Last Time

• Red-Black Trees
  – One-to-one mapping to 2-3 Trees
  – \textcolor{red}{RED} links represent 3-Nodes
  – \texttt{rotateLeft()}, \texttt{rotateRight()}, and \texttt{flipColors()} functions for tree repair
  – Insertion and retrieval
  – A quick delete discussion
Hashing

- **Hashing** – We can reference key-value pairs using arrays by doing arithmetic operations to transform keys into array indices.
  - **Hash Function** – A function to computationally transform the search key into an array index.
  - **Collision Resolution** – Two or more keys may hash to the same array index.

- **Advantage:** With hashing, you can implement search and insert in **CONSTANT** time per operation.
Hash Functions

- If we have a storage array with $M$ available locations, we need a hash function with an output range of $[0, M - 1]$.
- The hash function should be easy to compute, and should uniformly distribute the keys.
- Hash functions are dependent on data type:
  - If the key involves a number, we can start with that number.
  - If the key involves a string, we can convert that string into a number and then process it as above.
  - If the key has multiple parts, we want to combine those parts before processing.
Modular Hashing

• Choose the size of the array $M$ to be a prime number. (why?)

• For any positive integer key $k$, compute $k \% M$ and use the result as the array index.

• Disperses the keys randomly through the array... as long as the values of $k$ have no relationship to the values of $M$. 
Multiplicative Hashing for Floating Point

- For floating-point keys, one naïve approach would be to multiply the fractional component by $M$, and round down to get a value in the $[0, M - 1]$ range.
  - **Problem:** More weigh is given to the most significant bits of the key, and the least significant bits (which likely have more variability) play no role.
  - **Solution:** Don’t do multiplicative, keep doing modular. Java uses modular hashing on the binary representation of the number.
Modular Hashing for Strings

• We can hash strings with the modular approach by treating them as a large integer.
  – The code below is equivalent to treating the string as an N-digit, base-R integer.

```java
int hash = 0;
for (int i = 0; i < s.length(); i++) {
    hash = (R * hash + s.charAt(i)) % M;
}
```

Modular Hashing for Compound Strings

• If our key has multiple fields, we can mix them together in a way similar to the string behavior.
  – The method generalizes to handle any number of fields.

```c
int hash = (((day * R + month) % M) * R + year) % M;
```
Using `hashCode()`

- Java includes a `hashCode()` function in every datatype.
  - `hashCode()` must be consistent with `equals()`.
  - If you want to hash a user-defined type, you must override both `equals()` and `hashCode()`.
  - We can combine `hashCode()` with modular hashing to produce integers \([0, M - 1]\):
    \[
    (x\.hashCode() \& 0x7FFFFFFF) \% M
    \]
Collision Resolution

• With only $M$ array locations to choose from, it is possible to have hash collisions – two or more keys get inserted into the same index. Two approaches:
  – Separate Chaining – Make each array index into a linked list that we can search through.
  – Linear Probing – Move the colliding item into a new array index.
Separate Chaining

![Table and Diagram]

<table>
<thead>
<tr>
<th>key</th>
<th>hash</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>R</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>H</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>X</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>M</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>P</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>L</td>
<td>3</td>
<td>11</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>12</td>
</tr>
</tbody>
</table>

First node connections:
- first node: A8 -> E12
- first node: null
- first node: X7 -> S0
- first node: L11 -> P10
- first node: M9 -> H5 -> C4 -> R3

Legend:
- independent SequentialSearchST objects

Separate Chaining

```java
void put(Key key, Value val) {
    st[hash(key)].put(key, val);
} //put

Value get(Key key) {
    return (Value) st[hash(key)].get(key);
} //get
```
Separate Chaining – Evaluation

• With $M$ lists and $N$ keys, the average length of a list is $N/M$.
  
  – This may not necessarily be a good thing – the average list length is still $N/M$ if all $N$ items are in one list and the other $M-1$ lists are empty.
  
  – But! We have our assumption that our hash function will uniformly distribute keys across all $M$ lists.
Separate Chaining – Evaluation

• **Theorem:** In a separate chaining hash table with $M$ lists and $N$ keys, the probability that the number of keys in a list is within a small constant factor of $N/M$ is extremely close to 1.

• **Proof:**
  – The probability that a given list will contain exactly $k$ keys is given by the binomial distribution: $P(k) = \binom{N}{k} \left( \frac{1}{M} \right)^k \left( \frac{M-1}{M} \right)^{N-k}$
Separate Chaining - Evaluation

![Graph of list lengths and frequency](image)

- $k \cdot e^{-\alpha} / k! = 10.711...$

![Graph of equality tests and operations](image)

- Cumulative average: 3.9

02/24/2015 Hash Tables
Time-Space Tradeoff

• If there were no memory limitation, we could do any search with only one member access by simply using the key as an index in a huge array.

• If there were no time limitation, we could do any search with sequential search, minimizing the amount of space to the size of all keys.

• Our goal is to strike a balance between these two extremes – don’t waste memory on empty chains, but also don’t waste time on long chains.
Linear Probing

• On insertion, when we have a collision, just move to the next open array index.
• On search, if we don’t find what we’re looking for, try looking at the next open slot.
## Linear Probing

<table>
<thead>
<tr>
<th>Key</th>
<th>Hash</th>
<th>Value</th>
<th>Hash Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>6</td>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>E</td>
<td>10</td>
<td>1</td>
<td>E</td>
</tr>
<tr>
<td>A</td>
<td>4</td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>R</td>
<td>14</td>
<td>3</td>
<td>R</td>
</tr>
<tr>
<td>C</td>
<td>5</td>
<td>4</td>
<td>C</td>
</tr>
<tr>
<td>H</td>
<td>4</td>
<td>5</td>
<td>H</td>
</tr>
<tr>
<td>E</td>
<td>10</td>
<td>6</td>
<td>E</td>
</tr>
<tr>
<td>X</td>
<td>15</td>
<td>7</td>
<td>X</td>
</tr>
<tr>
<td>A</td>
<td>4</td>
<td>8</td>
<td>A</td>
</tr>
<tr>
<td>M</td>
<td>1</td>
<td>9</td>
<td>M</td>
</tr>
<tr>
<td>P</td>
<td>14</td>
<td>10</td>
<td>P</td>
</tr>
<tr>
<td>L</td>
<td>6</td>
<td>11</td>
<td>L</td>
</tr>
<tr>
<td>E</td>
<td>10</td>
<td>12</td>
<td>E</td>
</tr>
</tbody>
</table>

- **Entries in red** are new.
- **Entries in gray** are untouched.
- **Keys in black** are probes.

**Note:**
- Probe sequence wraps to 0.
- Keys[] and vals[].
Linear Probing

```java
void put(Key key, Value val) {
    int i;
    for (i = hash(key); keys[i] != null; 
         i = (i+1)%M) {
        if (keys[i].equals(key)) {
            vals[i] = val;
            return;
        } //if
    } //for
    keys[i] = key;
    vals[i] = val;
} //put
```
Value get(Key key) {
    for (int i = hash(key); keys[i] != null;
        i = (i+1)%M) {
        if (keys[i].equals(key)) {
            return vals[i];
        } //if
    } //for
    return null;
} //get
Linear Probing – Analysis

• As with Separate Chaining, the performance depends on the $N/M$ ratio.

• Linear probing has an extra issue – when the array fills, an insert will cause an infinite loop looking for an open index.

• For good performance, we use array resizing.
  – We want to guarantee that the load factor is between 1/8 and 1/2.
Linear Probing

```java
void resize(int cap) {
    LinearProbingST<Key, Value> t;
    t = new LinearProbingST<Key, Value>(cap);
    for (int i = 0; i < M; i++) {
        if (keys[i] != null) {
            t.put(keys[i], vals[i]);
        } //if
    } //for
    keys = t.keys;
    vals = t.vals;
    M = t.M;
} //put

void put(Key key, Value val) {
    if (N >= M/2) {
        resize(2*M);
    } //if
    .......
```
Linear Probing – Evaluation

cumulative average

operations

equality tests

0 10
0 14350

3.2
Any Questions?