Objectives of Lecture 21
Searching

- Introduce two techniques for searching for an element in a collection;
- Learn sequential search algorithm;
- Learn the binary search algorithm for ordered collections.
- Learn how to evaluate the complexity of an algorithm and compare between algorithms.

Outline of Lecture 21
Searching

- Review the simple array examples
- Sequential search approach
- Complexity of sequential search
- Binary search approach
- Complexity of binary search
- Compare sequential search and binary search

Array Example

// Find the largest element in an array of ints
int markArray[] = {50, 37, 71, 99, 63};
int max;
index = 0;
max = markArray[index];
for (index = 1; index < markArray.length; index++)
if (markArray[index] > max)
max = markArray[index];
System.out.println(max);

Array Example2

// Find the index of the largest element in an array of ints
int markArray[] = {50, 37, 71, 99, 63};
int indexOfMax;
index = 0;
indexOfMax = 0;
for (index = 1; index < markArray.length; index++)
if (markArray[index] > markArray[indexOfMax])
indexOfMax = index;
System.out.println(indexOfMax);
Outline of Lecture 21

- Review the simple array examples
- Sequential search approach
- Complexity of sequential search
- Binary search approach
- Complexity of binary search
- Compare sequential search and binary search

The Search Problem

- Given a container, find the index of a particular element, called the key.
- Technique applies for vectors, arrays, files, etc.
- Applications: information retrieval, database querying, etc.

Collection

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>50</td>
<td>10</td>
<td>95</td>
<td>75</td>
<td>30</td>
<td>70</td>
<td>55</td>
<td>60</td>
<td>80</td>
</tr>
</tbody>
</table>

Sequential Search

- Compare the key to each element in turn, until the correct element is found, and return its index.

```
/* a sequential search code (first tentative) */
public static int sequential_search( int data[], int key ) {
  boolean found = false;
  int index = 0;
  while ( !found) {
    if ( key == data[index] )
      found = true;
    else
      index = index + 1;
  }
  return index;
}
```

Sequential Search Code

Compare all elements of the collection until we find the key.

```
public static int sequential_search( int data[], int key ) {
  boolean found = false;
  int index = 0;
  while ( !found) {
    if ( key == data[index] )
      found = true;
    else
      index = index + 1;
  }
  return index;
}
```

Element not found

- We must take into account that the key we are searching for may not be in the array.
- In this case we must return a special index, say -1.

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>35</td>
<td>35</td>
<td>35</td>
<td>35</td>
<td>35</td>
<td>35</td>
<td>35</td>
<td>35</td>
<td>35</td>
</tr>
</tbody>
</table>
```

Search Algorithm

```
INPUT: data: array of int; key: int;
OUTPUT: index : an int such that data[index] == key if key is in data, or -1 if key is not stored in data.

Method:
1. index = 0; found=false;
2. While ( not found and index < data.length )
   check similarity data[index] and key
   index = index + 1
3. if not found then index = -1;
```
/* a sequential search method */
public static int sequential_search( int data[], int key ) {
    boolean found = false;
    int index = 0;
    while ( !found && index < data.length ) {
        if ( key == data[index] )
            found = true;
        else
            index = index + 1;
    }
    if  (!found) index = -1;
    return index;
}

Revised Sequential Search Code

Outline of Lecture 21

• Review the simple array examples
• Sequential search approach
  • Complexity of sequential search
  • Binary search approach
  • Complexity of binary search
• Compare sequential search and binary search

Complexity Analysis

• How efficient is this algorithm?
• In general if we have an algorithm that does something with \( n \) objects, we want to express the time efficiency of the algorithm as a function of \( n \).
• Such an expression is called the time complexity of the algorithm.
• In the case of search, we can count the number of comparison operations between the key and the elements.

Worst, Best and Average cases

• In fact, we usually have multiple expressions:
  – the worst case complexity,
  – the best case complexity
  – the average case complexity.

Complexity of Sequential Search

• How many comparison operations are required for a sequential search of an \( n \)-element container?
• In the worst case \( \rightarrow n \).
• In the best case \( \rightarrow 1 \).
• In the average case:

\[
1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2} - \frac{n}{2}
\]

• In this case, we say the complexity of Search is in the order of \( n \), denoted as \( O(n) \).
• Can we improve this algorithm?
Binary Search

- If the elements are ordered, we can do better.
- Guess the middle and adjust accordingly.

The order is important.

Element not found

Element not found (con’t)

Strategy of Binary Search:
Given an ordered array of integers, and a value of integer, search for the value in the array using an approach of Divide and Conquer.

Binary Search Algorithm

```
guess = (low + high) / 2
```

```
while ( !found) {
    guess = (high+low)/2;
    if ( key == data[guess] ) found = true;
    else if (key < data[guess])  high=guess-1;
    else low = guess+1;
}
```

```
return guess;
```

Binary Search Code

/*  a binary search code of ordered array (first tentative) */
public static int binary_search( int data[], int key ) {
    boolean found = false;
    int guess; int low = 0; int high=data.length-1;
    while ( !found) {
        guess = (high+low)/2;
        if ( key == data[guess] ) found = true;
        else if (key < data[guess])  high=guess-1;
        else low = guess+1;
    }
    return guess;
}
**Binary Search Algorithm**

**INPUT:**
- data: an array of ordered integers
- key: an integer

**OUTPUT:**
- index: an integer such that `data[index] == key` if key is in data, or -1 if key is not stored in data.

**Method:**
1. lower = 0; upper = length;
2. While (not found && low <= upper):
   - index = (lower + upper) / 2;
   - check similarity `data[index]` and key
     - if similar then found, otherwise
     - if key < `data[index]`
       - upper = index-1;
     - else
       - lower = index +1;
3. If `(data[index] == key)` index = -1;

**Outline of Lecture 21**
- Review the simple array examples
- Sequential search approach
- Complexity of sequential search
- Binary search approach
- Complexity of binary search
- Compare sequential search and binary search

**Worst-case Binary Search**
- Each time we guess, we divide the list in half:
  - 10 elements, make guess 1, then
  - 5 elements, make guess 2, then
  - 2 elements, make guess 3, then
  - 1 element, make guess 4, done
- The formula is: \( \log_2(n) \) is number of times you have to divide \( n \) by 2 to get 1

**Average-case Binary Search**
- If there were 15 elements:
  - 1 element takes 1 guess
  - 2 elements take 2 guesses
  - 4 elements take 3 guesses
  - 8 elements take 4 guesses
- The average is: \( \frac{1}{1} + \frac{4}{2} + \frac{14}{4} + \frac{31}{3} \approx 3.7 \)
- The average case is about one less than the worst case, so this is: \( \log_2(n) \)
Time Complexity of Binary Search

The number of comparisons is proportional to the height of the following search tree:

```
    n
   / \
  n/  \  \n /   \   \n/     \   \n/       \   \n/         \   n
```

The height of the tree is in the order of $\log_2(n)$. Thus, the time complexity is $O(\log_2(n))$.

Outline of Lecture 21

- Review the simple array examples
- Sequential search approach
- Complexity of sequential search
- Binary search approach
- Complexity of binary search

Sequential and Binary Search

- For average and worst case sequential search, it takes: $\frac{n+1}{2}$ and $n$.
- For average and worst case binary search, it takes: $\lfloor \log_2(n) \rfloor$ and $\lceil \log_2(n) + 1 \rceil$.

<table>
<thead>
<tr>
<th>List size</th>
<th>Sequential average</th>
<th>Sequential worst</th>
<th>Binary average</th>
<th>Binary worst</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>6</td>
<td>10</td>
<td>3</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>100</td>
<td>51</td>
<td>100</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>1000</td>
<td>501</td>
<td>1000</td>
<td>9</td>
<td>10</td>
<td>55</td>
</tr>
<tr>
<td>10000</td>
<td>5001</td>
<td>10000</td>
<td>13</td>
<td>14</td>
<td>384</td>
</tr>
</tbody>
</table>