Last Time

• Hash Tables
  – Transform a key into an array index through use of a hash function
  – Need to resolve collisions if two different keys are inserted into the same array index
    • Separate Chaining – each array index is a linked list
    • Linear Probing – if an array index is full, move to the next one
    • Double Hashing – if an array index is full, jump a fixed value to a new location
Why a Whole Unit on Strings?

• Most communication occurs through strings:
  – Web searches? Strings.
  – Programs? Strings.
  – Text messages? Strings.
  – Emails? Strings.
  – All written language? Strings.
  – Speeches and lectures? Encoded as strings.
  – Humans? DNA can be represented as a string.
Why Start with String Sorting?

• Strings were a special condition in our sorting algorithms that we didn’t talk about in detail.
  – Integers and characters are easy to show which is greater and which is smaller.
  – How fast does `String.compareTo()` run?
  – Do we sort based on the first character, or the last?
  – Do we sort based on characters or on string length?
  – What if our string contains numbers, or special characters?
  – Can we efficiently handle all of these cases, depending on the needs of our system?
Key-Indexed Counting

• **Input:** Key-Value pairs consisting of a collection of students and the “section” that they belong to

• **Output:** Students sorted into sections, sections sorted into a sensible order

<table>
<thead>
<tr>
<th>name</th>
<th>section</th>
<th>sorted result (by section)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anderson</td>
<td>2</td>
<td>Harris 1</td>
</tr>
<tr>
<td>Brown</td>
<td>3</td>
<td>Martin 1</td>
</tr>
<tr>
<td>Davis</td>
<td>3</td>
<td>Moore 1</td>
</tr>
<tr>
<td>Garcia</td>
<td>4</td>
<td>Anderson 2</td>
</tr>
<tr>
<td>Harris</td>
<td>1</td>
<td>Martinez 2</td>
</tr>
<tr>
<td>Jackson</td>
<td>3</td>
<td>Miller 2</td>
</tr>
<tr>
<td>Johnson</td>
<td>4</td>
<td>Robinson 2</td>
</tr>
<tr>
<td>Jones</td>
<td>3</td>
<td>White 2</td>
</tr>
<tr>
<td>Martin</td>
<td>1</td>
<td>Brown 3</td>
</tr>
<tr>
<td>Martinez</td>
<td>2</td>
<td>Davis 3</td>
</tr>
<tr>
<td>Miller</td>
<td>2</td>
<td>Jackson 3</td>
</tr>
<tr>
<td>Moore</td>
<td>1</td>
<td>Jones 3</td>
</tr>
<tr>
<td>Robinson</td>
<td>2</td>
<td>Taylor 3</td>
</tr>
<tr>
<td>Smith</td>
<td>4</td>
<td>Williams 3</td>
</tr>
<tr>
<td>Taylor</td>
<td>3</td>
<td>Garcia 4</td>
</tr>
<tr>
<td>Thomas</td>
<td>4</td>
<td>Johnson 4</td>
</tr>
<tr>
<td>Thompson</td>
<td>4</td>
<td>Smith 4</td>
</tr>
<tr>
<td>White</td>
<td>2</td>
<td>Thomas 4</td>
</tr>
<tr>
<td>Williams</td>
<td>3</td>
<td>Thompson 4</td>
</tr>
<tr>
<td>Wilson</td>
<td>4</td>
<td>Wilson 4</td>
</tr>
</tbody>
</table>
Key-Indexed Counting

- Step 1: Compute frequency counts for each key/section

```java
for (int i = 0; i < N; i++) {
    count[a[i].key() + 1]++;
}
```

- Note that if the key/section value is r, we are incrementing count[r+1]
Key-Indexed Counting

• Step 2: Transform counts to indices
  – For each key value, the starting index in the sorted order of items must be computed (where do the 2s start, the 3s start, etc.).

```java
for (int r = 0; r < R; r++) {
    count[r+1] += count[r];
} //for
```
Key-Indexed Counting

• Step 3: Distribute the data
  – Move the data into an auxiliary array, moving them to the position indicated in the `count[]` array, and increment the value in `count[]` for the next data item

```java
for (int i = 0; i < N; i++) {
    aux[count[a[i].key()]]++ = a[i];
}
```

03/09/2016 String Sorts
### Key-Indexed Counting

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>3</td>
<td>8</td>
<td>14</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>4</td>
<td>8</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>4</td>
<td>9</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>4</td>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>4</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>4</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>4</td>
<td>11</td>
<td>15</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>4</td>
<td>11</td>
<td>16</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>4</td>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>4</td>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>5</td>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>6</td>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>6</td>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td>13</td>
<td>3</td>
<td>7</td>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td>14</td>
<td>3</td>
<td>7</td>
<td>12</td>
<td>17</td>
</tr>
<tr>
<td>15</td>
<td>3</td>
<td>7</td>
<td>13</td>
<td>17</td>
</tr>
<tr>
<td>16</td>
<td>3</td>
<td>7</td>
<td>13</td>
<td>18</td>
</tr>
<tr>
<td>17</td>
<td>3</td>
<td>7</td>
<td>13</td>
<td>19</td>
</tr>
<tr>
<td>18</td>
<td>3</td>
<td>8</td>
<td>13</td>
<td>19</td>
</tr>
<tr>
<td>19</td>
<td>3</td>
<td>8</td>
<td>14</td>
<td>19</td>
</tr>
</tbody>
</table>

a[0] Anderson 2  Harris 1 aux[0]
a[12] Robinson 2  Taylor 3 aux[12]
a[16] Thompson 4  Smith 4 aux[16]
a[18] Williams 3  Thompson 4 aux[18]
Key-Indexed Counting

• Step 4: Copy the data back from aux[] into a[]

```c
for (int i = 0; i < N; i++) {
    a[i] = aux[i];
}
```

Key-Indexed Counting – Analysis

• **Theorem:** Key-indexed counting uses $11N + 4R + 1$ array accesses to stably sort $N$ items whose keys are integers between 0 and $R - 1$.

• **Proof:** Immediate from the code.
  – Initializing the arrays uses $N + R + 1$ accesses.
  – First loop increments a counter for each of the $N$ items ($3N$ accesses).
  – Second loop does $R$ additions ($3R$ accesses).
  – Third loop does $N$ counter increments and $N$ data moves ($5N$ accesses).
  – Fourth loop does $N$ data moves ($2N$ accesses).
Wait, Did We Just Break the \( O(n \times \log(n)) \) Bound?

- **Theorem:** Key-indexed counting uses \( 11N + 4R + 1 \) array accesses to stably sort \( N \) items whose keys are integers between 0 and \( R - 1 \).

- Well, no.
  - Our proposition put an \( O(n \times \log(n)) \) bound on the number of compares. Key-indexed counting does no compares, accessing the data only through \texttt{key()}.  

- When \( R \) is within a constant factor of \( N \), we have a **linear-time sort**.
Stability

- A sorting algorithm is **stable** if it preserves the relative order of equal keys in the array.

```plaintext
<table>
<thead>
<tr>
<th>Chicago</th>
<th>09:00:00</th>
<th>Chicago</th>
<th>09:25:52</th>
<th>Chicago</th>
<th>09:00:00</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phoenix</td>
<td>09:00:03</td>
<td>Chicago</td>
<td>09:03:13</td>
<td>Chicago</td>
<td>09:00:59</td>
</tr>
<tr>
<td>Houston</td>
<td>09:00:13</td>
<td>Chicago</td>
<td>09:21:05</td>
<td>Chicago</td>
<td>09:19:46</td>
</tr>
<tr>
<td>Chicago</td>
<td>09:00:59</td>
<td>Chicago</td>
<td>09:19:32</td>
<td>Chicago</td>
<td>09:19:32</td>
</tr>
<tr>
<td>Houston</td>
<td>09:01:10</td>
<td>Chicago</td>
<td>09:00:00</td>
<td>Chicago</td>
<td>09:35:21</td>
</tr>
<tr>
<td>Chicago</td>
<td>09:03:13</td>
<td>Chicago</td>
<td>09:19:32</td>
<td>Chicago</td>
<td>09:19:32</td>
</tr>
<tr>
<td>Seattle</td>
<td>09:10:11</td>
<td>Chicago</td>
<td>09:00:00</td>
<td>Chicago</td>
<td>09:35:21</td>
</tr>
<tr>
<td>Seattle</td>
<td>09:10:25</td>
<td>Houston</td>
<td>09:01:10</td>
<td>Houston</td>
<td>09:00:13</td>
</tr>
<tr>
<td>Phoenix</td>
<td>09:14:25</td>
<td>Houston</td>
<td>09:00:13</td>
<td>Phoenix</td>
<td>09:37:44</td>
</tr>
<tr>
<td>Chicago</td>
<td>09:19:32</td>
<td>Phoenix</td>
<td>09:00:03</td>
<td>Phoenix</td>
<td>09:14:25</td>
</tr>
<tr>
<td>Chicago</td>
<td>09:19:46</td>
<td>Phoenix</td>
<td>09:14:25</td>
<td>Phoenix</td>
<td>09:37:44</td>
</tr>
<tr>
<td>Chicago</td>
<td>09:21:05</td>
<td>Phoenix</td>
<td>09:14:25</td>
<td>Phoenix</td>
<td>09:14:25</td>
</tr>
<tr>
<td>Seattle</td>
<td>09:22:43</td>
<td>Seattle</td>
<td>09:10:25</td>
<td>Seattle</td>
<td>09:10:11</td>
</tr>
<tr>
<td>Seattle</td>
<td>09:22:54</td>
<td>Seattle</td>
<td>09:10:25</td>
<td>Seattle</td>
<td>09:10:25</td>
</tr>
<tr>
<td>Chicago</td>
<td>09:35:21</td>
<td>Seattle</td>
<td>09:10:11</td>
<td>Seattle</td>
<td>09:22:54</td>
</tr>
<tr>
<td>Seattle</td>
<td>09:36:14</td>
<td>Seattle</td>
<td>09:22:54</td>
<td>Seattle</td>
<td>09:36:14</td>
</tr>
</tbody>
</table>
```

- The table on the left is sorted by time, while the middle table is not stable and loses the original order of equal entries.
- The right table is stable, maintaining the original order of equal entries.

The diagram illustrates how sorting by location can preserve the order of equal entries, whereas sorting by time does not.
LSD String Sort

• “Least-Significant Digit First” (LSD) String Sort
  – Imagine that we have a bunch of Internet traffic, and we want to know the number of unique IPs that visit our site.
  – Imagine that a highway engineer sets up a camera to record license plates, and wants to know the number of different vehicles that used that stretch of highway.

• If each of the strings has length $W$, then we sort the strings $W$ times with key-indexed counting, using each position of the key scanned from right to left.
void sort(String a[], int W) {
    int N = a.length;
    int R = 256;
    String[] aux = new String[N];
    for (int d = W-1; d >= 0; d--) {
        int[] count = new int[R+1];
        for (int i = 0; i < N; i++)
            count[a[i].charAt(d)+1]++;
        for (int r = 0; r < R; r++)
            count[r+1] += count[r];
        for (int i = 0; i < N; i++)
            aux[count[a[i].charAt(d)]] = a[i];
    } //for
    for (int i = 0; i < N; i++)
        a[i] = aux[i];
} //sort
LSD String Sort – Visual

<table>
<thead>
<tr>
<th>input (W = 7)</th>
<th>d = 6</th>
<th>d = 5</th>
<th>d = 4</th>
<th>d = 3</th>
<th>d = 2</th>
<th>d = 1</th>
<th>d = 0</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>4PGC938</td>
<td>2IYE230</td>
<td>3CIO720</td>
<td>2IYE230</td>
<td>2RLA629</td>
<td>1ICK750</td>
<td>3ATW723</td>
<td>1ICK750</td>
<td>1ICK750</td>
</tr>
<tr>
<td>3CIO720</td>
<td>3CIO720</td>
<td>2RLA629</td>
<td>2RLA629</td>
<td>1ICK750</td>
<td>3CIO720</td>
<td>1ICK750</td>
<td>1ICK750</td>
<td>1ICK750</td>
</tr>
<tr>
<td>1ICK750</td>
<td>1ICK750</td>
<td>4JZY524</td>
<td>2RLA629</td>
<td>4PGC938</td>
<td>4PGC938</td>
<td>4PGC938</td>
<td>3CIO720</td>
<td>1ICK750</td>
</tr>
<tr>
<td>10HV845</td>
<td>3CIO720</td>
<td>2RLA629</td>
<td>3CIO720</td>
<td>1ICK750</td>
<td>10HV845</td>
<td>1ICK750</td>
<td>10HV845</td>
<td>10HV845</td>
</tr>
<tr>
<td>4JZY524</td>
<td>2RLA629</td>
<td>3CIO720</td>
<td>1ICK750</td>
<td>10HV845</td>
<td>2IYE230</td>
<td>2IYE230</td>
<td>2IYE230</td>
<td>2IYE230</td>
</tr>
<tr>
<td>1ICK750</td>
<td>4JZY524</td>
<td>3CIO720</td>
<td>3CIO720</td>
<td>3CIO720</td>
<td>3CIO720</td>
<td>4JZY524</td>
<td>2RLA629</td>
<td>2RLA629</td>
</tr>
<tr>
<td>3CIO720</td>
<td>2RLA629</td>
<td>4PGC938</td>
<td>1ICK750</td>
<td>3CIO720</td>
<td>10HV845</td>
<td>2RLA629</td>
<td>2RLA629</td>
<td>2RLA629</td>
</tr>
<tr>
<td>10HV845</td>
<td>10HV845</td>
<td>10HV845</td>
<td>10HV845</td>
<td>10HV845</td>
<td>10HV845</td>
<td>2RLA629</td>
<td>10HV845</td>
<td>3ATW723</td>
</tr>
<tr>
<td>10HV845</td>
<td>10HV845</td>
<td>10HV845</td>
<td>10HV845</td>
<td>10HV845</td>
<td>10HV845</td>
<td>3CIO720</td>
<td>3CIO720</td>
<td>3CIO720</td>
</tr>
<tr>
<td>2RLA629</td>
<td>4PGC938</td>
<td>10HV845</td>
<td>10HV845</td>
<td>10HV845</td>
<td>10HV845</td>
<td>3ATW723</td>
<td>4PGC938</td>
<td>3CIO720</td>
</tr>
<tr>
<td>2RLA629</td>
<td>3ATW723</td>
<td>1ICK750</td>
<td>10HV845</td>
<td>2IYE230</td>
<td>2RLA629</td>
<td>4JZY524</td>
<td>4JZY524</td>
<td>4JZY524</td>
</tr>
<tr>
<td>3ATW723</td>
<td>2RLA629</td>
<td>1ICK750</td>
<td>4PGC938</td>
<td>4JZY524</td>
<td>4PGC938</td>
<td>4PGC938</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
LSD String Sort - Analysis

- LSD String Sort depends on the stability of Key-Indexed sorting.
  - After sorting keys on their $i$ trailing characters, we know that any two keys appear in order in the array either because:
    - The first of their $i$ trailing characters is different, in which case the sort on that character puts them in order, or
    - Because the first of their $i$ trailing characters is the same, in which case they are in order because of stability (and by inductor, for $i - 1$).
LSD String Sort – Analysis

- **Theorem:** LSD String Sort uses \( \sim 7WN + 3WR \) array accesses, and extra space proportional to \( N + R \), to sort \( N \) items whose keys are \( W \)-character strings taken from an \( R \)-character alphabet.

- **Proof:** From the code:
  - The method is \( W \) passes of key-indexed counting.
  - The \texttt{aux[]} array is initialized only once.

- For typical applications, \( R \) is much smaller than \( N \), so the total running time is proportional to \( WN \).
  - An array of \( N \) strings that each have \( W \) characters has a total of \( WN \) characters, so the running time of LSD String Sort is linear with respect to the size of the input!
MSD String Sort

• “Most-Significant Digit First” (MSD) String Sort
  – General-purpose string sort, where strings are not necessarily all the same length.
  – Strings that start with “a” should appear before strings that start with “b,” and so forth.
  – We use key-indexed counting to sort the strings according to their first character, then recursively sort the subarrays corresponding to each first character, etc.
  – Similar to Quicksort, MSD String Sort partitions the array into subarrays that can be sorted independently.
Handling Unequal String Lengths

• Which comes first, “she” or “shell”?
  – The subarray for strings whose characters have all been examined should come first.
  – We do not want to recursively sort this subarray.
• We’ll handle this case by defining a custom `charAt()` function, which will return -1 if the requested character position is beyond the length of the string.
  – Because -1 is a possible return value, we now have R+1 different possible character values at each string position.
  – This means we need to add 1 to each returned value, in order to index `count[]` with a nonnegative number.
int M = 15; //cutoff to Insertion Sort
int R = 256;
String[] aux = new String[N];

void sort(String a[], int lo, int hi, int d) {
    if(hi <= lo + M) {
        Insertion.sort(a, lo, hi, d); return; } //if
    int count[] = new int[R+2];
    for(int i = lo; i <= hi; i++)
        count[charAt(a[i], d)+2]++;  
    for(int r = 0; r < R+1; r++)
        count[r+1] += count[r];
    for(int i = lo; i <= hi; i++)
        aux[count[charAt(a[i], d)+1]++] = a[i];
    for(int i = lo; i <= hi; i++)
        a[i] = aux[i - lo]
    for(int r = 0; r < R; r++)     // recursive sort for each char val
        sort(a, lo+count[r], lo+count[r+1]-1, d+1);
} //sort
MSD String Sort – Visual

```
input:
she          are          are          are          are          are          are          are          are
sells        by           by           by           by           by           by           by           by
seashells    she          sells        seashells   sea          seashells   seashells   seashells   seashells
by           sells        seashells   sea          seashells   seashells   seashells   seashells   seashells
the          seashells    sea          sells        sells        sells        sells        sells        sells
sea          sells        seashells   sells        sells        sells        sells        sells        sells
shore        shore        seashells   sells        sells        sells        sells        sells        sells
the          shells       she          she          she          she          she          she          she
shells       she          shore        shore        shore        shore        shore        shore        shore
she          sells        shells       shells       shells       shells       shells       shells       shells
sells        surely       she          she          she          she          she          she          she
are          seashells    surely       surely       surely       surely       surely       surely       surely
surely       the          the          the          the          the          the          the          the
seashells    the          the          the          the          the          the          the          the
```
MSD String Sort – Analysis

• Why include cutoff to Insertion Sort?
  – As we partition, we’ll eventually end up with a huge number of tiny subarrays, where we’ll be spending most of our sorting time.

• Worst case for MSD String Sort is when all keys are equal – we would need a recursive call for every character, when the array is already sorted to begin with.
  – Same argument goes for a large number of strings with the same prefix.
MSD String Sort – Analysis

• Running time of MSD String Sort depends on the data.
  – For random inputs, MSD examines just enough characters to distinguish among the keys. Running time is \textit{sublinear} in the number of characters in the data.
  – For nonrandom inputs, MSD still could be \textit{sublinear}, but might need to examine more characters than in the random case.
  – In the worst case, MSD examines all the characters in the keys, so the running time is \textit{linear} in the number of characters in the data.
MSD String Sort – Analysis

- **Theorem:** To sort N random strings from an R-character alphabet, MSD String Sort examines about $\sim N \times \log_R(N)$ characters in the average case.

- **Proof:** We expect the subarrays to all be roughly the same size, so the recurrence relation $C_N = R C_{N/R} + N$ approximately describes the performance.
MSD String Sort – Analysis

• **Theorem:** MSD String Sort uses between \(8N + 3R\) and \(\sim 7wN + 3wR\) array accesses to sort \(N\) strings taken from an \(R\)-character alphabet, where \(w\) is the average string length.

• **Proof:** Immediate from the code. In the best case, MSD uses just one pass. In the worst case, it performs like LSD.
## Summary

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Stable?</th>
<th>Inplace?</th>
<th>Running Time</th>
<th>Extra Space</th>
<th>Sweet Spot</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>insertion sort</strong></td>
<td>yes</td>
<td>yes</td>
<td>$N$ and $N^2$</td>
<td>1</td>
<td>small arrays, arrays in order</td>
</tr>
<tr>
<td><strong>for strings</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>quicksort</strong></td>
<td>no</td>
<td>yes</td>
<td>$N \log^2 N$</td>
<td>$\log N$</td>
<td>general-purpose when space is tight</td>
</tr>
<tr>
<td><strong>mergesort</strong></td>
<td>yes</td>
<td>no</td>
<td>$N \log^2 N$</td>
<td>$N$</td>
<td>general-purpose stable sort</td>
</tr>
<tr>
<td><strong>3-way quicksort</strong></td>
<td>no</td>
<td>yes</td>
<td>$N$ and $N \log N$</td>
<td>$\log N$</td>
<td>large numbers of equal keys</td>
</tr>
<tr>
<td><strong>LSD string sort</strong></td>
<td>yes</td>
<td>no</td>
<td>$NW$</td>
<td>$N$</td>
<td>short fixed-length strings</td>
</tr>
<tr>
<td><strong>MSD string sort</strong></td>
<td>yes</td>
<td>no</td>
<td>$N$ and $Nw$</td>
<td>$N + WR$</td>
<td>random strings</td>
</tr>
</tbody>
</table>
Any Questions?