University of Kelaniya Sri Lanka

COSC 21063

Searching

Lecturer : Ms.D.M.L.Maheshika Dissanayake

Structure of Lesson

- Searching
  - Keys
  - Analysis
- Sequential Search
- Binary Search
  - Definition
  - Operations
  - Implementation

Learning Outcomes

By the end of this lesson you should be able to

- Understand the concept of searching
- Implement the sequential and binary search

Searching

Searching is one of the most important applications of computers.

Examples :

- Given an account number, get the transactions occurring in that account.
- Given an employee name, get the personnel information of that employee.

Keys

We are given one piece of information, which we shall call a **key**, and we are asked to find a record that contains other information associated with **key**.

- There may be more than one record with the same key
- There may be no record at all with a given key.

Analysis

- Searching for the keys that locate records is often the most time consuming action
- Therefore the way the records are arranged and the choice of method used for searching can make a substantial difference in the program's performance.
- We will look at how **much work is done** by each of the algorithms we develop.
- Counting the number of times that one key is compared with another gives an excellent measure of the total amount of work that the algorithm will do and the total amount of computer time it will require when it is run.

External and Internal searching

External Searching

- If there are many records, perhaps each one quite large, then it will be necessary to store the records in files on disk or tape, external to the computer memory.

Internal Searching

- The records to be searched are stored entirely within the computer memory.

Parameters

Each searching function we write will have two parameters:

1. target
2. the list being searched

The return value will be the location of the target.

- If the search is successful, then the function returns the position in the list where the target was found
- If the search is unsuccessful, then the function returns –1.

Parameters

When searching we need to compare two keys to determine which comes first, or whether or not they are equal.

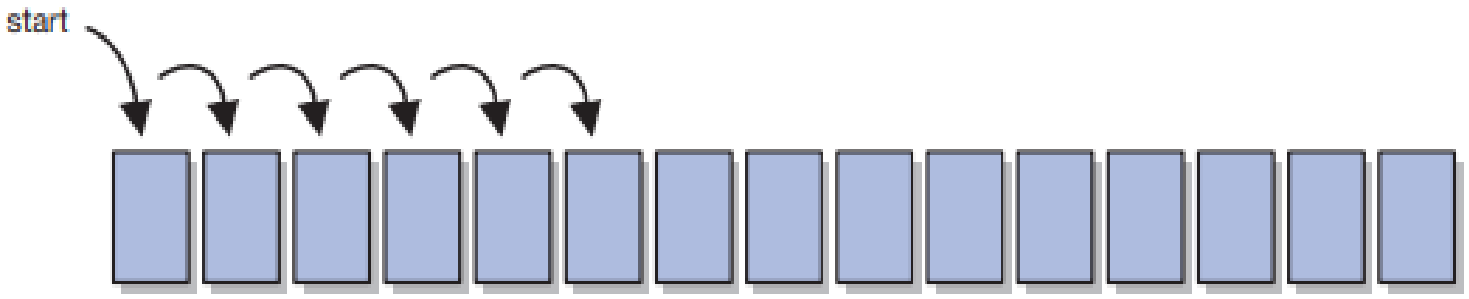
If the keys are numbers, we use ‘<’ and ‘==’

If the keys are character strings, we have to use standard string functions.

We will code our algorithms in such a way that will cover all the possibilities at once. So that we can change our code from processing numbers to processing strings.

Sequential Search

- The simplest way to do a search is to begin at one end of the list and scan down it until the desired target is found or the other end is reached.
- This is known as the sequential search



Sequential Search

- Contiguous version
- Pre: The contiguous list has been created.
- Post: If an entry in list has key equal to target, then the function returns the location of the first such entry (success). Otherwise the function returns -1 (failure).

Sequential Search

```
public int SequentialSearch(int key)
{
   for(int i=0; i <ListSize(); i++){
      if(key == ListEntry[i])
         return i;
   }
   return -1;
}
```

Analysis: Sequential Search

- **Use the count of key comparisons as our measure of the work done.**
- If the search is unsuccessful, then the answer is *n* key comparisons
- The best performance for a successful search is 1 comparison
- The worst is, *n* comparisons

Analysis

- For a successful sequential search simply add the number needed for all successful searches, and divide by *n*.

  The result is (1+2+3+4+…….+n) / n

  here 1+2+3+……+n = n(n+1)/2

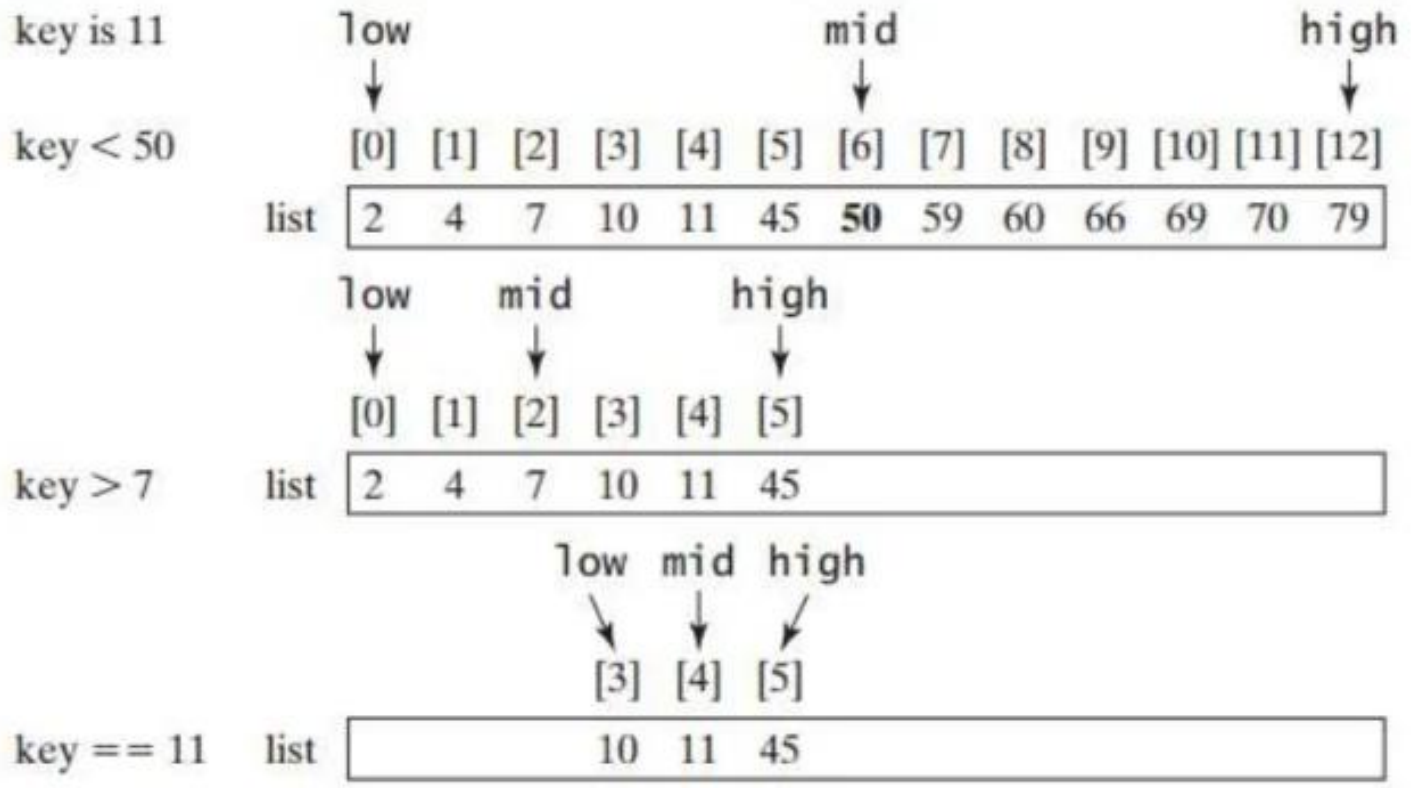
- So the average number of key comparisons done by sequential search in the successful case is

  n(n+1)/2n = (n+1)/2

Binary Search

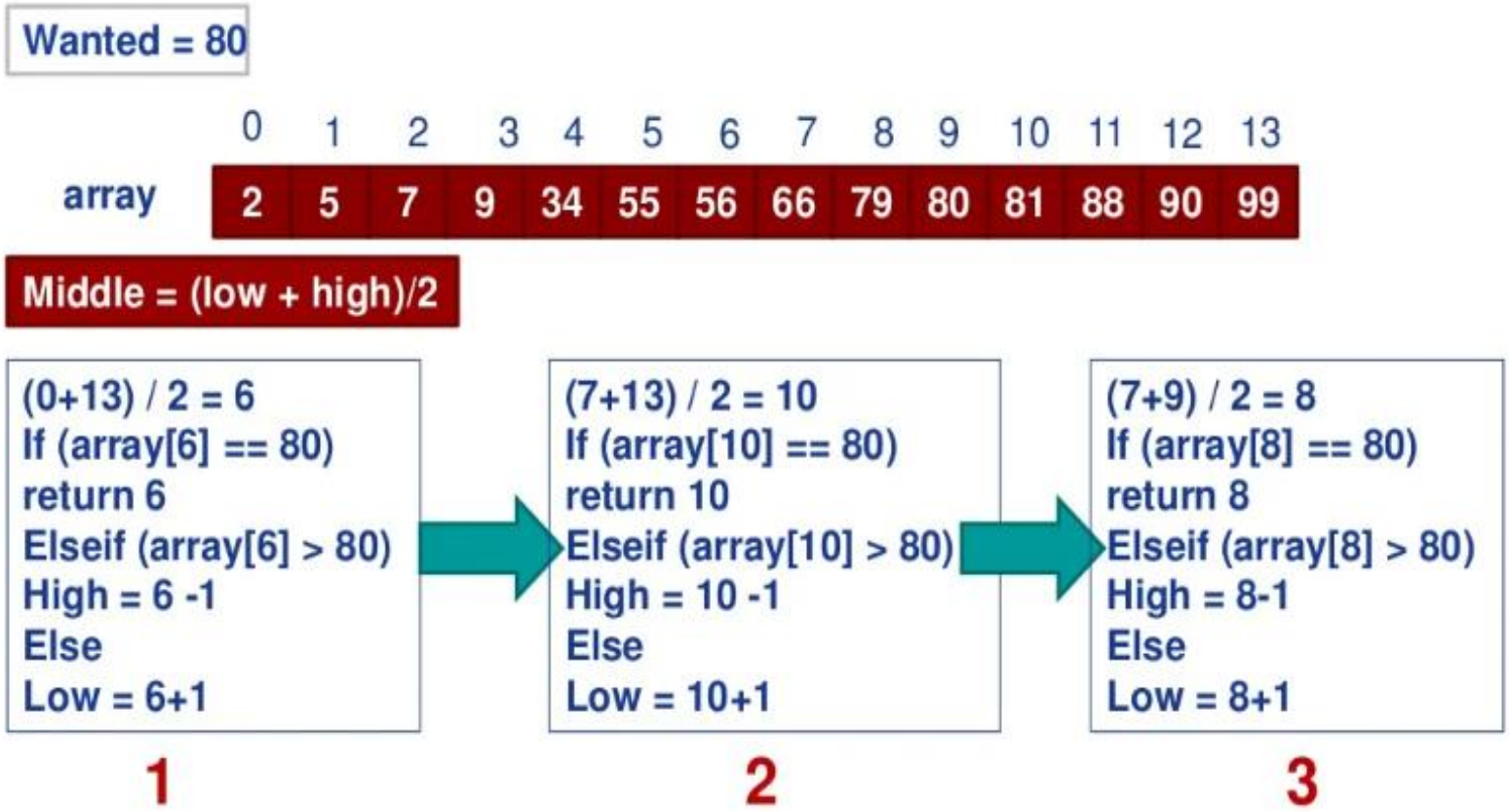
- Sequential search is easy to write and efficient for short list.
- Another way to search the target key in a list is to compare the target key with one in the center of the list and restrict our attention to only the first or second half of the list.
- In this way at each step we reduce the length of the list to be searched by half.
- This method is called **binary search**.
- This approach requires that the list to be ordered by the key.
- Since binary search requires jumping back and forth from one end of the list to the middle, it requires an implementation of the list in which this random access is rapid.

Binary Search

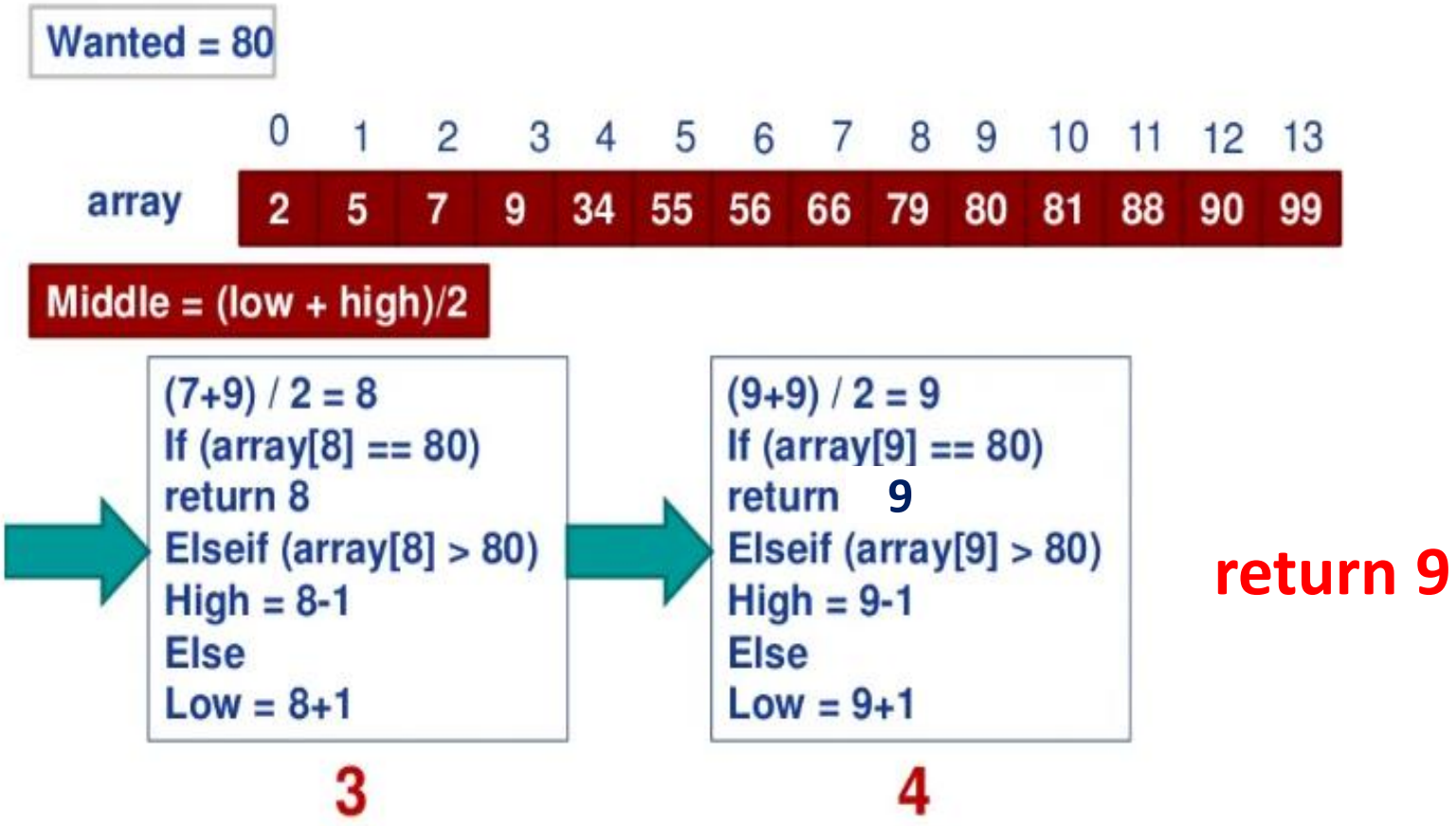


Binary search eliminates half of the list from further consideration after each

Binary Search



Binary Search



Binary Search

- Implement the recursive version of the Binary search
- Implement the iterative version of the Binary search

Binary Search - Iterative version

```
public  int binarySearch(int key) {
        int low = 0;
        int high = ListSize() - 1;

        while (low <= high) {
            int mid = (low + high) / 2;
            if (ListEntry[mid] < key) {
                low = mid + 1;
            } else if (ListEntry[mid] > key) {
                high = mid - 1;
            } else {
                return mid;
            }
        }
        return -1;
    }
```

Binary Search

The division of the list into sublists

<target	?	>= target

First part – entries are strictly less than target

Last part – entries greater than or equal to target

In this way when middle part of the list is reduced to size 1 and hits the target, it will be guaranteed to be the first occurrence of the target if it appears more than once in the list.

Binary Search - Recursive version

If the target value is not found, return its negative insertion point.

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103

```
int index  = binarySearch(data, 42);  // 10
int index2 = binarySearch(data, 66);  // -14
```

Binary Search - Recursive version

Using Array

```
public static int binarySearch(int[] a, int target) {
    return binarySearch(a, target, 0, a.length - 1);
}

private static int binarySearch(int[] a, int target,
                                int min, int max) {
    if (min > max) {
        return -1;
    } else {
        int mid = (min + max) / 2;
        if (a[mid] < target) {
            return binarySearch(a, target, mid + 1, max);
        } else if (a[mid] > target) {
            return binarySearch(a, target, min, mid - 1);
        } else {
            return mid;
        }
    }
}
```

Binary Search - Recursive version

```
public int binarySearch2(int target) {
        return binarySearch2(target,0,ListSize()-1);
    }

private int binarySearch2(int target,int min, int max) {
        if (min > max) {
            return -1;
        } else {
            int mid = (min + max) / 2;
            if (ListEntry[mid] < target) {
                return binarySearch2(target, mid + 1, max);
            } else if (ListEntry[mid] > target) {
                return binarySearch2(target, min, mid - 1);
            } else {
                return mid;
            } }
    }
```

Summary of main points

- Definition of searching
- Implementation of binary search
- Comparison of recursive and iterative implementations