ArraySet and Binary Search

Lecture 16

Menu

- Cost of ArraySet operations
- Binary Search
- Cost of SortedArraySet with Binary Search

ArrayList costs: Summary

• get	O(1)	
• set	O(1)	
• remove	O(n)	
• add (at i) (have to shift up may have to double capacity)	O(n)	(worst and average)
• add (at end)	O(1)	(most of the time)
(when doubles cap)	O(n)	(worst)
	O(1)	(amortised average) (if doubled each time)

ArraySet costs

What about ArraySet:

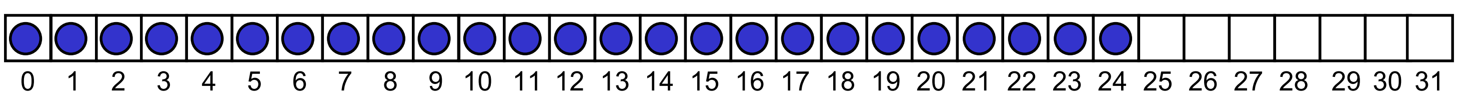
- Order is not significant

  ⇒ add() can choose to put a new item anywhere. where?

  ⇒ can reorder when removing an item. how?

- **Duplicates not allowed**.

  ⇒ must check if item already present before adding



ArraySet algorithms

Costs?

Contains(value):

search through array,
    if value equals item
        return true
return false

Add(value):

if not contains(value),
    place value at end, (doubling array if necessary)
    increment size

Remove(value):

search through array
    if value equals item
        replace item by item at end. (why?)
        decrement size
        return

ArraySet costs

<u>Costs</u>:

- contains, add, remove: O(n)
-

Question:

- How can we speed up the search?

Making ArraySet faster.

All the cost is in the searching:

- Searching for “Eel”

8

Bee	Dog	Ant	Fox	Hen	Gnu	Eel	Cat	
0	1	2	3	4	5	6	7	8

- but if sorted…

8

Ant	Bee	Cat	Dog	Eel	Fox	Gnu	Hen	

Making ArraySet faster

- Binary Search: Finding “Eel”

8

Ant	Bee	Cat	Dog	Eel	Fos	Gnu	Pig	
0	1	2	3	4	5	6	7	8

- If the items are sorted (“ordered”), then we can search fast
  - Look in the middle:
    - if item is middle item ⇒ return
    - if item is before middle item ⇒ look in left half
    - if item is after middle item ⇒ look in right half

Divide and Conquer

One of the **best-known** algorithm design techniques.

Idea:

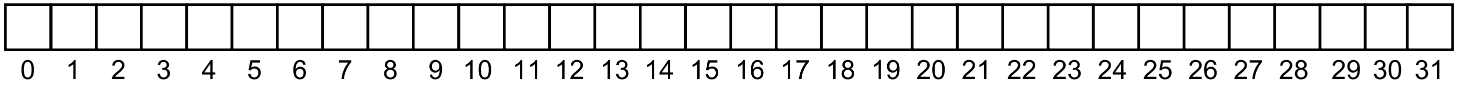
- A problem instance is ***divided*** into several **smaller** instances of the same problem, ideally of about same size
- The smaller instances are **solved**, typically **recursively**
- The solutions for the smaller instances are ***combined*** to get a solution to the original problem

Binary Search

```
private boolean contains(Object item){
    Comparable<E> value = (Comparable<E>) item;
    int low = 0;                                   // min possible index of item
    int high = count-1;                            // max possible index of item
                                                   // item in [low .. high]  (if present)
    while (low <= high){
        int mid  =  (low + high) / 2;
        int comp =  value.compareTo(data[mid]);
        if (comp == 0)                             // item is present
            return true;
        if (comp < 0)                              // item in [low .. mid-1]
            high = mid - 1;                        // item in [low .. high]
        else                                       // item in [mid+1 .. high]
            low = mid + 1;                         // item in [low .. high]
    }
    return false;    // item in [low .. high] and low > high,
                     // therefore item not present
}
```

Binary Search: Cost

- What is the cost of searching if *n* items in set?
  - key step = ?



- Iteration Size of range Cost of iteration

Iteration	Size of range	Cost of iteration
1	n	
2		
k	1	

Time complexity

Let $T(n)$ denote the time complexity of binary search algorithm on n numbers.

$$T(n) = \begin{cases} 1 & \text{if } n=1 \\ T(n/2) + 1 & \text{otherwise} \end{cases}$$

We call this formula a **recurrence**.

Recurrence

A recurrence is an equation or inequality that describes a function in terms of ***its value on smaller inputs***.

E.g.,

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n/2) + 1 & \text{if } n > 1 \end{cases}$$

To **solve** a recurrence is to derive ***asymptotic bounds*** on the solution

$\text{Log}_2(n)$ or $\log(n)$

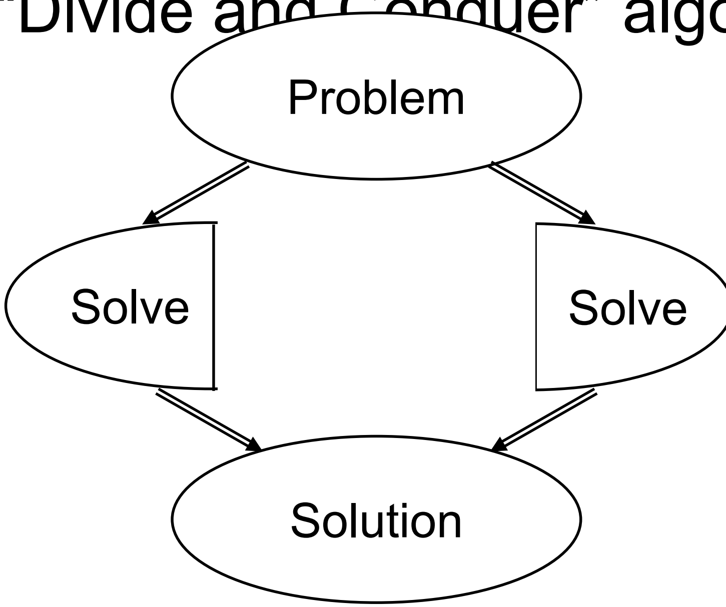
The number of times you can divide a set of *n* things in half.

log(1000) =10, log(1,000,000) = 20, log(1,000,000,000) =30

Every time you double *n*, you add one step to the cost! (why?)

- log(2n) = log(n)+log2 = log(n) +1
- Arises all over the place in analysing algorithms

Especially “Divide and Conquer” algorithms:



Substitution method

$$T(n) = \begin{cases} 1 & \text{if } n=1 \\ T(n/2) + 1 & \text{otherwise} \end{cases}$$

Make a guess, $T(n) \le 2 \log n$

We prove statement by *MI*.

Base case? When n=1, statement is FALSE!

L.H.S = $T(1) = 1$ R.H.S = $c \log 1 = 0 <$ L.H.S

Yet, when n=2,

L.H.S = $T(2) = T(1)+1 = 2$

R.H.S = $2 \log 2 = 2$

L.H.S $\le$ R.H.S

Substitution method

$$T(n) = \begin{cases} 1 & \text{if } n=1 \\ T(n/2)+1 & \text{otherwise} \end{cases}$$

Make a guess, $T(n) \le 2 \log n$

We prove statement by *MI*.

Assume true for all $n' < n$ [assume $T(n/2) \le 2 \log (n/2)$]

$T(n) = T(n/2) + 1$

$\le 2 \log (n/2) + 1$ ← *by hypothesis*

$= 2(\log n - 1) + 1$ ← ***$\log(n/2) = \log n - \log 2$***

$< 2\log n$

$\boxed{\textbf{i.e., } T(n) \le 2 \log n}$

More Example

Prove that $T(n) = \begin{cases} 1 & \text{if } n=1 \\ 2T(n/2) + n & \text{otherwise} \end{cases}$ is $O(n \log n)$

Guess: $T(n) \le 2\, n \log n$

More Example

Prove that $T(n) = \begin{cases} 1 & \text{if } n=1 \\ 2T(n/2) + n & \text{otherwise} \end{cases}$ is $O(n \log n)$

Guess: $T(n) \leq 2 n \log n$

Assume true for all $n' < n$ [assume $T(n/2) \leq 2 (n/2) \log(n/2)$]

$T(n)$

$\leq 2 (2 (n/2) \log (n/2)) + n$

$= 2 n (\log n - 1) + n$

$= 2 n \log n - 2n + n$

$\leq 2 n \log n$

For the base case when $n=2$,
L.H.S = $T(2) = 2T(1)+2 = 4$,
R.H.S = $2 * 2 \log 2 = 4$
L.H.S $\leq$ R.H.S

$\boxed{\textbf{i.e., } T(n) \leq 2 n \log n}$

ArraySet with Binary Search

ArraySet: unordered

- All cost in the searching: O(n)
  - contains: $O(n)$
  - add: $O(n)$
  - remove: $O(n)$

SortedArraySet: with Binary Search

- Binary Search is fast: $O(\log(n))$
  - contains: $O(\log(n))$
  - add:
  - remove:

- All the cost is in keeping it sorted!!!!

Making SortedArraySet fast

- If you have to call add() and/or remove() many items, then SortedArraySet is no better than ArraySet!
  - Both O(*n*)
  - Either pay to search
  - Or pay to keep it in order
- If you only have to construct the set once, and then many calls to contains(),
  then SortedArraySet is much better than ArraySet.
  - SortedArraySet contains() is O(log(*n*))
- But, how do you construct the set fast?
  - Adding each item, one at a time

Alternative Constuctor

- Sort the items all at once

```
public SortedArraySet(Collection<E> col){
    // Make space
    count=col.size();
    data = (E[]) new Object[count];

    // Put items from collection into the data array.
    col.toArray(data);

    // sort the data array.
    Arrays.sort(data);
}
```

- How do you sort?

Summary

- Cost of ArraySet operations
- Binary Search
- Cost of SortedArraySet with Binary Search

Readings

- [Mar07] Read 4.3
- [Mar13] Read 4.3