ECE 242
Data Structures and Algorithms

http://www.ecs.umass.edu/~polizzi/Teaching/ECE242/

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

- 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

- Todo: Java applet hashDouble.html – quad option
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

a) Successful search for 887
b) Unsuccessful search for 709
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 1 6</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*
Separate Chaining

- **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 the 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\n
- 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)

```java
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;
}
```
Complement: ASCII table

<table>
<thead>
<tr>
<th>ASCII control characters</th>
<th>ASCII printable characters</th>
<th>Extended ASCII characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEC HEX Simbolo ASCII</td>
<td>DEC HEX Simbolo</td>
<td>DEC HEX Simbolo</td>
</tr>
<tr>
<td>00 0h NULL   (carácter nulo)</td>
<td>32 20h espacio</td>
<td>128 80h Ç</td>
</tr>
<tr>
<td>01 01h EOF</td>
<td>33 21h !</td>
<td>129 81h Å</td>
</tr>
<tr>
<td>02 02h STX</td>
<td>34 22h &quot;</td>
<td>130 82h õ</td>
</tr>
<tr>
<td>03 03h ETX</td>
<td>35 23h #</td>
<td>131 83h å</td>
</tr>
<tr>
<td>04 04h EOT</td>
<td>36 24h $</td>
<td>132 84h à</td>
</tr>
<tr>
<td>05 05h ENQ</td>
<td>37 25h %</td>
<td>133 85h ã</td>
</tr>
<tr>
<td>06 06h ACK</td>
<td>38 26h &amp;</td>
<td>134 86h á</td>
</tr>
<tr>
<td>07 07h BEL</td>
<td>39 27h '</td>
<td>135 87h ç</td>
</tr>
<tr>
<td>08 08h BS</td>
<td>40 28h (</td>
<td>136 88h ñ</td>
</tr>
<tr>
<td>09 09h HT</td>
<td>41 29h )</td>
<td>137 89h ø</td>
</tr>
<tr>
<td>10 0Ah LF</td>
<td>42 2Ah ‘</td>
<td>138 8Ah Ô</td>
</tr>
<tr>
<td>11 0Bh VT</td>
<td>43 2Ch ,</td>
<td>139 8Bh ï</td>
</tr>
<tr>
<td>12 0Ch FF</td>
<td>44 2Dh .</td>
<td>140 8Ch ũ</td>
</tr>
<tr>
<td>13 0Dh CR</td>
<td>45 2 Eh /</td>
<td>141 8Ch Ý</td>
</tr>
<tr>
<td>14 0 Eh SO</td>
<td>46 2Fh</td>
<td></td>
</tr>
<tr>
<td>15 0 Fh SI</td>
<td>47 2 Gh /</td>
<td>143 8Ch Á</td>
</tr>
<tr>
<td>16 10h DLE</td>
<td>48 30h 0</td>
<td>144 8Ch È</td>
</tr>
<tr>
<td>17 11h DC1</td>
<td>49 31h 1</td>
<td>145 8Ch Ê</td>
</tr>
<tr>
<td>18 12h DC2</td>
<td>50 32h 2</td>
<td>146 8Ch Ë</td>
</tr>
<tr>
<td>19 13h DC3</td>
<td>51 33h 3</td>
<td>147 8Ch Ô</td>
</tr>
<tr>
<td>20 14h DC4</td>
<td>52 34h 4</td>
<td>148 8Ch Õ</td>
</tr>
<tr>
<td>21 15h NAK</td>
<td>53 35h 5</td>
<td>149 8Ch Õ</td>
</tr>
<tr>
<td>22 16h SYH</td>
<td>54 36h 6</td>
<td>150 8Ch Õ</td>
</tr>
<tr>
<td>23 17h ETB</td>
<td>55 37h 7</td>
<td>151 8Ch Õ</td>
</tr>
<tr>
<td>24 18h CAM</td>
<td>56 38h 8</td>
<td>152 8Ch Õ</td>
</tr>
<tr>
<td>25 19h EM</td>
<td>57 39h 9</td>
<td>153 8Ch Õ</td>
</tr>
<tr>
<td>26 1Ah SUB</td>
<td>58 3Ah 0</td>
<td>154 8Ch Õ</td>
</tr>
<tr>
<td>27 1Bh ESC</td>
<td>59 3Ah 1</td>
<td>155 8Ch Õ</td>
</tr>
<tr>
<td>28 1Ch FS</td>
<td>60 3Ah 2</td>
<td>156 8Ch Õ</td>
</tr>
<tr>
<td>29 1Dh GS</td>
<td>61 3Ah 3</td>
<td>157 8Ch Õ</td>
</tr>
<tr>
<td>30 1 Eh RS</td>
<td>62 3Ah 4</td>
<td>158 8Ch Õ</td>
</tr>
<tr>
<td>31 1Fh US</td>
<td>63 3Fh ?</td>
<td>159 8Fn f</td>
</tr>
</tbody>
</table>

*thoASCIIcode.com.ar*