Hashing

The need for speed

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

Readings and References

• Reading
  › Chapter 5, *Data Structures and Algorithm Analysis in C*, Weiss

• Other References
  › Hashing, *Introduction to Algorithms*, Cormen, Leiserson and Rivest
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 $U=\{0,1,…,m-1\}$
  - $m$ 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

<table>
<thead>
<tr>
<th>U (universe of keys)</th>
<th>K (Actual keys)</th>
<th>data key</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8 9</td>
<td>0 2 3 4 5 6 8 9</td>
<td>table</td>
</tr>
</tbody>
</table>

Direct Address Implementation

Delete(Table $t$, ElementType $x$)

$T[\text{key}[x]] = \text{NULL}$

Insert(Table $t$, ElementType $x$)

$T[\text{key}[x]] = x$

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

![Mapping the Keys Diagram]

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.
  - $A[0] = \{"CHEM 110", Size 89\}$
- 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*) (i.e., map from $U$ to index)
  - Then use this value to index into an array
  - $\text{Hash}("CSE 373") = 157, \text{Hash}("CSE 143") = 101$
- Output of the hash function
  - must always be less than size of array
  - must 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 key 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
    - $\text{hash}(s) = \lfloor s \cdot m \rfloor$
    - where $m$ is the size of the table

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 one-to-one (not necessarily onto)

integer key modulo table size

- One solution for a less constrained key set
  - modular arithmetic
    - $a \mod \text{size}$
    - remainder when "a" is divided by "size"
    - in C 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

Hash function: mod

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

• 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

Keys as Natural Numbers

• Most hash functions assume that the universe of keys is the natural numbers $N=\{0,1,…\}$
• 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

Hash Function: add chars

• If keys are strings can get an integer by adding up ASCII values of characters in key
  \[
  \text{hashValue} = 0; \\
  \text{while (*key != ‘\0’)} \\
  \quad \text{hashValue} += *\text{key}++; \\
  \]
  \[
  \begin{array}{c|c|c|c|c|c|c|c|c}
    \text{character} & C & S & R & 3 & 7 & 3 & <\0> \\
    \text{ASCII value} & 67 & 83 & 69 & 32 & 51 & 51 & 0 \\
  \end{array}
  \]
• We are converting a very large number ($c_0c_1c_2c_3c_4$) to a relatively small number ($c_0+c_1+c_2+c_3+c_4$)
Hash must cover the whole 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*127 = 1016
- Need to distribute keys over the entire table or the extra space is wasted

Issues with hash add char

- 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

Hash function: chars as digits

- Suppose keys can use any of 26 characters plus blank (27 characters numbered 0 to 26)
  - these are digits in a base 27 representation of a number
  - can use 32 instead of 27 and shift left by 5 bits for fast multiplication, ie, consider the number to be a base 32 value
- A key conversion function for short strings
  - “abc” = 1*32^2 + 2*32^1 + 3 =1091
  - “bca” = 2*32^2 + 3*32^1 + 1 =2243
  - “cab” = 3*32^2 + 1*32^1 + 2 =6342

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 Separate Chaining

- Each hash table cell holds pointer to linked list of records with same hash value (i, j, k in figure)
- 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 = 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.

- \( h_1(X) = (\text{Hash}(X) + F(i)) \mod \text{TableSize} \)
  - 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 \( h(K), h(K) + 1, h(K) + 2, \ldots \) 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 phenomenon

- 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

- No collision
- Collision in small cluster
- Collision in large cluster

Quadratic Probing

- When searching for $x$, check locations $h_1(x)$, $h_1(x) + i^2$, $h_1(x) + i^3$, ... 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 $h_1(x)$, $h_1(x) + h_2(x)$, $h_1(x) + 2h_2(x)$, ... 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

- When searching for $z$, check locations $h_1(z)$, $h_1(z) + h_2(z)$, $h_1(z) + 2h_2(z)$, $h_1(z) + 3h_2(z)$, ... until either
  - $z$ is found; or
  - we find an empty location ($z$ not present)
**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
- Double hashing is space efficient, fast (get initial hash and increment at the same time), needs careful implementation
- For average cost $t$
  - Max load for Linear Probe is $1 - \frac{1}{\sqrt{t}}$
  - Max load for Double Hashing is $1 - \frac{1}{t}$

**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 size $2*TableSize$) 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

**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
Positional Notation

- Each column in a number represents an additional power of the base number
- in base ten
  - \(1 = 1 \times 10^0, \ 30 = 3 \times 10^1, \ 200 = 2 \times 10^2\)
- in base sixteen
  - \(1 = 1 \times 16^0, \ 30 = 3 \times 16^1, \ 200 = 2 \times 16^2\)
  - we use A,B,C,D,E,F to represent the numbers between 9_{16} and 10_{16}

Positional Notation

- Each column in a number represents an additional power of the base number
- in base ten
  - \(1 = 1 \times 10^0, \ 30 = 3 \times 10^1, \ 200 = 2 \times 10^2\)
- in base sixteen
  - \(1 = 1 \times 16^0, \ 30 = 3 \times 16^1, \ 200 = 2 \times 16^2\)
  - we use A,B,C,D,E,F to represent the numbers between 9_{16} and 10_{16}

Binary, Hex, and Decimal

<table>
<thead>
<tr>
<th>N</th>
<th>N</th>
<th>N</th>
<th>N</th>
<th>N</th>
<th>N</th>
<th>Hex₁₆</th>
<th>Decimal₁₀</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>128</td>
<td>64</td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>255</td>
<td>127</td>
<td>63</td>
<td>31</td>
<td>15</td>
<td>7</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>FF</td>
<td>255</td>
</tr>
<tr>
<td>11</td>
<td>1001</td>
<td>1010</td>
<td>1111</td>
<td>10000</td>
<td>11111</td>
<td>11111</td>
<td>11111</td>
</tr>
<tr>
<td>1111</td>
<td>1010</td>
<td>1111</td>
<td>10000</td>
<td>11111</td>
<td>11111</td>
<td>11111</td>
<td>10</td>
</tr>
<tr>
<td>1111</td>
<td>1010</td>
<td>1111</td>
<td>10000</td>
<td>11111</td>
<td>11111</td>
<td>11111</td>
<td>15</td>
</tr>
<tr>
<td>1111</td>
<td>1010</td>
<td>1111</td>
<td>10000</td>
<td>11111</td>
<td>11111</td>
<td>11111</td>
<td>16</td>
</tr>
<tr>
<td>1111</td>
<td>1010</td>
<td>1111</td>
<td>10000</td>
<td>11111</td>
<td>11111</td>
<td>11111</td>
<td>31</td>
</tr>
<tr>
<td>1111</td>
<td>1010</td>
<td>1111</td>
<td>10000</td>
<td>11111</td>
<td>11111</td>
<td>11111</td>
<td>127</td>
</tr>
<tr>
<td>1111</td>
<td>1010</td>
<td>1111</td>
<td>10000</td>
<td>11111</td>
<td>11111</td>
<td>11111</td>
<td>255</td>
</tr>
</tbody>
</table>

Appendix
## Binary, Hex, and Decimal

<table>
<thead>
<tr>
<th>Binary 2</th>
<th>Hex</th>
<th>Binary 10</th>
<th>Hex 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>1010</td>
<td>A</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>1011</td>
<td>B</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>1100</td>
<td>C</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>1101</td>
<td>D</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>1110</td>
<td>E</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>1111</td>
<td>F</td>
<td>15</td>
<td>15</td>
</tr>
</tbody>
</table>

22-Apr-02 CSE 373 - Data Structures - 10 - Hashing