4.1 Doubly-linked lists

Doubly-linked list

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

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

PARTICIPATION ACTIVITY 4.1.1: Doubly-linked list data structure.

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

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

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

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

Appending a node to a doubly-linked list

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

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

PARTICIPATION ACTIVITY 4.1.2: Doubly-linked list: Appending a node.

Animation content:

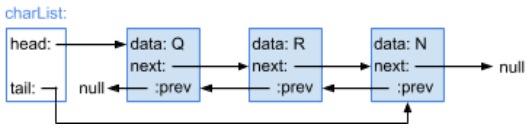
undefined

Animation captions:

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

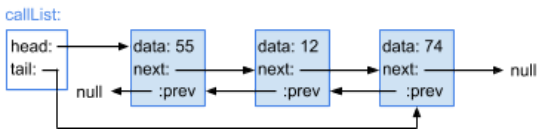
PARTICIPATION ACTIVITY 4.1.3: Doubly-linked list data structure.

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



- ○ after node Q
- ○ before node N
- ○ after node N

2) ListAppend(callList, node 5) executes which statement?



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

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

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

Prepending a node to a doubly-linked list

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

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

PARTICIPATION ACTIVITY 4.1.4: Doubly-linked list: Prepending a node.

Animation content:

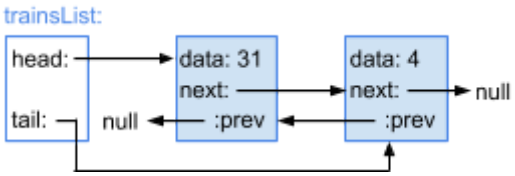
`undefined`

Animation captions:

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

PARTICIPATION ACTIVITY 4.1.5: Prepending a node in a doubly-linked list.

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

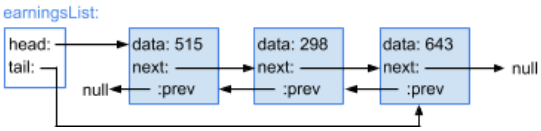


- 4
- 29
- 31

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

- True
- False

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



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

CHALLENGE ACTIVITY 4.1.1: Doubly-linked lists.

Start

numList = new List
ListAppend(numList, node 80)

ListAppend(numList, node 22)
ListAppend(numList, node 90)

numList is now: Ex: 1, 2, 3

Which node has a null next pointer? Ex: 5

Which node has a null previous pointer? Ex: 5

Check Next

4.2 Doubly-linked lists: Insert

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

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

PARTICIPATION ACTIVITY 4.2.1: Doubly-linked list: Inserting nodes.

Animation content:

undefined

Animation captions:

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

PARTICIPATION ACTIVITY 4.2.2: Inserting nodes in a doubly-linked list.

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

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

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

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

Reset

CHALLENGE ACTIVITY 4.2.1: Doubly-linked lists: Insert.

Start

numList: 65, 78
ListInsertAfter(numList, node 65, node 14)
ListInsertAfter(numList, node 78, node 44)
ListInsertAfter(numList, node 78, node 58)

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

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

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

1 2 3 4

Check Next

4.3 Doubly-linked lists: Remove

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

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

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

tail nodes, and sucNode and predNode are both null. So, the algorithm points the list's head and tail pointers to null, making the list empty.

PARTICIPATION ACTIVITY 4.3.1: Doubly-linked list: Node removal.

Animation content:

undefined

Animation captions:

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

PARTICIPATION ACTIVITY 4.3.2: Deleting nodes from a doubly-linked list.

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

1) numsList: 71, 29, 54

ListRemove(numsList, node 29)

numsList:

Check Show answer

2) numsList: 2, 8, 1

ListRemove(numsList, list tail)

numsList:

Check Show answer

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

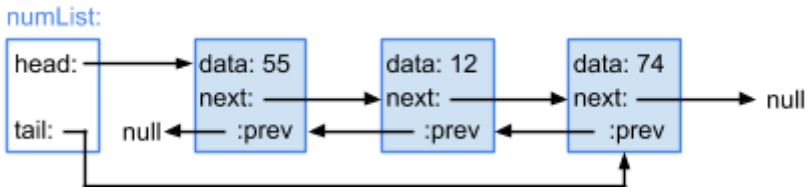
ListRemove(numsList, node 82)
ListRemove(numsList, node 357)
ListRemove(numsList, node 66)

numsList:

Check Show answer

PARTICIPATION ACTIVITY 4.3.3: ListRemove algorithm execution: Intermediate node.

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



1) sucNode⇢prev = predNode

- Yes
- No

2) predNode⇢next = sucNode

- Yes
- No

3) list⇢head = sucNode

- Yes
- No

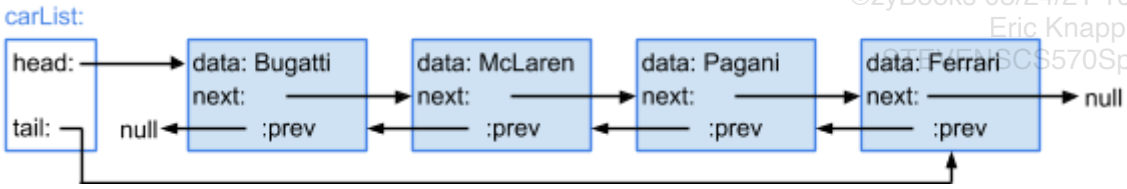
4) list⇢tail = predNode

- Yes

No

PARTICIPATION ACTIVITY 4.3.4: ListRemove algorithm execution: List head node.

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



1) sucNode⇢prev = predNode

- Yes
- No

2) predNode⇢next = sucNode

- Yes
- No

3) list⇢head = sucNode

- Yes
- No

4) list⇢tail = predNode

- Yes
- No

CHALLENGE ACTIVITY 4.3.1: Doubly-linked lists: Remove.

Start

Given list: 5, 2, 8, 9, 6, 3

What list results from the following operations?

ListRemoveAfter(list, node 6)
ListRemoveAfter(list, null)
ListRemoveAfter(list, node 8)

List items in order from head to tail.
Ex: 25, 42, 12

1 2 3 4 5

Check Next

4.4 Linked list traversal

Linked list traversal

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

Figure 4.4.1: Linked list traversal algorithm.

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

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

PARTICIPATION ACTIVITY 4.4.1: Singly-linked list: List traversal.

Animation content:

undefined

Animation captions:

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

PARTICIPATION ACTIVITY 4.4.2: List traversal.

1) ListTraverse begins with _____.
   - ○ a specified list node
   - ○ the list's head node
   - ○ the list's tail node

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

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

Doubly-linked list reverse traversal

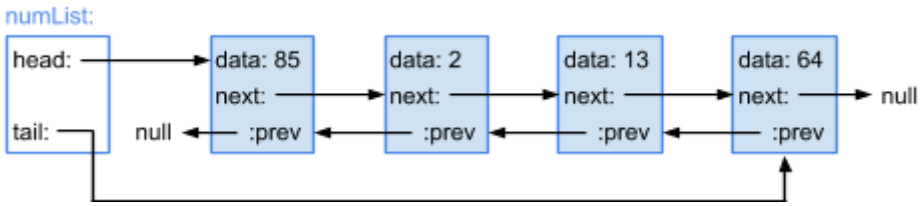
A doubly-linked list also supports a reverse traversal. A ***reverse traversal*** visits all nodes starting with the list's tail node and ending after visiting the list's head node.

Figure 4.4.2: Reverse traversal algorithm.

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

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


PARTICIPATION ACTIVITY 4.4.3: 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

4.5 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 4.5.1: Sorting a doubly-linked list with insertion sort.

Animation content:

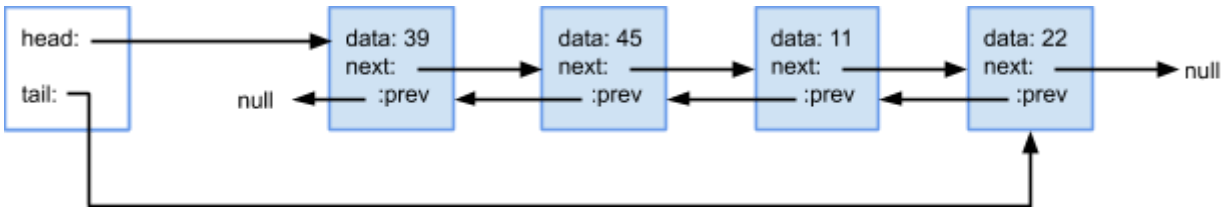
undefined

Animation captions:

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

PARTICIPATION ACTIVITY 4.5.2: Insertion sort for doubly-linked lists.

Suppose ListInsertionSortDoublyLinked is executed to sort the list below.



1) What is the first node that curNode will point to?
   - Node 39
   - Node 45
   - Node 11

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

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

Algorithm efficiency

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

Insertion sort for singly-linked lists

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

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

PARTICIPATION ACTIVITY 4.5.3: Sorting a singly-linked list with insertion sort.

Animation content:

`undefined`

Animation captions:

1. Insertion sort for a singly-linked list initializes curNode to point to the second list element, or node 56.
2. ListFindInsertionPosition searches the list from the head toward the current node to find the insertion position. ListFindInsertionPosition returns null, so Node 56 is prepended as the list head.
3. Node 64 is less than node 71. ListFindInsertionPosition returns a pointer to the node before node 71, or node 56. Then, node 64 is inserted after node 56.
4. The insertion position for node 87 is after node 71. Node 87 is already in the correct position and is not moved.

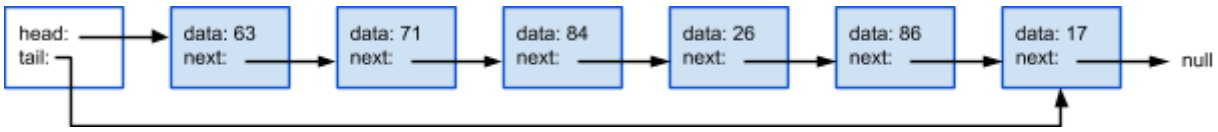
5. Although node 74 is only moved back one position, ListFindInsertionPosition compared node 74 with all other nodes' values to find the insertion position.

Figure 4.5.1: 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 4.5.4: Sorting singly-linked lists with insertion sort.

Given ListInsertionSortSinglyLinked is called to sort the list below.



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

- null
- Node 63
- Node 71
- Node 84

2) How many times is ListPrepend called?

- 0
- 1
- 2

3) How many times is ListInsertAfter called?

- 0
- 1
-

2

Algorithm efficiency

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

Sorting linked-lists vs. arrays

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

Table 4.5.1: 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 4.5.2: Sorting algorithms that cannot as efficiently sort linked lists.

Sorting algorithm	Challenge
Shell sort	Jumping the gap between elements cannot be done on a linked list, as each element between two elements must be traversed.
Quicksort	Partitioning requires backward traversal through the right portion of the array.

	Singly-linked lists do not support backward traversal.
Heap sort	Indexed access is required to find child nodes in constant time when percolating down.

PARTICIPATION ACTIVITY 4.5.5: Sorting linked-lists vs. sorting arrays.

1) What aspect of linked lists makes adapting array-based sorting algorithms to linked lists difficult?
   - Two elements in a linked list cannot be swapped in constant time.
   - Nodes in a linked list cannot be moved.
   - Elements in a linked list cannot be accessed by index.

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

3) Why are sorting algorithms for arrays generally more difficult to adapt to singly-linked lists than to doubly-linked lists?
   - Singly-linked lists do not support backward traversal.
   - Singly-linked do not support inserting nodes at arbitrary locations.

4.6 Linked list dummy nodes

Dummy nodes

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

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

PARTICIPATION ACTIVITY 4.6.1: Singly-linked lists with and without a dummy node.

Animation captions:

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

PARTICIPATION ACTIVITY 4.6.2: 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 4.6.3: Condition for an empty list.

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

- `myList→head == myList→tail`

Singly-linked list implementation

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

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

Figure 4.6.1: Singly-linked list with dummy node: append, prepend, insert after, and remove after operations.

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


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


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


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

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


PARTICIPATION ACTIVITY 4.6.4: Singly-linked list with dummy node.

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

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

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

2) Which is a requirement of the ListPrepend function?

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

PARTICIPATION ACTIVITY 4.6.5: Singly-linked list with dummy node.

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

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

```
lastItem = numbersList→tail
ListAppend(numbersList, node 25)
ListInsertAfter(lastItem, node 49)
```

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

2) Suppose the following statement is executed:

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

Which subsequent operations swap nodes 73 and 19?

- ListPrepend(numbersList, node19)
- ListInsertAfter(numbersList, numbersList⋯→head, node19)
- ListRemoveAfter(numbersList, numbersList⋯→head⋯→next) ListPrepend(numbersList, node19)

Doubly-linked list implementation

A dummy node can also be used in a doubly-linked list implementation. The dummy node in a doubly-linked list always has the prev pointer set to null. ListRemove's implementation does not allow removal of the dummy node.

Figure 4.6.2: Doubly-linked list with dummy node: append, prepend, insert after, and remove operations.

©zyBooks 03/24/21 10:58 926027
Eric Knapp
STEVENSCS570Spring2021

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


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

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

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

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


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


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

   sucNode = curNode→next
   predNode = curNode→prev

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

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

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


©zyBooks 03/24/21 10:58 926027
Eric Knapp
STEVENSCS570Spring2021

PARTICIPATION ACTIVITY 4.6.6: Doubly-linked list with dummy node.

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

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

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

Dummy head and tail nodes

A doubly-linked list implementation can also use 2 dummy nodes: one at the head and the other at the tail. Doing so removes additional conditionals and further simplifies the implementation of most methods.

PARTICIPATION ACTIVITY 4.6.7: Doubly-linked list append and prepend with 2 dummy nodes.

Animation content:

`undefined`

Animation captions:

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

Figure 4.6.3: Doubly-linked list with 2 dummy nodes: insert after and remove operations.

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

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

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

   sucNode = curNode→next
   predNode = curNode→prev

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

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


Removing if statements from ListInsertAfter and ListRemove

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

PARTICIPATION ACTIVITY 4.6.8: Comparing a doubly-linked list with 1 dummy node vs. 2 dummy nodes.

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

1) When list→head == list→tail is true in _____, the list is empty.

   - ○ a list with 1 dummy node

- a list with 2 dummy nodes
- either a list with 1 dummy node or a list with 2 dummy nodes

2) list⋯→tail may be null in _____.

- a list with 1 dummy node
- a list with 2 dummy nodes
- neither list type

3) list⋯→head⋯→next is always non-null in _____.

- a list with 1 dummy node
- a list with 2 dummy nodes
- neither list type

4.7 List data structure

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

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

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

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

PARTICIPATION ACTIVITY 4.7.1: Linked lists can be stored with or without a list data structure.

Animation content:

```
undefined
```

Animation captions:

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

PARTICIPATION ACTIVITY 4.7.2: Linked list data structure.

1) A linked list must have a list data structure.
   - True
   - False

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

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

4.8 Circular lists

A ***circular linked list*** is a linked list where the tail node's next pointer points to the head of the list, instead of null. A circular linked list can be used to represent repeating processes. Ex: Ocean water evaporates, forms clouds, rains down on land, and flows through rivers back into the ocean. The head of a circular linked list is often referred to as the *start* node.

A traversal through a circular linked list is similar to traversal through a standard linked list, but must terminate after reaching the head node a second time, as opposed to terminating when reaching null.

PARTICIPATION ACTIVITY 4.8.1: Circular list structure and traversal.

Animation content:

```
undefined
```

Animation captions:

1. In a circular linked list, the tail node's next pointer points to the head node.
2. In a circular doubly-linked list, the head node's previous pointer points to the tail node.
3. Instead of stopping when the "current" pointer is null, traversal through a circular list stops when current comes back to the head node.

PARTICIPATION ACTIVITY 4.8.2: Circular list concepts.

1) Only a doubly-linked list can be circular.
   - ○ True
   - ○ False

2) In a circular doubly-linked list with at least 2 nodes, where does the head node's previous pointer point to?
   - ○ List head
   - ○ List tail
   - ○ null

3) In a circular linked list with at least 2 nodes, where does the tail node's next pointer point to?
   - ○ List head
   - ○ List tail
   - ○ null

4) In a circular linked list with 1 node, the tail node's next pointer points to the tail.
   - ○ True
   - ○ False

5) The following code can be used to traverse a circular, doubly-linked list in

reverse order.

```
CircularListTraverseReverse(tail) {
   if (tail is not null) {
      current = tail
      do {
         visit current
         current = current→previous
      } while (current != tail)
   }
}
```

- True
- False

©zyBooks 03/24/21 10:58 926027
Eric Knapp
STEVENSCS570Spring2021