Motivation

- Arrays provide an indirect way to access a set.
- Many times we need an association between two sets, or a set of keys and associated data.
- Ideally we would like to access this data directly with the keys.
- We would like a data structure that supports fast search, insertion, and deletion.
  - Do not usually care about sorting.
- The abstract data type is usually called a Dictionary or Partial Map
  - float googleStockPrice = stocks[“Goog”].CurrentPrice;

Dictionaries

- What is the best way to implement this?
  - Linked Lists?
  - Double Linked Lists?
  - Queues?
  - Stacks?
  - Multiple indexed arrays (e.g., data[key[i]])?
- To answer this, ask what the complexity of the operations are:
  - Insertion
  - Deletion
  - Search

Direct Addressing

- Let’s look at an easy case, suppose:
  - The range of keys is 0..m-1
  - Keys are distinct
- Possible solution
  - Set up an array T[0..m-1] in which
    - $T[i] = x$ if $x \in T$ and key[$x$] = $i$
    - $T[i] = \text{NULL}$ otherwise
  - This is called a direct-address table
    - Operations take $O(1)$ time!
    - So what’s the problem?
Direct Addressing

- Direct addressing works well when the range $m$ of keys is relatively small.
- But what if the keys are 32-bit integers?
  - Problem 1: direct-address table will have $2^{32}$ entries, more than 4 billion entries.
  - Problem 2: even if memory is not an issue, the time to initialize the elements to NULL may be a concern.
- Solution: map keys to a smaller range $0..p-1$.
  - Desire $p = O(m)$.

Hash Table

- Hash Tables provide $O(1)$ support for all of these operations!
- The key is rather than index an array directly, index it through some function, $h(x)$, called a hash function.
  - $myArray[ h(index) ]$
- Key questions:
  - What is the set that the $x$ comes from?
  - What is $h()$ and what is its range?

Hash Table

- Consider this problem:
  - If I know a priori the $m$ keys from some finite set $U$, is it possible to develop a function $h(x)$ that will uniquely map the $m$ keys onto the set of numbers $0..m-1$?

Hash Functions

- In general a difficult problem. Try something simpler.
Hash Functions

- A collision occurs when \( h(x) \) maps two keys to the same location.

\[ h(x) = x \mod N \]

is a hash function for integer keys.

- The integer \( h(x) \) is called the hash value of \( x \).

- A hash table for a given key type consists of:
  - Hash function \( h \)
  - Array (called table) of size \( N \)
  - The goal is to store item \((k, o)\) at index \( i = h(k) \)

Example

- We design a hash table storing employees records using their social security number, SSN as the key.
  - SSN is a nine-digit positive integer

- Our hash table uses an array of size \( N = 10,000 \) and the hash function \( h(x) = \) last four digits of \( x \).

Our hash table uses an array of size \( N = 100 \).

- We have \( n = 49 \) employees.
  - Need a method to handle collisions.
  - As long as the chance for collision is low, we can achieve this goal.

- Setting \( N = 1000 \) and looking at the last four digits will reduce the chance of collision.
Collisions

- Can collisions be avoided?
  - In general, no. See perfect hashing for the case where the set of keys is static (not covered).
- Two primary techniques for resolving collisions:
  - **Chaining** – keep a collection at each key slot.
  - **Open addressing** – if the current slot is full use the *next open* one.

Chaining

- Chaining puts elements that hash to the same slot in a linked list:

  ![Diagram of Chaining]

  - How do we insert an element?
  - How do we delete an element?
    - Do we need a doubly-linked list for efficient delete?
Chaining

- How do we search for an element with a given key?

Open Addressing

- Basic idea:
  - To insert: if slot is full, try another slot, …, until an open slot is found (**probing**)
  - To search, follow same sequence of probes as would be used when inserting the element
    - If reach element with correct key, return it
    - If reach a NULL pointer, element is not in table
- Good for fixed sets (adding but no deletion)
  - Example: spell checking

Open Addressing

- The colliding item is placed in a different cell of the table.
  - No dynamic memory.
  - Fixed Table size.
- **Load factor**: $n/N$, where $n$ is the number of items to store and $N$ the size of the hash table.
  - Clearly, $n \leq N$, or $n/N \leq 1$.
  - To get a reasonable performance, $n/N < 0.5$.

Probing

- They key question is what should the next cell to try be?
- Random would be great, but we need to be able to repeat it.
- Three common techniques:
  - Linear Probing (useful for discussion only)
  - Quadratic Probing
  - Double Hashing
Linear Probing

- **Linear probing** handles collisions by placing the colliding item in the *next* (circularly) available table cell.
- Each table cell inspected is referred to as a **probe**.
- Colliding items lump together, causing future collisions to cause a longer sequence of probes.

Example:
- \[ h(x) = x \mod 13 \]
- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

<table>
<thead>
<tr>
<th></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>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>41</td>
<td>18</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

Search with Linear Probing

- Consider a hash table \( A \) that uses linear probing
- **get**\((k)\)
  - We start at cell \( h(k) \)
  - We probe consecutive locations until one of the following occurs
    - An item with key \( k \) is found, or
    - An empty cell is found, or
    - \( N \) cells have been unsuccessfully probed
  - To ensure the efficiency, if \( k \) is not in the table we want to find an empty cell as soon as possible. The load factor can NOT be close to 1.

```
Algorithm
    i ← h(k)
p ← 0
repeat
    c ← A[i]
    if c = ∅
        return null
    else if c.key() = k
        return c.element()
    else
        i ← (i + 1) mod N
        p ← p + 1
until p = N
return null
```

Linear Probing

- Search for key=20.
  - \( h(20) = 20 \mod 13 = 7 \).
  - Go through rank 8, 9, ..., 12, 0.
- Search for key=15
  - \( h(15) = 15 \mod 13 = 2 \).
  - Go through rank 2, 3 and return null.

Example:
- \[ h(x) = x \mod 13 \]
- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, 12, 20 in this order

<table>
<thead>
<tr>
<th></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>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>41</td>
<td>18</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

Updates with Linear Probing

- To handle insertions and deletions, we introduce a special object, called **AVAILABLE**, which replaces deleted elements
- **remove**\((k)\)
  - We search for an entry with key \( k \)
  - If such an entry \((k, o)\) is found, we replace it with the special item **AVAILABLE**
  - Have to modify other methods to skip available cells.
- **put**\((k, o)\)
  - We throw an exception if the table is full
  - We start at cell \( h(k) \)
  - We probe consecutive cells until one of the following occurs
    - A cell \( i \) is found that is either empty or stores **AVAILABLE**, or
    - \( N \) cells have been unsuccessfully probed
  - We store entry \((k, o)\) in cell \( i \)
Primary clustering occurs with linear probing because the same linear pattern:
- if a bin is inside a cluster, then the next bin must either:
  - also be in that cluster, or
  - expand the cluster
- Instead of searching forward in a linear fashion, consider searching forward using a quadratic function

Suppose that an element should appear in bin $h$:
- if bin $h$ is occupied, then check the following sequence of bins:
  $h + 1, h + 4, h + 9, h + 16, h + 25, \ldots$
  $h + 1^2, h + 2^2, h + 3^2, h + 4^2, h + 5^2, \ldots$
- For example, with $M = 17$:

If one of $h + i^2$ falls into a cluster, this does not imply the next one will

For example, suppose an element was to be inserted in bin 23 in a hash table with 31 bins
- The sequence in which the bins would be checked is:
  23, 24, 27, 1, 8, 17, 28, 10, 25, 11, 30, 20, 12, 6, 2, 0
Quadratic Probing

- Even if two bins are initially close, the sequence in which subsequent bins are checked varies greatly.
- Again, with $M = 31$ bins, compare the first 16 bins which are checked starting with 22 and 23:

  22, 23, 26, 0, 7, 16, 27, 9, 24, 10, 29, 19, 11, 5, 1, 30
  23, 24, 27, 1, 8, 17, 28, 10, 25, 11, 30, 20, 12, 6, 2, 0

Quadratic Probing

- Thus, quadratic probing solves the problem of primary clustering.
- Unfortunately, there is a second problem which must be dealt with.
- Suppose we have $M = 8$ bins:

  $1^2 \equiv 1$, $2^2 \equiv 4$, $3^2 \equiv 1$

- In this case, we are checking bin $h + 1$ twice having checked only one other bin.

Quadratic Probing

- Unfortunately, there is no guarantee that $h + i^2 \mod M$ will cycle through 0, 1, ..., $M - 1$.
- Solution:
  - require that $M$ be prime
  - in this case, $h + i^2 \mod M$ for $i = 0, ..., (M - 1)/2$ will cycle through exactly $(M + 1)/2$ values before repeating.

Quadratic Probing

- Example with $M = 11$:

  0, 1, 4, 9, 16 \equiv 5, 25 \equiv 3, 36 \equiv 3

- With $M = 13$:

  0, 1, 4, 9, 16 \equiv 3, 25 \equiv 12, 36 \equiv 10, 49 \equiv 10

- With $M = 17$:

  0, 1, 4, 9, 16, 25 \equiv 8, 36 \equiv 2, 49 \equiv 15, 64 \equiv 13, 81 \equiv 13
Quadratic Probing

- Thus, quadratic probing avoids primary clustering
- Unfortunately, we are not guaranteed that we will use all the bins
- In reality, if the hash function is reasonable, this is not a significant problem until $\lambda$ approaches 1

Secondary Clustering

- The phenomenon of primary clustering will not occur with quadratic probing
- However, if multiple items all hash to the same initial bin, the same sequence of numbers will be followed
- This is termed secondary clustering
- The effect is less significant than that of primary clustering

Double Hashing

- Use two hash functions
- If $M$ is prime, eventually will examine every position in the table
- `double_hash_insert(K)`
  - if(table is full) error
  - probe = $h1(K)$
  - offset = $h2(K)$
  - while (table[probe] occupied)
    - probe = (probe + offset) mod M
  - table[probe] = K

- Many of same (dis)advantages as linear probing
- Distributes keys more uniformly than linear probing does
- Notes:
  - $h2(x)$ should never return zero.
  - $M$ should be prime.
Double Hashing Example

- \( h_1(K) = K \mod 13 \)
- \( h_2(K) = 8 - K \mod 8 \)
  - we want \( h_2 \) to be an offset to add
  - 18 41 22 44 59 32 31 73

\[
\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
44 & 41 & 73 & 18 & 32 & 53 & 31 & 22 & & & & & \\
\end{array}
\]

Open Addressing Summary

- In general, the hash function contains two arguments now:
  - Key value
  - Probe number
  \( h(k,p), \ p=0,1,...,m-1 \)
- Probe sequences
  \( <h(k,0), h(k,1), ..., h(k,m-1)> \)
  - Should be a permutation of \( <0,1,...,m-1> \)
  - There are \( m! \) possible permutations
  - Good hash functions should be able to produce all \( m! \) probe sequences

Open Addressing Summary

- None of the methods discussed can generate more than \( m^2 \) different probing sequences.
- Linear Probing:
  - Clearly, only \( m \) probe sequences.
- Quadratic Probing:
  - The initial key determines a fixed probe sequence, so only \( m \) distinct probe sequences.
- Double Hashing
  - Each possible pair \( (h_1(k),h_2(k)) \) yields a distinct probe, so \( m^2 \) permutations.

Choosing A Hash Function

- Clearly choosing the hash function well is crucial.
  - What will a worst-case hash function do?
  - What will be the time to search in this case?
- What are desirable features of the hash function?
  - Should distribute keys uniformly into slots
  - Should not depend on patterns in the data
From Keys to Indices

- A hash function is usually the composition of two maps:
  - hash code map: key $\rightarrow$ integer
  - compression map: integer $\rightarrow$ $[0, N - 1]$
- An essential requirement of the hash function is to map equal keys to equal indices
- A “good” hash function minimizes the probability of collisions

Java Hash

- Java provides a `hashCode()` method for the Object class, which typically returns the 32-bit memory address of the object.
- This default hash code would work poorly for `Integer` and `String` objects
- The `hashCode()` method should be suitably redefined by classes.

Popular Hash-Code Maps

- **Integer cast**: for numeric types with 32 bits or less, we can reinterpret the bits of the number as an `int`
- **Component sum**: for numeric types with more than 32 bits (e.g., `long` and `double`), we can add the 32-bit components.

Polynomial accumulation: for strings of a natural language, combine the character values (ASCII or Unicode) $a_0 + a_1 x + ... + x_{n-1} a_{n-1}$
Popular Hash-Code Maps

- The polynomial is computed with \textit{Horner's rule}, ignoring overflows, at a fixed value $x$:
  
  \[
  a_0 + x (a_1 + x (a_2 + \ldots x (a_{n-2} + x a_{n-1} ) \ldots ))
  \]

- The choice $x = 33, 37, 39, \text{or} 41$ gives at most 6 collisions on a vocabulary of 50,000 English words

- Why is the component-sum hash code bad for strings?

Random Hashing

- Random hashing
  - Uses a simple random number generation technique
  - Scatters the items “randomly” throughout the hash table

Popular Compression Maps

- \textbf{Division}: $h(k) = \lvert k \rvert \mod N$
  - the choice $N = 2k$ is bad because not all the bits are taken into account
  - the table size $N$ is usually chosen as a prime number
  - certain patterns in the hash codes are propagated

- \textbf{Multiply, Add, and Divide (MAD)}:
  - $h(k) = \lvert ak + b \rvert \mod N$
  - eliminates patterns provided $a \mod N \neq 0$
  - same formula used in linear congruential (pseudo) random number generators

The Division Method

- $h(k) = k \mod m$
  - In words: hash $k$ into a table with $m$ slots using the slot given by the remainder of $k$ divided by $m$

  - \textit{What happens to elements with adjacent values of $k$?}

  - \textit{What happens if $m$ is a power of 2 (say $2^p$)?}

  - \textit{What if $m$ is a power of 10?}

  - Upshot: pick table size $m = \text{prime number}$ not too close to a power of 2 (or 10)
The Multiplication Method

- For a constant $A$, $0 < A < 1$:
  - $h(k) = \lfloor m (kA - \lfloor kA \rfloor) \rfloor$

  What does this term represent?

  - Fractional part of $kA$
  - Choose $m = 2^p$
  - Choose $A$ not too close to 0 or 1
  - Knuth: Good choice for $A = (\sqrt{5} - 1)/2$

Analysis of Chaining

- Assume simple uniform hashing: each key in table is equally likely to be hashed to any slot.
- Given $n$ keys and $m$ slots in the table:
  the load factor $\alpha = n/m = \text{average \# keys per slot}$

  What will be the average cost of an unsuccessful search for a key?

  - $O(1 + \alpha)$
Analysis of Chaining

- What will be the average cost of an unsuccessful search for a key?
- O(1+α)

- What will be the average cost of a successful search?
- O(1 + α/2) = O(1 + α)

Analysis of Chaining

- So the cost of searching = O(1 + α)
- If the number of keys n is proportional to the number of slots in the table, what is α?
- A: α = O(1)
  - In other words, we can make the expected cost of searching constant if we make α constant

Analysis of Open Addressing

- Consider the load factor, α, and assume each key is uniformly hashed.
- Probability that we hit an occupied cell is then α.
- Probability that we the next probe hits an occupied cell is also α.
- Will terminate if an unoccupied cell is hit: α(1- α).
- From Theorem 11.6, the expected number of probes in an unsuccessful search is at most 1/(1- α).
- Theorem 11.8: Expected number of probes in a successful search is at most:
  \[
  \frac{1}{α} \ln \left( \frac{1}{1-α} \right)
  \]