Data Structures and Algorithms

Rutgers University

Binary Search (review)

http://ds.cs.rutgers.edu

Linear Search Recall

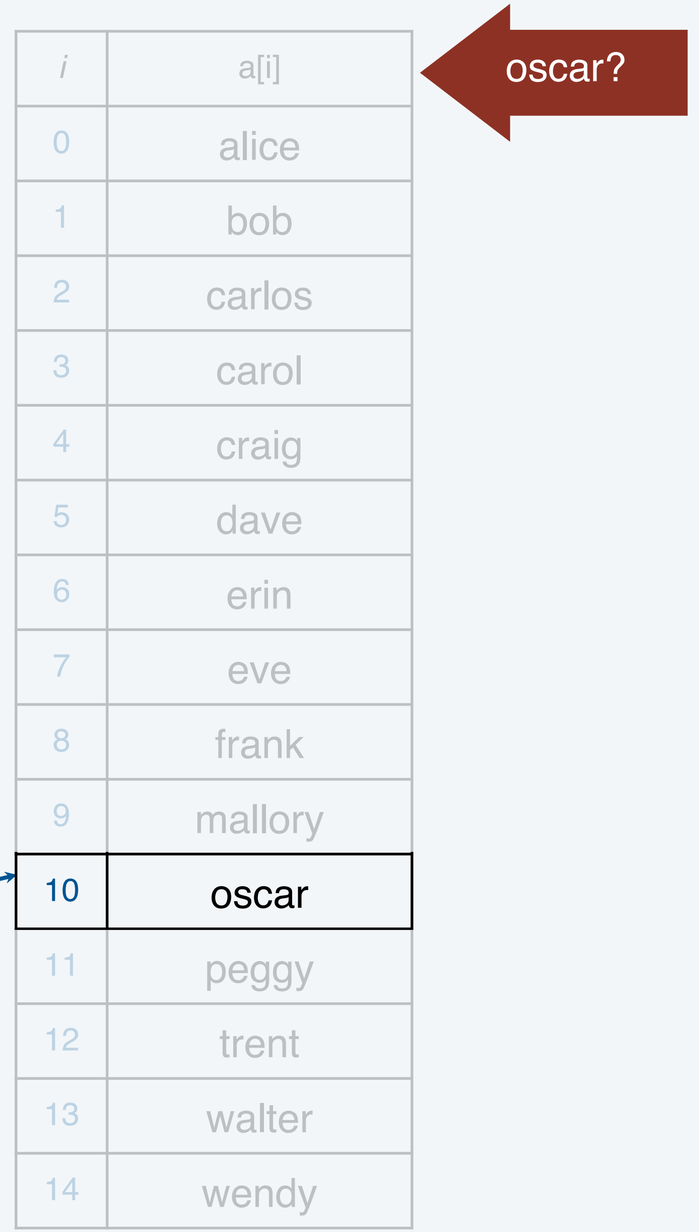
Linear search

- Keep the array in any order
- Examine the array starting from first entry until key is found or not found.

```
public static int linearSearch ( String key, String[] a )
{
   for (int i = 0; i < a.length; i++)
      if ( a[i].compareTo(key) == 0 ) return i;
   return -1;
}
```

Linear search

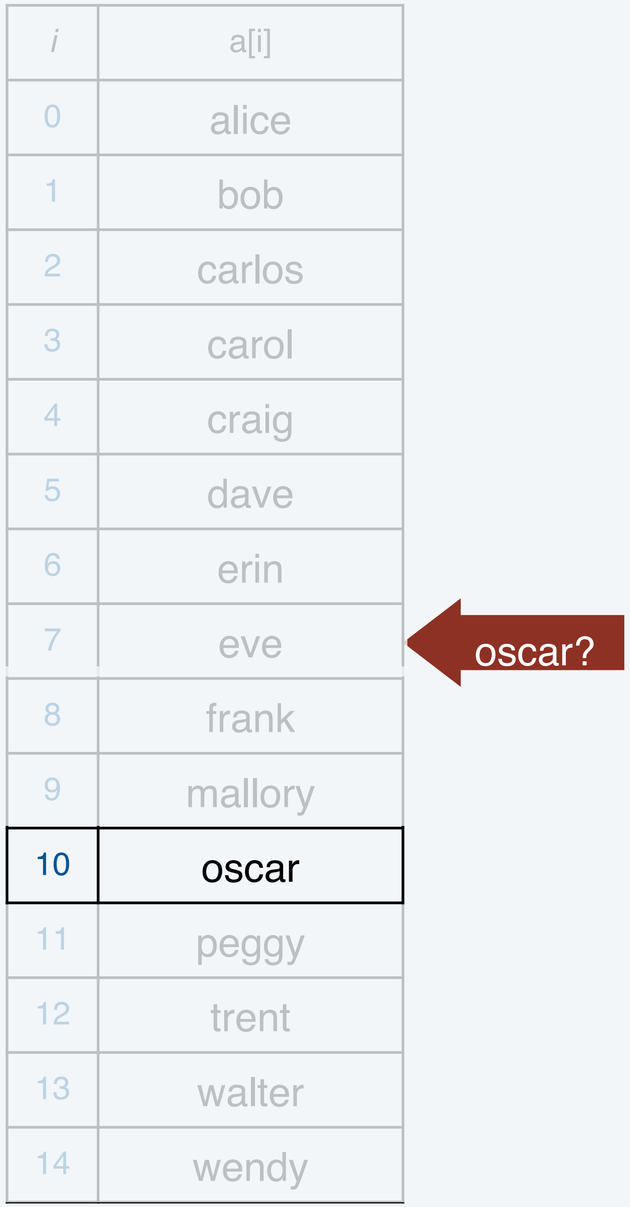
Match found.
Return 10



Binary Search Recall

Binary search

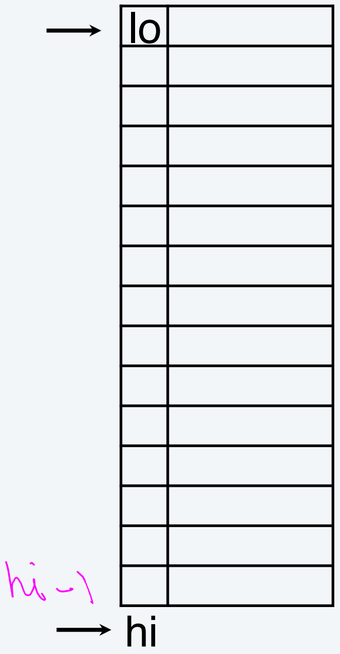
- Keep the array in sorted order
- Examine the middle key.
- If it matches, return its index.
- If it is larger, search the half with lower indices.
- If it is smaller, search the half with upper indices.



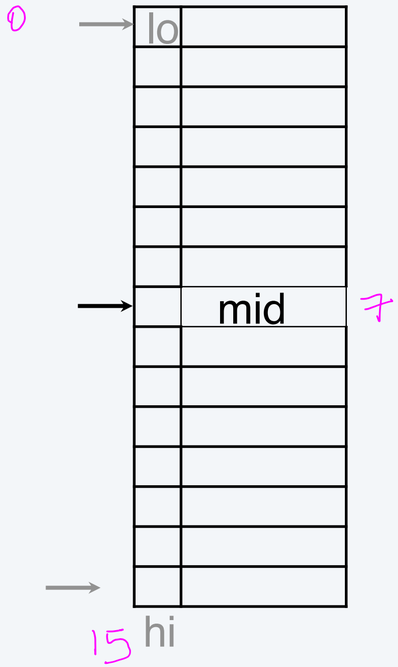
Binary search arithmetic

Notation. a[lo,hi) means a[lo], a[lo+1] ... a[hi-1] (does not include a[hi]).

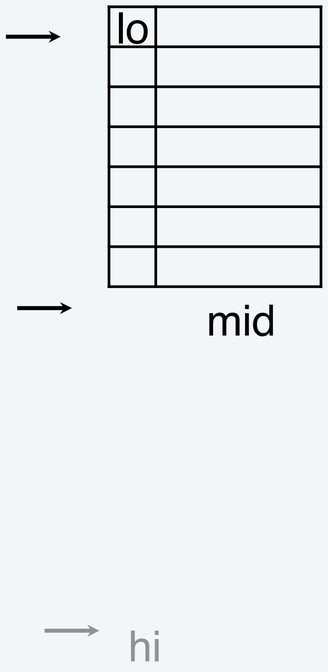
Search in a[lo,hi)



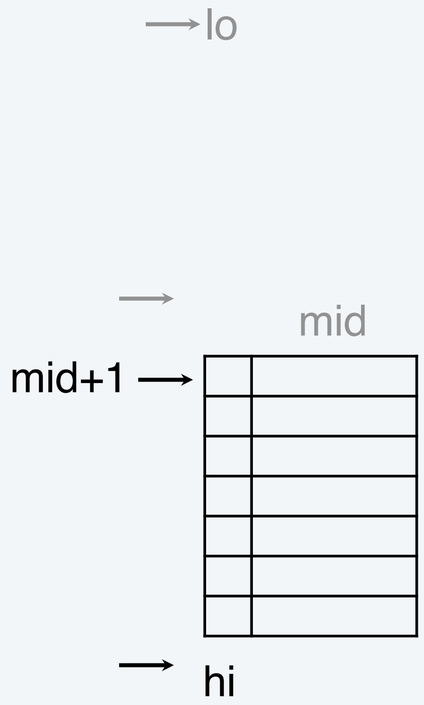
mid = lo + (hi-lo)/2



Lower half: a[lo,mid)



Upper half: a[mid+1,hi)

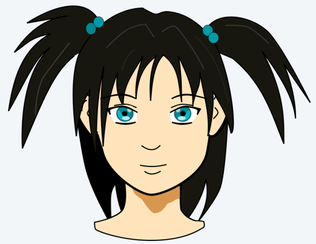


Binary search: Java implementation


```
public static int search(String key, String[] a)
{  return search(key, a, 0, a.length); }


public static int search(String key, String[] a, int lo, int hi)
{
   if (hi <= lo) return -1;
   int mid = lo + (hi - lo) / 2;
   int cmp = a[mid].compareTo(key);
   if      (cmp > 0) return search(key, a, lo, mid);
   else if (cmp < 0) return search(key, a, mid+1, hi);
   else              return mid;
}
```

a.compareTo(b) returns:

- **0** is a equals to b
- **< 0** is a is less than b
- **> 0** is a is greater than b

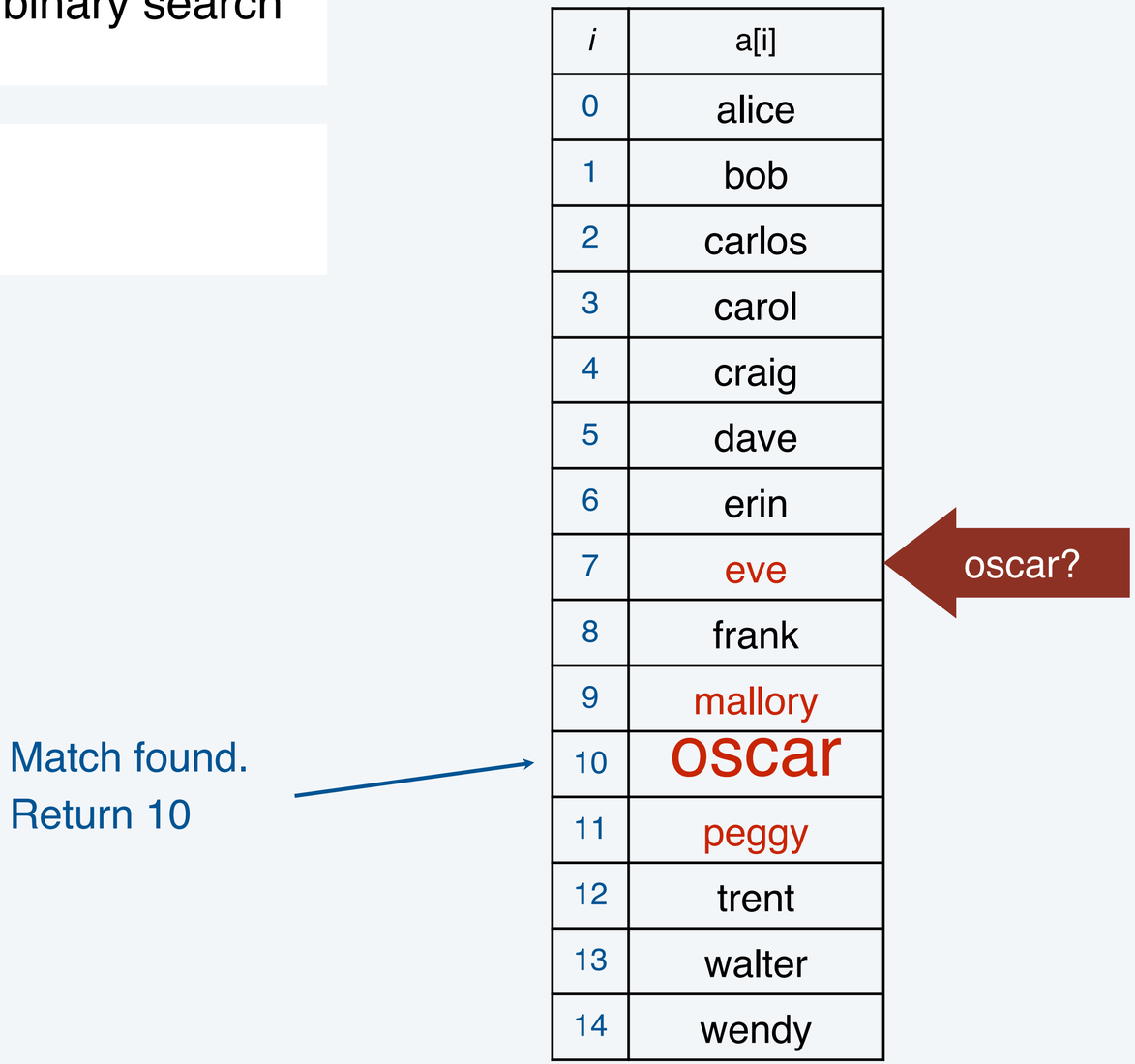


Still, this was easier than I thought!

Understanding binary search

Identify. All elements that are inspected during binary search

comparisons. 4 names inspected



Recursion trace for binary search


```
public static int search(String key, String[] a)
{  return search(key, a, 0, a.length); }

public static int search(String key, String[] a,
                         int lo, int hi)
{
  if (hi <= lo) return -1;
  int mid = lo + (hi - lo) / 2;
  int cmp = a[mid].compareTo(key);
  if      (cmp > 0) return search(key, a, lo, mid);
  else if (cmp < 0) return search(key, a, mid+1, hi);
  else              return mid;
}
```

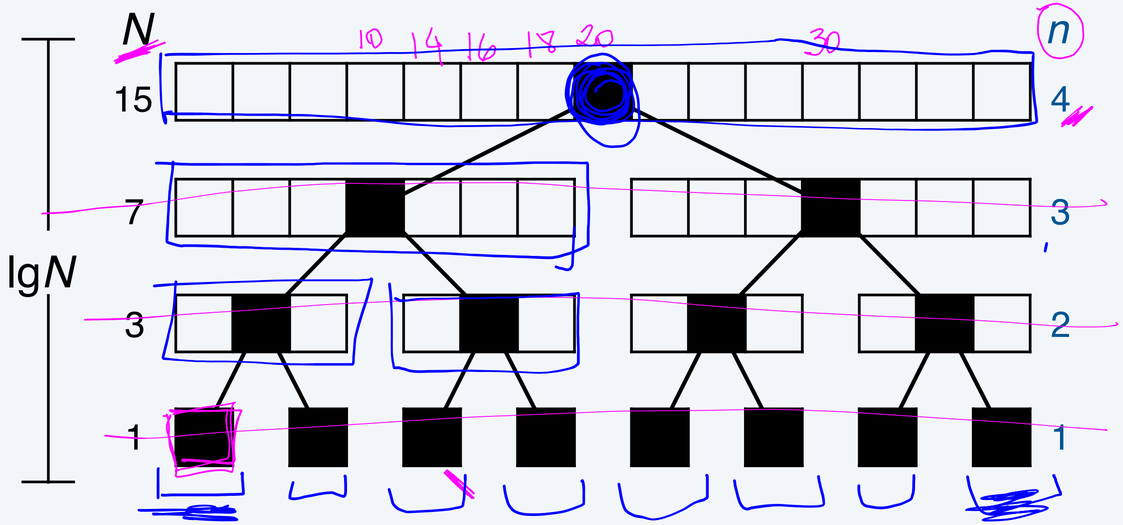
```
search("oscar")
  return search(... 0, 15);

search("oscar", a, 0, 15)
  mid = 7;
  > "eve"
  return search(... 8, 15);

search("oscar", a, 8, 15)
  mid = 11;
  < "peggy"
  return search(... 8, 10);

search("oscar", a, 8, 11)
  mid = 9;
  > "mallory"
  return search(... 10, 11);

search("oscar", a, 10, 11)
  mid = 10;
  == "oscar"
  return 10;
```

i	a[i]
0	alice
1	bob
2	carlos
3	carol
4	craig
5	dave
6	erin
7	eve
8	frank
9	mallory
10	oscar
11	peggy
12	trent
13	walter
14	wendy

Mathematical analysis of binary search (optional)

Exact analysis for search miss for $N = 2^n - 1$

- Note that $n = \lg(N+1) \sim \lg N$.
- Subarray size for 1st call is $2^n - 1$.
- Subarray size for 2nd call is $2^{n-1} - 1$.
- Subarray size for 3rd call is $2^{n-2} - 1$.
- ...
- Subarray size for nth call is 1.
- Total # compares (one per call): $n \sim \lg N$.



Every search miss is a top-to-bottom path in this tree.

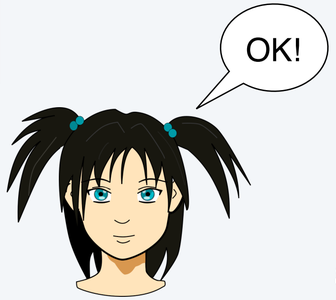
Proposition. Binary search uses ~lg N compares for a search miss.

Proof. An (easy) exercise in discrete math.

Proposition. Binary search uses ~lg N compares for a random search hit.

Proof. A slightly more difficult exercise in discrete math.

Interested in details?
Take a course in algorithms.

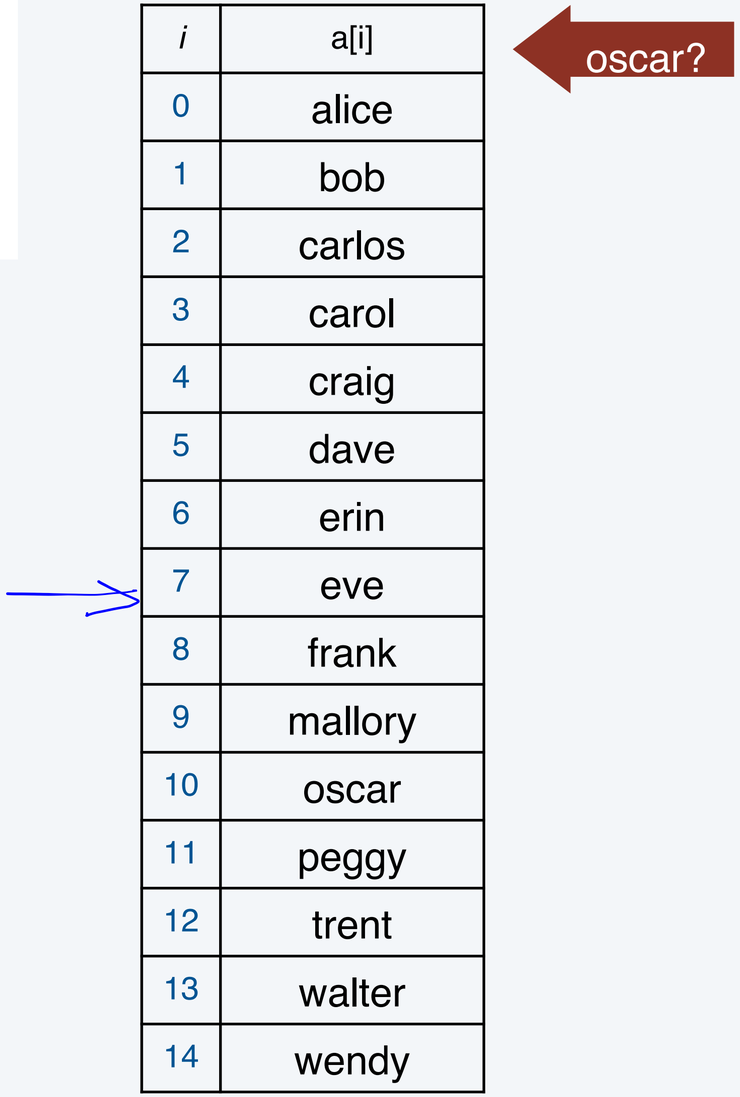


Best, worst and average case of binary search

Best case. Oscar is the a[mid] record (1 comparison)

Worst case. Oscar is end/begin key (log N comparisons)

Average case. Oscar is some key (log N comparisons)



Empirical tests of binary search

Whitelist filter scenario

- Whitelist of size *N*.
- 10*N* transactions.

N	T_N (seconds)	$T_N/T_{N/2}$	*transactions per second*
100,000	1		
200,000	3		
400,000	6	2	67,000
800,000	14	2.35	57,000
1,600,000	33	2.33	48,000
10.28 million	264	2	48,000

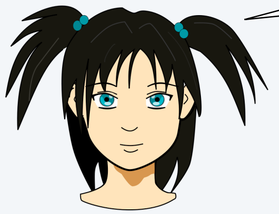
nearly 50,000 transactions per second, and holding

```
% java Generator 100000 ...
1 seconds
% java Generator 200000 ...
3 seconds
% java Generator 400000 ...
6 seconds
% java Generator 800000 ...
14 seconds
% java Generator 1600000 ...
33 seconds
```

... = 10 a-z l java TestBS
a-z = abcdefghijklmnopqrstuvwxyz

Validates hypothesis that order of growth is $N \log N$.

Will scale.

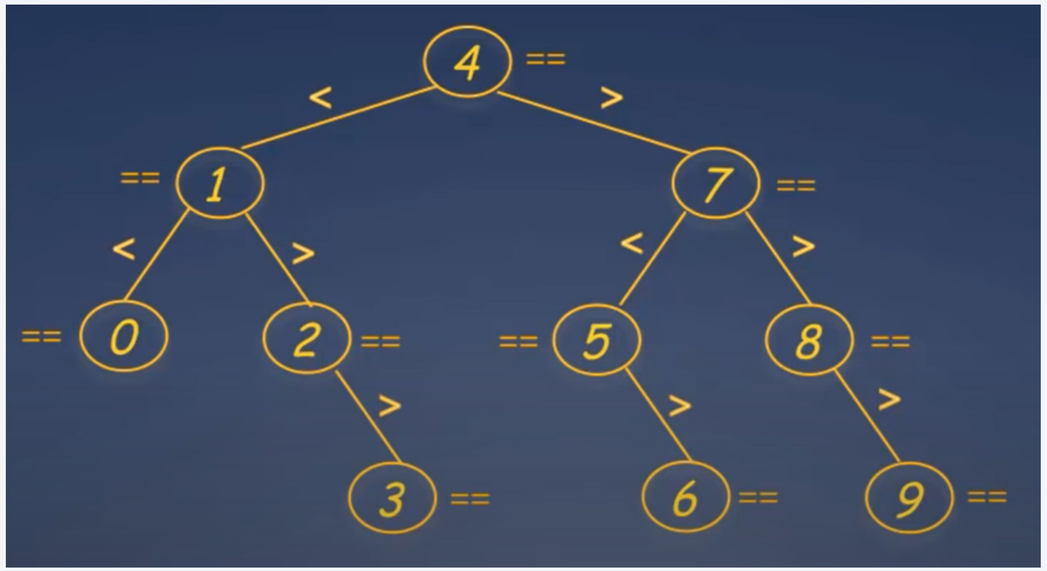


Great! But how do I get the list into sorted order at the beginning?

Decision Tree based Analysis of Binary Search

0	1	2	3	4	5	6	7	8	9
11	12	17	19	26	38	45	62	69	83

An array of size 10



The comparison tree shows the indices of the array elements that are examined in each comparison

The tree depicts all possible search paths for binary search. For keys that may or may not be present.

Question. What is the maximum number of comparisons to find (or not find) an element?

Answer. Proportional to height of the tree

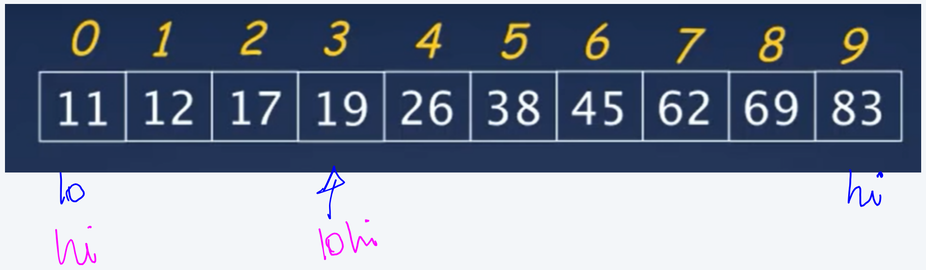
Image credits – Sesh Venugopal:

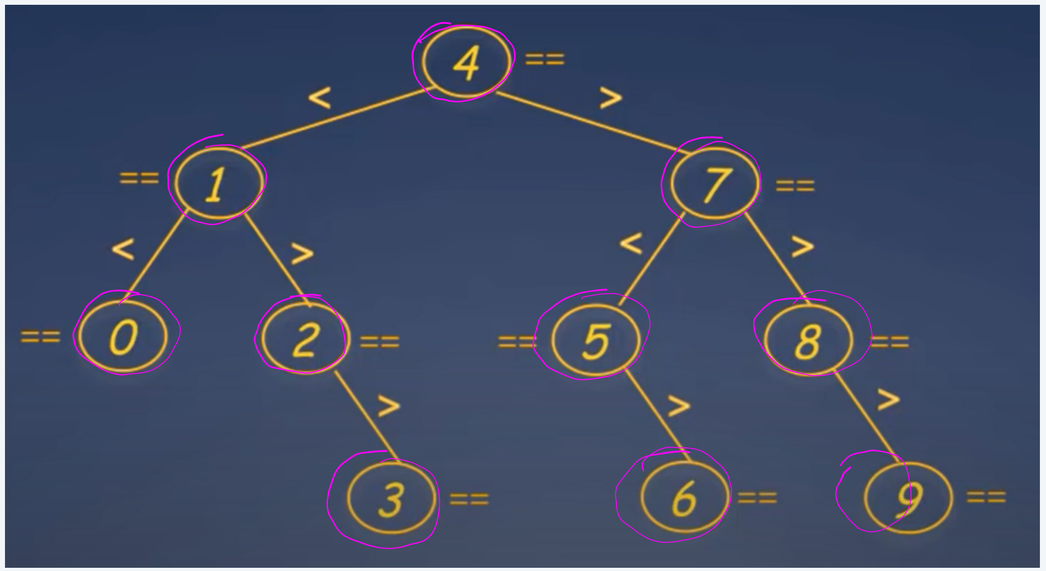
Building the Decision Tree

Use the following implementation of binary search to build the tree.

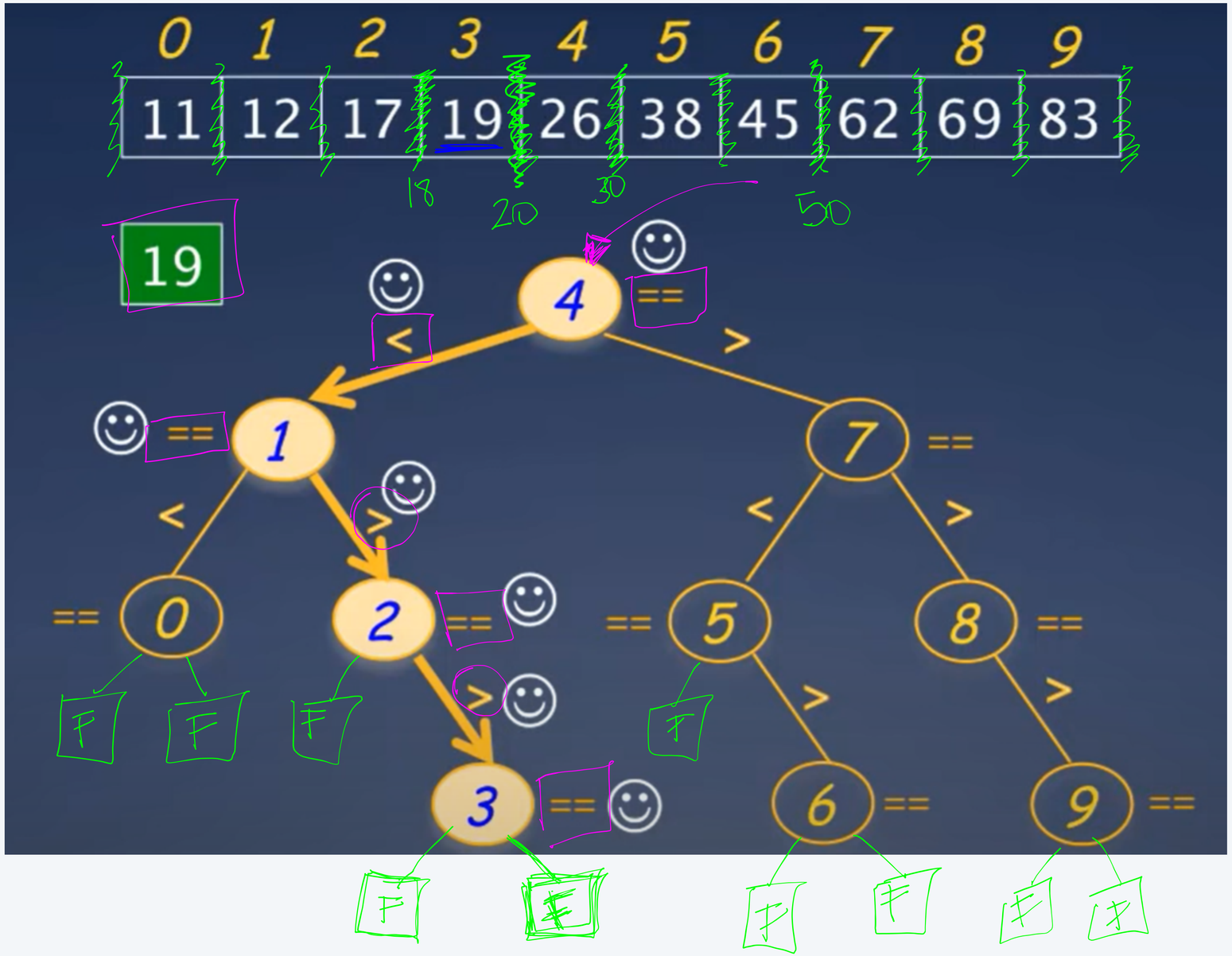
Search for every item in the array until the tree is fully built.

```
public static int indexOf (int [] a, int key) {

   int lo = 0;
   int hi = a.length - 1;

   while (lo <= hi) {
      int mid = lo + (hi - lo) / 2;
      if (key == a[mid]) return mid;
      if (key < a[mid]) hi = mid - 1;
      else lo = mid + 1;
   }
   return -1;  // key is not present in array a
}
```





An Example



Searching for Target 19
Found at array index 3

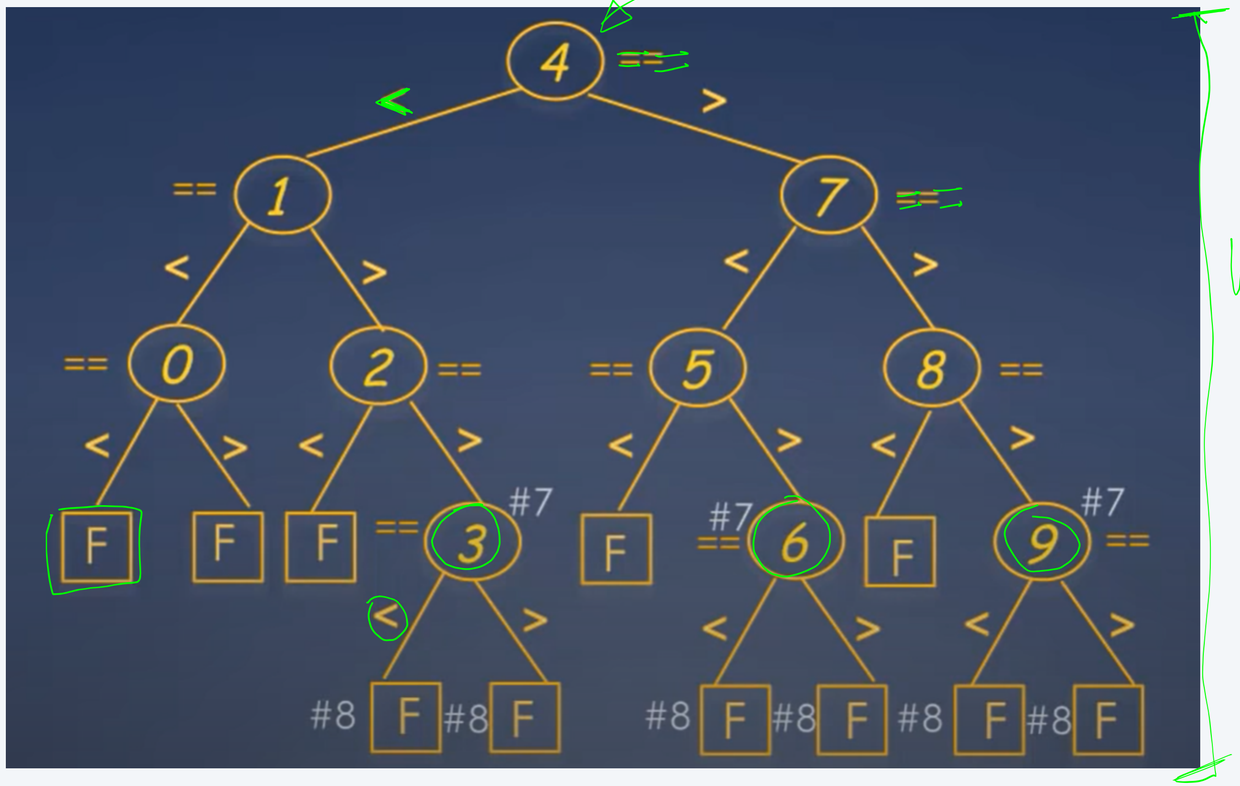
Count the symbols along the search path to find the number of compares.

Q. What is the number of comparisons to find 19?

A. 7

Image credits – Sesh Venugopal:

Worse Case Comparisons



Successful Search – index of the search element is an inner node

Unsuccessful Search – occurs at the leaf nodes marked as F

At each node there are at most 2 comparisons to decide which way to go.

Worst-case for successful search

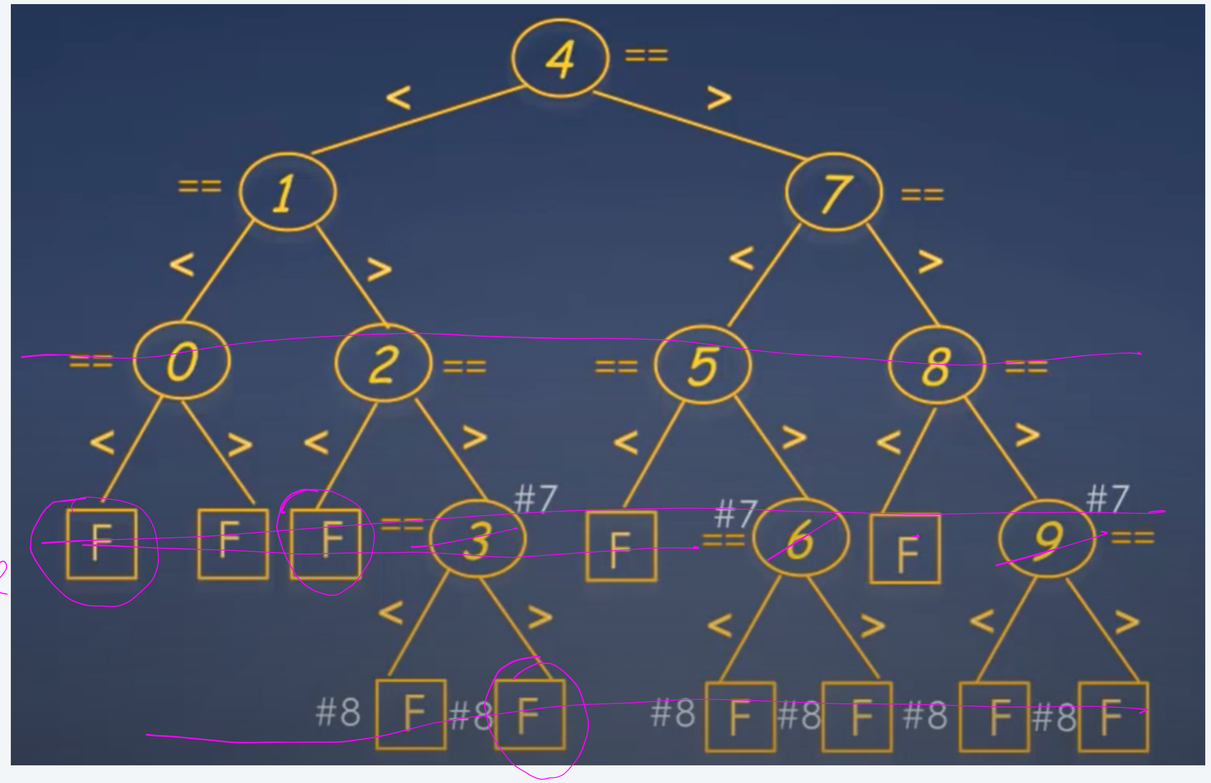
in this example the indices {3, 6, 9} have the longest path of 4 nodes for successful search

Worst-case for unsuccessful search

is one additional comparison after the worst-case successful search.

Complexity of the search ~ 2 log n

Average Case Comparisons



Successful Search – average the number of comparisons for all successful searches.

Index	#comparisons
4	1
1	3
7	3
0	5
2	5
5	5
8	5
3	7
6	7
9	7
	48 / 10 = 4.8

Average number of compares for success

ARRAY size

Unsuccessful Search – if we don't know how many keys are being searched, we can only say the average will be between 6-8.

6 — 8

$\frac{6+8}{2} = 7$

Data Structures and Algorithms

Rutgers University

Binary Search Review

http://ds.cs.rutgers.edu