Map, Jumble, DLLMap, and SkipMap

Victor Milenkovic

Department of Computer Science
University of Miami

CSC220 Programming II – Spring 2019

Map

Map

- A *Map* is what Java calls a PhoneDirectory.

Map

- A *Map* is what Java calls a PhoneDirectory.
- Let's have a look at the Map interface.

Map

- A *Map* is what Java calls a PhoneDirectory.
- Let's have a look at the Map interface.
  - **V put(K key, V value)** is like **addOrChangeEntry**.

Map

- A *Map* is what Java calls a PhoneDirectory.
- Let's have a look at the Map interface.
  - **V put(K key, V value)** is like **addOrChangeEntry**.
  - **V get(K key)** is like **lookupEntry**.

Map

- A *Map* is what Java calls a PhoneDirectory.
- Let's have a look at the Map interface.
  - **V put(K key, V value)** is like **addOrChangeEntry**.
  - **V get(K key)** is like **lookupEntry**.
  - **V remove(Object Key)** is like **removeEntry**.

Map

- A *Map* is what Java calls a PhoneDirectory.
- Let's have a look at the Map interface.
  - **V put(K key, V value)** is like **addOrChangeEntry**.
  - **V get(K key)** is like **lookupEntry**.
  - **V remove(Object Key)** is like **removeEntry**.
- We have already implemented a PhoneDirectory using an unsorted or sorted array or doubly linked list.

Map

- A *Map* is what Java calls a PhoneDirectory.
- Let's have a look at the Map interface.
  - **V put(K key, V value)** is like **addOrChangeEntry**.
  - **V get(K key)** is like **lookupEntry**.
  - **V remove(Object Key)** is like **removeEntry**.
- We have already implemented a PhoneDirectory using an unsorted or sorted array or doubly linked list.
- You are currently implementing Map as a BST.

Map

- A *Map* is what Java calls a PhoneDirectory.
- Let's have a look at the Map interface.
  - **V put(K key, V value)** is like **addOrChangeEntry**.
  - **V get(K key)** is like **lookupEntry**.
  - **V remove(Object Key)** is like **removeEntry**.
- We have already implemented a PhoneDirectory using an unsorted or sorted array or doubly linked list.
- You are currently implementing Map as a BST.
  - It still takes $O(n)$ to get or put in the worst case.

Map

- A *Map* is what Java calls a PhoneDirectory.
- Let's have a look at the Map interface.
  - **V put(K key, V value)** is like **addOrChangeEntry**.
  - **V get(K key)** is like **lookupEntry**.
  - **V remove(Object Key)** is like **removeEntry**.
- We have already implemented a PhoneDirectory using an unsorted or sorted array or doubly linked list.
- You are currently implementing Map as a BST.
  - It still takes $O(n)$ to get or put in the worst case.
  - If the items are not inserted in random order.

Map

- A *Map* is what Java calls a PhoneDirectory.
- Let's have a look at the Map interface.
  - **V put(K key, V value)** is like **addOrChangeEntry**.
  - **V get(K key)** is like **lookupEntry**.
  - **V remove(Object Key)** is like **removeEntry**.
- We have already implemented a PhoneDirectory using an unsorted or sorted array or doubly linked list.
- You are currently implementing Map as a BST.
  - It still takes $O(n)$ to get or put in the worst case.
  - If the items are not inserted in random order.
  - We need a way to keep the tree *balanced*.

This week's application

This week's application

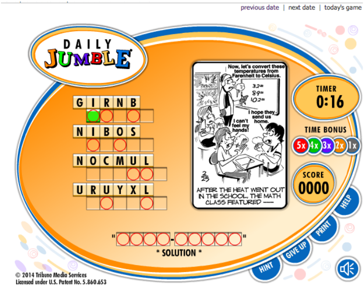
- We need a nice application for our Map.

This week's application

- We need a nice application for our Map.
  - Yet another game!

This week's application

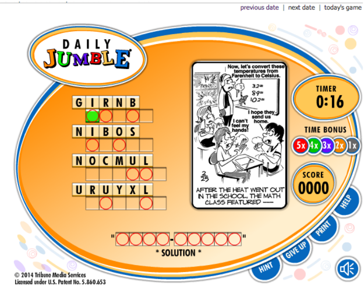
- We need a nice application for our Map.
  - Yet another game!



  - Daily Jumble

This week's application

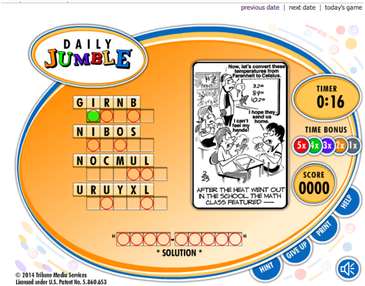
- We need a nice application for our Map.
  - Yet another game!



  - Daily Jumble
- Need to unscramble words.

This week's application

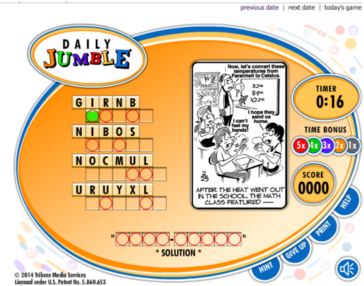
- We need a nice application for our Map.
  - Yet another game!



  - Daily Jumble
- Need to unscramble words.
  - Puzzle has "rtpocmue"?

This week's application

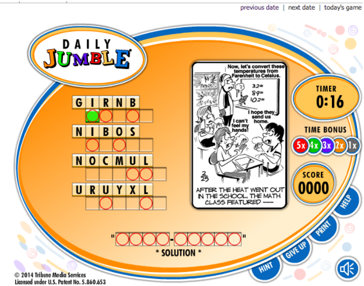
- We need a nice application for our Map.
  - Yet another game!



  - Daily Jumble
- Need to unscramble words.
  - Puzzle has "rtpocmue"?
  - Unscrambled is "computer".

This week's application

- We need a nice application for our Map.
  - Yet another game!



  - Daily Jumble
- Need to unscramble words.
  - Puzzle has “rtpocmue”?
  - Unscrambled is “computer”.
  - How can a Map help us to do that?

Slow Way

Slow Way

- We have a dictionary file.

Slow Way

- We have a dictionary file.
  - Read it in.

Slow Way

- We have a dictionary file.
  - Read it in.
  - Try every possible ordering of “rtpocmue”.

Slow Way

- We have a dictionary file.
  - Read it in.
  - Try every possible ordering of “rtpocmue”.
  - Look up each one in the dictionary.

Slow Way

- We have a dictionary file.
  - Read it in.
  - Try every possible ordering of “rtpocmue”.
  - Look up each one in the dictionary.
- What is the running time?

Slow Way

- We have a dictionary file.
  - Read it in.
  - Try every possible ordering of “rtpocmue”.
  - Look up each one in the dictionary.
- What is the running time?
  - Lookup might be $O(\log n)$ time, good.

Slow Way

- We have a dictionary file.
  - Read it in.
  - Try every possible ordering of “rtpocmue”.
  - Look up each one in the dictionary.
- What is the running time?
  - Lookup might be $O(\log n)$ time, good.
  - But the number of orderings is 8! = 40,320, bad!.

Using a Map

Using a Map

- Let's use a Map.

Using a Map

- Let's use a Map.
  - The value will be "computer".

Using a Map

- Let's use a Map.
  - The value will be "computer".
  - What will be the key?

Using a Map

- Let's use a Map.
  - The value will be "computer".
  - What will be the key?
  - How about the letters in alphabetical order?

Using a Map

- Let's use a Map.
  - The value will be "computer".
  - What will be the key?
  - How about the letters in alphabetical order?
  - That is "cemoprtu".

Using a Map

- Let's use a Map.
  - The value will be "computer".
  - What will be the key?
  - How about the letters in alphabetical order?
  - That is "cemoprtu".
- To get ready:

Using a Map

- Let's use a Map.
  - The value will be "computer".
  - What will be the key?
  - How about the letters in alphabetical order?
  - That is "cemoprtu".
- To get ready:
  - Read each word from the dictionary file,

Using a Map

- Let’s use a Map.
  - The value will be "computer".
  - What will be the key?
  - How about the letters in alphabetical order?
  - That is "cemoprtu".
- To get ready:
  - Read each word from the dictionary file,
  - Put it into the Map.

Using a Map

- Let's use a Map.
  - The value will be "computer".
  - What will be the key?
  - How about the letters in alphabetical order?
  - That is "cemoprtu".
- To get ready:
  - Read each word from the dictionary file,
  - Put it into the Map.
  - The key will be the letters of the word in alphabetical order.

Using a Map

- Let's use a Map.
  - The value will be "computer".
  - What will be the key?
  - How about the letters in alphabetical order?
  - That is "cemoprtu".
- To get ready:
  - Read each word from the dictionary file,
  - Put it into the Map.
  - The key will be the letters of the word in alphabetical order.
  - The value will be the word.

Using a Map

- Let's use a Map.
  - The value will be "computer".
  - What will be the key?
  - How about the letters in alphabetical order?
  - That is "cemoprtu".
- To get ready:
  - Read each word from the dictionary file,
  - Put it into the Map.
  - The key will be the letters of the word in alphabetical order.
  - The value will be the word.
- To solve a scramble "rtpmceuo":

Using a Map

- Let's use a Map.
  - The value will be "computer".
  - What will be the key?
  - How about the letters in alphabetical order?
  - That is "cemoprtu".
- To get ready:
  - Read each word from the dictionary file,
  - Put it into the Map.
  - The key will be the letters of the word in alphabetical order.
  - The value will be the word.
- To solve a scramble "rtpmceuo":
  - Alphabetize it to "cemoprtu".

Using a Map

- Let's use a Map.
  - The value will be "computer".
  - What will be the key?
  - How about the letters in alphabetical order?
  - That is "cemoprtu".
- To get ready:
  - Read each word from the dictionary file,
  - Put it into the Map.
  - The key will be the letters of the word in alphabetical order.
  - The value will be the word.
- To solve a scramble "rtpmceuo":
  - Alphabetize it to "cemoprtu".
  - Look it up in the map: "computer".

Using a Map

- Let's use a Map.
  - The value will be "computer".
  - What will be the key?
  - How about the letters in alphabetical order?
  - That is "cemoprtu".
- To get ready:
  - Read each word from the dictionary file,
  - Put it into the Map.
  - The key will be the letters of the word in alphabetical order.
  - The value will be the word.
- To solve a scramble "rtpmceuo":
  - Alphabetize it to "cemoprtu".
  - Look it up in the map: "computer".
- Does anyone see a problem?

Using a Map

- Let's use a Map.
  - The value will be "computer".
  - What will be the key?
  - How about the letters in alphabetical order?
  - That is "cemoprtu".
- To get ready:
  - Read each word from the dictionary file,
  - Put it into the Map.
  - The key will be the letters of the word in alphabetical order.
  - The value will be the word.
- To solve a scramble "rtpmceuo":
  - Alphabetize it to "cemoprtu".
  - Look it up in the map: "computer".
- Does anyone see a problem?
  - The words "dare", "dear", and "read"

Using a Map

- Let’s use a Map.
  - The value will be "computer".
  - What will be the key?
  - How about the letters in alphabetical order?
  - That is "cemoprtu".
- To get ready:
  - Read each word from the dictionary file,
  - Put it into the Map.
  - The key will be the letters of the word in alphabetical order.
  - The value will be the word.
- To solve a scramble “rtpmceuo”:
  - Alphabetize it to “cemoprtu”.
  - Look it up in the map: “computer”.
- Does anyone see a problem?
  - The words “dare”, “dear”, and “read”
  - will all be stored under the key “ader”.

Using a Map

- Let's use a Map.
  - The value will be "computer".
  - What will be the key?
  - How about the letters in alphabetical order?
  - That is "cemoprtu".
- To get ready:
  - Read each word from the dictionary file,
  - Put it into the Map.
  - The key will be the letters of the word in alphabetical order.
  - The value will be the word.
- To solve a scramble “rtpmceuo”:
  - Alphabetize it to “cemoprtu”.
  - Look it up in the map: “computer”.
- Does anyone see a problem?
  - The words “dare”, “dear”, and “read”
  - will all be stored under the key “ader”.
  - So the value will be “read” because it is last.

Using a Map

- Let's use a Map.
  - The value will be "computer".
  - What will be the key?
  - How about the letters in alphabetical order?
  - That is "cemoprtu".
- To get ready:
  - Read each word from the dictionary file,
  - Put it into the Map.
  - The key will be the letters of the word in alphabetical order.
  - The value will be the word.
- To solve a scramble "rtpmceuo":
  - Alphabetize it to "cemoprtu".
  - Look it up in the map: "computer".
- Does anyone see a problem?
  - The words "dare", "dear", and "read"
  - will all be stored under the key "ader".
  - So the value will be "read" because it is last.
  - Solution is to use **List<String>** as the value type.

Using a Map

- Let's use a Map.
  - The value will be "computer".
  - What will be the key?
  - How about the letters in alphabetical order?
  - That is "cemoprtu".
- To get ready:
  - Read each word from the dictionary file,
  - Put it into the Map.
  - The key will be the letters of the word in alphabetical order.
  - The value will be the word.
- To solve a scramble "rtpmceuo":
  - Alphabetize it to "cemoprtu".
  - Look it up in the map: "computer".
- Does anyone see a problem?
  - The words "dare", "dear", and "read"
  - will all be stored under the key "ader".
  - So the value will be "read" because it is last.
  - Solution is to use **List<String>** as the value type.
  - But we won't do that this time.

Implementation using our Map implementations

Implementation using our Map implementations

- I will run the Jumble solver for you using the our different implementations of PhoneDirectory or Map.

Implementation using our Map implementations

- I will run the Jumble solver for you using the our different implementations of PhoneDirectory or Map.
- PDMap is a way to convert a PhoneDirectory to a Map.

Implementation using our Map implementations

- I will run the Jumble solver for you using the our different implementations of PhoneDirectory or Map.
- PDMap is a way to convert a PhoneDirectory to a Map.
  - The lookup might be as fast as $\mathrm{O}(n)$ (for SortedPD) but that's not the problem.

Implementation using our Map implementations

- I will run the Jumble solver for you using the our different implementations of PhoneDirectory or Map.
- PDMap is a way to convert a PhoneDirectory to a Map.
  - The lookup might be as fast as $O(n)$ (for SortedPD) but that's not the problem.
  - The problem is adding all the words in the dictionary.

Implementation using our Map implementations

- I will run the Jumble solver for you using the our different implementations of PhoneDirectory or Map.
- PDMap is a way to convert a PhoneDirectory to a Map.
  - The lookup might be as fast as $O(n)$ (for SortedPD) but that's not the problem.
  - The problem is adding all the words in the dictionary.
  - We never got add faster than $O(n)$

Implementation using our Map implementations

- I will run the Jumble solver for you using the our different implementations of PhoneDirectory or Map.
- PDMap is a way to convert a PhoneDirectory to a Map.
  - The lookup might be as fast as $O(n)$ (for SortedPD) but that's not the problem.
  - The problem is adding all the words in the dictionary.
  - We never got add faster than $O(n)$
  - If we add n words, that's $O(n^2)$,

Implementation using our Map implementations

- I will run the Jumble solver for you using the our different implementations of PhoneDirectory or Map.
- PDMap is a way to convert a PhoneDirectory to a Map.
  - The lookup might be as fast as $O(n)$ (for SortedPD) but that's not the problem.
  - The problem is adding all the words in the dictionary.
  - We never got add faster than $O(n)$
  - If we add n words, that's $O(n^2)$,
  - which is pretty slow.

Implementation using our Map implementations

- I will run the Jumble solver for you using the our different implementations of PhoneDirectory or Map.
- PDMap is a way to convert a PhoneDirectory to a Map.
  - The lookup might be as fast as $O(n)$ (for SortedPD) but that's not the problem.
  - The problem is adding all the words in the dictionary.
  - We never got add faster than $O(n)$
  - If we add n words, that's $O(n^2)$,
  - which is pretty slow.
- It's even slower for a bigger dictionary, as I will show you.

Implementation using our Map implementations

- I will run the Jumble solver for you using the our different implementations of PhoneDirectory or Map.
- PDMap is a way to convert a PhoneDirectory to a Map.
  - The lookup might be as fast as $O(n)$ (for SortedPD) but that's not the problem.
  - The problem is adding all the words in the dictionary.
  - We never got add faster than $O(n)$
  - If we add n words, that's $O(n^2)$,
  - which is pretty slow.
- It's even slower for a bigger dictionary, as I will show you.
  - To get to entry i in the list takes i steps.

Implementation using our Map implementations

- I will run the Jumble solver for you using the our different implementations of PhoneDirectory or Map.
- PDMap is a way to convert a PhoneDirectory to a Map.
  - The lookup might be as fast as $O(n)$ (for SortedPD) but that's not the problem.
  - The problem is adding all the words in the dictionary.
  - We never got add faster than $O(n)$
  - If we add n words, that's $O(n^2)$,
  - which is pretty slow.
- It's even slower for a bigger dictionary, as I will show you.
  - To get to entry i in the list takes i steps.
  - An array can do it in one step (**theArray[i]**)

Implementation using our Map implementations

- I will run the Jumble solver for you using the our different implementations of PhoneDirectory or Map.
- PDMap is a way to convert a PhoneDirectory to a Map.
  - The lookup might be as fast as $O(n)$ (for SortedPD) but that's not the problem.
  - The problem is adding all the words in the dictionary.
  - We never got add faster than $O(n)$
  - If we add n words, that's $O(n^2)$,
  - which is pretty slow.
- It's even slower for a bigger dictionary, as I will show you.
  - To get to entry *i* in the list takes *i* steps.
  - An array can do it in one step (**theArray[i]**)
  - but then adding at the head will cost $O(n)$

Implementation using our Map implementations

- I will run the Jumble solver for you using the our different implementations of PhoneDirectory or Map.
- PDMap is a way to convert a PhoneDirectory to a Map.
  - The lookup might be as fast as $O(n)$ (for SortedPD) but that's not the problem.
  - The problem is adding all the words in the dictionary.
  - We never got add faster than $O(n)$
  - If we add n words, that's $O(n^2)$,
  - which is pretty slow.
- It's even slower for a bigger dictionary, as I will show you.
  - To get to entry *i* in the list takes *i* steps.
  - An array can do it in one step (**theArray[i]**)
  - but then adding at the head will cost $O(n)$
  - The linked list lets us add quickly once we get there,

Implementation using our Map implementations

- I will run the Jumble solver for you using the our different implementations of PhoneDirectory or Map.
- PDMap is a way to convert a PhoneDirectory to a Map.
  - The lookup might be as fast as $O(n)$ (for SortedPD) but that's not the problem.
  - The problem is adding all the words in the dictionary.
  - We never got add faster than $O(n)$
  - If we add n words, that's $O(n^2)$,
  - which is pretty slow.
- It's even slower for a bigger dictionary, as I will show you.
  - To get to entry *i* in the list takes *i* steps.
  - An array can do it in one step (**theArray[i]**)
  - but then adding at the head will cost $O(n)$
  - The linked list lets us add quickly once we get there,
  - but it takes a while to get there.

Implementation using our Map implementations

- I will run the Jumble solver for you using the our different implementations of PhoneDirectory or Map.
- PDMap is a way to convert a PhoneDirectory to a Map.
  - The lookup might be as fast as $O(n)$ (for SortedPD) but that's not the problem.
  - The problem is adding all the words in the dictionary.
  - We never got add faster than $O(n)$
  - If we add n words, that's $O(n^2)$,
  - which is pretty slow.
- It's even slower for a bigger dictionary, as I will show you.
  - To get to entry *i* in the list takes *i* steps.
  - An array can do it in one step (**theArray[i]**)
  - but then adding at the head will cost $O(n)$
  - The linked list lets us add quickly once we get there,
  - but it takes a while to get there.
  - BST is faster,

Implementation using our Map implementations

- I will run the Jumble solver for you using the our different implementations of PhoneDirectory or Map.
- PDMap is a way to convert a PhoneDirectory to a Map.
  - The lookup might be as fast as $O(n)$ (for SortedPD) but that's not the problem.
  - The problem is adding all the words in the dictionary.
  - We never got add faster than $O(n)$
  - If we add n words, that's $O(n^2)$,
  - which is pretty slow.
- It's even slower for a bigger dictionary, as I will show you.
  - To get to entry *i* in the list takes *i* steps.
  - An array can do it in one step (**theArray[i]**)
  - but then adding at the head will cost $O(n)$
  - The linked list lets us add quickly once we get there,
  - but it takes a while to get there.
  - BST is faster,
  - but still much slower than a balanced tree (TreeMap).

Implementation using our Map implementations

- I will run the Jumble solver for you using the our different implementations of PhoneDirectory or Map.
- PDMap is a way to convert a PhoneDirectory to a Map.
  - The lookup might be as fast as $O(n)$ (for SortedPD) but that's not the problem.
  - The problem is adding all the words in the dictionary.
  - We never got add faster than $O(n)$
  - If we add n words, that's $O(n^2)$,
  - which is pretty slow.
- It's even slower for a bigger dictionary, as I will show you.
  - To get to entry i in the list takes i steps.
  - An array can do it in one step (**theArray[i]**)
  - but then adding at the head will cost $O(n)$
  - The linked list lets us add quickly once we get there,
  - but it takes a while to get there.
  - BST is faster,
  - but still much slower than a balanced tree (TreeMap).
- We need to learn how to balance the tree.

Cost of reallocate

Cost of reallocate

- Let's think about ArrayBasedPD first!

Cost of reallocate

- Let's think about ArrayBasedPD first!
- What is the cost of *add* (not *addOrChangeEntry*)?

Cost of reallocate

- Let’s think about ArrayBasedPD first!
- What is the cost of *add* (not *addOrChangeEntry*)?
- addOrChangeEntry is $\mathrm{O}(n)$ because?

Cost of reallocate

- Let's think about ArrayBasedPD first!
- What is the cost of *add* (not *addOrChangeEntry*)?
- addOrChangeEntry is $\mathrm{O}(n)$ because?
- add is $\mathrm{O}(n)$ when we have to reallocate,

Cost of reallocate

- Let's think about ArrayBasedPD first!
- What is the cost of *add* (not *addOrChangeEntry*)?
- addOrChangeEntry is $\mathrm{O}(n)$ because?
- add is $\mathrm{O}(n)$ when we have to reallocate,
- but we don't have to reallocate very often.

Cost of reallocate

- Let's think about ArrayBasedPD first!
- What is the cost of *add* (not *addOrChangeEntry*)?
- addOrChangeEntry is $O(n)$ because?
- add is $O(n)$ when we have to reallocate,
- but we don't have to reallocate very often.
- Call add n times (starting with an empty array) costs $O(n)$, not $O(n^2)$.

Cost of Add

Cost of Add

- Let's start with an array of size 1 and count the number of array assignments.

Cost of Add

- Let's start with an array of size 1 and count the number of array assignments.
- add(1)

Cost of Add

- Let's start with an array of size 1 and count the number of array assignments.
- add(1)
- 1 : n=1 a=1

Cost of Add

- Let's start with an array of size 1 and count the number of array assignments.
- add(1)
- 1 : n=1 a=1
- add(2) (need to reallocate)

Cost of Add

- Let's start with an array of size 1 and count the number of array assignments.
- add(1)
- 1 : n=1 a=1
- add(2) (need to reallocate)
- 1 2 : n=2 a=3

Cost of Add

- Let's start with an array of size 1 and count the number of array assignments.
- add(1)
- 1 : n=1 a=1
- add(2) (need to reallocate)
- 1 2 : n=2 a=3
- add(3) (need to reallocate)

Cost of Add

- Let's start with an array of size 1 and count the number of array assignments.
- add(1)
- 1 : n=1 a=1
- add(2) (need to reallocate)
- 1 2 : n=2 a=3
- add(3) (need to reallocate)
- 1 2 3 _ : n=3 a=6

Cost of Add

- Let's start with an array of size 1 and count the number of array assignments.
- add(1)
- 1 : n=1 a=1
- add(2) (need to reallocate)
- 1 2 : n=2 a=3
- add(3) (need to reallocate)
- 1 2 3 _ : n=3 a=6
- add(4)

Cost of Add

- Let's start with an array of size 1 and count the number of array assignments.
- add(1)
- 1 : n=1 a=1
- add(2) (need to reallocate)
- 1 2 : n=2 a=3
- add(3) (need to reallocate)
- 1 2 3 _ : n=3 a=6
- add(4)
- 1 2 3 4 : n=4 a=7

Cost of Add

- Let's start with an array of size 1 and count the number of array assignments.
- add(1)
- 1 : n=1 a=1
- add(2) (need to reallocate)
- 1 2 : n=2 a=3
- add(3) (need to reallocate)
- 1 2 3 _ : n=3 a=6
- add(4)
- 1 2 3 4 : n=4 a=7
- add(5) (need to reallocate)

Cost of Add

- Let's start with an array of size 1 and count the number of array assignments.
- add(1)
- 1 : n=1 a=1
- add(2) (need to reallocate)
- 1 2 : n=2 a=3
- add(3) (need to reallocate)
- 1 2 3 _ : n=3 a=6
- add(4)
- 1 2 3 4 : n=4 a=7
- add(5) (need to reallocate)
- 1 2 3 4 5 _ _ _ : n=4 a=12

Cost of Add

- Let's start with an array of size 1 and count the number of array assignments.
- add(1)
- 1 : n=1 a=1
- add(2) (need to reallocate)
- 1 2 : n=2 a=3
- add(3) (need to reallocate)
- 1 2 3 _ : n=3 a=6
- add(4)
- 1 2 3 4 : n=4 a=7
- add(5) (need to reallocate)
- 1 2 3 4 5 _ _ _ : n=4 a=12
- add(6), add(7), add(8)

Cost of Add

- Let's start with an array of size 1 and count the number of array assignments.
- add(1)
- 1 : n=1 a=1
- add(2) (need to reallocate)
- 1 2 : n=2 a=3
- add(3) (need to reallocate)
- 1 2 3 _ : n=3 a=6
- add(4)
- 1 2 3 4 : n=4 a=7
- add(5) (need to reallocate)
- 1 2 3 4 5 _ _ _ : n=4 a=12
- add(6), add(7), add(8)
- 1 2 3 4 5 6 7 8 : n=8 a=15

Cost of Add

- Let's start with an array of size 1 and count the number of array assignments.
- add(1)
- 1 : n=1 a=1
- add(2) (need to reallocate)
- 1 2 : n=2 a=3
- add(3) (need to reallocate)
- 1 2 3 _ : n=3 a=6
- add(4)
- 1 2 3 4 : n=4 a=7
- add(5) (need to reallocate)
- 1 2 3 4 5 _ _ _ : n=4 a=12
- add(6), add(7), add(8)
- 1 2 3 4 5 6 7 8 : n=8 a=15
- add(9) (need to reallocate)

Cost of Add

- Let's start with an array of size 1 and count the number of array assignments.
- add(1)
- 1 : n=1 a=1
- add(2) (need to reallocate)
- 1 2 : n=2 a=3
- add(3) (need to reallocate)
- 1 2 3 _ : n=3 a=6
- add(4)
- 1 2 3 4 : n=4 a=7
- add(5) (need to reallocate)
- 1 2 3 4 5 _ _ _ : n=4 a=12
- add(6), add(7), add(8)
- 1 2 3 4 5 6 7 8 : n=8 a=15
- add(9) (need to reallocate)
- 1 2 3 4 5 6 7 8 9 _ _ _ _ _ _ _: n=9 a=24

Cost of Add

- Let's start with an array of size 1 and count the number of array assignments.
- add(1)
- 1 : n=1 a=1
- add(2) (need to reallocate)
- 1 2 : n=2 a=3
- add(3) (need to reallocate)
- 1 2 3 _ : n=3 a=6
- add(4)
- 1 2 3 4 : n=4 a=7
- add(5) (need to reallocate)
- 1 2 3 4 5 _ _ _ : n=4 a=12
- add(6), add(7), add(8)
- 1 2 3 4 5 6 7 8 : n=8 a=15
- add(9) (need to reallocate)
- 1 2 3 4 5 6 7 8 9 _ _ _ _ _ _ _: n=9 a=24
- add(10) - add(16)

Cost of Add

- Let's start with an array of size 1 and count the number of array assignments.
- add(1)
- 1 : n=1 a=1
- add(2) (need to reallocate)
- 1 2 : n=2 a=3
- add(3) (need to reallocate)
- 1 2 3 _ : n=3 a=6
- add(4)
- 1 2 3 4 : n=4 a=7
- add(5) (need to reallocate)
- 1 2 3 4 5 _ _ _ : n=4 a=12
- add(6), add(7), add(8)
- 1 2 3 4 5 6 7 8 : n=8 a=15
- add(9) (need to reallocate)
- 1 2 3 4 5 6 7 8 9 _ _ _ _ _ _ _: n=9 a=24
- add(10) - add(16)
- 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 : n=16 a=31

Cost of Add

- Let's start with an array of size 1 and count the number of array assignments.
- add(1)
- 1 : n=1 a=1
- add(2) (need to reallocate)
- 1 2 : n=2 a=3
- add(3) (need to reallocate)
- 1 2 3 _ : n=3 a=6
- add(4)
- 1 2 3 4 : n=4 a=7
- add(5) (need to reallocate)
- 1 2 3 4 5 _ _ _ : n=4 a=12
- add(6), add(7), add(8)
- 1 2 3 4 5 6 7 8 : n=8 a=15
- add(9) (need to reallocate)
- 1 2 3 4 5 6 7 8 9 _ _ _ _ _ _ _ _: n=9 a=24
- add(10) - add(16)
- 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 : n=16 a=31
- add(17)

Cost of Add

- Let's start with an array of size 1 and count the number of array assignments.
- add(1)
- 1 : n=1 a=1
- add(2) (need to reallocate)
- 1 2 : n=2 a=3
- add(3) (need to reallocate)
- 1 2 3 _ : n=3 a=6
- add(4)
- 1 2 3 4 : n=4 a=7
- add(5) (need to reallocate)
- 1 2 3 4 5 _ _ _ : n=4 a=12
- add(6), add(7), add(8)
- 1 2 3 4 5 6 7 8 : n=8 a=15
- add(9) (need to reallocate)
- 1 2 3 4 5 6 7 8 9 _ _ _ _ _ _ _: n=9 a=24
- add(10) - add(16)
- 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 : n=16 a=31
- add(17)
- 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ : n=16 a=48

Cost of Add

Cost of Add

- The total number of assignments is never more than $3n$ (check!). Why?

Cost of Add

- The total number of assignments is never more than $3n$ (check!). Why?
- Adding an item the *right half* of the array requires *one* assignment now

Cost of Add

- The total number of assignments is never more than $3n$ (check!). Why?
- Adding an item the *right half* of the array requires *one* assignment now
- plus *two more* in the future to move it and one item in the left half.

Cost of Add

- The total number of assignments is never more than $3n$ (check!). Why?
- Adding an item the *right half* of the array requires *one* assignment now
- plus *two more* in the future to move it and one item in the left half.
- After that, it is always in the *left half*, so a newer item will pay to move it from now on.

Cost of Add

- The total number of assignments is never more than $3n$ (check!). Why?
- Adding an item the *right half* of the array requires *one* assignment now
- plus *two more* in the future to move it and one item in the left half.
- After that, it is always in the *left half*, so a newer item will pay to move it from now on.
- It's actually very much like Social Security!

Cost of Add

- The total number of assignments is never more than $3n$ (check!). Why?
- Adding an item the *right half* of the array requires *one* assignment now
- plus *two more* in the future to move it and one item in the left half.
- After that, it is always in the *left half*, so a newer item will pay to move it from now on.
- It's actually very much like Social Security!
- Question: will this still work if we only increase the array size by 50% when we reallocate?

Cost of Add

- The total number of assignments is never more than $3n$ (check!). Why?
- Adding an item the *right half* of the array requires *one* assignment now
- plus *two more* in the future to move it and one item in the left half.
- After that, it is always in the *left half*, so a newer item will pay to move it from now on.
- It's actually very much like Social Security!
- Question: will this still work if we only increase the array size by 50% when we reallocate?
- Yes. Each item added to the *right third* will pay 1 now and 3 in the future to move it and 2 items in the *left two-thirds*.

Cost of Add

- The total number of assignments is never more than $3n$ (check!). Why?
- Adding an item the *right half* of the array requires *one* assignment now
- plus *two more* in the future to move it and one item in the left half.
- After that, it is always in the *left half*, so a newer item will pay to move it from now on.
- It's actually very much like Social Security!
- Question: will this still work if we only increase the array size by 50% when we reallocate?
- Yes. Each item added to the *right third* will pay 1 now and 3 in the future to move it and 2 items in the *left two-thirds*.
- So the total never exceeds $4n$.

Cost of Add

- The total number of assignments is never more than $3n$ (check!). Why?
- Adding an item the *right half* of the array requires *one* assignment now
- plus *two more* in the future to move it and one item in the left half.
- After that, it is always in the *left half*, so a newer item will pay to move it from now on.
- It's actually very much like Social Security!
- Question: will this still work if we only increase the array size by 50% when we reallocate?
- Yes. Each item added to the *right third* will pay 1 now and 3 in the future to move it and 2 items in the *left two-thirds*.
- So the total never exceeds $4n$.
- Which is still $\mathrm{O}(n)$.

Cost of Add

- The total number of assignments is never more than $3n$ (check!). Why?
- Adding an item the *right half* of the array requires *one* assignment now
- plus *two more* in the future to move it and one item in the left half.
- After that, it is always in the *left half*, so a newer item will pay to move it from now on.
- It's actually very much like Social Security!
- Question: will this still work if we only increase the array size by 50% when we reallocate?
- Yes. Each item added to the *right third* will pay 1 now and 3 in the future to move it and 2 items in the *left two-thirds*.
- So the total never exceeds $4n$.
- Which is still $\mathrm{O}(n)$.
- This is called *amortized analysis*.

Tree Balancing

Tree Balancing

- What does this have to do with balancing trees?

Tree Balancing

- What does this have to do with balancing trees?
- Suppose we start with a balanced tree with 100 nodes in the left and right half.

Tree Balancing

- What does this have to do with balancing trees?
- Suppose we start with a balanced tree with 100 nodes in the left and right half.
- Then we add nodes using the add method you implemented in lab before break.

Tree Balancing

- What does this have to do with balancing trees?
- Suppose we start with a balanced tree with 100 nodes in the left and right half.
- Then we add nodes using the add method you implemented in lab before break.
- When the right half of a (sub)tree is more than twice as big as the left half, we rebalance that tree.

Tree Balancing

- What does this have to do with balancing trees?
- Suppose we start with a balanced tree with 100 nodes in the left and right half.
- Then we add nodes using the add method you implemented in lab before break.
- When the right half of a (sub)tree is more than twice as big as the left half, we rebalance that tree.
- For this to happen, we have to add at least 100 more nodes to the tree.

Tree Balancing

- What does this have to do with balancing trees?
- Suppose we start with a balanced tree with 100 nodes in the left and right half.
- Then we add nodes using the add method you implemented in lab before break.
- When the right half of a (sub)tree is more than twice as big as the left half, we rebalance that tree.
- For this to happen, we have to add at least 100 more nodes to the tree.
- So when we rebalance, the tree has to become *50% bigger* before we have to rebalance.

Tree Balancing

- What does this have to do with balancing trees?
- Suppose we start with a balanced tree with 100 nodes in the left and right half.
- Then we add nodes using the add method you implemented in lab before break.
- When the right half of a (sub)tree is more than twice as big as the left half, we rebalance that tree.
- For this to happen, we have to add at least 100 more nodes to the tree.
- So when we rebalance, the tree has to become *50% bigger* before we have to rebalance.
- Does that sound familiar?

Tree Balancing

- What does this have to do with balancing trees?
- Suppose we start with a balanced tree with 100 nodes in the left and right half.
- Then we add nodes using the add method you implemented in lab before break.
- When the right half of a (sub)tree is more than twice as big as the left half, we rebalance that tree.
- For this to happen, we have to add at least 100 more nodes to the tree.
- So when we rebalance, the tree has to become *50% bigger* before we have to rebalance.
- Does that sound familiar?
- That tells us the total cost of rebalancing never exceeds $O(n)$ per subtree to which we add each item.

Tree Balancing

- What does this have to do with balancing trees?
- Suppose we start with a balanced tree with 100 nodes in the left and right half.
- Then we add nodes using the add method you implemented in lab before break.
- When the right half of a (sub)tree is more than twice as big as the left half, we rebalance that tree.
- For this to happen, we have to add at least 100 more nodes to the tree.
- So when we rebalance, the tree has to become *50% bigger* before we have to rebalance.
- Does that sound familiar?
- That tells us the total cost of rebalancing never exceeds $\mathrm{O}(n)$ per subtree to which we add each item.
- But each subtree is at most $2/3$ of its parent tree

Tree Balancing

- What does this have to do with balancing trees?
- Suppose we start with a balanced tree with 100 nodes in the left and right half.
- Then we add nodes using the add method you implemented in lab before break.
- When the right half of a (sub)tree is more than twice as big as the left half, we rebalance that tree.
- For this to happen, we have to add at least 100 more nodes to the tree.
- So when we rebalance, the tree has to become *50% bigger* before we have to rebalance.
- Does that sound familiar?
- That tells us the total cost of rebalancing never exceeds $\mathrm{O}(n)$ per subtree to which we add each item.
- But each subtree is at most $2/3$ of its parent tree
- because the subtree is it most twice as big as the other subtree.

Tree Balancing

- What does this have to do with balancing trees?
- Suppose we start with a balanced tree with 100 nodes in the left and right half.
- Then we add nodes using the add method you implemented in lab before break.
- When the right half of a (sub)tree is more than twice as big as the left half, we rebalance that tree.
- For this to happen, we have to add at least 100 more nodes to the tree.
- So when we rebalance, the tree has to become *50% bigger* before we have to rebalance.
- Does that sound familiar?
- That tells us the total cost of rebalancing never exceeds $\mathrm{O}(n)$ per subtree to which we add each item.
- But each subtree is at most $2/3$ of its parent tree
- because the subtree is it most twice as big as the other subtree.
- So we add each item to trees of size $n, (2/3)n, (2/3)^2 n, (2/3)^3 n, \ldots$.

Tree Balancing

- What does this have to do with balancing trees?
- Suppose we start with a balanced tree with 100 nodes in the left and right half.
- Then we add nodes using the add method you implemented in lab before break.
- When the right half of a (sub)tree is more than twice as big as the left half, we rebalance that tree.
- For this to happen, we have to add at least 100 more nodes to the tree.
- So when we rebalance, the tree has to become *50% bigger* before we have to rebalance.
- Does that sound familiar?
- That tells us the total cost of rebalancing never exceeds $\mathrm{O}(n)$ per subtree to which we add each item.
- But each subtree is at most $2/3$ of its parent tree
- because the subtree is it most twice as big as the other subtree.
- So we add each item to trees of size $n, (2/3)n, (2/3)^2 n, (2/3)^3 n, \ldots$.
- If $k > \log_{1.5} n$ then $(2/3)^k n < 1$, so not more than $\log_{1.5} n$ trees.

Tree Balancing

- What does this have to do with balancing trees?
- Suppose we start with a balanced tree with 100 nodes in the left and right half.
- Then we add nodes using the add method you implemented in lab before break.
- When the right half of a (sub)tree is more than twice as big as the left half, we rebalance that tree.
- For this to happen, we have to add at least 100 more nodes to the tree.
- So when we rebalance, the tree has to become *50% bigger* before we have to rebalance.
- Does that sound familiar?
- That tells us the total cost of rebalancing never exceeds $\mathrm{O}(n)$ per subtree to which we add each item.
- But each subtree is at most $2/3$ of its parent tree
- because the subtree is it most twice as big as the other subtree.
- So we add each item to trees of size $n, (2/3)n, (2/3)^2 n, (2/3)^3 n, \ldots$.
- If $k > \log_{1.5} n$ then $(2/3)^k n < 1$, so not more than $\log_{1.5} n$ trees.
- So we only add each item to $\mathrm{O}(\log n)$ trees.

Tree Balancing

- What does this have to do with balancing trees?
- Suppose we start with a balanced tree with 100 nodes in the left and right half.
- Then we add nodes using the add method you implemented in lab before break.
- When the right half of a (sub)tree is more than twice as big as the left half, we rebalance that tree.
- For this to happen, we have to add at least 100 more nodes to the tree.
- So when we rebalance, the tree has to become *50% bigger* before we have to rebalance.
- Does that sound familiar?
- That tells us the total cost of rebalancing never exceeds $\mathrm{O}(n)$ per subtree to which we add each item.
- But each subtree is at most $2/3$ of its parent tree
- because the subtree is it most twice as big as the other subtree.
- So we add each item to trees of size $n, (2/3)n, (2/3)^2n, (2/3)^3n, \ldots$.
- If $k > \log_{1.5} n$ then $(2/3)^k n < 1$, so not more than $\log_{1.5} n$ trees.
- So we only add each item to $\mathrm{O}(\log n)$ trees.
- So the cost of adding n items is $\mathrm{O}(n \log n)$.

Rebalancing using Linked Lists

Rebalancing using Linked Lists

- Question: How can we rebalance an entire tree without losing the correct order of the nodes?

Rebalancing using Linked Lists

- Question: How can we rebalance an entire tree without losing the correct order of the nodes?
- Answer: The nodes will *also* have next and previous fields. The class will have a root pointer *and* a head and tail pointer.

Rebalancing using Linked Lists

- Question: How can we rebalance an entire tree without losing the correct order of the nodes?
- Answer: The nodes will *also* have next and previous fields. The class will have a root pointer *and* a head and tail pointer.
- It will be a binary tree *and* a linked list. (Bwa ha ha!!)

Rebalancing using Linked Lists

- Question: How can we rebalance an entire tree without losing the correct order of the nodes?
- Answer: The nodes will *also* have next and previous fields. The class will have a root pointer *and* a head and tail pointer.
- It will be a binary tree *and* a linked list. (Bwa ha ha!!)
- So when we want to rebalance, we will just create a balanced tree from the list starting at the minimum of the tree and with the same size as the tree.

Linked List into Balanced Tree

Linked List into Balanced Tree

- rebalance(head, size) will take a list starting at head with size nodes

Linked List into Balanced Tree

- rebalance(head, size) will take a list starting at head with size nodes
- and return (the root) of a balanced tree.

Linked List into Balanced Tree

- rebalance(head, size) will take a list starting at head with size nodes
- and return (the root) of a balanced tree.
- rebalance calls rebalanceLeft(head, size - size/2) to get the root and left subtree

Linked List into Balanced Tree

- rebalance(head, size) will take a list starting at head with size nodes
- and return (the root) of a balanced tree.
- rebalance calls rebalanceLeft(head, size - size/2) to get the root and left subtree
- and then it calls rebalance(root.next, size/2) to get the right subtree.

Linked List into Balanced Tree

- rebalance(head, size) will take a list starting at head with size nodes
- and return (the root) of a balanced tree.
- rebalance calls rebalanceLeft(head, size - size/2) to get the root and left subtree
- and then it calls rebalance(root.next, size/2) to get the right subtree.
- rebalanceLeft(head, size) calls rebalanceLeft(head, size/2) to get the root of the left subtree

Linked List into Balanced Tree

- rebalance(head, size) will take a list starting at head with size nodes
- and return (the root) of a balanced tree.
- rebalance calls rebalanceLeft(head, size - size/2) to get the root and left subtree
- and then it calls rebalance(root.next, size/2) to get the right subtree.
- rebalanceLeft(head, size) calls rebalanceLeft(head, size/2) to get the root of the left subtree
- and rebalanceLeft(root.next, size - size/2) to get the root and the right subtree.

Linked List into Balanced Tree

- rebalance(head, size) will take a list starting at head with size nodes
- and return (the root) of a balanced tree.
- rebalance calls rebalanceLeft(head, size - size/2) to get the root and left subtree
- and then it calls rebalance(root.next, size/2) to get the right subtree.
- rebalanceLeft(head, size) calls rebalanceLeft(head, size/2) to get the root of the left subtree
- and rebalanceLeft(root.next, size - size/2) to get the root and the right subtree.
- Some assembly required!!