Hash Tables
Map

- A searchable collection of key-value pairs
- Multiple entries with the same key are not allowed
- Also known as dictionary (python), associative array (perl)
HashTables

• A hash table is an array of size $N$
  ▫ associated hash function $h$ that maps a key to integers into [0, N-1]
  ▫ item $(k, v)$ is stored at index $h(k)$

• $h(x) = x \% N$ is such a function for integers
public class SimpleHT {
    private String[] backingArray;
    public SimpleHT() {
        backingArray = new String[4];
    }
    private int h(int k) {
        return k%4;
    }
    public void put(Integer key, String value) {
        backingArray[h(key)] = value;
    }
    public String get(Integer key) {
        return backingArray[h(key)];
    }
}
HashTable Example

```java
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));
}
```

adding item with key=0 value=a
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 $\rightarrow$ $[0, N-1]$
Hash Codes

- Polynomial accumulation: partition bits of key into a sequence of components of fixed length $a_0a_1\ldots a_{n-1}$
- Evaluate the polynomial
  \[ p(z) = a_0 + a_1 z + a_2 z^2 + \ldots + a_{n-1} z^{n-1} \]
Polynomial accumulation on Strings

- Recommended by textbook
- Handles really large numbers

```java
static int POLY_MULT = 33;

public int stringHasher(String ss) {
    BigInteger ll = 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)));
        ll = ll.add(bb);
    }
    ll = ll.mod(BigInteger.valueOf(backingArray.length));
    return ll.intValue();
}
```

$33^{15} = 59938945498865420543457$
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
  • Alternative to rehashing
    • Separate Chaining
    • Probing
Separate Chaining

• Idea: each spot in hashtable holds a linked 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 object 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>762</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>217</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>29</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>7</td>
<td>7</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>

CS206 12
Linear Probing

- Store only \(^{<}K,V\)^ at each location in array
- No awkward linked 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:
    • $<3, A>$
    • $<10, B>$
    • $<17, C>$
    • $<24, Z>$
    • $<3, D>$
    • $<4, E>$
Mini-homework

• Show the final contents of the hashtable using linear probing assuming
  • table size is 23
  • \( h(t) = t \mod 23 \)
• 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?