

BACS2063 Data Structures and Algorithms

# Algorithms for Searching

Chapter 8a

## Chapter 8 Part 1

# Algorithms for Searching

# Learning Outcomes

At the end of this part, you should be able to

- Implement sequential search and binary search algorithms
- Assess the time efficiency of sequential search and binary search algorithms

# The Problem



Fig. 16-1 Searching is an every day occurrence.

# Searching: Introduction

- Searching is a common task in computing
- Given a table of records or list of entries, searching is the process of locating a record whose key is equal to the target key value.
- A search algorithm accepts an argument **k** and tries to find a record whose key is **k**. A successful search is called a retrieval.
- Searching is time consuming, therefore different techniques are deployed to different data arrangement to achieve better performance.

# Searching Techniques

*Searching Techniques:*

1. **Sequential (linear) search** - for unsorted & sorted arrays, as well as unsorted & unsorted linked lists.
2. **Binary search** - for sorted arrays only.

# Sequential Search

- The sequential search algorithm searches through a list sequentially from the beginning, searching for a target value.
- a.k.a. *linear search*, *serial search*
- The method **contains** in **ArrayList.java** (from Chapter 4) and **LinkedList.java** (from Chapter 5) implements the sequential search algorithm.

# Sequential Search - *Unsorted* Array (1)

- *Iterative* sequential search algorithm

```
public boolean contains (T anEntry) {  
    boolean found = false;  
    for (int index = 0;  
        !found && (index < length); index++){  
        if (anEntry.equals(list[index]))  
            found = true ;  
    }  
    return found;  
}
```



# Sequential Search - *Unsorted* Array (2)

**(a) A 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.

Fig. 16-2 An iterative sequential search of an array that  
(a) finds its target

# Sequential Search - *Unsorted* Array (3)

## (b) A 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.

Fig. 16-2 An iterative sequential search of an array that (b) does not find its target

# Recursive Sequential Search on *Unsorted Array*: Algorithm

- Pseudocode for a recursive algorithm to search an array.

```
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 on *Unsorted Array*: Method

```
public boolean contains(T anEntry) {  
    return search(0, length - 1, anEntry);  
}  
  
private boolean search(int first, int last, T desiredItem){  
    boolean found;  
  
    if (first > last)  
        found = false; // no elements to search  
    else if (desiredItem.equals(list[first]))  
        found = true;  
    else  
        found = search(first + 1, last, desiredItem);  
    return found;  
}
```

# Recursive Sequential Search on Unsorted Array Example: Target Found

## (a) A 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 on Unsorted Array Example: Target Not Found

## (b) A 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.

# Recursive Sequential Search on Unsorted Array Example: Target Not Found (cont'd)

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.

# Efficiency of a Sequential Search

- Best case  $O(1)$ 
  - Locate desired item first
- Worst case  $O(n)$ 
  - Must look at all the items
- Average case  $O(n)$ 
  - Must look at half the items
  - $O(n/2)$  is still  $O(n)$
- The time it takes (on average, and worst case) is *linear*, or  $O(N)$ , in the length of the list.



# Searching a **Sorted** Array

- A sequential search can be more efficient if the data is sorted



Fig. 16-4 Coins sorted by their mint dates.

- It is also more useful for the sequential search method to *return the location of the target* in the array.

# Binary Search

- Can only be applied to *sorted* arrays.
- Repeatedly divides the array in half until the element is found or there is nothing left to search



Fig. 16-5 Ignoring one-half of the data when the data is sorted.

# Binary Search: General Idea

Divide the array in half and compare the target with the element at the array's mid-point:

- If they match, the target is found;
- Otherwise, if the target is less than the middle element, search the left subarray;
- Otherwise, search the right subarray.

# Recursive Binary Search Algorithm

```
Algorithm binarySearch (a, first, last, desiredItem)
    mid = (first + last) / 2
    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)
```

# Recursive Binary Search Method

```
private boolean binarySearch(int first, int last, T desiredItem) {  
    boolean found;  
    int mid = (first + last)/2;  
    if (first > last)  
        found = false;  
    else if (desiredItem.equals(list[mid]))  
        found = true;  
    else if (desiredItem.compareTo(list[mid]) < 0)  
        found = binarySearch(first, mid - 1, desiredItem);  
    else  
        found = binarySearch(mid + 1, last, desiredItem);  
    return found;  
}
```

- Sample code in Chapter8\searching folder:
  - [SortedArrayList.java](#)
  - [TestBinarySearch.java](#)

# Binary Search Example: Target Found

## (a) A search for 8

Look at the middle entry, 10:

2	4	5	7	8	<b>10</b>	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	<b>5</b>	7	8
0	1	2	3	4

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

# Binary Search Example: Target Found (ctd)

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 Example: Target Not Found

## (b) A search for 16

Look at the middle entry, 10:

2	4	5	7	8	<b>10</b>	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	<b>18</b>	21	24	26
6	7	8	9	10	11

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



# Binary Search Example: Target Not Found (ctd)

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.

## Java Class Library: The Method **binarySearch**

- The class `Arrays` in `java.util` defines versions of a static method with following specification:

```
public static int binarySearch(type[] array,  
    type desiredItem) ;
```

# Efficiency of a Binary Search

- Best case  $O(1)$  Locate desired item first
- Worst case  $O(\log n)$  Must look at all the items
- Average case  $O(\log n)$
- In the recursive binary algorithm, with each comparison we halve the size of the list under consideration. Since  $n$  is the size of the array, the consecutive sizes of the arrays in the succeeding comparisons will be

$$\frac{n}{2}, \frac{n}{4}, \frac{n}{8}, \frac{n}{16}, \dots \quad \text{OR} \quad \frac{n}{2}, \frac{n}{2^2}, \frac{n}{2^3}, \frac{n}{2^4}, \dots$$

# Iterative Sequential Search of an Unsorted Chain

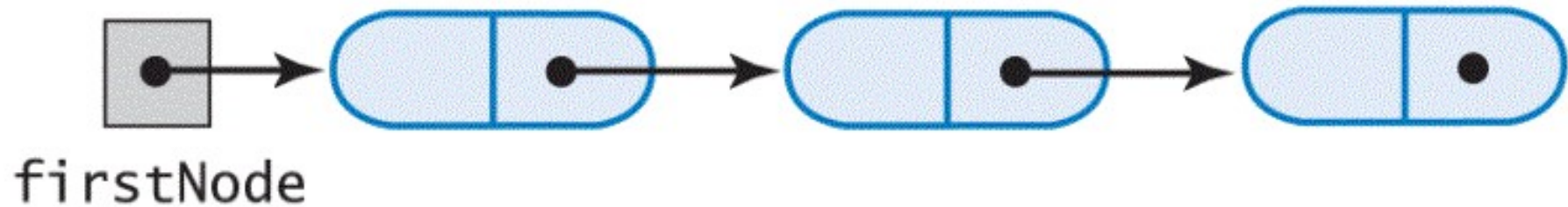


Fig. 16-7 A chain of linked nodes that contain the entries in a list.

# Sequential Search of an Unsorted Chain

- View iterative sequential search
- View recursive sequential search
  - Note method **contains** which calls it
  - Formal parameter **currentNode** in search method
    - The recursive search method initializes the parameter **currentNode** to **firstNode** (formal parameter) in the method **contains**.
    - The iterative method initializes local variable **currentNode** to **firstNode**.

# *Iterative* Sequential Search of an Unsorted Chain

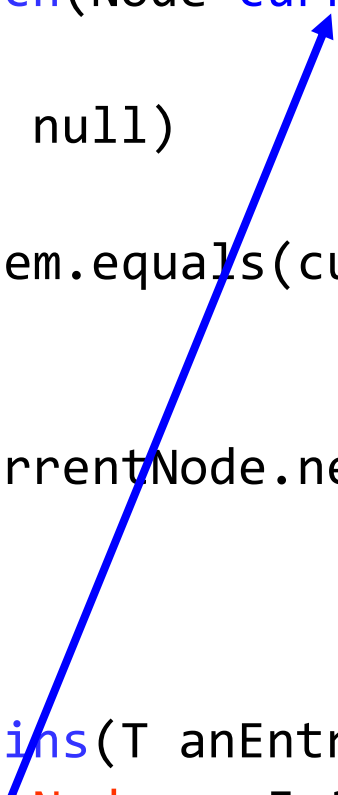
```
public boolean contains(T anEntry) {  
    boolean found = false;  
    Node currentNode = firstNode;  
  
    while (!found && (currentNode != null)) {  
        if (anEntry.equals(currentNode.data))  
            found = true;  
        else  
            currentNode = currentNode.next;  
    } // end while  
    return found;  
} // end contains
```

Local reference  
variable **currentNode**  
moves from a node to other.

# *Recursive* Sequential Search of an Unsorted Chain

```
private boolean search(Node currentNode, T desiredItem) {
    boolean found;
    if (currentNode == null)
        found = false;
    else if (desiredItem.equals(currentNode.data))
        found = true;
    else
        found = search(currentNode.next, desiredItem);
    return found;
} // end search

public boolean contains(T anEntry){
    return search(firstNode, anEntry);
} // end contains
```



# Sequential Search of a Sorted Chain

```
public boolean contains(T anEntry){  
    Node currentNode = firstNode;  
    while ((currentNode != null) &&  
        (anEntry.compareTo(currentNode.data) > 0)) {  
        currentNode = currentNode.next;  
    } // end while  
    return (currentNode != null) &&  
        anEntry.equals(currentNode.data);  
} // end contains
```



# Efficiency of a Sequential Search of a Chain

- Best case  $O(1)$ 
  - Locate desired item first
- Worst case  $O(n)$ 
  - Must look at all the items
- Average case  $O(n)$ 
  - Must look at half the items
  - $O(n/2)$  is just  $O(n)$

# Binary search of a Sorted Chain

- Binary search works perfectly if lists are implemented with arrays. But, it is **impractical** with linked implementations of lists.
- With a linked implementation we cannot find the middle of the list in constant time: it takes linear time and this takes away the speedup we get from binary search.

# Choosing between a sequential search and a binary search

- Both of search algorithms are applicable to array.
- If the array size is small, use sequential search.
- If the array is large and already sorted, a binary search is much faster than a sequential search.
- A binary search of a chain of linked nodes is impractical.

# Choosing a Search Method

	Best case	Average case	Worst case
Sequential search (unsorted data)	$O(1)$	$O(n)$	$O(n)$
Sequential search (sorted data)	$O(1)$	$O(n)$	$O(n)$
Binary search (sorted array)	$O(1)$	$O(\log n)$	$O(\log n)$

Fig. 16-8 The time efficiency of searching, expressed in Big Oh notation

# Choosing between an iterative search and a recursive search

- The recursive sequential search is tail recursion, it can save some time and space using the iterative version of the search.
- The binary search is fast, so using recursion will not require much additional space for the recursive calls.
- For coding the binary search recursively is easier than coding it iteratively.

# Exercise



**Binary search**

**vs**

**Sequential search**

# Exercise



Given the array:

5	10	15	20	25	30	35	40	45	50
---	----	----	----	----	----	----	----	----	----

- How will binary search for the target value 50.
- What are the best-case and worst-case Big-O running times of the binary search algorithm.

$\text{mid}_1 = (0 + 9) / 2 = 4$ . The value at index 4 is 25.

Since the target is **50** > 25, search the right subarray [5..9].

$\text{mid}_2 = (5 + 9) / 2 = 7$ . The value at index 7 is 40.

Since the target is **50** > 40, search the right subarray [8..9].

$\text{mid}_3 = (8 + 9) / 2 = 8$ . The value at index 8 is 45.

Since the target is **50** > 45, search the right subarray [9..9].

$\text{mid}_4 = (9 + 9) / 2 = 9$ . The value at index 9 is 50.

Therefore, the target value has been located at index 9.

- **Best case:** target is at the array's mid. Only 1 comparison is required, i.e.  $O(1)$ .
- **Worst case:** when the target is not found and its value is either smaller than the smallest value in the array or larger than the largest value in the array. As each comparison reduces the number of elements to be searched to half of the existing number of (sub)array elements, it gives running time i.e.  $O(\log_2 n)$ .

# Review of Learning Outcomes

You should now be able to

- Implement sequential search and binary search algorithms
- Assess the time efficiency of sequential search and binary search algorithms



# To Do

- Review the slides and source code for this chapter.
- Read up the relevant portions of the recommended text.
- Do the practical questions for this chapter.

# References

- Carrano, F. M., 2019, Data Structures and Abstractions with Java, 5th edn, Pearson
- Liang, Y.D., 2018. Introduction to Java Programming and Data Structures.11th ed.United Kingdom:Pearson