

Linked Lists

Victor Milenkovic

Department of Computer Science
University of Miami

CSC220 Programming II – Spring 2019



Outline

Arrays are slow

Arrays are slow

- ▶ Speed of SortedPD addOrChangeEntry:



Arrays are slow

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



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.



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 $- 1$.



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 $O(\log n)$.
 - ▶ 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.
 - ▶ 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 **DLLEntry** 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 **DLLEntry class**.
 - ▶ Extends prog02.DirectoryEntry.



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 **DLLEntry** class.
 - ▶ Extends `prog02.DirectoryEntry`.
 - ▶ Adds 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 **DLLEntry** class.
 - ▶ Extends prog02.DirectoryEntry.
 - ▶ Adds next and previous field
 - ▶ with get and set 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 **DLLEntry class**.
 - ▶ Extends prog02.DirectoryEntry.
 - ▶ Adds next and previous field
 - ▶ with get and set 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 getNext() or getPrevious() 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 getNext() or getPrevious() 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 getNext() or getPrevious() 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 getNext() or getPrevious() repeatedly.
 - ▶ Get to any other element.
- ▶ For convenience, it is customary to store references to
 - ▶ **head**, 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 getNext() or getPrevious() repeatedly.
 - ▶ Get to any other element.
- ▶ For convenience, it is customary to store references to
 - ▶ **head**, the first entry in the list
 - ▶ **tail**, 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 getNext() or getPrevious() repeatedly.
 - ▶ Get to any other element.
- ▶ For convenience, it is customary to store references to
 - ▶ **head**, the first entry in the list
 - ▶ **tail**, 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? (Jay)



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? (Jay)
- ▶ 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? (Jay)
- ▶ 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? (Jay)
- ▶ 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? (Jay)
- ▶ 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? (Jay)
- ▶ 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? (Jay)
- ▶ 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 lan, 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? (Jay)
- ▶ 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 lan,
 - ▶ but what if lan 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? (Jay)
- ▶ 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?
 - ▶ What will stop the loop?



Finding a name

- ▶ To find lan, 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? (Jay)
- ▶ 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 lan,
 - ▶ but what if lan is not there?
 - ▶ What will stop the loop?
 - ▶ What value of entry tells us that we have seen everything in the list?



Removing an entry

Removing an entry

- ▶ `removeEntry`

Removing an entry

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



Removing an entry

- ▶ removeEntry
 - ▶ calls find("Ian") to find its entry in the list.
 - ▶ Then it sets variables next and previous.



Removing an entry

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



Removing an entry

- ▶ removeEntry
 - ▶ calls find("Ian") to find its entry in the list.
 - ▶ Then it sets variables next and previous.
 - ▶ How does it set them?
- ▶ Next removeEntry must tell Zoe's entry to use Ann's entry as its next entry.



Removing an entry

- ▶ removeEntry
 - ▶ calls find("Ian") to find its entry in the list.
 - ▶ Then it sets variables next and previous.
 - ▶ How does it set them?
- ▶ Next removeEntry must tell Zoe's entry to use Ann's entry as its next entry.
 - ▶ What method sets Zoe's next entry pointer?



Removing an entry

- ▶ removeEntry
 - ▶ calls find("Ian") to find its entry in the list.
 - ▶ Then it sets variables next and previous.
 - ▶ How does it set them?
- ▶ Next removeEntry must tell Zoe's entry to use Ann's entry as its next entry.
 - ▶ What method sets Zoe's next entry pointer?
 - ▶ How do we invoke it?



Removing an entry

- ▶ removeEntry
 - ▶ calls find("Ian") to find its entry in the list.
 - ▶ Then it sets variables next and previous.
 - ▶ How does it set them?
- ▶ Next removeEntry must tell Zoe's entry to use Ann's entry as its next entry.
 - ▶ What method sets Zoe's next entry pointer?
 - ▶ How do we invoke it?
 - ▶ What value do we give it?



Removing an entry

- ▶ removeEntry
 - ▶ calls find("Ian") to find its entry in the list.
 - ▶ Then it sets variables next and previous.
 - ▶ How does it set them?
- ▶ Next removeEntry must tell Zoe's entry to use Ann's entry as its next entry.
 - ▶ What method sets Zoe's next entry pointer?
 - ▶ How do we invoke it?
 - ▶ What value do we give it?
 - ▶ Similarly Ann's must point back to Zoe's.



Removing an entry

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



Removing an entry

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



Removing an entry

- ▶ removeEntry
 - ▶ calls find("Ian") to find its entry in the list.
 - ▶ Then it sets variables next and previous.
 - ▶ How does it set them?
- ▶ Next removeEntry must tell Zoe's entry to use Ann's entry as its next entry.
 - ▶ What method sets Zoe's next entry pointer?
 - ▶ How do we invoke it?
 - ▶ What value do we give it?
 - ▶ Similarly Ann's must point back to Zoe'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
 - ▶ calls find("Ian") to find its entry in the list.
 - ▶ Then it sets variables next and previous.
 - ▶ How does it set them?
- ▶ Next removeEntry must tell Zoe's entry to use Ann's entry as its next entry.
 - ▶ What method sets Zoe's next entry pointer?
 - ▶ How do we invoke it?
 - ▶ What value do we give it?
 - ▶ Similarly Ann's must point back to Zoe'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.



Adding Ian

Adding Ian

- ▶ How do we add Ian back again?



Adding Ian

- ▶ How do we add Ian back again?
 - ▶ Let's just put him at the end.



Adding Ian

- ▶ How do we add Ian back again?
 - ▶ Let's just put him at the end.
 - ▶ Three values have to change for this to happen.



Adding Ian

- ▶ How do we add Ian back again?
 - ▶ Let's just put him at the end.
 - ▶ Three values have to change for this to happen.
 - ▶ Which values?



Adding Ian

- ▶ How do we add Ian back again?
 - ▶ Let's just put him at the end.
 - ▶ Three values have to change for this to happen.
 - ▶ Which values?
 - ▶ How do we set them?



Adding Ian

- ▶ How do we add Ian back again?
 - ▶ Let's just put him at the end.
 - ▶ Three values have to change for this to happen.
 - ▶ Which values?
 - ▶ How do we set them?
 - ▶ To what?



SortedDLLPD find and add

SortedDLLPD find and add

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



SortedDLLPD find and add

- ▶ SortedDLLPD find must tell us where to put lan if he is not there.
 - ▶ In that case it returns the entry after lan.



SortedDLLPD find and add

- ▶ SortedDLLPD find must tell us where to put lan if he is not there.
 - ▶ In that case it returns the entry after lan.
 - ▶ How does it know it has reached this entry?



SortedDLLPD find and add

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



SortedDLLPD find and add

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



SortedDLLPD find and add

- ▶ SortedDLLPD find must tell us where to put lan if he is not there.
 - ▶ In that case it returns the entry after lan.
 - ▶ How does it know it has reached this entry?
 - ▶ What does it return if we were adding Zora?
- ▶ SortedDLLPD add uses the output of find.
 - ▶ It sets the variable next to the entry that should be next after lan.



SortedDLLPD find and add

- ▶ SortedDLLPD find must tell us where to put lan if he is not there.
 - ▶ In that case it returns the entry after lan.
 - ▶ How does it know it has reached this entry?
 - ▶ What does it return if we were adding Zora?
- ▶ SortedDLLPD add uses the output of find.
 - ▶ It sets the variable next to the entry that should be next after lan.
 - ▶ How does it set next? (Easy!!)



SortedDLLPD find and add

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



SortedDLLPD find and add

- ▶ SortedDLLPD find must tell us where to put lan if he is not there.
 - ▶ In that case it returns the entry after lan.
 - ▶ How does it know it has reached this entry?
 - ▶ What does it return if we were adding Zora?
- ▶ SortedDLLPD add uses the output of find.
 - ▶ It sets the variable next to the entry that should be next after lan.
 - ▶ How does it set next? (Easy!!)
 - ▶ It sets the variable previous to the entry that should be before lan.
 - ▶ How does it set previous? (A little harder.)



SortedDLLPD find and add

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



SortedDLLPD find and add

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



SortedDLLPD find and add

- ▶ SortedDLLPD find must tell us where to put lan if he is not there.
 - ▶ In that case it returns the entry after lan.
 - ▶ How does it know it has reached this entry?
 - ▶ What does it return if we were adding Zora?
- ▶ SortedDLLPD add uses the output of find.
 - ▶ It sets the variable next to the entry that should be next after lan.
 - ▶ How does it set next? (Easy!!)
 - ▶ It sets the variable previous to the entry that should be before lan.
 - ▶ How does it set previous? (A little harder.)
- ▶ To insert new entry lan between next and previous, add has to set four pointers.
 - ▶ What are they?
 - ▶ What method do we use?



SortedDLLPD find and add

- ▶ SortedDLLPD find must tell us where to put lan if he is not there.
 - ▶ In that case it returns the entry after lan.
 - ▶ How does it know it has reached this entry?
 - ▶ What does it return if we were adding Zora?
- ▶ SortedDLLPD add uses the output of find.
 - ▶ It sets the variable next to the entry that should be next after lan.
 - ▶ How does it set next? (Easy!!)
 - ▶ It sets the variable previous to the entry that should be before lan.
 - ▶ How does it set previous? (A little harder.)
- ▶ To insert new entry lan between next and previous, add has to set four pointers.
 - ▶ What are they?
 - ▶ What method do we use?
 - ▶ For each of the four times, how do we call it and what is the value?



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 = head;`



Keep it simple

- ▶ Keep each line of your program simple
- ▶ It should involve at most two variables.
- ▶ For example:
 - ▶ `entry = head;`
 - ▶ `previous = next.getPrevious();`



Keep it simple

- ▶ Keep each line of your program simple
- ▶ It should involve at most two variables.
- ▶ For example:
 - ▶ `entry = head;`
 - ▶ `previous = next.getPrevious();`
 - ▶ `entry.setNext(next);`



Keep it simple

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



Keep it simple

- ▶ Keep each line of your program simple
- ▶ It should involve at most two variables.
- ▶ For example:
 - ▶ `entry = head;`
 - ▶ `previous = next.getPrevious();`
 - ▶ `entry.setNext(next);`
 - ▶ `if (next == null)`
- ▶ Of course you will use `getNext`, `setPrevious`, and other variable names!



Keep it simple

- ▶ Keep each line of your program simple
- ▶ It should involve at most two variables.
- ▶ For example:
 - ▶ `entry = head;`
 - ▶ `previous = next.getPrevious();`
 - ▶ `entry.setNext(next);`
 - ▶ `if (next == null)`
- ▶ Of course you will use `getNext`, `setPrevious`, and other variable names!
- ▶ 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 = head;`
 - ▶ `previous = next.getPrevious();`
 - ▶ `entry.setNext(next);`
 - ▶ `if (next == null)`
- ▶ Of course you will use `getNext`, `setPrevious`, and other variable names!
- ▶ 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 = head;`
 - ▶ `previous = next.getPrevious();`
 - ▶ `entry.setNext(next);`
 - ▶ `if (next == null)`
- ▶ Of course you will use `getNext`, `setPrevious`, and other variable names!
- ▶ 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 = head;`
 - ▶ `previous = next.getPrevious();`
 - ▶ `entry.setNext(next);`
 - ▶ `if (next == null)`
- ▶ Of course you will use `getNext`, `setPrevious`, and other variable names!
- ▶ 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

- ▶ For DLLBasedPD (unsorted):



- ▶ For DLLBasedPD (unsorted):
 - ▶ find is still $O(n)$

- ▶ For DLLBasedPD (unsorted):
 - ▶ find is still $O(n)$
 - ▶ removal is still $O(1)$, but for a different reason

- ▶ For DLLBasedPD (unsorted):
 - ▶ find is still $O(n)$
 - ▶ removal is still $O(1)$, but for a different reason
 - ▶ but removeEntry must call find

- ▶ For DLLBasedPD (unsorted):
 - ▶ find is still $O(n)$
 - ▶ removal is still $O(1)$, but for a different reason
 - ▶ but removeEntry must call find
 - ▶ so it is still $O(n)$

- ▶ For DLLBasedPD (unsorted):
 - ▶ find is still $O(n)$
 - ▶ removal is still $O(1)$, but for a different reason
 - ▶ but removeEntry must call find
 - ▶ so it is still $O(n)$
 - ▶ Similarly, addOrChangeEntry is still $O(n)$

- ▶ For DLLBasedPD (unsorted):
 - ▶ find is still $O(n)$
 - ▶ removal is still $O(1)$, but for a different reason
 - ▶ but removeEntry must call find
 - ▶ so it is still $O(n)$
 - ▶ Similarly, addOrChangeEntry is still $O(n)$
- ▶ For SortedDLLPD (sorted):

- ▶ For DLLBasedPD (unsorted):
 - ▶ find is still $O(n)$
 - ▶ removal is still $O(1)$, but for a different reason
 - ▶ but removeEntry must call find
 - ▶ so it is still $O(n)$
 - ▶ Similarly, addOrChangeEntry is still $O(n)$
- ▶ For SortedDLLPD (sorted):
 - ▶ find is still $O(n)$:-)

- ▶ For DLLBasedPD (unsorted):
 - ▶ find is still $O(n)$
 - ▶ removal is still $O(1)$, but for a different reason
 - ▶ but removeEntry must call find
 - ▶ so it is still $O(n)$
 - ▶ Similarly, addOrChangeEntry is still $O(n)$
- ▶ For SortedDLLPD (sorted):
 - ▶ find is still $O(n)$:-(
 - ▶ binary search doesn't help

- ▶ For DLLBasedPD (unsorted):
 - ▶ find is still $O(n)$
 - ▶ removal is still $O(1)$, but for a different reason
 - ▶ but removeEntry must call find
 - ▶ so it is still $O(n)$
 - ▶ Similarly, addOrChangeEntry is still $O(n)$
- ▶ For SortedDLLPD (sorted):
 - ▶ find is still $O(n)$:-)
 - ▶ binary search doesn't help
 - ▶ because it takes $O(n)$ to get to the middle element!

- ▶ For DLLBasedPD (unsorted):
 - ▶ find is still $O(n)$
 - ▶ removal is still $O(1)$, but for a different reason
 - ▶ but removeEntry must call find
 - ▶ so it is still $O(n)$
 - ▶ Similarly, addOrChangeEntry is still $O(n)$
- ▶ For SortedDLLPD (sorted):
 - ▶ find is still $O(n)$:-)
 - ▶ binary search doesn't help
 - ▶ because it takes $O(n)$ to get to the middle element!
 - ▶ removal is now $O(1)$!

- ▶ For DLLBasedPD (unsorted):
 - ▶ find is still $O(n)$
 - ▶ removal is still $O(1)$, but for a different reason
 - ▶ but removeEntry must call find
 - ▶ so it is still $O(n)$
 - ▶ Similarly, addOrChangeEntry is still $O(n)$
- ▶ For SortedDLLPD (sorted):
 - ▶ find is still $O(n)$:-(
 - ▶ binary search doesn't help
 - ▶ because it takes $O(n)$ to get to the middle element!
 - ▶ removal is now $O(1)$!
 - ▶ but removeEntry must call find

- ▶ For DLLBasedPD (unsorted):
 - ▶ find is still $O(n)$
 - ▶ removal is still $O(1)$, but for a different reason
 - ▶ but removeEntry must call find
 - ▶ so it is still $O(n)$
 - ▶ Similarly, addOrChangeEntry is still $O(n)$
- ▶ For SortedDLLPD (sorted):
 - ▶ find is still $O(n)$:-(
 - ▶ binary search doesn't help
 - ▶ because it takes $O(n)$ to get to the middle element!
 - ▶ removal is now $O(1)$!
 - ▶ but removeEntry must call find
 - ▶ so it is still $O(n)$

- ▶ For DLLBasedPD (unsorted):
 - ▶ find is still $O(n)$
 - ▶ removal is still $O(1)$, but for a different reason
 - ▶ but removeEntry must call find
 - ▶ so it is still $O(n)$
 - ▶ Similarly, addOrChangeEntry is still $O(n)$
- ▶ For SortedDLLPD (sorted):
 - ▶ find is still $O(n)$:-(
 - ▶ binary search doesn't help
 - ▶ because it takes $O(n)$ to get to the middle element!
 - ▶ removal is now $O(1)$!
 - ▶ but removeEntry must call find
 - ▶ so it is still $O(n)$
 - ▶ add is now $O(1)$

- ▶ For DLLBasedPD (unsorted):
 - ▶ find is still $O(n)$
 - ▶ removal is still $O(1)$, but for a different reason
 - ▶ but removeEntry must call find
 - ▶ so it is still $O(n)$
 - ▶ Similarly, addOrChangeEntry is still $O(n)$
- ▶ For SortedDLLPD (sorted):
 - ▶ find is still $O(n)$:-(
 - ▶ binary search doesn't help
 - ▶ because it takes $O(n)$ to get to the middle element!
 - ▶ removal is now $O(1)$!
 - ▶ but removeEntry must call find
 - ▶ so it is still $O(n)$
 - ▶ add is now $O(1)$
 - ▶ but addOrChangeEntry must call find

- ▶ For DLLBasedPD (unsorted):
 - ▶ find is still $O(n)$
 - ▶ removal is still $O(1)$, but for a different reason
 - ▶ but removeEntry must call find
 - ▶ so it is still $O(n)$
 - ▶ Similarly, addOrChangeEntry is still $O(n)$
- ▶ For SortedDLLPD (sorted):
 - ▶ find is still $O(n)$:-(
 - ▶ binary search doesn't help
 - ▶ because it takes $O(n)$ to get to the middle element!
 - ▶ removal is now $O(1)$!
 - ▶ but removeEntry must call find
 - ▶ so it is still $O(n)$
 - ▶ add is now $O(1)$
 - ▶ but addOrChangeEntry must call find
 - ▶ so it is still $O(n)$.

- ▶ For DLLBasedPD (unsorted):
 - ▶ find is still $O(n)$
 - ▶ removal is still $O(1)$, but for a different reason
 - ▶ but removeEntry must call find
 - ▶ so it is still $O(n)$
 - ▶ Similarly, addOrChangeEntry is still $O(n)$
- ▶ For SortedDLLPD (sorted):
 - ▶ find is still $O(n)$:- (
 - ▶ binary search doesn't help
 - ▶ because it takes $O(n)$ to get to the middle element!
 - ▶ removal is now $O(1)$!
 - ▶ but removeEntry must call find
 - ▶ so it is still $O(n)$
 - ▶ add is now $O(1)$
 - ▶ but addOrChangeEntry must call find
 - ▶ so it is still $O(n)$.
- ▶ 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 i th 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 $O(1)$,
 - ▶ 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.



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 keep 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 $O(1)$,
 - ▶ 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 keep 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 $O(1)$,
 - ▶ 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 keep 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 keep 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 $O(1)$,
 - ▶ 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 keep 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 keep 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!

