math.abs(height(left subtree) - height(right subtree))} \leq 1 \]

AVL stands for Adelson-Velsky and Landis (the inventors of the data structure)

Dictionary Operations:

- `get()` – same as BST
- `containsKey()` – same as BST
- `put()` – same as BST + rebalance
- `remove()` – same as BST + rebalance
How long does AVL insert take?

AVL insert time = BST insert time + time it takes to rebalance the tree
  = $O(\log n)$ + time it takes to rebalance the tree

How long does rebalancing take?
- Assume we store in each node the height of its subtree.
- How long to find an unbalanced node:
  - Just go back up the tree from where we inserted. $\leftarrow O(\log n)$
- How many rotations might we have to do?
  - Just a single or double rotation on the lowest unbalanced node. $\leftarrow O(1)$

AVL insert time = $O(\log n) + O(\log n) + O(1) = O(\log n)$
Why are we so obsessed with Dictionaries? **It’s all about data baby!**

When dealing with data:
- Adding data to your collection
- Getting data out of your collection
- Rearranging data in your collection

**SUPER common in comp sci**
- Databases
- Network router tables
- Compilers and Interpreters

<table>
<thead>
<tr>
<th>Operation</th>
<th>ArrayList</th>
<th>LinkedList</th>
<th>BST</th>
<th>AVLTree</th>
</tr>
</thead>
<tbody>
<tr>
<td>put(key, value)</td>
<td>best</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td></td>
<td>average</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(logn)</td>
</tr>
<tr>
<td></td>
<td>worst</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>get(key)</td>
<td>best</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td></td>
<td>average</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(logn)</td>
</tr>
<tr>
<td></td>
<td>worst</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>remove(key)</td>
<td>best</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(logn)</td>
</tr>
<tr>
<td></td>
<td>average</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(logn)</td>
</tr>
<tr>
<td></td>
<td>worst</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>
Hashing

HashMap<Integer>

**state**
- `Data[]`
- `size`

**behavior**
- `put mod key by table size, put item at result`
- `get mod key by table size, get item at result`
- `containsKey mod key by table size, return data[result] == null remove mod key by table size, nullify element at result`
- `size return count of items in dictionary`

<table>
<thead>
<tr>
<th>Operation</th>
<th>Separate Chaining</th>
<th>Probing</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>put(key, value)</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>best</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>average</td>
<td>O(1 + λ)</td>
<td>O(1 + λ)</td>
</tr>
<tr>
<td>worst</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td><code>get(key)</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>best</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>average</td>
<td>O(1 + λ)</td>
<td>O(1 + λ)</td>
</tr>
<tr>
<td>worst</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td><code>remove(key)</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>best</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>average</td>
<td>O(1 + λ)</td>
<td>O(1 + λ)</td>
</tr>
<tr>
<td>worst</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>