Readings and References

• Reading
  › Chapter 5
The Need for Speed

• Data structures we have looked at so far
  › Use comparison operations to find items
  › Need $O(\log N)$ time for Find and Insert

• In real world applications, $N$ is typically between 100 and 100,000 (or more)
  › $\log N$ is between 6.6 and 16.6

• Hash tables are an abstract data type designed for $O(1)$ Find and Inserts
Fewer Functions Faster

• compare lists and stacks
  › by reducing the flexibility of what we are allowed to do, we can increase the performance of the remaining operations
  › insert(L,X) into a list versus push(S,X) onto a stack

• compare trees and hash tables
  › trees provide for known ordering of all elements
  › hash tables just let you (quickly) find an element
Limited Set of Hash Operations

• For many applications, a limited set of operations is all that is needed
  › Insert, Find, and Delete
  › Note that no ordering of elements is implied

• For example, a compiler needs to maintain information about the symbols in a program
  › user defined
  › language keywords
Direct Address Tables

• Direct addressing using an array is very fast
• Assume
  › keys are integers in the set $U = \{0, 1, \ldots, m-1\}$
  › $m$ is small
  › no two elements have the same key
• Then just store each element at the array location $\text{array}[\text{key}]$
  › search, insert, and delete are trivial
Direct Access Table

[U (universe of keys)]

[K (Actual keys)]

[Cormen, et al]
Direct Address Implementation

Delete(Table t, ElementType x)
    T[key[x]] = NULL

Insert(Table t, ElementType x)
    T[key[x]] = x

Find(Table t, Key k)
    return T[k]
An Issue

• The largest possible key in U may be much larger than the number of elements actually stored (|U| much greater than |K|)
  › the table is very sparse and wastes space
  › in worst case, table too large to have in memory

• If most keys in U are used
  › direct addressing can work very well

• If most keys in U are not used
  › need to map U to a smaller set closer in size to K
Mapping the Keys

Hash Function

Table indices

Key Universe

U

432 0 72345 62
928104 103673

K

254 3456 54724 81

Hashing - Lecture 10

10/21/02
Hashing Schemes

- We want to store N items in a table of size M, at a location computed from the key K
- Hash function
  - Method for computing table index from key
- Collision resolution strategy
  - How to handle two keys that hash to the same index
Looking for an Element

• Data records can be stored in arrays.
  › A[0] = {“CHEM 110”, Size 89}
  › A[17] = {“CSE 373”, Size 85}

• Class size for CSE 373?
  › Linear search the array – O(N) worst case time
  › Binary search - O(log N) worst case
Go Directly to the Element

- What if we could directly index into the array using the key?
  - $A[“CSE 373”] = \{\text{Size 85}\}$

- Main idea behind hash tables
  - Use a key based on some aspect of the data element to index directly into an array
  - $O(1)$ time to access records
Indexing into Hash Table

• Need a fast *hash function* to convert the element key (string or number) to an integer (the *hash value*) (ie, map from U to index)
  › Then use this value to index into an array
  › Hash(“CSE 373”) = 157, Hash(“CSE 143”) = 101

• Output of the hash function
  › must always be less than size of array
  › should be as evenly distributed as possible
Choosing the Hash Function

• What properties do we want from a hash function?
  › Want universe of hash values to be distributed randomly to minimize collisions
  › Don’t want systematic nonrandom pattern in selection of keys to lead to systematic collisions
  › Want hash value to depend on all values in entire key and their positions
The Key Values are Important

• Notice that one issue with all the hash functions is that the actual content of the key set matters
• The elements in K (the keys that are used) are quite possibly a restricted subset of U, not just a random collection
  › variable names, words in the English language, reserved keywords, telephone numbers, etc, etc
Simple Hashes

- It's possible to have very simple hash functions if you are certain of your keys
- For example,
  - suppose we know that the keys $s$ will be real numbers uniformly distributed over $0 \leq s < 1$
  - Then a very fast, very good hash function is
    - $\text{hash}(s) = \text{floor}(s \cdot m)$
    - where $m$ is the size of the table
Very Simple Mapping

- $hash(s) = \text{floor}(s \cdot m)$ maps from $0 \leq s < 1$ to $0..m-1$

  \[ m = 10 \]

Note the even distribution. There are collisions, but we will deal with them later.
Perfect Hashing

- In some cases it's possible to map a known set of keys uniquely to a set of index values.
- You must know every single key beforehand and be able to derive a function that works \textit{one-to-one} (not necessarily \textit{onto}).
Mod Hash Function

- One solution for a less constrained key set
  - modular arithmetic
- \( a \mod \text{size} \)
  - remainder when "a" is divided by "size"
  - in C or Java this is written as \( r = a \% \text{size}; \)
  - If TableSize = 251
    - 408 mod 251 = 157
    - 352 mod 251 = 101
Modulo Mapping

- $a \mod m$ maps from integers to 0..m-1
  - one to one? no
  - onto? yes

\[ x \mod 4 \]
Hashing Integers

• If keys are integers, we can use the hash function:
  › Hash(key) = key mod TableSize

• Problem 1: What if TableSize is 11 and all keys are 2 repeated digits? (eg, 22, 33, …)
  › all keys map to the same index
  › Need to pick TableSize carefully: often, a prime number
Nonnumerical Keys

- Many hash functions assume that the universe of keys is the natural numbers $\mathbb{N} = \{0, 1, \ldots\}$
- Need to find a function to convert the actual key to a natural number quickly and effectively before or during the hash calculation
- Generally work with the ASCII character codes when converting strings to numbers
Characters to Integers

- If keys are strings can get an integer by adding up ASCII values of characters in key
- We are converting a very large string \(c_0c_1c_2 \ldots c_n\) to a relatively small number \(c_0+c_1+c_2+\ldots+c_n \mod \text{size.}\)

<table>
<thead>
<tr>
<th>character</th>
<th>C</th>
<th>S</th>
<th>E</th>
<th>3</th>
<th>7</th>
<th>3</th>
<th>&lt;0&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCII value</td>
<td>67</td>
<td>83</td>
<td>69</td>
<td>32</td>
<td>51</td>
<td>55</td>
<td>51</td>
</tr>
</tbody>
</table>
Hash Must be Onto Table

- Problem 2: What if TableSize is 10,000 and all keys are 8 or less characters long?
  - chars have values between 0 and 127
  - Keys will hash only to positions 0 through $8 \times 127 = 1016$

- Need to distribute keys over the entire table or the extra space is wasted
Problems with Adding Characters

- Problems with adding up character values for string keys
  - If string keys are short, will not hash evenly to all of the hash table
  - Different character combinations hash to same value
    - “abc”, “bca”, and “cab” all add up to the same value
Characters as Integers

• An character string can be thought of as a base 256 number. The string $c_1c_2\ldots c_n$ can be thought of as the number $c_n + 256c_{n-1} + 256^2c_{n-2} + \ldots + 256^{n-1} c_1$

• Use Horner’s Rule to Hash!

```plaintext
r = 0;
for i = 1 to n do
    r := (c[i] + 256*r) mod TableSize
```
Collisions

- A **collision** occurs when two different keys hash to the same value
  - E.g. For TableSize = 17, the keys 18 and 35 hash to the same value
    - 18 mod 17 = 1 and 35 mod 17 = 1
- Cannot store both data records in the same slot in array!
Collision Resolution

- Separate Chaining
  - Use data structure (such as a linked list) to store multiple items that hash to the same slot
- Open addressing (or probing)
  - search for empty slots using a second function and store item in first empty slot that is found
Resolution by Chaining

- Each hash table cell holds pointer to linked list of records with same hash value
- Collision: Insert item into linked list
- To Find an item: compute hash value, then do Find on linked list
- Note that there are potentially as many as TableSize lists
Why Lists?

- Can use List ADT for Find/Insert/Delete in linked list
  - $O(N)$ runtime where $N$ is the number of elements in the particular chain
- Can also use Binary Search Trees
  - $O(\log N)$ time instead of $O(N)$
  - But the number of elements to search through should be small
  - generally not worth the overhead of BSTs
Load Factor of a Hash Table

• Let \( N \) = number of items to be stored
• Load factor \( \lambda = \frac{N}{\text{TableSize}} \)
  ‣ TableSize = 101 and \( N = 505 \), then \( \lambda = 5 \)
  ‣ TableSize = 101 and \( N = 10 \), then \( \lambda = 0.1 \)
• Average length of chained list = \( \lambda \) and so average time for accessing an item = \( O(1) + O(\lambda) \)
  ‣ Want \( \lambda \) to be close to 1 (i.e. TableSize \( \approx N \))
  ‣ But chaining continues to work for \( \lambda > 1 \)
Resolution by Open Addressing

• No links, all keys are in the table
  › reduced overhead saves space
• When searching for \( x \), check locations \( h_1(x) \), \( h_2(x) \), \( h_3(x) \), \( \ldots \) until either
  › \( x \) is found; or
  › we find an empty location (\( x \) not present)
• Various flavors of open addressing differ in which probe sequence they use
Cell Full? Keep Looking.

- \( h_i(X) = (\text{Hash}(X) + F(i)) \mod \text{TableSize} \)
  - Define \( F(0) = 0 \)
- \( F \) is the collision resolution function.
  Some possibilities:
  - **Linear**: \( F(i) = i \)
  - **Quadratic**: \( F(i) = i^2 \)
  - **Double Hashing**: \( F(i) = i \cdot \text{Hash}_2(X) \)
Linear Probing

• When searching for $K$, check locations $h(K)$, $h(K) + 1$, $h(K) + 2$, ... mod TableSize until either
  › $K$ is found; or
  › we find an empty location ($K$ not present)
• If table is very sparse, almost like separate chaining.
• When table starts filling, we get clustering but still constant average search time.
• Full table $\Rightarrow$ infinite loop.
Primary Clustering Problem

- Once a block of a few contiguous occupied positions emerges in table, it becomes a “target” for subsequent collisions
- As clusters grow, they also merge to form larger clusters.
- Primary clustering: elements that hash to different cells probe same alternative cells
Linear Probing – Clustering

[Diagram showing no collision, collision in small cluster, and collision in large cluster.]

[R. Sedgewick]
Quadratic Probing

• When searching for $x$, check locations $h_1(X), h_1(X) + i^2, h_1(X) + i^3, \ldots \mod \text{TableSize}$ until either
  › $x$ is found; or
  › we find an empty location ($x$ not present)

• No primary clustering but secondary clustering possible
Double Hashing

- When searching for \( x \), check locations \( h_1(X) \), \( h_1(X) + h_2(X) \), \( h_1(X) + 2h_2(X) \), \ldots \) mod Tablesize until either
  - \( x \) is found; or
  - we find an empty location (\( x \) not present)

- Must be careful about \( h_2(X) \)
  - Not 0 and not a divisor of \( M \)
  - eg, \( h_1(k) = k \mod m_1, \ h_2(k) = 1 + (k \mod m_2) \)
  - where \( m_2 \) is slightly less than \( m_1 \)
Double Hashing

\[ h_1(x) + h_2(x), \quad h_1(x) + 2h_2(x), \quad h_1(x) + 3h_2(x), \ldots \]

[no collision, try again at \( h_1(x) + h_2(x) \)]
[no collision, try again at \( h_1(z) + h_2(z), \) at \( h_1(z) + 2h_2(z), \) at \( h_1(z) + 3h_2(z), \ldots \)]

[R. Sedgewick]
Rules of Thumb

• Separate chaining is simple but wastes space…
• Linear probing uses space better, is fast when tables are sparse, interacts well with paging and caching
• Double hashing is space efficient, fast (get initial hash and increment at the same time), needs careful implementation
• For average cost about $t + O(1)$
  › Max load for Linear Probing is $1 - 1/\sqrt{t}$
  › Max load for Double Hashing is $1 - 1/t$
Rehashing – Rebuild the Table

• Need to use lazy deletion if we use probing (why?)
  › Need to mark array slots as deleted after Delete
  › consequently, deleting doesn’t make the table any less full than it was before the delete

• If table gets too full ($\lambda \approx 1$) or if many deletions have occurred, running time gets too long and Inserts may fail
Rehashing

- Build a bigger hash table of approximately twice the size when $\lambda$ exceeds a particular value
  - Go through old hash table, ignoring items marked deleted
  - Recompute hash value for each non-deleted key and put the item in new position in new table
  - Cannot just copy data from old table because the bigger table has a new hash function
- Running time is $O(N)$ but happens very infrequently
  - Not good for real-time safety critical applications
Rehashing Example

- Open hashing – $h_1(x) = x \mod 5$ rehashes to $h_2(x) = x \mod 11$.

\[ \lambda = 1 \]

\begin{align*}
0 & \quad 1 & \quad 2 & \quad 3 & \quad 4 \\
25 & \quad 37 & \quad 83 & \quad 52 & \quad 98
\end{align*}

\[ \lambda = \frac{5}{11} \]

\begin{align*}
0 & \quad 1 & \quad 2 & \quad 3 & \quad 4 & \quad 5 & \quad 6 & \quad 7 & \quad 8 & \quad 9 & \quad 10 \\
25 & \quad 37 & \quad 83 & \quad 52 & \quad 98
\end{align*}
Rehashing Picture

- Starting with table of size 2, double when load factor > 1.
Amortized Analysis of Rehashing

- Cost of inserting n keys is < 3n
- $2^k + 1 \leq n \leq 2^{k+1}$
  - Hashes = n
  - Rehashes = $2 + 2^2 + \ldots + 2^k = 2^{k+1} - 2$
  - Total = $n + 2^{k+1} - 2 < 3n$
- Example
  - n = 33, Total = $33 + 64 - 2 = 95 < 99$
Caveats

• Hash functions are very often the cause of performance bugs.
• Hash functions often make the code not portable.
• Sometime a poor HF distribution-wise is faster overall.
• Always check where the time goes