Linked Lists

Victor Milenkovic

Department of Computer Science
University of Miami

CSC220 Programming II – Spring 2022

Outline

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.

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.
- This week will will implement it using a linked list or a skip 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.
- This week will will implement it using a linked list or a skip list.
- A linked list is just as slow as an array (actually slower).

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.
- This week will will implement it using a linked list or a skip list.
- A linked list is just as slow as an array (actually slower).
- But a skip list is MUCH faster.

Arrays are slow

Arrays are slow

- Speed of SortedPD addOrChangeEntry:

Arrays are slow

- Speed of SortedPD addOrChangeEntry:
  - The find method for SortedPD is $\mathrm{O}(\log n)$.

Arrays are slow

- Speed of SortedPD addOrChangeEntry:
  - The find method for SortedPD is $\mathrm{O}(\log n)$.
  - But add must also insert a new entry at the index i returned by find.

Arrays are slow

- Speed of SortedPD addOrChangeEntry:
  - The find method for SortedPD is $\mathrm{O}(\log n)$.
  - But add must also insert a new entry at the index i returned by find.
  - Move all the entries from i to size $-$ 1.

Arrays are slow

- Speed of SortedPD addOrChangeEntry:
  - The find method for SortedPD is $O(\log n)$.
  - But add must also insert a new entry at the index i returned by find.
  - Move all the entries from i to size $- 1$.
  - That takes $O(n)$, which dominates the running time.

Arrays are slow

- Speed of SortedPD addOrChangeEntry:
  - The find method for SortedPD is $\mathrm{O}(\log n)$.
  - But add must also insert a new entry at the index i returned by find.
  - Move all the entries from i to size $-$ 1.
  - That takes $\mathrm{O}(n)$, which dominates the running time.
  - No hope of a fast addOrChange method for large n.

Linked List

Linked List

- Doubly Linked List

Linked List

- Doubly Linked List
  - A different way of storing a list.

Linked List

- Doubly Linked List
  - A different way of storing a list.
  - Allows us to add or remove an entry in O(1) time.

Linked List

- Doubly Linked List
  - A different way of storing a list.
  - Allows us to add or remove an entry in O(1) time.
- The LinkedMap.Entry class

Linked List

- Doubly Linked List
  - A different way of storing a list.
  - Allows us to add or remove an entry in O(1) time.
- The LinkedMap.Entry class
  - Has next and *previous* field

Linked List

- Doubly Linked List
  - A different way of storing a list.
  - Allows us to add or remove an entry in O(1) time.
- The LinkedMap.Entry class
  - Has next and *previous* field
  - with getValue and setValue methods.

Linked List

- Doubly Linked List
  - A different way of storing a list.
  - Allows us to add or remove an entry in O(1) time.
- The LinkedMap.Entry class
  - Has next and *previous* field
  - with getValue and setValue methods.
  - References to the next and previous entries in the list.

No array needed

No array needed

- So we don't need an array anymore.

No array needed

- So we don’t need an array anymore.
- All we need is a reference to any element of the list

No array needed

- So we don't need an array anymore.
- All we need is a reference to any element of the list
  - Call .next or .previous repeatedly.

No array needed

- So we don't need an array anymore.
- All we need is a reference to any element of the list
  - Call .next or .previous repeatedly.
  - Get to any other element.

No array needed

- So we don’t need an array anymore.
- All we need is a reference to any element of the list
  - Call .next or .previous repeatedly.
  - Get to any other element.
- For convenience, it is customary to store references to

No array needed

- So we don't need an array anymore.
- All we need is a reference to any element of the list
  - Call .next or .previous repeatedly.
  - Get to any other element.
- For convenience, it is customary to store references to
  - **first**, the first entry in the list

No array needed

- So we don't need an array anymore.
- All we need is a reference to any element of the list
  - Call .next or .previous repeatedly.
  - Get to any other element.
- For convenience, it is customary to store references to
  - **first**, the first entry in the list
  - **last**, the last entry in the list

No array needed

- So we don't need an array anymore.
- All we need is a reference to any element of the list
  - Call .next or .previous repeatedly.
  - Get to any other element.
- For convenience, it is customary to store references to
  - **first**, the first entry in the list
  - **last**, the last entry in the list
- The slides show how to use this structure to implement a phone directory.

Finding a name

Finding a name

- To find Ian, we need to look at each entry.

Finding a name

- To find Ian, we need to look at each entry.
  - Set the variable entry to the first one.

Finding a name

- To find Ian, we need to look at each entry.
  - Set the variable entry to the first one.
  - How do we do that?

Finding a name

- To find Ian, we need to look at each entry.
  - Set the variable entry to the first one.
  - How do we do that?
  - Compare the name at entry to the name we are looking for.

Finding a name

- To find Ian, we need to look at each entry.
  - Set the variable entry to the first one.
  - How do we do that?
  - Compare the name at entry to the name we are looking for.
  - How do we get the name at entry? (Bob)

Finding a name

- To find Ian, we need to look at each entry.
  - Set the variable entry to the first one.
  - How do we do that?
  - Compare the name at entry to the name we are looking for.
  - How do we get the name at entry? (Bob)
- That's not the one we want,

Finding a name

- To find Ian, we need to look at each entry.
  - Set the variable entry to the first one.
  - How do we do that?
  - Compare the name at entry to the name we are looking for.
  - How do we get the name at entry? (Bob)
- That's not the one we want,
  - so we need to move entry forward one.

Finding a name

- To find Ian, we need to look at each entry.
  - Set the variable entry to the first one.
  - How do we do that?
  - Compare the name at entry to the name we are looking for.
  - How do we get the name at entry? (Bob)
- That's not the one we want,
  - so we need to move entry forward one.
  - The only way to change the value of entry is an assignment

Finding a name

- To find Ian, we need to look at each entry.
  - Set the variable entry to the first one.
  - How do we do that?
  - Compare the name at entry to the name we are looking for.
  - How do we get the name at entry? (Bob)
- That's not the one we want,
  - so we need to move entry forward one.
  - The only way to change the value of entry is an assignment
  - entry =

Finding a name

- To find Ian, we need to look at each entry.
  - Set the variable entry to the first one.
  - How do we do that?
  - Compare the name at entry to the name we are looking for.
  - How do we get the name at entry? (Bob)
- That's not the one we want,
  - so we need to move entry forward one.
  - The only way to change the value of entry is an assignment
  - entry =
  - What do we set entry equal to?

Finding a name

- To find Ian, we need to look at each entry.
  - Set the variable entry to the first one.
  - How do we do that?
  - Compare the name at entry to the name we are looking for.
  - How do we get the name at entry? (Bob)
- That's not the one we want,
  - so we need to move entry forward one.
  - The only way to change the value of entry is an assignment
  - entry =
  - What do we set entry equal to?
- The loop should return when it finds Ian,

Finding a name

- To find Ian, we need to look at each entry.
  - Set the variable entry to the first one.
  - How do we do that?
  - Compare the name at entry to the name we are looking for.
  - How do we get the name at entry? (Bob)
- That's not the one we want,
  - so we need to move entry forward one.
  - The only way to change the value of entry is an assignment
  - entry =
  - What do we set entry equal to?
- The loop should return when it finds Ian,
  - but what if Ian is not there?

Finding a name

- To find Ian, we need to look at each entry.
  - Set the variable entry to the first one.
  - How do we do that?
  - Compare the name at entry to the name we are looking for.
  - How do we get the name at entry? (Bob)
- That's not the one we want,
  - so we need to move entry forward one.
  - The only way to change the value of entry is an assignment
  - entry =
  - What do we set entry equal to?
- The loop should return when it finds Ian,
  - but what if Ian is not there?
  - How can we tell?

Finding a name

- To find Ian, we need to look at each entry.
  - Set the variable entry to the first one.
  - How do we do that?
  - Compare the name at entry to the name we are looking for.
  - How do we get the name at entry? (Bob)
- That's not the one we want,
  - so we need to move entry forward one.
  - The only way to change the value of entry is an assignment
  - entry =
  - What do we set entry equal to?
- The loop should return when it finds Ian,
  - but what if Ian is not there?
  - How can we tell?
  - Two possibilities.

Removing an entry

Removing an entry

- removeEntry("Ian")

Removing an entry

- removeEntry("Ian")
  - calls find("Ian") to find its entry in the list.

Removing an entry

- removeEntry("Ian")
  - calls find("Ian") to find its entry in the list.
  - Then calls remove(position) with that entry.

Removing an entry

- removeEntry("Ian")
  - calls find("Ian") to find its entry in the list.
  - Then calls remove(position) with that entry.
  - remove(position) sets variables next and previous.

Removing an entry

- removeEntry("Ian")
  - calls find("Ian") to find its entry in the list.
  - Then calls remove(position) with that entry.
  - remove(position) sets variables next and previous.
  - How does it set them?

Removing an entry

- removeEntry("Ian")
  - calls find("Ian") to find its entry in the list.
  - Then calls remove(position) with that entry.
  - remove(position) sets variables next and previous.
  - How does it set them?
- Next remove must tell Eve's entry to use Jay's entry as its next entry.

Removing an entry

- removeEntry("Ian")
  - calls find("Ian") to find its entry in the list.
  - Then calls remove(position) with that entry.
  - remove(position) sets variables next and previous.
  - How does it set them?
- Next remove must tell Eve's entry to use Jay's entry as its next entry.
  - How do we set Eve's next entry pointer?

Removing an entry

- removeEntry("Ian")
  - calls find("Ian") to find its entry in the list.
  - Then calls remove(position) with that entry.
  - remove(position) sets variables next and previous.
  - How does it set them?
- Next remove must tell Eve's entry to use Jay's entry as its next entry.
  - How do we set Eve's next entry pointer?
  - What value do we give it?

Removing an entry

- removeEntry("Ian")
  - calls find("Ian") to find its entry in the list.
  - Then calls remove(position) with that entry.
  - remove(position) sets variables next and previous.
  - How does it set them?
- Next remove must tell Eve's entry to use Jay's entry as its next entry.
  - How do we set Eve's next entry pointer?
  - What value do we give it?
  - Similarly Jay's must point back to Eve's.

Removing an entry

- removeEntry("Ian")
  - calls find("Ian") to find its entry in the list.
  - Then calls remove(position) with that entry.
  - remove(position) sets variables next and previous.
  - How does it set them?
- Next remove must tell Eve's entry to use Jay's entry as its next entry.
  - How do we set Eve's next entry pointer?
  - What value do we give it?
  - Similarly Jay's must point back to Eve's.
- At this point Ian thinks he is still in the list,

Removing an entry

- removeEntry("Ian")
  - calls find("Ian") to find its entry in the list.
  - Then calls remove(position) with that entry.
  - remove(position) sets variables next and previous.
  - How does it set them?
- Next remove must tell Eve's entry to use Jay's entry as its next entry.
  - How do we set Eve's next entry pointer?
  - What value do we give it?
  - Similarly Jay's must point back to Eve's.
- At this point Ian thinks he is still in the list,
  - but he really isn't.

Removing an entry

- removeEntry("Ian")
  - calls find("Ian") to find its entry in the list.
  - Then calls remove(position) with that entry.
  - remove(position) sets variables next and previous.
  - How does it set them?
- Next remove must tell Eve's entry to use Jay's entry as its next entry.
  - How do we set Eve's next entry pointer?
  - What value do we give it?
  - Similarly Jay's must point back to Eve's.
- At this point Ian thinks he is still in the list,
  - but he really isn't.
  - Everyone is ignoring him!

Removing an entry

- removeEntry("Ian")
  - calls find("Ian") to find its entry in the list.
  - Then calls remove(position) with that entry.
  - remove(position) sets variables next and previous.
  - How does it set them?
- Next remove must tell Eve's entry to use Jay's entry as its next entry.
  - How do we set Eve's next entry pointer?
  - What value do we give it?
  - Similarly Jay's must point back to Eve's.
- At this point Ian thinks he is still in the list,
  - but he really isn't.
  - Everyone is ignoring him!
  - Similar to entries in array with index bigger than size.

LinkedMap find and add

LinkedMap find and add

- LinkedMap find must tell us where to put Ian if he is not there.

LinkedMap find and add

- LinkedMap find must tell us where to put Ian if he is not there.
  - In that case it returns the entry *before* Ian.

LinkedMap find and add

- LinkedMap find must tell us where to put Ian if he is not there.
  - In that case it returns the entry *before* Ian.
  - How does it know it has reached this entry?

LinkedMap find and add

- LinkedMap find must tell us where to put Ian if he is not there.
  - In that case it returns the entry *before* Ian.
  - How does it know it has reached this entry?
  - What does it return if we were adding Aaron?

LinkedMap find and add

- LinkedMap find must tell us where to put Ian if he is not there.
  - In that case it returns the entry *before* Ian.
  - How does it know it has reached this entry?
  - What does it return if we were adding Aaron?
- LinkedMap add uses the output of find.

LinkedMap find and add

- LinkedMap find must tell us where to put Ian if he is not there.
  - In that case it returns the entry *before* Ian.
  - How does it know it has reached this entry?
  - What does it return if we were adding Aaron?
- LinkedMap add uses the output of find.
  - add(previous) sets the variable next to the entry that should be next after Ian.

LinkedMap find and add

- LinkedMap find must tell us where to put Ian if he is not there.
  - In that case it returns the entry *before* Ian.
  - How does it know it has reached this entry?
  - What does it return if we were adding Aaron?
- LinkedMap add uses the output of find.
  - add(previous) sets the variable next to the entry that should be next after Ian.
  - How does it set next?

LinkedMap find and add

- LinkedMap find must tell us where to put Ian if he is not there.
  - In that case it returns the entry *before* Ian.
  - How does it know it has reached this entry?
  - What does it return if we were adding Aaron?
- LinkedMap add uses the output of find.
  - add(previous) sets the variable next to the entry that should be next after Ian.
  - How does it set next?
- To insert new entry Ian between next and previous, add has to set four pointers.

LinkedMap find and add

- LinkedMap find must tell us where to put Ian if he is not there.
  - In that case it returns the entry *before* Ian.
  - How does it know it has reached this entry?
  - What does it return if we were adding Aaron?
- LinkedMap add uses the output of find.
  - add(previous) sets the variable next to the entry that should be next after Ian.
  - How does it set next?
- To insert new entry Ian between next and previous, add has to set four pointers.
  - What are they?

LinkedMap find and add

- LinkedMap find must tell us where to put Ian if he is not there.
  - In that case it returns the entry *before* Ian.
  - How does it know it has reached this entry?
  - What does it return if we were adding Aaron?
- LinkedMap add uses the output of find.
  - add(previous) sets the variable next to the entry that should be next after Ian.
  - How does it set next?
- To insert new entry Ian between next and previous, add has to set four pointers.
  - What are they?
  - What values do we give them?

Keep it simple

Keep it simple

- Keep each line of your program simple

Keep it simple

- Keep each line of your program simple
- It should involve at most two variables.

Keep it simple

- Keep each line of your program simple
- It should involve at most two variables.
- For example:

Keep it simple

- Keep each line of your program simple
- It should involve at most two variables.
- For example:
  - entry = first;

Keep it simple

- Keep each line of your program simple
- It should involve at most two variables.
- For example:
  - entry = first;
  - previous = next.previous;

Keep it simple

- Keep each line of your program simple
- It should involve at most two variables.
- For example:
  - entry = first;
  - previous = next.previous;
  - entry.next = next;

Keep it simple

- Keep each line of your program simple
- It should involve at most two variables.
- For example:
  - entry = first;
  - previous = next.previous;
  - entry.next = next;
  - if (next == null)

Keep it simple

- Keep each line of your program simple
- It should involve at most two variables.
- For example:
  - entry = first;
  - previous = next.previous;
  - entry.next = next;
  - if (next == null)
- It should use only one .next or .previous.

Keep it simple

- Keep each line of your program simple
- It should involve at most two variables.
- For example:
  - entry = first;
  - previous = next.previous;
  - entry.next = next;
  - if (next == null)
- It should use only one .next or .previous.
- And the three parts of a for-loop control should each be considered a "line":

Keep it simple

- Keep each line of your program simple
- It should involve at most two variables.
- For example:
  - entry = first;
  - previous = next.previous;
  - entry.next = next;
  - if (next == null)
- It should use only one .next or .previous.
- And the three parts of a for-loop control should each be considered a "line":
  - for (line1; line2; line3) {

Keep it simple

- Keep each line of your program simple
- It should involve at most two variables.
- For example:
  - entry = first;
  - previous = next.previous;
  - entry.next = next;
  - if (next == null)
- It should use only one .next or .previous.
- And the three parts of a for-loop control should each be considered a "line":
  - for (line1; line2; line3) {
- Draw the diagram of what should happen.

Keep it simple

- Keep each line of your program simple
- It should involve at most two variables.
- For example:
  - entry = first;
  - previous = next.previous;
  - entry.next = next;
  - if (next == null)
- It should use only one .next or .previous.
- And the three parts of a for-loop control should each be considered a "line":
  - for (line1; line2; line3) {
- Draw the diagram of what should happen.
- Write the line that makes that change happen.

speed

speed

- find is still $\mathrm{O}(n)$:-(

speed

- find is still $\mathrm{O}(n)$:-(
- binary search doesn't help

speed

- find is still $\mathrm{O}(n)$:-(
- binary search doesn't help
- because it takes $\mathrm{O}(n)$ to get to the middle element!

speed

- find is still $\mathrm{O}(n)$:-(
- binary search doesn't help
- because it takes $\mathrm{O}(n)$ to get to the middle element!
- remove(position) is now $\mathrm{O}(1)$!

speed

- find is still $O(n)$:-(
- binary search doesn't help
- because it takes $O(n)$ to get to the middle element!
- remove(position) is now $O(1)$!
- but remove(name) (removeEntry(name)) must call find

speed

- find is still $\mathrm{O}(n)$:-(
- binary search doesn't help
- because it takes $\mathrm{O}(n)$ to get to the middle element!
- remove(position) is now $\mathrm{O}(1)$!
- but remove(name) (removeEntry(name)) must call find
- so it is still $\mathrm{O}(n)$

speed

- find is still $O(n)$:-(
- binary search doesn't help
- because it takes $O(n)$ to get to the middle element!
- remove(position) is now $O(1)$!
- but remove(name) (removeEntry(name)) must call find
- so it is still $O(n)$
- add is now $O(1)$

speed

- find is still $O(n)$:-(
- binary search doesn't help
- because it takes $O(n)$ to get to the middle element!
- remove(position) is now $O(1)$!
- but remove(name) (removeEntry(name)) must call find
- so it is still $O(n)$
- add is now $O(1)$
- but put (addOrChangeEntry) must call find

speed

- find is still $O(n)$:-(
- binary search doesn't help
- because it takes $O(n)$ to get to the middle element!
- remove(position) is now $O(1)$!
- but remove(name) (removeEntry(name)) must call find
- so it is still $O(n)$
- add is now $O(1)$
- but put (addOrChangeEntry) must call find
- so it is still $O(n)$.

speed

- find is still $\mathrm{O}(n)$:-(
- binary search doesn't help
- because it takes $\mathrm{O}(n)$ to get to the middle element!
- remove(position) is now $\mathrm{O}(1)$!
- but remove(name) (removeEntry(name)) must call find
- so it is still $\mathrm{O}(n)$
- add is now $\mathrm{O}(1)$
- but put (addOrChangeEntry) must call find
- so it is still $\mathrm{O}(n)$.
- One step forward, two steps back!

Summary

Summary

- The *(doubly) linked list* is a new way to store a list.

Summary

- The *(doubly) linked list* is a new way to store a list.
  - Adding or removing an entry *at a known location* is $O(1)$,

Summary

- The *(doubly) linked list* is a new way to store a list.
  - Adding or removing an entry *at a known location* is $O(1)$,
  - in contrast to $O(n)$ for an array.

Summary

- The *(doubly) linked list* is a new way to store a list.
  - Adding or removing an entry *at a known location* is $O(1)$,
  - in contrast to $O(n)$ for an array.
  - But getting to the ith element takes $O(n)$,

Summary

- The *(doubly) linked list* is a new way to store a list.
  - Adding or removing an entry *at a known location* is $\mathrm{O}(1)$,
  - in contrast to $\mathrm{O}(n)$ for an array.
  - But getting to the *i*th element takes $\mathrm{O}(n)$,
  - in contrast to $\mathrm{O}(1)$ for an array.

Summary

- The *(doubly) linked list* is a new way to store a list.
  - Adding or removing an entry *at a known location* is $\mathrm{O}(1)$,
  - in contrast to $\mathrm{O}(n)$ for an array.
  - But getting to the ith element takes $\mathrm{O}(n)$,
  - in contrast to $\mathrm{O}(1)$ for an array.
  - We will have to keeping working on improving the running time.

Summary

- The *(doubly) linked list* is a new way to store a list.
  - Adding or removing an entry *at a known location* is $\mathrm{O}(1)$,
  - in contrast to $\mathrm{O}(n)$ for an array.
  - But getting to the *i*th element takes $\mathrm{O}(n)$,
  - in contrast to $\mathrm{O}(1)$ for an array.
  - We will have to keeping working on improving the running time.
- When programming a linked list:

Summary

- The *(doubly) linked list* is a new way to store a list.
  - Adding or removing an entry *at a known location* is $\mathrm{O}(1)$,
  - in contrast to $\mathrm{O}(n)$ for an array.
  - But getting to the ith element takes $\mathrm{O}(n)$,
  - in contrast to $\mathrm{O}(1)$ for an array.
  - We will have to keeping working on improving the running time.
- When programming a linked list:
  - Draw the diagram of each change.

Summary

- The *(doubly) linked list* is a new way to store a list.
  - Adding or removing an entry *at a known location* is $O(1)$,
  - in contrast to $O(n)$ for an array.
  - But getting to the *i*th element takes $O(n)$,
  - in contrast to $O(1)$ for an array.
  - We will have to keeping working on improving the running time.
- When programming a linked list:
  - Draw the diagram of each change.
  - Program each change as a line

Summary

- The *(doubly) linked list* is a new way to store a list.
  - Adding or removing an entry *at a known location* is $\mathrm{O}(1)$,
  - in contrast to $\mathrm{O}(n)$ for an array.
  - But getting to the ith element takes $\mathrm{O}(n)$,
  - in contrast to $\mathrm{O}(1)$ for an array.
  - We will have to keeping working on improving the running time.
- When programming a linked list:
  - Draw the diagram of each change.
  - Program each change as a line
  - with only two variables.

Summary

- The *(doubly) linked list* is a new way to store a list.
  - Adding or removing an entry *at a known location* is $O(1)$,
  - in contrast to $O(n)$ for an array.
  - But getting to the *i*th element takes $O(n)$,
  - in contrast to $O(1)$ for an array.
  - We will have to keeping working on improving the running time.
- When programming a linked list:
  - Draw the diagram of each change.
  - Program each change as a line
  - with only two variables.
  - Keep each step simple!

This week's application

This week's application

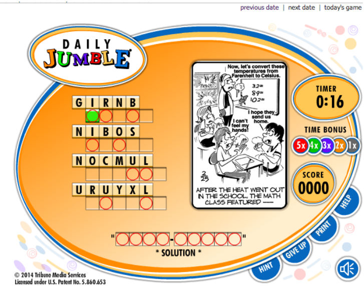
- We need a nice application for our Map.

This week's application

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

This week's application

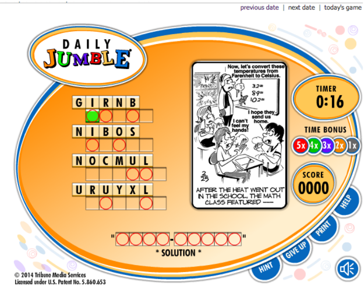
- We need a nice application for our Map.
  - Yet another game!



  - Daily Jumble

This week's application

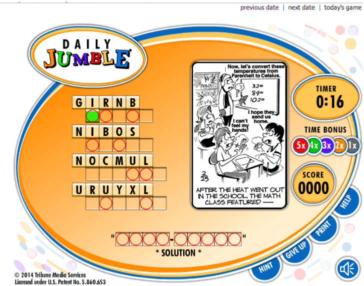
- We need a nice application for our Map.
  - Yet another game!



  - Daily Jumble
- Need to unscramble words.

This week's application

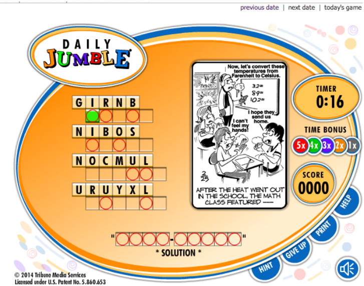
- We need a nice application for our Map.
  - Yet another game!



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

This week's application

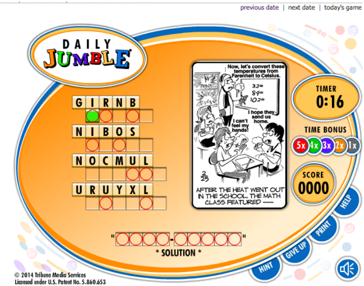
- We need a nice application for our Map.
  - Yet another game!



  - 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.

We’re going to need a bigger...dictionary.

We’re going to need a bigger...dictionary.

- words doesn’t have the solution to “zagboe”

We’re going to need a bigger...dictionary.

- words doesn’t have the solution to “zagboe”
- Let’s try dict.

We’re going to need a bigger...dictionary.

- words doesn’t have the solution to “zagboe”
- Let’s try dict. 483423 words!!

We’re going to need a bigger...dictionary.

- words doesn’t have the solution to “zagboe”
- Let’s try dict. 483423 words!!
- Run Jumble using LinkedMap.

We're going to need a bigger...dictionary.

- words doesn't have the solution to "zagboe"
- Let's try dict. 483423 words!!
- Run Jumble using LinkedMap. Seems to be taking a while...

We’re going to need a bigger...dictionary.

- words doesn’t have the solution to “zagboe”
- Let’s try dict. 483423 words!!
- Run Jumble using LinkedMap. Seems to be taking a while...
- put (addOrChangeEntry) has to call find.

We're going to need a bigger...dictionary.

- words doesn't have the solution to "zagboe"
- Let's try dict. 483423 words!!
- Run Jumble using LinkedMap. Seems to be taking a while...
- put (addOrChangeEntry) has to call find. Which is $O(n)$.

We're going to need a bigger...dictionary.

- words doesn't have the solution to "zagboe"
- Let's try dict. 483423 words!!
- Run Jumble using LinkedMap. Seems to be taking a while...
- put (addOrChangeEntry) has to call find. Which is $O(n)$.
- What is the $O()$ to read all n words?

We’re going to need a bigger...dictionary.

- words doesn’t have the solution to “zagboe”
- Let’s try dict. 483423 words!!
- Run Jumble using LinkedMap. Seems to be taking a while...
- put (addOrChangeEntry) has to call find. Which is $\mathrm{O}(n)$.
- What is the $\mathrm{O}()$ to read all n words?
- n times $\mathrm{O}(n)$?

We're going to need a bigger...dictionary.

- words doesn't have the solution to "zagboe"
- Let's try dict. 483423 words!!
- Run Jumble using LinkedMap. Seems to be taking a while...
- put (addOrChangeEntry) has to call find. Which is $O(n)$.
- What is the $O()$ to read all n words?
- n times $O(n)$?
- $n \cdot (c \cdot n) = c \cdot n^2$

We're going to need a bigger...dictionary.

- words doesn't have the solution to "zagboe"
- Let's try dict. 483423 words!!
- Run Jumble using LinkedMap. Seems to be taking a while...
- put (addOrChangeEntry) has to call find. Which is $\mathrm{O}(n)$.
- What is the $\mathrm{O}()$ to read all n words?
- n times $\mathrm{O}(n)$?
- $n \cdot (c \cdot n) = c \cdot n^2$
- $\mathrm{O}(n^2)$!

We're going to need a bigger...dictionary.

- words doesn't have the solution to "zagboe"
- Let's try dict. 483423 words!!
- Run Jumble using LinkedMap. Seems to be taking a while...
- put (addOrChangeEntry) has to call find. Which is $\mathrm{O}(n)$.
- What is the $\mathrm{O}()$ to read all n words?
- n times $\mathrm{O}(n)$?
- $n \cdot (c \cdot n) = c \cdot n^2$
- $\mathrm{O}(n^2)$!
- $n^2 = 233{,}697{,}796{,}929$.

We're going to need a bigger...dictionary.

- words doesn't have the solution to "zagboe"
- Let's try dict. 483423 words!!
- Run Jumble using LinkedMap. Seems to be taking a while...
- put (addOrChangeEntry) has to call find. Which is $\mathrm{O}(n)$.
- What is the $\mathrm{O}()$ to read all n words?
- n times $\mathrm{O}(n)$?
- $n \cdot (c \cdot n) = c \cdot n^2$
- $\mathrm{O}(n^2)$!
- $n^2 = 233{,}697{,}796{,}929$.
- A half million squared is a quarter trillion.

We're going to need a bigger...dictionary.

- words doesn't have the solution to "zagboe"
- Let's try dict. 483423 words!!
- Run Jumble using LinkedMap. Seems to be taking a while...
- put (addOrChangeEntry) has to call find. Which is $\mathrm{O}(n)$.
- What is the $\mathrm{O}()$ to read all n words?
- n times $\mathrm{O}(n)$?
- $n \cdot (c \cdot n) = c \cdot n^2$
- $\mathrm{O}(n^2)$!
- $n^2 = 233{,}697{,}796{,}929$.
- A half million squared is a quarter trillion.
- A computer that can do a billion operations in a second

We're going to need a bigger...dictionary.

- words doesn't have the solution to "zagboe"
- Let's try dict. 483423 words!!
- Run Jumble using LinkedMap. Seems to be taking a while...
- put (addOrChangeEntry) has to call find. Which is $\mathrm{O}(n)$.
- What is the $\mathrm{O}()$ to read all n words?
- n times $\mathrm{O}(n)$?
- $n \cdot (c \cdot n) = c \cdot n^2$
- $\mathrm{O}(n^2)$!
- $n^2 = 233{,}697{,}796{,}929$.
- A half million squared is a quarter trillion.
- A computer that can do a billion operations in a second
- will take 233 seconds times the number of operations per find.