Class 07 – Searching

CSIS 3475

Data Structures and Algorithms

The Problem

- Searching is an everyday occurrence

Iterative Sequential Search of an Unsorted Array

(a) A successful search for 8

Look at 9:

9	5	8	4	7

$8 \neq 9$, so continue searching.

Look at 5:

9	**5**	8	4	7

$8 \neq 5$, so continue searching.

Look at 8:

9	5	**8**	4	7

$8 = 8$, so the search has found 8.

Iterative Sequential Search of an Unsorted Array

(b) An unsuccessful search for 6

Look at 9:

9	5	8	4	7

$6 \neq 9$, so continue searching.

Look at 5:

9	**5**	8	4	7

$6 \neq 5$, so continue searching.

Look at 8:

9	5	**8**	4	7

$6 \neq 8$, so continue searching.

Look at 4:

9	5	8	**4**	7

$6 \neq 4$, so continue searching.

Look at 7:

9	5	8	4	**7**

$6 \neq 7$, so continue searching.

No entries are left to consider, so the search ends. 6 is not in the array.

Iterative Sequential Search of an Unsorted Array

- Use a loop to search

```
public static <T> boolean searchUnsortedArrayIterative(T[] anArray, T anEntry) {
    boolean found = false;
    int index = 0;

    while (!found && (index < anArray.length)) {
        if (anEntry.equals(anArray[index]))
            found = true;
        index++;
    }

    return found;
}
```

Recursive Sequential Search of an Unsorted Array

- Pseudocode of the logic of our recursive algorithm.

Algorithm to search **a[first]** *through* **a[last]** *for* **desiredItem**

if (*there are no elements to search*)

return false

else if (desiredItem *equals* a[first])

return true else

return *the result of searching* a[first + 1] *through* a[last]

Recursive Sequential Search of an Unsorted Array

(a) A successful search for 8

Look at the first entry, 9:

9	5	8	4	7

$8 \neq 9$, so search the next subarray.

Look at the first entry, 5:

5	8	4	7

$8 \neq 5$, so search the next subarray.

Look at the first entry, 8:

8	4	7

$8 = 8$, so the search has found 8.

Recursive Sequential Search of an Unsorted Array

(b) An unsuccessful search for 6

Look at the first entry, 9:

9	5	8	4	7

$6 \neq 9$, so search the next subarray.

Look at the first entry, 5:

5	8	4	7

$6 \neq 5$, so search the next subarray.

Look at the first entry, 8:

8	4	7

$6 \neq 8$, so search the next subarray.

Look at the first entry, 4:

4	7

$6 \neq 4$, so search the next subarray.

Look at the first entry, 7:

7

$6 \neq 7$, so search an empty array.

No entries are left to consider, so the search ends. 6 is not in the array.

Recursive Sequential Search of an Unsorted Array

```java
public static <T> boolean searchUnsortedArrayRecursive(T[] anArray, T anEntry) {
    return search(anArray, 0, anArray.length - 1, anEntry);
}

/**
 * Recursively searches anArray[first] through anArray[last] for desiredItem.
 * first >= 0 and < anArray.length.
 * last >= 0 and < anArray.length.
 *
 * @param anArray
 * @param first
 * @param last
 * @param desiredItem
 * @return
 */
private static <T> boolean search(T[] anArray, int first, int last, T desiredItem) {
    boolean found = false;

    if (first > last)
        found = false; // No elements to search
    else if (desiredItem.equals(anArray[first]))
        found = true;
    else
        // go to next one
        found = search(anArray, first + 1, last, desiredItem);

    return found;
}
```

Efficiency of a Sequential Search of an Array

- The time efficiency of a sequential search of an array.
  - Best case $O(1)$
  - Worst case: $O(n)$
  - Average case: $O(n)$

Sequential Search of a Sorted Array

- Coins sorted by their mint dates

Binary Search of a Sorted Array

- Ignoring one half of the data when the data is sorted

Binary Search of a Sorted Array

- First draft of an algorithm for a binary search of an array

set mid.

Algorithm to search a[0] *through* a[n – 1] *for* desiredItem

mid = *approximate midpoint between* 0 *and* n – 1

if (desiredItem *equals* a[mid])

  return true

else if (desiredItem < a[mid])

  return *the result of searching* a[0] *through* a[mid – 1]

else if (desiredItem > a[mid])

  return *the result of searching* a[mid + 1] *through* a[n – 1]

Binary Search of a Sorted Array

- Revision of binary search algorithm as method – use recursion

Algorithm **binarySearch(a, first, last, desiredItem)**

mid = *approximate midpoint between* first *and* last

if (desiredItem *equals* a[mid])

  return true

else if (desiredItem < a[mid])

  return binarySearch(a, first, mid − 1, desiredItem)

else if (desiredItem > a[mid])

  return binarySearch(a, mid + 1, last, desiredItem)

Binary Search of a Sorted Array

- Refine the logic a bit, get a more complete algorithm – calculate midpoint

Algorithm **binarySearch(a, first, last, desiredItem)**

mid = (first + last) / 2 // *Approximate midpoint*

if (first > last)

  return false

else if (desiredItem *equals* a[mid])

  return true

else if (desiredItem < a[mid])

  return binarySearch(a, first, mid − 1, desiredItem)

else // desiredItem > a[mid]

  return binarySearch(a, mid + 1, last, desiredItem)

Binary Search of a Sorted Array

(a) A successful search for 8

Look at the middle entry, 10:

2	4	5	7	8	**10**	12	15	18	21	24	26
0	1	2	3	4	5	6	7	8	9	10	11

$8 < 10$, so search the left half of the array.

Look at the middle entry, 5:

2	4	**5**	7	8
0	1	2	3	4

$8 > 5$, so search the right half of the array.

Look at the middle entry, 7:

7	8
3	4

$8 > 7$, so search the right half of the array.

Look at the middle entry, 8:

8
4

$8 = 8$, so the search ends. 8 is in the array.

Binary Search of a Sorted Array

(b) An unsuccessful search for 16

Look at the middle entry, 10:

2	4	5	7	8	**10**	12	15	18	21	24	26
0	1	2	3	4	5	6	7	8	9	10	11

16 > 10, so search the right half of the array.

Look at the middle entry, 18:

12	15	**18**	21	24	26
6	7	8	9	10	11

16 < 18, so search the left half of the array.

Look at the middle entry, 12:

12	15
6	7

16 > 12, so search the right half of the array.

Look at the middle entry, 15:

15
7

16 > 15, so search the right half of the array.

The next subarray is empty, so the search ends. 16 is not in the array.

Binary Search of a Sorted Array

```java
public static <T extends Comparable<T>> boolean searchSortedArrayRecursive(T[] anArray, T anEntry) {
    return binarySearch(anArray, 0, anArray.length - 1, anEntry);
}

/**
 * Searches anArray[first] through anArray[last] for desiredItem.
 * @param anArray
 * @param first first >= 0 and < anArray.length.
 * @param last last >= 0 and < anArray.length
 * @param desiredItem
 * @return
 */
private static <T extends Comparable<T>> boolean binarySearch(T[] anArray, int first, int last, T desiredItem) {
    boolean found;

    // calculate midpoint
    int mid = first + (last - first) / 2;

    // if we are at the end we didn't find it
    // if we found it, exit
    // otherwise divide and call recursively depending on comparison
    if (first > last)
            found = false;
    else if (desiredItem.equals(anArray[mid]))
            found = true;
    else if (desiredItem.compareTo(anArray[mid]) < 0)
            found = binarySearch(anArray, first, mid - 1, desiredItem);
    else
            found = binarySearch(anArray, mid + 1, last, desiredItem);

    return found;
}
```

prevent integer overflow -

Java Class Library: The Method `binarySearch`

- Static method `binarySearch` specification
  - returns position or where it should be inserted
- See ArraySearchDemo

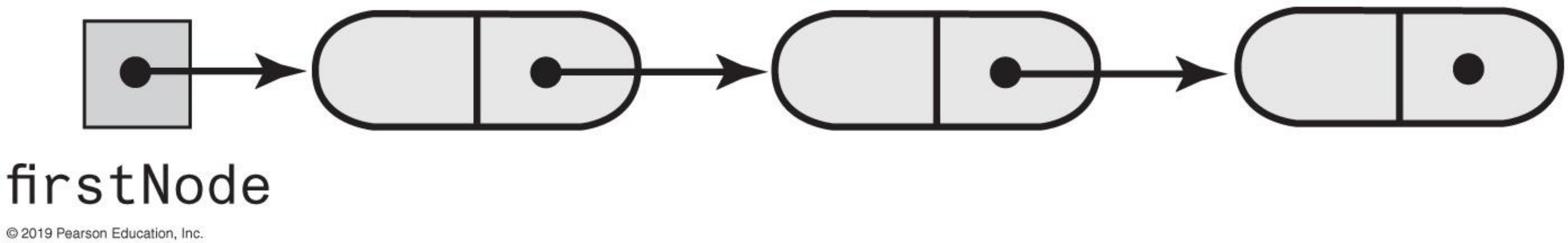
```
/** Searches an entire array for a given item.
 @param array    An array sorted in ascending order.
 @param desiredItem The item to be found in the array.
 @return Index of the array entry that equals desiredItem;
                    otherwise returns –belongsAt – 1, where belongsAt is
 the index of the array element that should contain desiredItem. */
public static int binarySearch(type[] array, type desiredItem);
```

Efficiency of a Binary Search of an Array

- The time efficiency of a binary search of an array
  - Best case: $O(1)$
  - Worst case: $O(\log n)$
  - Average case: $O(\log n)$

Iterative Sequential Search of an Unsorted Chain

- A chain of linked nodes that contain the entries in a list



Iterative Sequential Search of an Unsorted Chain

```java
/**
 * Searches an unsorted chain for anEntry iteratively.
 * @param anEntry
 * @return
 */
public boolean searchUnsortedChainIterative(T anEntry) {
     boolean found = false;
     Node<T> currentNode = firstNode;
     while (!found && (currentNode != null)) {
          if (anEntry.equals(currentNode.getData()))
               found = true;
          else
               currentNode = currentNode.getNextNode();
     }
     return found;
}
```

Recursive Sequential Search of an Unsorted Chain

```
 * Searches an unsorted chain for anEntry by calling a recursive private method.
 * @param anEntry
 * @return
 */
public boolean searchUnsortedChainRecursive(T anEntry) {
    return search(firstNode, anEntry);
}

/**
 * Recursively searches a chain of nodes sequentially for desiredItem,
 *  beginning with the node that currentNode references.
 * @param currentNode
 * @param desiredItem
 * @return
 */
private boolean search(Node<T> currentNode, T desiredItem) {
    boolean found;
    if (currentNode == null)
        found = false;
    else if (desiredItem.equals(currentNode.getData()))
        found = true;
    else
        found = search(currentNode.getNextNode(), desiredItem);
    return found;
}
```

Iterative Sequential Search of a Sorted Chain

- Use of compareTo() instead of equals()

```java
/**
 * Searches a sorted chain for anEntry sequentially and iteratively.
 * @param anEntry
 * @return
 */
public boolean searchSortedChainIterative(T anEntry) {
    Node<T> currentNode = firstNode;
    while ((currentNode != null) && (anEntry.compareTo(currentNode.getData()) > 0)) {
        currentNode = currentNode.getNextNode();
    }

    return (currentNode != null) && anEntry.equals(currentNode.getData());
}
```

Binary Search of a Sorted Chain

- First find middle of the chain:
  - You must traverse the whole chain
  - Then traverse one of the halves to find the middle of that half
- Conclusion
  - Hard to implement
  - Less efficient than sequential search

Choosing between Iterative Search and Recursive Search

- The time efficiency of searching, expressed in Big Oh notation

Operation	Best Case	Average Case	Linked
`Sequential Search(unsorted data)`	O(1)	O(n)	O(n)
`Sequential Search(sorted data)`	O(1)	O(n)	O(n)
`Binary Search (sorted array)`	O(1)	O(log n)	O(log n)

Choosing between Iterative Search and Recursive Search

- Iterative Searches
  - Can save some time and space
- Recursive Searches
  - Will not require much additional space for the recursive calls
    - Generally, these are tail recursive, equivalent to iterative
  - Coding binary search recursively is easier