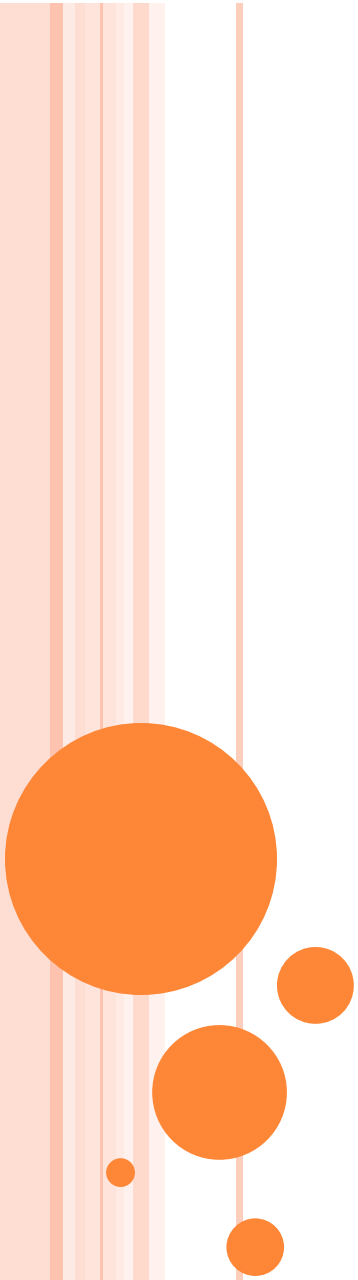


Binary Search

CS A250 – C++ Programming II

Search Algorithms

- Search algorithms are very common
  - They search a list
  - Look at each item in the list and compare to the search item
- We will consider two ways to search:
  - **Linear**
    - Also called **sequential** search
    - Can be done in an **ordered/unordered** list
  - **Binary**
    - Can be done ***only*** in an ***ordered*** list

Linear Search

- **Linear Search** searches through the elements of an arbitrary sequence (in this case, a **vector v**) until
  - The match is found **OR**
  - It reaches the end of the sequence

```
int count = 0;
bool found = false;
int size = static_cast<int>(v.size());
while (!found && count < size)
{
    if (v[count] == value)
       found = true;
    else
       ++count;
}
```

Assumption: All elements in the list are ***unique***.

How Many Comparisons?

- You always look at the **worst case**:
  - Assuming your list has 20 elements
    - You compare each element at most once with the given element
    - If the element you are searching for happens to be the last element in the list:
      - You have made **20 comparisons**
      - **Generalize:** 20 is denoted by ***n*** (a list of ***n*** elements)

- Therefore, the worst case in a sequential search is ***n* comparisons**.

Binary Search

- **Binary search**
  - Is faster than **linear search**
    - **BUT** assumes **array is sorted**
  - Breaks the list in **half**
    - Determines if item in 1st or 2nd half
    - Then searches again just that half
      - Can be done **recursively**.

Execution of Binary Search

- A **sorted** array of 10 elements
  - Search for **63**

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
15	20	35	41	57	63	75	80	85	90

Execution of Binary Search (cont.)

- A **sorted** array of 10 elements
  - Search for **63**

first last

Find the **middle** ➔ (0 + 9) / 2 = **4**

Since indices are type **int** result will be truncated

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
15	20	35	41	57	63	75	80	85	90

mid

Execution of Binary Search (cont.)

- A **sorted** array of 10 elements
  - Search for **63**

Find the **middle** ➔ (0 + 9) / 2 = 4

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
15	20	35	41	57	63	75	80	85	90

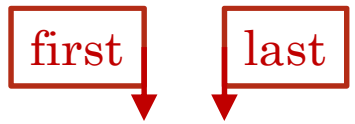
mid

Is [4] equal to **63**? No
Is **63** > or < than [4]? >
Check between [5] and [9]

1st comparison

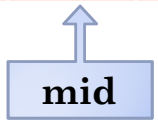
Execution of Binary Search (cont.)

- A **sorted** array of 10 elements
  - Search for **63**



Find the **middle** ➔ (5 + 9) / 2 = **7**

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
15	20	35	41	57	63	75	80	85	90



Execution of Binary Search (cont.)

- A **sorted** array of 10 elements
  - Search for **63**

Find the **middle** ➔ (5 + 9) / 2 = **7**

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
15	20	35	41	57	63	75	80	85	90

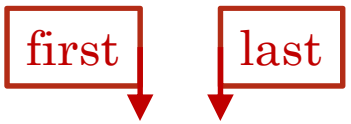
mid

Is [7] equal to **63**? No
Is **63** > or < than [7]? <
Check between [5] and [6]

2nd comparison

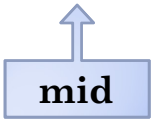
Execution of Binary Search (cont.)

- A **sorted** array of 10 elements
  - Search for **63**



Find the **middle** ➔ (5 + 6) / 2 = **5**

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
15	20	35	41	57	63	75	80	85	90

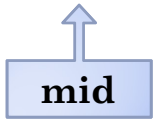


Execution of Binary Search (cont.)

- A **sorted** array of 10 elements
  - Search for **63**

Find the **middle** ➔ (5 + 6) / 2 = **5**

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
15	20	35	41	57	63	75	80	85	90



Is [**5**] equal to **63**? Yes
Stopping case → **63** found
Location = **mid**

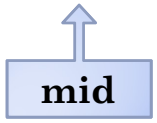
3rd comparison

Execution of Binary Search (cont.)

- A **sorted** array of 10 elements
  - Search for **63**

Number of **comparisons** ➔ **3**

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
15	20	35	41	57	63	75	80	85	90



Another Example

- A **sorted** array of 10 elements
  - Search for **38**

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
15	20	35	41	57	63	75	80	85	90

Another Example (cont.)

- A **sorted** array of 10 elements
  - Search for **38**

first last

Find the **middle** ➔ (0 + 9) / 2 = **4**

Since indices are type **int** result will be truncated

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
15	20	35	41	57	63	75	80	85	90

mid

Another Example (cont.)

- A **sorted** array of 10 elements
  - Search for **38**

Find the **middle** ➔ (0 + 9) / 2 = 4

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
15	20	35	41	57	63	75	80	85	90

mid

Is [4] equal to **38**? No
Is **38** > or < than [4]? <
Check between [0] and [3]

1st comparison

Another Example (cont.)

- A **sorted** array of 10 elements
  - Search for **38**

first last

Find the **middle** ➔ (0 + 3) / 2 = **1**

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
15	20	35	41	57	63	75	80	85	90

mid

Another Example (cont.)

- A **sorted** array of 10 elements
  - Search for **38**

Find the **middle** ➔ (0 + 3) / 2 = **1**

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
15	20	35	41	57	63	75	80	85	90

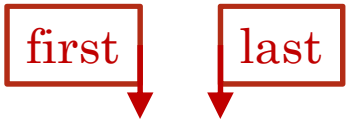
mid

Is [1] equal to **38**? No
Is **38** > or < than [1]? >
Check between [2] and [3]

2nd comparison

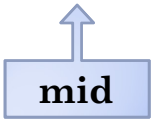
Another Example (cont.)

- A **sorted** array of 10 elements
  - Search for **38**



Find the **middle** ➔ (2 + 3) / 2 = **2**

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
15	20	35	41	57	63	75	80	85	90



Another Example (cont.)

- A **sorted** array of 10 elements
  - Search for **38**

Find the **middle** ➔ (2 + 3) / 2 = **2**

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
15	20	35	41	57	63	75	80	85	90

mid

Is [2] equal to **38**? No
Is **38** > or < than [2]? >
Check between [3] and [3]

3rd comparison

Another Example (cont.)

- A **sorted** array of 10 elements
  - Search for **38**

Find the **middle** ➔ (2 + 3) / 2 = **2**

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
15	20	35	41	57	63	75	80	85	90

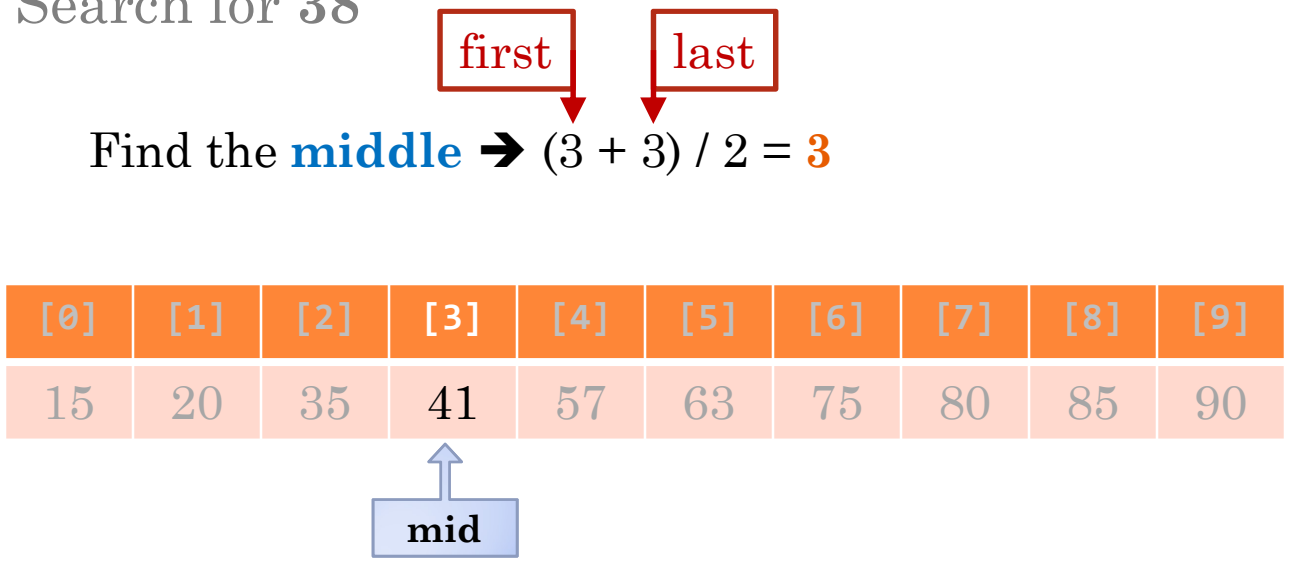
mid

Is [2] equal to **38**? No
Is **38** > or < than [2]? >
Check between [3] and [3]

You need to check until **first > last**

Another Example (cont.)

- A **sorted** array of 10 elements
  - Search for **38**



Another Example (cont.)

- A **sorted** array of 10 elements
  - Search for **38**

Find the **middle** ➔ (3 + 3) / 2 = **3**

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
15	20	35	41	57	63	75	80	85	90

mid

Is [3] equal to **38**? No
Is **38** > or < than [3]? <
Check between [3] and [2]

first > last

Another Example (cont.)

- A **sorted** array of 10 elements
  - Search for **38**

Number of **comparisons** ➔ 4

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
15	20	35	41	57	63	75	80	85	90

Key 38 not found.

IMPLEMENTATION

- **Binary search** can be implemented
  - **Iteratively**
    - Using a **while** loop
  - **Recursively**
    - **IF/ELSE** statement and **call to itself**

  - **Stopping case** for both implementations:
    - **if (first > last)** → no elements between them, so **key** cannot be there!
    - **if (key == a[mid])** → found!

Efficiency of Binary Search

- **Binary search** is ***very efficient***
  - Extremely ***fast,*** compared to **sequential search**
- Half of array eliminated at start!
  - Then a quarter, then 1/8, etc.
  - Essentially eliminate half with each call
- Example: Array of 100 elements
  - In this case, a binary search never needs more than 7 compares!

STL ALGORITHM: binary_search

- The **STL <algorithm>** header has a **binary_search** function that performs a binary search on **sorted** containers

```
template <class ForwardIterator, class T>
     bool binary_search (ForwardIterator first,
                ForwardIterator last, const T& val);
```

STL ALGORITHM: binary_search (cont.)

Using the **binary_search** function on a **sorted** vector.

```
vector<int> myVector = { 23, 34, 45, 56, 58, 61, 73, 88 };

if ( binary_search(myVector.begin(), myVector.end(), 61) )
    cout << "Value 61 was found." << endl;
else
    cout << "Value 61 was not found." << endl;
```

Binary Search (end)