Amortized Analysis and Heaps Intro

David Kauchak
cs140
Fall 2022

1

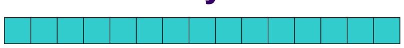
Admin

Assignment 3 back soon (sorry for the delay!)

Assignment 4 due Sunday

2

Extensible array



Sequential locations in memory in linear order

Elements are accessed via index
- Access of particular indices is O(1)

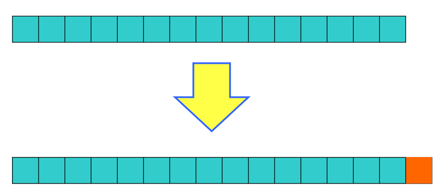
Say we want to implement an array that supports *add* (i.e. *addToBack*)
- ArrayList or Vector in Java
- lists in Python, perl, Ruby, …

How can we do it?

3

Extensible array

Idea 1: Each time we call *add*, create a new array one element large, copy the data over and add the element



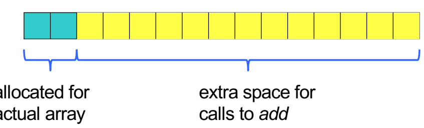
Running time: $\Theta(n)$

4

Extensible array

Idea 2: Allocate extra, unused memory and save room to add elements

For example: new ArrayList(2)

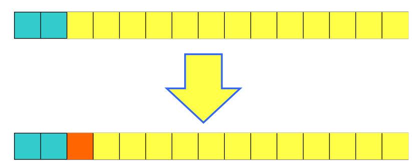


5

Extensible array

Idea 2: Allocate extra, unused memory and save room to add elements

Adding an item:



Running time: Θ(1) Problems?

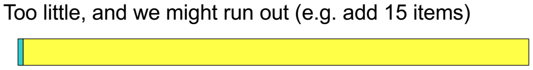
6

Extensible array

Idea 2: Allocate extra, unused memory and save room to add elements

How much extra space do we allocate?

Too little, and we might run out (e.g. add 15 items)



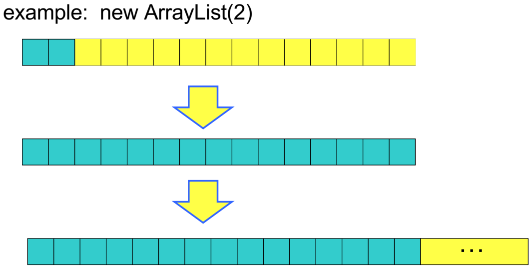
Too much, and we waste lots of memory Ideas?

7

Extensible array

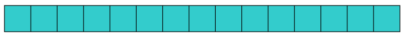
Idea 3: Allocate some extra memory and when it fills up, allocate some more and copy

For example: new ArrayList(2)



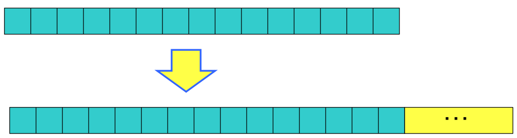
8

Extensible array



Idea 3: Allocate some extra memory and when it fills up, allocate some more and copy

For example: new ArrayList(2)



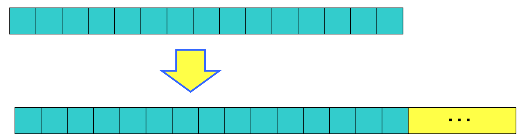
Running time: Θ(n)

9

Extensible array

Idea 3: Allocate some extra memory and when it fills up, allocate some more and copy

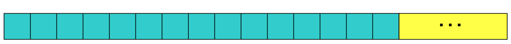
For example: new ArrayList(2)



How much extra memory should we allocate?

10

Extensible array



Challenge: most of the calls to *add* will be O(1)

How else might we talk about runtime?

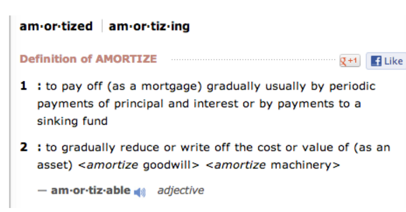
What is the ***average*** running time of *add* in the ***worst case***?

- Note this is different than the *average-case* running time

11

Amortized analysis

What does "amortize" mean?



am·or·tized | **am·or·tiz·ing**

Definition of AMORTIZE

1 **:** to pay off (as a mortgage) gradually usually by periodic payments of principal and interest or by payments to a sinking fund

2 **:** to gradually reduce or write off the cost or value of (as an asset) <*amortize* goodwill> <*amortize* machinery>

— **am·or·tiz·able** *adjective*

12

What are the costs?

Insertion: 1 2 3 4 5 6 7 8 9 10

size: 1 2 4 4 8 8 8 8 16 16

cost:

17

What are the costs?

Insertion: 1 2 3 4 5 6 7 8 9 10

size: 1 2 4 4 8 8 8 8 16 16

cost: 1 2 3 1 5 1 1 1 9 1

18

What are the costs?

Insertion: 1 2 3 4 5 6 7 8 9 10

size: 1 2 4 4 8 8 8 8 16 16

basic cost: 1 1 1 1 1 1 1 1 1 1

double cost: 0 1 2 0 4 0 0 0 8 0

19

What are the costs?

Insertion: 1 2 3 4 5 6 7 8 9 10

size: 1 2 4 4 8 8 8 8 16 16

basic cost: 1 1 1 1 1 1 1 1 1 1

double cost: 0 1 2 0 4 0 0 0 8 0

What is the sum of basic cost for n operations?

What is the sum of the copy cost for n operations?

20

Amortized analysis

More generally:

total_cost(n) = basic_cost(n) + double_cost(n)

basic_cost(n) = n

double_cost(n) = 1 + 2 + 4 + 8 + … + n = 2n

total_cost(n) = 3n

over n operations:
amortized *O(1)*

21

Amortized analysis vs. worse case

What is the worse case of *add*?

- Still O(n)
- If you have an application that needs it to be O(1), this implementation **will not work!**

amortized analysis give you the cost of *n* operations (i.e. average cost) **not** the cost of any individual operation

22

Extensible arrays

What if instead of doubling the array, we add instead increase the array by a fixed amount (call it *k*) each time

Is the amortized run-time still O(1)?

- No!
- Why?

23

Amortized analysis

Consider the cost of *n* insertions for some constant *k*

total_cost(n) = basic_cost(n) + double_cost(n)

basic_cost(n) = $O(n)$

double_cost(n) =k+2k+3k+4k+5k+...+n

$$= \sum_{i=1}^{n/k} ki$$

$$= k \sum_{i=1}^{n/k} i$$

$$= k \frac{\frac{n}{k}\left(\frac{n}{k}+1\right)}{2} = O(n^2)$$

24

Amortized analysis

Consider the cost of *n* insertions for some constant *k*

$$\text{total\_cost(n)} = O(n) + O(n^2)$$
$$= O(n^2)$$

amortized *O(n)!*

25

Accounting method

Each operation has an amount we charge to accomplish it (this is really the run-time for this operation)

We deduct from that charge the actual cost of the operation

If there is anything left over, put it in the bank

An operation may also use the bank to offset the cost of the operation

Key idea: charge more for low-cost operations and save that up to offset the cost of expensive operations

26

Insertion:	1	2	3	4	5	6	7	8	9	10
size:	1	2	4	4	8	8	8	8	16	16
cost:	1	2	3	1	5	1	1	1	9	1
bank:										

How much should we pay for each insert?

27

Insertion:	1	2	3	4	5	6	7	8	9	10
size:	1	2	4	4	8	8	8	8	16	16
cost:	1	2	3	1	5	1	1	1	9	1
bank:										

Try insert: 2

28

Insertion:	1	2	3	4	5	6	7	8	9	10
size:	1	2	4	4	8	8	8	8	16	16
cost:	1	2	3	1	5	1	1	1	9	1
bank:										

Try insert: 2

How much is left?

29

Insertion:	1	2	3	4	5	6	7	8	9	10
size:	1	2	4	4	8	8	8	8	16	16
cost:	1	2	3	1	5	1	1	1	9	1
bank:	1									

Try insert: 2

30

Insertion:	1	2	3	4	5	6	7	8	9	10
size:	1	2	4	4	8	8	8	8	16	16
cost:	1	2	3	1	5	1	1	1	9	1
bank:	1									

Try insert: 2

How much is left?

31

Insertion:	1	2	3	4	5	6	7	8	9	10
size:	1	2	4	4	8	8	8	8	16	16
cost:	1	2	3	1	5	1	1	1	9	1
bank:	1	1								

Try insert: 2

32

Insertion:	1	2	3	4	5	6	7	8	9	10
size:	1	2	4	4	8	8	8	8	16	16
cost:	1	2	3	1	5	1	1	1	9	1
bank:	1	1	0							

Try insert: 2

33

Insertion:	1	2	3	4	5	6	7	8	9	10
size:	1	2	4	4	8	8	8	8	16	16
cost:	1	2	3	1	5	1	1	1	9	1
bank:	1	1	0	1						

Try insert: 2

34

Insertion:	1	2	3	4	5	6	7	8	9	10
size:	1	2	4	4	8	8	8	8	16	16
cost:	1	2	3	1	5	1	1	1	9	1
bank:	1	1	0	1						

Try insert: 2

How much is left?

35

Insertion:	1	2	3	4	5	6	7	8	9	10
size:	1	2	4	4	8	8	8	8	16	16
cost:	1	2	3	1	5	1	1	1	9	1
bank:	1	1	0	1						

Try insert: 2

-2!!

36

Insertion:	1	2	3	4	5	6	7	8	9	10
size:	1	2	4	4	8	8	8	8	16	16
cost:	1	2	3	1	5	1	1	1	9	1
bank:										

Try insert: ??

37

Insertion:	1	2	3	4	5	6	7	8	9	10
size:	1	2	4	4	8	8	8	8	16	16
cost:	1	2	3	1	5	1	1	1	9	1
bank:										

Try insert: 3

How much is left?

38

Insertion:	1	2	3	4	5	6	7	8	9	10
size:	1	2	4	4	8	8	8	8	16	16
cost:	1	2	3	1	5	1	1	1	9	1
bank:	2									

Try insert: 3

39

Insertion:	1	2	3	4	5	6	7	8	9	10
size:	1	2	4	4	8	8	8	8	16	16
cost:	1	2	3	1	5	1	1	1	9	1
bank:	2	3								

Try insert: 3

40

Insertion: 1 2 3 4 5 6 7 8 9 10

size: 1 2 4 4 8 8 8 8 16 16

cost: 1 2 3 1 5 1 1 1 9 1

bank: 2 3 3

Try insert: 3

41

Insertion: 1 2 3 4 5 6 7 8 9 10

size: 1 2 4 4 8 8 8 8 16 16

cost: 1 2 3 1 5 1 1 1 9 1

bank: 2 3 3

Try insert: 3

42

Insertion: 1 2 3 4 5 6 7 8 9 10

size: 1 2 4 4 8 8 8 8 16 16

cost: 1 2 3 1 5 1 1 1 9 1

bank: 2 3 3 5

Try insert: 3

43

Insertion: 1 2 3 4 5 6 7 8 9 10

size: 1 2 4 4 8 8 8 8 16 16

cost: 1 2 3 1 5 1 1 1 9 1

bank: 2 3 3 5 3 5 7 9

Try insert: 3

44

Insertion:	1	2	3	4	5	6	7	8	9	10
size:	1	2	4	4	8	8	8	8	16	16
cost:	1	2	3	1	5	1	1	1	9	1
bank:	2	3	3	5	3	5	7	9	3	

Try insert: 3

45

Insertion:	1	2	3	4	5	6	7	8	9	10
size:	1	2	4	4	8	8	8	8	16	16
cost:	1	2	3	1	5	1	1	1	9	1
bank:	2	3	3	5	3	5	7	9	3	

Try insert: 3

Will this work??

46

Insertion:	1	2	3	4	5	6	7	8	9	10
size:	1	2	4	4	8	8	8	8	16	16
cost:	1	2	3	1	5	1	1	1	9	1
bank:	2	3	3	5	3	5	7	9		

last copy happened here

Try insert: 3

1: pay for our operation
Getting ready for the copy:
1: pay for our copy
1: pay to copy from an item first half

47

Accounting method

Insert pay 3 = O(1)!

Particularly useful when there are multiple operations

48

Another set data structure

We want to support fast lookup and insertion (i.e. faster than linear)

Arrays can easily made to be fast for one or the other
- fast search: keep list sorted
  - O(n) insert
  - O(log n) search
- fast insert: extensible array
  - O(1) insert (amortized)
  - O(n) search

49

Another set data structure

Idea: store data in a collection of arrays
- array *i* has size 2^i
- an array is either full or empty (never partially full)
- each array is stored in sorted order
- no relationship between arrays

50

Another set data structure

Which arrays are full and empty are based on the number of elements
- specifically, binary representation of the number of elements
- 4 items = 100 = A2-full, A1-empty, A_0-empty
- 11 items = 1011 = A_3-full, A_2-empty, A_1-full, A_0-full

A_0: [5]
A_1: [4, 8]
A_2: empty
A_3: [2, 6, 9, 12, 13, 16, 20, 25]

Lookup: binary search through each array
- Worse case runtime?

51

Another set data structure

A_0: [5]
A_1: [4, 8]
A_2: empty
A_3: [2, 6, 9, 12, 13, 16, 20, 25]

Lookup: binary search through each array

Worse case: all arrays are full
- number of arrays = number of digits = log n
- binary search cost for each array = O(log n)
- O(log n log n)

52

Another set data structure

Insert(A, item)

- starting at i = 0
- current = [item]
- as long as the level *i* is full
  - merge current with A_i using *merge* procedure
  - store to current
  - A_i = empty
  - i++
- A_i = current

53

Insert 5

A_0: empty

Insert

- starting at i = 0
- current = [item]
- as long as the level *i* is full
  - merge current with A_i using *merge* procedure
  - store to current
  - A_i = empty
  - i++
- A_i = current

54

Insert 5

A_0: [5]

Insert

- starting at i = 0
- current = [item]
- as long as the level *i* is full
  - merge current with A_i using *merge* procedure
  - store to current
  - A_i = empty
  - i++
- A_i = current

55

Insert 6

A_0: [5]

Insert

- starting at i = 0
- current = [item]
- as long as the level *i* is full
  - merge current with A_i using *merge* procedure
  - store to current
  - A_i = empty
  - i++
- A_i = current

56

Insert 6

A_0: empty
A_1: [5, 6]

Insert
- starting at i = 0
- current = [item]
- as long as the level *i* is full
  - merge current with A_i using *merge* procedure
  - store to current
  - A_i = empty
  - i++
- A_i = current

57

Insert 12

A_0: empty
A_1: [5, 6]

Insert
- starting at i = 0
- current = [item]
- as long as the level *i* is full
  - merge current with A_i using *merge* procedure
  - store to current
  - A_i = empty
  - i++
- A_i = current

58

Insert 12

A_0: [12]
A_1: [5, 6]

Insert
- starting at i = 0
- current = [item]
- as long as the level *i* is full
  - merge current with A_i using *merge* procedure
  - store to current
  - A_i = empty
  - i++
- A_i = current

59

Insert 4

A_0: [12]
A_1: [5, 6]

Insert
- starting at i = 0
- current = [item]
- as long as the level *i* is full
  - merge current with A_i using *merge* procedure
  - store to current
  - A_i = empty
  - i++
- A_i = current

60

Insert 4

A_0: empty
A_1: empty
A_2: [4, 5, 6, 12]

Insert
- starting at i = 0
- current = [item]
- as long as the level *i* is full
  - merge current with A_i using *merge* procedure
  - store to current
  - A_i = empty
  - i++
- A_i = current

61

Insert 23

A_0: empty
A_1: empty
A_2: [4, 5, 6, 12]

Insert
- starting at i = 0
- current = [item]
- as long as the level *i* is full
  - merge current with A_i using *merge* procedure
  - store to current
  - A_i = empty
  - i++
- A_i = current

62

Insert 23

A_0: [23]
A_1: empty
A_2: [4, 5, 6, 12]

Insert
- starting at i = 0
- current = [item]
- as long as the level *i* is full
  - merge current with A_i using *merge* procedure
  - store to current
  - A_i = empty
  - i++
- A_i = current

63

Another set data structure

Insert
- starting at i = 0
- current = [item]
- as long as the level *i* is full
  - merge current with A_i using *merge* procedure
  - store to current
  - A_i = empty
  - i++
- A_i = current

running time?

64

Insert running time

Worst case

- merge at each level
- 2 + 4 + 8 + … + n/2 + n = O(n)

There are many insertions that won't fall into this worse case

What is the amortized worse case for insertion?

65

insert: amortized analysis

Consider inserting *n* numbers

- how many times will A_0 be empty?
- how many times will we need to merge with A_0?
- how many times will we need to merge with A_1?
- how many times will we need to merge with A_2?
- …
- how many times will we need to merge with $A_{\log n}$?

66

insert: amortized analysis

Consider inserting *n* numbers

	times
how many times will A_0 be empty?	n/2
how many times will we need to merge with A_0?	n/2
how many times will we need to merge with A_1?	n/4
how many times will we need to merge with A_2?	n/8
…	
how many times will we need to merge with $A_{\log n}$?	1

cost of each of these steps?

67

insert: amortized analysis

- Consider inserting *n* numbers

	times	cost
how many times will A_0 be empty?	n/2	O(1)
how many times will we need to merge with A_0?	n/2	2
how many times will we need to merge with A_1?	n/4	4
how many times will we need to merge with A_2?	n/8	8
…		
how many times will we need to merge with $A_{\log n}$?	1	n

total cost:

68

insert: amortized analysis

- Consider inserting *n* numbers

	times	cost
how many times will A_0 be empty?	n/2	O(1)
how many times will we need to merge with A_0?	n/2	2
how many times will we need to merge with A_1?	n/4	4
how many times will we need to merge with A_2?	n/8	8
…		
how many times will we need to merge with $A_{\log n}$?	1	n

total cost: log n levels * O(n) each level
O(n log n) cost for *n* inserts
O(log n) amortized cost!

69

Binary heap

70

Binary heap

A binary tree where the value of a parent is greater than or equal to the value of its children

Additional restriction: all levels of the tree are **complete** except the last

Max heap vs. min heap

71

Binary heap - operations

Maximum(S) - return the largest element in the set

ExtractMax(S) – Return and remove the largest element in the set

Insert(S, val) – insert val into the set

IncreaseElement(S, x, val) – increase the value of element x to val

BuildHeap(A) – build a heap from an array of elements

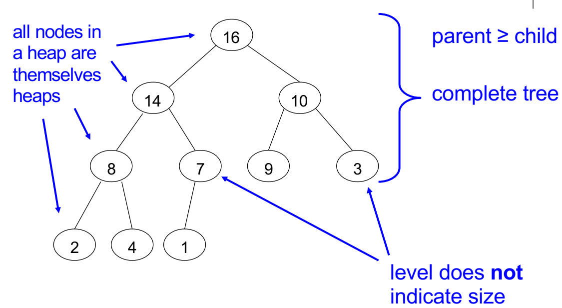
72

Binary heap

How can we represent a heap?

73

Binary heap - references



74

Binary heap - array

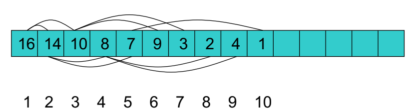
PARENT(i)
    return $\lfloor i/2 \rfloor$

LEFT(i)
    return $2i$

RIGHT(i)
    return $2i + 1$

75

Binary heap - array



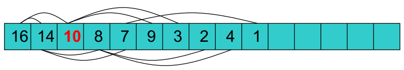
PARENT(i)
    return $\lfloor i/2 \rfloor$

LEFT(i)
    return $2i$

RIGHT(i)
    return $2i + 1$

76

Binary heap - array



1 2 3 4 5 6 7 8 9 10

Left child of A[3]?

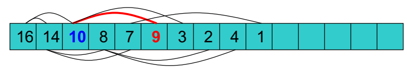
PARENT(i)
return $\lfloor i/2 \rfloor$

LEFT(i)
return $2i$

RIGHT(i)
return $2i+1$

77

Binary heap - array



1 2 3 4 5 6 7 8 9 10

Left child of A[3]?

2*3 = 6

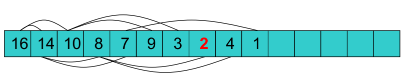
PARENT(i)
return $\lfloor i/2 \rfloor$

LEFT(i)
return $2i$

RIGHT(i)
return $2i+1$

78

Binary heap - array



1 2 3 4 5 6 7 8 9 10

Parent of A[8]?

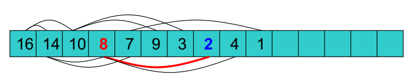
PARENT(i)
return $\lfloor i/2 \rfloor$

LEFT(i)
return $2i$

RIGHT(i)
return $2i+1$

79

Binary heap - array



1 2 3 4 5 6 7 8 9 10

Parent of A[8]?

$\lfloor 8/2 \rfloor = 4$

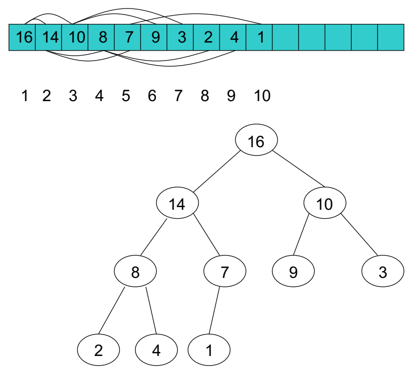
PARENT(i)
return $\lfloor i/2 \rfloor$

LEFT(i)
return $2i$

RIGHT(i)
return $2i+1$

80

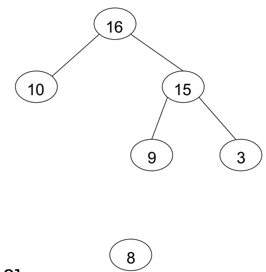
Binary heap - array



81

Identify the valid heaps

[15, 12, 3, 11, 10, 2, 1, 7, 8]



[20, 18, 10, 17, 16, 15, 9, 14, 13]

82

Heaps

Procedure	Binary heap (worst-case)	Binomial heap (worst-case)	Fibonacci heap (amortized)
BUILD-HEAP	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
INSERT	$\Theta(\log n)$	$O(\log n)$	$\Theta(1)$
MAXIMUM	$\Theta(1)$	$O(\log n)$	$\Theta(1)$
EXTRAC-MAX	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$
UNION	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$
INCREASE-ELEMENT	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
DELETE	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$

(adapted from Figure 19.1, pg. 456 [1])

Should you always use a Fibonacci heap?

83

Heaps

Procedure	Binary heap (worst-case)	Binomial heap (worst-case)	Fibonacci heap (amortized)
BUILD-HEAP	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
INSERT	$\Theta(\log n)$	$O(\log n)$	$\Theta(1)$
MAXIMUM	$\Theta(1)$	$O(\log n)$	$\Theta(1)$
EXTRAC-MAX	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$
UNION	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$
INCREASE-ELEMENT	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
DELETE	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$

(adapted from Figure 19.1, pg. 456 [1])

- Extract-Max and Delete are O(n) worst case
- Constants can be large on some of the operations
- Complicated to implement

84

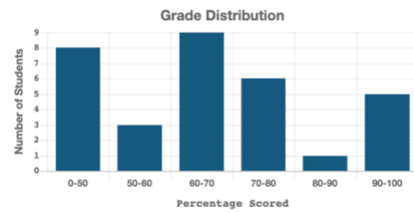
Heaps

Procedure	Binary heap (worst-case)	Binomial heap (worst-case)	Fibonacci heap (amortized)
BUILD-HEAP	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
INSERT	$\Theta(\log n)$	$O(\log n)$	$\Theta(1)$
MAXIMUM	$\Theta(1)$	$O(\log n)$	$\Theta(1)$
EXTRAC-MAX	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$
UNION	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$
INCREASE-ELEMENT	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
DELETE	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$

(adapted from Figure 19.1, pg. 456 [1])

Can we do better?

85



Average (mean) grade	16.63
Median grade	16.25
Standard deviation	4.26

86