Dictionaries
Hash Tables
public class OutCLGen<R, S> {
    /**
     * The inner class, Generically
     */
    private class InnCl<Y, Z> {
        private Y value1; // a value
        private Z value2; // another value

        public InnCl(Y v1, Z v2) {
            this.value1 = v1;
            this.value2 = v2;
        }

        public String toString() {
            return value1 + " " + value2;
        }
    }

    public void worker(R rValue, S sValue) {
        InnCl<String, String> icl1 = new InnCl<>("Alice", "Bob");
        InnCl<R, S> icl2 = new InnCl<>(rValue, sValue);
        icl1.value1 = 3;
        System.out.println(icl1 + "\n" + icl2);
    }

    public static void main(String[] args) {
        OutCLGen<Integer, String> ocl = new OutCLGen<>();
        ocl.worker(42, "Carol");
    }
}

Inner class has different generic parameters than its surrounding class. Realistically, they will almost always be the same, but this allows for difference.
Dictionary (Map)

- A searchable collection of key-value pairs
- A lot of this course will be involve key-value pairs
  - A lot of life is about key-value pairs
    - SSN, tax history
    - BMID no, student record
    - ....
- Multiple entries with the same key are not allowed
- AKA associative array (perl)
What do you do with dictionaries (Physical)

• Look up based on a key item (word)
  • to get defintion
• Add items (word and definition)
• Remove Items (word)
• Others??
Map Interface

- https://docs.oracle.com/javase/7/docs/api/java/util/Map.html

```java
public interface Map<K, V> {
    public void put(K key, V val);
    public V get(K key);
    public boolean containsKey(K key);
    public int size();
    public ListImpl<K> keySet();
}
```

Overwrite if key is already in map

Alternately return an ArrayList or a Set. The java definition returns a Set.
public class Map151Impl<K, V> implements Map151<K, V>{
    private ArrayList<Pair<K, V>> underlying = new ArrayList<>();

    private class Pair<L, W> {
        public L ky;
        public W vl;

        Pair(L key, W value) {
            ky = key;
            vl = value;
        }
    }
}

Uses Java standard class rather than List151Impl
public boolean containsKey(K key) {
    return null != getKV(key);
}

private Pair<K,V> getKV(K ky) {
    for (Pair<K,V> pair : underlying) {
        if (pair.key.equals(ky)) {
            return pair;
        }
    }
    return null;
}

public int size() {
    return underlying.size();
}

public List151Impl<K> keySet() {
    // Write Me
}

public void put(K key, V val) {
    Pair<K,V> pair = getKV(key);
    if (pair == null) {
        Pair<K,V> np = new Pair<>(key, val);
        underlying.add(np);
    } else {
        pair.value = val;
    }
}

public V get(K key) {
    Pair<K,V> pair = getKV(key);
    if (pair != null) {
        return pair.value;
    }
    return null;
}

In the book’s parlance, this is an unsorted, ArrayList-based dictionary.
Maps - time complexity

• Put
  • need to search the existing items (iContainsKey)
  • so time to add 1 item in O(n)
    • if you are adding n items the you are doing an O(n) process n times so your time is O(n^2)
• Get
  • similar analysis so O(n^2) to do n gets
• Conclusion: maps get slow when n gets large.
Using Maps to track stock positions

• Suppose you are an active stock trader, buying and selling all the time.
• One challenge is simply keeping track of how much you have of what
• Further suppose,
  • you have a csv file
    • STOCK ID, AMT
• Idea
  • Use ReadCSV and Map151 to track
```java
Map151Impl<String, Integer> positions;

public StockTracker() {
    positions = new Map151Impl<>();
}

public void track(String filename) {
    ReadCSV csvReader;
    try {
        csvReader = new ReadCSV(filename, 4000);
    } catch (IOException ioe) {
        System.err.println("Ending. Cannot read. "+ ioe.toString());
        return;
    }
    for (String[] datum : csvReader) {
        int newamt = Integer.parseInt(datum[1]);
        Integer oldVal = positions.get(datum[0]);
        if (oldVal != null) {
            newamt += oldVal;
        }
        positions.put(datum[0], newamt);
    }
    System.out.println(positions);
}
```

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 $[0, N-1]$
    - For example, given integer keys, $h(x) = x \% N$ is a hash function.
- pair(K,V) is stored at index $h(k)$
- pair is an inner class a la Map.
public class SimpleHT {
    private class Pair {
        // the key.  Once set it cannot be changed
        public final Integer key;
        // the value
        public String value;
        // Create a key value pair.
        Pair(Integer ky, String val) {
            key = ky;
            value = val;
        }
    }
    private Pair[] backingArray = new Pair[4];
    private int h(int k) {
        return k%4;
    }
    public void put(Integer key, String value) {
        backingArray[h(key)] = new Pair(key, value);
    }
    public String get(Integer key) {
        return backingArray[h(key)].value;
    }
}
public static void main(String[] args) {
    SimpleHT sht = new SimpleHT();
    for (int i=0; i<10; i++) {
        System.out.println("adding item with key=" + i + " value=" + String.format("%c", 'a'+i));
        sht.put(i, String.format("%c", 'a'+i));
    }
    for (int i=0; i<10; i++)
        System.out.println("getting key="+i+" value="+sht.get(i));
}

Two problems:
1. a poor hashing function.
2. Storing more than there is room for

adding item with key=0 value=a
adding item with key=1 value=b
adding item with key=2 value=c
adding item with key=3 value=d
adding item with key=4 value=e
adding item with key=5 value=f
adding item with key=6 value=g
adding item with key=7 value=h
adding item with key=8 value=i
adding item with key=9 value=j
getting key=0  value=i
getting key=1  value=j
getting key=2  value=g
getting key=3  value=h
getting key=4  value=i
getting key=5  value=j
getting key=6  value=g
getting key=7  value=h
getting key=8  value=i
getting key=9  value=j
Hash Functions

• The goal is to “disperse” the keys in an appropriately random way
• A hash function is usually specified as the composition of two functions:
  • hash code: key $\rightarrow$ integers
  • compression: integers $\rightarrow$ [0, N-1]

see SepChainHT.java
Hash Functions

• Goals:
  • Minimize collisions
  • Quickly transform key to integer
• Common Approach for non-integer keys:
  • 1. Transform key into String
  • 2. Do something character-by-character on string
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>
</tbody>
</table>
## ASCII Table

<table>
<thead>
<tr>
<th>dec</th>
<th>hex</th>
<th>oct</th>
<th>char</th>
<th>dec</th>
<th>hex</th>
<th>oct</th>
<th>char</th>
<th>dec</th>
<th>hex</th>
<th>oct</th>
<th>char</th>
<th>dec</th>
<th>hex</th>
<th>oct</th>
<th>char</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td></td>
<td>NULL</td>
<td>32</td>
<td>20</td>
<td>040</td>
<td>space</td>
<td>64</td>
<td>40</td>
<td>100</td>
<td>@</td>
<td>96</td>
<td>60</td>
<td>140</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td></td>
<td>SOH</td>
<td>33</td>
<td>21</td>
<td>041</td>
<td>A</td>
<td>65</td>
<td>41</td>
<td>101</td>
<td>B</td>
<td>97</td>
<td>61</td>
<td>141</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>002</td>
<td></td>
<td>STX</td>
<td>34</td>
<td>22</td>
<td>042</td>
<td>*</td>
<td>66</td>
<td>42</td>
<td>102</td>
<td>C</td>
<td>98</td>
<td>62</td>
<td>142 b</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>003</td>
<td></td>
<td>ETX</td>
<td>35</td>
<td>23</td>
<td>043</td>
<td>#</td>
<td>67</td>
<td>43</td>
<td>103</td>
<td>D</td>
<td>99</td>
<td>63</td>
<td>143 c</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>004</td>
<td></td>
<td>EOT</td>
<td>36</td>
<td>24</td>
<td>044</td>
<td>$</td>
<td>68</td>
<td>44</td>
<td>104</td>
<td>E</td>
<td>100</td>
<td>64</td>
<td>144 d</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>005</td>
<td></td>
<td>ENQ</td>
<td>37</td>
<td>25</td>
<td>045</td>
<td>%</td>
<td>69</td>
<td>45</td>
<td>105</td>
<td>F</td>
<td>101</td>
<td>65</td>
<td>145 e</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>006</td>
<td></td>
<td>ACK</td>
<td>38</td>
<td>26</td>
<td>046</td>
<td>&amp;</td>
<td>70</td>
<td>46</td>
<td>106</td>
<td>G</td>
<td>102</td>
<td>66</td>
<td>146 f</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>007</td>
<td></td>
<td>BEL</td>
<td>39</td>
<td>27</td>
<td>047</td>
<td>'</td>
<td>71</td>
<td>47</td>
<td>107</td>
<td>H</td>
<td>103</td>
<td>67</td>
<td>147 g</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>010</td>
<td></td>
<td>BS</td>
<td>40</td>
<td>28</td>
<td>050</td>
<td>(</td>
<td>72</td>
<td>48</td>
<td>110</td>
<td>I</td>
<td>104</td>
<td>68</td>
<td>150 h</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>011</td>
<td></td>
<td>TAB</td>
<td>41</td>
<td>29</td>
<td>051</td>
<td>)</td>
<td>73</td>
<td>49</td>
<td>111</td>
<td>J</td>
<td>105</td>
<td>69</td>
<td>151 i</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>012</td>
<td></td>
<td>LF</td>
<td>42</td>
<td>2a</td>
<td>052</td>
<td>*</td>
<td>74</td>
<td>4a</td>
<td>112</td>
<td>K</td>
<td>106</td>
<td>6a</td>
<td>152 j</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>013</td>
<td></td>
<td>VT</td>
<td>43</td>
<td>2b</td>
<td>053</td>
<td>+</td>
<td>75</td>
<td>4b</td>
<td>113</td>
<td>L</td>
<td>107</td>
<td>6b</td>
<td>153 k</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>014</td>
<td></td>
<td>FF</td>
<td>44</td>
<td>2c</td>
<td>054</td>
<td>,</td>
<td>76</td>
<td>4c</td>
<td>114</td>
<td>M</td>
<td>108</td>
<td>6c</td>
<td>154 l</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>015</td>
<td></td>
<td>CR</td>
<td>45</td>
<td>2d</td>
<td>055</td>
<td>.</td>
<td>77</td>
<td>4d</td>
<td>115</td>
<td>N</td>
<td>109</td>
<td>6d</td>
<td>155 m</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>016</td>
<td></td>
<td>SO</td>
<td>46</td>
<td>2e</td>
<td>056</td>
<td>;</td>
<td>78</td>
<td>4e</td>
<td>116</td>
<td>O</td>
<td>110</td>
<td>6e</td>
<td>156 n</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>017</td>
<td></td>
<td>SI</td>
<td>47</td>
<td>2f</td>
<td>057</td>
<td>/</td>
<td>79</td>
<td>4f</td>
<td>117</td>
<td>P</td>
<td>111</td>
<td>6f</td>
<td>157 o</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>020</td>
<td></td>
<td>DLE</td>
<td>48</td>
<td>30</td>
<td>060</td>
<td>0</td>
<td>80</td>
<td>50</td>
<td>120</td>
<td>Q</td>
<td>112</td>
<td>70</td>
<td>160 p</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>021</td>
<td></td>
<td>DC1</td>
<td>49</td>
<td>31</td>
<td>061</td>
<td>1</td>
<td>81</td>
<td>51</td>
<td>121</td>
<td>R</td>
<td>113</td>
<td>71</td>
<td>161 q</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>022</td>
<td></td>
<td>DC2</td>
<td>50</td>
<td>32</td>
<td>062</td>
<td>2</td>
<td>82</td>
<td>52</td>
<td>122</td>
<td>S</td>
<td>114</td>
<td>72</td>
<td>162 r</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>023</td>
<td></td>
<td>DC3</td>
<td>51</td>
<td>33</td>
<td>063</td>
<td>3</td>
<td>83</td>
<td>53</td>
<td>123</td>
<td>T</td>
<td>115</td>
<td>73</td>
<td>163 s</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>024</td>
<td></td>
<td>DC4</td>
<td>52</td>
<td>34</td>
<td>064</td>
<td>4</td>
<td>84</td>
<td>54</td>
<td>124</td>
<td>U</td>
<td>116</td>
<td>74</td>
<td>164 t</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>025</td>
<td></td>
<td>NAK</td>
<td>53</td>
<td>35</td>
<td>065</td>
<td>5</td>
<td>85</td>
<td>55</td>
<td>125</td>
<td>V</td>
<td>117</td>
<td>75</td>
<td>165 u</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>026</td>
<td></td>
<td>SYN</td>
<td>54</td>
<td>36</td>
<td>066</td>
<td>6</td>
<td>86</td>
<td>56</td>
<td>126</td>
<td>W</td>
<td>118</td>
<td>76</td>
<td>166 v</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>027</td>
<td></td>
<td>ETB</td>
<td>55</td>
<td>37</td>
<td>067</td>
<td>7</td>
<td>87</td>
<td>57</td>
<td>127</td>
<td>X</td>
<td>119</td>
<td>77</td>
<td>167 w</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>030</td>
<td></td>
<td>CAN</td>
<td>56</td>
<td>38</td>
<td>070</td>
<td>8</td>
<td>88</td>
<td>58</td>
<td>130</td>
<td>Y</td>
<td>120</td>
<td>78</td>
<td>170 x</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>031</td>
<td></td>
<td>EM</td>
<td>57</td>
<td>39</td>
<td>071</td>
<td>9</td>
<td>89</td>
<td>59</td>
<td>131</td>
<td>Z</td>
<td>121</td>
<td>79</td>
<td>171 y</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>032</td>
<td></td>
<td>SUB</td>
<td>58</td>
<td>3a</td>
<td>072</td>
<td>:</td>
<td>90</td>
<td>5a</td>
<td>132</td>
<td>a</td>
<td>122</td>
<td>7a</td>
<td>172 z</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>033</td>
<td></td>
<td>ESC</td>
<td>59</td>
<td>3b</td>
<td>073</td>
<td>;</td>
<td>91</td>
<td>5b</td>
<td>133</td>
<td>b</td>
<td>123</td>
<td>7b</td>
<td>173</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>034</td>
<td></td>
<td>FS</td>
<td>60</td>
<td>3c</td>
<td>074</td>
<td>&lt;</td>
<td>92</td>
<td>5c</td>
<td>134</td>
<td>c</td>
<td>124</td>
<td>7c</td>
<td>174</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>035</td>
<td></td>
<td>GS</td>
<td>61</td>
<td>3d</td>
<td>075</td>
<td>&gt;</td>
<td>93</td>
<td>5d</td>
<td>135</td>
<td>d</td>
<td>125</td>
<td>7d</td>
<td>175</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>036</td>
<td></td>
<td>RS</td>
<td>62</td>
<td>3e</td>
<td>076</td>
<td>^</td>
<td>94</td>
<td>5e</td>
<td>136</td>
<td>e</td>
<td>126</td>
<td>7e</td>
<td>176</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>037</td>
<td></td>
<td>US</td>
<td>63</td>
<td>3f</td>
<td>077</td>
<td>?</td>
<td>95</td>
<td>5f</td>
<td>137</td>
<td>f</td>
<td>127</td>
<td>7f</td>
<td>177</td>
<td></td>
</tr>
</tbody>
</table>

[www.alpharithms.com](http://www.alpharithms.com)
Horner’s method: Convert any object to integer

Start with an object, then just call its toString

Almost any prime number

Handles really large numbers

```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;
}
```

$33^{15} = 59938945498865420543457$
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
- Rehashing
  - make the table bigger
    - O(n) time so want to avoid
    - Also, simply making table bigger does not always eliminate collisions
- Alternative to rehashing
  - Separate Chaining
  - Probing
Separate Chaining

- Idea: each spot in hashtable holds a 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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>762</td>
<td>0</td>
<td>210</td>
</tr>
<tr>
<td>1</td>
<td>217</td>
<td>1</td>
<td>342</td>
</tr>
<tr>
<td>2</td>
<td>29</td>
<td>2</td>
<td>252</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3</td>
<td>136</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>4</td>
<td>55</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>
In class exercise

• Show the final contents of the hashtable using separate chaining assuming
  • table size is 7
  • $h(t) = t \mod 7$
• 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>
• What is the longest chain?

23
Open Addressing 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

- If spot is occupied, go to spot+1, spot+2, ...
- Suppose
  - hashtable size is 7
  - \( h(t) = t \% 7 \)
  - add:
    - \(<3,A>\)
    - \(<10,B>\)
    - \(<17,C>\)
    - \(<24,Z>\)
    - \(<3,D>\)
    - \(<4,E>\)
Probing Distance

• Given a hash value $h(x)$, linear probing generates $h(x), h(x) + 1, h(x) + 2, \ldots$
  • Primary clustering – the bigger the cluster gets, the faster it grows

• Quadratic probing – $h(x), h(x) + 1, h(x) + 4, h(x) + 9, \ldots$
  • 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
In class exercise

- Show the final contents of the hashtable using linear probing assuming
  - table size is 13
  - \( h(t) = t \% 13 \)
- 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?
Using Hashtables

- No worries about hashing functions, rehashing, ...
- Someone else responsibility
- Example: who is visiting my site, and how often?
- for instance, hackers?
- web servers keep access logs
- java.util.HashMap