Searching (Binary Search)

T#11

Binary Search

- Input:
  - A sorted array A[ ] of size n
  - A value b to be searched for in A[ ]
- Output:
  - If b is found, the index k where A[k]=b
  - If b is not found, return -1
- Definition: An A[ ] is said to be *sorted* if:

  $A[0] \leq A[1] \leq A[2] \leq \ldots \leq A[n-1]$

Binary Search

Example: sorted array of integer keys. Key=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

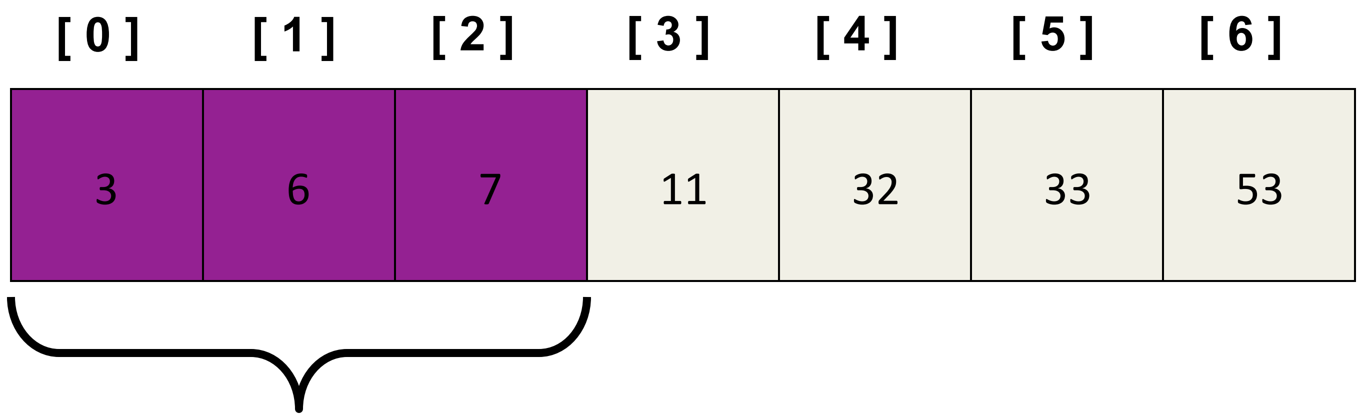
Example: sorted array of integer keys. Target=7.

[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.



Search for the target in the area before midpoint.

Binary Search

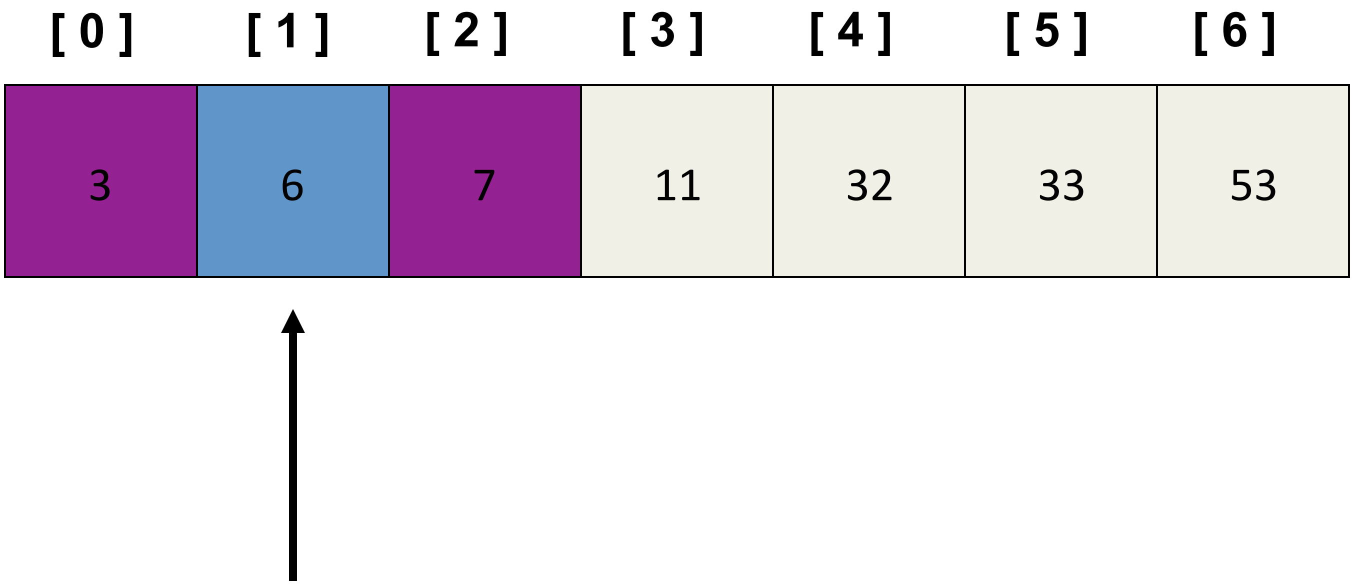
Example: sorted array of integer keys. Target=7.

[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.



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.

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

```
int binarySearch(T key)
  return bsearch(0, n − 1, key)


int bsearch(int low, int high, T key)
  if low > high then
    throw KeyNotFoundException
  else
    mid = ⌊(low + high)/2⌋
    if (A[mid] > key) then
      return bsearch(low, mid − 1, key)
    else if (A[mid] < key) then
      return bsearch(mid + 1, high, key)
    else
      return mid
    end if
  end if
```

Binary Search Pre-Condition 1

Precondition 1:

a) A is an array with length $A.length = n \geq 1$ storing values of some *ordered* type T

b) $A[i] < A[i+1]$ for every integer i such that $0 \leq i < n-1$

c) *key* is a value of type T that is stored in A

d) *low* and *high* are integers such that

- $0 \leq low \leq n$
- $-1 \leq high \leq n-1$
- $low \leq high + 1$
- $A[h] < key$ for $0 \leq h < low$
- $A[h] > key$ for $high < h \leq n-1$

Binary Search Pre-Condition 2

Precondition 2:

a) A is an array with length $A.length = n \geq 1$ storing values of some *ordered* type T

b) $A[i] < A[i+1]$ for every integer i such that $0 \leq i < n-1$

c) *key* is a value of type T that is *not* stored in A

d) *low* and *high* are integers such that

- $0 \leq low \leq n$
- $-1 \leq high \leq n-1$
- $low \leq high + 1$
- $A[h] < key$ for $0 \leq h < low$
- $A[h] > key$ for $high < h \leq n-1$

- Confirm that the one of the preconditions for the subroutine is satisfied whenever the subroutine is called by the main algorithm (with low = 0 and high = n−1), if A is a sorted array with length n.

int bsearch(int low, int high, int key)

Confirm that the preconditions for the bsearch subroutine each imply that key is not an element of the array A if high = low−1.

d) *low* and *high* are integers such that

- $0 \leq low \leq n$
- $-1 \leq high \leq n-1$
- $low \leq high + 1$
- $A[h] < key$ for $0 \leq h < low$
- $A[h] > key$ for $high < h \leq n-1$

- Confirm that preconditions for the bsearch subroutine each imply that key is not an element of the array A if high = low−1.

- Confirm that the each of the preconditions implies that if low = high then either A[low] = key or key is not an element of the array at all.

In case Low >= High

- We can argue that if bsearch is called with either of its **preconditions satisfie**d, and with **low ≥ high**, then bsearch produces the output it should: The corresponding postcondition is satisfied on termination of the algorithm, because the **algorithm returns an index h such that A[h] = key** if key is an element of the array, and it throws **a notFoundException otherwise.**

In case low < high:

- Argue that if either one of the subroutine's preconditions is satisfied when the algorithm is called, and the algorithm then calls itself recursively, then the same precondition is still satisfied — and the array A and value key have not been changed — at the beginning of the recursive call that is made.

- Suppose:

high – low + 1 = 2h

```
int bsearch(int low, int high, T key)
  if low > high then
    throw KeyNotFoundException
  else
    mid = ⌊(low + high)/2⌋
    if (A[mid] > key) then
      return bsearch(low, mid − 1, key)
    else if (A[mid] < key) then
      return bsearch(mid + 1, high, key)
    else
      return mid
    end if
  end if
```

- Suppose:

high = low + 2h

Show that *if* bsearch calls itself recursively, *then* it does so to search a part of the array that has length at most h in this case, as well.

```
int bsearch(int low, int high, T key)
  if low > high then
    throw KeyNotFoundException
  else
    mid = ⌊(low + high)/2⌋
    if (A[mid] > key) then
      return bsearch(low, mid − 1, key)
    else if (A[mid] < key) then
      return bsearch(mid + 1, high, key)
    else
      return mid
    end if
  end if
```

Analysis of running time

- Show that if the Binary Search is run on an array of length n, for $n \geq 1$, then the total number of calls made to bsearch is at most $\log_2 n + 2$.

Time Complexity of binarySearch

- Call T(n) the time of binarySearch when the array size is n.
- $T(n) = T(n/2) + c$, where c is some constant representing the time of the basis step and the if-statement
- Assume for simplicity that $n= 2^k$. (so $k=\log_2 n$)
- $T(2^k)=T(2^{k-1})+c=T(2^{k-2})+c+c=T(2^{k-3})+c+c+c = \ldots = T(2^0)+c+c+\ldots c=T(1)+kc=O(k)=O(\log n)$
- Therefore, $T(n)=O(\log n)$.