CSCI-GA.1144-001

PAC II

Lecture 8: Algorithms II

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com
A Quick Refresh

• We assume we execute our algorithm on RAM.
  – In RAM, instructions are executed one after the other, with no concurrency.
  – RAM model contains instructions available in common computers.
  – Each instruction takes a constant amount of time.
  – We do not attempt to model memory hierarchy.

• We care more about the worst-case scenario.
Sorting

• **Input:** sequence of n numbers
  \[<a_1, a_2, \ldots, a_n>\]

• **Output:** a permutation of the input sequence \[<b_1, b_2, \ldots, b_n>\] such that:
  \[b_1 \leq b_2 \leq \ldots \leq b_n\]
Insertion Sort

• Adding a new element to a sorted list will keep the list sorted if the element is inserted in the correct place.

• A single element list is sorted.

• Inserting a second element in the proper place keeps the list sorted.

• This is repeated until all the elements have been inserted into the sorted part of the list.
Insertion Sort

**Algorithm:**

1. **for** $j = 2$ to length[$A$]
2.  \hspace{1em} key = $A[j]$
3. \hspace{1em} // Insert $A[j]$ into the sorted sequence $A[1...j-1]$
4. \hspace{1em} $i = j - 1$
5. \hspace{1em} while $i > 0$ and $A[i] > key$
6. \hspace{2.5em} $A[i+1] = A[i]$
7. \hspace{1em} $i = i - 1$
8. \hspace{1em} $A[i+1] = key$

*Source:* “Introduction to Algorithms” 3rd Edition
# Insertion Sort

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

Sorted already

Not yet processed
Algorithm Analysis

• In general, the time taken by an algorithm grows with the size of the input.
• So, it is traditional to describe the running time of a program as a function of the size of its input.
• The running time of an algorithm on a particular input is the number of primitive operations executed.
• We care about the worst-case scenario.
Important note before we start

When a **for** or **while** loop exits in the usual way (i.e., due to the test in the loop header), the test is executed one time more than the loop body.
Analyzing Insertion Sort

**INSERTION-SORT (A)**

1. for j = 2 to length[A]  
2.   key = A[j]  
3.   // Insert A[j] into the sorted sequence A[1...j-1]  
4.   i = j - 1  
5.   while i > 0 and A[i] > key  
7.     i = i - 1  
8.   A[i+1] = key

$t_j$ is the number of times the while loop test in step 5 is executed for that value of j.

Source: “Introduction to Algorithms” 3rd Edition
Analyzing Insertion Sort

\[ T(n) = c_1 n + c_2 (n - 1) + c_4 (n - 1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8 (n - 1). \]

**Best case:**
A is sorted

\[ t_j = 1 \text{ in step 5 for all } j \]

\[ T(n) = c_1 n + c_2 (n - 1) + c_4 (n - 1) + c_5 \left( \frac{n(n+1)}{2} - 1 \right) + c_6 \left( \frac{n(n-1)}{2} \right) + c_7 \left( \frac{n(n-1)}{2} \right) + c_8 (n - 1) \]

\[ T(n) = \left( \frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2} \right) n^2 + \left( c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8 \right) n - \left( c_2 + c_4 + c_5 + c_8 \right). \]

**Worst case:**
A is reverse sorted

\[ t_j = j \]

\[ T(n) = an^2 + bn + n \]

*Source: “Introduction to Algorithms” 3rd Edition*
How to Design An Algorithm

• Incremental approach: similar to insertion sort

• Divide-and-conquer approach:
  – **Divide**: break the problem into subproblems similar to the original problem but smaller in size
  – **Conquer**: solve the subproblems recursively
  – **Combine**: combine the solutions to create the solution of the original problem
Merge Sort

Sorts the elements of subarray A[p..r].
Initially: p = 1 and r = length[A]

```
MERGE-SORT(A, p, r)
1    if p < r
2        q = [(p + r)/2]
3        MERGE-SORT(A, p, q)
4        MERGE-SORT(A, q + 1, r)
5        MERGE(A, p, q, r)
```
Merge Sort

\[
\text{MERGE}(A, p, q, r)
\]
\[
1 \quad n_1 = q - p + 1 \\
2 \quad n_2 = r - q \\
3 \quad \text{let } L[1..n_1 + 1] \text{ and } R[1..n_2 + 1] \text{ be new arrays} \\
4 \quad \text{for } i = 1 \text{ to } n_1 \\
5 \quad \quad L[i] = A[p + i - 1] \\
6 \quad \text{for } j = 1 \text{ to } n_2 \\
7 \quad \quad R[j] = A[q + j] \\
8 \quad L[n_1 + 1] = \infty \\
9 \quad R[n_2 + 1] = \infty \\
10 \quad i = 1 \\
11 \quad j = 1 \\
12 \quad \text{for } k = p \text{ to } r \\
13 \quad \quad \text{if } L[i] \leq R[j] \\
14 \quad \quad \quad A[k] = L[i] \\
15 \quad \quad \quad i = i + 1 \\
16 \quad \quad \text{else } A[k] = R[j] \\
17 \quad \quad \quad j = j + 1
\]

Source: “Introduction to Algorithms” 3rd Edition
Execution Example

• Partition

```
7  2  9  4
7  2
7
```

```
3  8  6  1
3  8
3
```

```
1  3  8  6
1  3  8
1
```

```
4  9
4
```

```
2  7
2
```

```
9  4
9
```

```
4  9
4
```

```
3  8
3  8
3
```

```
6  1
6
```

```
1  6
1
```

```
2  7
2
```

```
9  4
9
```

```
4  9
4
```

```
3  8
3
```

```
6  1
6
```

```
1  6
1
```

```
2  7
2
```

```
9  4
9
```

```
4  9
4
```

```
3  8
3
```
Execution Example (cont.)

- Recursive call, partition
Execution Example (cont.)

- Recursive call, partition

```
7 2 9 4 | 3 8 6 1
```

```
7 2 | 9 4
```

```
7 | 2
```

```
1 2 3 4 6 7 8 9
```
Execution Example (cont.)

- Recursive call, base case

```
7 2 9 4 | 3 8 6 1
```

```
7 2 9 4
```

```
7 2
7
```

```
7 2 9 4
```

```
7 2
7
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```

```
7 2 9 4
```
Execution Example (cont.)

- Recursive call, base case

```
7  2  9  4  →  3  8  6  1
```

```
7  2  9  4
```

```
7  2  9  4
```

```
7  2  9  4
```

```
7  2  9  4
```

```
7  2  9  4
```

```
7  2  9  4
```

```
7  2  9  4
```

```
7  2  9  4
```

```
7  2  9  4
```

```
7  2  9  4
```

```
7  2  9  4
```

```
7  2  9  4
```

```
7  2  9  4
```

```
7  2  9  4
```

```
7  2  9  4
```

```
7  2  9  4
```

```
7  2  9  4
```

```
7  2  9  4
```

```
7  2  9  4
```

```
7  2  9  4
```

```
7  2  9  4
```

```
7  2  9  4
```

```
7  2  9  4
```

```
7  2  9  4
```

```
7  2  9  4
```

```
7  2  9  4
```

```
7  2  9  4
```

```
7  2  9  4
```

```
7  2  9  4
```

```
7  2  9  4
```

```
7  2  9  4
```

```
7  2  9  4
```

```
7  2  9  4
```

```
7  2  9  4
```

```
7  2  9  4
```

```
7  2  9  4
```
Execution Example (cont.)

• Merge

\[
\begin{array}{cccc}
7 & 2 & 9 & 4 \\
\hline
3 & 8 & 6 & 1 \\
\end{array}
\]
Execution Example (cont.)

- Recursive call, ..., base case, merge
Execution Example (cont.)

- Merge

```
7 2 9 4 | 3 8 6 1
7 2 9 4 | 2 4 7 9
7 2 2 7 | 9 4 4 9
7 2 2 2 9 4 9 4 4 6
7 2 2 2 9 4 6 1
```

```
7 2 9 4 | 3 8 6 1
7 2 9 4 | 2 4 7 9
7 2 2 7 | 9 4 4 9
7 2 2 2 9 4 9 4 4 6
7 2 2 2 9 4 6 1
```
Execution Example (cont.)

• Recursive call, ..., merge, merge
Execution Example (cont.)

- Merge

```
7 2 9 4 | 3 8 6 1 → 1 2 3 4 6 7 8 9
```

```
7 2 | 9 4 → 2 4 7 9
```

```
3 8 6 1 → 1 3 6 8
```

```
7 | 2 → 2 7
```

```
9 4 → 4 9
```

```
3 8 → 3 8
```

```
6 1 → 1 6
```

```
7 → 7
```

```
2 → 2
```

```
9 → 9
```

```
4 → 4
```

```
3 → 3
```

```
8 → 8
```

```
6 → 6
```

```
1 → 1
```
Analyzing Merge Sort

\[ T(n) = \text{divide work} + \text{conquer work} + \text{combine work} \]

- Calculate the middle of the array
- Recursively solve 2 subproblems each of size \( n/2 \)
- Combine the elements
Analyzing Merge Sort

- \( T(n) = \text{divide work} + \text{conquer work} + \text{combine work} \)
  
  \[ T(n) = D(n) + 2T(n/2) + C(n) \]
  
  \[ T(n) = c + 2T(n/2) + cn \]
Analyzing Merge Sort

• \( T(n) = \text{divide work} + \text{conquer work} + \text{combine work} \)
  
  \[
  = D(n) + 2T(n/2) + C(n) \\
  = c + 2T(n/2) + cn
  \]

Source: “Introduction to Algorithms” 3rd Edition
Bubble Sort

• If we *compare* pairs of adjacent elements and none are out of order, the list is sorted

• If any are out of order, we must swap them to get an ordered list

• Bubble sort will make *passes* though the list swapping any adjacent elements that are out of order
Bubble Sort

• After the first pass, we know that the largest element must be in the correct place

• After the second pass, we know that the second largest element must be in the correct place

• Because of this, we can shorten each successive pass of the comparison loop
Bubble Sort Example

<table>
<thead>
<tr>
<th>16</th>
<th>7</th>
<th>10</th>
<th>1</th>
<th>5</th>
<th>11</th>
<th>3</th>
<th>8</th>
<th>14</th>
<th>4</th>
<th>2</th>
<th>12</th>
<th>6</th>
<th>13</th>
<th>9</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>4</td>
<td>2</td>
<td>8</td>
<td>6</td>
<td>10</td>
<td>9</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Bubble Sort Algorithm

numberOfPairs = N
swappedElements = true

while (swappedElements) do
    numberOfPairs = numberOfPairs - 1
    swappedElements = false
    for i = 1 to numberOfPairs do
        if (A[i] > A[i + 1]) then
            Swap(A[i], A[i + 1])
            swappedElements = true
        end if
    end for
end while
Best-Case Analysis

• If the elements start in sorted order, the for loop will compare the adjacent pairs but not make any changes

• So the swappedElements variable will still be false and the while loop is only done once

• There are \( N - 1 \) comparisons in the best case
Worst-Case Analysis

• In the worst case the while loop must be done as many times as possible. This happens when the data set is in the reverse order.

• Each pass of the for loop must make at least one swap of the elements

• The number of comparisons will be:

\[ W(N) = \sum_{i=1}^{N-1} (N - i) = \sum_{k=N-1}^{1} k = \sum_{i=1}^{N-1} i = \frac{(N - 1) * N}{2} = O(N^2) \]
Quicksort Algorithm

• Another divide-and-conquer algorithm
• Quicksort is usually $O(n \log n)$ but in the worst case slows to $O(n^2)$

Given an array of $n$ elements (e.g., integers):
• If array only contains one element, return
• Else
  – pick one element to use as pivot.
  – Partition elements into two sub-arrays:
    • Elements less than or equal to pivot
    • Elements greater than pivot
  – Quicksort two sub-arrays
  – Return results
Quicksort

- Divide step:
  - Pick any element (pivot) \( v \) in \( S \)
  - Partition \( S - \{v\} \) into two disjoint groups
    \( S_1 = \{x \in S - \{v\} \mid x \leq v\} \)
    \( S_2 = \{x \in S - \{v\} \mid x \geq v\} \)

- Conquer step: recursively sort \( S_1 \) and \( S_2 \)

- Combine step: the sorted \( S_1 \) (by the time returned from recursion), followed by \( v \), followed by the sorted \( S_2 \) (i.e., nothing extra needs to be done)
Example
quicksort small

quicksort large
Pseudo-code

QUICKSORT($A, p, r$)
1    if $p < r$
2         $q = \text{PARTITION}(A, p, r)$
3    QUICKSORT($A, p, q - 1$)
4    QUICKSORT($A, q + 1, r$)

PARTITION($A, p, r$)
1    $x = A[r]$
2    $i = p - 1$
3    for $j = p$ to $r - 1$
4        if $A[j] \leq x$
5            $i = i + 1$
6        exchange $A[i]$ with $A[j]$
7    exchange $A[i + 1]$ with $A[r]$
8    return $i + 1$
More Sorting Algorithms

• Shell sort
• Heap sort
• Radix sort
• Counting sort
• Bucket sort
• ...
Now that we have a sorted array, what is the most efficient way to search an element in it?
Binary Search

- Binary search. Given value and sorted array $a[]$, find index $i$ such that $a[i] = \text{value}$, or report that no such index exists.

- Ex. Binary search for 33.
Binary Search
Binary Search

-lo

↑

hi
Binary Search
Binary Search
Binary Search
Binary Search

lo

hi
Binary Search
Binary Search
Efficiency of binary search

- If $n$ represents the number of names, the maximum number of searches $x$ necessary to find a name is the smallest integer that satisfies the inequality $2^x > n$.

\[
\begin{align*}
2^x &> n \\
\log (2^x) &> \log n \\
x \log 2 &> \log n
\end{align*}
\]

The maximum number of searches is the smallest integer greater than $\log n / \log 2$. 
# Efficiency of binary search

With the incredible speed of today’s computers, a binary search becomes necessary only when the number of elements is large.

<table>
<thead>
<tr>
<th># of elements</th>
<th>Maximum sequential searches necessary</th>
<th>Maximum binary searches necessary</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>7</td>
</tr>
<tr>
<td>1,000</td>
<td>1,000</td>
<td>10</td>
</tr>
<tr>
<td>5,000</td>
<td>5,000</td>
<td>13</td>
</tr>
<tr>
<td>10,000</td>
<td>10,000</td>
<td>14</td>
</tr>
<tr>
<td>50,000</td>
<td>50,000</td>
<td>16</td>
</tr>
<tr>
<td>100,000</td>
<td>100,000</td>
<td>17</td>
</tr>
<tr>
<td>1,000,000</td>
<td>1,000,000</td>
<td>20</td>
</tr>
<tr>
<td>10,000,000</td>
<td>10,000,000</td>
<td>24</td>
</tr>
</tbody>
</table>
Don’t you think that binary search is related to trees?
Tree Example: Linux File Structure
Another Tree Example: Compiler Parse Tree

Parse tree for:
\[ x=1 \]
\[ y=2 \]
\[ 3 \ast (x+y) \]
So ... What is a tree?

- A tree is a **finite set of one or more nodes** such that:
  - There is a specially designated node called the **root**.
  - The remaining nodes are partitioned into \( n \geq 0 \) disjoint sets \( T_1, \ldots, T_n \), where each of these sets is a tree.
- We call \( T_1, \ldots, T_n \) the **subtrees** of the root.
Some Definitions

- The **degree of a node** is the number of subtrees of the node.
- The node with **degree 0** is a leaf or terminal node.
- A node that has subtrees is the **parent** of the roots of the subtrees.
- The roots of these subtrees are the **children** of the node.
- Children of the same parent are **siblings**.
- The **ancestors** of a node are all the nodes along the path from the root to the node.
- The **level or depth** of a node \( n \) is the length of the unique path from the root to \( n \).
A Tree Node

• Every tree node:
  – object - useful information
  – children - pointers to its children nodes
Left Child - Right Sibling
Example: Tree Implementation

```c
struct tnode {
    int key;
    struct tnode* lchild;
    struct tnode* sibling;
};
```

Example of operations:
- Create a tree with three nodes (one root & two children)
- Insert a new node (in tree with root R, as a new child at level L)
- Delete a node (in tree with root R, the first child at level L)
Binary Trees

- A special class of trees: max degree for each node is 2
- Recursive definition: A binary tree is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called the left subtree and the right subtree.
Example: Is this a binary tree?
Example of Binary Trees

Skewed Binary Tree

Complete Binary Tree
Maximum Number of Nodes in BT

• The maximum number of nodes on level $i$ of a binary tree is $2^{i-1}$, $i \geq 1$ (assuming root is at level 1)
• The maximum number of nodes in a binary tree of depth $k$ is $2^{k-1}$, $k \geq 1$. 
Full BT vs. Complete BT

- A full binary tree of depth $k$ is a binary tree of depth $k$ having $2^k - 1$ nodes, $k \geq 0$ (root is at depth 1).
- A binary tree with $n$ nodes and depth $k$ is complete iff its nodes correspond to the nodes numbered from 1 to $n$ in the full binary tree of depth $k$.
Binary Tree Representations: Array

- If a complete binary tree with $n$ nodes is represented sequentially, then for any node with index $i$, $1 \leq i \leq n$, we have:
  - $\text{parent}(i)$ is at $i/2$ if $i!=1$. If $i=1$, $i$ is at the root and has no parent.
  - $\text{leftChild}(i)$ is at $2i$ if $2i \leq n$. If $2i > n$, then $i$ has no left child.
  - $\text{rightChild}(i)$ is at $2i+1$ if $2i+1 \leq n$. If $2i+1 > n$, then $i$ has no right child.
Array presentation (aka Sequential presentation)

(1) waste space
(2) insertion/deletion problem
Tree Presentation: Linked Representation

typedef struct tnode *ptnode;
typedef struct tnode {
    int data;
    ptnode left, right;
};
Binary Tree Traversals

- There are six possible combinations of traversal:
  - lRr, lrR, Rlr, Rrl, rRl, rlr
- Adopt convention that we traverse **left before right**, only 3 traversals remain:
  - lRr, lrR, Rlr
  - inorder, postorder, preorder
Example:
Arithmetic Expression Using BT

inorder traversal
A / B * C * D + E

infix expression
preorder traversal
+ * * / A B C D E

prefix expression
postorder traversal
A B / C * D * E +
postfix expression
void inorder(ptnode ptr)
/* inorder tree traversal */
{
    if (ptr) {
        inorder(ptr->left);
        printf("%d", ptr->data);
        inorder(ptr->right);
    }
}
void preorder(ptnode ptr)
/* preorder tree traversal */
{
    if (ptr) {
        printf("%d", ptr->data);
        preorder(ptr->left);
        preorder(ptr->right);
    }
}

+ * */ A B C D E
void postorder(ptnode ptr)
/* postorder tree traversal */
{
    if (ptr) {
        postorder(ptr->left);
        postdorder(ptr->right);
        printf("%d", ptr->data);
    }
}

A B / C * D * E +
Euler Tour Traversal

- generic traversal of a binary tree
- the preorder, inorder, and postorder traversals are special cases of the Euler tour traversal
- “walk around” the tree
Conclusions

• In this lecture, we have seen examples of basic algorithms used in many applications and compared their complexities.

• Heuristics are the way to go if we cannot get the exact/best results with reasonable resources.

• You already know stack and queues ... Now you know trees!