Hash Tables
HashTables

• 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

• Hash Function
  • $h(x) = x \% N$ 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 $\rightarrow$ integers
  • compression: integers $\rightarrow [0, N-1]$
    • where the backing array is of size $N$
Char-by-char in String

- String s = "abc";
  - s.charAt(0) == 'a';
  - s.charAt(0) == 97;
    - both are correct.

- Suppose Hash func is just add ASCII values of all chars in string.

<table>
<thead>
<tr>
<th>Key</th>
<th>Char values</th>
<th>As integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>aba</td>
<td>97+98+97</td>
<td>292</td>
</tr>
<tr>
<td>baa</td>
<td>98+97+97</td>
<td>292</td>
</tr>
<tr>
<td>aab</td>
<td>97+97+98</td>
<td>292</td>
</tr>
<tr>
<td>Dec</td>
<td>Hex</td>
<td>Oct</td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>0</td>
<td>000</td>
<td>000</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>001</td>
</tr>
<tr>
<td>2</td>
<td>002</td>
<td>002</td>
</tr>
<tr>
<td>3</td>
<td>003</td>
<td>003</td>
</tr>
<tr>
<td>4</td>
<td>004</td>
<td>004</td>
</tr>
<tr>
<td>5</td>
<td>005</td>
<td>005</td>
</tr>
<tr>
<td>6</td>
<td>006</td>
<td>006</td>
</tr>
<tr>
<td>7</td>
<td>007</td>
<td>007</td>
</tr>
<tr>
<td>8</td>
<td>010</td>
<td>010</td>
</tr>
<tr>
<td>9</td>
<td>011</td>
<td>011</td>
</tr>
<tr>
<td>A</td>
<td>012</td>
<td>012</td>
</tr>
<tr>
<td>B</td>
<td>013</td>
<td>013</td>
</tr>
<tr>
<td>C</td>
<td>014</td>
<td>014</td>
</tr>
<tr>
<td>D</td>
<td>015</td>
<td>015</td>
</tr>
<tr>
<td>E</td>
<td>016</td>
<td>016</td>
</tr>
<tr>
<td>F</td>
<td>017</td>
<td>017</td>
</tr>
<tr>
<td>10</td>
<td>020</td>
<td>020</td>
</tr>
<tr>
<td>11</td>
<td>021</td>
<td>021</td>
</tr>
<tr>
<td>12</td>
<td>022</td>
<td>022</td>
</tr>
<tr>
<td>13</td>
<td>023</td>
<td>023</td>
</tr>
<tr>
<td>14</td>
<td>024</td>
<td>024</td>
</tr>
<tr>
<td>21</td>
<td>025</td>
<td>025</td>
</tr>
<tr>
<td>22</td>
<td>026</td>
<td>026</td>
</tr>
<tr>
<td>23</td>
<td>027</td>
<td>027</td>
</tr>
<tr>
<td>24</td>
<td>030</td>
<td>030</td>
</tr>
<tr>
<td>25</td>
<td>031</td>
<td>031</td>
</tr>
<tr>
<td>26</td>
<td>032</td>
<td>032</td>
</tr>
<tr>
<td>27</td>
<td>033</td>
<td>033</td>
</tr>
<tr>
<td>28</td>
<td>034</td>
<td>034</td>
</tr>
<tr>
<td>29</td>
<td>035</td>
<td>035</td>
</tr>
<tr>
<td>30</td>
<td>036</td>
<td>036</td>
</tr>
<tr>
<td>31</td>
<td>037</td>
<td>037</td>
</tr>
</tbody>
</table>

Source: www.LookupTables.com
Hash Codes

- Why use ASCII values rather than $a==0, b==1, ...$
Horner’s method: Convert any object to integer

```java
public BigInteger objectHasher(Object ob) {
    return stringHasher(ob.toString());
}

public BigInteger stringHasher(String ss) {
    BigInteger mul = BigInteger.valueOf(23);
    BigInteger ll = BigInteger.valueOf(0);
    for (int i=0; i<ss.length(); i++) {
        ll = ll.multiply(mul);
        ll = ll.add(BigInteger.valueOf(ss.charAt(i)));
    }
    return ll;
}
```

Start with an object, then just call its `toString` method.

Almost any prime number can be used.

Handles really large numbers.

33^{15} = 59938945498865420543457
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 a array list of key value pairs when the key maps to that hashvalue.
• 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
Hash tables get crowded, chains get long

HT_SIZE=1009

Using unique words drawn from “Oliver Twist”. Unique count at top of table

<table>
<thead>
<tr>
<th>278</th>
<th>473</th>
<th>1550</th>
<th>2510</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>
Separate Chaining Example

- Suppose
  - hashtable size is 3
  - hashtable has lower case character keys and strings for values
  - \( h(x) = (x - 97) \mod 3 \)

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
<th>Hash Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;a, &quot;Neville&quot;&gt;</td>
<td>(97-97)%3</td>
<td></td>
</tr>
<tr>
<td>&lt;b, &quot;Ray&quot;&gt;</td>
<td>(98-97)%3</td>
<td></td>
</tr>
<tr>
<td>&lt;f, &quot;Hurum&quot;&gt;</td>
<td>(102-97)%3</td>
<td></td>
</tr>
<tr>
<td>&lt;g, &quot;Quinn&quot;&gt;</td>
<td>(103-97)%3</td>
<td></td>
</tr>
<tr>
<td>&lt;m, &quot;Amina&quot;&gt;</td>
<td>(109-97)%3</td>
<td></td>
</tr>
<tr>
<td>&lt;a, &quot;Juno&quot;&gt;</td>
<td>(97-97)%3</td>
<td></td>
</tr>
</tbody>
</table>
public class SepChainHT<K, V> implements Map151Interface<K, V> {
    private Map151Impl<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 objectHasher(k);
    }
}
**Separate Chaining Code**

```java
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;
}
```
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 Map151 for its chains (as in the previous slide) write:
  • boolean containsKey(K key)
  • V get(K key)
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>\>
    • \(<17,C>\>
    • \(<24,Z>\>
    • \(<4,E>\>
    • \(<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 list
• Efficient probing requires soft/lazy deletions – tombstoning
• de-tombstoning