Hash Tables

CS206
March 9
Lab from Friday

- What is the time complexity, big O, $O(\cdot)$ of these 2 functions?

```java
long complex(int[] data) {
    long res = 0;
    for (int i = 0; i < data.length / 2; i++) {
        for (int j = data.length / 2; j < data.length; j++) {
            res++;
        }  \(\text{O(n) loop}\)
    }
    return res;
}

long complex2(int[] data) {
    long res = 0;
    for (int i = 0; i < data.length; i++) {
        for (int j = 1; j < data.length; j=\text{j*2}) {
            res++;
        }  \(\text{O(lg n) loop}\)
    }
    return res;
}
```

$O(n^2)$  $O(n \text{lg}_2 n)$
Hash Tables

• A hash table is a form of a map that has better time complexity

• A hash table consists of
  • an array of size $N$
    ▫ an associated hash function $h$ that maps keys to integers in $[0, N-1]$
    ▫ A “collision” handling scheme

• $h(x) = x \% N$ is such a function for integers
  • item $(k, v)$ is stored at index $h(k)$

• Collision Handling
  • A “collision” occurs when two different keys hash to the same value
Hash Functions

• The goal of a hash function is to disperse the keys

• A hash function is usually specified as the composition of two functions:
  • hash code: key —> integers
  • compression: integers —> [0, N-1]
    • where the backing array is of size N
Hash Codes

- Polynomial accumulation: partition bits of key into a sequence of components of fixed length $a_0a_1...a_{n-1}$
- For instance, if the key is a string, then a’s are just the characters
  - convert characters to numbers such that a:97, ... z:122 (The ASCII table)
- Evaluate the polynomial $p(z) = a_0 + a_1z + a_2z^2 + ... + a_{n-1}z^{n-1}$
- Why not convert a=>0?
<table>
<thead>
<tr>
<th>Doc</th>
<th>Hx</th>
<th>Oct</th>
<th>HTML</th>
<th>Chr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$\text{null}$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>001</td>
<td>1</td>
<td>$\text{start of heading}$</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>002</td>
<td>2</td>
<td>$\text{start of text}$</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>003</td>
<td>3</td>
<td>$\text{end of text}$</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>004</td>
<td>4</td>
<td>$\text{end of transmission}$</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>005</td>
<td>5</td>
<td>$\text{ enquiry}$</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>006</td>
<td>6</td>
<td>$\text{acknowledgment}$</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>007</td>
<td>7</td>
<td>$\text{bell}$</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>010</td>
<td>8</td>
<td>$\text{backspace}$</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>011</td>
<td>9</td>
<td>$\text{horizontal tab}$</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>012</td>
<td>1</td>
<td>$\text{NL line feed, new line}$</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>013</td>
<td>1</td>
<td>$\text{vertical tab}$</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>014</td>
<td>1</td>
<td>$\text{NP form feed, new page}$</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>015</td>
<td>1</td>
<td>$\text{carriage return}$</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>016</td>
<td>1</td>
<td>$\text{shift out}$</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>017</td>
<td>1</td>
<td>$\text{shift in}$</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>020</td>
<td>1</td>
<td>$\text{data link escape}$</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>021</td>
<td>1</td>
<td>$\text{device control 1}$</td>
</tr>
<tr>
<td>18</td>
<td>1</td>
<td>022</td>
<td>1</td>
<td>$\text{device control 2}$</td>
</tr>
<tr>
<td>19</td>
<td>1</td>
<td>023</td>
<td>1</td>
<td>$\text{device control 3}$</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>024</td>
<td>1</td>
<td>$\text{device control 4}$</td>
</tr>
<tr>
<td>21</td>
<td>1</td>
<td>025</td>
<td>1</td>
<td>$\text{negative acknowledge}$</td>
</tr>
<tr>
<td>22</td>
<td>1</td>
<td>026</td>
<td>1</td>
<td>$\text{synchrornous idle}$</td>
</tr>
<tr>
<td>23</td>
<td>1</td>
<td>027</td>
<td>1</td>
<td>$\text{end of trans. block}$</td>
</tr>
<tr>
<td>24</td>
<td>1</td>
<td>030</td>
<td>1</td>
<td>$\text{cancel}$</td>
</tr>
<tr>
<td>25</td>
<td>1</td>
<td>031</td>
<td>1</td>
<td>$\text{end of medium}$</td>
</tr>
<tr>
<td>26</td>
<td>1</td>
<td>032</td>
<td>2</td>
<td>$\text{substitute}$</td>
</tr>
<tr>
<td>27</td>
<td>1</td>
<td>033</td>
<td>2</td>
<td>$\text{escape}$</td>
</tr>
<tr>
<td>28</td>
<td>1</td>
<td>034</td>
<td>3</td>
<td>$\text{file separator}$</td>
</tr>
<tr>
<td>29</td>
<td>1</td>
<td>035</td>
<td>5</td>
<td>$\text{group separator}$</td>
</tr>
<tr>
<td>30</td>
<td>1</td>
<td>036</td>
<td>5</td>
<td>$\text{record separator}$</td>
</tr>
<tr>
<td>31</td>
<td>1</td>
<td>037</td>
<td>5</td>
<td>$\text{unit separator}$</td>
</tr>
</tbody>
</table>

**ASCII**

American Standard Code for Information Interchange.

Source: [www.LookupTable.com](http://www.LookupTable.com)
Polynomial accumulation on Strings

```
static int POLY_MULT=33;
public int stringHasher(String ss) {
    BigInteger accumul = new BigInteger("0");
    for (int i=0; i<ss.length(); i++) {
        BigInteger bb = BigInteger.valueOf(POLY_MULT).pow(i).multiply(BigInteger.valueOf((int)ss.charAt(i)));
        accumul = accumul.add(bb);
    }
    accumul = accumul.mod(BigInteger.valueOf(backingArray.length));
    return accumul.intValue();
}
```

Recommended by textbook

Handles integers of unbounded size

$33^{15}=59938945498865420543457$

The ASCII value

Array storing the hashtable
Collisions

drawing 500 unique words from Oliver Twist and assuming a hashtable size of 1009, get these collisions

16 probable child when
42 fagins xxix importance that xv administering
104 stage pledge near
132 surgeon can night
271 things fang birth
341 alone sequel life
415 maylie check circumstances
418 mentioning containing growth
625 meet she first
732 there affording encounters
749 possible out acquainted
761 never xviii after goaded where
833 marks jew gentleman
985 adventures inseparable experience
Collisions

• Handling of collisions is one of the most important topics for hashtables
  • Approach 1:
    • Whenever you have a collision “Rehash”
      • make the table bigger
      • O(n) time so want to avoid
  • Approach 2
    • Separate Chaining
  • Approach 3
    • Probing
Separate Chaining

- Idea: each spot in hashtable holds an array list of key value pairs when the key maps to that hash value.
- Replace the item if the key is the same.
- Otherwise, add to list.
- Generally do not want more than about number of objects as size of table.
- Chains can get long.
A hash table is shown with the following information:

- HT_SIZE = 1009
- The table is being used to store unique words drawn from "Oliver Twist".
- Unique count at top of table:
  - 278 words
  - 473 words
  - 1550 words
  - 2510 words

The table contains the following entries:

<table>
<thead>
<tr>
<th>Hash Value</th>
<th>Word Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>762</td>
</tr>
<tr>
<td>1</td>
<td>217</td>
</tr>
<tr>
<td>2</td>
<td>29</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hash Value</th>
<th>Word Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>622</td>
</tr>
<tr>
<td>1</td>
<td>308</td>
</tr>
<tr>
<td>2</td>
<td>73</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hash Value</th>
<th>Word Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>210</td>
</tr>
<tr>
<td>1</td>
<td>342</td>
</tr>
<tr>
<td>2</td>
<td>252</td>
</tr>
<tr>
<td>3</td>
<td>136</td>
</tr>
<tr>
<td>4</td>
<td>55</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hash Value</th>
<th>Word Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>87</td>
</tr>
<tr>
<td>1</td>
<td>198</td>
</tr>
<tr>
<td>2</td>
<td>268</td>
</tr>
<tr>
<td>3</td>
<td>208</td>
</tr>
<tr>
<td>4</td>
<td>140</td>
</tr>
<tr>
<td>5</td>
<td>70</td>
</tr>
<tr>
<td>6</td>
<td>26</td>
</tr>
<tr>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
</tr>
</tbody>
</table>
public class SepChainHT<K, V> implements Map206Interface<K, V> {

    private Map206<K, V>[] backingArray;

    private int count;

    public SepChainHT(int size) {
        count = 0;
        backingArray = (Map206<K, V>[][]) new Map206[size];
    }

    private int h(K k) {
        return stringHasher(k.toString()) % backingArray.length;
    }

    // stringHasher – as on slide 5
In class exercise

- Show the final contents of the hashtable using separate chaining assuming. I.e. show the contents of all chains
  - table size is 7
  - h(t) = t % 7
  - Data: <0,a> <32,b> <39,c> <12,d> <14,e> <35,f> <27,g> <13,h> <15,i> <5,j> <12,k> <13,l> <4,m> <0,n> <35,o>,<17,o>,<3,o>

- For a separate chaining hashtable that uses Map206 for its chains (as in the previous slide) write:
  - boolean containsKey(K key)
  - V get(K key)
public void put(K key, V value) {
    int loc = h(key);
    if (backingArray[loc] == null) {
        backingArray[loc] = new Map206<>();
    }
    if (!backingArray[loc].containsKey(key)) {
        count++;
    }
    backingArray[loc].put(key, value);
}

public V get(K key) {
    int loc = h(key);
    if (backingArray[loc]==null) {
        return null;
    }
    return backingArray[loc].get(key);
}

public boolean containsKey(K key) {
    int loc = h(key);
    if (backingArray[loc] == null) {
        return false;
    }
    return backingArray[loc].containsKey(key);
}

public Set<K> keySet() {
    TreeSet<K> set = new TreeSet<>();
    for (int i = 0; i < backingArray.length; i++)
        if (backingArray[i] != null)
            set.addAll(backingArray[i].keySet());
    return set;
}
Open Addressing
Linear Probing

- Store only \(<K, V>\) at each location in array
- No awkward lists
- If key is different and location is in use then go to next spot in array
- repeat until free location found
Linear Probing Example

• Suppose
  • hashtable size is 7
  • h(t)=t%7
• add:
  • <3,A>
  • <10,B>
  • <17,C>
  • <24,Z>
  • <3,D>
  • <4,E>
Linear Probing

- Store only $<K,V>$ at each location in array
- If key is different and location is in use then go to next spot in array
  - if key is same, replace value
  - repeat until free location found
Probing Distance

• Given a hash value $h(x)$, linear probing generates $h(x)$, $h(x) + 1$, $h(x) + 2$, ...

  • Primary clustering – the bigger the cluster gets, the faster it grows

• Quadratic probing – $h(x)$, $h(x) + 1$, $h(x) + 4$, $h(x) + 9$, ...

  • Quadratic probing leads to secondary clustering, more subtle, not as dramatic, but still systematic

• Double hashing

  • Use a second hash function to determine jumps
Performance Analysis for probing

- In the worst case, searches, insertions and removals take $O(n)$ time
  - when all the keys collide
- The load factor $\alpha$ affects the performance of a hash table
  - expected number of probes for an insertion with open addressing is $\frac{1}{1 - \alpha}$
- Expected time of all operations is $O(1)$ provided $\alpha$ is not close to 1
  - NOTE: cheating here $O()$ is about true worst case
Open Addressing vs Chaining

- Probing is significantly faster in practice
- Locality of references – much faster to access a series of elements in an array than to follow the same number of pointers in a linked list
- Efficient probing requires soft/lazy deletions – tombstoning
- de-tombstoning
Lab

• Show the final contents of the hashtable using linear probing assuming
  • table size is 13
  • \( h(t) = t \mod 13 \)
  • Data: \( <0,a> <32,b> <39,c> <12,d> <14,e> <35,f> <27,g> <13,h> <15,i> <5,j> <12,k> <13,l> <4,m> <0,n> <35,o> \)

• What is the most number of steps you needed to take to find a free location?