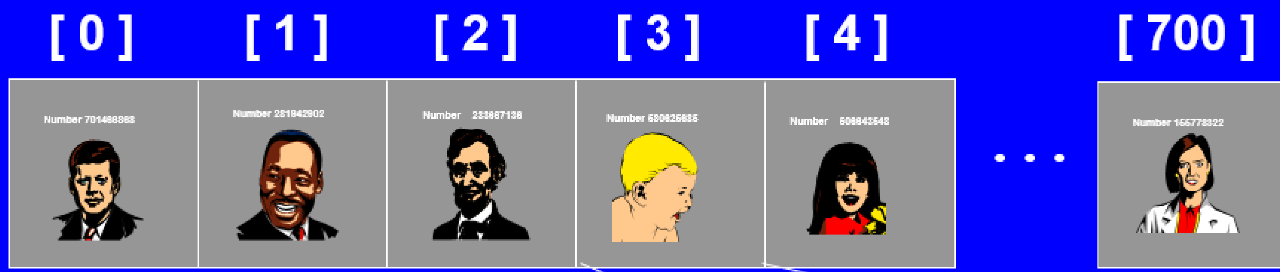
Analysis and Design of Algorithms

By
Syed Bakhtawar Shah Abid
Lecturer in Computer Science

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?

Sequential 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 Sequential 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 = 0; i < n; i ++ )
                if(a[i]==item)
                        return &a[i];
        // if we drop through loop, then desired item was not found
        return NULL;
```

Sequential 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 Sequential 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 Sequential 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 Sequential Search

Generalize for array size *n.*

Expression for average-case running time:

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

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

Binary Search

- Perhaps we can do better than O(n) in the average case?
- Assume that we are give 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

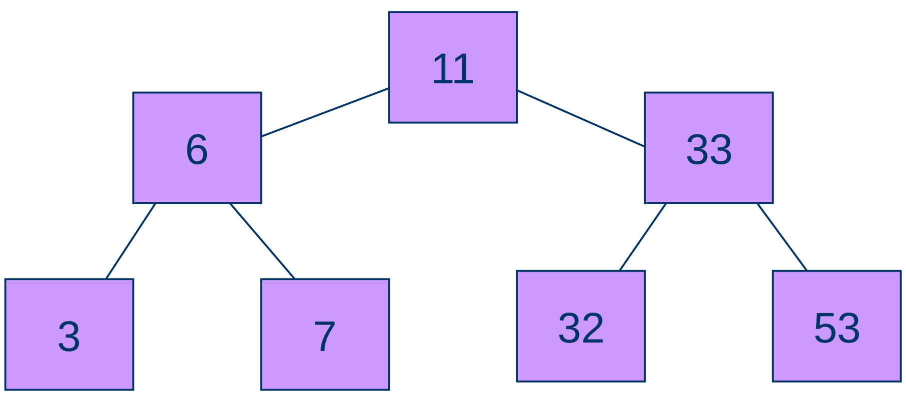
Find approximate midpoint.
Is target = midpoint key? YES.

Relation to Binary Search Tree

Array of previous example:

3	6	7	11	32	33	53

Corresponding complete binary search tree

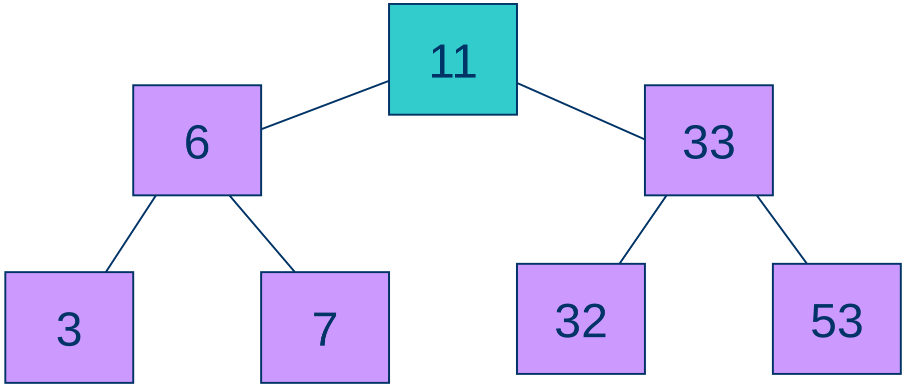


Search for target = 7

Find midpoint:

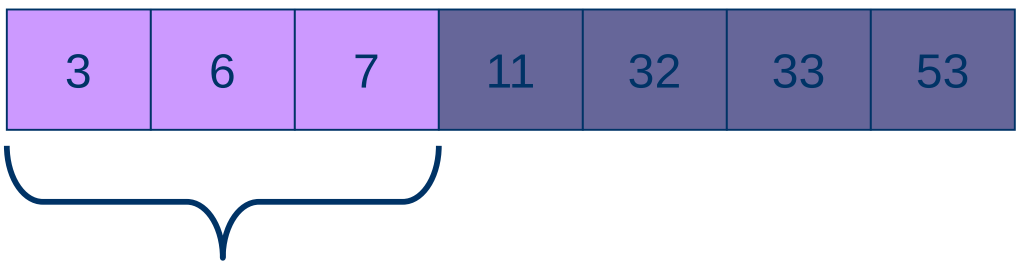
3	6	7	11	32	33	53

Start at root:

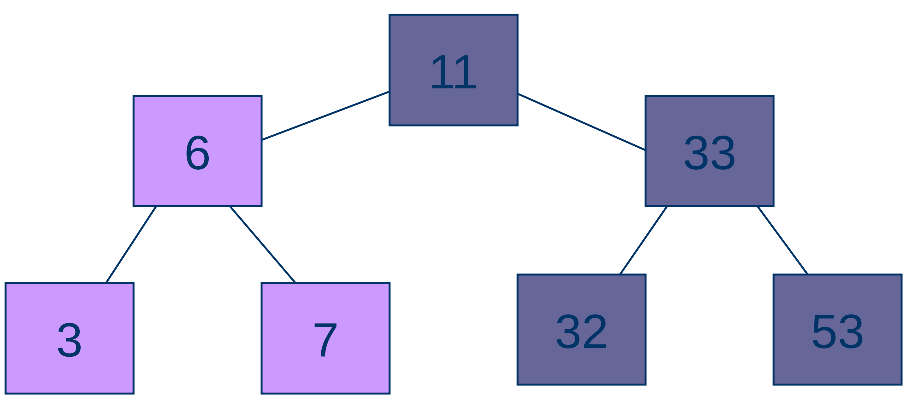


Search for target = 7

Search left subarray:

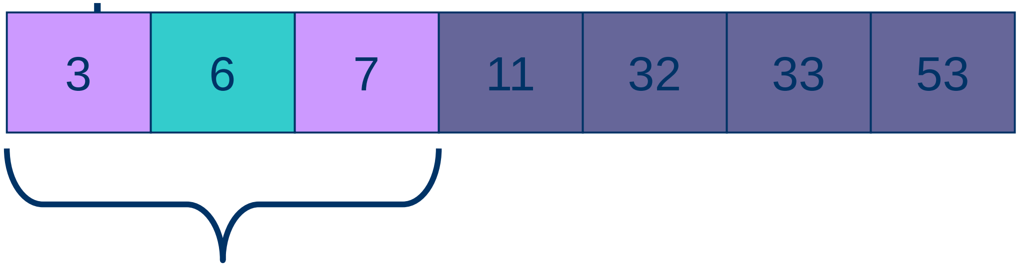


Search left subtree:

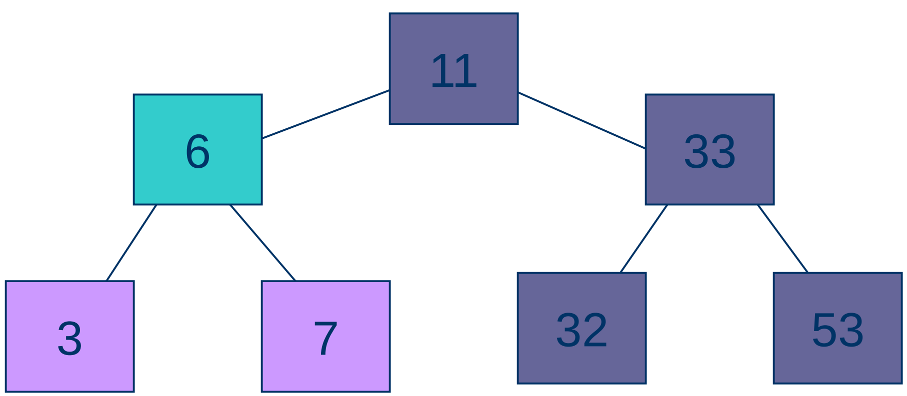


Search for target = 7

Find approximate midpoint of

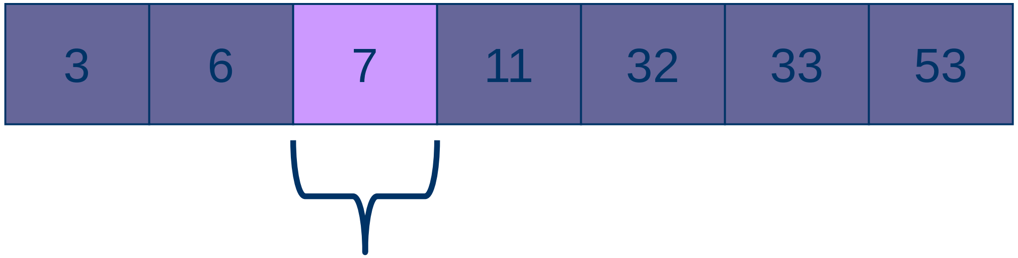


Visit root of subtree:

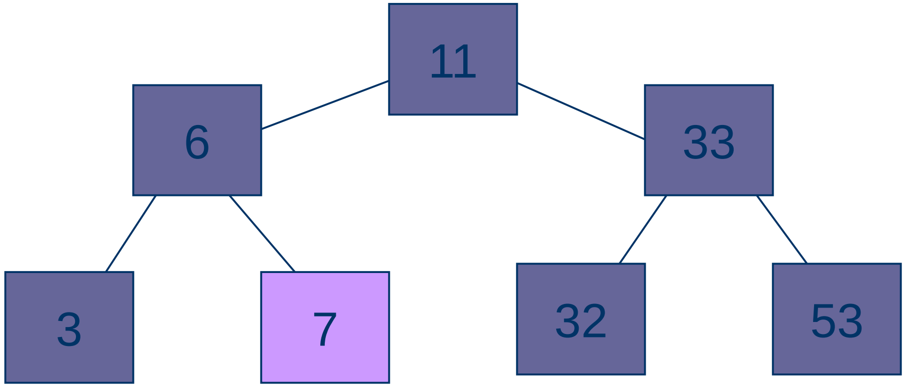


Search for target = 7

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 floor($\log_2 n$) and worst case = $O(\log_2 n)$.
- Average case is also = $O(\log_2 n)$.