

# **Design and Analysis of Algorithm (KCS503)**

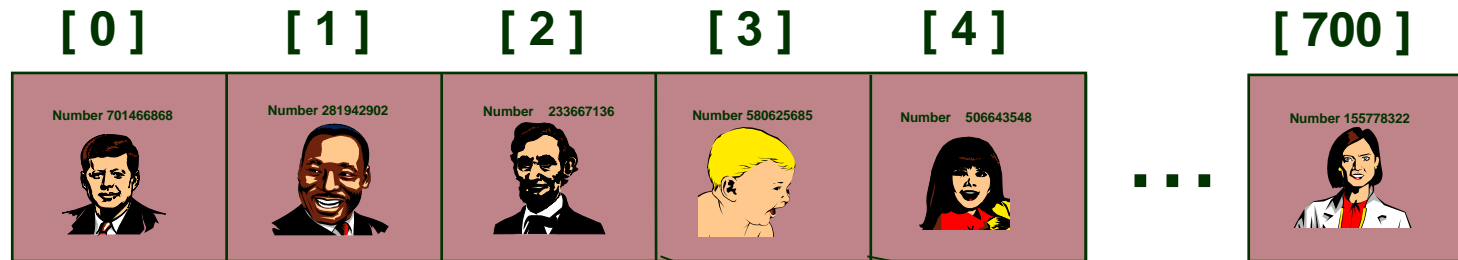
**Analysis of a given algorithm, Like  
Linear Search, Binary Search**

**Lecture - 2**

# Problem: Search

- We are given a list of records.
- Each record has an associated key.
- Give efficient algorithm for searching for a record containing a particular key.
- Efficiency is quantified in terms of average time analysis (number of comparisons) to retrieve an item.

# Search



Each record in list has an associated key.  
In this example, the keys are ID numbers.

Given a particular key, how can we efficiently retrieve the record from the list?



# Serial Search

- Step through array of records, one at a time.
- Look for record with matching key.
- Search stops when
  - record with matching key is found
  - or when search has examined all records without success.

# Pseudocode for Serial Search

```
// Search for a desired item in the n array elements
// starting at a[first].
// Returns pointer to desired record if found.
// Otherwise, return NULL
...
for(i = first; i < n; ++i )
    if(a[first+i] is desired item)
        return &a[first+i];

// if we drop through loop, then desired item was not found
return NULL;
```

# Serial Search Analysis

- What are the worst and average case running times for serial search?
- We must determine the O-notation for the number of operations required in search.
- Number of operations depends on  $n$ , the number of entries in the list.

# Worst Case Time for Serial Search

- For an array of  $n$  elements, the worst case time for serial search requires  $n$  array accesses:  $O(n)$ .
- Consider cases where we must loop over all  $n$  records:
  - desired record appears in the last position of the array
  - desired record does not appear in the array at all

# Average Case for Serial Search

## Assumptions:

1. All keys are equally likely in a search
2. We always search for a key that is in the array

## Example:

- We have an array of 10 records.
- If search for the first record, then it requires 1 array access; if the second, then 2 array accesses. *etc.*

The average of all these searches is:

$$(1+2+3+4+5+6+7+8+9+10)/10 = 5.5$$



# Average Case Time for Serial Search

Generalize for array size  $n$ .

Expression for average-case running time:

$$(1+2+\dots+n)/n = n(n+1)/2n = (n+1)/2$$

Therefore, average case time complexity for serial search is  $O(n)$ .

# Binary Search

- Perhaps we can do better than  $O(n)$  in the average case?
- Assume that we are given an array of records that is sorted. For instance:
  - an array of records with integer keys sorted from smallest to largest (e.g., ID numbers), or
  - an array of records with string keys sorted in alphabetical order (e.g., names).

# Binary Search Pseudocode

```
...
if(size == 0)
    found = false;
else {
    middle = index of approximate midpoint of array segment;
    if(target == a[middle])
        target has been found!
    else if(target < a[middle])
        search for target in area before midpoint;
    else
        search for target in area after midpoint;
}
...
```

# Binary Search

Example: sorted array of integer keys. Target=7.

[ 0 ]	[ 1 ]	[ 2 ]	[ 3 ]	[ 4 ]	[ 5 ]	[ 6 ]
3	6	7	11	32	33	53

# Binary Search

Example: sorted array of integer keys. Target=7.

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53



Find approximate midpoint

# Binary Search

Example: sorted array of integer keys. Target=7.

[ 0 ]	[ 1 ]	[ 2 ]	[ 3 ]	[ 4 ]	[ 5 ]	[ 6 ]
3	6	7	11	32	33	53



Is 7 = midpoint key? NO.

# Binary Search

Example: sorted array of integer keys. Target=7.

[ 0 ]	[ 1 ]	[ 2 ]	[ 3 ]	[ 4 ]	[ 5 ]	[ 6 ]
3	6	7	11	32	33	53




Is 7 < midpoint key? YES.

# Binary Search

Example: sorted array of integer keys. Target=7.

[ 0 ]	[ 1 ]	[ 2 ]	[ 3 ]	[ 4 ]	[ 5 ]	[ 6 ]
3	6	7	11	32	33	53



Search for the target in the area before midpoint.



# Binary Search

Example: sorted array of integer keys. Target=7.

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53



Find approximate midpoint

# Binary Search

Example: sorted array of integer keys. Target=7.

[ 0 ]	[ 1 ]	[ 2 ]	[ 3 ]	[ 4 ]	[ 5 ]	[ 6 ]
3	6	7	11	32	33	53



Target = key of midpoint? NO.

# Binary Search

Example: sorted array of integer keys. Target=7.

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53



Target < key of midpoint? NO.

# Binary Search

Example: sorted array of integer keys. Target=7.

[ 0 ]	[ 1 ]	[ 2 ]	[ 3 ]	[ 4 ]	[ 5 ]	[ 6 ]
3	6	7	11	32	33	53




Target > key of midpoint? YES.

# Binary Search

Example: sorted array of integer keys. Target=7.

[ 0 ]	[ 1 ]	[ 2 ]	[ 3 ]	[ 4 ]	[ 5 ]	[ 6 ]
3	6	7	11	32	33	53



Search for the target in the area after midpoint.

# Binary Search

Example: sorted array of integer keys. Target=7.

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53



Find approximate midpoint.  
Is target = midpoint key? YES.

# Binary Search Implementation

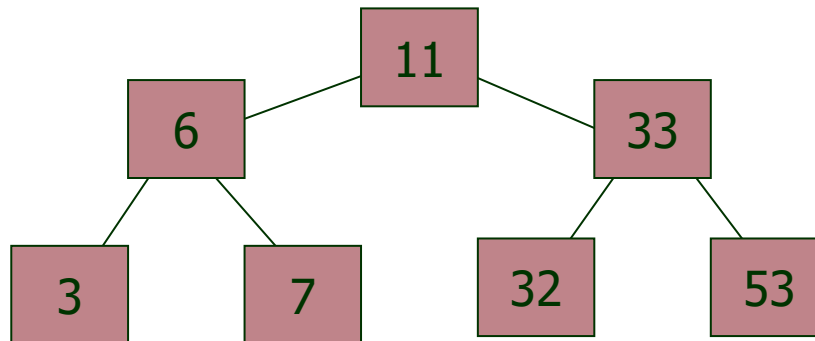
```
void search(const int a[ ], size_t first, size_t size, int target, bool& found, size_t&
location)
{
    size_t middle;
    if(size == 0) found = false;
    else {
        middle = first + size/2;
        if(target == a[middle]){
            location = middle;
            found = true;
        }
        else if (target < a[middle])
            // target is less than middle, so search subarray before middle
            search(a, first, size/2, target, found, location);
        else
            // target is greater than middle, so search subarray after middle
            search(a, middle+1, (size-1)/2, target, found, location);
    }
}
```

# Relation to Binary Search Tree

Array of previous example:

3	6	7	11	32	33	53
---	---	---	----	----	----	----

Corresponding complete binary search tree



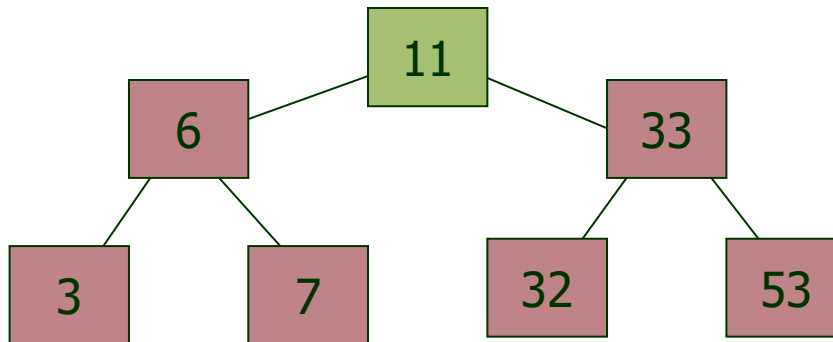


# Search for target = 7

Find midpoint:

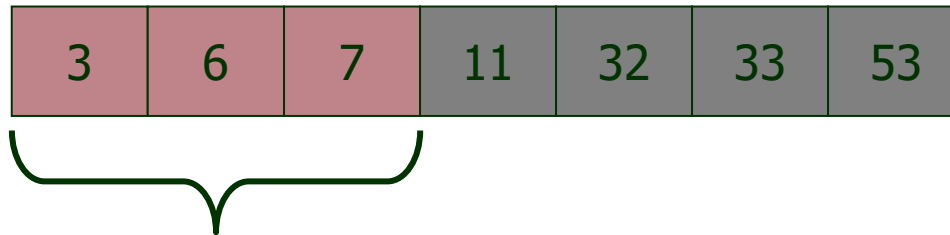
3	6	7	11	32	33	53
---	---	---	----	----	----	----

Start at root:

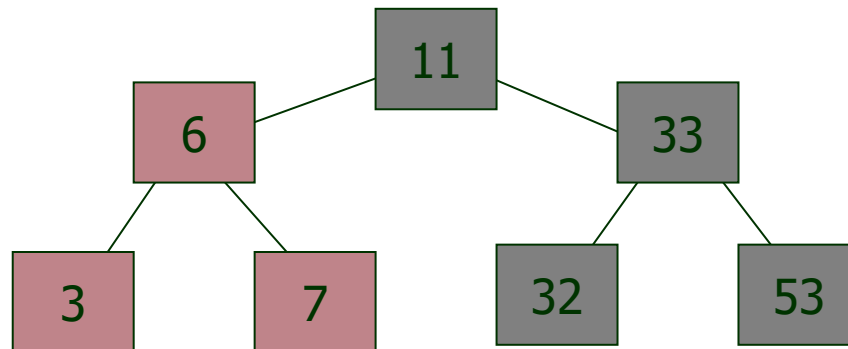


# Search for target = 7

Search left subarray:



Search left subtree:

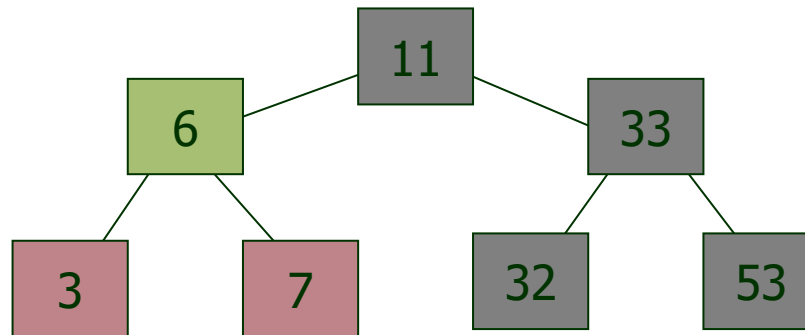


# Search for target = 7

Find approximate midpoint of

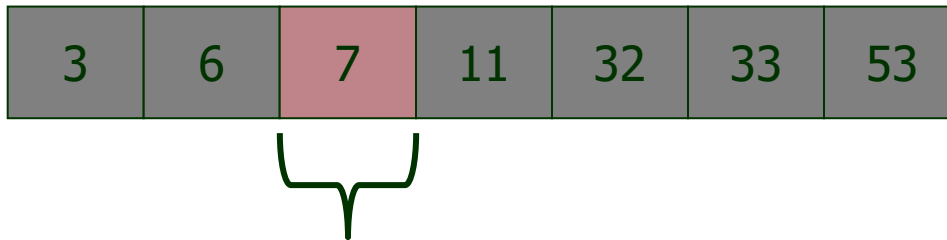


Visit root of subtree:

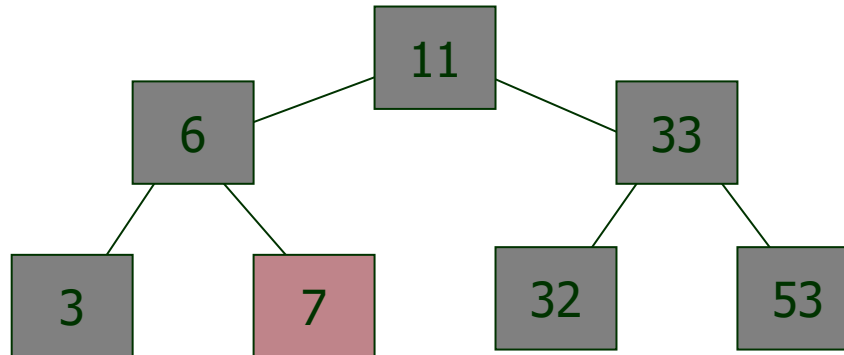


# Search for target = 7

Search right subarray:



Search right subtree:



# Binary Search: Analysis

- Worst case complexity?
- What is the maximum depth of recursive calls in binary search as function of  $n$ ?
- Each level in the recursion, we split the array in half (divide by two).
- Therefore maximum recursion depth is  $\text{floor}(\log_2 n)$  and worst case =  $O(\log_2 n)$ .
- Average case is also =  $O(\log_2 n)$ .

# Can we do better than $O(\log_2 n)$ ?

- Average and worst case of serial search =  $O(n)$
- Average and worst case of binary search =  $O(\log_2 n)$