Announcements

• Assignment 4 grades released in SPARK

• Assignment 6 posted.

• Complete quiz 9 before tomorrow morning.
Load Factor

- **Load factor**: the ratio of the number of items in the table to the table size.

Example:

A table of capacity 73 contains 40 elements, load factor = 40/73 = 56%
Load Factor

• Rehashing
  – When the load factor becomes too large, it’s necessary to increase the hash table capacity.
  – A bigger hash table is created.
  – Since the table size has changed, we need to rehash every element to find its location in the new HT. (You cannot simply copy elements over!!)
Separate Chaining

• Open Addressing resolves collisions by looking for an empty slot in the hash table.
• Another idea is to install a linked list at each slot to allow for multiple elements. → This is called separate chaining.
Separate Chaining

• Conceptually simple, but involves more coding.

• Workshop applet

• Here load factor can be more than 1.
• A real-world example: organizing letters.
Separate Chaining

• Some simplifications compared to open addressing:
  – Load factor
  – Deletion
  – Table size

• Disadvantages:
  – Using a linked list can be expensive.
Hash Functions

• Let’s come back to talk about hash functions.
• Quick computation
• Random keys
• Non-random keys
Hashing Strings

• Remember how we converted an arbitrary string to a key code.
  
  \[ \text{cats} \rightarrow 3 \times 27^3 + 1 \times 27^2 + 20 \times 27^1 + 19 \times 27^0 = 60,337 \]

• In general:

  \[ c_0 + c_1 \times 27^1 + c_2 \times 27^2 + c_3 \times 27^3 + \ldots + c_k \times 27^k \]

• Java code to implement this.
public static int hashFunc1(String key)
{
    int hashVal = 0;
    int pow27 = 1;  // 1, 27, 27*27, etc

    for(int j=key.length()-1; j>=0; j--) // right to left
    {
        int letter = key.charAt(j) - 96; // get char code
        hashVal += pow27 * letter;  // times power of 27
        pow27 *= 27;                 // next power of 27
    }
    return hashVal % arraySize;
}  // end hashFunc1()
Hashing Strings

• A more efficient version:

\[ c_0 + c_1 \times 27^1 + c_2 \times 27^2 + c_3 \times 27^3 + \ldots + c_k \times 27^k \]

\[ c_0 + 27 \times (c_1 + 27 \times (c_2 + 27 \times (c_3 \ldots + 27 \times c_k))) \]

• Java Code to implement this version.
public static int hashFunc2(String key)
{
    int hashVal = key.charAt(0) - 96;

    for(int j=1; j<key.length(); j++)  // left to right
    {
        int letter = key.charAt(j) - 96;  // get char code
        hashVal = hashVal * 27 + letter;  // multiply and add
    }

    return hashVal % arraySize;       // mod
}  // end hashFunc2()
Hashing Strings

• But a big problem with this hash function is that it can’t handle very long strings → the key code can easily go out of bound.

• How do we modify it to solve this problem?
Hashing Strings

• But a big problem with this hash function is that it can’t handle very long strings → the key code can easily go out of bound.

• How do we modify it to solve this problem?

• Remember we need `keycode % arraysize` in the end. So no matter how big the key code is, it will be turned into a small number in the end. Any idea?
Hashing Strings

• The final version:

```java
public static int hashFunc3(String key)
{
    int hashVal = 0;
    for(int j=0; j<key.length(); j++)  // left to right
    {
        int letter = key.charAt(j) - 96;  // get char code
        hashVal = (hashVal * 27 + letter) % arraySize;  // mod
    }
    return hashVal;  // no mod
}  // end hashFunc3()
```
Hashing Strings

• Java String’s `hashCode()` method.
  Sample a fixed number of characters in the string to avoid computing a large key code.
Hashing Efficiency

• If no collision occurs, O(1) cost.
• If collisions occur, the cost is dependent on the length of probes, which in turn relates to the load factor.
• How do we analyze this?
Hashing Efficiency

• This requires some knowledge of probability.
• Let’s talk about **linear probing**.
• Throw a coin, what’s the probability of the coin landing on head vs. tail?
Hashing Efficiency

• Let’s talk about linear probing.
• Throw a coin, what’s the probability of the coin landing on head vs. tail?
• The load factor $L$ is analogous to the tail probability.
• How many turns does it take to land on head?
Hashing Efficiency

• (Expected) Probe length for a successful search:

\[ P = \frac{1}{2} \left(1 + \frac{1}{1-L} \right) \]

• (Expected) Probe length for an unsuccessful search:

\[ P = \frac{1}{2} \left(1 + \frac{1}{(1-L)^2} \right) \]
Hashing Efficiency

• What about **quadratic probing** and **double hashing**?
Hashing Efficiency

• What about separate chaining?
Hashing Efficiency

• **Some general guidelines:**
  • For open addressing, try to maintain a load factor of 0.5.
  • For separate chaining, try to maintain a load factor of 1.0
Hashing Efficiency

• **Some general guidelines:**

  • If you have a good idea about the number of elements (such as in the case of hashing dictionary words), use open addressing.

  • Otherwise, use separate chaining.

  • If you go with open addressing, use double hashing.