4.1 List abstract data type (ADT)

List abstract data type

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

PARTICIPATION ACTIVITY 4.1.1: List ADT.

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

ages: 55 66

Print result: 55, 88, 66

Print result: 55, 66

Animation content:

Static figure:

Pictured are some code, a list that results from the code, and program print outputs that also result from the code.

Begin Pseudocode:
ages = new List()
Append(ages, 55)
Append(ages, 88)
Append(ages, 66)
Print(ages)

Remove(ages, 88)
Print(ages)
End Pseudocode.

There is a list named ages containing two integers: 55, 66

There are two print results, the first above the second: "55, 88, 66" and then "55, 66."

Step 1: A new list named "ages" is created. Items can be appended to the list. The items are ordered. Code execution begins on the first line. "ages" label appears to the right. Second line of code executes. List item 55 appears to the right of the ages label. Third line of code executes. List item 88 appears to the right of item 55. Fourth line of code executes. List item 66 appears to the right of item 88. The list "ages" thus is "55, 88, 66"

Step 2: Printing the list prints the items in order. Fifth line of code executes and the print result is shown: "55, 88, 66."

Step 3: Removing an item keeps the remaining items in order. Sixth line of code executes. List item 88 disappears and list item 66 slides over to list item 88's former location. List item 66 is still to the right of list item 55. Last line of code executes and the print result is shown: "55, 66."

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 4.1.2: List ADT.

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

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

Check **Show answer**

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

Remove(list, 3)

Check **Show answer**

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

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

Check **Show answer**

Common list ADT operations

Table 4.1.1: Some common operations for a list ADT.

Operation	Description	Example starting with list: (99, 77)
Append(list, X)	Inserts X at end of list	`Append(list, 44)` list: (99, 77, 44)
Prepend(list, X)	Inserts X at start of list	`Prepend(list, 44)` list: (44, 99, 77)
InsertAfter(list, W, X)	Inserts X after W	`InsertAfter(list, 99, 44)` list: (99, 44, 77)
Remove(list, X)	Removes X	`Remove(list, 77)` list: (99)
Contains(list, X)	Returns true if X is in the list, false otherwise	`Contains(list, 99)` returns true `Contains(list, 22)` returns false
Print(list)	Prints list's items in order	`Print(list)` outputs: 99, 77
Sort(list)	Sorts list's items in ascending order	`Sort(list)` list: (77, 99)
IsEmpty(list)	Returns true if list has no items	`IsEmpty(list)` returns false
GetLength(list)	Returns the number of items in the list	`GetLength(list)` returns 2

PARTICIPATION ACTIVITY 4.1.3: List ADT common operations.

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

- returns 4
- fails

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

- True
- False

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

- True
- False

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

- True
- False

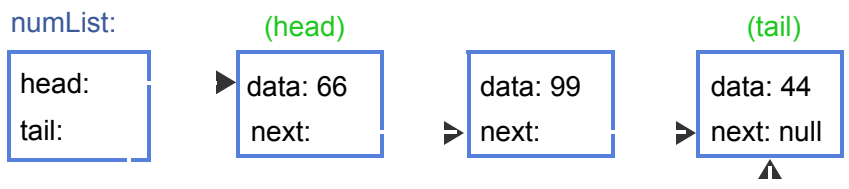
4.2 Singly-linked lists

Singly-linked list data structure

A ***singly-linked list*** is a data structure for implementing a list ADT, where each node has data and a pointer to the next node. The list structure typically has pointers to the list's first node and last node. A singly-linked list's first node is called the ***head***, and the last node the ***tail***. A singly-linked list is a type of ***positional list***: A list where elements contain pointers to the next and/or previous elements in the list.

PARTICIPATION ACTIVITY 4.2.1: Singly-linked list: Each node points to the next node.

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



Animation content:

Static figure: A set of instructions:
1. Create new list: numList
2. Append 99
3. Append 44
4. Prepend 66

A singly-linked list named numList with three nodes. numList's head pointer points to node 66. This node's next pointer points to node 99. This node's next pointer points to node 44. This node's next pointer is null. numList's tail pointer points to node 44.

Step 1: A linked list data structure has head and tail pointers, each initially null. The first instruction is highlighted: Create new list: numList. A new list is created with head and tail pointers null.

Step 2: Appending element 99 to the list requires allocation of a singly-linked list node. The node's data is 99, and the next pointer is null. The second instruction is highlighted: Append 99. A new node is created with data: 99 and next: null.

Step 3: To append the node to the list, numList's head and tail pointers are each reassigned to point to node 99.

Step 4: Another append points the last node's next pointer and the list's tail pointer to the new node. The third instruction is highlighted: Append 44. A new node is created with data: 44 and next: null. Node 99's next pointer points at node 44. numList's tail pointer points at node 44.

Step 5: A prepend operation inserts a node at the list's start instead of the end. Two pointers are assigned in the process: numList's head points to node 66 and node 66's next points to node 99. The fourth instruction is highlighted: Prepend 66. A new node is created with data: 66 and its next pointer pointing at node 99. numList's head pointer points at node 66.

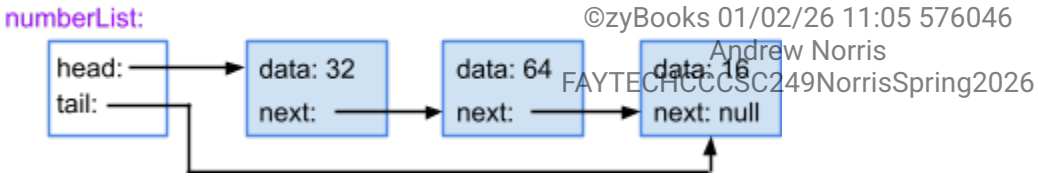
Step 6: The list's first node is called the head. The last node is the tail. Node 66 is labeled head. Node 44 is labeled tail.

Animation captions:

1. A linked list data structure has head and tail pointers, each initially null.
2. Appending element 99 to the list requires allocation of a singly-linked list node. The node's data is 99, and the next pointer is null.
3. To append the node to the list, numList's head and tail pointers are each reassigned to point to node 99.
4. Another append points the last node's next pointer and the list's tail pointer to the new node.
5. A prepend operation inserts a node at the list's start instead of the end. Two pointers are assigned in the process: numList's head points to node 66 and node 66's next points to node 99.
6. The list's first node is called the head. The last node is the tail.

PARTICIPATION ACTIVITY 4.2.2: Singly-linked list data structure.

Consider the following singly-linked list:



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

- null
- 32
- 16

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

- 32
- 64
- 16

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

- node 32
- node 64
- node 16

Implementing a list ADT with a singly-linked list

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

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

PARTICIPATION ACTIVITY 4.2.3: Allocating singly-linked list nodes.

```
node88 = Allocate new singly-linked list node
```

```
node88→data = 88
node88→next = null

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

node88→next = node44
```

Animation content:

Static figure: Begin code:
node88 = Allocate new singly-linked list node
node88⇢data = 88
node88⇢next = null
node44 = Allocate new singly-linked list node
node44⇢data = 44
node44⇢next = null
node44⇢next = node44
End code. Node 88 with data: 88 and next pointer pointing at node 44. Node 44 with data: 44 and next: null
Step 1: Space for a singly-linked list node's two data members, data and next, is allocated. Then data is assigned with 88 and next is assigned with null. The following lines of code are highlighted:
node88 = Allocate new singly-linked list node
node88⇢data = 88
node88⇢next = null
node88 is created with data: 88 and next: null.
Step 2: Node 44 is created similarly. The following lines of code are highlighted:
node44 = Allocate new singly-linked list node
node44⇢data = 44
node44⇢next = null
node44 is created with data: 44 and next: null.
Step 3: Allocated nodes can then be linked by assigning the next pointer. The line of code node44⇢next = node44 is highlighted. node88's next pointer points to node44.

Animation captions:

1. Space for a singly-linked list node's two data members, data and next, is allocated. Then data is assigned with 88 and next is assigned with null.
2. Node 44 is created similarly.
3. Allocated nodes can then be linked by assigning the next pointer.

PARTICIPATION ACTIVITY 4.2.4: Allocating singly-linked list nodes.

Refer to the animation above.

1) node88 is a _____ instance.

- singly-linked list
- singly-linked list node

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

- True
- False

PARTICIPATION ACTIVITY 4.2.5: List ADT.

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

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

Append operation

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

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

PARTICIPATION ACTIVITY 4.2.6: Singly-linked list: Append operation.

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

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

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

Animation content:

Static figure: Begin code:
ListAppend(list, item) {
  newNode = Allocate new singly-linked list node
  newNode⋯→data = item
  newNode⋯→next = null
  ListAppendNode(list, newNode)
}

ListAppendNode(list, newNode) {
  if (list⋯→head == null) { // List empty
    list⋯→head = newNode
    list⋯→tail = newNode
  }
  else {
    list⋯→tail⋯→next = newNode
    list⋯→tail = newNode
  }
}
End code. Two lines of instructions with checkmarks next to them. ListAppend(list, 95).

ListAppend(list, 42). A list with head pointer pointing at node 95. Node 95's next pointer points at node 42. Node 42's next pointer is null. Node 42 is labeled newNode. The list's tail pointer points at node 42.

Step 1: The ListAppend() function takes an integer list item as a parameter. A new node is allocated to hold the item. The instruction ListAppend(list, 95) is highlighted. The following lines of code are highlighted. ListAppend(list, item) {

```
  newNode = Allocate new singly-linked list node
  newNode⇢data = item
  newNode⇢next = null
```


A new node called newNode is created with data:95 and next: null.

Step 2: ListAppend() then calls ListAppendNode() to append the newly allocated node. The list is empty, so both the head and tail pointers are updated. The following lines of code are highlighted: ListAppendNode(list, newNode)

```
}

ListAppendNode(list, newNode) {
  if (list⇢head == null) { // List empty
    list⇢head = newNode
    list⇢tail = newNode
```

Since the list's head is null, the list's head pointer and tail pointer are both assigned to newNode.

Step 3: Appending to a non-empty list adds the new node after the tail node and updates the tail pointer. The instruction ListAppend(list, 42) is highlighted. The following lines of code are highlighted:

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

A new node called newNode is created with data:42 and next: null. The following lines of code are highlighted:

```
  ListAppendNode(list, newNode)
}

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

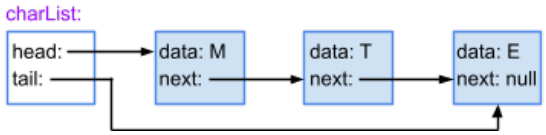
Since the list's head pointer is not null, the node at the list's tail, node 95's next pointer is assigned to newNode. The list's tail pointer is assigned to newNode.

Animation captions:

1. The ListAppend() function takes an integer list item as a parameter. A new node is allocated to hold the item.
2. ListAppend() then calls ListAppendNode() to append the newly allocated node. The list is empty, so both the head and tail pointers are updated.
3. Appending to a non-empty list adds the new node after the tail node and updates the tail pointer.

PARTICIPATION ACTIVITY 4.2.7: Appending a node to a singly-linked list.

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



- M
- T
- E

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

- Head and tail
- Head only
- Tail only

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

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

- `list→tail = newNode`

PARTICIPATION ACTIVITY 4.2.8: Appending a node vs appending a list item.

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

1) Both a list ADT and a singly-linked list have an item-appending function.
   - True
   - False
2) Both a list ADT and a singly-linked list have a node-appending function.
   - True
   - False
3) A list ADT implementation that uses a singly-linked list will have some functions that operate on list items and others that operate on singly-linked list nodes.
   - True
   - False

Prepend operation

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

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

PARTICIPATION ACTIVITY 4.2.9: Singly-linked list: Prepend operation.

```
ListPrepend(list, item) {
```

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

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

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


Animation content:

Static figure:
Begin pseudocode:

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

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

End pseudocode.
Implementation of ListPrepend() and ListPrependNode() functions are shown to the left. On the right are two list operations:

ListPrepend(list, 23)
ListPrepend(list, 17)
Below the code is a linked list with two nodes. The head node has data 17 and the next points to the tail node. The tail node has data 23 and the next pointer is null. The list object's head points to node 17 and the tail to node 23.

Step 1: The ListPrepend() function takes an integer list item as an argument. A new node is allocated to hold the item.
A box for the list data structure exists in the lower left, with the label "list:" above. Two labels exist within: "head: nullptr" and "tail: nullptr".
The list.Prepend(23) statement is highlighted to indicate start of execution. Highlight moves into Prepend() function implementation on the left. Allocation statement in first line executes and a node box appears below with two labels within: "data: 23" and "next: nullptr".

Step 2: ListPrepend() then calls ListPrependNode() to prepend the newly allocated node. The list is empty, so both the head and tail pointers are updated.
Prepend() calls PrependNode(), passing the newly allocated node. The if statement's condition is true, since the list's head is null. The list's head and tail are both assigned with newNode. The text "nullptr" disappears from after the "head:" and "tail:" labels within the list box, each replaced by an arrow that points to newNode.

Step 3: Prepending to a non-empty list first assigns newNode's next pointer with the list's head node.
list.Prepend(17) executes. A new node is allocated with data=17 and next=nullptr. When execution reaches PrependNode(), the if statement's condition is false. The else block executes. Node 17's "next: nullptr" changes to just "next:", and an arrow appears pointing to node 23.

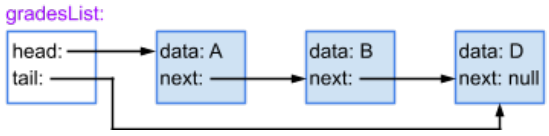
Step 4: Prepending then assigns the list's head pointer with the new node.
The list's head arrow is moved to point to node 17.

Animation captions:

1. The ListPrepend() function takes an integer list item as an argument. A new node is allocated to hold the item.
2. ListPrepend() then calls ListPrependNode() to prepend the newly allocated node. The list is empty, so both the head and tail pointers are updated.
3. Prepending to a non-empty list first assigns newNode's next pointer with the list's head node.
4. Prepending then assigns the list's head pointer with the new node.

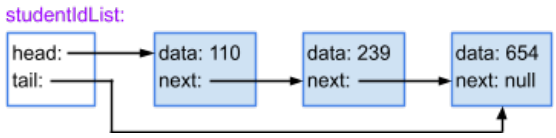
PARTICIPATION ACTIVITY 4.2.10: Prepending a node in a singly-linked list.

1) Prepending C to gradesList updates which pointer?



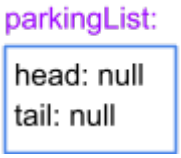
- The list's head pointer
- A's next pointer
- D's next pointer

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



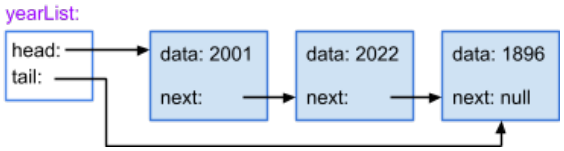
- True
- False

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



- True
- False

4) Prepending node 1999 to yearList executes which statement?



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

CHALLENGE ACTIVITY 4.2.1: Singly-linked lists.

704586.1152092.qx3zqy7

Start

What is numList after the following operations?


```
numList = new List
ListAppend(numList, 22)
ListAppend(numList, 31)
ListAppend(numList, 13)
ListAppend(numList, 92)
```

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

1	2	3	4	5

Check Next

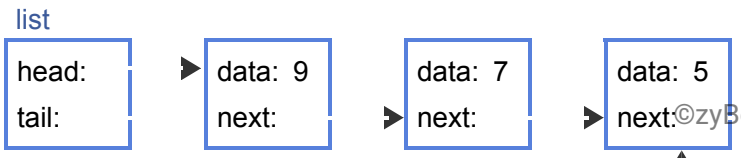
View solution *(Instructors only)*

4.3 Singly-linked lists: Search and insert

Search operation

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

PARTICIPATION ACTIVITY 4.3.1: Singly-linked list search operation.




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

ListSearch(list, 5) // Returns node 5

ListSearch(list, 11) // Returns null

Animation content:

Static figure: Begin code:
ListSearch(list, key) {
  curNode = list⋯→head
  while (curNode is not null) {
    if (curNode⋯→data == key) {
      return curNode
    }
    curNode = curNode⋯→next
  }
  return null
}
End code. Two function calls: ListSearch(list, 5) // Returns node 5
ListSearch(list, 11) // Returns null

A list with head pointer pointing at node 9. Node 9's next pointer points at node 7. Node 7's next pointer points at node 5. Node 5 next pointer is null. The list's tail pointer points at node 5.
Step 1: Search starts at list's head node. If node's data matches key, matching node is returned. The function call ListSearch(list, 5) is highlighted. The lines of code
ListSearch(list, key) {
  curNode = list⋯→head

are highlighted. curNode points to the list's head, which is node 9. The line of code if (curNode⇢data == key) { is highlighted. 9 is not equal to 5. The line of code curNode = curNode⇢next is highlighted. curNode now points at node 7. The line of code if (curNode⇢data == key) { is highlighted. 7 is not equal to 5. The line of code curNode = curNode⇢next is highlighted. curNode now points to 5. The line of code if (curNode⇢data == key) { is highlighted. 5 is equal to 5. The line of code return curNode is highlighted. ListSearch(list, 5) returns node 5.

Step 2: If no matching node is found, null is returned. The function call ListSearch(list, 11) is highlighted. The lines of code

```
ListSearch(list, key) {
  curNode = list⇢head
```

are highlighted. curNode points to the list's head, which is node 9. The line of code if (curNode⇢data == key) { is highlighted. 9 is not equal to 11. The line of code curNode = curNode⇢next is highlighted. curNode now points at node 7. The line of code if (curNode⇢data == key) { is highlighted. 7 is not equal to 11. The line of code curNode = curNode⇢next is highlighted. curNode now points at node 5. The line of code if (curNode⇢data == key) { is highlighted. 5 is not equal to 11. The line of code curNode = curNode⇢next is highlighted. curNode is now null. The line of code while (curNode is not null) is highlighted. The while loop is exited and the line of code return null is highlighted. ListSearch(list, 11) returns null.

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 4.3.2: Search algorithm execution.

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

Check **Show answer**

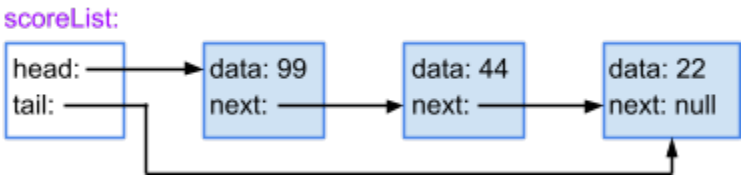
2) How many nodes are visited when searching for 48?

Check **Show answer**

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

Check **Show answer**

PARTICIPATION ACTIVITY 4.3.3: Searching a linked-list.



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

- Node 99
- Node 44
- Node 22

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

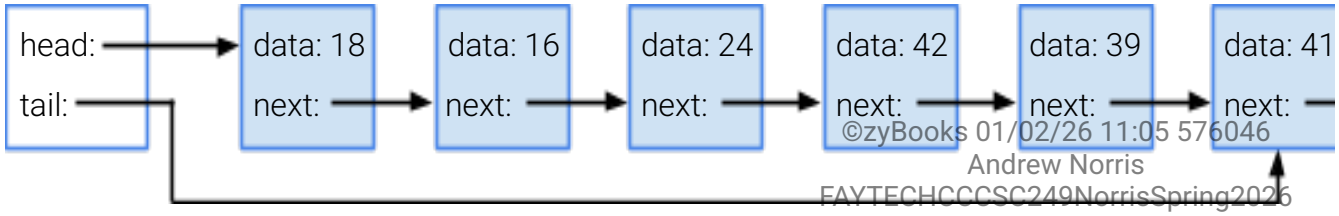
- `ListSearch(scoreList, 22)` only
- `ListSearch(scoreList, 33)` only
- `ListSearch(scoreList, 22)` and `ListSearch(scoreList, 33)`

CHALLENGE ACTIVITY 4.3.1: Linked list search.

704586.1152092.qx3zqy7

Start

numList:



`ListSearch(numList, 39)` points the current pointer to node Ex: 9 after checking node 16.

`ListSearch(numList, 39)` will make comparisons.

1 2

Check Next

View solution *(Instructors only)*

Insert-node-after operation

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

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

PARTICIPATION ACTIVITY 4.3.4: Singly-linked list insert-node-after operation.

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


Animation content:

Static figure:
Begin pseudocode:

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

End pseudocode.
ListInsertNodeAfter() code is shown. Below the code is a linked list with 3 nodes: 9, 4, and 15. To the right of the code are 3 ListInsertNodeAfter() calls:

©zyBooks 01/02/26 11:05 576046
Andrew Norris
FAYTECHCCCSC249NorrisSpring2026

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

Step 1: When the list is empty, the ListInsertNodeAfter() function assigns the list's head and tail pointers with the new node.
List object is shown with two labels within: "head: null" and "tail: null". Code execution begins for insertion of node 9. The code's first if statement's condition is true, so the list's head and tail pointers are both assigned with newNode. Arrows appear, stemming from "head:" and "tail:" labels within list object, each pointing to newNode.

Step 2: Inserting node15 after the list's tail, node9, first reassigns node9's next pointer.
Code execution begins for insertion of node 15. A "currentNode" label appears, with an arrow pointing to node 9. The first if statement's condition is false. The subsequent else-if's condition is true. The first of two statements in the else-if block executes. Node 9's "next: null" label changes to just "next:", followed by an arrow pointing to newNode (node 15).

Step 3: Then the list's tail pointer is assigned with newNode.
The second of the two statements in the else-if block executes. The list's tail arrow changes from pointing to node 9 to pointing to node 15.

Step 4: Inserting node4 after node9 first assigns node4's next with node15.
Code execution begins for insertion of node 4 after node 9. A "currentNode" label appears, with an arrow pointing to node 9. The first if statement's condition is false. The subsequent else-if's condition is false. So the first statement within the else block executes. An arrow appears stemming from newNode's "next:" label and pointing to node 19.

Step 5: Then currentNode's next is assigned with newNode.
The second statement within the else block executes. Node 9's next arrow changes to point to node 4. Then newNode moves to make the list's ordering more clear: node 9, then node 4, then node 15.

Animation captions:

1. When the list is empty, the ListInsertNodeAfter() function assigns the list's head and tail pointers with the new node.
2. Inserting node15 after the list's tail, node9, first reassigns node9's next pointer.
3. Then the list's tail pointer is assigned with newNode.
4. Inserting node4 after node9 first assigns node4's next with node15.
5. Then currentNode's next is assigned with newNode.

PARTICIPATION

ACTIVITY 4.3.5: Inserting nodes in a singly-linked list.

Select the list after the given operation(s).

1) numsList: (5, 9)

ListInsertNodeAfter(numsList, node9, node4)

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

2) numsList: (23, 17, 8)

ListInsertNodeAfter(numsList, node23, node5)

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

3) numsList: (1)

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

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

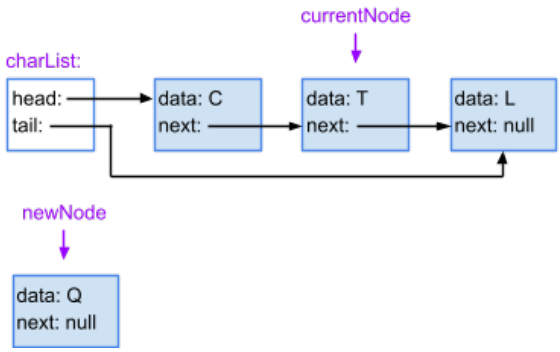
4) numsList: (77)

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

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

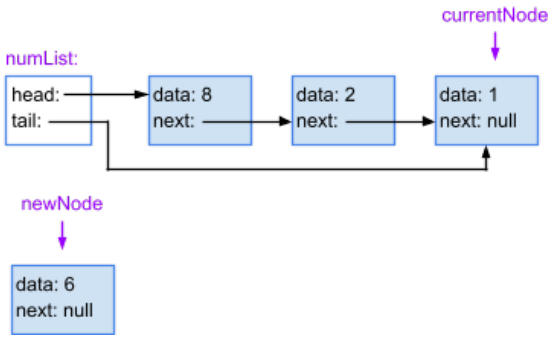
PARTICIPATION ACTIVITY 4.3.6: Singly-linked list insert-after algorithm.

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



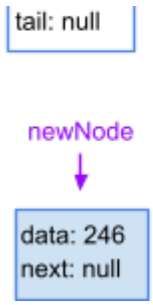
- currentNode⇢next
- charList's head node
- null

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



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

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



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

CHALLENGE ACTIVITY 4.3.2: Singly-linked lists: Insert.

704586.1152092.qx3zqy7

Start

Given numList: (67, 94)

What is numList after the following operations?

```
ListInsertNodeAfter(numList, node94, node64)
ListInsertNodeAfter(numList, node67, node98)
ListInsertNodeAfter(numList, node67, node74)
```

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

1 2 3 4

Check Next

View solution *(Instructors only)*

Insert-after operation

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

Figure 4.3.1: Singly-linked list ListInsertAfter() function.

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

PARTICIPATION ACTIVITY 4.3.7: Singly-linked list insert-after operation.

1) What happens if `ListInsertAfter(list, 22, 44)` is called for the list (11, 33, 55)?
   - An error occurs
   - ListInsertAfter() returns false, and the list is not changed
   - 44 is inserted after 33, yielding (11, 33, 44, 55)

2) ListInsertAfter() _____ calls ListSearch() and _____ calls ListInsertNodeAfter().
   - always, sometimes
   - sometimes, always
   - always, never

3) What is ListInsertAfter()'s *worst* case

runtime complexity?

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

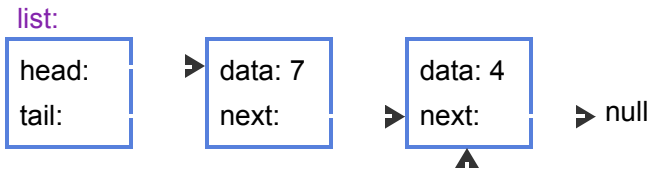
4.4 Singly-linked lists: Remove

Remove-node-after operation

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

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

PARTICIPATION ACTIVITY 4.4.1: Singly-linked list remove-node-after operation.




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

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

ListRemoveNodeAfter(list, null)

ListRemoveNodeAfter(list, node 4)

ListRemoveNodeAfter(list, node 4)

```
      sucNode = curNode→next→next
      curNode→next = sucNode

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


Animation content:

Static figure:
Begin pseudocode:

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

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

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

End pseudocode.
Three function calls:
ListRemoveNodeAfter(list, null)
ListRemoveNodeAfter(list, node 4)
ListRemoveNodeAfter(list, node 4)

A list with a head pointer pointing at node 7. Node 7's next pointer points to node 4. Node 4's next pointer is null. The list's tail pointer points to node 4. Another list labeled "list before removes". The head pointer points to node 9. Node 9's next pointer points to node 7. Node 7's next pointer points to node 4. Node 4's next pointer points to node 5. Node 5's next pointer points to node 2. Node 2's next pointer is null. The list's tail pointer points to node 2.

Step 1: When the argument for the current node is null, the list's head node will be removed.
The function call ListRemoveNodeAfter(list, null) is highlighted. The line of code ListRemoveNodeAfter(list, curNode) { is highlighted. null is passed into curNode. The following lines of code are highlighted:

```
if (curNode is null && list⇢head is not null) {
  sucNode = list⇢head⇢next
```

The list's head pointer points to node 9. Node 9's next pointer points to sucNode points to node 7. sucNode points to node 7.

Step 2: The list's head pointer is reassigned with the list head's successor node.
The line of code `list⇢head = sucNode` is highlighted. The list's head pointer points to node 7. Node 9 is removed from the list. The function returns.

Step 3: If a node exists after curNode, that node is removed. sucNode points to node after the next node (i.e., the next next node). The function call ListRemoveNodeAfter(list, node 4) is highlighted. The line of code ListRemoveNodeAfter(list, curNode) { is highlighted. Node 4 is passed into curNode. The line of code if (curNode is null && list⇢head is not null) { is highlighted. Next, the following lines of code are highlighted:

```
 else if (curNode⇢next is not null) {
  sucNode = curNode⇢next⇢next
```

curNode points to node 4. Node 4's next pointer points to node 5. Node 5's next pointer points to node 2. sucNode points to node 2.

Step 4: curNode's next pointer is pointed to sucNode. The line of code `curNode⇢next = sucNode` is highlighted. Node 4's next pointer is pointed to node 2. Node 5 is removed from the list. The function returns.

Step 5: When removing node 2, the successor is null.
The function call ListRemoveNodeAfter(list, node 4) is highlighted. The line of code ListRemoveNodeAfter(list, curNode) { is highlighted. Node 4 is passed to curNode. The line of code if (curNode is null && list⇢head is not null) { is highlighted. Next, the following lines of code are highlighted:

```
 else if (curNode⇢next is not null) {
  sucNode = curNode⇢next⇢next
```

curNode points to node 4. Node 4's next pointer points to node 2. Node 2's next pointer points at null. sucNode is pointed at null.

Step 6: So the current node's next pointer is reassigned with null.
The line of code `curNode⇢next = sucNode` is highlighted. Node 4's next pointer is pointed at null.

Step 7: Since the tail was removed, the list's tail pointer is reassigned with the current node.
The following lines of code are highlighted:

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

The list's tail pointer is pointed at node 4. Node 2 is removed from the list. The function returns.

Animation captions:

1. When the argument for the current node is null, the list's head node will be removed.
2. The list's head pointer is reassigned with the list head's successor node.
3. If a node exists after curNode, 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. When removing node 2, the successor is null.
6. So the current node's next pointer is reassigned with null.
7. Since the tail was removed, the list's tail pointer is reassigned with the current node.

PARTICIPATION ACTIVITY 4.4.2: Removing nodes from a singly-linked list.

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

1) numsList: (2, 5, 9)

ListRemoveNodeAfter(numsList, node 5)

numsList:

Check **Show answer**

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

ListRemoveNodeAfter(numsList, null)

numsList:

Check **Show answer**

3) numsList: (86, 99, 46)

ListRemoveNodeAfter(numsList,

node46)

numsList:

Check **Show answer**

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

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

numsList:

Check **Show answer**

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

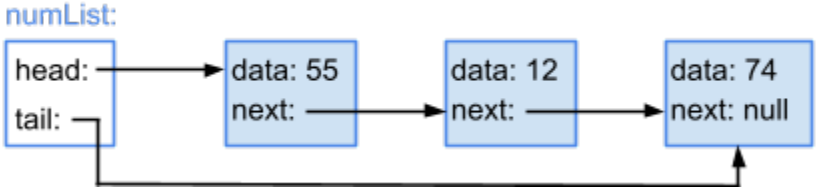
ListRemoveNodeAfter(numsList, node 60)
ListRemoveNodeAfter(numsList, node 77)
ListRemoveNodeAfter(numsList, null)

numsList:

Check **Show answer**

PARTICIPATION ACTIVITY 4.4.3: Remove-node-after algorithm execution: Intermediate node.

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



1) `sucNode = list→head→next`

- Yes
- No

2) `curNode→next = sucNode`

- Yes
- No

3) `list→head = sucNode`

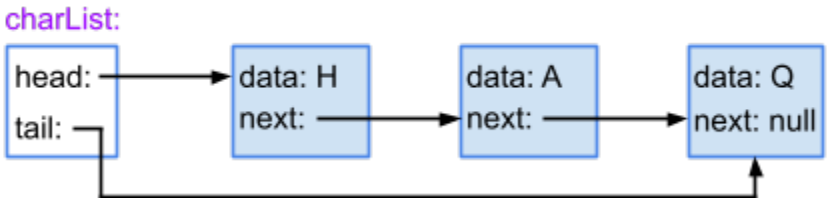
- Yes
- No

4) `list→tail = curNode`

- Yes
- No

PARTICIPATION ACTIVITY 4.4.4: Remove-node-after algorithm execution: List head node.

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



1) `sucNode = list→head→next`

- Yes
- No

2) `curNode→next = sucNode`

- Yes
- No

3) `list→head = sucNode`

- Yes
- No

4) `list→tail = curNode`

- Yes
- No

PARTICIPATION ACTIVITY 4.4.5: Removing from an empty list.

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

- True
- False

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

- True
- False

CHALLENGE ACTIVITY 4.4.1: Singly-linked lists: Remove.

704586.1152092.qx3zqy7

Start

Given numList: (6, 4, 3, 1, 9)

What is numList after the following operations?

```
ListRemoveNodeAfter(numList, node1)
ListRemoveNodeAfter(numList, null)
ListRemoveNodeAfter(numList, node4)
```

List items in order from head to tail

Check Next

View solution (*Instructors only*)

Remove-item operation

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

Figure 4.4.1: ListRemove() function.

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

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

   // Not found
   return false
}
```


PARTICIPATION ACTIVITY 4.4.6: Remove item operation.

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

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

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

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

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

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

4.5 C++: Singly-linked lists

SinglyLinkedNode and SinglyLinkedList classes

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

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

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

Figure 4.5.1: SinglyLinkedNode class definition.

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

   SinglyLinkedNode(int initialData) {
      data = initialData;
      next = nullptr;
   }
};
```


Figure 4.5.2: SinglyLinkedList class member variables and constructor.

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

public:
   SinglyLinkedList() {
      head = nullptr;
      tail = nullptr;
   }
};
```

PARTICIPATION ACTIVITY 4.5.1: SinglyLinkedNode and SinglyLinkedList classes.

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

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

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

- True
- False

4) The SinglyLinkedList class's head and tail node member variables are public.

- True
- False

Appending a node to a singly-linked list

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

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

Figure 4.5.3: SinglyLinkedList Append() and AppendNode() member functions.

```
void Append(int item) {
   AppendNode(new SinglyLinkedNode(item));
}

void AppendNode(SinglyLinkedNode* newNode) {
   if (head == nullptr) {
      head = newNode;
      tail = newNode;
   }
   else {
      tail->next = newNode;
      tail = newNode;
   }
}
```

PARTICIPATION ACTIVITY 4.5.2: SinglyLinkedList append methods.

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

- head
- tail

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

- `Append(numList, 15);`
- `numList.Append(15);`
- `numList.Append(new SinglyLinkedNode(15));`

Additional singly-linked list member functions

The following member functions are implemented in the SinglyLinkedList class.

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

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

Figure 4.5.4: SinglyLinkedList PrependNode() and Prepend() member functions.

```
void PrependNode(SinglyLinkedNode* newNode) {
   if (head == nullptr) {
      head = newNode;
      tail = newNode;
   }
   else {
      newNode->next = head;
      head = newNode;
   }
}

void Prepend(int item) {
   PrependNode(new SinglyLinkedNode(item));
}
```

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

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

The InsertAfter() member function also exists and takes integer arguments for the existing and new items.

Figure 4.5.5: SinglyLinkedList InsertNodeAfter() and InsertAfter() member functions.


```
void InsertNodeAfter(SinglyLinkedNode* currentNode, SinglyLinkedNode* newNode) {
   if (head == nullptr) {
      head = newNode;
      tail = newNode;
   }
   else if (currentNode == tail) {
      tail->next = newNode;
      tail = newNode;
   }
   else {
      newNode->next = currentNode->next;
      currentNode->next = newNode;
   }
}

bool InsertAfter(int currentItem, int newItem) {
   SinglyLinkedNode* currentNode = Search(currentItem);
   if (currentNode) {
      SinglyLinkedNode* newNode = new SinglyLinkedNode(newItem);
      InsertNodeAfter(currentNode, newNode);
      return true;
   }
   return false; // currentItem not found
}
```

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

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

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

Figure 4.5.6: SinglyLinkedList RemoveNodeAfter() and Remove() member functions.

```
void RemoveNodeAfter(SinglyLinkedNode* currentNode) {
   if (currentNode == nullptr && head) {
      // Special case: remove head
      SinglyLinkedNode* nodeBeingRemoved = head;
      head = head->next;
      delete nodeBeingRemoved;

      if (head == nullptr) {
         // Last item was removed
         tail = nullptr;
      }
   }
   else if (currentNode->next) {
      SinglyLinkedNode* nodeBeingRemoved = currentNode->next;
      SinglyLinkedNode* succeedingNode = currentNode->next->next;
      currentNode->next = succeedingNode;
      delete nodeBeingRemoved;
      if (succeedingNode == nullptr) {
         // Remove tail
         tail = currentNode;
      }
   }
}

bool Remove(int itemToRemove) {
   // Traverse to the node with data equal to itemToRemove,
   // keeping track of the previous node in the process
   SinglyLinkedNode* previous = nullptr;
   SinglyLinkedNode* current = head;
   while (current) {
      if (current->data == itemToRemove) {
         RemoveNodeAfter(previous);
         return true;
      }

      // Advance to next node
      previous = current;
      current = current->next;
   }

   // Not found
   return false;
}
```


PARTICIPATION ACTIVITY 4.5.3: Additional SinglyLinkedList operations.

©zyBooks 01/02/26 11:05 576046
Andrew Norris
FAYTECHCCCSC249NorrisSpring2026

1) Which is a parameter of PrependNode()?

- newNode
- currentNode

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

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

3) RemoveNodeAfter() removes the list's head node if the currentNode argument is _____.

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

4) When currentNode is null and the list is not empty, RemoveNodeAfter() _____.

- deallocates currentNode
- deallocates the list's head node
- does not deallocate any nodes

5) What are the parameter types for the Prepend(), InsertAfter(), and Remove() member functions?

- SinglyLinkedNode*
- int

SinglyLinkedList class destructor

The SinglyLinkedList class eventually deallocates any node added to the list. The RemoveNodeAfter() function deallocates the removed node. Any nodes left in the list when the SinglyLinkedList is destroyed are freed in the destructor.

Figure 4.5.7: SinglyLinkedList class destructor.

```
virtual ~SinglyLinkedList() {
   SinglyLinkedNode* currentNode = head;
   while (currentNode) {
      SinglyLinkedNode* toBeDeleted = currentNode;
      currentNode = currentNode->next;
      delete toBeDeleted;
   }
}
```


PARTICIPATION ACTIVITY 4.5.4: SinglyLinkedList class destructor.

1) The destructor calls RemoveNodeAfter() for each node in the list.
   - True
   - False

2) The destructor is the only SinglyLinkedList member function that deallocates nodes.
   - True
   - False

3) The destructor's while loop could be shortened to:

```
while (currentNode) {
   currentNode = currentNode->next;
   delete currentNode;
}
```

   - True
   - False

zyDE 4.5.1: Singly-linked list.

The following code implements several common list ADT operations using a singly-linked list. SinglyLinkedNode.h contains the SinglyLinkedNode class, and SinglyLinkedList.h contains the SinglyLinkedList class. SinglyLinkedList inherits from the ListADT class, defined in ListADT.h.

Current file: main.cpp Load default template...


```
#include <iostream>
#include "SinglyLinkedList.h"
using namespace std;

int main() {
   SinglyLinkedList numList;

   // Insert various items using Append(), Prepend(), and InsertAfter()
   numList.Append(14);          // List: 14
   numList.Append(2);           // List: 14, 2
   numList.Append(20);          // List: 14, 2, 20
   numList.Prepend(31);         // List: 31, 14, 2, 20
   numList.InsertAfter(2, 16);  // List: 31, 14, 2, 16, 20
   numList.InsertAfter(20, 55); // List: 31, 14, 2, 16, 20, 55

   // Output list
   cout << "List after adding items: ";
   numList.Print(cout);

   // Remove the tail node, then the head node
   numList.Remove(55); // List: 31, 14, 2, 16, 20
   numList.Remove(31); // List: 14, 2, 16, 20

   // Output list again
   cout << "List after removing first and last items: ";
   numList.Print(cout);

   // Insert three more items
   numList.Prepend(67);         // List: 67, 14, 2, 16, 20
   numList.InsertAfter(20, 58); // List: 67, 14, 2, 16, 20, 58
   numList.Append(89);          // List: 67, 14, 2, 16, 20, 58, 89

   // Output final list
   cout << "List after inserting three more items: ";
   numList.Print(cout);

   return 0;
}
```


Run

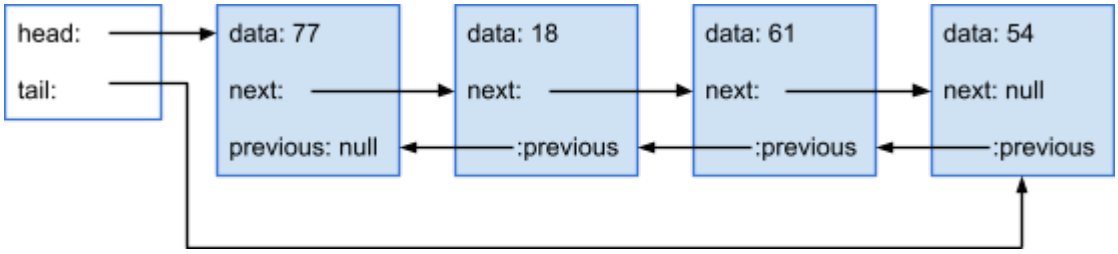
4.6 Doubly-linked lists

Doubly-linked list

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

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

Figure 4.6.1: Doubly-linked list with items: 77, 18, 61, 54.



PARTICIPATION ACTIVITY 4.6.1: Doubly-linked list data structure.

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

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

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

to node _____.

- 4
- 5

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

- 12
- 3

Append operation

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

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

PARTICIPATION ACTIVITY 4.6.2: Doubly-linked list: Append operation.

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

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

```
ListAppend(list, 9)

ListAppend(list, 15)
```


list:

Animation content:

Static figure:
Begin pseudocode:

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

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

End pseudocode.

Step 1: The ListAppend() function takes an integer list item as the second argument. A new node is allocated to hold the item.
Code is shown for the ListAppend() and ListAppendNode() functions. A box for the list data structure is shown below the code. The list's head and tail are null.
Execution of the statement list->Append(9) begins. Code highlighter moves inside the Append() function and highlights the allocation statement, "new DoublyLinkedNode(item)". A new node appears below, with the data=9, next=null, and previous=null.

Step 2: ListAppend() then calls ListAppendNode() to append the newly allocated node. The list is empty, so both the head and tail pointers are updated.
Execution continues, entering the AppendNode() function. The first if statement's condition is true, and so the corresponding block executes. An arrow appears, pointing from the "head:" label in the list to node 9. Another arrow appears, point from the "tail:" label in the list to node 9. Execution of

the call to append 9 completes.

Step 3: Appending to a non-empty list first assigns the current tail's next pointer with the new node. Execution of the statement list->Append(15) begins. The allocation statement executes and a new node appears with data=15, next=null, and previous=null. Execution enters AppendNode(), and the first if statement's condition is false. So execution moves to the else block. The first statement in the else block executes. The "null" after node 9's "next:" label disappears and is replaced by an arrow pointing to node 15.

Step 4: Then the new node's previous node is assigned with the current tail. Lastly, the list's tail pointer is assigned with newNode.
The remaining two statements of the else block execute. The first adds an arrow from node 15's previous to node 9. The second changes the list's tail arrow to point to node 15. Execution of the append statement completes.

Animation captions:

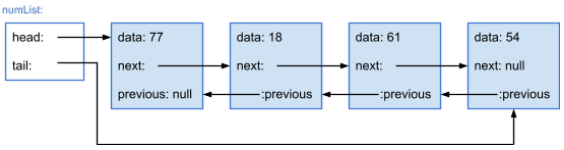
1. The ListAppend() function takes an integer list item as the second argument. A new node is allocated to hold the item.
2. ListAppend() then calls ListAppendNode() to append the newly allocated node. The list is empty, so both the head and tail pointers are updated.
3. Appending to a non-empty list first assigns the current tail's next pointer with the new node.
4. Then the new node's previous node is assigned with the current tail. Lastly, the list's tail pointer is assigned with newNode.

PARTICIPATION ACTIVITY 4.6.3: Doubly-linked list: Append operation.

1) A new node is allocated by which function(s)?
   - ○ ListAppend() only
   - ○ ListAppendNode() only
   - ○ Both ListAppend() and ListAppendNode()

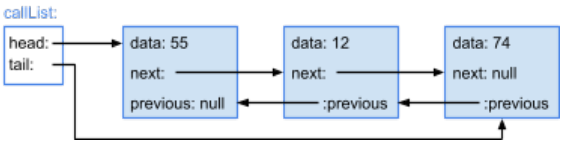
2) ListAppendNode() reassigns the list's head pointer _____.
   - ○ only when the list is empty
   - ○ only when the list is not empty
   - ○ in all cases

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



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

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



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

Prepend operation

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

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

PARTICIPATION ACTIVITY 4.6.4: Doubly-linked list: Prepend operation.

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

```
ListPrepend(list, 23)

ListPrepend(list, 17)
```

```
}

ListPrependNode(li
   if (list→head =
      list→head =
      list→tail = newNode
   }
   else {
      newNode→next = list→head
      list→head→prev = newNode
      list→head = newNode
   }
}
```


Animation content:

Static figure:
Begin pseudocode:

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

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

End pseudocode.
A linked list is shown with two nodes. List's head points to a node with data 17 and the tail points to a node with data 23. Node 17's next pointer points to node 23 and the previous pointer is null. Node 23's next pointer is null and the previous pointer points to node 17.
Two Prepend() calls exist outside the code box: list.Prepend(23) and list.Prepend(17).

Step 1: ListPrepend() takes an integer list item as the second argument. A new node is allocated to hold the item.

Code is shown for the DoublyLinkedList class's Prepend() and PrependNode() member functions. A box for the list data structure is shown below the code. The list's head and tail are null.
Execution of the statement list->Prepend(23) begins. Code highlighter moves inside the Prepend() function and highlights the allocation statement, "new DoublyLinkedNode(item)". A new node appears below, with the data=23, next=null, and previous=null.

Step 2: ListPrepend() then calls ListPrependNode() to prepend the newly allocated node. The list is empty, so both the head and tail pointers are updated.
Execution continues, entering the PrependNode() function. The first if statement's condition is true, and so the corresponding block executes. An arrow appears, pointing from the "head:" label in the list to node 23. Another arrow appears, point from the "tail:" label in the list to node 23. Execution of the call to prepend 23 completes.

Step 3: Prepending to a non-empty first assigns the new node's next pointer with the list's head.
Execution of the statement list->Prepend(17) begins. The allocation statement executes and a new node appears with data=17, next=null, and previous=null. Execution enters PrependNode(), and the first if statement's condition is false. So execution moves to the else block. The first statement in the else block executes. The "null" after node 17's "next:" label disappears and is replaced by an arrow pointing to node 23.

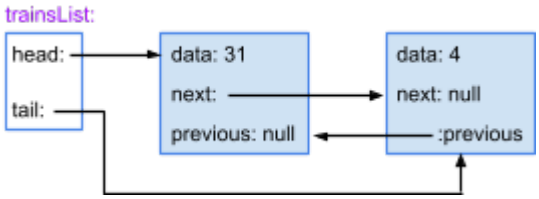
Step 4: Then the new head's previous node is assigned with the new node. Lastly, the list's head pointer is assigned with newNode.
The remaining two statements of the else block execute. The first adds an arrow from node 23's previous to node 17. The second changes the list's head arrow to point to node 17. Execution of the prepend statement completes.

Animation captions:

1. ListPrepend() takes an integer list item as the second argument. A new node is allocated to hold the item.
2. ListPrepend() then calls ListPrependNode() to prepend the newly allocated node. The list is empty, so both the head and tail pointers are updated.
3. Prepending to a non-empty first assigns the new node's next pointer with the list's head.
4. Then the new head's previous node is assigned with the new node. Lastly, the list's head pointer is assigned with newNode.

PARTICIPATION ACTIVITY 4.6.5: Prepending a node in a doubly-linked list.

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

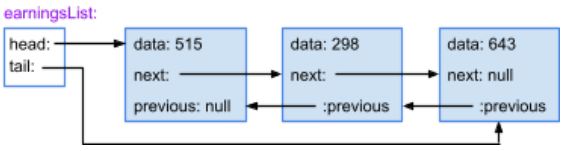


- 4
- 29
- 31

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

- True
- False

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



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

CHALLENGE ACTIVITY 4.6.1: Doubly-linked lists.

704586.1152092.qx3zqy7

Start

```
numList = new List
ListAppend(numList, 48)
ListAppend(numList, 46)
```


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

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

View solution *(Instructors only)*

4.7 Doubly-linked lists: Search and insert

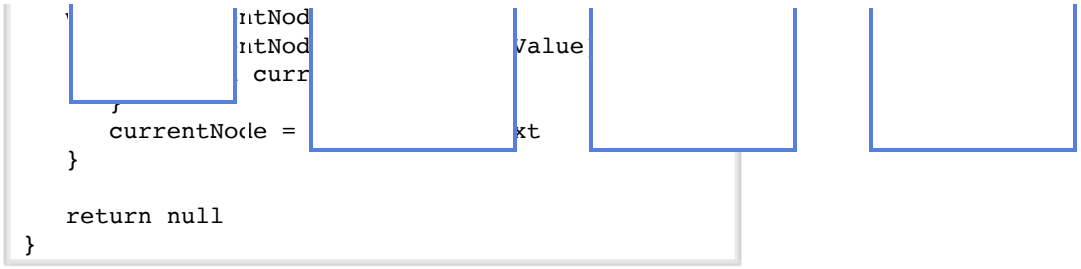
Search operation

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

PARTICIPATION ACTIVITY 4.7.1: Doubly-linked list search operation.

```
ListSearch(list, dataValue) {
   currentNode = list→head
```

`ListSearch(list, 68)` Returns node 68



Animation content:

Static figure:
Begin pseudocode:

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

  return null
}
```

End pseudocode.

Below the code is a linked list with three nodes: 71, 26, and 68. To the right of the code are two function calls:
ListSearch(list, 68), which returns node 68
ListSearch(list, 54), which returns null

Step 1: The ListSearch() function starts the search at the list's head node. The head node's data, 71, is not equal to 68.
A doubly-linked list with three nodes is shown: 71, 26, and 68. A list data structure is shown, with

the head pointing to node 71 and the tail node 68.
Execution of the statement ListSearch(list, 68) begins. A "currentNode" label appears above, and pointing to, node 71. The comparison 71 == 68 is false.

Step 2: The search advances to the next node and compares again. Node 68 is the first node with data equal to the dataValue argument. So node 68 is returned.
The currentNode label moves to node 26. The comparison 26 == 68 is false. Then the currentNode label moves to node 68. 68 == 68 is true, so node 68 is returned.

Step3: If no matching node is found, null is returned.
Execution of the statement ListSearch(list, 54) begins. The currentNode label advances through nodes, showing comparisons with data values. Each of the three comparisons, 71 == 54, 26 == 54, and 68 == 54, is false. So nullptr is returned.

Animation captions:

1. The ListSearch() function starts the search at the list's head node. The head node's data, 71, is not equal to 68.
2. The search advances to the next node and compares again. Node 68 is the first node with data equal to the dataValue argument. So node 68 is returned.
3. If no matching node is found, null is returned.

PARTICIPATION ACTIVITY 4.7.2: Doubly-linked list search operation.

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

- null
- Node 8
- Depends on the dataValue parameter

2) Which call returns null?

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

- ListSearch(numsList, 9)

3) Which call performs the most operations?

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

Insert-node-after operation

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

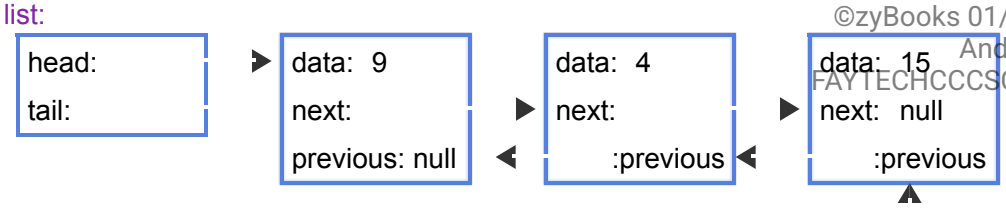
PARTICIPATION ACTIVITY 4.7.3: Doubly-linked list insert-after operation.

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

```
ListInsertNodeAfter(list, null, node9)

ListInsertNodeAfter(list, node9, node15

ListInsertNodeAfter(list, node9, node4)
```



Animation content:

Static figure:
Begin pseudocode:

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

End pseudocode.
Below the code is a linked list with three nodes: 9, 4, and 15. Three calls exist to the right of the code:
ListInsertNodeAfter(list, null, node9)
ListInsertNodeAfter(list, node9, node15)
ListInsertNodeAfter(list, node9, node4)

Step 1: When the list is empty, the ListInsertNodeAfter() function assigns the head and tail pointers with the new node.
The list object's head and tail members are initially null. The call to ListInsertNodeAfter(list, null, node9) assigns currentNode with null and newNode with node9. The first if statement assigns head and tail with newNode. The list now has one node.

Step 2: Inserting node15 after the list's tail, node9, begins by reassigning node9's next pointer.
The call to ListInsertNodeAfter(list, node9, node15) assigns currentNode with node9 and newNode with node15. The condition currentNode == list⇢tail is true, so list⇢tail⇢next = newNode executes. An arrow appears, pointing from currentNode's next label to newNode.

Step 3: Then newNode's previous is assigned with the list's tail. Lastly, the list's tail is assigned with

newNode.
The lines newNode⇢previous = tail and list⇢tail = newNode execute. The list now has node15 behind node9. node15's next is null. The list's tail points to node15.

Step 4: Inserting node4 after node9 begins by assigning node4's next with node9's successor, node15.
The call to ListInsertNodeAfter(list, node9, node4) assigns currentNode with node9 and newNode with node4. The function's else block executes the two lines:
successor = currentNode⇢next
newNode⇢next = successor

Step 5: Then newNode's previous is assigned with currentNode, and currentNode's next is assigned with newNode.
The following lines execute:
newNode⇢previous = currentNode
currentNode⇢next = newNode

Step 6: Lastly, successor's previous is assigned with newNode.
The line successor⇢previous = newNode executes. The list now has three nodes: node9, node4, node15. The list's head points to node9, and tail points to node15. Each node's next pointer points to the node that follows except node15's next, which is null. Each node's previous pointer points to the previous node except node9's previous, which is null.

Animation captions:

1. When the list is empty, the ListInsertNodeAfter() function assigns the head and tail pointers with the new node.
2. Inserting node15 after the list's tail, node9, begins by reassigning node9's next pointer.
3. Then newNode's previous is assigned with the list's tail. Lastly, the list's tail is assigned with newNode.
4. Inserting node4 after node9 begins by assigning node4's next with node9's successor, node15.
5. Then newNode's previous is assigned with currentNode, and currentNode's next is assigned with newNode.
6. Lastly, successor's previous is assigned with newNode.

PARTICIPATION ACTIVITY 4.7.4: Inserting nodes in a doubly-linked list.

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

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

How to use this tool

Mouse: Drag/drop
Keyboard: Grab/release Spacebar (or Enter). Move ↑ ↓ ← →. Cancel Esc

Node 76 **Node 12** **Node 45** **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 4.7.1: Doubly-linked lists: Insert.

704586.1152092.qx3zqy7

Start

Given numList: (52, 42)

What is numList after the following operations?

```
ListInsertNodeAfter(numList, node42, node23)
ListInsertNodeAfter(numList, node23, node97)
ListInsertNodeAfter(numList, node97, node10)
ListInsertNodeAfter(numList, node97, node16)
```


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

What node does node 16's next pointer point to? Select

1 2 3 4

Check Next

View solution *(Instructors only)*

Insert-after operation

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

Figure 4.7.1: ListInsertAfter() function.

```
ListInsertAfter(list, currentItem, newItem) {
   currentNode = ListSearch(list, currentItem)
   if (currentNode != null) {
      newNode = Allocate new doubly-linked node with newItem as data
      ListInsertNodeAfter(list, currentNode, newNode)
      return true
   }
   return false // currentItem not found
}
```

PARTICIPATION ACTIVITY 4.7.5: Doubly-linked list insert-item-after operation.

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

2) What list results from calling

`ListInsertAfter(list, 10, 99)` for the list (20, 10, 40, 10)?

- (20, 10, 40, 10)
- (20, 10, 99, 40, 10)
- (20, 10, 40, 10, 99)

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

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

4.8 Doubly-linked lists: Remove

Remove-node operation

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

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

PARTICIPATION ACTIVITY 4.8.1: Doubly-linked list: Node removal.

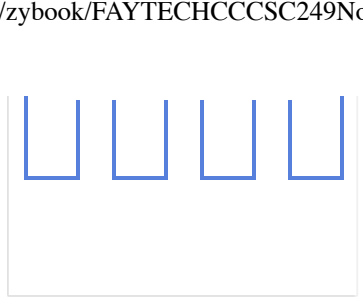
```
ListRemoveNode(list, currentNode) {
```

```
ListRemoveNode(list, node7
```

```
    successor = currentNode→next
    predecessor = currentNode→previous

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

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



Animation content:

Static figure:
Begin pseudocode:

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

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

  if (currentNode == list⇢head) {
    list⇢head = successor
  }
```

```
  if (currentNode == list⇢tail) {
    list⇢tail = predecessor
  }
}
```

End pseudocode.

Below the code are two linked lists. List on the left is empty. List on the right, labeled "list before removals", has items 99, 71, and 48.

Three removal calls exist to the right of the code:
ListRemoveNode(list, node71)
ListRemoveNode(list, node48)
ListRemoveNode(list, node99)

Step 1: The ListRemoveNode() function starts by assigning local variables successor and predecessor with currentNode's next and previous, respectively.
The doubly-linked list contains three nodes: 99, 71, 48. The call to list.RemoveNode(node71) assigns the parameter currentNode with node71. The first two lines of ListRemoveNode() execute and pointer labels appear near nodes in the linked list. "successor" points to node 48 and "predecessor" points to node 99.

Step 2: The successor, node 48, is non-null. So the successor's previous pointer is assigned with the predecessor, node 99.
The statement if (successor != null) is true, so the code successor⇢previous = predecessor executes. Node 48's previous now points to node 99.

Step 3: The predecessor is also non-null, so the predecessor's next pointer is assigned with the successor.
The statement if (predecessor != null) is true, so the code predecessor⇢next = successor executes. Node 99's next now points to node 48.

Step 4: currentNode is neither the list's head nor tail. No other nodes point to currentNode, so removal of node 71 is done.
Conditions for the remaining if statements in ListRemoveNode() are each false, so the function completes. Node 71 fades out. The list now contains two nodes: 99 and 48.

Step 5: Node 48 is the tail node, so no successor exists. Removal reassigns node 99's next pointer and the list's tail pointer.

The call to ListRemoveNode(list, node48) assigns the parameter currentNode with node48. predecessor is assigned with node99, and successor with null. The statement if (predecessor != null) is true, so the code predecessor⇢next = successor executes. Node 99's next is now null. The statement if (currentNode == list⇢tail) is true, so list⇢tail = predecessor executes. The list's tail is now node99. The list now has only one node: 99.

Step 6: Node 99 has no successor or predecessor, but is both the list's head and tail. Removal reassigns the list's head and tail with null.
The call to ListRemoveNode(list, node99) assigns the parameter currentNode with node99. successor and predecessor are both null. The statement if (currentNode == list⋯→head) is true, so list⋯→head = successor executes. The statement if (currentNode == list⋯→tail) is true, so list⋯→tail = predecessor executes. The list's head and tail are now null, and the list is empty.

Animation captions:

1. The ListRemoveNode() function starts by assigning local variables successor and predecessor with currentNode's next and previous, respectively.
2. The successor, node 48, is non-null. So the successor's previous pointer is assigned with the predecessor, node 99.
3. The predecessor is also non-null, so the predecessor's next pointer is assigned with the successor.
4. currentNode is neither the list's head nor tail. No other nodes point to currentNode, so removal of node 71 is done.
5. Node 48 is the tail node, so no successor exists. Removal reassigns node 99's next pointer and the list's tail pointer.
6. Node 99 has no successor or predecessor, but is both the list's head and tail. Removal reassigns the list's head and tail with null.

PARTICIPATION ACTIVITY 4.8.2: Removing nodes from a doubly-linked list.

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

1) numsList: 71, 29, 54

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

Check **Show answer**

2) numsList: 2, 8, 1

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

Check **Show answer**

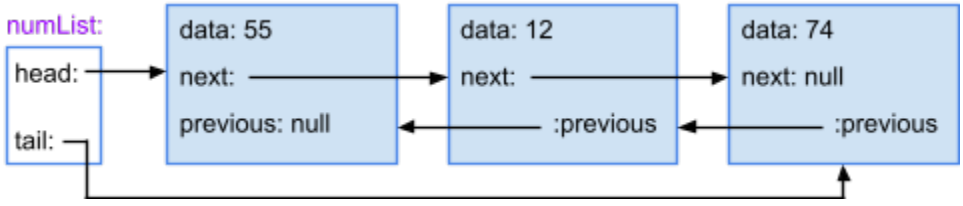
3) numsList: 70, 82, 41, 120, 357, 66

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


Check **Show answer**

PARTICIPATION ACTIVITY 4.8.3: Doubly-linked list remove algorithm execution: Intermediate node.

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



1) `successor→previous = predecessor`

- Yes
- No

2) `predecessor→next = successor`

- Yes
- No

3) `list→head = successor`

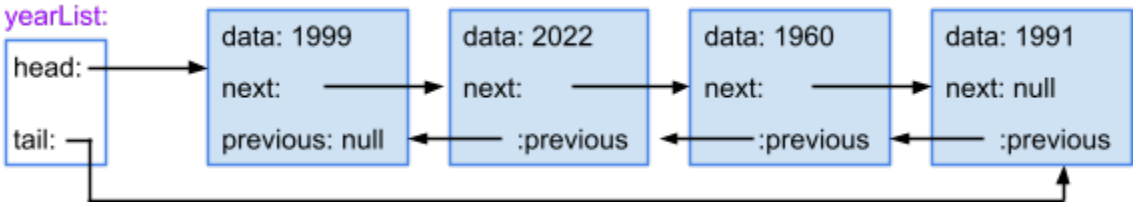
- Yes
- No

4) `list→tail = predecessor`

- Yes
- No

PARTICIPATION ACTIVITY 4.8.4: Doubly-linked list remove algorithm execution: List head node.

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



1) `successor→previous = predecessor`

- Yes
- No

2) `predecessor→next = successor`

- Yes
- No

3) `list→head = successor`

- Yes
- No

4) `list→tail = predecessor`

- Yes
- No

CHALLENGE ACTIVITY 4.8.1: Doubly-linked lists: Remove.

704586.1152092.qx3zqy7

Start

Given numList: (4, 1, 8, 2, 9, 3)

What is numList after the following operations?

```
ListRemoveNode(numList, node3)
ListRemoveNode(numList, node8)
ListRemoveNode(numList, node4)
```

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

1 2 3 4 5

Check Next

View solution *(Instructors only)*

Remove-item operation

A list ADT's remove operation has a parameter for a list *item*, not a *node*. So when using a doubly-linked list to implement a list ADT, a `ListRemove(list, item)` function is implemented. The function first calls ListSearch() to find the node containing the item to remove. If non-null, the node returned by ListSearch() is then passed to ListRemoveNode().

Figure 4.8.1: ListRemove() function.

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

   return false // not found
}
```


PARTICIPATION ACTIVITY 4.8.5: Remove item operation.

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

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

- True
- False

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

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

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

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

4.9 C++: Doubly-linked lists

DoublyLinkedNode and DoublyLinkedList classes

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

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

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

Figure 4.9.1: DoublyLinkedNode class definition for a doubly-linked node.

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

   DoublyLinkedNode(int initialData) {
      data = initialData;
      next = nullptr;
      previous = nullptr;
   }
};
```

Figure 4.9.2: DoublyLinkedList class definition for a doubly-linked list.

```
class DoublyLinkedList : public ListADT {
private:
   DoublyLinkedNode* head;
   DoublyLinkedNode* tail;

public:
   DoublyLinkedList() {
      head = nullptr;
      tail = nullptr;
   }
};
```


PARTICIPATION ACTIVITY 4.9.1: Doubly-linked lists and nodes.

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

- True
- False

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

- True
- False

3) Each node's data is an integer.

- True
- False

Appending a node to a doubly-linked list

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

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

Figure 4.9.3: DoublyLinkedList AppendNode() and Append() member functions.

```
void AppendNode(DoublyLinkedNode* newNode) {
   if (head == nullptr) {
      head = newNode;
      tail = newNode;
   }
   else {
      tail->next = newNode;
      newNode->previous = tail;
      tail = newNode;
   }
}

void Append(int item) {
   AppendNode(new DoublyLinkedNode(item));
}
```

PARTICIPATION ACTIVITY 4.9.2: Appending to a doubly-linked list.

Assume the code below has been executed.

```
DoublyLinkedList numberList;
numberList.Append(72);
numberList.Append(48);
numberList.Append(91);
```


1) If node48 is the node with data 48 then `node48->previous` is ____.
   - the node with data 72
   - the node with data 48
   - the node with data 91
   - null

2) If node91 is the node with data 91 then `node91->next` is ____.
   - the node with data 72
   - the node with data 48
   - null

3) Which node has previous assigned with null?
   - Node 72
   - Node 48
   - Node 91

Prepending a node to a doubly-linked list

The PrependNode() member function inserts a new node at the head of a doubly-linked list, making the new node the head of the list. Additionally, the old head node's previous must be set to point to the new node.

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

Figure 4.9.4: DoublyLinkedList PrependNode() and Prepend() member functions.

```
void PrependNode(DoublyLinkedNode* newNode) {
   if (head == nullptr) {
      head = newNode;
      tail = newNode;
   }
   else {
      newNode->next = head;
      head->previous = newNode;
      head = newNode;
   }
}

void Prepend(int item) {
   PrependNode(new DoublyLinkedNode(item));
}
```


PARTICIPATION ACTIVITY 4.9.3: PrependNode() member function.

How to use this tool

Mouse: Drag/drop
Keyboard: Grab/release **Spacebar** (or **Enter**). Move ↑ ↓ ← →. Cancel **Esc**

head **newNode** **tail**

Changed only if the list is empty.

`head->previous` is assigned with ____ if the list is not empty.

Always assigned with newNode.

Reset

Insertion and removal member functions

The InsertNodeAfter() member function has two node parameters:

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

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

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

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

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

Figure 4.9.5: DoublyLinkedList InsertNodeAfter() and insertAfter() member functions.

```
void InsertNodeAfter(DoublyLinkedNode* currentNode, DoublyLinkedNode* newNode) {
   if (head == nullptr) {
      head = newNode;
      tail = newNode;
   }
   else if (currentNode == tail) {
      tail->next = newNode;
      newNode->previous = tail;
      tail = newNode;
   }
   else {
      DoublyLinkedNode* successor = currentNode->next;
      newNode->next = successor;
      newNode->previous = currentNode;
      currentNode->next = newNode;
      successor->previous = newNode;
   }
}

bool InsertAfter(int currentItem, int newItem) {
   DoublyLinkedNode* currentNode = Search(currentItem);
   if (currentNode) {
      DoublyLinkedNode* newNode = new DoublyLinkedNode(newItem);
      InsertNodeAfter(currentNode, newNode);
      return true;
   }
   return false; // currentItem not found
}
```


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

RemoveNode() unlinks currentNode from the list by joining together the node before the removed node (the predecessor node) and the node after the removed node (the successor node). If currentNode is either the head or tail of the list, the function also updates the list's head and tail member variables. The removed node is deallocated.

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

Figure 4.9.6: DoublyLinkedList RemoveNode() and Remove() member functions.

```
void RemoveNode(DoublyLinkedNode* currentNode) {
   DoublyLinkedNode* successor = currentNode->next;
   DoublyLinkedNode* predecessor = currentNode->previous;

   if (successor) {
      successor->previous = predecessor;
   }
   if (predecessor) {
      predecessor->next = successor;
   }

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

   delete currentNode;
}

bool Remove(int itemToRemove) {
   DoublyLinkedNode* nodeToRemove = Search(itemToRemove);
   if (nodeToRemove) {
      RemoveNode(nodeToRemove);
      return true;
   }

   return false; // not found
}
```

PARTICIPATION ACTIVITY 4.9.4: Additional DoublyLinkedList member functions.

Consider the following code.

```
DoublyLinkedList numList;
numList.Append(1);
numList.Append(2);
numList.Append(3);
numList.Prepend(4);
numList.InsertAfter(1, 5);
```

```
numList.Remove(3);
```

1) What is the final list after the following code executes?

   - 1, 5, 2, 4
   - 4, 5, 1, 2
   - 4, 1, 5, 2

2) Replacing the line `numList.Append(2)` with ____ produces the same list.

   - `numList.Prepend(2)`
   - `numList.InsertAfter(1, 2)`
   - `numList.InsertAfter(2, 1)`

3) The `numList.Remove(3)` call deallocates one list node.

   - True
   - False

zyDE 4.9.1: Doubly-linked list.

The following code implements several common list ADT operations using a doubly-linked list. DoublyLinkedNode.h contains the DoublyLinkedNode class and DoublyLinkedList.h contains the DoublyLinkedList class. DoublyLinkedList inherits from the ListADT class, defined in ListADT.h.

Current file: main.cpp ▾ Load default template...


```
#include <iostream>
#include "DoublyLinkedList.h"
using namespace std;

int main() {
   DoublyLinkedList numList;

   // Insert various items using Append(), Prepend(), and InsertAfter()
   numList.Append(14);          // List: 14
   numList.Append(2);           // List: 14, 2
   numList.Append(20);          // List: 14, 2, 20
   numList.Prepend(31);         // List: 31, 14, 2, 20
   numList.InsertAfter(2, 16);  // List: 31, 14, 2, 16, 20
   numList.InsertAfter(20, 55); // List: 31, 14, 2, 16, 20, 55

   // Output list
   cout << "List after adding items: ";
   numList.Print(cout);

   // Remove the tail node, then the head node
   numList.Remove(55); // List: 31, 14, 2, 16, 20
   numList.Remove(31); // List: 14, 2, 16, 20

   // Output list again
   cout << "List after removing first and last items: ";
   numList.Print(cout);

   // Insert three more items
   numList.Prepend(67);         // List: 67, 14, 2, 16, 20
   numList.InsertAfter(20, 58); // List: 67, 14, 2, 16, 20, 58
   numList.Append(89);          // List: 67, 14, 2, 16, 20, 58, 89

   // Output final list
   cout << "List after inserting three more items: ";
   numList.Print(cout);

   return 0;
}
```

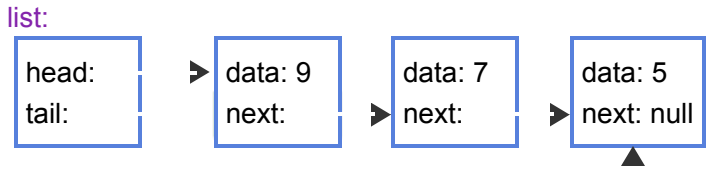

Run

4.10 Linked list traversal

Linked list traversal

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

PARTICIPATION ACTIVITY 4.10.1: Singly-linked list: List traversal.




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

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

9 7 5

Animation content:

Static figure:

```
Begin pseudocode:
ListTraverse(list) {
  curNode = list⇢head // Start at head

  while (curNode is not null) {
    Print curNode's data
    curNode = curNode⇢next
  }
}
End pseudocode.
```


4 list nodes. All connect to create a singly-linked list.
Node with data 9, next points to node with data 7.
Node with data 7, next points to node with data 5.
Node with data 5, next points to null.
Node labeled list, head points to 9 and tail points to 5.

An image of a monitor representing the output. It is initially empty.

Step 1:
Traverse starts at the list's head node.
The following lines of code are highlighted:

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

A new pointer called curNode is created that points to node 9.
The line of code, while (curNode is not null) {, is highlighted, initiating the while loop.
The line of code, Print curNode's data,is highlighted. The value 9 is outputted.

Step 2:
curNode's data is printed, and then curNode is pointed to the next node.
The line of code, curNode = curNode⇢next, is highlighted.
The curNode pointer now points to node 7.
The line of code, while (curNode is not null) {, is highlighted.
The line of code, Print curNode's data,is highlighted. The value 7 is outputted.
The line of code, curNode = curNode⇢next, is highlighted.
The curNode pointer now points to node 5.

Step 3:
After the list's tail node is printed, curNode is pointed to the tail node's next node, which does not exist.
The line of code, while (curNode is not null) {, is highlighted.
The line of code, Print curNode's data,is highlighted. The value 5 is outputted.

The line of code, curNode = curNode⋯➝next, is highlighted.
The curNode pointer now points to null.

Step 4:
The traversal ends when curNode is null.
The line of code, while (curNode is not null) {, is highlighted.
While loop exits.

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 4.10.2: List traversal.

1) ListTraverse() starts at _____.
   - a specified list node
   - the list's head node
   - the list's tail node

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

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

Doubly-linked list reverse traversal

A doubly-linked list also supports a reverse traversal. A ***reverse traversal*** visits all nodes starting with

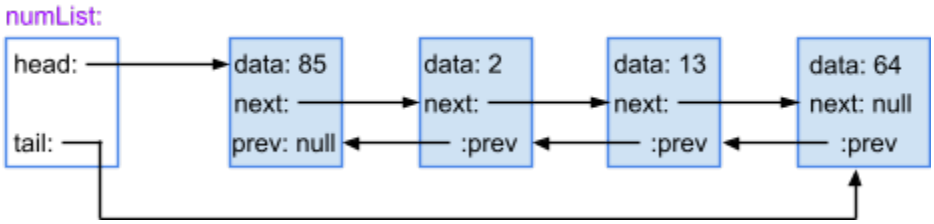
the list's tail node and ending after visiting the list's head node.

Figure 4.10.1: Reverse traversal algorithm.

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

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


PARTICIPATION ACTIVITY 4.10.3: Reverse traversal algorithm execution.



1) ListTraverseReverse() visits which node second?
   - Node 2
   - Node 13

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

4.11 Sorting linked lists

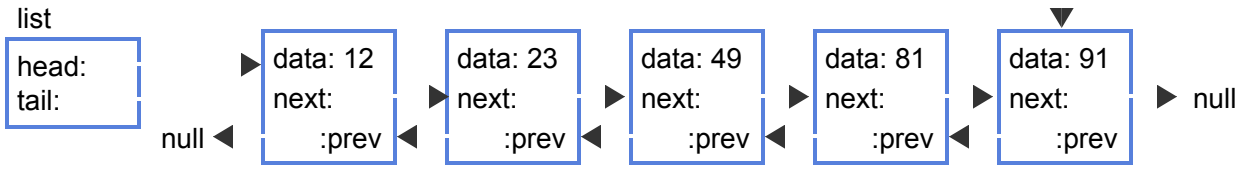
Insertion sort for doubly-linked lists

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


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



Animation content:

Static figure: A doubly linked list and code block.
Begin pseudocode:
ListInsertionSortDoublyLinked(list) {
  curNode = list⋯→head⋯→next
  while (curNode != null) {
    nextNode = curNode⋯→next
    searchNode = curNode⋯→prev
    while (searchNode != null and searchNode⋯→data > curNode⋯→data) {

```
      searchNode = searchNode⋯➝prev
    }
    // Remove and re-insert curNode
    ListRemoveNode(list, curNode)
    if (searchNode == null) {
      curNode⋯➝prev = null
      ListPrependNode(list, curNode)
    }
    else {
      ListInsertNodeAfter(list, searchNode, curNode)
    }
    // Advance to next node
    curNode = nextNode
  }
}
```

End pseudocode.

Step 1: The curNode pointer begins at node 91 in the list.
The code block and linked list are displayed. The linked list has five nodes. The linked list's head points to the first node, and tail points to the fifth node. First node: data: 81 next: pointer to second node. prev: pointer to null. Second node: data: 91 next: pointer to the third node prev: pointer to first node. Third node: data: 23 next: pointer to fourth node prev: pointer to second node. Fourth node: data: 49 next: pointer to fifth node prev: pointer to third node. Fifth node: data: 12 next: pointer to null prev: pointer to fourth node. The code ListInsertionSortDoublyLinked(list) { is highlighted, followed by the code curNode = list⋯➝head⋯➝next. The label curNode points to the second node

Step 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.
The code while (curNode != null) { is highlighted, followed by the code nextNode = curNode⋯➝next. The label nextNode points to the third node (data 23). The code searchNode = curNode⋯➝prev is highlighted. The label searchNode points to the first node (data 81). The while loop and then the Remove and re-insert curNode code lines are briefly highlighted. Then, the code curNode = nextNode is highlighted. curNode slides over to now point to the third node.

Step 3: For node 23, searchNode traverses the list backward until becoming null. Node 23 is prepended as the new list head.

The code while (curNode != null) { is highlighted, followed by the next two lines of code: nextNode = curNode⋯➝next.
searchNode = curNode⋯➝prev
The label nextNode points to the fourth node (data 49), and the label searchNode points to the second node (data 91). The while loop is highlighted. The searchNode slides down to point to null that the first node's prev points to. The following lines of code are highlighted:

```
ListRemove(list, curNode)
    if (searchNode == null) {
      curNode⇢prev = null
      ListPrepend(list, curNode)
    }
```

The third node (data 23) pointed to by curNode slides to the left to become the first node. The code curNode = nextNode is highlighted. curNode slides over to now point to the fourth node, while the searchNode and nextNode pointers fade away.

Step 4: Node 49 is inserted after node 23, using ListInsertAfter.
The code while (curNode != null) { is highlighted, followed by the next two lines of code: nextNode = curNode⇢next.
searchNode = curNode⇢prev
The label nextNode points to the fifth node (data 12), and the label searchNode points to the third node (data 91). The while loop is highlighted. The searchNode slides down to point to the first node. The Remove and re-insert curNode code lines are highlighted. The fourth node (data 49) pointed to by curNode slides to the left to become the second node. The code curNode = nextNode is highlighted. curNode slides over to now point to the fifth node (data 12), while the searchNode and nextNode pointers fade away.

Step 5: Node 12 is inserted before node 23, using ListPrepend, to complete the sort.
The code while (curNode != null) { is highlighted, followed by the next two lines of code: nextNode = curNode⇢next.
searchNode = curNode⇢prev
The label nextNode points to null, pointed to by the fifth node's next pointer, and the label searchNode points to the fourth node (data 91). The while loop is highlighted. The searchNode slides down to point to null, pointed to by the first node's prev pointer. The following lines of code are highlighted:

```
ListRemove(list, curNode)
    if (searchNode == null) {
      curNode⇢prev = null
      ListPrepend(list, curNode)
    }
```

The fifth node (data 12) pointed to by curNode slides to the left to become the first node. The code curNode = nextNode is highlighted. curNode slides over to point to null, pointed to by the fifth node's next pointer, while the searchNode and nextNode pointers fade away. The code while (curNode != null) { is highlighted, following the closing curly brace of the code.

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 ListInsertNodeAfter().
5. Node 12 is inserted before node 23, using ListPrependNode(), to complete the sort.

PARTICIPATION ACTIVITY 4.11.2: Improving ListInsertionSortDoublyLinked()

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

- zero nodes
- one node
- two nodes

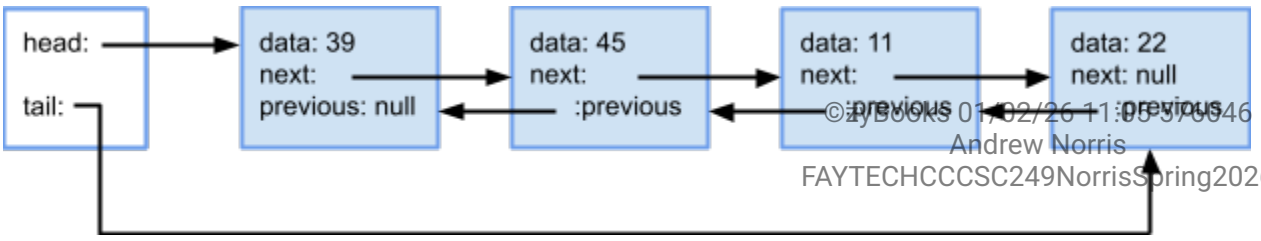
2) Wrapping ListInsertionSortDoublyLinked()'s body in which if statement addresses the minimum length problem?

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

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

PARTICIPATION ACTIVITY 4.11.3: Insertion sort for doubly-linked lists.

ListInsertionSortDoublyLinked() is called to sort the list below.



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

- Node 39
- Node 45

- Node 11

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

3) ListPrependNode() is called on which node(s)?
   - Node 11 only
   - Node 22 only
   - Nodes 11 and 22

Algorithm efficiency

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

Insertion sort for singly-linked lists

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

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

Figure 4.11.1: ListFindInsertionPosition algorithm.

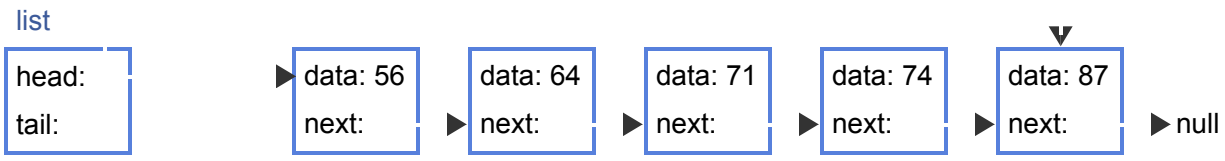
```
ListFindInsertionPosition(list, dataValue) {
   curNodeA = null
   curNodeB = list→head
   while (curNodeB != null and dataValue > curNodeB→data) {
      curNodeA = curNodeB
      curNodeB = curNodeB→next
   }
   return curNodeA
}
```


PARTICIPATION ACTIVITY 4.11.4: Sorting a singly-linked list with insertion sort.

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

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

      curNode = next
   }
}
```



Animation content:

Static figure: A singly linked list and code block.
Begin pseudocode:
ListInsertionSortSinglyLinked(list) {

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

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

    curNode = next
  }
}
```

End pseudocode.

Step 1: Insertion sort for a singly-linked list initializes curNode to point to the second list element, or node 56.

The code block and linked list are displayed. The linked list has five nodes. The linked list's head points to the first node, and tail points to fifth node. First node: data: 71 next: pointer to second node. Second node: data: 56 next: pointer to the third node. Third node: data: 64 next: pointer to fourth node. Fourth node: data: 87 next: pointer to fifth node. Fifth node: data: 74 next: pointer to null. The code ListInsertionSortSinglyLinked(list) { is highlighted, followed by the following four lines of code:

```
beforeCurrent = list⋯→head
curNode = list⋯→head⋯→next
while (curNode != null) {
  next = curNode⋯→next
```

The label beforeCurrent points to the first node (data 71), label curNode points to the second node (data 56), and label next points to the third node (data 64).

Step 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.

The code position = ListFindInsertionPosition(list, curNode⋯→data) is highlighted. The label position points to the word null that appears before the first node. The code if (position == beforeCurrent) is highlighted, followed by the following three lines of code:

```
ListRemoveAfter(list, beforeCurrent)
if (position == null)
  ListPrepend(list, curNode)
```

The second node (data 56), pointed to by curNode, slides to the left to become the first code. The code curNode = next is highlighted. The label curNode slides to the right to point to the third node (data 64), while the position pointer, the word null it points to, and next pointer all fade away. The closing curly brace to the while loop is now highlighted.

Step 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.
The code while (curNode != null) { is highlighted, followed by the following two lines of code:
next = curNode⇢next
position = ListFindInsertionPosition(list, curNode⇢data)
The label next points to the fourth node (data 74), and the label position points to the first node (data 56). The if else code block is highlighted. The third node (data 64), pointed to by curNode, slides over to become the second node. The code curNode = next is highlighted. curNode slides to the right to point to the fourth node (data 87), while the position and next pointers fade away.

Step 4: The insertion position for node 87 is after node 71. Node 87 is already in the correct position and is not moved.
The code while (curNode != null) { is highlighted, followed by the following two lines of code:
next = curNode⇢next
position = ListFindInsertionPosition(list, curNode⇢data)
The label next points to the fifth node (data 74), and the label position points to the third node (data 71), which also has the befpreCurrent pointer. The code if (position == beforeCurrent) is highlighted, followed by the code beforeCurrent = curNode. The label beforeCurrent slides over to point to fourth node (data 87). The code curNode = next is highlighted. curNode slides to the right to point to the fifth node (data 74), while the position and next pointers fade away.

Step 5: Although node 74 is only moved back one position, ListFindInsertionPosition compared node 74 with all other nodes' values to find the insertion position.
The code while (curNode != null) { is highlighted, followed by the following two lines of code:
next = curNode⇢next
position = ListFindInsertionPosition(list, curNode⇢data)
The label next points to the null, which the fifth node's next pointer points to, and the label position points to the third node (data 71). The if else code block is highlighted. The fifth node (data 74) slides over to become the fourth node. The code curNode = next is highlighted. curNode slides to the right to point to the null, which the fifth node's next pointer points to, while the position and next pointers fade away. The code while (curNode != null) { is highlighted, followed by the closing curly brace to the entire code block.

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.

PARTICIPATION ACTIVITY 4.11.5: ListInsertionSortSinglyLinked() function.

Refer to the animation above.

1) ListInsertionSortSinglyLinked() causes an error when the list is empty.
   - ○ True
   - ○ False

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

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

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

PARTICIPATION ACTIVITY 4.11.6: Sorting a singly-linked list with insertion sort.

ListInsertionSortSinglyLinked() is called to sort the list below.

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

- null
- Node 63
- Node 71
- Node 84

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

- 0
- 1
- 2

3) How many times is ListInsertNodeAfter() called?

- 0
- 1
- 2

Sorting linked-lists vs. arrays

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

Table 4.11.1: 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 4.11.2: Sorting algorithms difficult to adapt to efficiently sort linked lists.

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

PARTICIPATION ACTIVITY 4.11.7: Sorting linked-lists vs. sorting arrays.

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

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

2) Which sorting algorithm uses a gap

value to jump between elements, and is difficult to adapt to linked lists for this reason?

- Insertion sort
- Merge sort
- Shell sort

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

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

4.12 C++: Sorting linked lists

C++ insertion sort for doubly-linked lists

Doubly-linked list insertion sort is implemented as a member function of the DoublyLinkedList class. The MoveAfter() member function is added to DoublyLinkedList so that the InsertionSortDoublyLinked() member function can easily move a node within the list.

MoveAfter() first removes the node being moved, but does not deallocate the node. The node being moved is then inserted back into the list using either PrependNode() or InsertNodeAfter().

Figure 4.12.1: MoveAfter() and InsertionSortDoublyLinked() member functions.

```
// Moves nodeToMove immediately after nodeBefore. If nodeBefore is null,
// nodeToMove is moved to the front of the list.
void MoveAfter(DoublyLinkedNode* nodeToMove, DoublyLinkedNode* nodeBefore) {
   // First remove nodeToMove from the list, but do not deallocate
   DoublyLinkedNode* successor = nodeToMove->next;
   DoublyLinkedNode* predecessor = nodeToMove->previous;

   if (successor) {
      successor->previous = predecessor;
   }
   if (predecessor) {
      predecessor->next = successor;
   }
   if (nodeToMove == head) {
      head = successor;
   }
   if (nodeToMove == tail) {
      tail = predecessor;
   }

   nodeToMove->next = nullptr;
   nodeToMove->previous = nullptr;

   // If nodeBefore is non-null, use InsertNodeAfter(), otherwise use
   // PrependNode()
   if (nodeBefore) {
      InsertNodeAfter(nodeBefore, nodeToMove);
   }
   else {
      PrependNode(nodeToMove);
   }
}

void InsertionSortDoublyLinked() {
   if (!head) {
      return;
   }

   DoublyLinkedNode* currentNode = head->next;
   while (currentNode) {
      DoublyLinkedNode* nextNode = currentNode->next;
      DoublyLinkedNode* searchNode = currentNode->previous;

      while (searchNode && searchNode->data > currentNode->data) {
         searchNode = searchNode->previous;
      }

      // Move currentNode after searchNode
      MoveAfter(currentNode, searchNode);

      // Advance to next node
      currentNode = nextNode;
   }
}
```


©zyBooks 01/02/26 11:05 576046
Andrew Norris
FAYTECHCCCSC249NorrisSpring2026

PARTICIPATION ACTIVITY 4.12.1: Doubly-linked list insertion sort.

1) InsertionSortDoublyLinked() may allocate new nodes.
   - True
   - False

2) The following code could replace the call to the MoveAfter() function:

```
// Remove and re-insert
currentNode
RemoveNode(currentNode);
if (searchNode == nullptr) {
   currentNode->previous =
nullptr;
   PrependNode(currentNode);
}
else {
   InsertNodeAfter(searchNode,
currentNode);
}
```

   - True
   - False

3) If MoveAfter() is adapted to work for singly-linked lists, then InsertionSortDoublyLinked() works, without modifications, for singly-linked lists.
   - True
   - False

C++ insertion sort for singly-linked lists

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

Member functions InsertionSortSinglyLinked() and FindInsertionPosition() are added to the SinglyLinkedList class. The InsertionSortSinglyLinked() function calls the FindInsertionPosition() function to find where the current node should be inserted.

Figure 4.12.2: Insertion sort algorithm for singly-linked lists.

```
void InsertionSortSinglyLinked() {
   if (!head) {
      return;
   }

   SinglyLinkedNode* previousNode = head;
   SinglyLinkedNode* currentNode = head->next;
   while (currentNode) {
      SinglyLinkedNode* nextNode = currentNode->next;
      SinglyLinkedNode* position = FindInsertionPosition(currentNode->data);
      if (position == previousNode) {
         previousNode = currentNode;
      }
      else {
         previousNode->next = nextNode;
         if (nextNode == nullptr) {
            tail = previousNode; // Remove tail
         }
         currentNode->next = nullptr;

         if (position == nullptr) {
            PrependNode(currentNode);
         }
         else {
            InsertNodeAfter(position, currentNode);
         }
      }
      currentNode = nextNode;
   }
}

SinglyLinkedNode* FindInsertionPosition(int dataValue) const {
   SinglyLinkedNode* positionA = nullptr;
   SinglyLinkedNode* positionB = head;
   while (positionB && dataValue > positionB->data) {
      positionA = positionB;
      positionB = positionB->next;
   }
   return positionA;
}
```


PARTICIPATION ACTIVITY 4.12.2: Insertion sort for singly-linked lists.

1) InsertionSortSinglyLinked() never calls the _____ function.

- InsertNodeAfter()
- PrependNode()

- RemoveNodeAfter()

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

- True
- False

3) InsertionSortSinglyLinked() first initializes currentNode to _____.

- `head`
- `head->next`
- `nullptr`

zyDE 4.12.1: Insertion sort for linked lists.

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

- **SortingLinkedListsDemo.cpp** - The program's main() function
- **ListADT.h** - The ListADT abstract base class
- **SinglyLinkedList.h** - The SinglyLinkedList class that inherits from ListADT
- **SinglyLinkedNode.h** - The SinglyLinkedNode class
- **DoublyLinkedList.h** - The DoublyLinkedList class that inherits from ListADT
- **DoublyLinkedNode.h** - The DoublyLinkedNode class

Current file: SortingLinkedListsDemo.cpp Load default template...

```
#include <iostream>
#include <string>
#include <vector>
#include "SinglyLinkedList.h"
#include "DoublyLinkedList.h"
using namespace std;

int main() {
   vector<int> numbers = { 72, 91, 53, 12, 48, 19, 7, 1, 86 };

   // Create instances of SinglyLinkedList and DoublyLinkedList
   vector<ListADT*> linkedLists = {
      new SinglyLinkedList(),
      new DoublyLinkedList()
   };

   // Append numbers to each list
   for (ListADT* linkedList : linkedLists) {
      for (int number : numbers) {
         linkedList->Append(number);
      }
   }

   // For each list, print before sorting, sort, print again,
   // and then delete the list
   vector<string> listNames = {
      "Singly-linked",
      "Doubly-linked"
   };
   for (int i = 0; i < (int)linkedLists.size(); i++) {
      ListADT* linkedList = linkedLists[i];

      cout << listNames[i] << " list before sorting: ";
      linkedList->Print();
```

```
35        linkedList->Sort();
36
37        cout << listNames[i] << " list after sorting:  ";
38        linkedList->Print();
39
40        delete linkedList;
41    }
42
43    return 0;
44 }
```


Run

4.13 Linked list dummy nodes

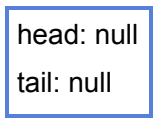
Dummy nodes

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

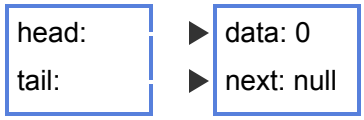
An empty list consists of the dummy node, which has the next pointer assigned with null, and the list's head and tail pointers both point to the dummy node.

PARTICIPATION ACTIVITY 4.13.1: Singly-linked lists with and without a dummy node.

Empty linked list without dummy node:



Empty linked list with dummy node:



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

Animation content:

Static figure: Four rows of different singly-linked lists.

Step 1: An empty linked list without a dummy node has null head and tail pointers.
The following text appears: Empty linked list without dummy node. Below the text, there is a box with the text head: null tail: null.

Step 2: An empty linked list with a dummy node has the head and tail pointing to a node with an unused data value.
The following text appears: Empty linked list with dummy node. Below the text, a linked list with a dummy node appears. The linked list's head and tail both point to this dummy node. Dummy node: data: 0 next: null. Dummy node is shaded orange.

Step 3: Without the dummy node, a non-empty list's head pointer points to the first list item.
The following text appears: List without dummy node and contents 82, 19, 56. Below the text, a linked list with three nodes appears. The linked list's head points to the first node, and the tail points to the third node. First node: data: 82 next: pointer to second node. Second node: data: 19 next: pointer to third node. Third node. data: 56 next: null.

Step 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.

The following text appears: List with dummy node and contents 82, 19, 56. Below the text, a linked list with four nodes appears. The linked list's head points to the first node, and the tail points to the fourth node. First node: data: 0 next: pointer to second node. This first node is a dummy node and shaded orange. Second node: data: 82 next: pointer to third node. Third node: data: 19 next: pointer to fourth node. Fourth node: data: 56 next: null.

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 an unused data value.
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 4.13.2: 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 4.13.3: Condition for an empty list.

If myList is a singly-linked list with a dummy node, which statement is true when the list is empty?

- `myList→head == null`
- `myList→tail == null`
- `myList→head == myList→tail`

Singly-linked list implementation

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

List operations such as append, prepend, insert-node-after, and remove-node-after are simpler to implement compared to a linked list without a dummy node since a special case is removed from each

implementation. ListAppendNode(), ListPrependNode(), and ListInsertNodeAfter() do not need to check if the list's head is null, since the list's head will always point to the dummy node. ListRemoveAfter() does not need a special case to allow removal of the first list item, since the first list item is after the dummy node.

Figure 4.13.1: Functions for a singly-linked list with a dummy node.


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


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


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


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

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


PARTICIPATION ACTIVITY 4.13.4: Singly-linked list with dummy node.

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

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

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

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

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

PARTICIPATION ACTIVITY 4.13.5: Singly-linked list with dummy node.

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

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

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

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

2) Suppose the following statement is executed:

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

Which subsequent operations swap nodes 73 and 19?

- ListPrepend(numbersList, node19)
- ListInsertNodeAfter(numbersList,

- numbersList⋯➝head, node19)
- ListRemoveAfter(numbersList, numbersList⋯➝head⋯➝next)
- ListPrepend(numbersList, node19)

Doubly-linked list implementation

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

Figure 4.13.2: Functions for a doubly-linked list with a dummy node.

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


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

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

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

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


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


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

   sucNode = curNode→next
   predNode = curNode→prev

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

   // Predecessor node is always non-null
```


©zyBooks 01/02/26 11:05 576046
Andrew Norris
FAYTECHCCCSC249NorrisSpring2026

```
   predNode→next = sucNode

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


PARTICIPATION ACTIVITY 4.13.6: Doubly-linked list with dummy node.

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

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

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

Dummy head and tail nodes

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

PARTICIPATION ACTIVITY 4.13.7: Doubly-linked list append and prepend with 2 dummy nodes.

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

ListPrependNode(list, node62)

ListAppendNode(list, node79)

```
    newNode→next = list→tail
    list→tail→prev→next = newNode
    list→tail→prev = newNode
}

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


Animation content:

Static figure: A code block and a doubly linked list. Two instructions are listed:
ListPrepend(list, node62)
ListAppend(list, node79)
Begin pseudocode:

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

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

End pseudocode.

Step 1: A list with two dummy nodes is initialized such that the list's head and tail point to two distinct nodes. Data is unused for both nodes.
A linked list with two nodes shaded orange appears. The linked list's head points to the first node. First node: data: 0 next: pointer to second node. prev: null. The linked list's tail points to the second node. Second node: data: 0 next: null prev: pointer to the first node.

Step 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.

The first instruction appears: ListPrepend(list, node62). The code ListPrepend(list, newNode) { is highlighted. A new node appears shaded in blue. New node: data: 62 next: pointer to the right prev: pointer to the left. The code firstNode = list⇢head⇢next is highlighted. The label firstNode points to the second dummy node. The next two lines of code are highlighted:
newNode⇢next = list⇢head⇢next
newNode⇢prev = list⇢head
The new node's next pointer points to the second dummy node, and its prev pointer points to the first dummy node. The next two lines of code are highlighted:
list⇢head⇢next = newNode
firstNode⇢prev = newNode
The first dummy node's next pointer now points to the new node. The second dummy node's prev pointer now points to the new node. The closing brace to this function is briefly highlighted and the firstNode pointer disappears.

Step 3: Appending inserts before the tail, since the list's tail pointer always points to the dummy node.
The second instruction appears: ListAppend(list, node79). The code ListAppend(list, newNode) { is highlighted. A new node appears shaded in blue. New node: data: 79 next: pointer to the right prev: pointer to the left. The next two lines of code are highlighted:
newNode⇢prev = list⇢tail⇢prev
newNode⇢next = list⇢tail
The new node's next pointer points to the second dummy node, and its prev pointer points to the second node with data 62. The next two lines of code are highlighted:
list⇢tail⇢prev⇢next = newNode
list⇢tail⇢prev = newNode
The second node's next pointer now points to the new node. The second dummy node's prev pointer now points to the new node. The closing brace to this function is briefly highlighted.

Animation captions:

1. A list with two dummy nodes is initialized such that the list's head and tail point to two distinct nodes. Data is unused 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 4.13.3: Doubly-linked list with two dummy nodes: insert-node-after and remove-node operations.


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

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

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

   sucNode = curNode→next
   predNode = curNode→prev

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

   // Predecessor node is never null
   predNode→next = sucNode
}
```

Removing if statements from ListInsertNodeAfter() and ListRemoveNode()

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

PARTICIPATION ACTIVITY 4.13.8: Comparing a doubly-linked list with one dummy node vs. two dummy nodes.

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

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

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

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

4.14 Linked lists: Recursion

Forward traversal

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

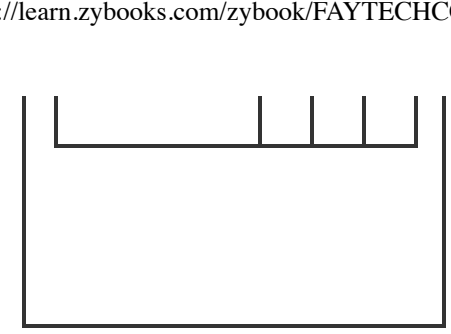
The ListTraverse() function takes a list as an argument, and traverses the entire list by calling ListTraverseRecursive() on the list's head.

PARTICIPATION ACTIVITY 4.14.1: Recursive forward traversal.

```
ListTraverse(list) {
   ListTraverseRecursive(list→head)
```

Function calls

```
        ers          e(r
    if (node is not null) {
        Visit node
        ListTraverseRecursive(node→next)
    }
}
```



Animation content:

Begin Pseudo Code:
ListTraverse(list) {
  ListTraverseRecursive(list⇢head)
}

ListTraverseRecursive(node) {
  if (node is not null) {
    Visit node
    ListTraverseRecursive(node⇢next)
  }
}
End Pseudo Code.

Step 1: Code for ListTraverse() and ListTraverseRecursive() is shown. A singly-linked list is shown below the code with contents: 23, 19, 41.
Execution of ListTraverse(list) begins. ListTraverse() calls ListTraverseRecursive(). A "node" label appears, pointing to node 23.

Step 2: Execution of ListTraverseRecursive() continues. Node 23 is visited. ListTraverseRecursive() is called again, with node 19 passed as the argument.

Step 3: Execution of ListTraverseRecursive() continues. Node 19 is visited. ListTraverseRecursive(node41) is called. Node 41 is visited. ListTraverseRecursive(null) is called. The if statement's condition is false, so the call returns without action. Each prior ListTraverseRecursive() call finishes, and execution completes.

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 4.14.2: Forward traversal in a linked list with 10 nodes.

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

- 9
- 10
- 11

PARTICIPATION ACTIVITY 4.14.3: Forward traversal concepts.

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

Searching

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

Figure 4.14.1: 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 4.14.4: Searching a linked list with 10 nodes.

Suppose a linked list has 10 nodes.

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

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

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

Reverse traversal

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

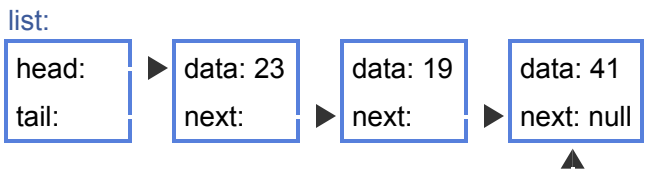
PARTICIPATION ACTIVITY 4.14.5: Recursive reverse traversal.


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

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

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

Visited nodes:	41	19	23



Animation content:

Begin Pseudo code:
ListTraverseReverse(list) {
  ListTraverseReverseRecursive(list⋯→head)
}

ListTraverseReverseRecursive(node) {
  if (node is not null) {
    ListTraverseReverseRecursive(node⋯→next)
    Visit node
  }
}
End Pseudo code.

Step 1: Code for ListTraverseReverse() and ListTraverseReverseRecursive() is shown. A singly-

linked list is shown below the code with contents: 23, 19, 41. A "Function calls" list exists on the right, initially empty.
Execution of ListTraverseReverse(list) begins. ListTraverseReverse() calls ListTraverseReverseRecursive(), passing the list head as an argument. Execution moves to the start of ListTraverseReverseRecursive(). A "node" label appears, pointing to node 23.
Functional calls list:
ListTraverseReverse(list)
ListTraverseReverseRecursive(node23)

Step 2: Execution of ListTraverseReverseRecursive() continues. The next recursive call occurs, and the "node" label moves to node 19.
Functional calls list:
ListTraverseReverse(list)
ListTraverseReverseRecursive(node23)
ListTraverseReverseRecursive(node19)

Step 3: Two more recursive calls occur, advancing the node pointer to node 41, then null. No nodes have been visited yet.
Functional calls list:
ListTraverseReverse(list)
ListTraverseReverseRecursive(node23)
ListTraverseReverseRecursive(node19)
ListTraverseReverseRecursive(node41)
ListTraverseReverseRecursive(null)

Step 4: Execution advances to the end of the current ListTraverseReverseRecursive() call. No other visual changes occur.

Step 5: The ListTraverseReverseRecursive(null) call completes and execution returns to ListTraverseReverseRecursive(node41). The node pointer points to node 41. Node 41 is visited. The "Visited nodes:" list appears.
Visited nodes: 41
Functional calls list:
ListTraverseReverse(list)
ListTraverseReverseRecursive(node23)
ListTraverseReverseRecursive(node19)
ListTraverseReverseRecursive(node41)

Step 6: Each recursive call completes, and nodes are visited. Execution is at the end of the ListTraverseReverseRecursive(node23) call. The node pointer points to node 23.
Visited nodes: 41, 19, 23
Functional calls list:

ListTraverseReverse(list)
ListTraverseReverseRecursive(node23)

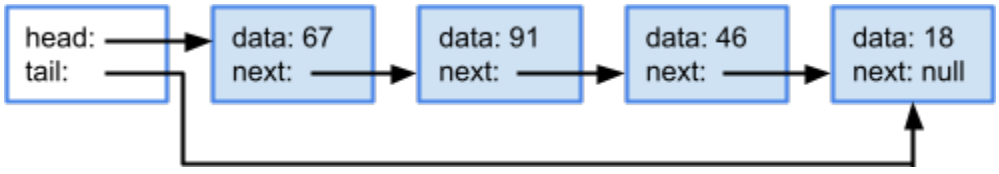
Step 7: Remaining calls complete, temporarily clearing the "Function calls" list. After execution completes, the entire list of function calls that occurred is shown:
ListTraverseReverse(list)
ListTraverseReverseRecursive(node23)
ListTraverseReverseRecursive(node19)
ListTraverseReverseRecursive(node41)
ListTraverseReverseRecursive(null)

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 occurs before visiting node 23.
3. Similarly, the recursive call on node 41 occurs before visiting node 19, and the recursive call on null occurs 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 parameter 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.
7. The last ListTraverseReverseRecursive() call returns to ListTraverseReverse(). The entire list has been visited in reverse order.

PARTICIPATION ACTIVITY 4.14.6: 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

4.15 Array-based lists

Array-based lists

An ***array-based list*** is a list ADT implemented using an array. An array-based list supports the common list ADT operations like append, prepend, insert-after, remove, and search.

In many programming languages, arrays have a fixed size. So an array-based list implementation first allocates a small array, perhaps with one to four elements. As list items are added, a length variable tracks how many array elements are in use. When the length equals the allocation size, adding a new item requires reallocating the array to a larger size.

Given a new element, the ***append*** operation for an array-based list of length X inserts the new element at the end of the list, or at index X.

PARTICIPATION ACTIVITY 4.15.1: Appending to array-based lists.

List implementation data:

array:

45	84	12	78

allocationSize: 4

length: 4

List operations:

Append(list, 45)
Append(list, 84)
Append(list, 12)
Append(list, 78)

Animation content:

Static image:
Two areas labeled "List implementation data:" and "List contents"

Step 1:
An array of length 4 is initialized for an empty list. Variables store the allocation size of 4 and list length of 0.
Under "List implementation data" is an empty array labeled "array".
There is a variable called allocationSize with value 4.
There is a variable called length with value 0.
Under "List contents" is the word "empty".

Step 2:
Appending 45 uses the first entry in the array, and the length is incremented to 1.
A list operation appears called Append(list, 45)
45 is added to the first cell of the array.
Length is now 1.
"List contents:" is now 45.

Step 3:
Appending 84, 12, and 78 uses the remaining space in the array.
3 more list operations appear: Append(list, 84), Append(list, 12), and Append(list, 78).
84 is added to the second cell of the array.
12 is added to the third cell of the array.
78 is added to the fourth and last cell of the array.
Length is now 4.
"List contents:" is now 45, 84, 12, 78.

©zyBooks 01/02/26 11:05 576046
Andrew Norris
FAYTECHCCCSC249NorrisSpring2026

Animation captions:

1. An array of length 4 is initialized for an empty list. Variables store the allocation size of 4 and list length of 0.
2. Appending 45 uses the first entry in the array, and the length is incremented to 1.
3. Appending 84, 12, and 78 uses the remaining space in the array.

PARTICIPATION ACTIVITY 4.15.2: Array-based lists.

1) The length of an array-based list equals the list's array allocation size.
   - True
   - False

2) 42 is appended to an array-based list with allocation size = 8 and length = 4. Appending assigns the array at index _____ with 42.
   - 4
   - 8

3) If an array-based list's allocation size = 8 and length = 8, then every array element is assigned a list item.
   - True
   - False

Resize operation

An array-based list must be resized if an item is added when the allocation size equals the list length. A new array is allocated with a length greater than the existing array. Doubling the current allocation size is common. The existing array elements are then copied to the new array, which becomes the list's storage array.

Because all existing elements must be copied from one array to another, the resize operation has a runtime complexity of $O(N)$.

PARTICIPATION ACTIVITY 4.15.3: Array-based list resize operation.


```
ArrayListAppend(list, newItem) {
   if (list→allocationSize == list→length) {
      ArrayListResize(list, list→length * 2)
   }
   list→array[list→length] = newItem
   list→length = list→length + 1
}
```

list:

array:

45	84	12	78	51			

allocationSize: 8

```
ArrayListResize(list, newAllocationSize) {
   newArray = new array of size newAllocationSize
   Copy all elements from list→array to newArray
   list→array = newArray
   list→allocationSize = newAllocationSize
}
```


Animation content:

Static image:
Begin pseudo code:
ArrayListAppend(list, newItem) {
  if (list⋯→allocationSize == list⋯→length) {
    ArrayListResize(list, list⋯→length * 2)
  }
  list⋯→array[list⋯→length] = newItem
  list⋯→length = list⋯→length + 1
}

ArrayListResize(list, newAllocationSize) {
  newArray = new array of size newAllocationSize
  Copy all elements from list⋯→array to newArray
  list⋯→array = newArray
  list⋯→allocationSize = newAllocationSize
}
End pseudo code.

An area labeled "list".
Under list, there is an array labeled "array". It contains values: 45, 84, 12, 78.
It has two variables. Variable labeled allocationSize with value 4 and variable length with value 4.
Array is full.

Step 1:
The allocation size and length of the list are both 4. An append operation cannot add to the existing array.
ArrayListAppend(list, 51) appears.
The following lines of code execute:
ArrayListAppend(list, newItem) {
  if (list⋯→allocationSize == list⋯→length) {

Step 2:
To resize, a new array is allocated of size 8, and the existing elements are copied to the new array. The new array replaces the list's array.
The line of code, ArrayListResize(list, list⋯→length * 2), executes.
The following code block executes:

```
ArrayListResize(list, newAllocationSize) {
  newArray = new array of size newAllocationSize
  Copy all elements from list⋯→array to newArray
  list⋯→array = newArray
  list⋯→allocationSize = newAllocationSize
}
```


newArray with size 8 is created. It contains the values: 45, 84, 12, 78 and has 4 empty cells.
array is now equal to newArray.
allocationSize is now 8.

Step 3:
51 can now be appended to the array.
The following lines of code execute:

```
list⋯→array[list⋯→length] = newItem
  list⋯→length = list⋯→length + 1
}
```

51 is appended to array. array now contains the values: 45, 84, 12, 78, 51.
Length is now 5.

Animation captions:

1. The allocation size and length of the list are both 4. An append operation cannot add to the existing array.
2. To resize, a new array is allocated of size 8, and the existing elements are copied to the new array. The new array replaces the list's array.
3. 51 can now be appended to the array.

PARTICIPATION ACTIVITY 4.15.4: Array-based list resize operation.

Consider numberList with variables:

array:

81	23	68	39	

allocationSize: 5

length: 4

The following operations are executed on the list:

ArrayListAppend(numberList, 98)
ArrayListAppend(numberList, 42)
ArrayListAppend(numberList, 63)

1) Which operation causes ArrayListResize() to be called?

- `ArrayListAppend(numberList, 98)`
- `ArrayListAppend(numberList, 42)`
- `ArrayListAppend(numberList, 63)`

2) What is the list's length after 63 is appended?

- 5
- 7
- 10

3) What is the list's allocation size after 63 is appended?

- 5
- 7
- 10

4) What was numberList's likely allocation size when the list was created and before any numbers were appended?

- 0
- 2
- 5

Prepend and insert-at operations

The ***prepend*** operation for an array-based list inserts a new item at the start of the list. First, if the allocation size equals the list length, the array is resized. Then all existing array elements are moved up one position, and the new item is inserted at index 0. Because all existing array elements are moved up

by one, the prepend operation has a runtime complexity of $O(N)$.

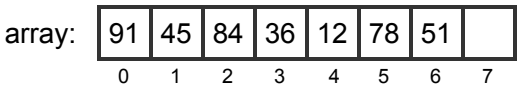
The ***insert-at*** operation for an array-based list inserts an item at a specified index. Ex: If the contents of `numbersList` is: 5, 8, 2, then `ArrayListInsertAt(numbersList, 2, 7)` yields: 5, 8, 7, 2. First, if the allocation size equals the list length, the array is resized. Next, all array elements from the specified index to the last index end are moved up by one position. Then the item is inserted at the specified index in the list's array. The insert-at operation has a best case runtime complexity of $O(1)$ and a worst case runtime complexity of $O(N)$.

PARTICIPATION ACTIVITY 4.15.5: Array-based list prepend and insert-at operations.

```
ArrayListPrepend(list, newItem) {
   if (list→allocationSize == list→length) {
      ArrayListResize(list, list→length * 2)
   }
   for (i = list→length; i > 0; i--) {
      list→array[i] = list→array[i - 1]
   }
   list→array[0] = newItem
   list→length = list→length + 1
}

ArrayListInsertAt(list, index, newItem) {
   if (list→allocationSize == list→length) {
      ArrayListResize(list, list→length * 2)
   }
   for (i = list→length; i > index; i--) {
      list→array[i] = list→array[i - 1]
   }
   list→array[index] = newItem
   list→length = list→length + 1
}
```

list:



allocationSize: 8
length: 7

ArrayListPrepend(list, 91)
ArrayListInsertAt(list, 3, 36)

Animation content:

Static image:

Begin pseudo code:

```
ArrayListPrepend(list, newItem) {
  if (list→allocationSize == list→length) {
    ArrayListResize(list, list→length * 2)
  }
  for (i = list→length; i > 0; i--) {
```

```
    list⋯→array[i] = list⋯→array[i - 1]
  }
  list⋯→array[0] = newItem
  list⋯→length = list⋯→length + 1
}

ArrayListInsertAt(list, index, newItem) {
  if (list⋯→allocationSize == list⋯→length) {
    ArrayListResize(list, list⋯→length * 2)
  }
  for (i = list⋯→length; i > index; i--) {
    list⋯→array[i] = list⋯→array[i - 1]
  }
  list⋯→array[index] = newItem
  list⋯→length = list⋯→length + 1
}
```


End pseudo code.

An area labeled "list".
Under list, there is an array labeled "array". It contains values: 45, 84, 12, 78, 51.
It has two variables. Variable labeled allocationSize with value 8 and variable length with value 5.
Array has 3 empty cells. The indices of array are labeled 0-7.

Step 1:
To prepend 91, every array element is first moved up one index.
ArrayListPrepend(list, 91) appears.
The following code block executes:

```
ArrayListPrepend(list, newItem) {
  if (list⋯→allocationSize == list⋯→length) {
    ArrayListResize(list, list⋯→length * 2)
  }
  for (i = list⋯→length; i > 0; i--) {
    list⋯→array[i] = list⋯→array[i - 1]
  }
```

array now contains values: 45, 45, 84, 12, 78, 51.

Step 2:
Item 91 is assigned to index 0 and length is incremented to 6.
The following lines of code execute:

```
list⋯→array[0] = newItem
  list⋯→length = list⋯→length + 1
}
```

The first cell in array is replaced with 91. array now contains values: 91, 45, 84, 12, 78, 51.
Length is now 6.

Step 3:
Inserting item 36 at index 3 requires elements at indices 3 and higher to be moved up one position.
Item 36 is then inserted at index 3.
ArrayListInsertAt(list, 3, 36) appears.
The following code block executes:


```
ArrayListInsertAt(list, index, newItem) {
  if (list⇢allocationSize == list⇢length) {
    ArrayListResize(list, list⇢length * 2)
  }
  for (i = list⇢length; i > index; i--) {
    list⇢array[i] = list⇢array[i - 1]
  }
  list⇢array[index] = newItem
  list⇢length = list⇢length + 1
}
```

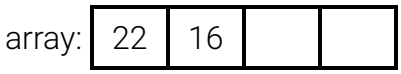
Since the length is less than the allocationSize (6 < 8), a resize is not necessary.
array becomes: 91, 45, 84, 12, 12, 78, 51.
The cell at index 3 is replaced with 36.
array is now: 91, 45, 84, 36, 12, 78, 51.
Length is now 7.

Animation captions:

1. To prepend 91, every array element is first moved up one index.
2. Item 91 is assigned to index 0 and length is incremented to 6.
3. Inserting item 36 at index 3 requires elements at indices 3 and higher to be moved up one position. Item 36 is then inserted at index 3.

PARTICIPATION ACTIVITY 4.15.6: Array-based list prepend and insert after operations.

Consider the list with variables:



allocationSize: 4

length: 2

The following operations are executed on the list:

ArrayListPrepend(list, 76)
ArrayListInsertAt(list, 2, 38)
ArrayListInsertAt(list, 4, 91)

1) Which operation causes ArrayListResize() to be called?

- ArrayListPrepend(list, 76)
- ArrayListInsertAt(list, 2, 38)
- ArrayListInsertAt(list, 4, 91)

2) What is the list's allocation size after all operations have completed?

- 5
- 8
- 10

3) What are the list's contents after all operations have completed?

- 22, 16, 76, 38, 91
- 76, 38, 22, 91, 16
- 76, 22, 38, 16, 91

4) Which ArrayListInsertAt() call is equivalent to ArrayListPrepend(list, 76)?

- ArrayListInsertAt(list, 0, 76)
- ArrayListInsertAt(list, 1, 76)

5) If ArrayListInsertAt() and length = 5, what does ArrayListInsertAt(list, 10, 55) do?

- Successfully inserts 55 and assigns length with 6

- Ignores the insert so the array remains unchanged
- Causes an out-of-bounds array access

Search and remove operations

Given an item, the ***search*** operation returns the index of the first equal list item, or -1 if not found.

Given the index of an item in an array-based list, the ***remove-at*** operation removes the item at that index. When removing an item at index X, each item after index X is moved down one position.

Both the search and remove operations have a worst case runtime complexity of $O(N)$.

PARTICIPATION ACTIVITY 4.15.7: Array-based list search and remove-at operations.

```
ArrayListSearch(list, item) {
   for (i = 0; i < list→length; i++) {
      if (list→array[i] == item) {
         return i
      }
   }
   return -1 // not found
}

ArrayListRemoveAt(list, index) {
   if (index >= 0 && index < list→length) {
      for (i = index; i < list→length - 1; i++) {
         list→array[i] = list→array[i + 1]
      }
      list→length = list→length - 1
   }
}
```

list:

array:

91	84	36	12	78	51		

allocationSize: 8

length: 6

`ArrayListSearch(list, 84)` Returns

`ArrayListRemoveAt(list, 1)`

`ArrayListSearch(list, 84)` Returns

Animation content:

Step 1: Code for ArrayListSearch() and ArrayListRemoveAt() functions is shown on the left. Data members for a list are shown to the right: array, allocationSize, and length.
"array:" label is followed by 8 boxes for the array's data: 91, 45, 84, 36, 12, 78, 51, (empty).
The other two labels are "allocationSize: 8" and "length: 7".
ArrayListSearch(list, 84) executes. Array elements 91, 45, and 84 are compared against the search item, 84. 84 == 84 is a match, and index 2 is returned.

Step 2: Execution of ArrayListRemoveAt(list, 1) begins. The function's for loop moves elements from indices 2 to 6 down one position, yielding: 91, 84, 36, 12, 78, 51, 51, (empty).

Step 3: ArrayListRemoveAt()'s last line executes, decrementing length to 6. Element 51 disappears from index 6, yielding: 91, 84, 36, 12, 78, 51, (empty), (empty).

Step 4: ArrayListSearch(list, 84) executes. Array elements 91, and 84 are compared against the search item, 84. 84 == 84 is a match, and index 1 is returned.

Animation captions:

1. The search for 84 compares against 3 elements before returning 2.
2. Removing the element at index 1 causes all elements after index 1 to be moved down to a lower index.
3. Decreasing the length by 1 then effectively removes the last 51.
4. The search for 84 now returns 1.

PARTICIPATION ACTIVITY 4.15.8: Search and remove-at operations.

Consider the list with variables:

array:

94	82	16	48	26	45

allocationSize: 6

length: 6

1) `ArrayListSearch(list, 33)` returns ____.

Check **Show answer**

2) When searching for 48, how many elements in the list are compared with 48?

Check **Show answer**

3) `ArrayListRemoveAt(list, 3)`

causes ____ elements to be moved down one index.

Check **Show answer**

4) `ArrayListRemoveAt(list, 5)` causes ____ elements to be moved down one index.

Check **Show answer**

PARTICIPATION ACTIVITY 4.15.9: Search and remove-at operations.

1) Removing at index 0 yields the best case runtime for remove-at.
   - True
   - False

2) Searching for an item that is not in the list yields the worst case runtime for search.
   - True
   - False

3) Neither search nor remove-at will resize the list's array.
   - True
   - False

CHALLENGE ACTIVITY 4.15.1: Array-based lists.

704586.1152092.qx3zqy7

Start

92	25	22

length: 3

1 2 3 4

Check Next

View solution *(Instructors only)*

4.16 C++: Array-based list

C++: Array-based list

C++'s vector type is implemented using an array-based data structure. To examine how an array-based list works, this section presents an ArrayList class implementing the array-based list data structure.

The ArrayList class uses an integer array as the internal data storage. The private member arrayListLength stores the number of array entries currently in use. The private member arrayAllocationSize stores in array's current allocation size, in number of elements.

Figure 4.16.1: ArrayList class: declaration of private members and constructor.

```
class ArrayList {
private:
   int* arrayData;           // Pointer to array data
   int arrayListLength;      // Number of elements in use
   int arrayAllocationSize; // Allocation size, in number of elements

public:
   ArrayList(int capacity = 4) {
      arrayAllocationSize = capacity;
      arrayData = new int[capacity];
      arrayListLength = 0;
   }
};
```


PARTICIPATION ACTIVITY 4.16.1: C++: Array-based list.

How to use this tool

Mouse: Drag/drop
Keyboard: Grab/release `Spacebar` (or `Enter`). Move `↑` `↓` `←` `→`. Cancel `Esc`

arrayListLength	arrayAllocationSize	arrayData
	Number of array elements currently in use.	
		Pointer to array data.
	Array's current allocation size, in number of elements.	

Reset

Append() and Resize() functions

Append() is implemented by inserting the new item at the index equal to the array's current length. If the array is filled all the way to the allocation size, then the array is doubled in size first.

Resize() works by first creating a new array of the indicated size and then copying all the items in the current array to the new array. The arrayData data member is then reassigned with the new array and the allocation size is updated.

Figure 4.16.2: Append() and Resize() functions.

```
void Append(int newItem) {
   // Resize if the array is full
   if (arrayAllocationSize == arrayListLength) {
      Resize(arrayListLength * 2);
   }

   // Insert the new item at index arrayListLength
   arrayData[arrayListLength] = newItem;

   // Increment arrayListLength to reflect the added item
   ++arrayListLength;
}

void Resize(int newAllocationSize) {
   // Create a new array with the indicated size
   int* newArray = new int[newAllocationSize];

   // Copy items in current array into the new array
   for (int i = 0; i < arrayListLength; ++i) {
      newArray[i] = arrayData[i];
   }

   // Free current array
   delete[] arrayData;

   // Assign the arrayData member with the new array
   arrayData = newArray;

   // Update allocation size
   arrayAllocationSize = newAllocationSize;
}
```


PARTICIPATION ACTIVITY 4.16.2: Array-based list implementation.

For each question, assume the following code has been executed:

```
int numbers[] = { 6, 2, 8, 4, 3 };
ArrayList myList(3);
for (int number : numbers) {
   myList.Append(number);
}
```


1) Append() uses data member arrayListLength as the index of the next available space in the internal array.

   - True
   - False

2) Resize() is called twice.

- True
- False

3) If the initial capacity were 1 instead of 3, then Append() would call Resize() three times.

- True
- False

Prepend() and InsertAfter() functions

Prepend() shifts the entire contents of the array one index to the right. Shifting is accomplished by copying the last item to the next index, then copying the second-to-last item to the index previously occupied by the last index, and so on until the first item is copied to index 1.

To do the shifting, the loop must use the indices in reverse order, from the last item down to the first item. Ex: If a list's length is 5, then the loop shifts by accessing indices 5, 4, 3, 2, 1, in that order. Since a new item is added to the list, the array is first checked to see if space is available, and resized if not.

InsertAfter() operates similarly to Prepend(). Items are shifted one space to the right, but only down to the provided index plus 1. All items from the given index down to 0 are not shifted, creating an empty space for the new item to be inserted into.

Figure 4.16.3: Prepend() and InsertAfter() functions.

```
void Prepend(int newItem) {
   // Resize if the array is full
   if (arrayAllocationSize == arrayListLength) {
      Resize(arrayListLength * 2);
   }

   // Shift all array items to the right,
   // starting from the last index and moving
   // down to the first index.
   for (int i = arrayListLength; i > 0; --i) {
      arrayData[i] = arrayData[i - 1];
   }

   // Insert the new item at index 0
   arrayData[0] = newItem;

   // Increment arrayListLength to reflect the added item
   ++arrayListLength;
}

void InsertAfter(int index, int newItem) {
   // Resize if the array is full
   if (arrayAllocationSize == arrayListLength) {
      Resize(arrayListLength * 2);
   }

   // Shift all the array items to the right,
   // starting from the last item and moving down to
   // the item just past the given index.
   for (int i = arrayListLength; i > index + 1; --i) {
      arrayData[i] = arrayData[i - 1];
   }

   // Insert the new item at the index just past the
   // given index.
   arrayData[index + 1] = newItem;

   // Increment arrayListLength to reflect the inserted item
   ++arrayListLength;
}
```


PARTICIPATION ACTIVITY 4.16.3: Prepend() and InsertAfter() functions.

For each question, assume the following code has been executed:

```
int numbers[] = { 6, 2, 8, 4, 3 };
ArrayList myList(3);
for (int number : numbers) {
   myList.Prepend(number);
}
myList.InsertAfter(2, 9);
```

1) What number is at index 0 in the list?

- 6
- 3
- 0

2) At what index is the number 9?

- 2
- 3
- The number 9 is not in the list.

3) What is the list's content after executing the additional statement below?

```
myList.InsertAfter(4, 5);
```

- 3, 4, 8, 2, 6, 5
- 3, 4, 8, 9, 2, 5, 6
- 3, 4, 8, 2, 9, 6, 5

Search() and RemoveAt() functions

Search() takes a target item as a parameter and then tests each item in the list, from index zero up to index arrayListLength - 1, for equality with the target item. The function returns the index where the first matching item is found, or -1 if no matching item exists in the array. The Search() function does not change the list's length or the internal array's allocation size.

RemoveAt() removes the item at the specified index by shifting the array items. The items from the parameter index plus 1 are shifted to the left by one index, up to the final item in the array. The RemoveAt() function decreases the list's length (assuming the specified index is valid in the current list), but does not change the internal array's allocation size.

Figure 4.16.4: Search() and RemoveAt() functions.

```
int Search(int item) {
   // Iterate through the entire array
   for (int i = 0; i < arrayListLength; ++i) {
      // If the current item matches the search
      // item, return the current index immediately.
      if (arrayData[i] == item) {
         return i;
      }
   }

   // If the above loop finishes without returning,
   // then the search item was not found.
   return -1;
}

void RemoveAt(int index) {
   // Make sure the index is valid for the current array
   if (index >= 0 && index < arrayListLength) {
      // Shift down all items after the given index
      for (int i = index; i < arrayListLength - 1; ++i) {
         arrayData[i] = arrayData[i + 1];
      }

      // Update arrayListLength to reflect the removed item
      --arrayListLength;
   }
}
```


PARTICIPATION ACTIVITY 4.16.4: Search() and RemoveAt() functions.

For each question, assume the following code has been executed:

```
int numbers[] = { 6, 2, 8, 4, 3 };
ArrayList myList(3);
for (int number : numbers) {
   myList.Append(number);
}
```

1) The function call `myList.RemoveAt(5)` has no effect.

   - True
   - False

2) The function call `myList.Search(1)` returns false.

   - True
   - False

3) The function call `myList.RemoveAt(4)` results in the list: 6, 2, 8, 3.

- True
- False

zyDE 4.16.1: Working with the ArrayList class.

©zyBooks 01/02/26 11:05 576046
Andrew Norris
FAYTECHCCCSC249NorrisSpring2026

The following program allows the user to enter a function name along with argument values to test the ArrayList implementation. The function name and arguments are separated by a single space. Ex: `InsertAfter 2 81`.

1. Test each function (Append(), Prepend(), InsertAfter(), Search() and RemoveAt()) with different arguments and see if you can predict the final output.
2. Add support for a command like `Remove 91` that will find and remove the first value of 91 found in the list. If the value is not found in the list, no action or output is required. *Do not modify the ArrayList class at all!* Adding the "Remove" command can be done using existing ArrayList functions.

Current file: **main.cpp** Load default template...

```
#include <iostream>
#include <string>
#include <vector>
#include "ArrayList.h"
using namespace std;

void InstructionSplit(const string& str, vector<string>& output) {
   size_t startIndex = 0;
   size_t endIndex = str.find(' ', startIndex);
   while (endIndex != string::npos) {
      output.push_back(str.substr(startIndex, endIndex - startIndex));
      startIndex = endIndex + 1;
      endIndex = str.find(' ', startIndex);
   }
   output.push_back(str.substr(startIndex));
}

int main() {
   int numbers[] = { 3, 2, 84, 18, 91, 6, 19, 12 };

   // Initialize a new ArrayList and add numbers
   ArrayList myList;
   for (int number : numbers) {
      myList.Append(number);
   }

   // Show the array before the operation
   cout << "-- Array before operation --" << endl;
   myList.PrintInfo(cout);
   cout << endl;
   myList.Print(cout);
   cout << endl;

   // Read an instruction
   string instructionLine;
```

```
36    getline(cin, instructionLine);
37    vector<string> instruction;
38    InstructionSplit(instructionLine, instruction);
39    string functionName = instruction[0];
40
41    int item, index;
42    if (functionName == "Append") {
43       item = stoi(instruction[1]);
44       myList.Append(item);
45    }
46    else if (functionName == "InsertAfter") {
47       index = stoi(instruction[1]);
48       item = stoi(instruction[2]);
49       myList.InsertAfter(index, item);
50    }
51    else if (functionName == "Prepend") {
52       item = stoi(instruction[1]);
53       myList.Prepend(item);
54    }
55    else if (functionName == "RemoveAt") {
56       index = stoi(instruction[1]);
57       myList.RemoveAt(index);
58    }
59    else if (functionName == "Search") {
60       item = stoi(instruction[1]);
61       cout << "Search result: " << myList.Search(item) << endl;
62    }
63    else {
64       cout << "Unknown function: " << functionName << endl;
65    }
66
67    cout << endl;
68    cout << "-- Array after operation --" << endl;
69    myList.PrintInfo(cout);
70    cout << endl;
71    myList.Print(cout);
72    cout << endl;
73 }
```


InsertAfter 5 3

Run

4.17 LAB: Sorted number list implementation with linked lists

LAB ACTIVITY | 4.17.1: LAB: Sorted number list implementation with linked lists

Full screen 0 / 10

Step 1: Inspect the NumberListNode.h file

Inspect the class declaration for a doubly-linked list node in NumberListNode.h. The NumberListNode class has three member variables:

- **data**: A double for the node's numerical data value
- **next**: A pointer to the next node
- **previous**: A pointer to the previous node

Each member variable is protected. So code outside of the class must use the provided getter and setter member functions to get or set a member variable.

NumberListNode.h is read-only since no changes are required.

Step 2: Inspect NumberList.h and SortedNumberList.h

NumberList.h contains the NumberList class definition. NumberList is a base class for a double-linked list of NumberListNodes. The list's head and tail pointers are protected member variables, and a Print() utility function exists to print the list's contents from head to tail. The file is read-only since no changes are required.

SortedNumberList.h contains the SortedNumberList class definition. SortedNumberList inherits from NumberList and adds Insert() and Remove() member functions.

Step 3: Implement Insert()

Implement SortedNumberList's Insert() member function to create a new node with the number argument as the node's data, then insert the node into the proper sorted position in the linked list. Ex: Suppose a SortedNumberList's current list is `23 → 47.25 → 86`, then `Insert(33.5)` is called. A new node with data value 33.5 is created and inserted between 23 and 47.25, thus preserving the list's sorted order and yielding: `23 → 33.5 → 47.25 → 86`.

Step 4: Run code and verify output

A vector of doubles named numbersToInsert is defined near the start of main(). Each is inserted

into a SortedNumberList, and the list is printed after each insertion. Try changing the vector's content, and verify that each output is a sorted list.

Step 5: Implement Remove()

Implement SortedNumberList's Remove() member function. The function takes an argument for the number to remove from the list. If the number does not exist in the list, the list is not changed and false is returned. Otherwise, the first instance of the number is removed from the list and true is returned.

Once Remove() is implemented, change the value of the includeRemovals variable, defined at the start of main(), from false to true. Run the code and make sure that the list is correct after each removal.

Try different tests in main() as needed, then submit code for grading. Unit tests are used to grade submitted code, so output from main() does not affect grading.

Open new tab Dock

4

Run History Tutorial

```
#include <iostream>
#include <iomanip>
#include <vector>
#include "SortedNumberList.h"
using namespace std;

int main() {
   bool includeRemovals = false;

   // Numbers to insert during first loop:
   vector<double> numbersToInsert = {
      77.75, 15.25, -4.25, 63.5, 18.25, -3.5
   };

   // Insert each number and print sorted list's contents after each insertion
   SortedNumberList list;
   cout << fixed << setprecision(2);
   for (auto number : numbersToInsert) {
      cout << "List after inserting " << number << ": ";
      list.Insert(number);
      list.Print(cout, ", ", "", "\n");
   }
```


DESKTOP CONSOLE

Model Solution

Run History Tutorial

main.cpp Read-only editor

```
#include <iostream>
#include <iomanip>
#include <vector>
#include "SortedNumberList.h"
using namespace std;

int main() {
   bool includeRemovals = false;

   // Numbers to insert during first loop:
   vector<double> numbersToInsert = {
      77.75, 15.25, -4.25, 63.5, 18.25, -3.5
   };

   // Insert each number and print sorted list's contents after each insertion
   SortedNumberList list;
   cout << fixed << setprecision(2);
   for (auto number : numbersToInsert) {
      cout << "List after inserting " << number << ": ";
      list.Insert(number);
      list.Print(cout, ", ", "", "\n");
   }

   if (includeRemovals) {
      cout << endl;

      vector<double> numbersToRemove = {
         77.75, // List's last element
         -4.25, // List's first element
         18.25, // Neither first nor last element

         // Remaining elements:
```


©zyBooks 01/02/26 11:05 576046
Andrew Norris
FAYTECHCCCSC249NorrisSpring2026