20.1 List abstract data type (ADT)

List abstract data type

A ***list*** is a common ADT for holding ordered data, having operations like append a data item, remove a data item, check if a data item exists, and print the list. Ex: For a given list item, after "Append 7", "Append 9", and "Append 5", then "Print" will print (7, 9, 5) in that order, and "Search 8" would indicate item not found. A user need not have knowledge of the internal implementation of the list ADT. Examples in this section assume the data items are integers, but a list commonly holds other kinds of data like strings or entire objects.

PARTICIPATION ACTIVITY 20.1.1: List ADT.

Start 2x speed

```
ages = new List()
Append(ages, 55)
Append(ages, 88)
Append(ages, 66)
Print(ages)
Remove(ages, 88)
Print(ages)
```

ages: 55 66

Print result: 55, 88, 66

Print result: 55, 66

Captions

PARTICIPATION ACTIVITY 20.1.2: List ADT.

Type the list after the given operations. Each question starts with an empty list. Type the list as: 5, 7, 9

1) Append(list, 3)
   Append(list, 2)

Check Show answer

2) Append(list, 3)
Append(list, 2)
Append(list, 1)
Remove(list, 3)

Check Show answer

3) After the following operations, will Search(list, 2) find an item? Type yes or no.

Append(list, 3)
Append(list, 2)
Append(list, 1)
Remove(list, 2)

Check Show answer

Common list ADT operations

Table 20.1.1: Some common operations for a list ADT.

Operation	Description	Example starting with list: (99, 77)
Append(list, X)	Inserts X at end of list	`Append(list, 44)` list: (99, 77, 44)
Prepend(list, X)	Inserts X at start of list	`Prepend(list, 44)` list: (44, 99, 77)
InsertAfter(list, W, X)	Inserts X after W	`InsertAfter(list, 99, 44)` list: (99, 44, 77)
Remove(list, X)	Removes X	`Remove(list, 77)` list: (99)
Contains(list, X)	Returns true if X is in the list, false otherwise	`Contains(list, 99)` returns true `Contains(list, 22)` returns false

Print(list)	Prints list's items in order	`Print(list)` outputs: 99, 77
Sort(list)	Sorts list's items in ascending order	`Sort(list)` list: (77, 99)
IsEmpty(list)	Returns true if list has no items	`IsEmpty(list)` returns false
GetLength(list)	Returns the number of items in the list	`GetLength(list)` returns 2

PARTICIPATION ACTIVITY 20.1.3: List ADT common operations.

1) Given the list: (40, 888, -3, 2).
`GetLength(list)` ____.
   - returns 4
   - fails

2) Given the list: ('Z', 'A', 'B'), Sort(list) yields ('A', 'B', 'Z').
   - True
   - False

3) To support all operations listed in the table above, a list must be implemented using an array.
   - True
   - False

4) The table above lists all common list ADT operations.
   - True
   - False

20.2 Singly-linked lists

Singly-linked list data structure

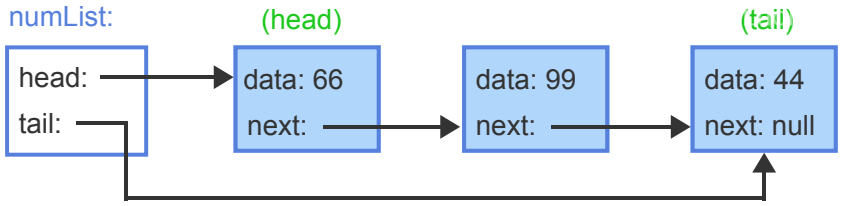
A ***singly-linked list*** is a data structure for implementing a list ADT, where each node has data and a pointer to the next node. The list structure typically has pointers to the list's first node and last node. A singly-linked

list's first node is called the ***head***, and the last node the ***tail***. A singly-linked list is a type of ***positional list***: A list where elements contain pointers to the next and/or previous elements in the list.

PARTICIPATION ACTIVITY 20.2.1: Singly-linked list: Each node points to the next node.

Start 2x speed

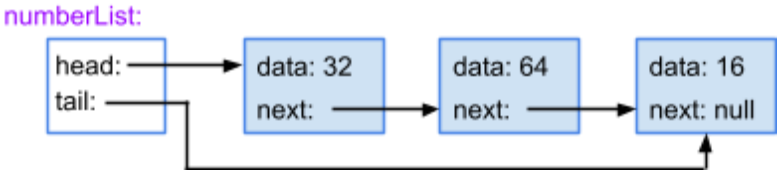
1. Create new list: numList
2. Append 99
3. Append 44
4. Prepend 66



Captions

PARTICIPATION ACTIVITY 20.2.2: Singly-linked list data structure.

Consider the following singly-linked list:



1) The head node's data value is _____.

- null
- 32
- 16

2) The tail node's data value is _____.

- 32
- 64
- 16

3) Node 32's next pointer value is _____.

- node 32
- node 64
- node 16

Implementing a list ADT with a singly-linked list

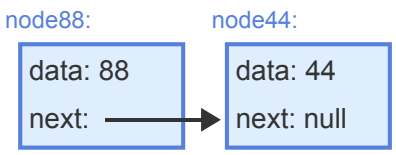
A singly-linked list is a common data structure used to implement a list ADT. List operations that insert new items, like the append, prepend, and insert-after operations, each require allocation of a new linked list node.

Node creation is specific to the programming language in use, but three steps are common between most languages: allocate space for the node, assign data with an initial value, and assign next with null.

PARTICIPATION ACTIVITY 20.2.3: Allocating singly-linked list nodes.

Start 2x speed

```
node88 = Allocate new singly-linked list node
node88→data = 88
node88→next = null

node44 = Allocate new singly-linked list node
node44→data = 44
node44→next = null

node88→next = node44
```



Captions

PARTICIPATION ACTIVITY 20.2.4: Allocating singly-linked list nodes.

Refer to the animation above.

1) node88 is a _____ instance.

- singly-linked list
- singly-linked list node

2) If the third line is changed from `node88→next = null` to `node88→next = node44`, the code produces the same result.

- True
- False

PARTICIPATION ACTIVITY 20.2.5: List ADT.

1) A singly-linked list _____ be used to implement a list ADT.

- is the only data structure that can
- is one of many data structures that can
- data structure can *not*

Append operation

The singly-linked list ***append*** operation inserts a new node after the list's tail node. The append operation's algorithm behavior differs if the list is empty versus not empty:

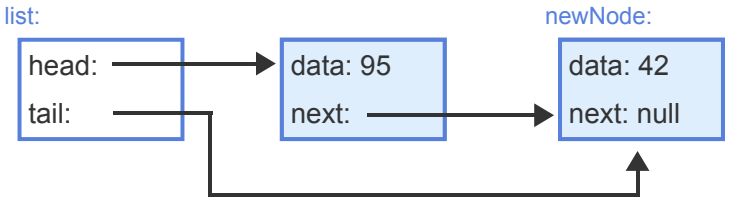
- *Append to empty list:* If the list's head is null, the list's head and tail are assigned with the new node.
- *Append to non-empty list:* If the list's head is not null, the tail node's next is first assigned with the new node, then the list's tail is assigned with the new node.

PARTICIPATION ACTIVITY 20.2.6: Singly-linked list: Append operation.

Start 2x speed

```
ListAppend(list, item) {
   newNode = Allocate new singly-linked list node
   newNode→data = item
   newNode→next = null
   ListAppendNode(list, newNode)
}

ListAppendNode(list, newNode) {
   if (list→head == null) { // List empty
      list→head = newNode
      list→tail = newNode
   }
   else {
      list→tail→next = newNode
      list→tail = newNode
   }
}
```

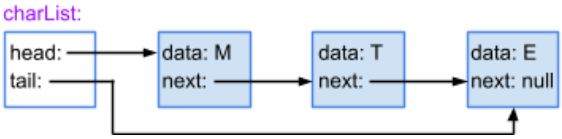
```
ListAppend(list, 95) ✓
ListAppend(list, 42) ✓
```



Captions

PARTICIPATION ACTIVITY 20.2.7: Appending a node to a singly-linked list.

1) Appending node D to charList updates which node's next pointer?



- M
- T
- E

2) Appending a node to an empty list updates which of the list's pointers?

- Head and tail
- Head only
- Tail only

3) Which statement is NOT executed when node 70 is appended to ticketList?

- `list→head = newNode`
- `list→tail→next = newNode`
- `list→tail = newNode`

PARTICIPATION ACTIVITY 20.2.8: Appending a node vs appending a list item.

Refer to the ListAppend() and ListAppendNode() functions in the animation above.

1) Both a list ADT and a singly-linked list have an item-appending function.

- True
- False

2) Both a list ADT and a singly-linked list have a node-appending function.

- True
- False

3) A list ADT implementation that uses a singly-linked list will have some functions that operate on list items and others that operate on singly-linked list nodes.

- True
- False

Prepend operation

Given a new node, the ***prepend*** operation for a singly-linked list inserts the new node before the list's head node. The prepend algorithm behavior differs if the list is empty versus not empty:

- *Prepend to empty list:* If the list's head pointer is null, the list's head and tail pointers are assigned with the new node.
- *Prepend to non-empty list:* If the list's head pointer is not null, the new node's next pointer is first assigned with the list's head node, then the list's head pointer is assigned with the new node.

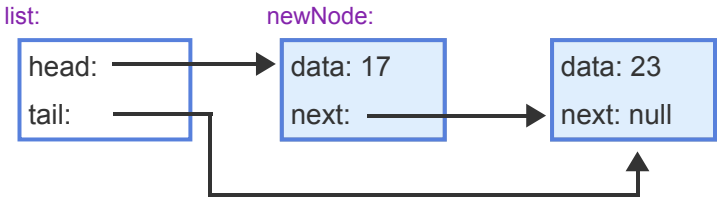
PARTICIPATION ACTIVITY 20.2.9: Singly-linked list: Prepend operation.

Start 2x speed

```
ListPrepend(list, item) {
   newNode = Allocate new node with item as data
   ListPrependNode(list, newNode)
}

ListPrependNode(list, newNode) {
   if (list→head == null) {
      list→head = newNode
      list→tail = newNode
   }
   else {
      newNode→next = list→head
      list→head = newNode
```

```
ListPrepend(list, 23) ✓
ListPrepend(list, 17) ✓
```

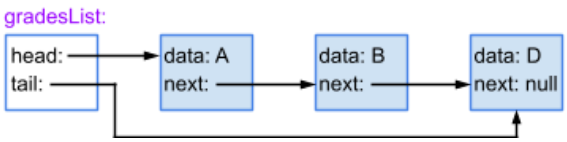
```
    }
}
```



Captions

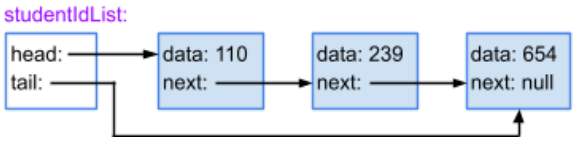
PARTICIPATION ACTIVITY 20.2.10: Prepending a node in a singly-linked list.

1) Prepending C to gradesList updates which pointer?



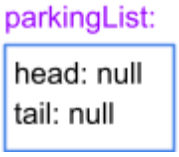
- The list's head pointer
- A's next pointer
- D's next pointer

2) Prepending node 789 to studentIdList updates the list's tail pointer.



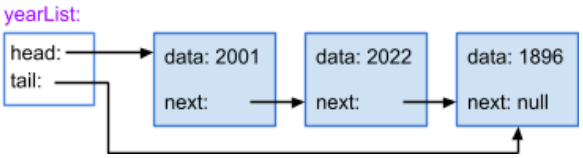
- True
- False

3) Prepending node 6 to parkingList updates the list's tail pointer.



- True
- False

4) Prepending node 1999 to yearList executes which statement?



- `list→head = null`
- `newNode→next = list→head`
- `list→tail = newNode`

CHALLENGE ACTIVITY 20.2.1: Singly-linked lists.

688294.4436218.qx3zqy7

Start

What is numList after the following operations?

```
numList = new List
ListAppend(numList, 63)
ListAppend(numList, 11)
ListAppend(numList, 61)
```

numList is now: Ex: 1, 2, 3 (comma between values)

1 2 3 4 5

Check Next

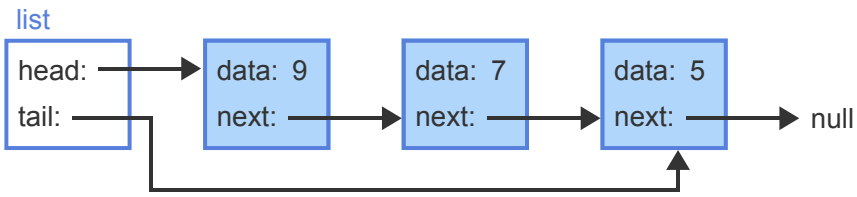
20.3 Singly-linked lists: Search and insert

Search operation

Given a data value, a linked list ***search*** operation returns the first node whose data equals that data value, or null if no such node exists. The search algorithm checks the current node (initially the list's head node), returning that node if a match, else assigning the current node with the next node and repeating. If the pointer to the current node is null, the algorithm returns null (matching node was not found).

PARTICIPATION ACTIVITY 20.3.1: Singly-linked list search operation.

Start 2x speed




```
ListSearch(list, key) {
   curNode = list→head
   while (curNode is not null) {
      if (curNode→data == key) {
         return curNode
      }
      curNode = curNode→next
   }
   return null
}
```

```
ListSearch(list, 5)    // Returns node 5
ListSearch(list, 11)   // Returns null
```

Captions

PARTICIPATION ACTIVITY 20.3.2: Search algorithm execution.

1) How many nodes are visited when searching for 54?

Check Show answer

2) How many nodes are visited when searching for 48?

Check Show answer

3) What is returned if the search key is not found?

Check Show answer

PARTICIPATION ACTIVITY 20.3.3: Searching a linked-list.

1) `ListSearch(scoreList, 22)` first assigns currentNode with ____.

- Node 99
- Node 44
- Node 22

2) Which call(s) end up assigning currentNode with node 22's next pointer?

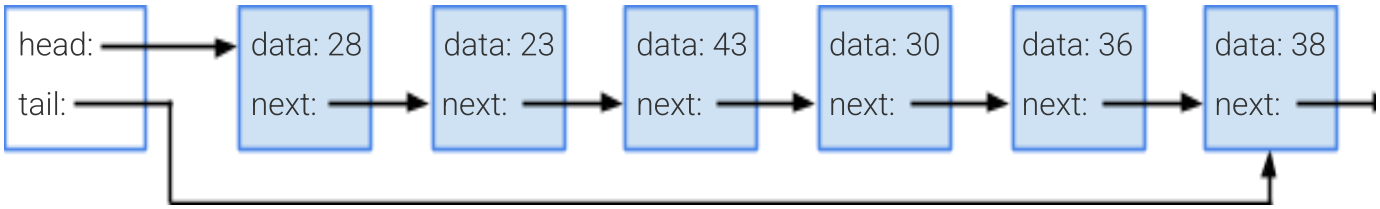
- `ListSearch(scoreList, 22)` only
- `ListSearch(scoreList, 33)` only
- `ListSearch(scoreList, 22)` and `ListSearch(scoreList, 33)`

CHALLENGE ACTIVITY 20.3.1: Linked list search.

688294.4436218.qx3zqy7

Start

numList:



`ListSearch(numList, 30)` points the current pointer to node [Ex: 9] after checking node 23.

`ListSearch(numList, 30)` will make [ ] comparisons.

1 2

Check Next

Insert-node-after operation

Given a new node, the ***insert-node-after*** operation for a singly-linked list inserts the new node after a provided existing list node. currentNode is a pointer to an existing list node but can be null when inserting into an empty list. The insert-node-after algorithm considers three insertion scenarios:

- *Insert as list's first node:* If the list's head pointer is null, the algorithm points the list's head and tail pointers to the new node.

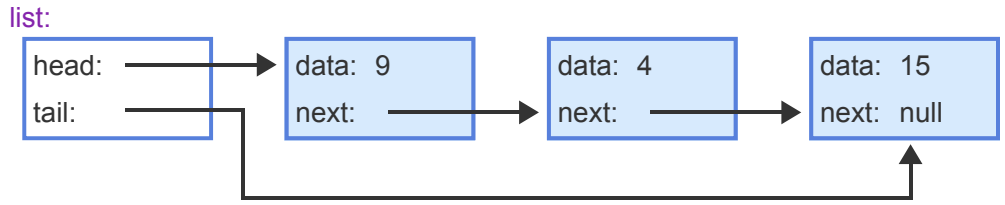
- *Insert after list's tail node:* If the list's head pointer is not null (list not empty) and currentNode points to the list's tail node, the algorithm points the tail node's next pointer and the list's tail pointer to the new node.
- *Insert in middle of list:* If the list's head pointer is not null (list not empty) and currentNode does not point to the list's tail node, the algorithm points the new node's next pointer to currentNode's next node, and then points currentNode's next to the new node.

PARTICIPATION ACTIVITY 20.3.4: Singly-linked list insert-node-after operation.

Start 2x speed

```
ListInsertNodeAfter(list, currentNode, newNode) {
   // Special case for empty list
   if (list→head == null) {
      list→head = newNode
      list→tail = newNode
   }
   else if (currentNode == list→tail) {
      list→tail→next = newNode
      list→tail = newNode
   }
   else {
      newNode→next = currentNode→next
      currentNode→next = newNode
   }
}
```

```
ListInsertNodeAfter(list, null, node9)   ✓
ListInsertNodeAfter(list, node9, node15) ✓
ListInsertNodeAfter(list, node9, node4)  ✓
```



Captions

PARTICIPATION ACTIVITY 20.3.5: Inserting nodes in a singly-linked list.

Select the list after the given operation(s).

1) numsList: (5, 9)

ListInsertNodeAfter(numsList, node9, node4)

- (4, 5, 9)
- (5, 4, 9)
- (5, 9, 4)

2) numsList: (23, 17, 8)

ListInsertNodeAfter(numsList, node23, node5)

- (5, 23, 17, 8)
- (23, 5, 17, 8)
- (5, 8, 17, 23)

3) numsList: (1)

ListInsertNodeAfter(numsList, node1, node6)
ListInsertNodeAfter(numsList, node1, node4)

- (1, 4, 6)
- (1, 6, 4)

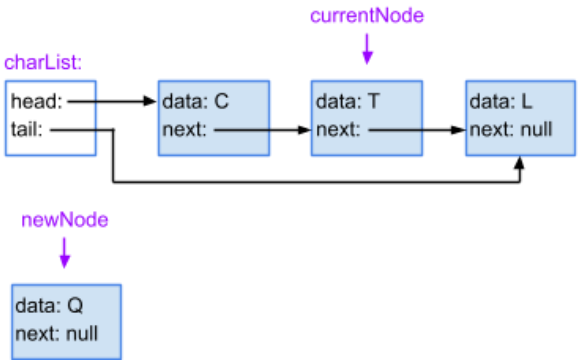
4) numsList: (77)

ListInsertNodeAfter(numsList, node77, node32)
ListInsertNodeAfter(numsList, node32, node50)
ListInsertNodeAfter(numsList, node32, node46)

- (77, 32, 46, 50)
- (77, 32, 50, 46)
- (32, 46, 50, 77)

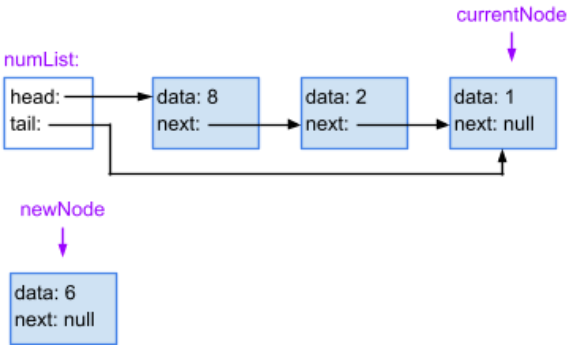
PARTICIPATION ACTIVITY 20.3.6: Singly-linked list insert-after algorithm.

1) `ListInsertNodeAfter(charList, nodeT, nodeQ)` assigns newNode's next pointer with _____.



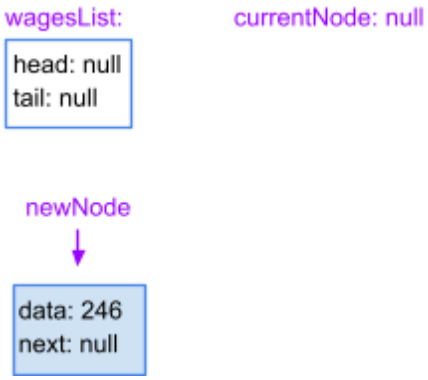
- currentNode⇢next
- charList's head node
- null

2) `ListInsertNodeAfter(numList, node1, node6)` executes which statement?



- `list⇢head = newNode`
- `newNode⇢next = currentNode⇢next`
- `list⇢tail⇢next = newNode`

3) `ListInsertNodeAfter(wagesList, null, node246)` executes which statement?



- `list→head = newNode`
- `list→tail→next = newNode`
- `currentNode→next = newNode`

CHALLENGE ACTIVITY 20.3.2: Singly-linked lists: Insert.

688294.4436218.qx3zqy7

Start

Given numList: (76, 15)

What is numList after the following operations?

```
ListInsertNodeAfter(numList, node76, node86)
ListInsertNodeAfter(numList, node86, node37)
ListInsertNodeAfter(numList, node86, node94)
```

numList is now: Ex: 1, 2, 3 (comma between values)

1 2 3 4

Check Next

Insert-after operation

A list ADT's insert-after operation has parameters for list *items*, not *nodes*. So when using a singly-linked list to implement a list ADT, a `ListInsertAfter(list, currentItem, newItem)` function is implemented. The function calls ListSearch() to find the node containing the current item. If found, a new node is allocated for the new item, and ListInsertNodeAfter() is called to insert the new node after the current node.

Figure 20.3.1: Singly-linked list ListInsertAfter() function.

```
ListInsertAfter(list, currentItem, newItem) {
   currentNode = ListSearch(list, currentItem)
   if (currentNode != null) {
      newNode = Allocate new singly-linked list node
      newNode→data = newItem
      newNode→next = null
      ListInsertNodeAfter(list, currentNode, newNode)
      return true
   }
   return false
}
```

PARTICIPATION ACTIVITY 20.3.7: Singly-linked list insert-after operation.

1) What happens if `ListInsertAfter(list, 22, 44)` is called for the list (11, 33, 55)?

   - An error occurs
   - ListInsertAfter() returns false, and the list is not changed
   - 44 is inserted after 33, yielding (11, 33, 44, 55)

2) ListInsertAfter() ____ calls ListSearch() and ____ calls ListInsertNodeAfter().

   - always, sometimes
   - sometimes, always
   - always, never

3) What is ListInsertAfter()'s *worst* case runtime complexity?

- $O(1)$
- $O(\log N)$
- $O(N)$

20.4 Singly-linked lists: Remove

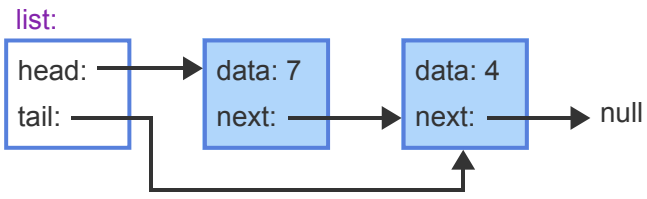
Remove-node-after operation

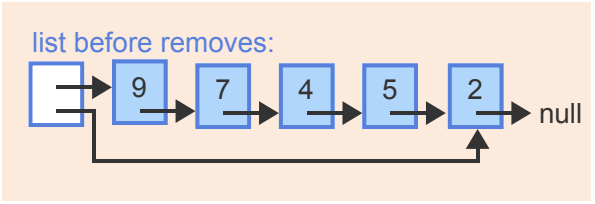
Given a specified existing node in a singly-linked list, the ***remove-node-after*** operation removes the node after the specified list node. The existing node must be specified because each node in a singly-linked list only maintains a pointer to the next node.

The existing node is specified with the currentNode parameter. If currentNode is null, the operation removes the list's first node. Otherwise, the node after currentNode is removed.

PARTICIPATION ACTIVITY 20.4.1: Singly-linked list remove-node-after operation.

Start 2x speed






```
ListRemoveNodeAfter(list, curNode) {
   // Special case, remove head
   if (curNode is null) {
      sucNode = list→head→next
      list→head = sucNode

      if (sucNode is null) { // Removed last item
         list→tail = null
      }
   }
   else if (curNode→next is not null) {
      sucNode = curNode→next→next
```

ListRemoveNodeAfter(list, null)

ListRemoveNodeAfter(list, node 4)

ListRemoveNodeAfter(list, node 4)

```
      curNode→next = sucNode

      if (sucNode is null) { // Removed tail
         list→tail = curNode
      }
   }
}
```

Captions

PARTICIPATION ACTIVITY 20.4.2: Removing nodes from a singly-linked list.

Type the list after the given operations. Type the list as: 4, 19, 3

1) numsList: (2, 5, 9)

ListRemoveNodeAfter(numsList, node 5)

numsList:

Check Show answer

2) numsList: (3, 57, 28, 40)

ListRemoveNodeAfter(numsList, null)

numsList:

Check Show answer

3) numsList: (86, 99, 46)

ListRemoveNodeAfter(numsList, node46)

numsList:

Check Show answer

4) numsList: (10, 20, 30, 40, 50, 60)

ListRemoveNodeAfter(numsList, node 40)
ListRemoveNodeAfter(numsList, node 20)

`numsList:`

Check Show answer

5) numsList: (91, 80, 77, 60, 75)

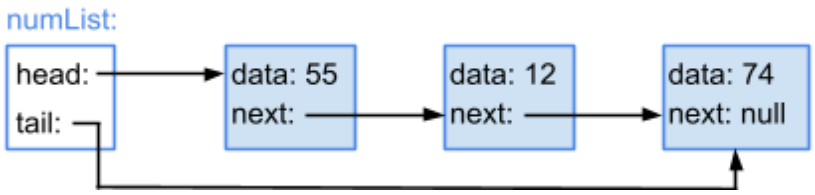
ListRemoveNodeAfter(numsList, node 60)
ListRemoveNodeAfter(numsList, node 77)
ListRemoveNodeAfter(numsList, null)

`numsList:`

Check Show answer

PARTICIPATION ACTIVITY 20.4.3: Remove-node-after algorithm execution: Intermediate node.

Given numList, ListRemoveNodeAfter(numList, node 55) executes which of the following statements?



1) `sucNode = list→head→next`

- Yes
- No

2) `curNode→next = sucNode`

- Yes
- No

3) `list→head = sucNode`

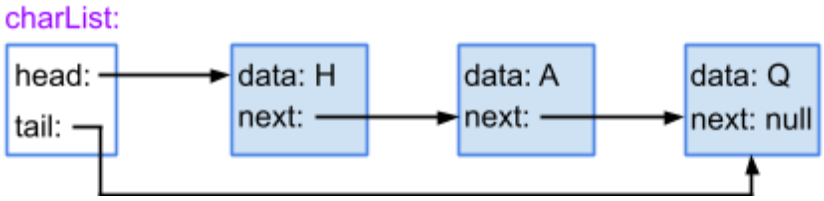
- Yes
- No

4) `list→tail = curNode`

- Yes
- No

PARTICIPATION ACTIVITY 20.4.4: Remove-node-after algorithm execution: List head node.

Given charList, ListRemoveNodeAfter(charList, null) executes which of the following statements?



1) `sucNode = list→head→next`

- Yes
- No

2) `curNode→next = sucNode`

- Yes
- No

3) `list→head = sucNode`

- Yes
- No

4) `list→tail = curNode`

- Yes
- No

PARTICIPATION ACTIVITY 20.4.5: Removing from an empty list.

1) If `numList` is an empty list, calling `ListRemoveNodeAfter(numList, null)` causes an error.

- True
- False

2) ListRemoveNodeAfter() should not be called on an empty list.

- True
- False

CHALLENGE ACTIVITY 20.4.1: Singly-linked lists: Remove.

688294.4436218.qx3zqy7

Start

Given numList: (4, 8, 5, 9, 3, 7)

What is numList after the following operations?

```
ListRemoveNodeAfter(numList, null)
ListRemoveNodeAfter(numList, node5)
ListRemoveNodeAfter(numList, node3)
```

List items in order, from head to tail.

Ex: 25, 42, 12

1 2 3 4 5

Check Next

Remove-item operation

A list ADT's remove operation has a parameter for a list *item*, not a *node*. So when using a singly-linked list to implement a list ADT, a `ListRemove(list, item)` function is implemented. The function finds the node containing the item to remove, keeping track of the previous node in the process. ListRemove() then calls ListRemoveNodeAfter(), passing the previous node as the argument.

Figure 20.4.1: ListRemove() function.

```
ListRemove(list, itemToRemove) {
   // Traverse to the node with data equal to itemToRemove,
   // keeping track of the previous node in the process
   previous = null
   current = list→head
   while (current != null) {
      if (current→data == itemToRemove) {
         ListRemoveNodeAfter(list, previous)
         return true
      }

      // Advance to next node
      previous = current
      current = current→next
   }

   // Not found
   return false
}
```

PARTICIPATION ACTIVITY 20.4.6: Remove item operation.

1) ListRemove() first initializes the previous variable with _____ and the current variable with _____.

   - the list's head, the list's tail
   - the list's head, null
   - null, the list's head

2) What happens if `ListRemove(list, 22)` is called for the list (11, 33, 55)?

   - An error occurs
   - Nothing changes and false is returned
   - Item 55 is removed and true is returned

3) What is the worst case time complexity of the remove-item operation?

- $O(1)$
- $O(\log N)$
- $O(N)$

20.5 Java: Singly-linked lists

SinglyLinkedNode and SinglyLinkedList classes

In Java, classes are used for both the linked list and the nodes that comprise the list. Each class has references to nodes (next node for the SinglyLinkedNode class and head and tail nodes for the SinglyLinkedList class).

The SinglyLinkedNode class implements a list node with two fields: a data value and the next node in the list. If the node has no next node, the next field is null.

The SinglyLinkedList class implements the list data structure and has two private fields, head and tail, which are assigned to nodes once the list is populated. Initially the list has no nodes, so head and tail are both initially null.

Figure 20.5.1: SinglyLinkedNode class definition.

```
class SinglyLinkedNode {
   public int data;
   public SinglyLinkedNode next;

   public SinglyLinkedNode(int data) {
      this.data = data;
      this.next = null;
   }
}
```

Figure 20.5.2: SinglyLinkedList class fields and constructor.

```
class SinglyLinkedList {
   private SinglyLinkedNode head;
   private SinglyLinkedNode tail;

   public SinglyLinkedList() {
```

```
        head = null;
        tail = null;
    }
};
```

PARTICIPATION ACTIVITY 20.5.1: SinglyLinkedNode and SinglyLinkedList classes.

1) The SinglyLinkedNode class has two member variables.
   - True
   - False

2) Each node's data is initialized to 0 in the SinglyLinkedNode class constructor.
   - True
   - False

3) An empty SinglyLinkedList has a single node with no data.
   - True
   - False

4) The SinglyLinkedList class's head and tail node fields are public.
   - True
   - False

Appending a node to a singly-linked list

The SinglyLinkedList class's appendNode() method adds a node to the end of the linked list, making the node the tail of the list. appendNode() has one parameter, which is the new node to be appended to the list.

The SinglyLinkedList class also has an append() method that takes an integer list item as an argument. append() allocates a new SinglyLinkedNode to hold the item, then calls appendNode() to append that node.

Figure 20.5.3: SinglyLinkedList append() and appendNode() methods.

```
public void append(int item) {
    appendNode(new SinglyLinkedNode(item));
}

public void appendNode(SinglyLinkedNode newNode) {
```

```
   if (head == null) {
      head = newNode;
      tail = newNode;
   }
   else {
      tail.next = newNode;
      tail = newNode;
   }
}
```

PARTICIPATION ACTIVITY 20.5.2: SinglyLinkedList append methods.

1) An appended node always becomes the _____ of a linked list.

   - head
   - tail

2) Which statements append a node with data value 15 to numList?

   - `append(numList, 15);`
   - `numList.append(15);`
   - `nodeA.append(new SinglyLinkedNode(15));`

Additional singly-linked list methods

The following methods are implemented in the SinglyLinkedList class.

The prependNode() method adds a node to the beginning of a linked list, making the node the head of the list. The method's parameter is the new node to be prepended to the list.

The prepend() method takes an integer list item as an argument, allocates a new SinglyLinkedNode to hold the item, then calls prependNode() to prepend that node.

Figure 20.5.4: SinglyLinkedList prependNode() and prepend() methods.

```
public void prependNode(SinglyLinkedNode newNode) {
   if (head == null) {
      head = newNode;
      tail = newNode;
   }
   else {
      newNode.next = head;
      head = newNode;
   }
```

```
}

public void prepend(int item) {
   prependNode(new SinglyLinkedNode(item));
}
```

The insertNodeAfter() method adds a node after an existing node in the list. The method has two parameters: the existing node (currentNode) and the new node to be inserted (newNode). Three possible cases exist:

- If the list's head is null, then the list is empty and the node is inserted as the only node in the list.
- If currentNode is the same node object as the list's tail, then the node is inserted at the end of the list.
- If neither of the above are true, then the new node's next value is assigned with the existing node's next value, and the existing node's next value is assigned with the new node.

The insertAfter() method also exists and takes integer arguments for the existing and new items.

Figure 20.5.5: SinglyLinkedList insertNodeAfter() and insertAfter() methods.

```
public void insertNodeAfter(SinglyLinkedNode currentNode, SinglyLinkedNode newNode) {
   if (head == null) {
      head = newNode;
      tail = newNode;
   }
   else if (currentNode == tail) {
      tail.next = newNode;
      tail = newNode;
   }
   else {
      newNode.next = currentNode.next;
      currentNode.next = newNode;
   }
}

public boolean insertAfter(int currentItem, int newItem) {
   SinglyLinkedNode currentNode = search(currentItem);
   if (currentNode != null) {
      SinglyLinkedNode newNode = new SinglyLinkedNode(newItem);
      insertNodeAfter(currentNode, newNode);
      return true;
   }
   return false; // currentItem not found
}
```


The removeNodeAfter() method has a currentNode parameter and removes currentNode's successor (the node after currentNode) from the list. If a successor exists, currentNode's next value is assigned with the successor node's next value, thus removing the successor node from the list.

removeNodeAfter() can also be used to remove the list's head node by passing null as the currentNode argument.

The remove() method also exists and takes an integer argument for the item to remove. remove() first searches for a node with data equal to the argument. The search keeps track of the node previous to the node with data equal to the argument. So if found, the node is removed via a call to removeNodeAfter().

Figure 20.5.6: SinglyLinkedList removeNodeAfter() and remove() methods.


```
public void removeNodeAfter(SinglyLinkedNode currentNode) {
   if (currentNode == null && head != null) {
      // Special case: remove head
      SinglyLinkedNode succeedingNode = head.next;
      head = succeedingNode;
      if (succeedingNode == null) {
         // Last item was removed
         tail = null;
      }
   }
   else if (currentNode.next != null) {
      SinglyLinkedNode succeedingNode = currentNode.next.next;
      currentNode.next = succeedingNode;
      if (succeedingNode == null) {
         // Remove tail
         tail = currentNode;
      }
   }
}

public boolean remove(int item) {
   // Traverse to the node with data equal to item,
   // keeping track of the previous node in the process
   SinglyLinkedNode previous = null;
   SinglyLinkedNode current = head;
   while (current != null) {
      if (current.data == item) {
         removeNodeAfter(previous);
         return true;
      }

      // Advance to next node
      previous = current;
      current = current.next;
   }

   // Not found
   return false;
}
```


PARTICIPATION ACTIVITY 20.5.3: Additional SinglyLinkedList operations.

1) Which is a parameter of prependNode()?

- newNode
- currentNode

2) In insertNodeAfter(), what does the variable currentNode signify?

- The node to be removed.
- The node to be added.
- The node before the node to be added.

3) removeNodeAfter() removes the list's head node if the currentNode argument is ____.

- the list's head node
- the list's tail node
- null

4) What are the parameter types for the prepend(), insertAfter(), and remove() methods?

- SinglyLinkedNode
- int

zyDE 20.5.1: Singly-linked list.

The following code implements several common list ADT operations using a singly-linked list:

- **ListADT.java** - The ListADT interface
- **SinglyLinkedList.java** - The SinglyLinkedNode class and the SinglyLinkedList class that implements ListADT
- **Program.java** - The Program class and main() method

Current file: Program.java Load default template...

```
1 public class Program {
2    public static void main(String[] args) {
3       SinglyLinkedList numList = new SinglyLinkedList();
4
5       // Insert various items using append(), prepend(), and insertAfter()
6       numList.append(14);          // List: 14
7       numList.append(2);           // List: 14 2
```

```
8       numList.append(20);          // List: 14, 2, 20
9       numList.prepend(31);         // List: 31, 14, 2, 20
10      numList.insertAfter(2, 16);  // List: 31, 14, 2, 16, 20
11      numList.insertAfter(20, 55); // List: 31, 14, 2, 16, 20, 55
12
13      // Output list
14      System.out.print("List after adding items: ");
15      numList.print(System.out);
```

Run

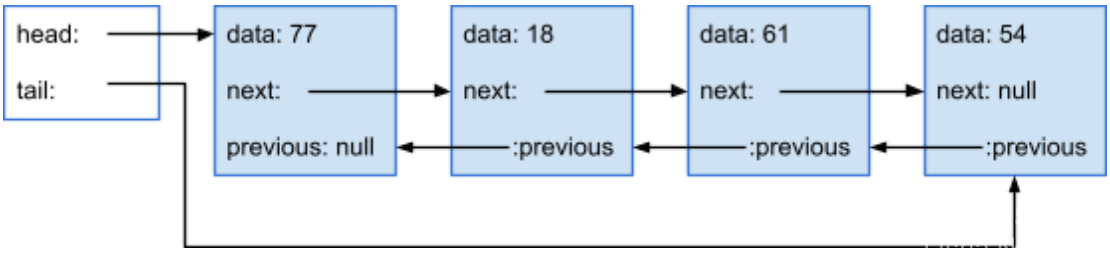
20.6 Doubly-linked lists

Doubly-linked list

A ***doubly-linked list*** is a data structure for implementing a list ADT, where each node has data, a pointer to the next node, and a pointer to the previous node. The list structure typically has pointers to the first node and the last node. The doubly-linked list's first node is called the head, and the last node the tail.

A doubly-linked list is similar to a singly-linked list, but instead of using a single pointer to the next node in the list, each node has a pointer to the next and previous nodes. Such a list is called "doubly-linked" because each node has two pointers, or "links". A doubly-linked list is a type of ***positional list***: A list where elements contain pointers to the next and/or previous elements in the list.

Figure 20.6.1: Doubly-linked list with items: 77, 18, 61, 54.



PARTICIPATION ACTIVITY 20.6.1: Doubly-linked list data structure.

1) Each node in a doubly-linked list contains data and _____ pointer(s).

- one
- two

2) Given a doubly-linked list with nodes 20, 67, 11, node 20 is the _____.

- head
- tail

3) Given a doubly-linked list with nodes 4, 7, 5, 1, node 7's previous pointer points to node _____.

- 4
- 5

4) Given a doubly-linked list with nodes 8, 12, 7, 3, node 7's next pointer points to node _____.

- 12
- 3

Append operation

Given a new node, the ***append*** operation for a doubly-linked list inserts the new node after the list's tail node. The append algorithm's behavior differs if the list is empty versus not empty:

- *Append to empty list:* If the list's head pointer is null, the algorithm points the list's head and tail pointers to the new node.
- *Append to non-empty list:* If the list's head pointer is not null, the algorithm points the tail node's next pointer to the new node, points the new node's previous pointer to the list's tail node, and points the list's tail pointer to the new node.

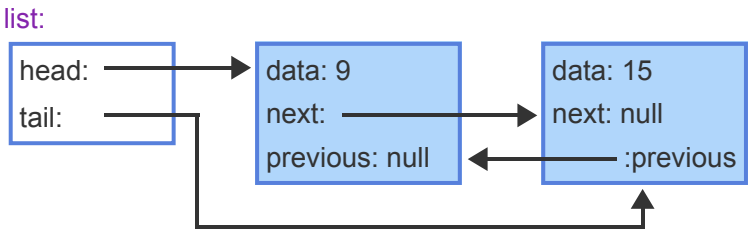
PARTICIPATION ACTIVITY 20.6.2: Doubly-linked list: Append operation.

Start 2x speed

```
ListAppend(list, item) {
   newNode = Allocate new node with item as data
   ListAppendNode(list, newNode)
}
```

```
ListAppend(list, 9)

ListAppend(list, 15)
```

```
ListAppendNode(list, newNode) {
   if (list→head == null) { // List empty
      list→head = newNode
      list→tail = newNode
   }
   else {
      list→tail→next = newNode
      newNode→prev = list→tail
      list→tail = newNode
   }
}
```



Captions

PARTICIPATION ACTIVITY 20.6.3: Doubly-linked list: Append operation.

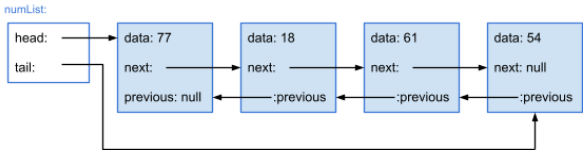
1) A new node is allocated by which function(s)?

- ListAppend() only
- ListAppendNode() only
- Both ListAppend() and ListAppendNode()

2) ListAppendNode() reassigns the list's head pointer _____.

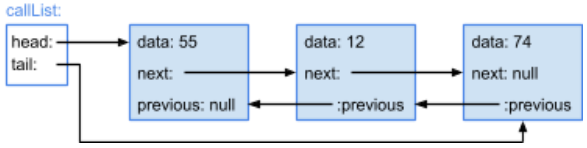
- only when the list is empty
- only when the list is not empty
- in all cases

3) `ListAppend(numList, 88)` inserts a node with data 88 _____.



- after node 77
- before node 54
- after node 54

4) `ListAppend(callList, 5)` executes which statement?



- `list→head = newNode`
- `list→tail→next = newNode`
- `newNode→next = list→tail`

Prepend operation

Given a new node, the ***prepend*** operation of a doubly-linked list inserts the new node before the list's head node and points the head pointer to the new node.

- *Prepend to empty list:* If the list's head pointer is null, the algorithm points the list's head and tail pointers to the new node.
- *Prepend to non-empty list:* If the list's head pointer is not null, the algorithm points the new node's next pointer to the list's head node, points the list head node's previous pointer to the new node, and then points the list's head pointer to the new node.

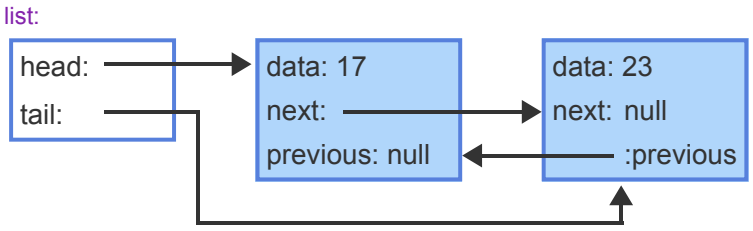
PARTICIPATION ACTIVITY 20.6.4: Doubly-linked list: Prepend operation.

Start 2x speed

```
ListPrepend(list, item) {
   newNode = Allocate new node with item as data
   ListPrependNode(list, newNode)
}

ListPrependNode(list, newNode) {
   if (list→head == null) {
      list→head = newNode
      list→tail = newNode
```

```
ListPrepend(list, 23)

ListPrepend(list, 17)
```

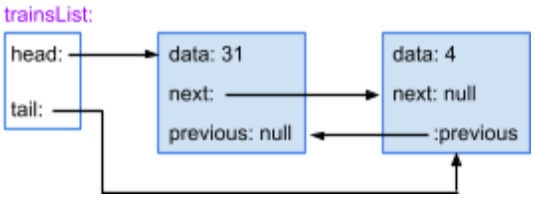
```
    }
    else {
       newNode→next = list→head
       list→head→prev = newNode
       list→head = newNode
    }
}
```



Captions

PARTICIPATION ACTIVITY 20.6.5: Prepending a node in a doubly-linked list.

1) Prepending 29 to trainsList reassigns the list's head with node ____.

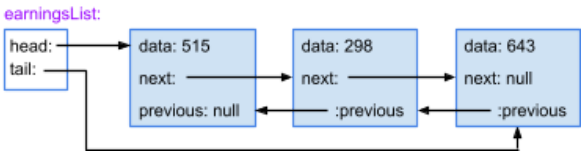


- 4
- 29
- 31

2) `ListPrepend(shoppingList, 86)` updates the list's tail pointer.

- True
- False

3) `ListPrepend(earningsList, 977)` executes which statement?



- `list→tail = newNode`
- `newNode→next = list→head`
- `newNode→next = list→tail`

CHALLENGE ACTIVITY 20.6.1: Doubly-linked lists.

688294.4436218.qx3zqy7

Start

```
numList = new List
ListAppend(numList, 72)
ListAppend(numList, 92)
ListAppend(numList, 39)
```

numList is now: Ex: 1, 2, 3 (comma between values)

Which node has a null previous pointer? Ex: 5

Which node has a null next pointer? Ex: 5

1 2 3 4 5

Check Next

20.7 Doubly-linked lists: Search and insert

Search operation

Given a data value, a linked list ***search*** operation returns the first node whose data equals that data value, or null if no such node exists. The search algorithm checks the current node (initially the list's head node), returning that node if a match. Otherwise, the current node moves to the next node, and the search continues. If the current node is null, the algorithm returns null (matching node was not found).

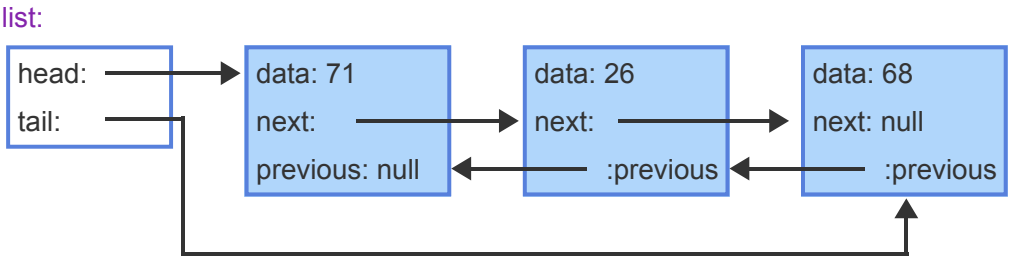
PARTICIPATION ACTIVITY 20.7.1: Doubly-linked list search operation.

Start 2x speed

```
ListSearch(list, dataValue) {
   currentNode = list→head
   while (currentNode != null) {
      if (currentNode→data == dataValue) {
         return currentNode
      }
      currentNode = currentNode→next
   }

   return null
}
```

`ListSearch(list, 68)` Returns node 68

`ListSearch(list, 54)` Returns null



Captions

PARTICIPATION ACTIVITY 20.7.2: Doubly-linked list search operation.

1) When searching numsList, what does ListSearch() initially assign currentNode with?

- null
- Node 8
- Depends on the dataValue parameter

2) Which call returns null?

- `ListSearch(numsList, 7)`
- `ListSearch(numsList, 8)`
- `ListSearch(numsList, 9)`

3) Which call performs the most operations?

- `ListSearch(numsList, 7)`
- `ListSearch(numsList, 8)`
- `ListSearch(numsList, 9)`

Insert-node-after operation

Given a new node, the ***insert-node-after*** operation for a doubly-linked list inserts the new node after a provided existing list node. The insert-node-after algorithm considers three insertion scenarios: insert into an empty list, insert after the list's tail node, or insert between two existing list nodes.

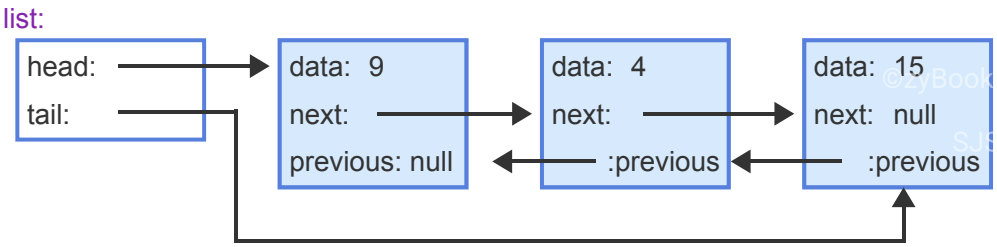
PARTICIPATION ACTIVITY 20.7.3: Doubly-linked list insert-after operation.

Start 2x speed

```
ListInsertNodeAfter(list, currentNode, newNode) {
   // Special case if list is empty
   if (list→head == null) {
      list→head = newNode
      list→tail = newNode
   }
   else if (currentNode == list→tail) {
      list→tail→next = newNode
      newNode→previous = list→tail
      list→tail = newNode
   }
   else {
      successor = currentNode→next
      newNode→next = successor
```

```
ListInsertNodeAfter(list, null, node9)   ✓
ListInsertNodeAfter(list, node9, node15) ✓
ListInsertNodeAfter(list, node9, node4)  ✓
```

```
        newNode→previous = currentNode
        currentNode→next = newNode
        successor→previous = newNode
    }
}
```



Captions

PARTICIPATION ACTIVITY 20.7.4: Inserting nodes in a doubly-linked list.

Given weeklySalesList: 12, 30
What is the node order after the following operations?

ListInsertNodeAfter(weeklySalesList, node30, node8)
ListInsertNodeAfter(weeklySalesList, node12, node45)
ListInsertNodeAfter(weeklySalesList, node45, node76)

How to use this tool

Node 8 **Node 45** **Node 76** **Node 12** **Node 30**

	Position 0 (list's head node)
	Position 1
	Position 2
	Position 3
	Position 4 (list's tail node)

Reset

CHALLENGE ACTIVITY 20.7.1: Doubly-linked lists: Insert.

688294.4436218.qx3zqy7

Start

Given numList: (99, 10)

What is numList after the following operations?

```
ListInsertNodeAfter(numList, node10, node30)
ListInsertNodeAfter(numList, node30, node79)
ListInsertNodeAfter(numList, node99, node35)
ListInsertNodeAfter(numList, node99, node84)
```

numList is now: Ex: 1, 2, 3 (comma between values)

What node does node 35's next pointer point to? Select

What node does node 35's previous pointer point to? Select

1 2 3 4

Check Next

Insert-after operation

A list ADT's insert-after operation has parameters for list *items*, not *nodes*. So when using a doubly-linked list to implement a list ADT, an `ListInsertAfter(list, currentItem, newItem)` function is implemented. The currentItem and newItem parameters are list data items, not linked list nodes. ListInsertAfter() calls ListSearch() to find the node containing the current item. If found, a new node is allocated for the new item, and ListInsertNodeAfter() is called to insert the new node after the current node.

Figure 20.7.1: ListInsertAfter() function.

```
ListInsertAfter(list, currentItem, newItem) {
   currentNode = ListSearch(list, currentItem)
   if (currentNode != null) {
      newNode = Allocate new doubly-linked node with newItem as data
      ListInsertNodeAfter(list, currentNode, newNode)
      return true
   }
```

```
   return false // currentItem not found
}
```

PARTICIPATION ACTIVITY 20.7.5: Singly-linked list insert-item-after operation.

1) Given integersList: (11, 33, 55). `ListInsertAfter(integersList, 22, 11)` yields the list (11, 22, 33, 55).
   - True
   - False

2) What list results from calling `ListInsertAfter(list, 10, 99)` for the list (20, 10, 40, 10)?
   - (20, 10, 40, 10)
   - (20, 10, 99, 40, 10)
   - (20, 10, 40, 10, 99)

3) What is ListInsertAfter()'s *best* case runtime complexity?
   - $O(1)$
   - $O(\log N)$
   - $O(N)$

20.8 Doubly-linked lists: Remove

Remove-node operation

The ***remove-node*** operation for a doubly-linked list removes a provided existing list node. currentNode is a pointer to an existing list node. The algorithm first determines the node's successor (the next node) and predecessor (the previous node). The algorithm uses four separate checks to update each pointer:

- *Successor exists:* If the successor is not null (successor exists), the algorithm points the successor's previous pointer to the predecessor node.

- *Predecessor exists:* If the predecessor is not null (predecessor exists), the algorithm points the predecessor's next pointer to the successor node.
- *Removing list's head node:* If currentNode points to the list's head node, the algorithm points the list's head pointer to the successor node.
- *Removing list's tail node:* If currentNode points to the list's tail node, the algorithm points the list's tail pointer to the predecessor node.

PARTICIPATION ACTIVITY 20.8.1: Doubly-linked list: Node removal.

Start 2x speed

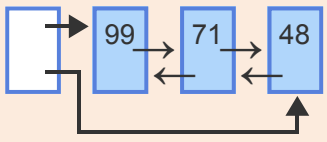
```
ListRemoveNode(list, currentNode) {
   successor = currentNode→next
   predecessor = currentNode→previous

   if (successor != null) {
      successor→previous = predecessor
   }
   if (predecessor != null) {
      predecessor→next = successor
   }

   if (currentNode == list→head) {
      list→head = successor
   }
   if (currentNode == list→tail) {
      list→tail = predecessor
   }
}
```

```
ListRemoveNode(list, node71)

ListRemoveNode(list, node48)

ListRemoveNode(list, node99)
```

list:

head:	null
tail:	null

List before removals:



Captions

PARTICIPATION ACTIVITY 20.8.2: Removing nodes from a doubly-linked list.

Type numsList's content after the given operation(s). Type the list as: 4, 19, 3

1) numsList: 71, 29, 54

```
ListRemoveNode(numsList,
node29)
```

Check Show answer

2) numsList: 2, 8, 1

```
ListRemoveNode(numsList,
node1)
```

Check Show answer

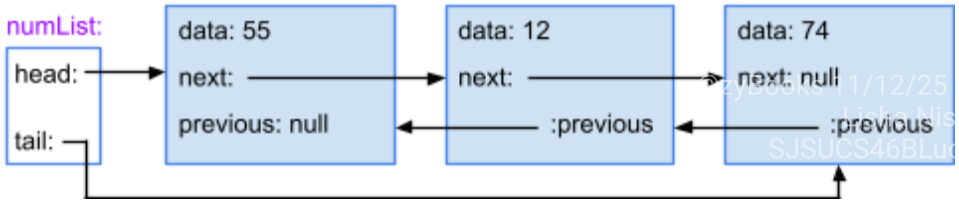
3) numsList: 70, 82, 41, 120, 357, 66

```
ListRemoveNode(numsList, node82)
ListRemoveNode(numsList,
node357)
ListRemoveNode(numsList, node66)
```

Check Show answer

PARTICIPATION ACTIVITY 20.8.3: Doubly-linked list remove algorithm execution: Intermediate node.

Given numList below, `ListRemoveNode(numList, node12)` executes which of the following statements?



1) `successor→previous = predecessor`

- Yes
- No

2) `predecessor→next = successor`

- Yes
- No

3) `list→head = successor`

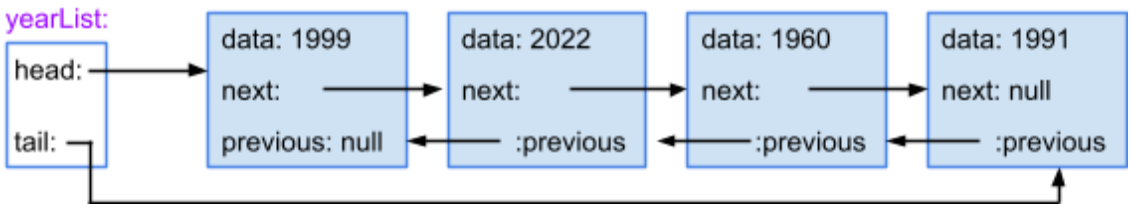
- Yes
- No

4) `list→tail = predecessor`

- Yes
- No

PARTICIPATION ACTIVITY 20.8.4: Doubly-linked list remove algorithm execution: List head node.

Given yearList, `ListRemoveNode(yearList, node1999)` executes which of the following statements?



1) `successor→previous = predecessor`

- Yes
- No

2) `predecessor→next = successor`

- Yes
- No

3) `list→head = successor`

- Yes
- No

4) `list→tail = predecessor`

- Yes
- No

CHALLENGE ACTIVITY 20.8.1: Doubly-linked lists: Remove.

688294.4436218.qx3zqy7

Start

Given numList: (1, 3, 9, 8, 2)

What is numList after the following operations?

```
ListRemoveNode(numList, node9)
ListRemoveNode(numList, node2)
ListRemoveNode(numList, node1)
```

numList is now: (Ex: 25, 42, 12)

1 2 3 4 5

Check Next

Remove-item operation

A list ADT's remove operation has a parameter for a list *item*, not a *node*. So when using a doubly-linked list to implement a list ADT, a `ListRemove(list, item)` function is implemented. The function first calls

ListSearch() to find the node containing the item to remove. If non-null, the node returned by ListSearch() is then passed to ListRemoveNode().

Figure 20.8.1: ListRemove() function.

```
ListRemove(list, itemToRemove) {
   nodeToRemove = ListSearch(list, itemToRemove)
   if (nodeToRemove != null) {
      ListRemoveNode(list, nodeToRemove)
      return true
   }

   return false // not found
}
```


PARTICIPATION ACTIVITY 20.8.5: Remove item operation.

1) ListRemove()'s implementation could be compacted into a single statement:

```
return ListRemoveNode(list,
ListSearch(itemToRemove))
```

- ○ True
- ○ False

2) What happens if `ListRemove(list, 22)` is called for the list (11, 33, 55)?

- ○ An error occurs
- ○ Nothing changes, and false is returned
- ○ Item 11 is removed, and true is returned

3) What is the worst case time complexity of the remove-item operation?

- ○ $O(1)$
- ○ $O(\log N)$
- ○ $O(N)$

20.9 Java: Doubly-linked lists

DoublyLinkedNode and DoublyLinkedList classes

In Java, classes are used for both the linked list and the nodes that comprise the list. Each class has references to nodes (next and previous nodes for the DoublyLinkedNode class and head and tail nodes for the DoublyLinkedList class).

The DoublyLinkedNode class implements a list node with three fields: a data value, the next node in the list, and the previous node in the list. If the node has no next node, the next field is null. If the node has no previous node, the previous field is null.

The DoublyLinkedList class implements the list data structure and has two private fields, head and tail, which are assigned to nodes once the list is populated. Initially the list has no nodes, so both are initially null.

Figure 20.9.1: DoublyLinkedNode class definition for a doubly-linked node.

```
class DoublyLinkedNode {
   public int data;
   public DoublyLinkedNode next;
   public DoublyLinkedNode previous;

   public DoublyLinkedNode(int data) {
      this.data = data;
      next = null;
      previous = null;
   }
}
```

Figure 20.9.2: DoublyLinkedList class definition for a doubly-linked list.

```
class DoublyLinkedList implements ListADT {
   private DoublyLinkedNode head;
   private DoublyLinkedNode tail;

   public DoublyLinkedList() {
      head = null;
      tail = null;
   }
}
```


PARTICIPATION ACTIVITY 20.9.1: Doubly-linked lists and nodes.

1) A doubly-linked node has both next and previous fields.

- True
- False

2) In a doubly-linked list, the first node's previous is null.

- True
- False

3) Each node's data is an integer.

- True
- False

Appending a node to a doubly-linked list

DoublyLinkedList's appendNode() method for doubly-linked nodes is similar to the appendNode() method for singly-linked nodes, but has an added line of code. Before the method assigns the list's tail with the new node, **`newNode.previous`** is assigned with the list's old tail.

The DoublyLinkedList class also has an append() method that takes an integer list item as an argument. append() allocates a new DoublyLinkedNode to hold the item, then calls appendNode() to append that node.

Figure 20.9.3: DoublyLinkedList appendNode() and append() methods.

```
public void appendNode(DoublyLinkedNode newNode) {
   if (head == null) {
      head = newNode;
      tail = newNode;
   }
   else {
      tail.next = newNode;
      newNode.previous = tail;
      tail = newNode;
   }
}

public void append(int item) {
   appendNode(new DoublyLinkedNode(item));
}
```


PARTICIPATION ACTIVITY 20.9.2: Appending to a doubly-linked list.

Assume the code below has been executed.

```
DoublyLinkedList numberList = new DoublyLinkedList();
numberList.append(72);
numberList.append(48);
numberList.append(91);
```

1) If node48 is the node with data 48 then `node48.previous` is ____.

- the node with data 72
- the node with data 48
- the node with data 91
- null

2) If node91 is the node with data 91 then `node91.next` is ____.

- the node with data 72
- the node with data 48
- null

3) Which node has previous assigned with null?

- Node 72
- Node 48
- Node 91

Prepending a node to a doubly-linked list

The prependNode() method inserts a new node at the head of a doubly-linked list, making the new node the head of the list. Additionally, the old head node's previous must be set to reference the new node.

The prepend() method takes an integer list item as an argument, allocates a new DoublyLinkedNode to hold the item, then calls prependNode() to prepend that node.

Figure 20.9.4: DoublyLinkedList prependNode() and prepend() methods.

```
public void prependNode(DoublyLinkedNode newNode) {
   if (head == null) {
      head = newNode;
      tail = newNode;
   }
   else {
      newNode.next = head;
      head.previous = newNode;
```

```
        head = newNode;
    }
}

public void prepend(int item) {
    prependNode(new DoublyLinkedNode(item));
}
```

PARTICIPATION ACTIVITY 20.9.3: prependNode() method.

How to use this tool

head | tail | newNode

	Changed only if the list is empty.
	`head.previous` is assigned with ____ if the list is not empty.
	Always assigned with newNode.

Reset

Insertion and removal methods

The insertNodeAfter() method has two node parameters:

1. currentNode is the node that already exists in the list and will reside immediately before the new node after insertion.
2. newNode is the new node to insert into the list.

insertNodeAfter() can affect currentNode's ***successor*** node: the node that comes immediately after currentNode in the list. Three nodes can be affected by the insertNodeAfter() method:

- The new node (newNode), which needs both next and previous fields to be assigned.
- The existing node (currentNode), whose next field must be assigned.
- The current node's successor (successor), whose previous field must be assigned.

In the general case, the new node's previous is assigned with currentNode, the new node's next is assigned with currentNode's successor, the current node's next is assigned with newNode, and the successor node's previous is assigned with newNode.

Two special cases also must be handled: when the list is currently empty (so that the new node will become the only node in the list), and when the current node is the tail of the list.

Figure 20.9.5: DoublyLinkedList insertNodeAfter() and insertAfter() methods.

```
public void insertNodeAfter(DoublyLinkedNode currentNode, DoublyLinkedNode newNode) {
   if (head == null) {
      head = newNode;
      tail = newNode;
   }
   else if (currentNode == tail) {
      tail.next = newNode;
      newNode.previous = tail;
      tail = newNode;
   }
   else {
      DoublyLinkedNode successor = currentNode.next;
      newNode.next = successor;
      newNode.previous = currentNode;
      currentNode.next = newNode;
      successor.previous = newNode;
   }
}

public boolean insertAfter(int currentItem, int newItem) {
   DoublyLinkedNode currentNode = search(currentItem);
   if (currentNode != null) {
      DoublyLinkedNode newNode = new DoublyLinkedNode(newItem);
      insertNodeAfter(currentNode, newNode);
      return true;
   }
   return false; // currentItem not found
}
```

The removeNode() method has one parameter, currentNode, which is the node to remove from the list. In addition to affecting the successor node, removal can affect currentNode's ***predecessor*** node: the node that comes immediately before currentNode in the list.

removeNode() unlinks currentNode from the list by joining together the node before the removed node (the predecessor node) and the node after the removed node (the successor node). If currentNode is either the head or tail of the list, the method also updates the list's head and tail fields.

The remove() method also exists and takes an integer argument for the item to remove.

Figure 20.9.6: DoublyLinkedList removeNode() and remove() methods.

```
public void removeNode(DoublyLinkedNode currentNode) {
   DoublyLinkedNode successor = currentNode.next;
   DoublyLinkedNode predecessor = currentNode.previous;

   if (successor != null) {
```

```
            successor.previous = predecessor;
        }
        if (predecessor != null) {
            predecessor.next = successor;
        }

        if (currentNode == head) {
            head = successor;
        }
        if (currentNode == tail) {
            tail = predecessor;
        }
    }

    public boolean remove(int itemToRemove) {
        DoublyLinkedNode nodeToRemove = search(itemToRemove);
        if (nodeToRemove != null) {
            removeNode(nodeToRemove);
            return true;
        }

        return false; // not found
    }
```


PARTICIPATION ACTIVITY 20.9.4: Additional DoublyLinkedList methods.

1) Which DoublyLinkedList method inserts a new node in the middle of a list?

Check Show answer

2) What is the final list after the following code is executed? Enter the list like: 7, 22, 3.

```
DoublyLinkedList numList = new
DoublyLinkedList();
numList.append(1);
numList.append(2);
numList.append(3);
numList.prepend(4);
numList.insertAfter(1, 5);
numList.remove(3);
```

Check Show answer

zyDE 20.9.1: Doubly-linked list.

The following code implements several common list ADT operations using a doubly-linked list:

- **ListADT.java** - The ListADT interface
- **DoublyLinkedList.java** - The DoublyLinkedNode class and the DoublyLinkedList class that implements ListADT
- **Program.java** - The Program class and main() method

Current file: Program.java Load default template...

```
1  public class Program {
2     public static void main(String[] args) {
3        DoublyLinkedList numList = new DoublyLinkedList();
4
5        // Insert various items using append(), prepend(), and insertAfter()
6        numList.append(14);          // List: 14
7        numList.append(2);           // List: 14, 2
8        numList.append(20);          // List: 14, 2, 20
9        numList.prepend(31);         // List: 31, 14, 2, 20
10       numList.insertAfter(2, 16);  // List: 31, 14, 2, 16, 20
11       numList.insertAfter(20, 55); // List: 31, 14, 2, 16, 20, 55
12
13       // Output list
14       System.out.print("List after adding items: ");
15       numList.print(System.out);
```

Run

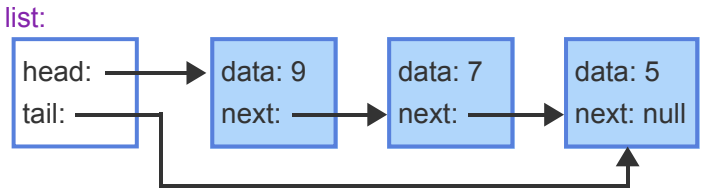
20.10 Linked list traversal

Linked list traversal

A ***list traversal*** algorithm visits all nodes in the list once and performs an operation on each node. A common traversal operation prints all list nodes. The algorithm starts by pointing a curNode pointer to the list's head node. While curNode is not null, the algorithm prints the current node, and then points curNode to the next node. After the list's tail node is visited, curNode is pointed to the tail node's next node, which does not exist. So, curNode is null, and the traversal ends. The traversal algorithm supports both singly-linked and doubly-linked lists.

PARTICIPATION ACTIVITY 20.10.1: Singly-linked list: List traversal.

Start 2x speed




```
ListTraverse(list) {
   curNode = list→head // Start at head

   while (curNode is not null) {
      Print curNode's data
      curNode = curNode→next
   }
}
```

```
9 7 5
```

Captions

PARTICIPATION ACTIVITY 20.10.2: List traversal.

1) ListTraverse() starts at _____.

- a specified list node
- the list's head node
- the list's tail node

2) Given numsList is: (5, 8, 2, 1). `ListTraverse(numsList)` visits _____ node(s).

- one
- two
- four

3) ListTraverse() can be used to traverse a doubly-linked list.

- True
- False

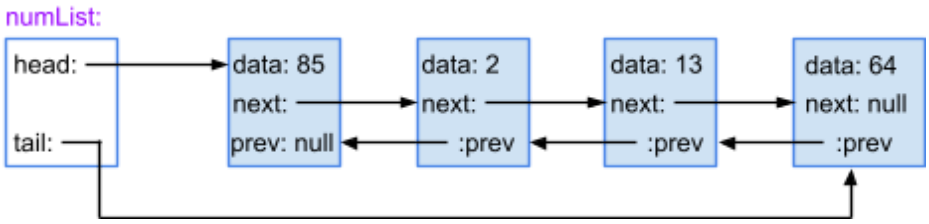
Doubly-linked list reverse traversal

A doubly-linked list also supports a reverse traversal. A ***reverse traversal*** visits all nodes starting with the list's tail node and ending after visiting the list's head node.

Figure 20.10.1: Reverse traversal algorithm.

```
ListTraverseReverse(list) {
   curNode = list→tail // Start at tail

   while (curNode is not null) {
      Print curNode's data
      curNode = curNode→prev
   }
}
```

PARTICIPATION ACTIVITY 20.10.3: Reverse traversal algorithm execution.



1) ListTraverseReverse() visits which node second?

- Node 2
- Node 13

2) ListTraverseReverse() can be used to traverse a singly-linked list.

- True
- False

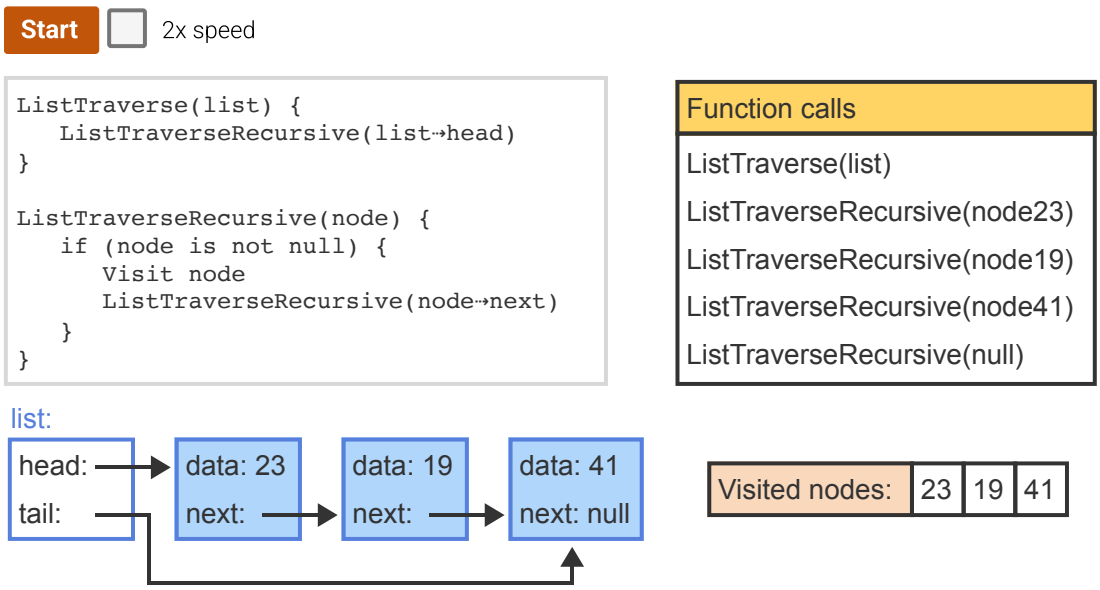
20.11 Linked lists: Recursion

Forward traversal

Forward traversal through a linked list can be implemented using a recursive function with a node parameter. If non-null, the node is visited first. Then, a recursive call is made on the node's next pointer to traverse the remainder of the list.

The ListTraverse() function takes a list as an argument, and traverses the entire list by calling ListTraverseRecursive() on the list's head.

PARTICIPATION ACTIVITY 20.11.1: Recursive forward traversal.



Captions

PARTICIPATION ACTIVITY 20.11.2: Forward traversal in a linked list with 10 nodes.

1) If ListTraverse() is called to traverse a list with 10 nodes, how many calls to ListTraverseRecursive() occur?
   - 9
   - 10
   - 11

PARTICIPATION ACTIVITY 20.11.3: Forward traversal concepts.

1) ListTraverseRecursive() works for both singly-linked and doubly-linked lists.
   - True
   - False
2) ListTraverseRecursive() works for an empty list.
   - True
   - False

Searching

A recursive linked list search is implemented similar to forward traversal. Each call examines one node. If the node is null, then null is returned. Otherwise, the node's data is compared to the search key. If a match occurs, the node is returned, otherwise the remainder of the list is searched recursively.

Figure 20.11.1: ListSearch() and ListSearchRecursive() functions.

```
ListSearch(list, key) {
   return ListSearchRecursive(key, list→head)
}

ListSearchRecursive(key, node) {
   if (node is not null) {
      if (node→data == key) {
         return node
      }
      return ListSearchRecursive(key, node→next)
   }
   return null
}
```

PARTICIPATION ACTIVITY 20.11.4: Searching a linked list with 10 nodes.

Suppose a linked list has 10 nodes.

1) When more than one of the list's nodes contains the search key, ListSearch() returns _____ node containing the key.
   - the first
   - the last
   - a random

2) Calling ListSearch() results in a minimum of _____ calls to ListSearchRecursive().
   - 1
   - 2
   - 10
   - 11

3) When the key is not found, ListSearch() returns _____.
   - the list's head
   - the list's tail
   - null

Reverse traversal

Forward traversal visits a node first, then recursively traverses the remainder of the list. If the order is swapped, such that the recursive call occurs first, the list is traversed in reverse order.

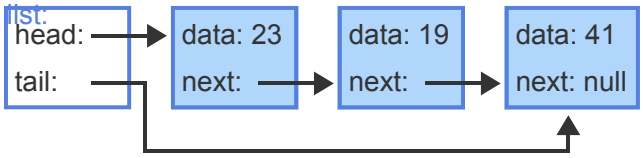
PARTICIPATION ACTIVITY 20.11.5: Recursive reverse traversal.

Start 2x speed

```
ListTraverseReverse(list) {
   ListTraverseReverseRecursive(list→head)
}

ListTraverseReverseRecursive(node) {
   if (node is not null) {
      ListTraverseReverseRecursive(node→next)
      Visit node
   }
}
```

Function calls

ListTraverseReverse(list)
ListTraverseReverseRecursive(node23)
ListTraverseReverseRecursive(node19)
ListTraverseReverseRecursive(node41)
ListTraverseReverseRecursive(null)



Visited nodes: 41 19 23

Captions

PARTICIPATION ACTIVITY 20.11.6: Reverse traversal concepts.

Suppose ListTraverseReverse() is called on the following list.

1) ListTraverseReverse() passes _____ as the argument to ListTraverseReverseRecursive().
   - node 67
   - node 18
   - null

2) ListTraverseReverseRecursive() has been called for each of the list's nodes by the time the tail node is visited.
   - True
   - False

3) If ListTraverseReverseRecursive() were called directly on node 91, the nodes visited would be: _____.
   - node 91 and node 67
   - node 18, node 46, and node 91
   - node 18, node 46, node 91, and node 67

20.12 Linked list dummy nodes

Dummy nodes

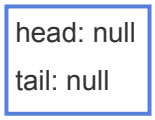
A linked list implementation may use a ***dummy node*** (or ***header node***): A node with an unused data member that always resides at the list head and cannot be removed. Using a dummy node simplifies the algorithms for a linked list because the head and tail pointers are never null.

An empty list consists of the dummy node, which has the next pointer assigned with null, and the list's head and tail pointers both point to the dummy node.

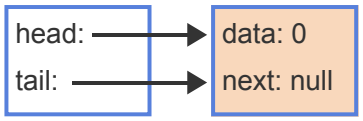
PARTICIPATION ACTIVITY 20.12.1: Singly-linked lists with and without a dummy node.

Start 2x speed

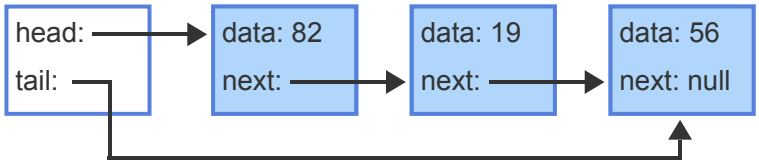
Empty linked list without dummy node:



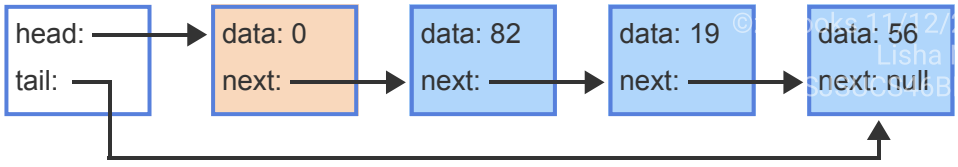
Empty linked list with dummy node:



List without dummy node and contents 82, 19, 56:



List with dummy node and contents 82, 19, 56:



Captions

PARTICIPATION ACTIVITY 20.12.2: Singly linked lists with a dummy node.

1) The head and tail pointers always point to the dummy node.
   - ○ True
   - ○ False

2) The dummy node's next pointer points to the first list item.
   - ○ True
   - ○ False

PARTICIPATION ACTIVITY 20.12.3: Condition for an empty list.

1) If myList is a singly-linked list with a dummy node, which statement is true when the list is empty?
   - ○ `myList→head == null`
   - ○ `myList→tail == null`
   - ○ `myList→head == myList→tail`

Singly-linked list implementation

When a singly-linked list with a dummy node is created, the dummy node is allocated and the list's head and tail are assigned with the dummy node.

List operations such as append, prepend, insert-node-after, and remove-node-after are simpler to implement compared to a linked list without a dummy node since a special case is removed from each implementation. ListAppendNode(), ListPrependNode(), and ListInsertNodeAfter() do not need to check if the list's head is null, since the list's head will always point to the dummy node. ListRemoveAfter() does not need a special case to allow removal of the first list item, since the first list item is after the dummy node.

Figure 20.12.1: Functions for a singly-linked list with a dummy node.

```
ListAppendNode(list, newNode) {
   list→tail→next = newNode
   list→tail = newNode
}
```

```
ListPrependNode(list, newNode) {
   newNode→next = list→head→next
   list→head→next = newNode
   if (list→head == list→tail) { // empty list
      list→tail = newNode;
   }
}


ListInsertNodeAfter(list, curNode, newNode) {
   if (curNode == list→tail) { // Insert after tail
      list→tail→next = newNode
      list→tail = newNode
   }
   else {
      newNode→next = curNode→next
      curNode→next = newNode
   }
}


ListRemoveNodeAfter(list, curNode) {
   if (curNode != null and curNode→next != null) {
      sucNode = curNode→next→next // Get successor of node to remove
      curNode→next = sucNode

      if (sucNode == null) {
         // Removed tail
         list→tail = curNode
      }
   }
}
```


PARTICIPATION ACTIVITY 20.12.4: Singly-linked list with dummy node.

Suppose dataList is a singly-linked list with a dummy node.

1) Which statement removes the first item from the list?

- `ListRemoveNodeAfter(list, null)`
- `ListRemoveNodeAfter(list, list→head)`
- `ListRemoveNodeAfter(list, list→tail)`

2) Which is a requirement of the ListPrependNode() function?

- The list is empty
- The list is not empty
- newNode is not null

PARTICIPATION ACTIVITY 20.12.5: Singly-linked list with dummy node.

Suppose numbersList is a singly-linked list with items 73, 19, and 86. Item 86 is at the list's tail.

1) What is the list's contents after the following operations?

```
lastItem = numbersList→tail
ListAppend(numbersList, node25)
ListInsertNodeAfter(numbersList,
lastItem, node49)
```

- 73, 19, 86, 25, 49
- 73, 19, 86, 49, 25
- 73, 19, 25, 49, 86

2) Suppose the following statement is executed:

```
node19 =
numbersList→head→next→next
```

Which subsequent operations swap nodes 73 and 19?

- ListPrepend(numbersList, node19)
- ListInsertNodeAfter(numbersList, numbersList→head, node19)
- ListRemoveAfter(numbersList, numbersList→head→next) ListPrepend(numbersList, node19)

Doubly-linked list implementation

A dummy node can also be used in a doubly-linked list implementation. The dummy node in a doubly-linked list always has the previous pointer assigned with null. ListRemoveNode()'s implementation does not allow removal of the dummy node.

Figure 20.12.2: Functions for a doubly-linked list with a dummy node.

```
ListAppendNode(list, newNode) {
   list→tail→next = newNode
   newNode→prev = list→tail
   list→tail = newNode
}


ListPrependNode(list, newNode) {
   firstNode = list→head→next

   // Set the next and prev pointers for newNode
   newNode→next = list→head→next
   newNode→prev = list→head

   // Set the dummy node's next pointer
   list→head→next = newNode

   if (firstNode != null) {
      // Set prev on former first node
      firstNode→prev = newNode
   }
   else {
      tail = newNode;
   }
}


ListInsertNodeAfter(list, curNode, newNode) {
   if (curNode == list→tail) { // Insert after tail
      list→tail→next = newNode
      newNode→prev = list→tail
      list→tail = newNode
   }
   else {
      sucNode = curNode→next
      newNode→next = sucNode
      newNode→prev = curNode
      curNode→next = newNode
      sucNode→prev = newNode
   }
}


ListRemoveNode(list, curNode) {
   if (curNode == list→head) {
      // Dummy node cannot be removed
      return
   }

   sucNode = curNode→next
   predNode = curNode→prev

   if (sucNode is not null) {
      sucNode→prev = predNode
   }
```

```
   // Predecessor node is always non-null
   predNode→next = sucNode

   if (curNode == list→tail) { // Removed tail
      list→tail = predNode
   }
}
```

PARTICIPATION ACTIVITY 20.12.6: Doubly-linked list with dummy node.

1) `ListPrepend(list, newNode)` is equivalent to `ListInsertNodeAfter(list, list→head, newNode)`.
   - ○ True
   - ○ False

2) ListRemove's implementation must not allow removal of the dummy node.
   - ○ True
   - ○ False

3) `ListInsertNodeAfter(list, null, newNode)` will insert newNode before the list's dummy node.
   - ○ True
   - ○ False

Dummy head and tail nodes

A doubly-linked list implementation can also use two dummy nodes: one at the head and the other at the tail. Doing so removes additional conditionals and further simplifies the implementation of most functions.

PARTICIPATION ACTIVITY 20.12.7: Doubly-linked list append and prepend with 2 dummy nodes.

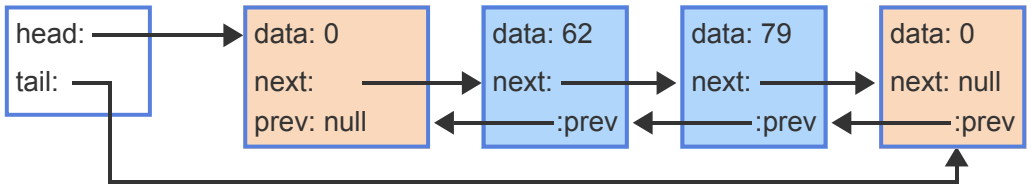
Start ☐ 2x speed

```
ListAppendNode(list, newNode) {
   newNode→prev = list→tail→prev
   newNode→next = list→tail
   list→tail→prev→next = newNode
   list→tail→prev = newNode
}
```

ListPrependNode(list, node62)

ListAppendNode(list, node79)

```
ListPrependNode(list, newNode) {
   firstNode = list→head→next
   newNode→next = list→head→next
   newNode→prev = list→head
   list→head→next = newNode
   firstNode→prev = newNode
}
```



Captions

Figure 20.12.3: Doubly-linked list with two dummy nodes: insert-node-after and remove-node operations.

```
ListInsertNodeAfter(list, curNode, newNode) {
   if (curNode == list→tail) {
      // Can't insert after dummy tail
      return
   }

   sucNode = curNode→next
   newNode→next = sucNode
   newNode→prev = curNode
   curNode→next = newNode
   sucNode→prev = newNode
}

ListRemoveNode(list, curNode) {
   if (curNode == list→head || curNode == list→tail) {
      // Dummy nodes cannot be removed
      return
   }

   sucNode = curNode→next
   predNode = curNode→prev

   // Successor node is never null
   sucNode→prev = predNode

   // Predecessor node is never null
```

```
    predNode->next = sucNode
}
```

Removing if statements from ListInsertNodeAfter() and ListRemoveNode()

The if statement at the beginning of ListInsertNodeAfter() may be removed in favor of having a requirement that curNode cannot point to the dummy tail node. Likewise, ListRemoveNode() can remove the if statement and have a requirement that curNode cannot point to either dummy node. If such requirements are met, neither function requires any if statements.

PARTICIPATION ACTIVITY 20.12.8: Comparing a doubly-linked list with one dummy node vs. two dummy nodes.

For each question, assume two implementations exist: a doubly-linked list with one dummy node at the list's head, and a doubly-linked list with two dummy nodes, one at the head and the other at the tail.

1) When `list->head == list->tail` is true in _____, the list is empty.
   - a list with one dummy node
   - a list with two dummy nodes
   - either list

2) The list's tail may be null in _____.
   - a list with one dummy node
   - a list with two dummy nodes
   - neither list type

3) `list->head->next` is always non-null in _____.
   - a list with one dummy node
   - a list with two dummy nodes
   - neither list type

20.13 Sorting linked lists

This section has been set as Optional by your instructor; the activity will not show in score reports or be synced to your LMS.

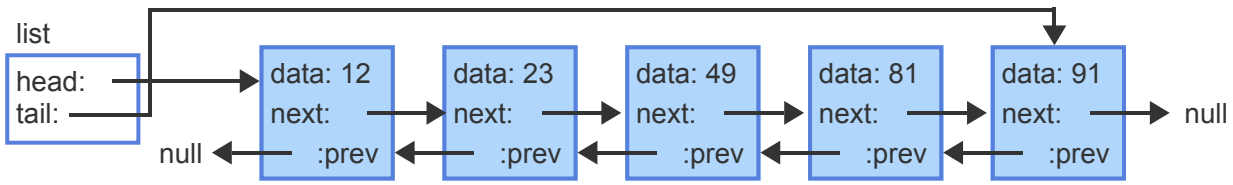
Insertion sort for doubly-linked lists

Insertion sort for a doubly-linked list operates similarly to the insertion sort algorithm used for arrays. Starting with the second list element, each element in the linked list is visited. Each visited element is moved back as needed and inserted into the correct position in the list's sorted portion. The list must be a doubly-linked list, since backward traversal is not possible in a singly-linked list.

PARTICIPATION ACTIVITY 20.13.1: Sorting a doubly-linked list with insertion sort.

Start 2x speed

```
ListInsertionSortDoublyLinked(list) {
   curNode = list→head→next
   while (curNode != null) {
      nextNode = curNode→next
      searchNode = curNode→prev
      while (searchNode != null and searchNode→data > curNode→data) {
         searchNode = searchNode→prev
      }
      // Remove and re-insert curNode
      ListRemoveNode(list, curNode)
      if (searchNode == null) {
         curNode→prev = null
         ListPrependNode(list, curNode)
      }
      else {
         ListInsertNodeAfter(list, searchNode, curNode)
      }
      // Advance to next node
      curNode = nextNode
   }
}
```



Captions

PARTICIPATION ACTIVITY 20.13.2: Improving ListInsertionSortDoublyLinked().

1) ListInsertionSortDoublyLinked() requires a minimum list length of _____.

- zero nodes
- one node
- two nodes

2) Wrapping ListInsertionSortDoublyLinked()'s body in which if statement addresses the minimum length problem?

-
```
if (list→head != null) {
   ...
}
```

-
```
if (list→head→next != null)
{
   ...
}
```

PARTICIPATION ACTIVITY 20.13.3: Insertion sort for doubly-linked lists.

ListInsertionSortDoublyLinked() is called to sort the list below.

1) What is the first node that curNode will point to?

- Node 39
- Node 45
- Node 11

2) The ordering of list nodes is not altered when node 45 is removed and then inserted after node 39.

- True
- False

3) ListPrependNode() is called on which node(s)?

- Node 11 only
- Node 22 only
- Nodes 11 and 22

Algorithm efficiency

Insertion sort's average runtime is $O(N^2)$. If a list has N elements, the outer loop executes N - 1 times. For each outer loop execution, the inner loop may need to examine all elements in the sorted part. Thus, the inner loop executes on average $N/2$ times. So the total number of comparisons is proportional to $(N-1) \cdot (N/2)$, or $O(N^2)$. In the best case scenario, the list is already sorted, and the runtime complexity is $O(N)$.

Insertion sort for singly-linked lists

Insertion sort can sort a singly-linked list by changing how each visited element is inserted into the sorted portion of the list. The standard insertion sort algorithm traverses the list from the current element toward the list head to find the insertion position. For a singly-linked list, the insertion sort algorithm can find the insertion position by traversing the list from the list head toward the current element.

Since a singly-linked list only supports inserting a node after an existing list node, the ListFindInsertionPosition() function searches the list for the insertion position and returns the list node after which the current node should be inserted. If the current node should be inserted at the head, ListFindInsertionPosition() returns null.

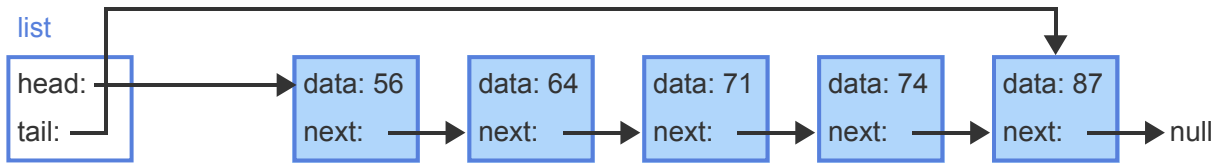
Figure 20.13.1: ListFindInsertionPosition algorithm.

```
ListFindInsertionPosition(list, dataValue) {
   curNodeA = null
   curNodeB = list→head
   while (curNodeB != null and dataValue > curNodeB→data) {
      curNodeA = curNodeB
      curNodeB = curNodeB→next
   }
   return curNodeA
}
```


PARTICIPATION ACTIVITY 20.13.4: Sorting a singly-linked list with insertion sort.

Start 2x speed

```
ListInsertionSortSinglyLinked(list) {
   beforeCurrent = list→head
   curNode = list→head→next
   while (curNode != null) {
      next = curNode→next
      position = ListFindInsertionPosition(list, curNode→data)

      if (position == beforeCurrent)
         beforeCurrent = curNode
      else {
         ListRemoveNodeAfter(list, beforeCurrent)
         if (position == null)
            ListPrependNode(list, curNode)
         else
            ListInsertNodeAfter(list, position, curNode)
      }

      curNode = next
   }
}
```



Captions

PARTICIPATION ACTIVITY 20.13.5: ListInsertionSortSinglyLinked() function.

Refer to the animation above.

1) ListInsertionSortSinglyLinked() causes an error when the list is empty.

- True
- False

2) During the final loop iteration, curNode's data value is 74. Since node 74 is only moved back one position, 74 is only compared against the prior node's data value, 87.

- True
- False

3) ListInsertionSortSinglyLinked()'s *best* case is when the list is already sorted in ascending order.

- True
- False

4) ListInsertionSortSinglyLinked()'s *best* case runtime complexity is $O(N)$.

- True
- False

PARTICIPATION ACTIVITY 20.13.6: Sorting a singly-linked list with insertion sort.

ListInsertionSortSinglyLinked() is called to sort the list below.

1) What is returned by the first call to ListFindInsertionPosition()?

- null
- Node 63
- Node 71
- Node 84

2) How many nodes are moved to the list's head?

- 0
- 1
- 2

3) How many times is ListInsertNodeAfter() called?

- 0
- 1
- 2

Sorting linked-lists vs. arrays

Sorting algorithms for arrays, such as quicksort and heapsort, require constant-time access to arbitrary, indexed locations to operate efficiently. Linked lists do not allow indexed access, making for difficult adaptation of such sorting algorithms to operate on linked lists. The tables below provide a brief overview of the challenges in adapting array sorting algorithms for linked lists.

Table 20.13.1: Sorting algorithms easily adapted to efficiently sort linked lists.

Sorting algorithm	Adaptation to linked lists
Insertion sort	Operates similarly on doubly-linked lists. Requires searching from the head of the list for an element's insertion position for singly-linked lists.
Merge sort	Finding the middle of the list requires searching linearly from the head of the list. The merge algorithm can also merge lists without additional storage.

Table 20.13.2: Sorting algorithms difficult to adapt to efficiently sort linked lists.

Sorting algorithm	Challenge
Shell sort	Jumping the gap between elements cannot be done on a linked list, as each element between two elements must be traversed.
Quicksort	Partitioning requires backward traversal through the right portion of elements. Singly-linked lists do not support backward traversal.
Heap sort	Indexed access is required to find child nodes in constant time when percolating down.

PARTICIPATION ACTIVITY 20.13.7: Sorting linked-lists vs. sorting arrays.

1) What aspect of linked lists makes adapting array-based sorting algorithms to linked lists difficult?

- Two elements in a linked list cannot be swapped in constant time.
- Nodes in a linked list cannot be moved.
- Elements in a linked list cannot be accessed by index.

2) Which sorting algorithm uses a gap value to jump between elements, and is difficult to adapt to linked lists for this reason?

- Insertion sort
- Merge sort
- Shell sort

3) Why are sorting algorithms for arrays generally more difficult to adapt to singly-linked lists than to doubly-linked lists?

- Singly-linked lists do not support backward traversal.
- Singly-linked do not support inserting nodes at arbitrary locations.

20.14 Java: Sorting linked lists

This section has been set as Optional by your instructor; the activity will not show in score reports or be synced to your LMS.

Java insertion sort for doubly-linked lists

The insertionSortDoublyLinked() method in the DoublyLinkedList class implements insertion sort for a doubly-linked list. Nodes are moved backward by first removing the node using the removeNode() method, then reinserting using either the prependNode() method (if the node becomes the head) or the insertNodeAfter() method.

Figure 20.14.1: insertionSortDoublyLinked() method.

```
public void insertionSortDoublyLinked() {
   if (head == null) {
      return;
   }

   DoublyLinkedNode currentNode = head.next;
   while (currentNode != null) {
      DoublyLinkedNode nextNode = currentNode.next;
      DoublyLinkedNode searchNode = currentNode.previous;

      while (searchNode != null && searchNode.data > currentNode.data) {
         searchNode = searchNode.previous;
      }

      if (searchNode == null) {
         removeNode(currentNode);
         currentNode.previous = null;
         prependNode(currentNode);
      }
      else if (searchNode.next != currentNode) {
         removeNode(currentNode);
         insertNodeAfter(searchNode, currentNode);
      }
      currentNode = nextNode;
   }
}
```


PARTICIPATION ACTIVITY 20.14.1: Doubly-linked list insertion sort.

1) Which combination of DoublyLinkedList methods can move a doubly-linked node backward?

   - removeNodeAfter() and appendNode()
   - removeNode() and prependNode()

2) What prevents singly-linked lists from using the doubly-linked list insertion sort algorithm?

   - Lacking a previous field.
   - Having a next field.

Java insertion sort for singly-linked lists

Insertion sort can be changed to work for a singly-linked list by traversing forward from the start of the list to find a node's insertion point. The node is then moved backward by first removing the node, then using the prependNode() or insertNodeAfter() method to insert the node at the insertion point.

Methods insertionSortSinglyLinked() and findInsertionPosition() are added to the SinglyLinkedList class. The insertionSortSinglyLinked() method calls the findInsertionPosition() method to find where the current node should be inserted.

Figure 20.14.2: Insertion sort algorithm for singly-linked lists.

```
public void insertionSortSinglyLinked() {
   if (head == null) {
      return;
   }

   SinglyLinkedNode previousNode = head;
   SinglyLinkedNode currentNode = head.next;

   while (currentNode != null) {
      SinglyLinkedNode nextNode = currentNode.next;
      SinglyLinkedNode position = findInsertionPosition(currentNode.data);

      if (position == previousNode) {
         previousNode = currentNode;
      }
      else {
         removeNodeAfter(previousNode);
         if (position == null) {
            prependNode(currentNode);
         }
         else {
            insertNodeAfter(position, currentNode);
         }
      }

      currentNode = nextNode;
   }
}

public SinglyLinkedNode findInsertionPosition(int dataValue) {
   SinglyLinkedNode positionA = null;
   SinglyLinkedNode positionB = head;
   while (positionB != null && dataValue > positionB.data) {
      positionA = positionB;
      positionB = positionB.next;
   }
   return positionA;
}
```


PARTICIPATION ACTIVITY 20.14.2: Insertion sort for singly-linked lists.

1) findInsertionPosition() returns null if the dataValue argument is ≤ the head node's data.

- True
- False

2) When findInsertionPosition() returns a non-null node not equal to previousNode, insertionSortSinglyLinked() inserts currentNode after the returned node.

- True
- False

3) The insertionSortSinglyLinked() method can also sort a doubly-linked list.

- True
- False

zyDE 20.14.1: Insertion sort for linked lists.

The following code provides two implementations of a list ADT, one with a singly-linked list and the other with a doubly-linked list. Each implements a sort() method that sorts the linked list with insertion sort.

- **Program.java** - The Program class and main() method
- **ListADT.java** - The ListADT interface
- **SinglyLinkedList.java** - The SinglyLinkedNode class and the SinglyLinkedList class that implements ListADT
- **DoublyLinkedList.java** - The DoublyLinkedNode class and the DoublyLinkedList class that implements ListADT

Current file: Program.java Load default template...

```
1  public class Program {
2     public static void main(String[] args) {
3        int[] numbers = { 72, 91, 53, 12, 48, 19, 7, 1, 86 };
4
5        // Create instances of SinglyLinkedList and DoublyLinkedList
6        ListADT[] linkedLists = {
7           new SinglyLinkedList(),
8           new DoublyLinkedList()
9        };
10
11       // Append numbers to each list
12       for (ListADT linkedList : linkedLists) {
```

```
13        for (int number : numbers) {
14           linkedList.append(number);
15        }
```

Run

20.15 LAB: List count

LAB ACTIVITY | 20.15.1: LAB: List count | Full screen | 10 / 10

Given the IntNode class, define the getCount() method in the CustomLinkedList class that returns the number of items in the list not including the head node.

Ex: If the list contains:

```
head -> 14 -> 19 -> 4
```

getCount(headObj) returns 3.

Ex: If the list contains:

```
head ->
```

getCount(headObj) returns 0.

Open new tab | Dock

Run History Tutorial

Submit for grading

Coding trail of your work What is this?

```
11/2 U 10 min:1
```


Latest submission - 11:42 PM PST on 11/02/25

Submission passed all tests ✓ Total score: 10 / 10

Only show failing tests

Open submission's code

1: Unit test 3 / 3

Test getCount() returns 3 for list with three items

Feedback

```
getCount() correctly returned 3
```

2: Unit test 3 / 3

Test getCount() returns 0 for an empty list

Feedback

```
getCount() correctly returned 0
```

3: Unit test 2 / 2

Test getCount() returns 15 for list with 15 items

Feedback

```
getCount() correctly returned 15
```

4: Unit test 2 / 2

Test getCount() returns 1000 for list with 1000 items

Feedback

```
getCount() correctly returned 1000
```

20.16 LAB: Find max in list

LAB ACTIVITY | 20.16.1: LAB: Find max in list | Full screen | 10 / 10

Given the IntNode class, define the findMax() method in the CustomLinkedList class that returns the largest value in the list or returns -99 if the list is empty. Assume all values in the list are non-negative.

Ex: If the list contains:

```
head -> 14 -> 191 -> 186 -> 181
```

findMax(headObj) returns 191.

Ex: If the list contains:

```
head ->
```

findMax(headObj) returns -99.

Open new tab | Dock

Submit for grading

Coding trail of your work What is this?

```
11/2 U 10 min:1
```

Latest submission - 11:43 PM PST on 11/02/25 — Submission passed all tests ✓ — Total score: 10 / 10

☐ Only show failing tests — Open submission's code

1: Unit test 3 / 3

Test findMax() returns largest item (at 2nd element of list)

Feedback

```
findMax() correctly returned 191
```

2: Unit test 2 / 2

Test findMax() returns -99 with an empty list

Feedback

```
findMax() correctly returned -99 for an empty list
```

3: Unit test 3 / 3

Test findMax() returns largest item (at end of list)

Feedback

```
findMax() correctly returned 10
```

4: Unit test 2 / 2

Test findMax() returns largest item (only one item in list)

Feedback

```
findMax() correctly returned 123 for list with one item
```

20.17 LAB: Index of list item

LAB ACTIVITY | 20.17.1: LAB: Index of list item | Full screen | 10 / 10

Given the IntNode class, implement the indexOf() method in the CustomLinkedList class that returns the index of parameter `target` or returns -1 if not found.

Note: The first index after the head node is 0.

Ex: If the list contains:

```
head -> 14 -> 191 -> 22 -> 99
```

indexOf(headObj, 22) returns 2.

Ex: If the list contains:

```
head ->
```

indexOf(headObj, 22) returns -1.

Open new tab | Dock

Run History Tutorial

Submit for grading

Coding trail of your work What is this?

```
11/2 U 10 min:1
```


Latest submission - 11:43 PM PST on 11/02/25

Submission passed all tests ✓ Total score: 10 / 10

Only show failing tests

Open submission's code

1: Unit test 2 / 2

Test indexOf() returns 2 for target in 3rd position

Feedback

```
indexOf() correctly returned 2
```

2: Unit test 2 / 2

Test indexOf() returns -1 for an empty list

Feedback

```
indexOf() correctly returned -1 for an empty list
```

3: Unit test 2 / 2

Test indexOf() returns 0 for target in 1st position of list

Feedback

```
indexOf() correctly returned 0
```

4: Unit test 2 / 2

Test indexOf() returns -1 if target not found

Feedback

```
indexOf() correctly returned -1
```

5: Unit test 2 / 2

Test indexOf() returns 76 for 77th time in list

Feedback

```
indexOf() correctly returned 76
```


20.18 LAB: Is list sorted

LAB ACTIVITY 20.18.1: LAB: Is list sorted

Full screen 10 / 10

Given the IntNode class, define the isSorted() method in the CustomLinkedList class that takes the head node of a linked list as a parameter and determines if the numbers in the list are in ascending order. isSorted() returns true if the list is in ascending order, has only one item, or is empty; otherwise, isSorted() returns false.

Ex: If the list contains:

```
head -> 14 -> 19 -> 22 -> 99
```

isSorted(headObj) returns true.

Ex: If the list contains:

```
head -> 14 -> 19 -> 22 -> 99 -> 14 -> 100
```

isSorted(headObj) returns false.

Ex: If the list contains:

```
head ->
```

isSorted(headObj) returns true.

Open new tab | Dock

Run History Tutorial

Submit for grading

Coding trail of your work What is this?

```
11/2 U 10 min:1
```


Latest submission - 11:44 PM PST on 11/02/25

Submission passed all tests ✓ Total score: 10 / 10

Only show failing tests

Open submission's code

1: Unit test 3 / 3

Test isSorted() returns true when items in sorted order

Feedback

```
isSorted() correctly returned true
```

2: Unit test 3 / 3

Test isSorted() returns false when items are not in sorted order

Feedback

```
isSorted() correctly returned false
```

3: Unit test 2 / 2

Test isSorted() returns true when the list is empty

Feedback

```
isSorted() correctly returned true for an empty list
```

4: Unit test 2 / 2

Test isSorted() returns true when list contains one item

Feedback

```
isSorted() correctly returned true for list with one item
```

20.19 LAB: Mileage tracker for a runner

LAB ACTIVITY | 20.19.1: LAB: Mileage tracker for a runner Full screen 10 / 10

Given the `MileageTrackerNode` class, complete main() in the `MileageTrackerLinkedList` class to insert nodes into a linked list (using the insertAfter() method). The first user-input value is the number of nodes in the linked list. Use the printNodeData() method to print the entire linked list. **DO NOT print the dummy head node.**

Ex. If the input is:

```
3
2.2
7/2/18
3.2
7/7/18
4.5
7/16/18
```


the output is:

```
2.2, 7/2/18
3.2, 7/7/18
4.5, 7/16/18
```

Submit for grading

Coding trail of your work What is this?

```
11/2 U 10 min:1
```


Latest submission - 11:44 PM PST on 11/02/25

Submission passed all tests ✓ Total score: 10 / 10

☐ Only show failing tests

Open submission's code

1: Compare output 2 / 2

Compare output

Input

```
3
2.2
7/2/18
3.2
7/7/18
4.5
7/16/18
```


Your Output

```
1. 2.2, 7/2/18
2. 3.2, 7/7/18
3. 4.5, 7/16/18
4.
```

Expected output

```
1. 2.2, 7/2/18
2. 3.2, 7/7/18
3. 4.5, 7/16/18
4.
```

2: Compare output 2 / 2

Compare output

Input

```
2
8.0
8/1/18
8.5
8/3/18
```

Your Output

```
1. 8.0, 8/1/18
2. 8.5, 8/3/18
3.
```

Expected output

```
1. 8.0, 8/1/18
2. 8.5, 8/3/18
3.
```

3: Compare output 2 / 2

Compare output

```
4
1.0
3/1/18
1.5
```

Input

```
3/3/18
2.0
3/5/18
2.2
3/8/18
```

Your Output

```
1. 1.0, 3/1/18
2. 1.5, 3/3/18
3. 2.0, 3/5/18
4. 2.2, 3/8/18
5.
```

Expected output

```
1. 1.0, 3/1/18
2. 1.5, 3/3/18
3. 2.0, 3/5/18
4. 2.2, 3/8/18
5.
```

4: Compare output

2 / 2

Compare output

Input

```
1
4.8
2/2/17
```

Your Output

```
1. 4.8, 2/2/17
2.
```

Expected output

```
1. 4.8, 2/2/17
2.
```

5: Compare output

2 / 2

Compare output

Input

```
5
18
5/1/17
18.2
5/3/17
18.9
5/5/17
20.5
5/8/17
24
5/15/17
```

Your Output	Expected output
18.0, 5/1/17 18.2, 5/3/17 18.9, 5/5/17 20.5, 5/8/17 24.0, 5/15/17	18.0, 5/1/17 18.2, 5/3/17 18.9, 5/5/17 20.5, 5/8/17 24.0, 5/15/17

20.20 LAB: Inventory (linked lists: insert at the front of a list)

LAB ACTIVITY | 20.20.1: LAB: Inventory (linked lists: insert at the front of a list) Full screen 10 / 10

Given main() in the `Inventory` class, define an insertAtFront() method in the `InventoryNode` class that inserts items at the front of a linked list (after the dummy head node).

Ex. If the input is:

```
4
plates 100
spoons 200
cups 150
forks 200
```

the output is:

```
200 forks
150 cups
200 spoons
100 plates
```

Open new tab | Dock

Run History Tutorial

Submit for grading

Coding trail of your work What is this?

```
11/2 U 0 ,0 ,10 min:3
```


Latest submission - 11:47 PM PST on 11/02/25

Submission passed all tests ✓ Total score: 10 / 10

Only show failing tests

Open submission's code

1: Compare output 2 / 2

Compare output

Input

```
4
plates 100
spoons 200
cups 150
forks 200
```


Your Output

```
1. 200 forks
2. 150 cups
3. 200 spoons
4. 100 plates
5.
```

Expected output

```
1. 200 forks
2. 150 cups
3. 200 spoons
4. 100 plates
5.
```

2: Compare output 2 / 2

Compare output

Input

```
2
elliptical 4
treadmill 8
```

Your Output

```
1. 8 treadmill
2. 4 elliptical
3.
```

Expected output

```
1. 8 treadmill
2. 4 elliptical
3.
```

3: Compare output 2 / 2

Compare output

Input

```
6
Refrigerator 2
Microwave 4
Oven 1
Dishwasher 6
```

```
Stand-Mixer 12
Blender 24
```

Your Output

```
1. 24 Blender
2. 12 Stand-Mixer
3. 6 Dishwasher
4. 1 Oven
5. 4 Microwave
6. 2 Refrigerator
7.
```

Expected output

```
1. 24 Blender
2. 12 Stand-Mixer
3. 6 Dishwasher
4. 1 Oven
5. 4 Microwave
6. 2 Refrigerator
7.
```

4: Compare output 2 / 2

Compare output

Input

```
3
Crib 4
Bouncer 25
Swing 25
```

Your Output

```
1. 25 Swing
2. 25 Bouncer
3. 4 Crib
4.
```

Expected output

```
1. 25 Swing
2. 25 Bouncer
3. 4 Crib
4.
```

5: Compare output 2 / 2

Compare output

Input

```
4
Notebooks 800
Laptop 120
Pens 1000
Paper 100
```


Your Output

```
1. 100 Paper
2. 1000 Pens
```

Expected output

```
1. 100 Paper
2. 1000 Pens
```

```
120 Laptop
800 Notebooks
```

```
120 Laptop
800 Notebooks
```

Previous submissions

11:45 PM on 11/2/25	0 / 10	View
11:45 PM on 11/2/25	0 / 10	View

20.21 LAB: Playlist (output linked list)

LAB ACTIVITY | 20.21.1: LAB: Playlist (output linked list) | Full screen | 0 / 10

Given main(), complete the `SongNode` class to include the printSongInfo() method. Then write the `Playlist` class' printPlaylist() method to print all songs in the playlist. **DO NOT print the dummy head node.**

Ex: If the input is:

```
Stomp!
380
The Brothers Johnson
The Dude
337
Quincy Jones
You Don't Own Me
151
Lesley Gore
-1
```

the output is:

```
LIST OF SONGS
-------------
Title: Stomp!
Length: 380
Artist: The Brothers Johnson

Title: The Dude
Length: 337
```

```
Artist: Quincy Jones

Title: You Don't Own Me
Length: 151
Artist: Lesley Gore
```

Open new tab | Dock

Run History Tutorial

1

Submit for grading

Coding trail of your work What is this?

```
History of your effort will appear here once you begin working
on this zyLab.
```

Latest submission

No submissions yet

20.22 LAB: Grocery shopping list (linked list: inserting at the end of a list)

LAB ACTIVITY | 20.22.1: LAB: Grocery shopping list (linked list: inserting at the end of a list) | Full screen | 0 / 10

Given main() in the `ShoppingList` class, define an insertAtEnd() method in the `ItemNode` class that adds an element to the end of a linked list. **DO NOT print the dummy head node.**

Ex. if the input is:

```
4
Kale
Lettuce
Carrots
Peanuts
```

where 4 is the number of items to be inserted; Kale, Lettuce, Carrots, Peanuts are the names of the items to be added at the end of the list.

The output is:

```
Kale
Lettuce
Carrots
Peanuts
```

Open new tab | Dock

Run History Tutorial

```
1
```

Submit for grading

Coding trail of your work What is this?

```
History of your effort will appear here once you begin working
on this zyLab.
```

Latest submission

No submissions yet

20.23 LAB: Warm up: Contacts

LAB ACTIVITY | 20.23.1: LAB: Warm up: Contacts | Full screen | 0 / 10

A linked list is built in this lab. Make sure to keep track of the head node.

(1) Complete two files to submit.

- ContactNode.java - Class definition
- ContactList.java - Contains main() method

(2) Build the ContactNode class per the following specifications:

- Private fields
  - String contactName
  - String contactPhoneNumber
  - ContactNode nextNodePtr
- Constructor with parameters for name followed by phone number (1 pt)
- Public member methods
  - getName() - Accessor (1 pt)
  - getPhoneNumber() - Accessor (1 pt)
  - insertAfter() (2 pts)
  - getNext() - Accessor (1 pt)
  - printContactNode()

Ex: If the name is Roxanne Hughes and the phone number is 443-555-2864, printContactNode() outputs:

```
Name: Roxanne Hughes
Phone number: 443-555-2864
```

(3) Define main() to read the name and phone number for three contacts and output each contact. Create three ContactNodes and use the nodes to build a linked list. (2 pts)

Ex: If the input is:

```
Roxanne Hughes
443-555-2864
Juan Alberto Jr.
410-555-9385
Rachel Phillips
310-555-6610
```


the output is:

```
Person 1: Roxanne Hughes, 443-555-2864
Person 2: Juan Alberto Jr., 410-555-9385
Person 3: Rachel Phillips, 310-555-6610
```

(4) Output the linked list using a loop to output contacts one at a time. (2 pts)

Ex:

```
CONTACT LIST
Name: Roxanne Hughes
Phone number: 443-555-2864

Name: Juan Alberto Jr.
Phone number: 410-555-9385

Name: Rachel Phillips
Phone number: 310-555-6610
```

Run History Tutorial

Submit for grading

Coding trail of your work What is this?

```
History of your effort will appear here once you begin working
on this zyLab.
```

Latest submission

No submissions yet

20.24 LAB: Output a linked list

This section's content is not available for print.

20.25 LAB: Library book sorting

LAB ACTIVITY | 20.25.1: LAB: Library book sorting | Full screen | 0 / 10

Two sorted lists have been created, one implemented using a linked list (`LinkedListLibrary linkedListLibrary`) and the other implemented using the built-in ArrayList class (`ArrayListLibrary arrayListLibrary`). Each list contains 100 books (title, ISBN number, author), sorted in ascending order by ISBN number.

Complete main() by inserting a new book into each list using the respective `LinkedListLibrary` and `ArrayListLibrary` insertSorted() methods and outputting the number of operations the computer must perform to insert the new book. Each insertSorted() returns the number of operations the computer performs.

Ex: If the input is:

```
The Catcher in the Rye
9787543321724
J.D. Salinger
```

the output is:

```
Number of linked list operations: 1
Number of ArrayList operations: 1
```

Which list do you think will require the most operations? Why?

Open new tab | Dock

Submit for grading

Coding trail of your work What is this?

```
History of your effort will appear here once you begin working
on this zyLab.
```

Latest submission

No submissions yet

20.26 LAB*: Program: Playlist

This section's content is not available for print.