Chapter 5 - Searching

- Sequential Search
  - In an unordered list
  - In an ordered list
- Binary Search
  - Forgetful Version
  - Recognizing Equality
  - Comparison Tree
- Linked List vs. Contiguous List

Searching

- We are given a list of records. Each record is associated with a key.
- We are given one key (target), and are asked to search the list to find the record(s) whose key is the same as the target.
- May be more than one record with the same key.
- May be no record with a given key.

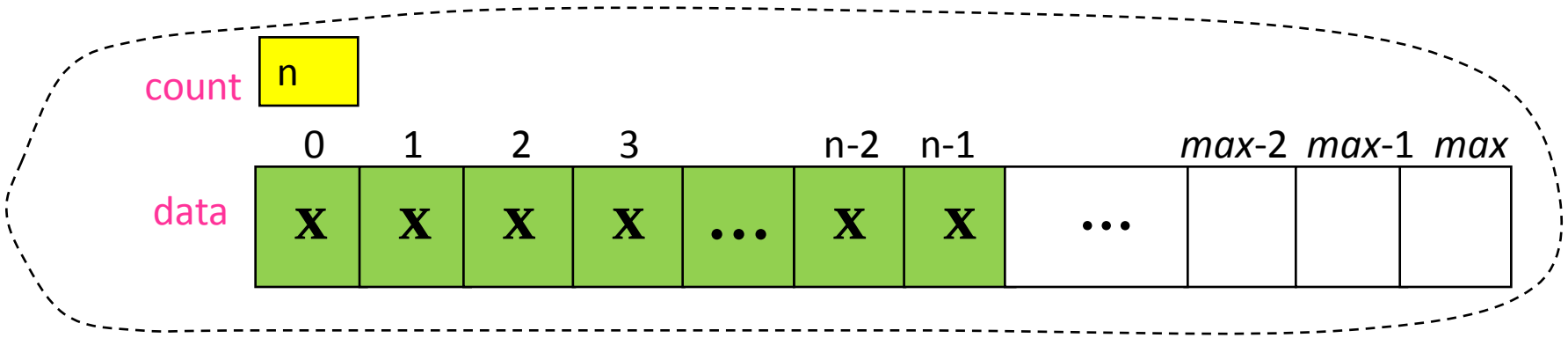
Searching

- We often asked how many times one key is compared with another during a search.
- Internal searching: all records are kept in the high-speed memory.
- External searching: most of the records are kept in disk files.

Sequential Search

Begin at one end of the list and scan down until the desired key is found or the other end is reached

Contiguous List



List

count <integer>

data <array of <DataType> >

End List

General DataType:
DataType
key <KeyType>
field1 <…>
field2 <…>
…
fieldn <…>
End DataType

Sequential Search in an unordered list

ErrorCode **Sequential_Search_1**(val target <KeyType>,
ref position <int>)

Searches the list to find the record whose key is the same as the target

Post If an entry in the list has key equal to target, then return *found* and position locates such an entry within the list. Otherwise return *notFound* and position becomes invalid.

1. position = 0
2. loop (position < number of elements in the list)
    1. if ($data_{position}$.key = target)
        1. return *found*
    2. position = position + 1
3. return *notFound*

End Sequential_Search_1

Sequential Search in an unordered list (cont.)

The number of comparisons of keys done in sequential search of a list of length n is

- Unsuccessful search: n comparisons.
- Successful search, best case: 1 comparison.
- Successful search, worst case: n comparisons.
- Successful search, average case: $\frac{1}{2}(n+1)$ comparisons.

Sequential Search in an ordered list

DEFINITION An ***ordered list*** is a list in which each entry contains a key, such that the keys are in order. That is, if entry i comes before entry j in the list, then the key of entry i is less than or equal to the key of entry j.

Sequential Search in an ordered list

<ErrorCode> **Sequential_Search_2**(val target <KeyType>,
ref position <int>)

Searches the list to find the record whose key is the same as the target

Pre list is an ordered list.

Post If an entry in the list has key equal to target, then return *found* and position locates such an entry within the list. Otherwise return *notFound* and position becomes invalid.

1. position = 0
2. loop ((position <number of elements in the list) AND
   ($data_{position}$.key <= target))
   1. if ($data_{position}$.key = target)
      1. return *found*
   2. position = position + 1
3. return *notFound*

End Sequential_Search2

Binary Search (only in ordered list)

IDEA: In searching an ordered list,

- First compare the target to the key in the center of the list.
- If it is smaller, restrict the search to the left half;
- Otherwise restrict the search to the right half, and repeat.
- In this way, at each step we reduce the length of the list to be searched by half.

Binary Search (cont.)

- The idea of binary search is very simple.
- But it is exceedingly easy to program it incorrectly.
- The method dates back at least to 1946, but the first version free of errors and unnecessary restrictions seems to have appeared only in 1962!
- One study showed that about 90% of professional programmers fail to code binary search correctly, even after working on it for a full hour.
- Another study found correct solutions in only five out of twenty textbooks.

Binary Search Algorithm

- Keep two indices, top and bottom, that will bracket the part of the list still to be searched.
- The target key, provided it is present, will be found between the indices bottom and top, inclusive.
- Initialization: Set bottom = 0; top = the_list.Size() – 1;
- Compare target with the Record at the midpoint,

```
mid = (bottom+ top) / 2;      // (bottom <= mid < top)
```

- Change the appropriate index top or bottom to restrict the search to the appropriate half of the list.

Binary Search Algorithm (cont.)

- Loop terminates when top <= bottom, if it has not terminated earlier by finding the target.
- Make progress toward termination by ensuring that the number of items remaining to be searched, top – bottom + 1, strictly decreases at each iteration of the process.

The Forgetful Version of Binary Search

Forget the possibility that the Key target might be found quickly and continue, whether target has been found or not, to subdivide the list until what remains has length 1.

ErrorCode **Binary_Search_1** (val target <KeyType>,
ref position <int>)

Searches the list to find the record whose key is the same as the target

Pre list is an ordered list.

Post If an entry in the list has key equal to target, then return *found* and position locates such an entry within the list. Otherwise return *notFound* and position becomes invalid.

The Forgetful Version of Binary Search (cont.)

ErrorCode **Binary_Search_1** (val target <KeyType>,
ref position <int>)

Searches the list to find the record whose key is the same as the target

1. bottom = 0
2. top = number of elements in the list – 1
3. **loop** (bottom < top)
    1. mid = (bottom + top) / 2
    2. **if** (target > $data_{mid}$)
        1. bottom = mid + 1
    3. **else**
        1. top = mid
4. **if** (top < bottom)
    1. return *notFound*
6. **else**
    1. position = bottom
    2. **if** (target = $data_{position}$)
        1. return *found*
    3. **else**
        1. return *notFound*

End Binary_Search_1

The Forgetful Version of Binary Search (cont.)

- The division of the list into sublists is described in the following diagram:

- Only entries strictly less than target appear in the first part of the list, but the last part contains entries greater than or equal to target.

- When the middle part of the list is reduced to size 1, it will be guaranteed to be the first occurrence of the target if it appears more than once in the list.

The Forgetful Version of Binary Search (cont.)

- We must prove carefully that the search makes progress towards termination.

- This requires checking the calculations with indices to make sure that the size of the remaining sublist strictly decreases at each iteration.

- It is also necessary to check that the comparison of keys corresponds to the division into sublists in the above diagram.

Recognizing Equality in Binary Search

Check at each stage to see if the target has been found.

ErrorCode **Binary_Search_2** (val target <KeyType>,
ref position <int>)

Searches the list to find the record whose key is the same as the target

Pre list is an ordered list.

Post If an entry in the list has key equal to target, then return *found* and position locates such an entry within the list. Otherwise return *notFound* and position becomes invalid.

Recognizing Equality in Binary Search (cont.)

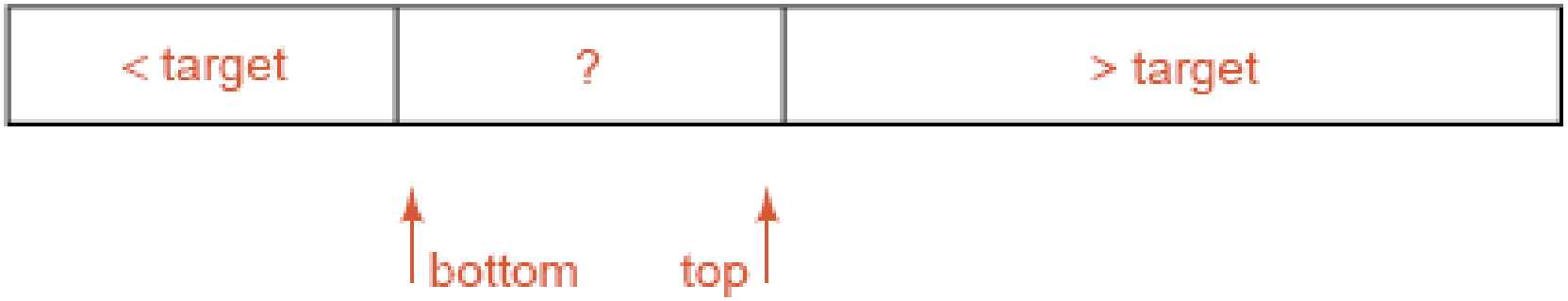
ErrorCode **Binary_Search_2** (val target <KeyType>,
ref position <int>)

1. bottom = 0
2. top = number of elements in the list -1
3. **loop** (bottom <= top)
    1. mid = (bottom + top) /2
    2. **if** (target = $data_{mid}$)
        1. position = mid
        2. **return** found
    3. **if** (target > $data_{mid}$)
        1. bottom = mid + 1
    4. **else**
        1. top = mid -1

End Binary_Search_2

Recognizing Equality in Binary Search (cont.)

- The division of the list into sublists is described in the following diagram:



- The first part of the list contains only entries strictly less than target, and the last part contains entries strictly greater than target.

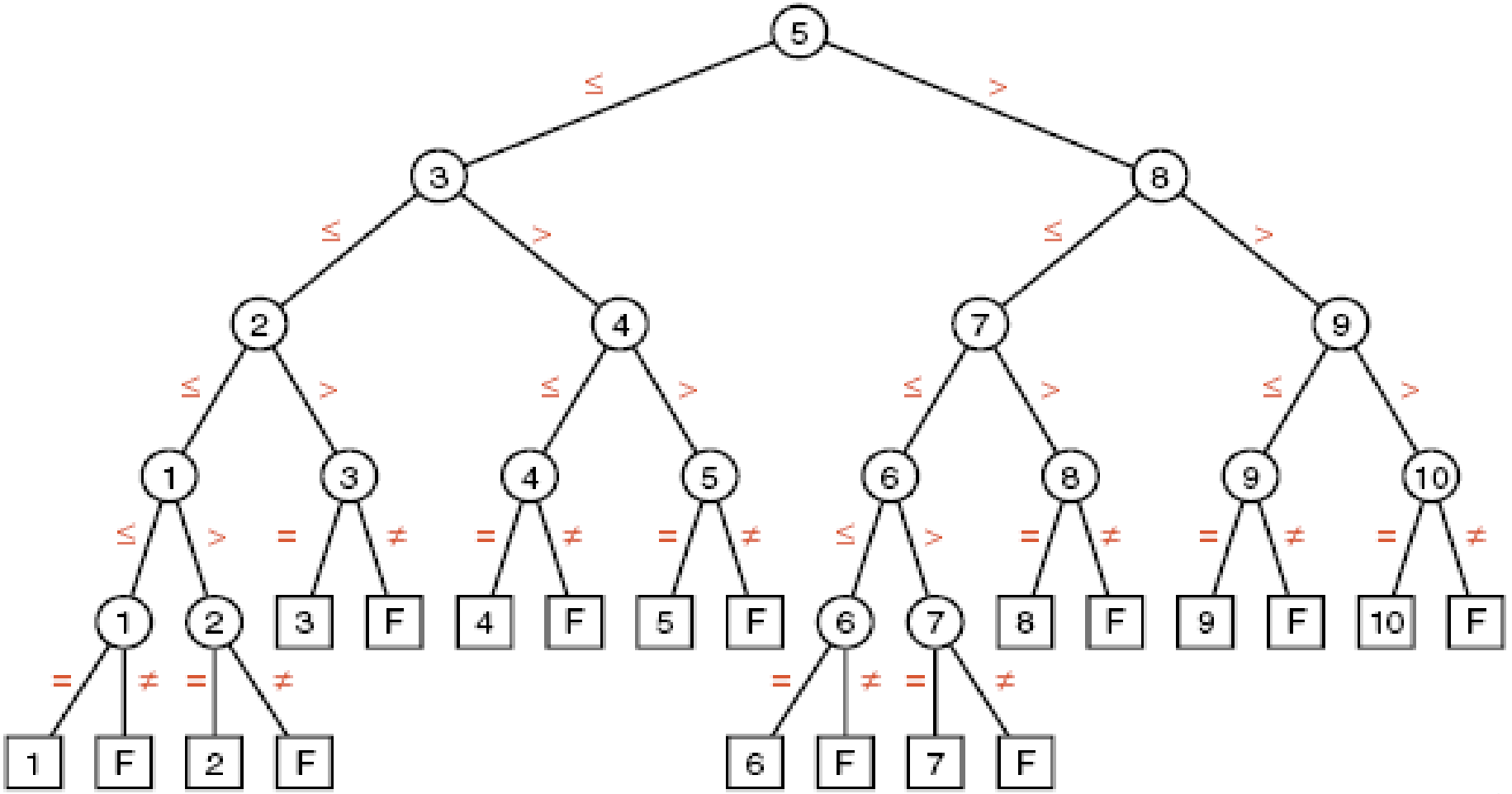
- If target appears more than once in the list, then the search may return any instance of the target (There's the disadvantage of the second version of binary search).

Recognizing Equality in Binary Search (cont.)

- Proof of progress toward termination is easier than for the first method.

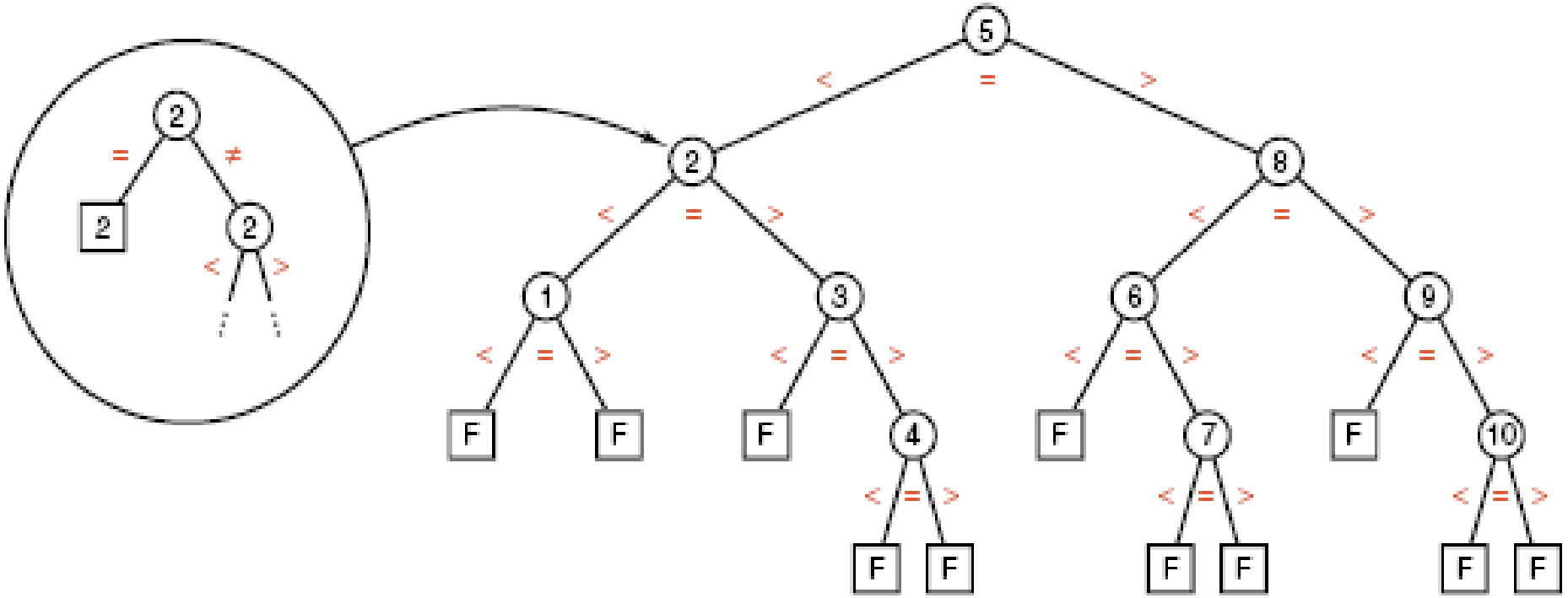
Comparison Trees for Binary Search

First version:



Comparison Trees for Binary Search

Second version:



Binary Search Analysis

The number of comparisons of keys done by binary_search_1 in searching a list of n items is approximately

$$\lg n + 1$$

in the worst case and

$$\lg n$$

in the average case. The number of comparisons is essentially independent of whether the search is successful or not.

The number of comparisons done in an unsuccessful search by binary_search_2 is approximately $2 \lg(n + 1)$.

In a successful search of a list of n entries, binary_search_2 does approximately

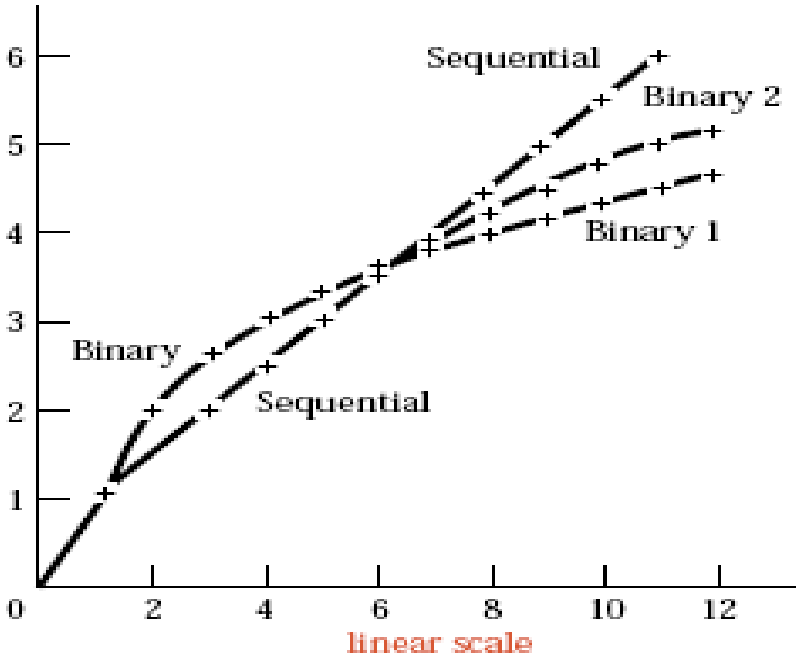
$$\frac{2(n+1)}{n} \lg(n+1) - 3$$

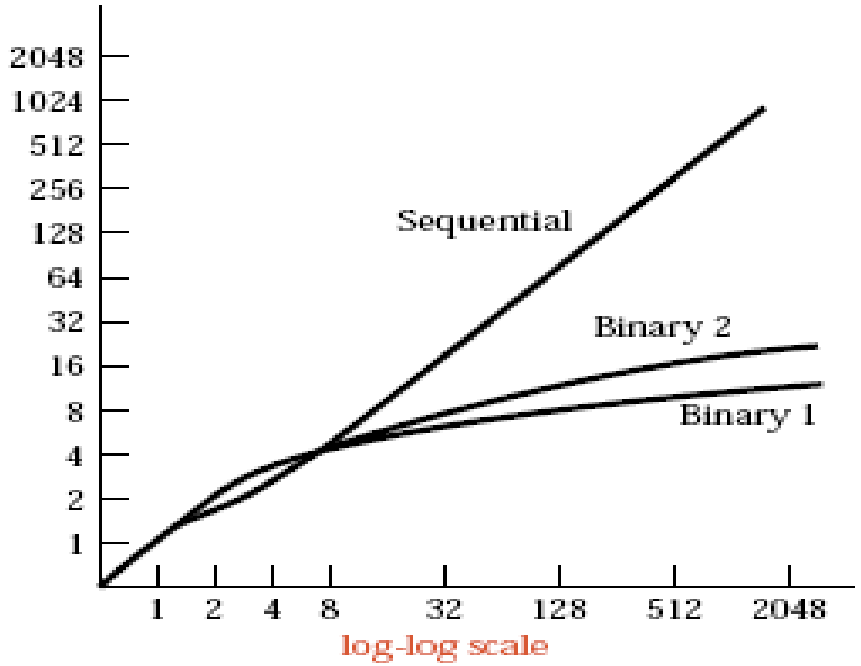
comparisons of keys.

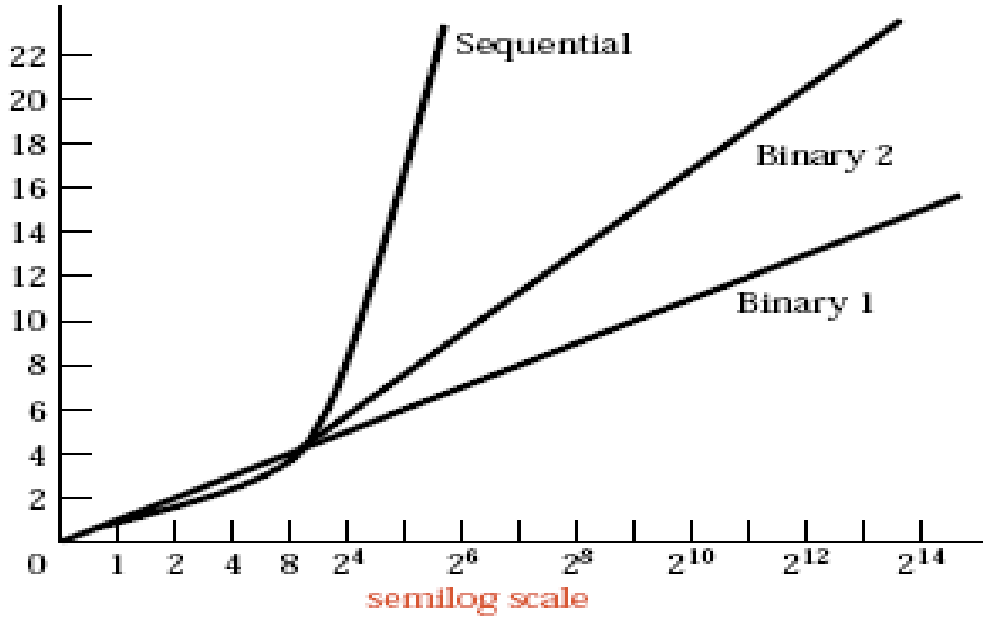
Binary Search (cont.)

	Successful search	*Unsuccessful search*
binary_search_1	$\lg n + 1$	$\lg n + 1$
binary_search_2	$2 \lg n - 3$	$2 \lg n$

Numbers of comparisons for average successful searches







Binary Search in Contiguous Lists

OK!

Binary Search in Linked Lists

- How to implement $data_{position}$?
  - Retrieve?

    Cost to retrieve element at position p is p

    ⟹ Have to re-calculate the cost of the search !
  - GetNext?

    NO: GetNext gets elements in sequential order
- Two solutions:
  - Assume that retrieving an element at position p cost 1

    Binary search
  - Use other searching method