CSE 250
Data Structures

Dr. Eric Mikida
epmikida@buffalo.edu
208 Capen Hall

Lec 15: Binary Search

Announcements

- WA2 due yesterday, submissions close tonight!
    - Answer key will be released tomorrow for study purposes
- Midterm Friday – see Piazza @213

Recap - Merge Sort

Divide: Split the sequence in half

$D(n) = \Theta(n)$ (can do in $\Theta(1)$)

Conquer: Sort the left and right halves

$a = 2$, $b = 2$, $c = 1$

Combine: Merge halves together

$C(n) = \Theta(n)$

Benefits of a Sorted List

So in ***O*(*n* log(*n*))** we can sort a list using the merge sort algorithm…

But how does that benefit us?

Binary vs Linear Search

Consider searching for a particular value in an **Array** (or **ArrayList**)...

How long does that search take?

Binary vs Linear Search

Consider searching for a particular value in an **Array** (or **ArrayList**)...

How long does that search take? ***O*(*n*)**, we have to check all ***n*** elements

This is called a **Linear Search** (it takes linear time)

Binary vs Linear Search

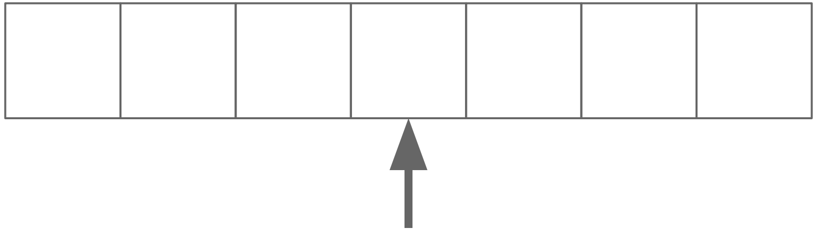
Consider searching for a particular value in an **Array** (or **ArrayList**)...

How long does that search take? ***O*(*n*)**, we have to check all ***n*** elements

This is called a **Linear Search** (it takes linear time)

What if our list is sorted? Can we do better?

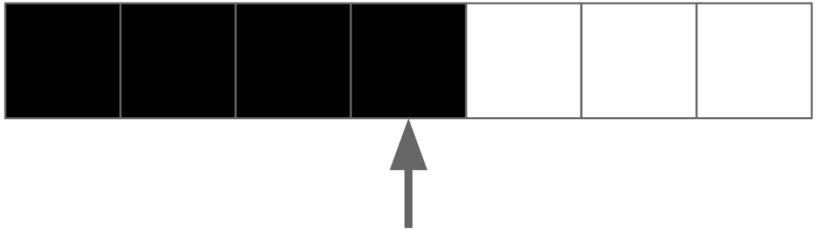
Binary vs Linear Search



Check the middle element (which we can access in constant time)

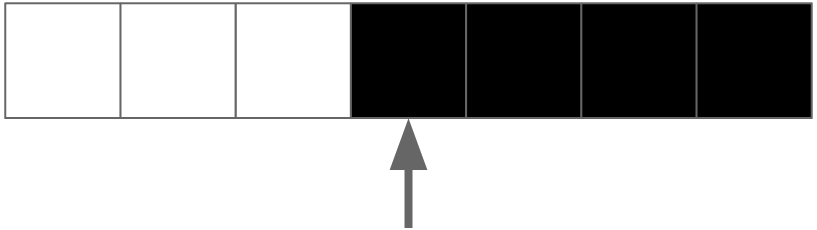
Binary vs Linear Search

We can ignore half the list



Check the middle element (which we can access in constant time)
If it is smaller than what we are looking for, then our target must be to the right (because our list is sorted)

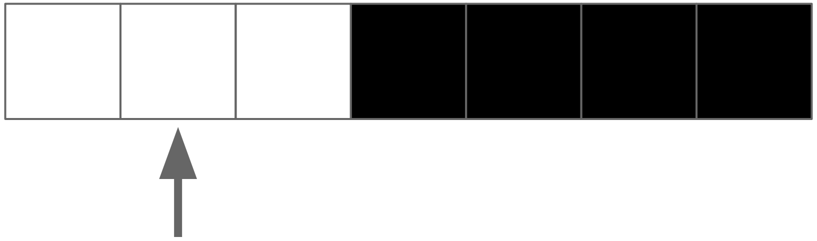
Binary vs Linear Search



We can ignore half the list

Check the middle element (which we can access in constant time)
If it is larger than what we are looking for, then our target must be to the left (because our list is sorted)

Binary vs Linear Search



We can ignore half the list

Check the middle element (which we can access in constant time)
If it is larger than what we are looking for, then our target must be to the left (because our list is sorted)
Repeat this process recursively with the remaining elements

Binary vs Linear Search

We can ignore half the list

Check the middle element (which we can access in constant time)
If it is larger than what we are looking for, then our target must be to the left (because our list is sorted)
Repeat this process recursively with the remaining elements

What is the runtime to search in this fashion?

Binary vs Linear Search

We can ignore half the list

Check the middle element (which we can access in constant time)
If it is larger than what we are looking for, then our target must be to the left (because our list is sorted)
Repeat this process recursively with the remaining elements

What is the runtime to search in this fashion? ***O*(log(*n*))**

Binary vs Linear Search

Linear search:

- Removes one element from consideration each step, ***O*(*n*)**
- Does not require list to be sorted
- Does not require constant time random access

Binary search:

- Removes half of the elements from consideration each step, ***O*(log(*n*))**
- Requires list to be sorted
- **Requires constant time random access**
    - (binary search on a linked list is still linear time…)

Sets and Bags (...so far)

	LinkedList	ArrayList
`Set.add`	$O(n)$	$O(n)$
`Set.contains`	$O(n)$	$O(n)$
`Set.remove`	$O(n)$	$\Theta(n)$
`Bag.add`	$O(1)$	$O(n)$, Amortized $\Theta(1)$
`Bag.contains`	$\Theta(n)$	$\Theta(n)$
`Bag.remove`	$O(n)$	$\Theta(n)$

Potential Improvements

How could we improve these implementations?

Thought…does order matter for sets?

Potential Improvements

How could we improve these implementations?

Thought…does order matter for sets? **No!**

Can we somehow take advantage of that?

Small Improvement

Notice how the ArrayList version of remove was $\Theta(n)$ because we had to shift over elements to fill the hole after removing the target…

If we don't need to maintain order, we don't need to shift everything to fill the hole, we can just fill it with the last item!

Set Pseudocode (w/ArrayList)

Runtime?

```
ArrayList<T> data;
remove(elem):
  idx ← 0
  while idx < data.size():
    if data[idx] == elem:
      data[idx] =
          data[data.size()-1]
      data.remove(data.size()-1)
      return true
    idx = idx + 1
  return false
```

Set Pseudocode (w/ArrayList)

Runtime? Still ***O***(***n***)...but now Ω(1)

Just a tactical optimization, doesn't change the asymptotic runtime...

```
ArrayList<T> data;
remove(elem):
  idx ← 0
  while idx < data.size():
    if data[idx] == elem:
      data[idx] =
          data[data.size()-1]
      data.remove(data.size()-1)
      return true
    idx = idx + 1
  return false
```

Slightly Better Improvement

What if we were to store elements in sorted order instead of the order they were added…

Slightly Better Improvement

What if we were to store elements in sorted order instead of the order they were added…

`contains` can now use binary search instead of linear search!

`contains` becomes an $\boldsymbol{O}(\mathbf{log}(\boldsymbol{n}))$ operation

Slightly Better Improvement

What if we were to store elements in sorted order instead of the order they were added…

What about `add`/`remove`?

`add` **must** insert elements in sorted order, but that still takes ***O***(***n***)

`remove` can find the element faster…but will still have to shift elements to fill the hole, so still takes ***O***(***n***)

Sets and Bags (...so far)

	LinkedList	ArrayList	ArrayList (sorted)
`Set.add`	$O(n)$	$O(n)$	$O(n)$
`Set.contains`	$O(n)$	$O(n)$	$O(\log(n))$
`Set.remove`	$O(n)$	$\Theta(n)$	$O(n)$
`Bag.add`	$O(1)$	$O(n)$, Amortized $\Theta(1)$	
`Bag.contains`	$\Theta(n)$	$\Theta(n)$	
`Bag.remove`	$O(n)$	$\Theta(n)$	

Sets and Bags (...so far)

	LinkedList	ArrayList	ArrayList (sorted)
`Set.add`	$O(n)$	$O(n)$	$O(n)$
`Set.contains`	$O(n)$	$O(n)$	$O(\log(n))$
`Set.remove`	$O(n)$	$\Theta(n)$	$O(n)$
`Bag.add`	$O(1)$	$O(n)$, Amortized $\Theta(1)$	
`Bag.contains`	$\Theta(n)$	$\Theta(n)$	What about Bag?
`Bag.remove`	$O(n)$	$\Theta(n)$	

Sets and Bags (...so far)

	LinkedList	ArrayList	ArrayList (sorted)
`Set.add`	$O(n)$	$O(n)$	$O(n)$
`Set.contains`	$O(n)$	$O(n)$	$O(\log(n))$
`Set.remove`	$O(n)$	$O(n)$	$O(n)$
`Bag.add`	$O(1)$	1)	$O(n)$
`Bag.contains`	$\Theta(n)$		$O(n)$
`Bag.remove`	$O(n)$	$\Theta(n)$	$O(n)$

We have to count all the duplicates and there could be as many as ***n***