Hash Tables II

Lecture 25

Prof. Eric Polizzi
Summary previous lecture- Hash Tables

- **Motivation**: optimal insertion and search O(1)
- Array-based implementation
- **Hashing**: converts a 'number key' (integer, String, etc.) that belongs to a large range into a much smaller index array number. It is a mapping.
- We then introduce the hash function: `int hash(key)`
  - Simple approach: `smallNumber=largeNumber%smallRange`
  - `hash` should be easy to calculate and provide evenly distributed indexes
  - It is however not possible to avoid collisions
- There exist two main approaches to deal with collisions
  - Open addressing
    - Use x2 size array
    - If a cell is already occupied, one must find another location
    - 3 options: Linear probing, Quadratic Probing, Double hashing
  - Separate Chaining
Summary previous lecture- Linear Probing

- **Methods:**
  - Insert, Search, Delete
  - Keep incrementing the probe (originally at hash value) linearly (by 1 cell) until empty cell or key item is found

- **Problems:**
  - Clustering can degrade performances
  - Expanding the array is also problematic (cannot use a direct copy), it involves a time consuming rehashing process
  - One then must design a hash table (and hash function) such that the array never becomes more than half or at most two-third full
Open Addressing: Quadratic Probing

- Loadfactor = number of Items/arraySize

- Example: 10,000 size array with 6667 items has a load factor of 2/3

- Problem: Clusters can form even if load factor is not high

- Quadratic probing aims at probing more widely separated cells instead of those adjacent to the primary hash site

- In comparison to linear probing:
  - It represents an attempt to keep clusters from forming
  - It can tolerate a bit higher load factors before performances start degrading
    - Linear is: hash, hash+1, hash+2, hash+3, etc.
    - **Quadratic is:** hash, hash+1², hash+2², hash+3², etc.

- **Important:** It is good practice to always make the array size a prime number to avoid endless sequence of numbers and get a chance to visit all cells
Open Addressing: Quadratic Probing

- Examples with search

- Todo: Java applet hashDouble.html – quad option

- Clustering may still appear. All the keys follow the same sequence before to find a vacant cell (1, 4, 9, 16, and so on)

- A better strategy exists: double hashing
Open Addressing: Double Hashing

- **Idea:** Reduce the chances of clustering by generating a probe sequences that depend on the key instead of being the same for every key (also reduce the length of the probing)

- **Solution:** hash a second time using a different hash and use the result as the step size.

- Different keys may hash to the same index but they will (most likely) generate different constant step sizes (used for probing).

- This second hash function
  - Must not be the same than the first one
  - Must not return 0

- It happens that solutions of this form works quite well:
  - stepSize= constant – (key% constant) where constant<arraySize and prime

- Example: stepSize=5-(key%5) → stepSize in the range between 1 and 5
Open Addressing: Double Hashing

- Examples with search

- Todo: Java applet hashDouble.html
Open Addressing: Double Hashing

```java
public int hash1(int key){
    return key%size;
}

public int hash2(int key){
    return 5-key%5;
}

public void insert(DataItem item){
    int h = hash1(item.getKey()); // key
    int s = hash2(item.getKey()); // step
    while (table[h]!=null && table[h].getKey()!=-1) { // until empty spot or -1
        h=(h+s)%size; // if occupied, increment by s
    }
    table[h]=item; // enter item into table
}

public DataItem find(int key){
    int h = hash1(key); // compute hash of item
    int s = hash2(key); // compute step
    while (table[h]!=null && table[h].getKey()!=key){ // find matching item or null
        h=(h+s)%size; // if no match, increment by s
    }
    return table[h]; // return item or null
}
```
### Open Addressing: Double Hashing

#### Example (p550): Size hash table 23, step size 1-5

<table>
<thead>
<tr>
<th>Item Number</th>
<th>Key</th>
<th>Hash Value</th>
<th>Step Size</th>
<th>Cells in Probe Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>38</td>
<td>15</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>37</td>
<td>14</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>16</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>20</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>11</td>
<td>11</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>24</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>16</td>
<td>16</td>
<td>4</td>
<td>20 1 5 9</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>10</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>31</td>
<td>8</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>18</td>
<td>18</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>12</td>
<td>12</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>30</td>
<td>7</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>5 9 13</td>
</tr>
<tr>
<td>17</td>
<td>19</td>
<td>19</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>36</td>
<td>13</td>
<td>4</td>
<td>17</td>
</tr>
<tr>
<td>19</td>
<td>41</td>
<td>18</td>
<td>4</td>
<td>22</td>
</tr>
<tr>
<td>20</td>
<td>15</td>
<td>15</td>
<td>5</td>
<td>20 2 7 12 17 22 4</td>
</tr>
<tr>
<td>21</td>
<td>25</td>
<td>2</td>
<td>5</td>
<td>7 12 17 22 4 9 14 19 16</td>
</tr>
</tbody>
</table>

**Final array**: 1 24 3 15 5 25 30 31 16 10 11 12 1 37 38 16 36 18 19 20 41

**Anomaly**: Hash table too full
**Idea:** Each index of the hash table is associated with a linked-list
- We handle collision by storing the collided records in the linked list.

- Conceptually easier than open addressing, coding is a bit more involved
Separate Chaining

- **Todo: Java applet hashChain.html**

- The load factor can be equal to 1

- **Problem:** if there are many items in the list (M items), access time is \( O(M) \)

- As a result, we do not want the list to become too full!

- Complete code using sorted list p555 textbook (sorted list can get the time of unsuccessful search and deletion in half, insertion however takes more time)

```java
class HashTable{
    private SortedList table;
    private int size;

    public HashTable(int size){
        this.size=size;
        table = new SortedList[size];
        for(int i=0; i<size; i++)
            table[i]=new SortedList();
    }
}
```
Hash functions

- **Questions:**
  - How to make a good hash function?
  - How can we hash Strings?
- The major advantage of hash table is its speed, so a hash function must be computed quickly
- **Goal:** key values (that may be random or more or less order) must be distributed randomly across all the indexes of the hash table
- If your keys are random
  - It is good to simply choose: index=key%arraySize; with arraySize a prime number
- If your keys are not random
  - Examples: 06-12-1973-33 (for birthdate month/day/year/county code); part numbers; hybrid number-Strings; etc.
  - You need to examine carefully your keys and tailor your hash function to remove any irregularity in the distribution of keys
**Hash functions - Hashing Strings**

- **Idea:** every character in a word contributes in a unique way to the final number

- Analogy: 7,546 means
  - $7 \times 1000 + 5 \times 100 + 40 \times 10 + 6$
  - $7 \times 10^3 + 5 \times 10^2 + 4 \times 10^1 + 6 \times 10^0$

- We can decompose a word into its letters, and convert the letters into their numerical equivalent
  - We consider 27 letters (lower cases from a to z, including the blank space) → arithmetic with base 27
  - A key for the word *cats* is then: $3 \times 27^3 + 1 \times 27^2 + 20 \times 27^1 + 19 \times 27^0$
  - The total is 60,337
  - Remark: the word *zzzzzzzzzz* generates $7,000,000,000,000$

- Two problems:
  - How to calculate efficiently powers (direct method is inefficient)?
  - The int type can't handle String longer than about 7 letters (key is a big number). Long type will also reach its limitations
Solutions for two problems:

How to calculate efficiently powers (direct method is inefficient)?

Solution: Horner's method

\[ v_3n^3 + v_2n^2 + v_1n + v_0n^0 \] equivalent to \((v_3n+v_2)n+v_1)n+v_0\)

key is a big number

Solution: apply the modulo operator \(\%\) at each step of the Horner calculation (size of the array could be the size of the dictionary*2)

class int hash(String key) {
    int h=0; // initialize hash value
    for (int j=0; j<key.length(); j++) {
        int letter=key.charAt(j)-96; // get char code from ASCII
        h=(h*27+letter)%size; // Horner+modulo
    }
    return h;
}