6.1 Java example: Map values using a generic method

©zyBooks 12/08/22 21:53 1361995
John Farrell
COLOSTATECS165WakefieldFall2022

zyDE 6.1.1: Map a value using a generic method.

The program below uses a generic method to map numeric, string, or character value shorter list of values. The program demonstrates a mapping for integers using a tabl

100
200
300
400
500
600

The program gets an integer value from a user and returns the first value in the table greater than or equal to the user value, or the user value itself if that value is greater t largest value in the table. Ex:

165 returns 200
444 returns 500
888 returns 888

1. Run the program and notice the input value 137 is mapped to 200. Try changin input value and running again.
2. Modify the program to call the getMapping method for a double and a string, si the integer.
3. Run the program again and enter an integer, a double, and a string

Load default ter

```
1  import java.util.Scanner;
2
3  public class GenericMappingArrays {
4     public static <MapType extends Comparable<MapType>>
5                    MapType getMapping(MapType mapMe, MapType [] mappings) {
6        MapType result;
7        int i;
8        int len;
9        boolean keepLooking;
10
11       result = mapMe;
12       len = mappings.length;
13       keepLooking = true;
14
15       System.out.println();
16       System.out.print("Mapping range: ");
```

```
17        for (i = 0; i < len; ++i) {
```

```
137
```

Run

zyDE 6.1.2: Map a value using a generic method (solution).

A solution to the above problem follows.

Load default ter

```
1  import java.util.Scanner;
2
3  public class GenericMappingArraysSolution {
4     public static <MapType extends Comparable<MapType>>
5                    MapType getMapping(MapType mapMe, MapType[] mappings) {
6        MapType result;
7        int i;
8        int len;
9        boolean keepLooking;
10
11       result = mapMe;
12       len = mappings.length;
13       keepLooking = true;
14
15       System.out.println();
16       System.out.print("Mapping range: ");
17       for (i = 0; i < len; ++i) {
```

```
137
4.44444
Hi
```

Run

6.2 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, search whether 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 6.2.1: List ADT.

Animation captions:

1. A new list named "ages" is created. Items can be appended to the list. The items are ordered.
2. Printing the list prints the items in order.
3. Removing an item keeps the remaining items in order.

PARTICIPATION ACTIVITY 6.2.2: 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 6.2.1: 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
Search(list, x)	Returns item if found, else returns null	Search(list, 99), returns item 99 Search(list, 22), returns null
Print(list)	Prints list's items in order	Print(list) outputs: 99, 77
PrintReverse(list)	Prints list's items in reverse order	PrintReverse(list) outputs: 77, 99
Sort(list)	Sorts the lists items in ascending order	list becomes: 77, 99
IsEmpty(list)	Returns true if list has no items	For list 99, 77, IsEmpty(list) returns false
GetLength(list)	Returns the number of items in the list	GetLength(list) returns 2

PARTICIPATION ACTIVITY 6.2.3: List ADT common operations.

1) Given a list with items 40, 888, -3, 2, what does GetLength(list) return?

- 4
- Fails

2) Given a list with items 'Z', 'A', 'B', Sort(list) yields 'A', 'B', 'Z'.

- True
- False

3) If a list ADT has operations like Sort or PrintReverse, the list is clearly implemented using an array.

- True
- False

6.3 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.

null

null *is a special value indicating a pointer points to nothing.*

The name used to represent a pointer (or reference) that points to nothing varies between programming languages and includes nil, nullptr, None, NULL, and even the value 0.

PARTICIPATION ACTIVITY 6.3.1: Singly-linked list: Each node points to the next node.

Animation content:

undefined

Animation captions:

1. A new list item is created, with the head and tail pointers pointing to nothing (null), meaning

the list is empty.

2. ListAppend points the list's head and tail pointers to a new node, whose next pointer points to null.
3. Another append points the last node's next pointer and the list's tail pointer to the new node.
4. Operations like ListAppend, ListPrepend, ListInsertAfter, and ListRemove, update just a few relevant pointers.
5. The list's first node is called the head. The last node is the tail.

PARTICIPATION ACTIVITY 6.3.2: Singly-linked list data structure.

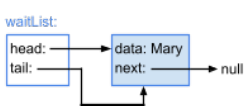
1) Given charList, C's next pointer value is _____.

Check Show answer

2) Given attendList, the head node's data value is _____. (Answer "None" if no head exists)

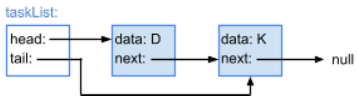
Check Show answer

3) Given waitList, the tail node's data value is _____. (Answer "None" if no tail exists)



Check Show answer

4) Given taskList, node D is followed by node _____.



Check Show answer

Appending a node to a singly-linked list

Given a new node, the ***Append*** operation for a singly-linked list inserts the new node after the list's tail node. Ex: ListAppend(numsList, node 45) appends node 45 to numsList. The notation "node 45" represents a pointer to a node with a data value of 45. This material does not discuss language-specific topics on object creation or memory allocation.

The append algorithm behavior differs if the list is empty versus not empty:

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

PARTICIPATION ACTIVITY 6.3.3: Singly-linked list: Appending a node.

Animation content:

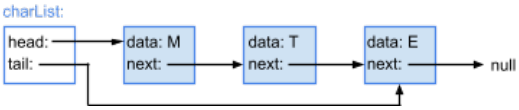
undefined

Animation captions:

1. Appending an item to an empty list updates both the list's head and tail pointers.
2. Appending to a non-empty list adds the new node after the tail node and updates the tail pointer.

PARTICIPATION ACTIVITY 6.3.4: Appending a node to a singly-linked list.

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

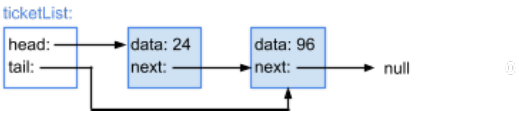


- M
- T
- E

2) Appending node W to sampleList updates which of sampleList'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`

Prepending a node to a singly-linked list

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 (empty), 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 (not empty), the algorithm points

the new node's next pointer to the head node, and then points the list's head pointer to the new node.

PARTICIPATION ACTIVITY 6.3.5: Singly-linked list: Prepending a node.

Animation content:

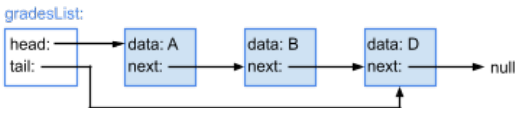
undefined

Animation captions:

1. Prepending an item to an empty list points the list's head and tail pointers to new node.
2. Prepending to a non-empty list points the new node's next pointer to the list's head node.
3. Prepending then points the list's head pointer to the new node.

PARTICIPATION ACTIVITY 6.3.6: Prepending a node in a singly-linked list.

1) Prepending C to gradesList updates which pointer?

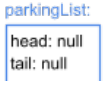


- The list's head pointer
- A's next pointer
- D's next pointer

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 Evelyn to deliveryList executes which statement?

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

CHALLENGE ACTIVITY 6.3.1: Singly-linked lists.

422352.2723990.qx3zqy7

Start

numList = new List
ListAppend(numList, node 60)
ListAppend(numList, node 95)
ListAppend(numList, node 56)
ListAppend(numList, node 96)

numList is now: Ex: 1, 2, 3

1 2 3 4 5

Check Next

6.4 Singly-linked lists: Insert

Given a new node, the ***InsertAfter*** operation for a singly-linked list inserts the new node after a provided existing list node. curNode is a pointer to an existing list node, but can be null when inserting into an empty list. The InsertAfter 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 curNode 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 curNode does not point to the list's tail node, the algorithm points the new node's next pointer to curNode's next node, and then points curNode's next pointer to the new node.

PARTICIPATION ACTIVITY 6.4.1: Singly-linked list: Insert nodes.

Animation content:

`undefined`

Animation captions:

1. Inserting the list's first node points the list's head and tail pointers to newNode.
2. Inserting after the tail node points the tail node's next pointer to newNode.
3. Then, the list's tail pointer is pointed to newNode.
4. Inserting into the middle of the list points newNode's next pointer to curNode's next node.
5. Then, curNode's next pointer is pointed to newNode.

PARTICIPATION ACTIVITY 6.4.2: Inserting nodes in a singly-linked list.

Type the list after the given operations. Type the list as: 5, 7, 9

1) numsList: 5, 9

ListInsertAfter(numsList, node 9, node 4)

numsList:

2) [illegible] 3, 17, 8

Check Show answer

ListInsertAfter(numsList, node 23, node 5)

numsList:

Check Show answer

3) numsList: 1

ListInsertAfter(numsList, node 1, node 6)
ListInsertAfter(numsList, node 1, node 4)

numsList:

Check Show answer

4) numsList: 77

ListInsertAfter(numsList, node 77, node 32)
ListInsertAfter(numsList, node 32, node 50)
ListInsertAfter(numsList, node 32, node 46)

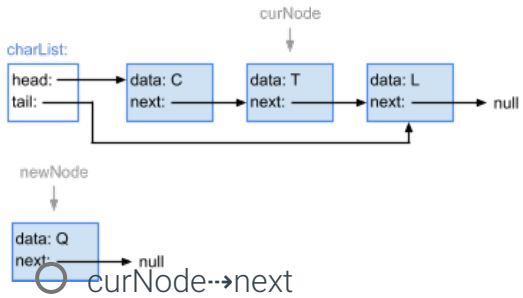
numsList:

Check Show answer

PARTICIPATION ACTIVITY 6.4.3: Singly-linked list insert-after algorithm.

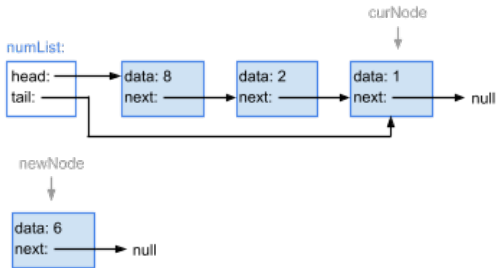
1) ListInsertAfter(charList, node T, node

Q) assigns newNode's next pointer with _____.



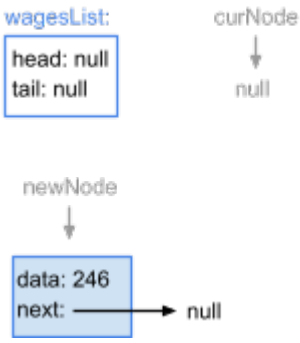
- curNode⇢next
- charList's head node
- null

2) ListInsertAfter(numList, node 1, node 6) executes which statement?



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

3) ListInsertAfter(wagesList, list head, node 246) executes which statement?



- `list→head = newNode`

CHALLENGE ACTIVITY 6.4.1: Singly-linked lists: Insert.

422352.2723990.qx3zqy7

Start

What is numList after the following operations?

numList: 84, 80
ListInsertAfter(numList, node 80, node 59)
ListInsertAfter(numList, node 84, node 99)
ListInsertAfter(numList, node 84, node 83)

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

1 2 3 4

Check Next

6.5 Singly-linked lists: Remove

Given a specified existing node in a singly-linked list, the ***RemoveAfter*** 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 curNode parameter. If curNode is null, RemoveAfter removes the list's first node. Otherwise, the algorithm removes the node after curNode.

- *Remove list's head node (special case):* If curNode is null, the algorithm points sucNode to the

head node's next node, and points the list's head pointer to sucNode. If sucNode is null, the only list node was removed, so the list's tail pointer is pointed to null (indicating the list is now empty).

- *Remove node after curNode:* If curNode's next pointer is not null (a node after curNode exists), the algorithm points sucNode to the node after curNode's next node. Then curNode's next pointer is pointed to sucNode. If sucNode is null, the list's tail node was removed, so the algorithm points the list's tail pointer to curNode (the new tail node).

PARTICIPATION ACTIVITY 6.5.1: Singly-linked list: Node removal.

Animation content:

undefined

Animation captions:

1. If curNode is null, the list's head node is removed.
2. The list's head pointer is pointed to the list head's successor node.
3. If node exists after curNode exists, that node is removed. sucNode points to node after the next node (i.e., the next next node).
4. curNode's next pointer is pointed to sucNode.
5. If sucNode is null, the list's tail node was removed. curNode is now the list tail node.
6. If list's tail node is removed, curNode's next pointer is null.
7. If list's tail node is removed, the list's tail pointer is pointed to curNode.

PARTICIPATION ACTIVITY 6.5.2: 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

ListRemoveAfter(numsList, node 5)

numsList:

Check Show answer

2) numsList: 3, 57, 28, 40

ListRemoveAfter(numsList, null)

numsList:

Check Show answer

3) numsList: 9, 4, 11, 7

ListRemoveAfter(numsList, node 11)

numsList:

Check Show answer

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

ListRemoveAfter(numsList, node 40)
ListRemoveAfter(numsList, node 20)

numsList:

Check Show answer

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

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

numsList:

Check Show answer

©zyBooks 12/08/22 21:53 1361995
John Farrell
COLOSTATECS165WakefieldFall2022

PARTICIPATION ACTIVITY 6.5.3: ListRemoveAfter algorithm execution: Intermediate node.

Given numList, ListRemoveAfter(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

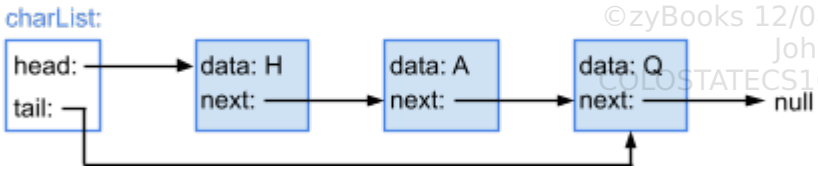
- Yes
- No

4) list→tail = curNode

- Yes
- No

PARTICIPATION ACTIVITY 6.5.4: ListRemoveAfter algorithm execution: List head node.

Given charList, ListRemoveAfter(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

CHALLENGE ACTIVITY 6.5.1: Singly-linked lists: Remove.

422352.2723990.qx3zqy7

Start

Given list: 9, 2, 8, 4, 7, 5
What list results from the following operations?

ListRemoveAfter(list, node 2)
ListRemoveAfter(list, node 7)
ListRemoveAfter(list, null)

List items in order, from head to tail.

Ex: 25, 42, 12

1	2	3	4	5

Check Next

6.6 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 points 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.

PARTICIPATION ACTIVITY 6.6.1: 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

Appending a node to a doubly-linked list

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 behavior differs if the list is empty versus not empty:

- *Append to empty list:* If the list's head pointer is null (empty), 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 (not empty), 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 6.6.2: Doubly-linked list: Appending a node.

Animation content:

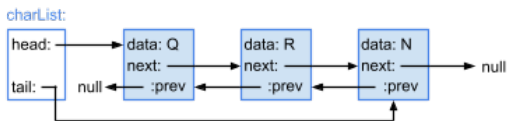
undefined

Animation captions:

1. Appending an item to an empty list updates the list's head and tail pointers.
2. Appending to a non-empty list adds the new node after the tail node and updates the tail pointer.
3. newNode's previous pointer is pointed to the list's tail node.
4. The list's tail pointer is then pointed to the new node.

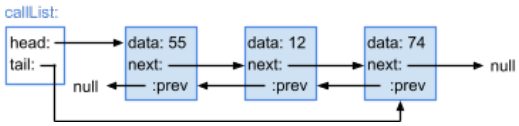
PARTICIPATION ACTIVITY 6.6.3: Doubly-linked list data structure.

1) ListAppend(charList, node F) inserts node F _____.



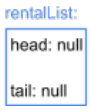
- after node Q

2) ListAppend(callList, node 5) executes which statement?
   - before node N
   - after node N



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

3) Appending node K to rentalList executes which of the following statements?



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

Prepending a node to a doubly-linked list

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 (empty), 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 (not empty), 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 6.6.4: Doubly-linked list: Prepending a node.

Animation content:

undefined

Animation captions:

1. Prepending an item to an empty list points the list's head and tail pointers to new node.
2. Prepending to a non-empty list points new node's next pointer to the list's head node.
3. Prepending then points the head node's previous pointer to the new node.
4. Then the list's head pointer is pointed to the new node.

PARTICIPATION ACTIVITY 6.6.5: Prepending a node in a doubly-linked list.

1) Prepending 29 to trainsList updates the list's head pointer to point to node _____.

- 4
- 29
- 31

2) ListPrepend(shoppingList, node Milk) updates the list's tail pointer.

- True
- False

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

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

CHALLENGE ACTIVITY 6.6.1: Doubly-linked lists.

422352.2723990.qx3zqy7

Start

numList = new List
ListAppend(numList, node 80)
ListAppend(numList, node 71)

numList is now: Ex: 1, 2, 3

Which node has a null next pointer? Ex: 5

Which node has a null previous pointer? Ex: 5

1	2	3	4	5

Check Next

6.7 Doubly-linked lists: Insert

Given a new node, the ***InsertAfter*** operation for a doubly-linked list inserts the new node after a provided existing list node. curNode is a pointer to an existing list node. The InsertAfter algorithm considers three insertion scenarios:

- *Insert as first node:* If the list's head pointer is null (list is empty), 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 is not empty) and curNode points to the list's tail node, the new node is inserted after the tail node. 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 then points the list's tail pointer to the new node.
- *Insert in middle of list:* If the list's head pointer is not null (list is not empty) and curNode does not point to the list's tail node, the algorithm updates the current, new, and successor nodes' next and previous pointers to achieve the ordering {curNode newNode sucNode}, which requires four pointer updates: point the new node's next pointer to sucNode, point the new node's previous pointer to curNode, point curNode's next pointer to the new node, and point sucNode's previous pointer to the new node.

PARTICIPATION ACTIVITY 6.7.1: Doubly-linked list: Inserting nodes.

Animation content:

undefined

Animation captions:

1. Inserting a first node into the list points the list's head and tail pointers to the new node.
2. Inserting after the list's tail node points the tail node's next pointer to the new node.
3. Then the new node's previous pointer is pointed to the list's tail node. Finally, the list's tail pointer is pointed to the new node.
4. Inserting in the middle of a list points sucNode to curNode's successor (curNode's next node), then points newNode's next pointer to the successor node....
5. ...then points newNode's previous pointer to curNode...
6. ...and finally points curNode's next pointer to the new node.
7. Finally, points sucNode's previous pointer to the new node. At most, four pointers are updated to insert a new node in the list.

PARTICIPATION ACTIVITY 6.7.2: Inserting nodes in a doubly-linked list.

Given weeklySalesList: 12, 30
Show the node order after the following operations:

ListInsertAfter(weeklySalesList, list tail, node 8)
ListInsertAfter(weeklySalesList, list head, node 45)
ListInsertAfter(weeklySalesList, node 45, node 76)

If unable to drag and drop, refresh the page.

node 45 **node 12** **node 76** **node 30** **node 8**

Position 0 (list's head node)

Position 1

Position 2

Position 3

Position 4 (list's tail node)

Reset

CHALLENGE ACTIVITY 6.7.1: Doubly-linked lists: Insert.

422352.2723990.qx3zqy7

Start

numList: 52, 29
ListInsertAfter(numList, node 52, node 61)
ListInsertAfter(numList, node 61, node 14)
ListInsertAfter(numList, node 29, node 81)
ListInsertAfter(numList, node 29, node 25)

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

What node does node 81's next pointer point to? node 14

What node does node 81's previous pointer point to? node 14

1 2 3 4

Check Next

6.8 Doubly-linked lists: Remove

The ***Remove*** operation for a doubly-linked list removes a provided existing list node. curNode 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 variable sucNode points to the node's successor, and the variable predNode points to the node's predecessor. The algorithm uses four separate checks to update each pointer:

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

When removing a node in the middle of the list, both the predecessor and successor nodes exist, and the algorithm updates the predecessor and successor nodes' pointers to achieve the ordering {predNode sucNode}. When removing the only node in a list, curNode points to both the list's head and tail nodes, and sucNode and predNode are both null. So, the algorithm points the list's head and tail pointers to null, making the list empty.

PARTICIPATION ACTIVITY 6.8.1: Doubly-linked list: Node removal.

Animation content:

`undefined`

Animation captions:

1. curNode points to the node to be removed. sucNode points to curNode's successor (curNode's next node). predNode points to curNode's predecessor (curNode's previous node).
2. sucNode's previous pointer is pointed to the node preceding curNode.
3. If curNode points to the list's head node, the list's head pointer is pointed to the successor

node. With the pointers updated, curNode can be removed.

4. curNode points to node 5, which will be removed. sucNode points to node 2. predNode points node 4.
5. The predecessor node's next pointer is pointed to the successor node. The successor node's previous pointer is pointed to the predecessor node. With pointers updated, curNode can be removed.
6. curNode points to node 2, which will be removed. sucNode points to nothing (null). predNode points to node 4.
7. The predecessor node's next pointer is pointed to the successor node. If curNode points to the list's tail node, the list's tail pointer is assigned with predNode. With pointers updated, curNode can be removed.

PARTICIPATION ACTIVITY 6.8.2: Deleting nodes from a doubly-linked list.

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

1) numsList: 71, 29, 54

ListRemove(numsList, node 29)

numsList:

Check Show answer

2) numsList: 2, 8, 1

ListRemove(numsList, list tail)

numsList:

Check Show answer

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

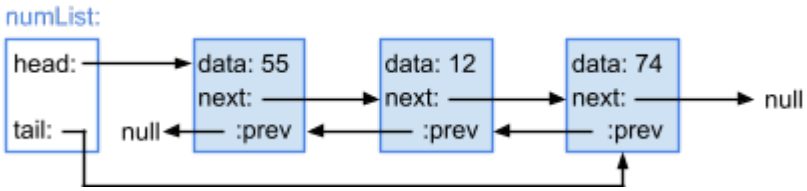
ListRemove(numsList, node 82)
ListRemove(numsList, node 357)
ListRemove(numsList, node 66)

numsList:

Check Show answer

PARTICIPATION ACTIVITY 6.8.3: ListRemove algorithm execution: Intermediate node.

Given numList, ListRemove(numList, node 12) executes which of the following statements?



1) sucNode⇢prev = predNode

- Yes
- No

2) predNode⇢next = sucNode

- Yes
- No

3) list⇢head = sucNode

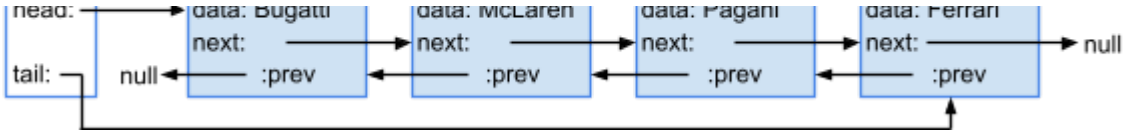
- Yes
- No

4) list⇢tail = predNode

- Yes
- No

PARTICIPATION ACTIVITY 6.8.4: ListRemove algorithm execution: List head node.

Given carList, ListRemove(carList, node Bugatti) executes which of the following statements?



1) sucNode⇢prev = predNode

- Yes
- No

2) predNode⇢next = sucNode

- Yes
- No

3) list⇢head = sucNode

- Yes
- No

4) list⇢tail = predNode

- Yes
- No

CHALLENGE ACTIVITY 6.8.1: Doubly-linked lists: Remove.

422352.2723990.qx3zqy7

Start

Given list: 7, 6, 9, 2, 1
What list results from the following operations?

ListRemove(list, node 1)
ListRemove(list, node 7)
ListRemove(list, node 9)

List items in order, from head to tail.

1 2 3 4 5

Check Next

6.9 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.

Figure 6.9.1: Linked list traversal algorithm.

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

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


PARTICIPATION ACTIVITY 6.9.1: Singly-linked list: List traversal.

Animation content:

undefined

Animation captions:

1. Traverse starts at the list's head node.
2. curNode's data is printed, and then curNode is pointed to the next node.
3. After the list's tail node is printed, curNode is pointed to the tail node's next node, which does not exist.
4. The traversal ends when curNode is null.

PARTICIPATION ACTIVITY 6.9.2: List traversal.

1) ListTraverse begins with _____.
   - 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

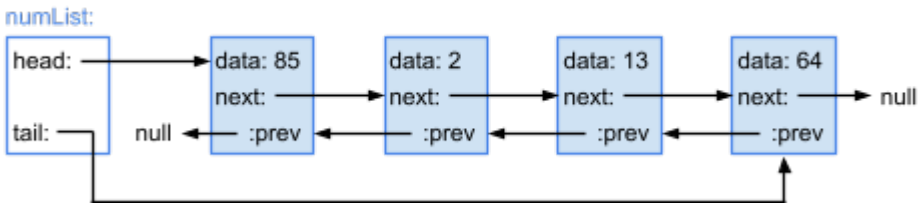
A doubly-linked list also supports a reverse traversal. A ***reverse traversal*** visits all nodes starting with the list's tail node and ending after visiting the list's head node.

Figure 6.9.2: 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 6.9.3: 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

6.10 Sorting linked lists

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 6.10.1: Sorting a doubly-linked list with insertion sort.

Animation content:

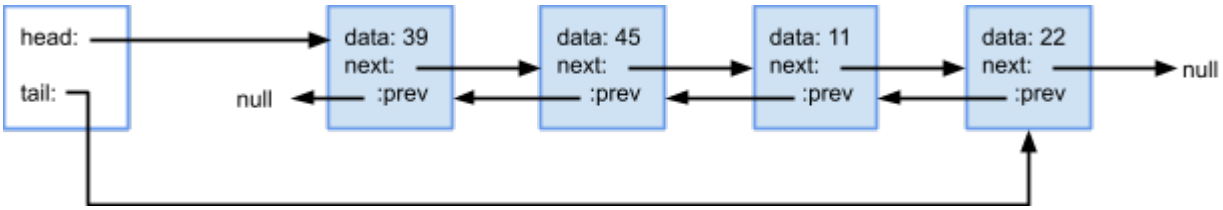
undefined

Animation captions:

1. The curNode pointer begins at node 91 in the list.
2. searchNode starts at node 81 and does not move because 81 is not greater than 91. Removing and re-inserting node 91 after node 81 does not change the list.
3. For node 23, searchNode traverses the list backward until becoming null. Node 23 is prepended as the new list head.
4. Node 49 is inserted after node 23, using ListInsertAfter.
5. Node 12 is inserted before node 23, using ListPrepend, to complete the sort.

PARTICIPATION ACTIVITY 6.10.2: Insertion sort for doubly-linked lists.

Suppose ListInsertionSortDoublyLinked is executed 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) ListPrepend is called on which

node(s)?

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

Algorithm efficiency

Insertion sort's typical 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 algorithm 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.

PARTICIPATION ACTIVITY 6.10.3: Sorting a singly-linked list with insertion sort.

Animation content:

undefined

Animation captions:

1. Insertion sort for a singly-linked list initializes curNode to point to the second list element,

or node 56.

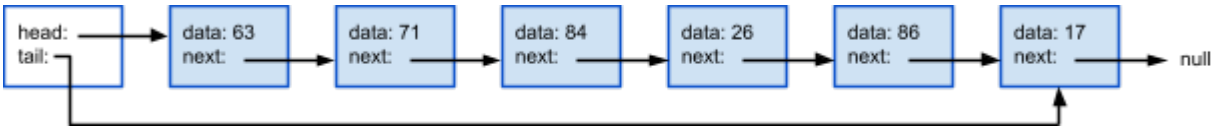
2. ListFindInsertionPosition searches the list from the head toward the current node to find the insertion position. ListFindInsertionPosition returns null, so Node 56 is prepended as the list head.
3. Node 64 is less than node 71. ListFindInsertionPosition returns a pointer to the node before node 71, or node 56. Then, node 64 is inserted after node 56.
4. The insertion position for node 87 is after node 71. Node 87 is already in the correct position and is not moved.
5. Although node 74 is only moved back one position, ListFindInsertionPosition compared node 74 with all other nodes' values to find the insertion position.

Figure 6.10.1: 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 6.10.4: Sorting singly-linked lists with insertion sort.

Given 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 times is ListPrepend

called?

- 0
- 1
- 2

3) How many times is ListInsertAfter called?

- 0
- 1
- 2

Algorithm efficiency

The average and worst case runtime of ListInsertionSortSinglyLinked is O(N^2). The best case runtime is O(N), which occurs when the list is sorted in descending order.

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.

©zyBooks 12/08/22 21:53 1361995
John Farrell
COLOSTATECS165WakefieldFall2022

Table 6.10.1: 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 6.10.2: 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 the array. 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 6.10.5: 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.

6.11 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 head of the list 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 set to null, and the list's head and tail pointers both point to the dummy node.

PARTICIPATION ACTIVITY 6.11.1: Singly-linked lists with and without a dummy node.

Animation captions:

1. An empty linked list without a dummy node has null head and tail pointers.
2. An empty linked list with a dummy node has the head and tail pointing to a node with null data.
3. Without the dummy node, a non-empty list's head pointer points to the first list item.
4. With a dummy node, the list's head pointer always points to the dummy node. The dummy node's next pointer points to the first list item.

PARTICIPATION ACTIVITY 6.11.2: 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 6.11.3: 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 pointers are set to point to the dummy node.

List operations such as append, prepend, insert after, and remove after are simpler to implement compared to a linked list without a dummy node, since a special case is removed from each implementation. ListAppend, ListPrepend, and ListInsertAfter 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 6.11.1: Singly-linked list with dummy node: append, prepend, insert after, and remove after operations.

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


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


ListInsertAfter(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
   }
}


ListRemoveAfter(list, curNode) {
   if (curNode is not null and curNode→next is not
null) {
      sucNode = curNode→next→next
      curNode→next = sucNode

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


©zyBooks 12/08/22 21:53 1361995
John Farrell
COLOSTATECS165WakefieldFall2022

PARTICIPATION ACTIVITY 6.11.4: 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?

- `ListRemoveAfter(dataList, null)`
- `ListRemoveAfter(dataList, dataList→head)`
- `ListRemoveAfter(dataList, dataList→tail)`

2) Which is a requirement of the ListPrepend function?

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

PARTICIPATION ACTIVITY 6.11.5: 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, node 25)
ListInsertAfter(numbersList, lastItem, node 49)
```

- 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)

- ListInsertAfter(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 prev pointer set to null. ListRemove's implementation does not allow removal of the dummy node.

©zyBooks 12/08/22 21:53 1361995
John Farrell
COLOSTATECS165WakefieldFall2022

Figure 6.11.2: Doubly-linked list with dummy node: append, prepend, insert after, and remove operations.

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


ListPrepend(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

   // Set prev on former first node
   if (firstNode is not null) {
      firstNode→prev = newNode
   }
}


ListInsertAfter(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
   }
}


ListRemove(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
```

```
            sucNode→prev = predNode
        }

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

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


PARTICIPATION ACTIVITY 6.11.6: Doubly-linked list with dummy node.

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

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

3) `ListInsertAfter(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 2 dummy nodes: one at the head and the other at the tail. Doing so removes additional conditionals and further simplifies the implementation of most methods.

PARTICIPATION ACTIVITY 6.11.7: Doubly-linked list append and prepend with 2 dummy nodes.

Animation content:

undefined

Animation captions:

1. A list with 2 dummy nodes is initialized such that the list's head and tail point to 2 distinct nodes. Data is null for both nodes.
2. Prepending inserts after the head. The list head's next pointer is never null, even when the list is empty, because of the dummy node at the tail.
3. Appending inserts before the tail, since the list's tail pointer always points to the dummy node.

Figure 6.11.3: Doubly-linked list with 2 dummy nodes: insert after and remove operations.

```
ListInsertAfter(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
}

ListRemove(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 ListInsertAfter and ListRemove

The if statement at the beginning of ListInsertAfter may be removed in favor of having a precondition that curNode cannot point to the dummy tail node. Likewise, ListRemove can remove the if statement and have a precondition that curNode cannot point to either dummy node. If such preconditions are met, neither function requires any if statements.

PARTICIPATION ACTIVITY 6.11.8: Comparing a doubly-linked list with 1 dummy node vs. 2 dummy nodes.

For each question, assume 2 list types are available: a doubly-linked list with 1 dummy node at the list's head, and a doubly-linked list with 2 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 1 dummy node
   - a list with 2 dummy nodes
   - either a list with 1 dummy node or a list with 2 dummy nodes

2) list⇢tail may be null in _____.
   - a list with 1 dummy node
   - a list with 2 dummy nodes
   - neither list type

3) list⇢head⇢next is always non-null in _____.
   - a list with 1 dummy node
   - a list with 2 dummy nodes
   - neither list type

6.12 Linked lists: Recursion

Forward traversal

Forward traversal through a linked list can be implemented using a recursive function that takes a node as an argument. 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 searches the entire list by calling ListTraverseRecursive on the list's head.

PARTICIPATION ACTIVITY 6.12.1: Recursive forward traversal.

Animation content:

`undefined`

Animation captions:

1. ListTraverse begins traversal by calling the recursive function, ListTraverseRecursive, on the list's head.
2. The recursive function visits the node and calls itself for the next node.
3. Nodes 19 is visited and an additional recursive call visits node 41. The last recursive call encounters a null node and stops.

PARTICIPATION ACTIVITY 6.12.2: 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 are made?
   - ○ 9
   - ○ 10
   - ○ 11

PARTICIPATION ACTIVITY 6.12.3: 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 1 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 6.12.1: 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 6.12.4: Searching a linked list with 10 nodes.

Suppose a linked list has 10 nodes.

1) When more than 1 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 is made first, the list is traversed in reverse order.

PARTICIPATION ACTIVITY 6.12.5: Recursive reverse traversal.

Animation content:

undefined

Animation captions:

1. ListTraverseReverse is called to traverse the list. Much like a forward traversal, ListTraverseReverseRecursive is called for the list's head.
2. The recursive call on node 19 is made before visiting node 23.
3. Similarly, the recursive call on node 41 is made before visiting node 19, and the recursive call on null is made before visiting node 41.
4. The recursive call with the null node argument takes no action and is the first to return.
5. Execution returns to the line after the ListTraverseReverseRecursive(null) call. The node argument then points to node 41, which is the first node visited.
6. As the recursive calls complete, the remaining nodes are visited in reverse order. The last ListTraverseReverseRecursive call returns to ListTraverseReverse.
7. The entire list has been visited in reverse order.

PARTICIPATION ACTIVITY 6.12.6: 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

6.13 Linked list search

Given a key, a ***search*** algorithm returns the first node whose data matches that key, or returns null if a matching node was not found. A simple linked list search algorithm checks the current node (initially the list's head node), returning that node if a match, else pointing the current node to 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 6.13.1: Singly-linked list: Searching.

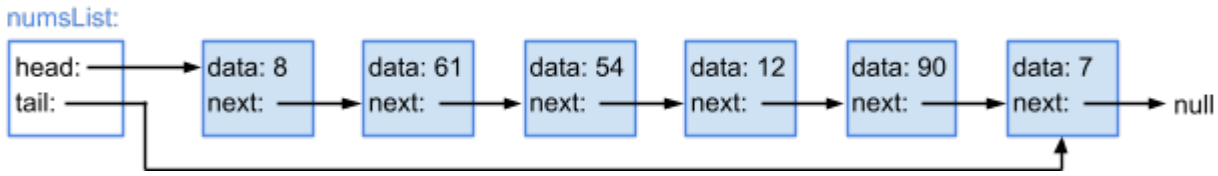
Animation content:

undefined

Animation captions:

1. Search starts at list's head node. If node's data matches key, matching node is returned.
2. If no matching node is found, null is returned.

PARTICIPATION ACTIVITY 6.13.2: ListSearch algorithm execution.



1) How many nodes will ListSearch visit when searching for 54?

Check Show answer

2) How many nodes will ListSearch visit when searching for 48?

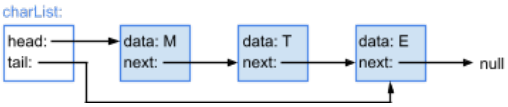
Check Show answer

3) What value does ListSearch return if the search key is not found?

Check Show answer

PARTICIPATION ACTIVITY 6.13.3: Searching a linked-list.

1) ListSearch(charList, E) first assigns curNode to _____.



- Node M
- Node T
- Node E

2) For ListSearch(captainList, Sisko), after checking node Riker, to which node is curNode pointed?

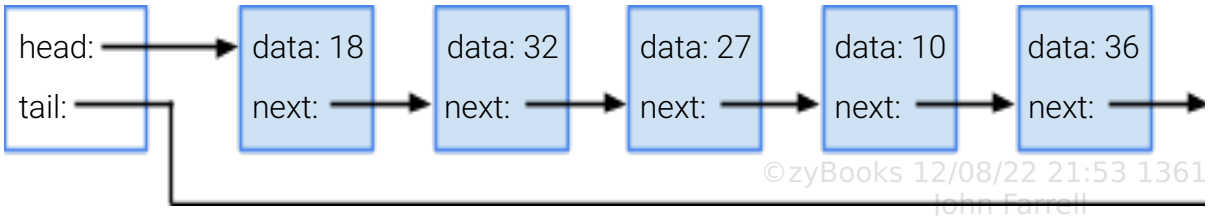
- node Riker
- node Kirk

CHALLENGE ACTIVITY 6.13.1: Linked list search.

422352.2723990.qx3zqy7

Start

numList:



ListSearch(numList, 36) points the current pointer to node Ex: 9 after checking node 18

ListSearch(numList, 36) will make comparisons.

1 2

Check Next

6.14 List data structure

A common approach for implementing a linked list is using two data structures:

1. List data structure: A ***list data structure*** is a data structure containing the list's head and tail, and may also include additional information, such as the list's size.
2. List node data structure: The list node data structure maintains the data for each list element, including the element's data and pointers to the other list element.

A list data structure is not required to implement a linked list, but offers a convenient way to store the list's head and tail. When using a list data structure, functions that operate on a list can use a single parameter for the list's data structure to manage the list.

A linked list can also be implemented without using a list data structure, which minimally requires using separate list node pointer variables to keep track of the list's head.

PARTICIPATION ACTIVITY 6.14.1: Linked lists can be stored with or without a list data structure.

Animation content:

`undefined`

Animation captions:

1. A linked list can be maintained without a list data structure, but a pointer to the head and tail of the list must be stored elsewhere, often as local variables.
2. A list data structure stores both the head and tail pointers in one object.

PARTICIPATION ACTIVITY 6.14.2: Linked list data structure.

1) A linked list must have a list data structure.

- True
- False

2) A list data structure can have additional information besides the head and tail pointers.

- True
- False

3) A linked list has O(n) space complexity, whether a list data structure is used or not.

- True
- False

6.15 List: LinkedList

Linked list container

The ***List*** interface defined within the Java Collections Framework defines a Collection of ordered elements, i.e., a sequence. The List interface supports methods for adding, modifying, and removing elements.

A LinkedList is one of several types that implement the List interface. The LinkedList type is an ADT implemented as a generic class that supports different types of elements. A LinkedList can be declared and created as `LinkedList<T> linkedList = new LinkedList<T>();` where T represents the LinkedList's type, such as Integer or String. The statement `import java.util.LinkedList;` enables use of a LinkedList within a program.

A LinkedList supports insertion of elements either at the end of the list or at a specified index. If an index is not provided, as in `authorList.add("Martin");`, the add() method adds the element at the end of the list. If an index is specified, as in `authorList.add(0, "Rowling");`, the element is inserted at the specified index, with the list element previously located at the specified index appearing after the inserted element.

PARTICIPATION ACTIVITY 6.15.1: LinkedList: add() method adds elements to end of list or at specified index.

Animation captions:

1. The add() method adds an element, such as a String, to the end of the list.

2. Using the add() method with an index inserts an element at the specified index.

PARTICIPATION ACTIVITY 6.15.2: LinkedList add() method.

Given the following code that creates and initializes a LinkedList:


```
LinkedList<String> wordsFromFile = new LinkedList<String>();
wordsFromFile.add("The");
wordsFromFile.add("fowl");
wordsFromFile.add("is");
wordsFromFile.add("the");
wordsFromFile.add("term");
```

1) At what index does the following statement insert the word "end"? Enter a number.

```
wordsFromFile.add("end");
```

Check Show answer

2) Given the original list initialization above, how many elements does wordsFromFile contain after the following statement? Enter a number.

```
wordsFromFile.add(4,
"big");
```

Check Show answer

3) Write a statement to insert the word "not" between the elements "is" and "the".

```
wordsFromFile.add(
);
```

Check Show answer

Common LinkedList methods

The get() method returns the element at the specified index. To access the specified list element, the get() method will start at the first list element and then traverse the list element by element until the specified index is reached. Ex: `playerName.get(3)` starts at the element located at index 0, moves to the element at index 1, then moves to index 2, and finally moves to index 3 returning that element.

The set() method replaces the element at the specified location with a new element.

PARTICIPATION ACTIVITY 6.15.3: LinkedList: get() returns elements at specified location and set() replaces element at specified location.

Animation captions:

1. The get() method searches through the LinkedList to return the element at the specified index.
2. The set() method replaces the element at the specified position.

The remove() method removes the element at the specified index.

PARTICIPATION ACTIVITY 6.15.4: LinkedList: remove() method removes element from specified position.

Animation captions:

1. The remove() method removes the element at the specified position.

The LinkedList class implements several methods, defined in the List interface, for getting information about a list and accessing and modifying a list's elements.

Table 6.15.1: Common LinkedList methods.

get()	`get(index)` Returns element at specified index.	`// Assume List is: 5, 10, 11` `exList.get(2);  // Returns 11`

set()	`set(index, newElement)`
	
	Replaces element at specified index with newElement. Returns element previously at specified index.
		`// Assume List is: 5, 8, 11`
		`exList.set(0, new Integer(3));`
		`// Returns 5`
		`// List is now: 3, 8, 11`

add()	`add(newElement)`
	
	Adds newElement to the end of the List. List's size is increased by one.
	
	`add(index, newElement)`
	
	Adds newElement to the List at the specified index. Indices of the elements previously at that specified index and higher are increased by one. List's size is increased by one.
		`// Assume List is empty`
		`exList.add(new Integer(7));`
		`// List is: 7`
		`exList.add(14);`
		`// List is: 7, 14`
		
		`exList.add(0, new Integer(21));`
		`// List is: 21, 7, 14`
		`exList.add(2, 9);`
		`// List is: 21, 7, 9, 14`

size()	`size()`
	
	Returns the number of elements in the List.
		`// Assume List is empty`
		`exList.size(); // Returns 0`
		
		`exList.add(8);`
		`exList.add(22);`
		`// List is now: 8, 22`
		
		`exList.size(); // Returns 2`

	`remove(index)` Removes element at specified index. Indices for elements from higher positions are	

remove()	decreased by one. List size is decreased by one. Returns reference to element removed from List.
	
	`remove(existingElement)`
	
	Removes the first occurrence of an element which is equal to existingElement. Indices for elements from higher positions are decreased by one. List size is decreased by one. Returns true if specified element was found and removed.
		`// Assume List is: 22, 8, 4, 4`
		`exList.remove(1); // Returns 8`
		`// List is now: 22, 4, 4`
		
		`exList.remove(5); // Throws exception`
		`// List is still: 22, 4, 4`
		
		`exList.remove(2); // Returns 4`
		`// List is now: 22, 4`

PARTICIPATION ACTIVITY 6.15.5: Using LinkedList's get(), set(), and remove() methods.

Given the following code that creates and initializes a LinkedList:

```
LinkedList<Double> accelerometerValues = new LinkedList<Double>();
accelerometerValues.add(9.8);
accelerometerValues.add(10.2);
accelerometerValues.add(15.4);
```

1) Complete the statement to print the first list element.

```
System.out.println(accelerometerValues.
                   );
```

Check Show answer

2) Complete the statement to assign currentValue with the element at index 2.

```
currentValue =
accelerometerValues.
                   ;
```

Check Show answer

3) Complete the statement to update the element at index 1 to 10.6.

```
accelerometerValues.
                    ;
```


Check Show answer

4) Write a statement to remove the value 9.8 from the list.

```
accelerometerValues.
                    ;
```

Check Show answer

Iterating through a list

LinkedList methods with index parameters, such as get() or set(), cause the list to be traversed from the first element to the specified element each time the method is called. Thus, using the LinkedLists' get() or set() methods within a loop that iterates through all list elements is inefficient.

PARTICIPATION ACTIVITY 6.15.6: Using get() within a loop that iterates through all LinkedList elements is inefficient.

Animation captions:

1. The get() method always traverses the list sequentially from the first element to the desired element.

Efficient iteration through a LinkedList necessitates keeping track of the current position in the loop without using an index. A ***ListIterator*** is an object that points to a location in a List and provides methods to access an element and advance the ListIterator to the next position in the list.

LinkedList's listIterator() method returns a ListIterator object for traversing a list. The statement `import java.util.ListIterator;` enables use of a ListIterator in a program.

PARTICIPATION ACTIVITY 6.15.7: ListIterator provides methods to access elements of a LinkedList.

Animation captions:

1. listIterator() method returns a ListIterator object for iterating through list elements.
2. ListIterator's hasNext() method returns true if the list has more elements. Otherwise, returns false.
3. ListIterator's next() method returns the next element in the list and moves the iterator to the next location.

A ListIterator is located between two elements in a list. The next() method returns the next element in the list and moves the ListIterator to the next location. The hasNext() method returns true if there is a next element, and false otherwise. A good practice is to always call hasNext() before calling next() to ensure a list element exists.

The ListIterator's set() method replaces the last element accessed by the iterator, e.g., the element returned by the prior next() call.

PARTICIPATION ACTIVITY 6.15.8: ListIterator's set() method replaces the prior element accessed by the iterator.

Animation captions:

1. ListIterator's set() method replaces the last element accessed by the iterator, such as the prior next() call.
2. All strings in the list have been replaced with uppercase versions.

The ListIterator class implements several methods for traversing a list and accessing and modifying a list's elements.

Table 6.15.2: Common ListIterator methods.

next()	`next()`
	
	Returns the next element in the List and moves the ListIterator after that element.
		`// Assume exList is:  3 4`
		`exIterator = exList.listIterator();`
		`// exList is now:  3 4`
		`// Iterator position: ^`
		
		`exIterator.next(); // Returns 3`
		`// exList is now:  3 4`
		`// Iterator position:  ^`

nextIndex()	`nextIndex()`
	
	Returns the index of the next element.
		`// Assume exList is:  3 4`
		`// Iterator position: ^`
		
		`exIterator.nextIndex(); // Returns 0`
		
		`exIterator.next();  // Returns 3`
		`// exList is now:  3 4`
		`// Iterator position:  ^`
		
		`exIterator.nextIndex(); // Returns 1`

previous()	`previous()`
	
	Returns the previous element in the List and moves the ListIterator before that element.
		`// Assume exList is:  3 4`
		`exIterator = exList.listIterator();`
		`// exList is now:  3 4`
		`// Iterator position: ^`
		
		`exIterator.next();  // Returns 3`
		`// exList is now:  3 4`
		`// Iterator position:  ^`
		
		`exIterator.previous(); // Returns 3`
		`// exList is now:  3 4`
		`// Iterator position: ^`

previousIndex()	`previousIndex()`
	
	Returns the index of the previous element.
		`// Assume exList is:  3 4 5`
		`// Iterator position:  ^`
		
		`exIterator.previousIndex(); // Returns 2`
		
		`exIterator.previous(); // Returns 5`
		`// exList is now:  3 4 5`
		`// Iterator position:  ^`
		
		`exIterator.previousIndex(); // Returns 1`

		`// Assume exList is:  3 4`

hasNext()	`hasNext()`
	
	Returns true if ListIterator has a next element. Otherwise, returns false.
		`// Iterator position: ^`
		`exIterator.hasNext(); // Returns true`
		
		`exIterator.next(); // Returns 4`
		`// exList is now: 3 4`
		`// Iterator position: ^`
		
		`exIterator.hasNext(); // Returns false`

hasPrevious()	`hasPrevious()`
	
	Returns true if ListIterator has a previous element. Otherwise, returns false.
		`// Assume exList is: 3 4`
		`// Iterator position: ^`
		`exIterator.hasPrevious(); // Returns true`
		
		`exIterator.previous(); // Returns '3'`
		`// exList is now: 3 4`
		`// Iterator position: ^`
		
		`exIterator.hasPrevious(); // Returns false`

add()	`add(newElement)`
	
	Adds the newElement between the next and previous elements and moves the ListIterator after newElement.
		`// Assume exList is: 4 5`
		`// Iterator position: ^`
		`exIterator.add(3);`
		`// exList is now: 3 4 5`
		`// Iterator position: ^`
		
		`exIterator.add(11);`
		`// exList is now: 3 11 4 5`
		`// Iterator position: ^`

remove()	`remove()`
	
	Removes the element returned by the prior call to next() or previous(). Fails if used more than once per call to next() or previous(). Fails if add() has already been called since the last call to next() or previous().
		`// Assume exList is: 3 4 5`
		`// Iterator position: ^`
		`exIterator.next(); // Returns 4`
		`// exList is now: 3 4 5`
		`// Iterator position: ^`
		
		`exIterator.remove(); // Removes 4`
		`// exList is now: 3 5`
		`// Iterator position: ^`
		
		`exIterator.previous(); // Returns 3`
		`// exList is now: 3 5`
		`// Iterator position: ^`
		`exIterator.remove(); // Removes 3`
		`// exList is now: 5`
		`// Iterator position: ^`

		`// Assume exList is: 3 4 5`

set()	`set(newElement)`
	
	Replaces the element returned by the prior call next() or previous() with newElement.
		`// Assume exList is:  3 4 5`
		`// Iterator position:  ^`
		`exIterator.next();  // Returns 4`
		`// exList is now:  3 4 5`
		`// Iterator position:  ^`
		
		`exIterator.set(44);`
		`// exList is now:  3 44 5`
		`// Iterator position:  ^`
		`exIterator.set(21);`
		`// exList is now:  3 21 5`
		`// Iterator position:  ^`
		
		`exIterator.previous(); // Returns 21`
		`// exList is now:  3 21 5`
		`// Iterator position:  ^`
		
		`exIterator.set(15);`
		`// exList is now:  3 15 5`
		`// Iterator position:  ^`

PARTICIPATION ACTIVITY 6.15.9: Using ListIterator's next(), hasNext(), and set() methods to traverse and modify a List.

Answer the questions given the following code that creates and initializes a LinkedList and a ListIterator. For every question, assume that the ListIterator is located before the first list element (i.e., the starting position).

```
LinkedList<Integer> numbersList = new LinkedList<Integer>();
ListIterator<Integer> numberIterator;

numbersList.add(3);
numbersList.add(1);
numbersList.add(4);

numberIterator = numbersList.listIterator();
```

1) What does numberIterator.hasNext() return the first time?

Check Show answer

2) What value does numberIterator.hasNext() return *after* three calls to

numberIterator.next().

Check Show answer

3) Given the original list initialization above, what is the value of numVal after executing the following statements?

```
numberIterator.next();
numVal =
numberIterator.next();
```

Check Show answer

4) Complete the code to replace the next list element with newDigit.

```
numberIterator.next(); // Necessary!
numberIterator.
                    ;
```

Check Show answer

CHALLENGE ACTIVITY 6.15.1: List: LinkedList.

422352.2723990.qx3zqy7

Start

Type the program's output

```
import java.util.LinkedList;

public class Letters {
   public static void main(String[] args) {
      LinkedList<Character> letters = new LinkedList<Character>();
      Character letter;
      int i;

      letters.add('C');
      letters.add('D');
      letters.add('E');
      letters.add('F');

      letters.add(1, 'G');
      letters.set(4, 'H');
      letters.remove(3);

      for (i = 0; i < letters.size(); ++i) {
         letter = letters.get(i);
         System.out.print(letter);
      }
      System.out.println();
```


CGDH

LinkedList vs. ArrayList

LinkedList and ArrayList are ADTs implementing the List interface. Although both LinkedList and ArrayList implement a List, a programmer should select the implementation that is appropriate for the intended task. A LinkedList typically provides faster element insertion and removal at the list's ends (and middle if using ListIterator), whereas an ArrayList offers faster positional access with indices. In this material, we use the LinkedList class, but the examples can be modified to use ArrayList.

Exploring further:

- LinkedList from Oracle's Java documentation.
- ListIterator from Oracle's Java documentation.
- ArrayList from Oracle's Java documentation.
- Java Collections Framework Overview from Oracle's Java documentation.

6.16 A first linked list

A common use of objects and references is to create a list of items such that an item can be efficiently inserted somewhere in the middle of the list, without the shifting of later items as required for an ArrayList. The following program illustrates how such a list can be created. A class is defined to represent each list item, known as a ***list node***. A node is comprised of the data to be stored in each list item, in this case just one int, and a reference to the next node in the list. A special node named head is created to represent the front of the list, after which regular items can be inserted.

©zyBooks 12/08/22 21:53 1361995
John Farrell
COLOSTATECS165WakefieldFall2022

Figure 6.16.1: A basic example to introduce linked lists.

IntNode.java

```
public class IntNode {
   private int dataVal;         // Node data
   private IntNode nextNodePtr; // Reference to the next node

   public IntNode() {
      dataVal = 0;
      nextNodePtr = null;
   }

   // Constructor
   public IntNode(int dataInit) {
      this.dataVal = dataInit;
      this.nextNodePtr = null;
   }

   // Constructor
   public IntNode(int dataInit, IntNode nextLoc) {
      this.dataVal = dataInit;
      this.nextNodePtr = nextLoc;
   }

   /* Insert node after this node.
    Before: this -- next
    After:  this -- node -- next
    */
   public void insertAfter(IntNode nodeLoc) {
      IntNode tmpNext;

      tmpNext = this.nextNodePtr;
      this.nextNodePtr = nodeLoc;
      nodeLoc.nextNodePtr = tmpNext;
   }

   // Get location pointed by nextNodePtr
   public IntNode getNext() {
      return this.nextNodePtr;
   }

   public void printNodeData() {
      System.out.println(this.dataVal);
   }
}
```


CustomLinkedList.java

```
public class CustomLinkedList {
   public static void main(String[] args) {
      IntNode headObj;  // Create IntNode reference variables
      IntNode nodeObj1;
      IntNode nodeObj2;
      IntNode nodeObj3;
```

```
        IntNode currObj;

        // Front of nodes list
        headObj = new IntNode(-1);

        // Insert more nodes
        nodeObj1 = new IntNode(555);
        headObj.insertAfter(nodeObj1);

        nodeObj2 = new IntNode(999);
        nodeObj1.insertAfter(nodeObj2);

        nodeObj3 = new IntNode(777);
        nodeObj1.insertAfter(nodeObj3);

        // Print linked list
        currObj = headObj;
        while (currObj != null) {
           currObj.printNodeData();
           currObj = currObj.getNext();
        }
     }
  }
```

```
-1
555
777
999
```


PARTICIPATION ACTIVITY 6.16.1: Inserting nodes into a basic linked list.

Animation captions:

1. The headObj pointer points to a special node that represents the front of the list. When the list is first created, no list items exists, so the head node's nextNodePtr pointer is null.
2. To insert a node in the list, the new node nodeObj1 is first created with the value 555.
3. To insert the new node, tmpNext is pointed to the head node's next node, the head node's nextNodePtr is pointed to the new node, and the new node's nextNodePtr is pointed to tmpNext.
4. A second node nodeObj2 with the value 999 is inserted at the end of the list, and a third node nodeObj3 with the value 777 is created.
5. To insert nodeObj3 after nodeObj1, tmpNext is pointed to the nodeObj1's next node, the nodeObj1's nextNodePtr is pointed to the nodeObj3, and nodeObj3's nextNodePtr is pointed to tmpNext.

PARTICIPATION ACTIVITY 6.16.2: A first linked list.

Some questions refer to the above linked list code and animation.

1) A linked list has what key advantage over a sequential storage approach like an array or ArrayList?

- An item can be inserted somewhere in the middle of the list without having to shift all subsequent items.
- Uses less memory overall.
- Can store items other than int variables.

2) What is the purpose of a list's head node?

- Stores the first item in the list.
- Provides a reference to the first item's node in the list, if such an item exists.
- Stores all the data of the list.

3) After the above list is done having items inserted, at what memory address is the last list item's node located?

- 80
- 82
- 84
- 86

4) After the above list has items inserted as above, if a fourth item was inserted at the front of the list, what would happen to the location of node1?

- Changes from 84 to 86.
- Changes from 84 to 82.
- Stays at 84.

In contrast to the above program that declares one reference variable for each item allocated by the new operator, a program commonly declares just one or a few variables to manage a large number of items allocated using the new operator. The following example replaces the above main() method, showing how just two reference variables, currObj and lastObj, can manage 20 allocated items in the list.

©zyBooks 12/08/22 21:53 1361995
John Farrell
COLOSTATECS165WakefieldFall2022

Figure 6.16.2: Managing many new items using just a few reference variables.

IntNode.java

```
public class IntNode {
   private int dataVal;         // Node data
   private IntNode nextNodePtr; // Reference to the next node

   public IntNode() {
      dataVal = 0;
      nextNodePtr = null;
   }

   // Constructor
   public IntNode(int dataInit) {
      this.dataVal = dataInit;
      this.nextNodePtr = null;
   }

   // Constructor
   public IntNode(int dataInit, IntNode nextLoc) {
      this.dataVal = dataInit;
      this.nextNodePtr = nextLoc;
   }

   /* Insert node after this node.
    Before: this -- next
    After:  this -- node -- next
    */
   public void insertAfter(IntNode nodeLoc) {
      IntNode tmpNext;

      tmpNext = this.nextNodePtr;
      this.nextNodePtr = nodeLoc;
      nodeLoc.nextNodePtr = tmpNext;
   }

   // Get location pointed by nextNodePtr
   public IntNode getNext() {
      return this.nextNodePtr;
   }

   public void printNodeData() {
      System.out.println(this.dataVal);
   }
}
```


CustomLinkedList.java

```
public class CustomLinkedList {
   public static void main(String[] args) {
      IntNode headObj; // Create IntNode reference variables
      IntNode currObj;
      IntNode lastObj;
```

```
      IntNode lastObj;
      int i;           // Loop index

      headObj = new IntNode(-1); // Front of nodes list
      lastObj = headObj;

      for (i = 0; i < 20; ++i) { // Append 20 rand nums
         int rand = (int)(Math.random() * 100000); // random int
(0-99999)
         currObj = new IntNode(rand);

         lastObj.insertAfter(currObj); // Append curr
         lastObj = currObj;
      }

      currObj = headObj; // Print the list
      while (currObj != null) {
         currObj.printNodeData();
         currObj = currObj.getNext();
      }
   }
}
```



```
-1
40271
6951
29273
86846
64952
65650
98162
51229
30690
61008
17489
87486
24318
44035
32368
10906
75441
88659
65688
18443
```

zyDE 6.16.1: Managing a linked list.

Finish the program so that it finds and prints the smallest value in the linked list.

Current file: IntNode.java ▾ ault template...

Run

```
1
2 public class IntNode {
3    private int dataVal;         // Node da
4    private IntNode nextNodePtr; // Referen
5
```

Normally, a linl t would be maintained by member methods of another class, such as IntList. Private fields c class might include the list head (a list node allocated by the list class constructor), tl size, and the list tail (the last node in the list). Public member methods might include insertA nsert a new node after the given node), pushBack (insert a new node after the last node), pus t (insert a new node at the front of the list, just after the head), deleteNode (deletes the nc om the list), etc.

```
6     public IntNode() {
7        dataVal = 0;
8        nextNodePtr = null;
9     }
10
11    // Constructor
12    public IntNode(int dataInit) {
13       this.dataVal = dataInit;
14       this.nextNodePtr = null;
15    }
16
17    // Constructor
```


Exploring fu

- More on linked lists from Oracle's Java tutorials

CHALLENGE ACTIVITY 6.16.1: Enter the output of the program using the linked list.

422352.2723990.qx3zqy7

Start

Type the program's output

CallPlaylistSong.java | PlaylistSong.java

```
public class CallPlaylistSong {
   public static void main(String[] args) {
      PlaylistSong headObj = null;
      PlaylistSong firstSong = null;
      PlaylistSong secondSong = null;
      PlaylistSong thirdSong = null;
      PlaylistSong currObj = null;

      headObj = new PlaylistSong("head");

      firstSong = new PlaylistSong("Egmont");
      headObj.InsertAfter(firstSong);

      secondSong = new PlaylistSong("Fidelio");
      firstSong.InsertAfter(secondSong);

      thirdSong = new PlaylistSong("Canon");
      secondSong.InsertAfter(thirdSong);

      currObj = headObj;

      while (currObj != null) {
         currObj.PrintNodeData();
         currObj = currObj.GetNext();
      }
   }
}
```

```
head
Egmont
Fidelio
Canon
```


1 2

Check Next

CHALLENGE ACTIVITY 6.16.2: Linked list negative values counting.

Assign negativeCntr with the number of negative values in the linked list.

422352.2723990.qx3zqy7

```
1  // ===== Code from file IntNode.java =====
2  public class IntNode {
3     private int dataVal;
4     private IntNode nextNodePtr;
5
6     public IntNode(int dataInit, IntNode nextLoc) {
7        this.dataVal = dataInit;
8        this.nextNodePtr = nextLoc;
9     }
10
11    public IntNode(int dataInit) {
12       this.dataVal = dataInit;
13       this.nextNodePtr = null;
14    }
15
16    /* Insert node after this node.
17    * Before: this -- next
```

Run

6.17 LAB: Grocery shopping list (LinkedList)

Given a `ListItem` class, complete main() using the built-in LinkedList type to create a linked list called **shoppingList**. The program should read items from input (ending with -1), adding each item to shoppingList, and output each item in shoppingList using the printNodeData() method.

Ex. If the input is:

```
milk
bread
eggs
waffles
cereal
-1
```


the output is:

```
milk
bread
eggs
waffles
cereal
```

422352.2723990.qx3zqy7

LAB ACTIVITY 6.17.1: LAB: Grocery shopping list (LinkedList) 0 / 10

Current file: **ShoppingList.java** Load default template...

```
1  import java.util.Scanner;
2  import java.util.LinkedList;
3
4  public class ShoppingList {
5     public static void main (String[] args) {
6        Scanner scnr = new Scanner(System.in);
7
8        // TODO: Declare a LinkedList called shoppingList of type ListItem
9
10       String item;
11
12       // TODO: Scan inputs (items) and add them to the shoppingList LinkedList
13       //       Read inputs until a -1 is input
14
15
16       // TODO: Print the shoppingList LinkedList using the printNodeData() method
17
```


Develop mode Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → ShoppingList.java (Your program) →

Program output displayed here

Coding trail of your work What is this?

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

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

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
```


422352.2723990.qx3zqy7

LAB ACTIVITY | 6.18.1: LAB: Inventory (linked lists: insert at the front of a list) 0 / 10

File is marked as read only Current file: Inventory.java

```
1  import java.util.Scanner;
2
3  public class Inventory {
4     public static void main (String[] args) {
5        Scanner scnr = new Scanner(System.in);
6
7        InventoryNode headNode;
8        InventoryNode currNode;
9        InventoryNode lastNode;
10
11       String item;
12       int numberOfItems;
13       int i;
14
15       // Front of nodes list
16       headNode = new InventoryNode();
17       lastNode = headNode;
```


Develop mode | **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → **Inventory.java** (Your program) →

Program output displayed here

Coding trail of your work What is this?

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

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

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
```

422352.2723990.qx3zqy7

LAB ACTIVITY 6.19.1: LAB: Grocery shopping list (linked list: inserting at the end of a list)0 / 10

File is marked as read only Current file: **ShoppingList.java**


```
1  import java.util.Scanner;
2
3  public class ShoppingList {
4     public static void main (String[] args) {
5        Scanner scnr = new Scanner(System.in);
6
7        ItemNode headNode;  // Create intNode objects
8        ItemNode currNode;
9        ItemNode lastNode;
10
11       String item;
12       int i;
13
```

Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program Input (from above) → **ShoppingList.java** (Your program) →

Program output displayed here

Coding trail of your work What is this?

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

6.20 LAB: Library book sorting

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?

422352.2723990.qx3zqy7

LAB ACTIVITY 6.20.1: LAB: Library book sorting 0 / 10

Current file: **Library.java** Load default template...

```
1  import java.util.Scanner;
2  import java.io.FileInputStream;
3  import java.io.IOException;
4
5  public class Library {
6
7     public static void fillLibraries(LinkedListLibrary linkedListLibrary, ArrayListLibrary a
8        FileInputStream fileByteStream = null; // File input stream
9        Scanner inFS = null;                   // Scanner object
10       int linkedListOperations = 0;
11       int arrayListOperations = 0;
12
13       BookNode currNode;
14       Book tempBook;
15
16       String bookTitle;
17       String bookAuthor;
```

Develop mode **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program Input (from above) → **Library.java** (Your program) → O

Program output displayed here

Coding trail of your work What is this?

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


6.21 Lab 9 - Singly Linked List

Module 5: Lab 9 - Singly Linked List

Implementing a Singly Linked List

The purpose of the lab is to help give you a conceptual understanding of how a singly linked list works by actually implementing one in Java.

The following files are included in this lab:

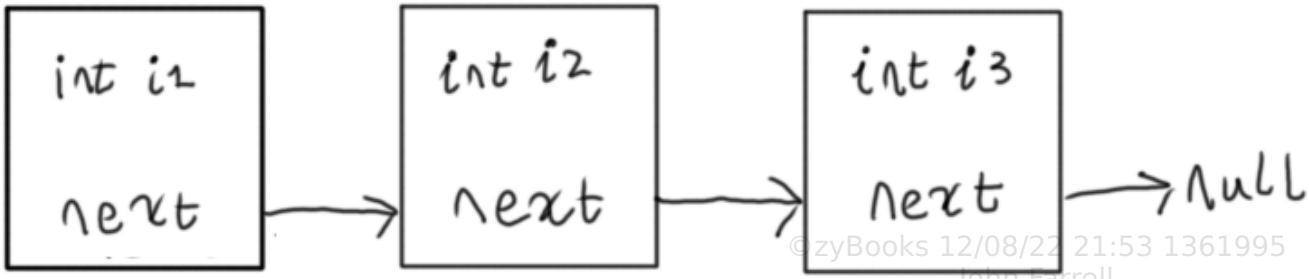
L9
├── MyLinkedList.java (The file you will modify.)
├── MiniList.java (The interface you will implement.)
├── LLTest.java (*The read-only, **main** in zyLabs.)

Use L9.jar to do this in another environment.

The Node Subclass

The most important component, or the building block, of a linked list is a **node**. A **node** has two components: the data (in this lab it will be an int) and the next/previous pointer (in a singly linked list the node only has a next pointer). The next pointer is of type node and it has a value of the next node object or `null` if there is none.

A singly linked list of integers looks like this:



This linked list has a size of three and each node contains an integer with an arbitrary name and value. The data item for the first node is i1 its node pointer called next is equal to the node with data item i2. The last node containing i3 points to null which indicates there is nothing after it.

The List Interface

The List interface is an interface provided by Oracle with the documentation here. Java has its own implementation of a linked list. You will be making your own implementation in this lab which will have most of the methods from the List interface (minus the ones which are currently out of the scope). For this, we have a miniature version of the list interface for you to play with.

What to Complete

You will need to complete all of the methods in the the given MiniList interface. The documentation for these methods match the ones in Oracle's documentation for a linked list. You will also need to write the node class which should contain two member variables: data and next.

Testing

Use LLTest.java to test and run your code. You have been given some test cases, however we would encourage you to write some of your own (you could even bust out some unit tests if you're feeling adventurous). *Note: LLTest.java is read-only on zyLabs since it is being used to test your code when you submit, but you can do your own testing in another environment.*

422352.2723990.qx3zqy7

LAB ACTIVITY | 6.21.1: Lab 9 - Singly Linked List | 0 / 7

Downloadable files

`MyLinkedList.java` , `MiniList.java` , and `LLTest.java` **Download**

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

```
1  /** Linked List Lab
2   * Made by Toby Patterson 5/29/2020
3   * For CS165 at CSU
4   */
5
6  public class MyLinkedList implements MiniList<Integer>{
7      /* Private member variables that you need to declare:
8       ** The head pointer
9       ** The tail pointer
10      */
11     Node head;
12     Node tail;
13
14     public class Node {
15         // declare member variables (data and next)
16          int data;
17          Node next;
```


Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → MyLinkedList.java (Your program) →

Program output displayed here

Coding trail of your work What is this?

```
9/27 T--------- W---------0-------0,0 R---- min:118
```


6.22 Lab 10 - Doubly Linked List

Module 5: Lab 10 - Doubly Linked List

Doubly Linked List

A doubly linked list is very similar to the singly linked list, the only difference being each node in the list has not only a next pointer, but also a previous pointer. This means that you can traverse the list forward and backwards. It also increases the complexity of add and remove operations because of the additional pointers to keep track of. The diagram below displays this:

The following files are included in this lab:

L10

- MyLinkedList.java (The file you will modify.)
- MiniList.java (The interface you will implement.)
- LLTest.java (*The read-only, **main** in zyLabs.)

Note: these files are named the same as L9, but they have been modified.

Download L10.jar to run the code in another environment.

Linked Lists and Generics

Just like with ArrayLists, you can make a linked list generic, allowing it to hold any type. If you look at Oracle's implementation of the linked list, they do it this way. In the singly linked list lab you just did, MyLinkedList implemented a generic interface, but was written to only hold integers. The doubly linked list you make should be able to hold any object.

What to Complete

Implement a doubly linked list making sure to complete all of the methods in the MiniList interface. Now you will need to test for data types other than integers.

Submission

Submit your code in Submit Mode here in zyLabs.

422352.2723990.qx3zqy7

LAB ACTIVITY | 6.22.1: Lab 10 - Doubly Linked List | 0 / 7

Downloadable files

`MyLinkedList.java` , `MiniList.java` , and `LLTest.java` **Download**

Current file: **MyLinkedList.java** Load default template...

```
1  /** Linked List Lab
2   * Made by Toby Patterson 5/31/2020
3   * For CS165 at CSU
4   */
5
6  public class MyLinkedList<E> implements MiniList<E>{
7      /* Private member variables that you need to declare:
8       ** The head pointer
9       ** The tail pointer
10      */
11
12     public class Node {
13         // declare member variables (data, prev and next)
14
15         // finish these constructors
16         public Node(E data, Node prev, Node next) {}
17         public Node(E data) {} // HINT: use this() with next = prev = null
```

Develop mode | **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program Input (from above) → **MyLinkedList.java** (Your program) →

Program output displayed here

Coding trail of your work What is this?

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

6.23 LAB: Find max in list

This section's content is not available for print.

©zyBooks 12/08/22 21:53 1361995
John Farrell
COLOSTATECS165WakefieldFall2022