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

PARTICIPATION ACTIVITY 4.1.1: List ADT.

Animation captions:

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

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

PARTICIPATION ACTIVITY 4.1.3: List ADT common operations.

1) Given a list with items 40, 888, -3, 2, what does GetLength(list) return?
   - 4
   - Fails

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

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

- True
- False

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.

null

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

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

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

Animation content:

undefined

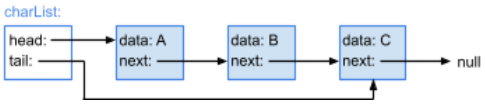
Animation captions:

1. A new list item is created, with the head and tail pointers pointing to nothing (null), meaning the list is empty.
2. ListAppend points the list's head and tail pointers to a new node, whose next pointer points to null.
3. Another append points the last node's next pointer and the list's tail pointer to the new node.

4. Operations like ListAppend, ListPrepend, ListInsertAfter, and ListRemove, update just a few relevant pointers.
5. The list's first node is called the head. The last node is the tail.

PARTICIPATION ACTIVITY 4.2.2: Singly-linked list data structure.

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

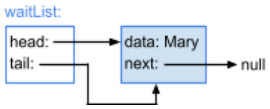


Check Show answer

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

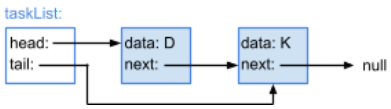
Check Show answer

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



Check Show answer

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



Check Show answer

Appending a node to a singly-linked list

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

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

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

PARTICIPATION ACTIVITY 4.2.3: Singly-linked list: Appending a node.

Animation content:

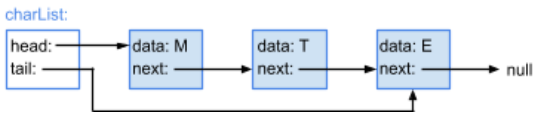
undefined

Animation captions:

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

PARTICIPATION ACTIVITY 4.2.4: Appending a node to a singly-linked list.

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

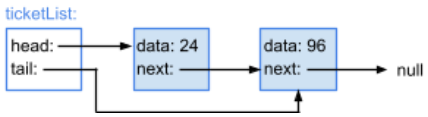


- M
- T
- E

2) Appending node W to sampleList updates which of sampleList's pointers?

- head and tail
- head only
- tail only

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



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

Prepending a node to a singly-linked list

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

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

PARTICIPATION ACTIVITY 4.2.5: Singly-linked list: Prepending a node.

Animation content:

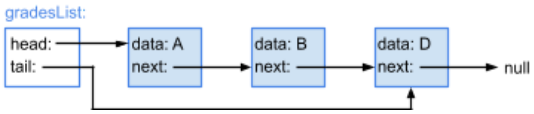
undefined

Animation captions:

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

PARTICIPATION ACTIVITY 4.2.6: Prepending a node in a singly-linked list.

1) Prepending C to gradesList updates which pointer?



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

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

- True
- False

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

- True
- False

4) Prepending Evelyn to deliveryList executes which statement?

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

CHALLENGE ACTIVITY 4.2.1: Singly-linked lists.

502696.2096894.qx3zqy7

Start

What is numList after the following operations?

numList = new List

ListAppend(numList, node 23)
ListAppend(numList, node 11)
ListAppend(numList, node 32)

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

1 2 3 4 5

Check Next

4.3 Singly-linked lists: Insert

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

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

PARTICIPATION ACTIVITY 4.3.1: Singly-linked list: Insert nodes.

Animation content:

undefined

Animation captions:

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

PARTICIPATION ACTIVITY 4.3.2: Inserting nodes in a singly-linked list.

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

1) numsList: 5, 9

ListInsertAfter(numsList, node 9,
node 4)

numsList:

Check Show answer

2) numsList: 23, 17, 8

ListInsertAfter(numsList, node 23, node 5)

numsList:

Check Show answer

3) numsList: 1

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

numsList:

Check Show answer

4) numsList: 77

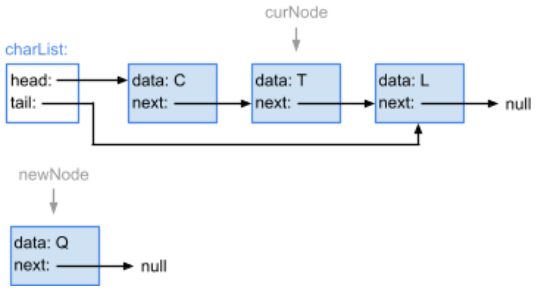
ListInsertAfter(numsList, node 77, node 32)
ListInsertAfter(numsList, node 32, node 50)
ListInsertAfter(numsList, node 32, node 46)

numsList:

Check Show answer

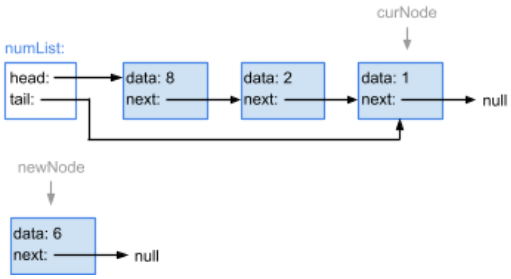
PARTICIPATION ACTIVITY 4.3.3: Singly-linked list insert-after algorithm.

1) ListInsertAfter(charList, node T, node Q) assigns newNode's next pointer with _____.



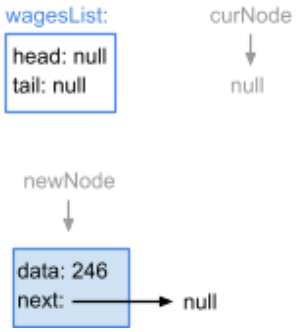
- curNode⇢next
- charList's head node
- null

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



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

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



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

CHALLENGE ACTIVITY 4.3.1: Singly-linked lists: Insert.

502696.2096894.qx3zqy7

Start

What is numList after the following operations?

numList: 14, 33
ListInsertAfter(numList, node 33, node 55)
ListInsertAfter(numList, node 55, node 57)
ListInsertAfter(numList, node 55, node 38)

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

1 2 3 4

Check Next

4.4 Singly-linked lists: Remove

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

The existing node is specified with the curNode parameter. If curNode is null, RemoveAfter removes the list's first node. Otherwise, the algorithm removes the node after curNode.

- *Remove list's head node (special case):* If curNode is null, the algorithm points sucNode to the head node's next node, and points the list's head pointer to sucNode. If sucNode is null, the only list node was removed, so the list's tail pointer is pointed to null (indicating the list is now empty).
- *Remove node after curNode:* If curNode's next pointer is not null (a node after curNode exists), the algorithm points sucNode to the node after curNode's next node. Then curNode's next pointer is

pointed to sucNode. If sucNode is null, the list's tail node was removed, so the algorithm points the list's tail pointer to curNode (the new tail node).

PARTICIPATION ACTIVITY 4.4.1: Singly-linked list: Node removal.

Animation content:

Static figure:
Begin pseudocode:

```
ListRemoveNodeAfter(list, curNode) {
  // Special case, remove head
  if (curNode is null && list⇢head is not 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&pos;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

ListRemoveAfter(numsList, node 5)

numsList:

Check Show answer

2) numsList: 3, 57, 28, 40

ListRemoveAfter(numsList, null)

numsList:

Check Show answer

3) numsList: 9, 4, 11, 7

ListRemoveAfter(numsList, node 11)

numsList:

Check Show answer

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

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

numsList:

Check Show answer

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

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

numsList:

Check Show answer

PARTICIPATION ACTIVITY 4.4.3: ListRemoveAfter algorithm execution: Intermediate node.

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

1) sucNode = list⋯→head⋯→next

- Yes
- No

2) curNode⋯→next = sucNode

- Yes
- No

3) list⋯→head = sucNode

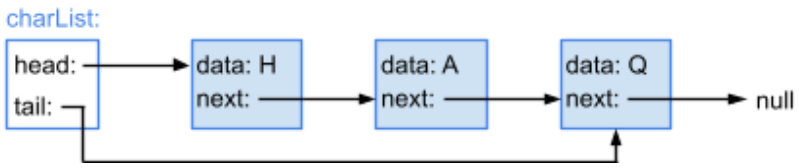
- Yes
- No

4) list⋯→tail = curNode

- Yes
- No

PARTICIPATION ACTIVITY 4.4.4: ListRemoveAfter algorithm execution: List head node.

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



1) sucNode = list⋯→head⋯→next

- Yes
- No

2) curNode⋯→next = sucNode

- Yes
- No

3) list⋯→head = sucNode

- Yes
- No

4) list⋯→tail = curNode

- Yes
- No

CHALLENGE ACTIVITY 4.4.1: Singly-linked lists: Remove.

502696.2096894.qx3zqy7

Start

Given list: 8, 5, 2, 7, 1
What list results from the following operations?

ListRemoveAfter(list, node 7)
ListRemoveAfter(list, node 5)
ListRemoveAfter(list, null)

List items in order, from head to tail.

Ex: 25, 42, 12

1 2 3 4 5

Check Next

4.5 Linked list search

Given a key, a ***search*** algorithm returns the first node whose data matches that key, or returns null if a matching node was not found. A simple linked list search algorithm checks the current node (initially the list's head node), returning that node if a match, else pointing the current node to the next node and repeating. If the pointer to the current node is null, the algorithm returns null (matching node was not found).

PARTICIPATION ACTIVITY 4.5.1: Singly-linked list: Searching.

Animation content:

undefined

Animation captions:

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

PARTICIPATION ACTIVITY 4.5.2: ListSearch algorithm execution.

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

Check Show answer

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

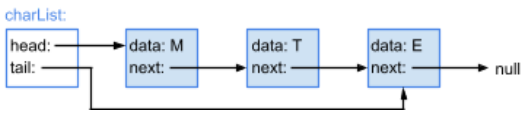
Check Show answer

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

Check Show answer

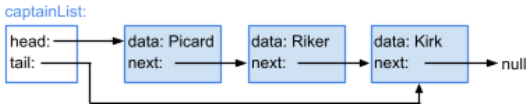
PARTICIPATION ACTIVITY 4.5.3: Searching a linked-list.

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



- Node M
- Node T
- Node E

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



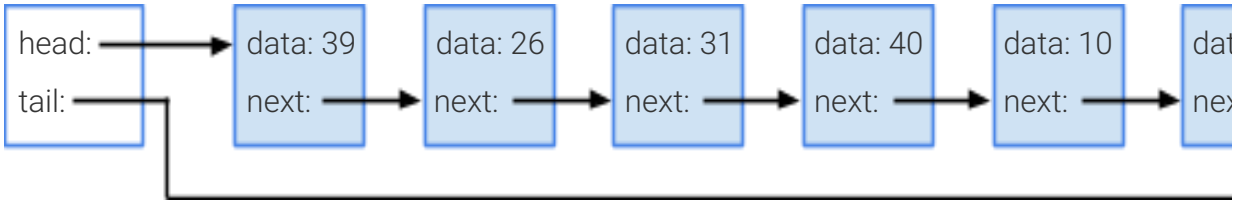
- node Riker
- node Kirk

CHALLENGE ACTIVITY 4.5.1: Linked list search.

502696.2096894.qx3zqy7

Start

numList:



ListSearch(numList, 13) points the current pointer to node Ex: 9 after checking node 31.

ListSearch(numList, 13) will make comparisons.

1 2

Check Next

4.6 Python: Singly-linked lists

Constructing the node and list class

In Python, classes are used for both the linked list and the nodes that comprise the list. Each class includes references to nodes (next node for the Node class and head and tail nodes for the LinkedList

class).

The Node class implements a list node with two data members, a data value and the next node in the list. If the node has no next node, the next data member is assigned with None, the Python term signifying the absence of a value.

The LinkedList class implements the list data structure and contains two data members, head and tail, which are assigned to nodes once the list is populated. Initially the list has no nodes, so both data members are initially assigned with None.

Figure 4.6.1: Node class definition and node initialization.

Class definition	Node initialization

`class Node:`
`  def __init__(self, initial_data):`
`  self.data = initial_data`
`  self.next = None`
	`node_a = Node(95)  # Creates a node with a data value of 95`

Figure 4.6.2: Singly-linked list class definition and linked list initialization.

Class definition	LinkedList initialization

`class LinkedList:`
`  def __init__(self):`
`  self.head = None`
`  self.tail = None`
	`num_list = LinkedList()  # Creates an empty linked list`

PARTICIPATION ACTIVITY 4.6.1: Node and LinkedList class.

1) The Node class has two data members.

- True
- False

2) The data value of a node is set to None if the node is not in a list.

- True
- False

3) An empty LinkedList has a single node with a data value of None.

- True
- False

Appending a node to a singly-linked list

The append() method is a method within the LinkedList class. The append() method adds a node to the end of a linked list, making the node the tail of the list. The append() method has two parameters, the second of which is the new node to be appended to the list. Note that the definition for append() and all other class methods contain a self parameter. This parameter refers to the class object itself, and thus is not passed as an argument to the class methods.

Figure 4.6.3: LinkedList append() method and method call.

append() method definition

```
def append(self,
new_node):
    if self.head == None:
        self.head =
new_node
        self.tail =
new_node
    else:
        self.tail.next =
new_node
        self.tail =
new_node
```

append() method call

```
node_a = Node(95)    # Creates a node with a
data value of 95
node_b = Node(42)    # Creates a node with a
data value of 42

num_list.append(node_a)  # Method call has
one argument
num_list.append(node_b)
```


PARTICIPATION ACTIVITY 4.6.2: LinkedList append() method.

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

- head
- tail

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

-
```
node_a = Node(15)
num_list.append(self,
node_a)
```

-
```
node_a = Node(15)
num_list.append(node_a)
```

-
```
node_a = Node(15)
node_a.append(num_list)
```

-
```
node_a = Node(15)
append(num_list, node_a)
```


Additional singly-linked list methods

The following methods are implemented within the LinkedList class.

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

Figure 4.6.4: LinkedList prepend() method.

```
def prepend(self, new_node):
    if self.head == None:
        self.head = new_node
        self.tail = new_node
    else:
        new_node.next =
self.head
        self.head = new_node
```

The insert_after() method adds a node after an existing node by assigning the new node's next data member to the existing node's next data member and assigning the existing node's next data member to the new node. The method takes three parameters: self, the existing node (current_node), and the new node to be inserted. The Python identity operator "is" is used to see if current_node is the same node object as self.tail, thus testing that current_node and self.tail share the same memory location.

Figure 4.6.5: LinkedList insert_after() method.

```
def insert_after(self, current_node,
new_node):
    if self.head == None:
        self.head = new_node
        self.tail = new_node
    elif current_node is self.tail:
        self.tail.next = new_node
        self.tail = new_node
    else:
        new_node.next = current_node.next
        current_node.next = new_node
```


The remove_after() method removes a node after a specified node by assigning the node's next value to the specified node's next data member. The method's second parameter is the node before the node to be removed.

Figure 4.6.6: LinkedList remove_after() method.

```
def remove_after(self, current_node):
    # Special case, remove head
    if (current_node == None) and (self.head !=
None):
        succeeding_node = self.head.next
        self.head = succeeding_node
        if succeeding_node == None: # Remove last
item
            self.tail = None
    elif current_node.next != None:
        succeeding_node = current_node.next.next
        current_node.next = succeeding_node
        if succeeding_node == None: # Remove tail
            self.tail = current_node
```

PARTICIPATION ACTIVITY 4.6.3: Additional LinkedList methods.

1) Which variable is a parameter of prepend()?

- new_node
- current_node

2) Which operator is used to test if two objects share the same memory location?

- ==
- in
- is

3) In remove_after(), what does the variable current_node signify?

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

zyDE 4.6.1: Singly linked-list data structures and algorithms.

The above code has been split into three files, main.py, Node.py, and LinkedList.py.

Current file: main.py Load default template

```
1  from Node import Node
2  from LinkedList import LinkedList
3
4
5  num_list = LinkedList()
6
7  node_a = Node(66)
8  node_b = Node(99)
9  node_c = Node(44)
10 node_d = Node(95)
11 node_e = Node(42)
12 node_f = Node(17)
13
14 num_list.append(node_b)   # Add 99
15 num_list.append(node_c)   # Add 44, make the tail
```

Run

4.7 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 4.7.1: 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 4.7.2: Doubly-linked list: Appending a node.

Animation content:

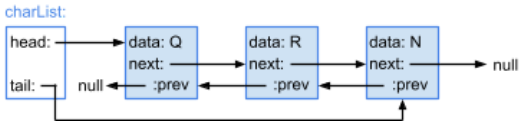
undefined

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.

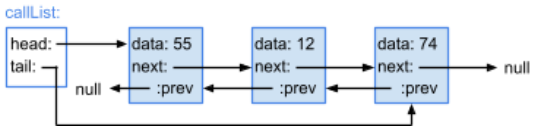
PARTICIPATION ACTIVITY 4.7.3: Doubly-linked list data structure.

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



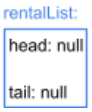
- after node Q
- before node N
- after node N

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 4.7.4: 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 4.7.5: Prepending a node in a doubly-linked list.

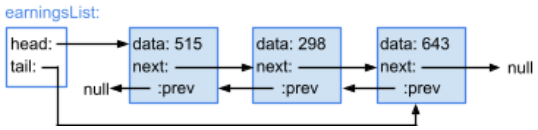
1) Prepending 29 to trainsList updates the list's head pointer to point to node _____.

- 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 4.7.1: Doubly-linked lists.

502696.2096894.qx3zqy7

Start

numList = new List
ListAppend(numList, node 22)
ListAppend(numList, node 78)

ListAppend(numList, node 77)
ListAppend(numList, node 61)
numList is now: Ex: 1, 2, 3 (comma between values)

Which node has a null previous pointer? Ex: 5

Which node has a null next pointer? Ex: 5

Check Next

4.8 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 4.8.1: 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 4.8.2: Inserting nodes in a doubly-linked list.

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

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

If unable to drag and drop, refresh the page.

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

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

Reset

CHALLENGE ACTIVITY 4.8.1: Doubly-linked lists: Insert.

502696.2096894.qx3zqy7

Start

What is numList after the following operations?

numList: 84, 81
ListInsertAfter(numList, node 84, node 10)
ListInsertAfter(numList, node 81, node 29)
ListInsertAfter(numList, node 29, node 59)
ListInsertAfter(numList, node 29, node 46)

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

What node does node 29's next pointer point to?

What node does node 29's previous pointer point to?

1 2 3 4

Check Next

4.9 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 4.9.1: 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 4.9.2: 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

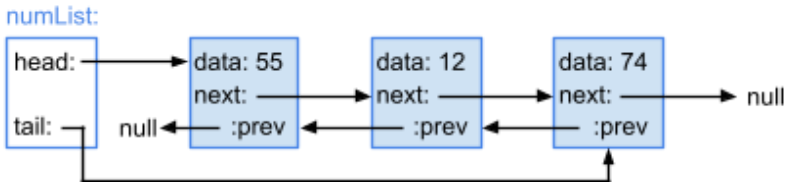
ListRemove(numsList, node 82)
ListRemove(numsList, node 357)
ListRemove(numsList, node 66)

numsList:

Check Show answer

PARTICIPATION ACTIVITY 4.9.3: 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

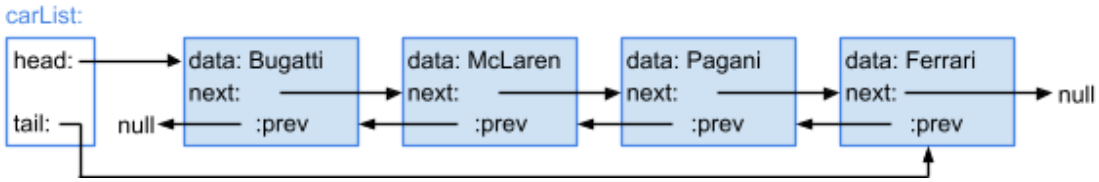
- Yes
- No

4) list⇢tail = predNode

- Yes
- No

PARTICIPATION ACTIVITY 4.9.4: 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 4.9.1: Doubly-linked lists: Remove.

502696.2096894.qx3zqy7

Start

Given numList: 6, 7, 1, 2, 5, 9
What is numList after the following operations?

ListRemove(numList, node 9)
ListRemove(numList, node 1)
ListRemove(numList, node 6)

numList is now: Ex: 25, 42, 12

1 2 3 4 5

Check Next

4.10 Python: Doubly-linked lists

Adding a reference to the previous node

In a previous section, the LinkedList class was defined, making use of the Node class. The Node class defined previously can be extended from the singly-linked list version to include a reference variable called prev that refers to the previous node in the list. When a new node is first constructed, the prev variable is assigned with None.

Creating a doubly-linked node or a doubly-linked list is still the same as creating a singly-linked node and a singly-linked list.

Figure 4.10.1: Node class definition for a doubly-linked node.

```
class Node:
    def __init__(self,
initial_data):
        self.data = initial_data
        self.next = None
        self.prev = None
```


Figure 4.10.2: Creating a doubly-linked node and doubly-linked list.

```
node_1 = Node(95)
num_list =
LinkedList()
```

PARTICIPATION ACTIVITY 4.10.1: Doubly-linked lists and nodes.

1) A doubly-linked node has both next and prev data members.
   - True
   - False

2) The first node in a doubly-linked list has a prev value of None.
   - True
   - False

Appending a node to a doubly-linked list

LinkedList's append() method for doubly-linked nodes is similar to the same method for singly-linked nodes, but has an added line of code. Before the method assigns the list's tail with the new_node, the method assigns the new node's prev variable with the old tail of the list: `new_node.prev = self.tail`.

Figure 4.10.3: LinkedList append() method using doubly-linked nodes.

```
def append(self, new_node):
    if self.head == None:
        self.head = new_node
        self.tail = new_node
    else:
        self.tail.next =
new_node
        new_node.prev =
self.tail
        self.tail = new_node
```


PARTICIPATION ACTIVITY 4.10.2: Appending to a doubly-linked list.

Assume the code below has been executed using a doubly-linked list.

```
word_list = LinkedList()
node_a = Node("Python")
node_b = Node("Is")
node_c = Node("Fun!")

word_list.append(node_a)
word_list.append(node_b)
word_list.append(node_c)
```

1) For `node_b` what is the `prev` data member ?

- node_a
- node_b
- node_c
- None

2) Which node does node_c's next data member refer to?

- node_a
- node_b
- node_c
- None of the above; the variable's value is None.

Additional doubly-linked list methods

The prepend() method inserts a new node at the head of a doubly-linked list , making the new node the head of the list. Additionally, the prev variable of the old head node must be set to point to the new node:

`self.head.prev = new_node`.

Figure 4.10.4: LinkedList prepend() method using doubly-linked nodes.

```
def prepend(self, new_node):
   if self.head == None:
      self.head = new_node
      self.tail = new_node
   else:
      new_node.next =
self.head
      self.head.prev =
new_node
      self.head = new_node
```


The insert_after() method requires two node parameters: the new node to be inserted, and the node already in the list that will come immediately before the new node.

Three nodes can be affected by the insert_after() method:

- The new node (new_node), which needs both next and prev data members to be assigned.
- The existing node (current_node), whose next data member must be assigned.
- The node that currently follows current_node in the list (successor_node), whose prev data member must be assigned.

In the general case, the new node's prev is assigned with current_node, the new node's next is assigned with successor_node, the current node's next is assigned with new_node, and the successor node's prev is assigned with new_node.

Two special cases need to be handled as well: 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 4.10.5: LinkedList insert_after() method using doubly-linked nodes.

```
def insert_after(self, current_node,
new_node):
    if self.head is None:
        self.head = new_node
        self.tail = new_node
    elif current_node is self.tail:
        self.tail.next = new_node
        new_node.prev = self.tail
        self.tail = new_node
    else:
        successor_node = current_node.next
        new_node.next = successor_node
        new_node.prev = current_node
        current_node.next = new_node
        successor_node.prev = new_node
```


The remove() method unlinks a given node (current_node) from the list, joining together the node before the removed node (predecessor_node) and the node after the removed node (successor_node).

If either the head or tail of the list is removed, the method updates the lists' LinkedList head and tail data members .

Figure 4.10.6: LinkedList remove() method using doubly-linked nodes.

```
def remove(self, current_node):
    successor_node = current_node.next
    predecessor_node = current_node.prev

    if successor_node is not None:
        successor_node.prev =
predecessor_node

    if predecessor_node is not None:
        predecessor_node.next =
successor_node

    if current_node is self.head:
        self.head = successor_node

    if current_node is self.tail:
        self.tail = predecessor_node
```


PARTICIPATION ACTIVITY 4.10.3: Additional LinkedList methods using doubly-linked nodes.

1) Which LinkedList method inserts a new node at the beginning of a list?

Check Show answer

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

```
num_list = LinkedList()
node_a = Node(1)
node_b = Node(2)
node_c = Node(3)
node_d = Node(4)
node_e = Node(5)

num_list.append(node_a)
num_list.append(node_b)
num_list.append(node_c)
num_list.prepend(node_d)
num_list.insert_after(node_a,
node_e)
num_list.remove(node_c)
```

Check Show answer

zyDE 4.10.1: Doubly linked-list data structures and algorithms.

The code from this section has been split into three files, main.py, Node.py, and LinkedList

Current file: main.py Load default template

```
1  from Node import Node
2  from LinkedList import LinkedList
3
4
5  num_list = LinkedList()
6
7  node_a = Node(14)
8  node_b = Node(2)
9  node_c = Node(20)
10 node_d = Node(31)
11 node_e = Node(16)
12 node_f = Node(55)
13
14 num_list.append(node_a)  # Add 14
```

```
15 num_list.append(node_b)    # Add 2, make the tail
```

Run

4.11 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 4.11.1: 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 4.11.1: 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 4.11.2: 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

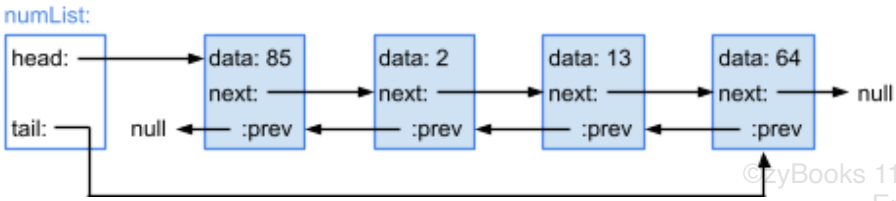
A doubly-linked list also supports a reverse traversal. A ***reverse traversal*** visits all nodes starting with the list's tail node and ending after visiting the list's head node.

Figure 4.11.2: 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 4.11.3: 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

4.12 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 4.12.1: Sorting a doubly-linked list with insertion sort.

Animation content:

undefined

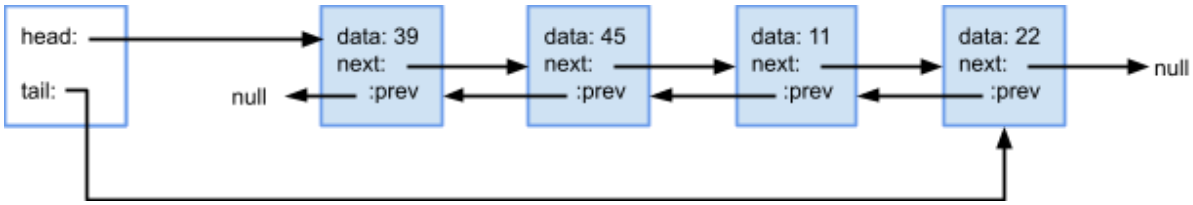
Animation captions:

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 4.12.2: 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( ). 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 times. So the total number of comparisons is proportional to , or O(

). In the best case scenario, the list is already sorted, and the runtime complexity is O( ).

Insertion sort for singly-linked lists

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

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

PARTICIPATION ACTIVITY 4.12.3: Sorting a singly-linked list with insertion sort.

Animation content:

undefined

Animation captions:

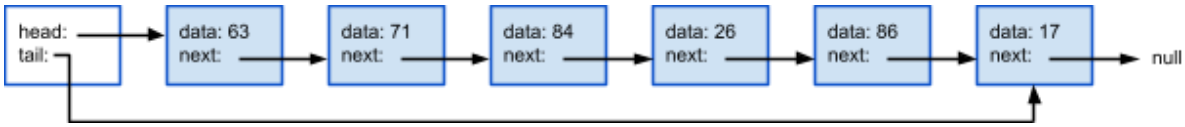
1. Insertion sort for a singly-linked list initializes curNode to point to the second list element, or node 56.
2. ListFindInsertionPosition searches the list from the head toward the current node to find the insertion position. ListFindInsertionPosition returns null, so Node 56 is prepended as the list 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 4.12.1: 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 4.12.4: 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( ). The best case runtime is O( ), which occurs when the list is sorted in descending order.

Sorting linked-lists vs. arrays

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

Table 4.12.1: 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 4.12.2: Sorting algorithms difficult to adapt to efficiently sort linked lists.

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

PARTICIPATION ACTIVITY 4.12.5: 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.

©zyBooks 11/20/23 11:00 1048447
Eric Quezada
UTEPCS2302ValeraFall2023

4.13 Python: Sorting linked lists

Python insertion sort algorithm for doubly-linked lists

Insertion sort for a doubly-linked list visits all list nodes and shifts the nodes backward to be placed in the correctly sorted positions. The list is traversed backward to find a node's new insertion point. In Python,

the doubly-linked list insertion sort is implemented as a method of the doubly-linked list LinkedList class because a singly-linked list cannot be traversed backward. To move a node backward to an earlier position in the list, the node is removed using the remove() method and re-inserted using either the prepend() (if the node becomes the head) or insert_after() method.

Figure 4.13.1: Insertion sort algorithm for doubly-linked lists.


```
def
insertion_sort_doubly_linked(self):
    current_node = self.head.next
    while current_node != None:
        next_node =
current_node.next
        search_node =
current_node.prev

        while ((search_node !=
None) and
               (search_node.data >
current_node.data)):
            search_node =
search_node.prev

        # Remove and re-insert
curNode
        self.remove(current_node)

        if search_node == None:
            current_node.prev =
None

self.prepend(current_node)
        else:

self.insert_after(search_node,
current_node)

        # Advance to next node
        current_node = next_node
```

```
from LinkedList import LinkedList
from Node import Node

num_list = LinkedList()
node_a = Node(72)
node_b = Node(91)
node_c = Node(53)
node_d = Node(12)

num_list.append(node_a)
num_list.append(node_b)
num_list.append(node_c)
num_list.append(node_d)

# Output list
print('List after adding nodes:',
end=' ')
node = num_list.head
while node != None:
    print(node.data, end=' ')
    node = node.next
print()

num_list.insertion_sort_doubly_linked()

# Output list
print('List after insertion sort:',
end=' ')
node = num_list.head
while node != None:
    print(node.data, end=' ')
    node = node.next
print()
```

```
List after adding nodes: 72 91 53 12
List after insertion sort: 12 53 72 91
```


PARTICIPATION ACTIVITY 4.13.1: Doubly-linked list insertion sort.

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

- remove_after() and append()
- remove() and prepend()

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

- Lacking a `prev` data member
- Having a `next` data member.

Python insertion sort variant for singly-linked lists

Insertion sort can be changed to also work for singly-linked list by traversing the list in a forwards manner to identify a node's insertion point and then using methods prepend() or insert_after() to move the node backward to the insertion point. In Python, the singly-linked list insertion sort is implemented as a method of the singly-linked list LinkedList class. The insertion_sort_singly_linked() method calls the find_insertion_position() method to move nodes to find where the current node should be located. find_insertion_position() is also a method of the LinkedList class.

Figure 4.13.2: Insertion sort algorithm for singly-linked lists.

```
def insertion_sort_singly_linked(self):
    before_current = self.head
    current_node = self.head.next
    while current_node != None:
        next_node = current_node.next
        position =
self.find_insertion_position(current_node.data)
        if position == before_current:
            before_current = current_node
        else:
            self.remove_after(before_current)
            if position == None:
                self.prepend(current_node)
            else:
                self.insert_after(position, current_node)
        current_node = next_node

def find_insertion_position(self, data_value):
    position_a = None
    position_b = self.head
    while (position_b != None) and (data_value >
position_b.data):
        position_a = position_b
        position_b = position_b.next
    return position_a
```

PARTICIPATION ACTIVITY 4.13.2: Singly-linked list insertion sort variant.

1) The current node to be sorted is placed _____ the location returned by find_insertion_position().

   - ○ before
   - ○ after

2) The insertion_sort_singly_linked() method would also sort a doubly-linked list.

   - ○ True
   - ○ False

zyDE 4.13.1: Insertion sort algorithm for a doubly-linked list data structure.

Current file: main.py Load default template

```
1  from LinkedList import LinkedList
2  from Node import Node
3
4  num_list = LinkedList()
5  node_a = Node(72)
6  node_b = Node(91)
7  node_c = Node(53)
8  node_d = Node(12)
9
10 num_list.append(node_a)
11 num_list.append(node_b)
12 num_list.append(node_c)
13 num_list.append(node_d)
14
15 # Output list
```


Run

zyDE 4.13.2: Insertion sort algorithm for a singly-linked list data structure.

Current file: main.py Load default template

```
1  from LinkedList import LinkedList
2  from Node import Node
3
4  num_list = LinkedList()
5  node_a = Node(54)
6  node_b = Node(2)
7  node_c = Node(72)
8  node_d = Node(36)
9
10 num_list.append(node_a)
11 num_list.append(node_b)
12 num_list.append(node_c)
13 num_list.append(node_d)
14
15 # Output list
```


Run

4.14 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 4.14.1: 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 4.14.2: 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 4.14.3: 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 4.14.1: 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 4.14.4: 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 4.14.5: Singly-linked list with dummy node.

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

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

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

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

2) Suppose the following statement is executed:
`node19 = numbersList→head→next→next`
Which subsequent operations swap nodes 73 and 19?

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

Doubly-linked list implementation

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

Figure 4.14.2: Doubly-linked list with dummy node: append, prepend, insert after, and remove operations.

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


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

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

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

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


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


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

   sucNode = curNode→next
   predNode = curNode→prev

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

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

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


©zyBooks 11/20/23 11:00 1048447
Eric Quezada
UTEPCS2302ValeraFall2023

PARTICIPATION ACTIVITY 4.14.6: 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 4.14.7: 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 4.14.3: 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 4.14.8: 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

4.15 Linked lists: Recursion

Forward traversal

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

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

PARTICIPATION ACTIVITY 4.15.1: Recursive forward traversal.

Animation content:

undefined

Animation captions:

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

PARTICIPATION ACTIVITY 4.15.2: Forward traversal in a linked list with 10 nodes.

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

PARTICIPATION ACTIVITY 4.15.3: Forward traversal concepts.

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

Searching

A recursive linked list search is implemented similar to forward traversal. Each call examines 1 node. If the node is null, then null is returned. Otherwise, the node's data is compared to the search key. If a match occurs, the node is returned, otherwise the remainder of the list is searched recursively.

Figure 4.15.1: 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 4.15.4: Searching a linked list with 10 nodes.

Suppose a linked list has 10 nodes.

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

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

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

Reverse traversal

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

PARTICIPATION ACTIVITY 4.15.5: Recursive reverse traversal.

Animation content:

undefined

Animation captions:

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

PARTICIPATION ACTIVITY 4.15.6: 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

4.16 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, such as append, prepend, insert after, remove, and search.

In many programming languages, arrays have a fixed size. An array-based list implementation will dynamically allocate the array as needed as the number of elements changes. Initially, the array-based list implementation allocates a fixed size array and uses a length variable to keep track of how many array elements are in use. The list starts with a default allocation size, greater than or equal to 1. A default size of 1 to 10 is common.

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 4.16.1: Appending to array-based lists.

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 4.16.2: 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 allocationSize = 8 and length = 4. Appending assigns the array at index _____ with 42.
   - ○ 4
   - ○ 8

3) An array-based list can have a default allocation size of 0.
   - ○ 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. Allocating the new array with twice the current length is a common approach. 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 1 array to another, the resize operation has a runtime complexity of O( ).

PARTICIPATION ACTIVITY 4.16.3: Array-based list resize operation.

Animation content:

undefined

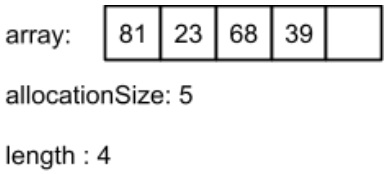
Animation captions:

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 4.16.4: Array-based list resize operation.

Assume the following operations are executed on the list shown below:

ArrayListAppend(list, 98)
ArrayListAppend(list, 42)
ArrayListAppend(list, 63)



1) Which operation causes ArrayListResize to be called?
   - ArrayListAppend(list, 98)
   - ArrayListAppend(list, 42)
   - ArrayListAppend(list, 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

Prepend and insert after 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 by 1 position, and the new item is inserted at the list start, or index 0. Because all existing array elements must be moved up by 1, the prepend operation has a runtime complexity of O( ).

The ***InsertAfter*** operation for an array-based list inserts a new item after a specified index. Ex: If the contents of `numbersList` is: 5, 8, 2, `ArrayListInsertAfter(numbersList, 1, 7)` produces: 5, 8, 7, 2. First, if the allocation size equals the list length, the array is resized. Next, all elements in the array

residing after the specified index are moved up by 1 position. Then, the new item is inserted at index (specified index + 1) in the list's array. The InsertAfter operation has a best case runtime complexity of O( ) and a worst case runtime complexity of O( ).

InsertAt operation.

Array-based lists often support the InsertAt operation, which inserts an item at a specified index. Inserting an item at a desired index X can be achieved by using InsertAfter to insert after index X - 1.

PARTICIPATION ACTIVITY 4.16.5: Array-based list prepend and insert after operations.

Animation content:

Step 1: Data members for a list are shown: array, allocationSize, and length.
"array:" label is followed by 8 boxes for the array's data. Entries at indices 0 to 4 are: 45, 84, 12, 78, 51. Entries at indices 5, 6, and 7 are empty.
The other two labels are "allocationSize: 8" and "length: 5".
ArrayListPrepend(list, 91) begins execution. The first if statement's condition is false. Then the for loop executes, moving items up in the array, yielding: 45, 45, 84, 12, 78, 51. Array boxes for indices 6 and 7 remain empty.

Step 2: Item 91 is assigned to index 0 in the array, yielding: 91, 45, 84, 12, 78, 51. Array boxes for indices 6 and 7 remain empty. Then length is incremented to 6.

Step 3: ArrayListInsertAfter(list, 2, 36) executes. The first if statement's condition is false, since 6 ≠ 8. The for loop moves items at indices 3, 4, and 5 up one, yielding: 91, 45, 84, 12, 12, 78, 51. Then 36 is assigned to index 3, yielding: 91, 45, 84, 36, 12, 78, 51. The array box for index 7 remains empty. Lastly, length is incremented to 7.

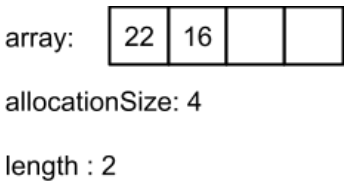
Animation captions:

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 after index 2 requires elements at indices 3 and higher to be moved up 1. Item 36 is inserted at index 3.

PARTICIPATION ACTIVITY 4.16.6: Array-based list prepend and insert after operations.

Assume the following operations are executed on the list shown below:

ArrayListPrepend(list, 76)
ArrayListInsertAfter(list, 1, 38)
ArrayListInsertAfter(list, 3, 91)



1) Which operation causes ArrayListResize to be called?
   - ArrayListPrepend(list, 76)
   - ArrayListInsertAfter(list, 1, 38)
   - ArrayListInsertAfter(list, 3, 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

Search and removal operations

Given a key, the ***search*** operation returns the index for the first element whose data matches that key, 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 by 1 position.

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

PARTICIPATION ACTIVITY 4.16.7: Array-based list search and remove-at operations.

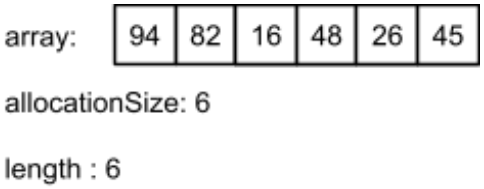
Animation content:

undefined

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 effectively removes the last 51.
4. The search for 84 now returns 1.

PARTICIPATION ACTIVITY 4.16.8: Search and remove-at operations.



1) What is the return value from ArrayListSearch(list, 33)?

Check Show answer

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

Check Show answer

3) ArrayListRemoveAt(list, 3) causes how many items to be moved down by 1 index?

Check Show answer

4) ArrayListRemoveAt(list, 5) causes how many items to be moved down by 1 index?

Check Show answer

PARTICIPATION ACTIVITY 4.16.9: Search and remove-at operations.

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

2) Searching for a key 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 4.16.1: Array-based lists.

502696.2096894.qx3zqy7

Start

numList: | 16 | 84 | |

allocationSize: 3
length: 2

Determine the length and allocation size of numList after each operation. If an item is added w the allocation size equals the array length, a new array with twice the current length is allocate

Operation	Length	Allocation size
ArrayListAppend(numList, 17)	Ex: 1	Ex:1

ArrayListAppend(numList, 75)		
ArrayListAppend(numList, 90)		
ArrayListAppend(numList, 13)		
ArrayListAppend(numList, 48)		

1 2 3 4

Check Next

4.17 Python: Array-based list

Python: Array-based list

Python's built-in List type is implemented using an array-based data structure. To examine how an array-based lists works, this section presents an ArrayList class implementing the array-based list data structure. The presented implementation is similar to Python's implementation of the List type.

The ArrayList class uses a List object as the internal array, with the entire list allocated and populated with None values to start.

Figure 4.17.1: The ArrayList class using a List as the internal array.

```
class ArrayList:
    def __init__(self, initial_allocation_size =
10):
        self.allocation_size =
initial_allocation_size
        self.length = 0
        self.array = [None] *
initial_allocation_size
```


append() and resize() methods

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 array data member is then reassigned with the new array, and allocation_size is updated.

Figure 4.17.2: append() and resize() methods.


```
def append(self, new_item):
    # resize() if the array is full
    if self.allocation_size == self.length:
        self.resize(self.length * 2)

    # insert the new item at index equal to
self.length
    self.array[self.length] = new_item

    # increment the array's length
    self.length = self.length + 1

def resize(self, new_allocation_size):
    # create a new array with the indicated size
    new_array = [None] * new_allocation_size

    # copy items in current array into the new array
    for i in range(self.length):
        new_array[i] = self.array[i]

    # assign the array data member with the new
array
    self.array = new_array

    # update the allocation size
    self.allocation_size = new_allocation_size
```

PARTICIPATION ACTIVITY 4.17.1: Array-based list implementation.

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

```
python_list = [6, 2, 8, 4, 3]
my_list = ArrayList(3)
for item in python_list:
    my_list.append(item)
```


1) The data member self.length is used as the index of the next available space in the internal array.

   - ○ True
   - ○ False

2) The method resize() is called twice.

- True
- False

3) The output for `print(my_list.array)` is:
`[6, 2, 8, 4, 3, None]`

- True
- False

prepend() and insert_after() methods

The prepend() method 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 the location at 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. The range() function is used to generate the indices 1, 2, 3, etc. up to self.length, and then the reversed() function is used to iterate backward through those indices. For a list of length 5, `range(1, self.array.length + 1)` produces the list 1, 2, 3, 4, 5, so `reversed(range(1, self.array.length + 1))` produces the indices 5, 4, 3, 2, 1, as required. Since a new item is added to the list, the array is first checked to see if space is available, and resized if not.

The insert_after() method operates almost identically 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. The combination of range() and reversed() is used again to iterate through the indices in the proper order.

Figure 4.17.3: prepend() and insert_after() methods.

```
def prepend(self, new_item):
    # resize() if the array is full
    if self.allocation_size == self.length:
        self.resize(self.length * 2)

    # Shift all array items to the right,
    # starting from the last index and moving
    # down to the first index.
    for i in reversed(range(1, self.length+1)):
        self.array[i] = self.array[i-1]

    # Insert the new item at index 0
    self.array[0] = new_item

    # Update the array's length
    self.length = self.length + 1

def insert_after(self, index, new_item):
    # resize() if the array is full
    if self.allocation_size == self.length:
        self.resize(self.length * 2)

    # Shift all the array items to the right,
    # starting from the last index and moving down
to
    # the index just past the given index.
    for i in reversed(range(index+1,
self.length+1)):
        self.array[i] = self.array[i-1]

    # Insert the new item at the index just past
the
    # given index.
    self.array[index+1] = new_item

    # Update the array's length
    self.length = self.length + 1
```


PARTICIPATION ACTIVITY 4.17.2: prepend() and insert_after() methods.

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

```
python_list = [6, 2, 8, 4, 3]
my_list = ArrayList(3)
for item in python_list:
    my_list.prepend(item)
```

1) What is the order of the integers generated by:
`reverse(range(4, 9))`

- 9, 8, 7, 6, 5, 4
- 9, 8, 7, 6, 5
- 8, 7, 6, 5, 4

2) Which shows the correct output of `print(my_list.array)`?

- [3, 4, 8, 2, 6, None]
- [None, 3, 4, 8, 2, 6]
- [2, 3, 4, 6, 8, None]

3) What is the output of the following?

```
my_list.insert_after(2, 3)
print(my_list.array)
```

- `[3, 4, 8, 2, 6, 2]`
- `[3, 4, 8, 2, 3, 6]`
- `[3, 4, 8, 3, 2, 6]`

search() and remove_at() methods

The search() method takes a target item as a parameter, and then tests each item in the list, from index zero up to index length-1, for equality with the target item. The method returns the index where the first matching item is found, or -1 if no matching item exists in the array. The search() method does not change the list's length or the internal array's allocation size.

remove_at() 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 remove_at() method 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 4.17.4: search() and remove_at() methods.

```
def search(self, item):
    # Iterate through the entire array
    for i in range(self.length):
        # If the current item matches the search
        # item, return the current index immediately.
        if self.array[i] == item:
            return i

    # If the above loop finishes without returning,
    # it means the search item was not found.
    return -1

def remove_at(self, index):
    # Make sure the index is valid for the current
array
    if index >= 0 and index < self.length:
        # Shift items from the given index up to the
        # end of the array.
        for i in range(index, self.length-1):
            self.array[i] = self.array[i+1]

        # Update the array's length
        self.length = self.length - 1
```


PARTICIPATION ACTIVITY 4.17.3: search() and remove_at() methods.

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

```
python_list = [6, 2, 8, 4, 3]
my_list = ArrayList(3)
for item in python_list:
    my_list.append(item)
```

1) The method call `my_list.remove_at(5)` has no effect.

   - True
   - False

2) The method call `my_list.search(1)` returns False.

   - True
   - False

3) The method call `my_list.remove_at(4)` results in the list: 6, 2, 8, 3.

- True
- False

zyDE 4.17.1: Working with the ArrayList class.

The following program allows the user to enter a method name along with argument value test the ArrayList implementation.

1. Test each method (append(), prepend(), insert_at(), search() and remove_at()) with different arguments and see if you can predict the final output.
2. Add a support for a command that looks like: `remove 91`. This command will find a remove the first value of 91 found in the list. If the value is not found in the list, no ac or output is required. *Do not modify the ArrayList class at all!* Adding the "remove" command can be done using existing ArrayList methods.

Current file: main.py Load default template

```
1
2 from ArrayList import ArrayList
3 my_list = ArrayList(4)
4 for item in [ 3, 2, 84, 18, 91, 6, 19, 12 ]:
5     my_list.append(item)
6
7 print("-- Array before operation --")
8 print("Allocation size: %d, length: %d" % (my_list.allocation_size, m
9 print(my_list.array)
10
11 instruction = input().split()
12 method = instruction[0]
13 if method == "append":
14     item = int(instruction[1])
15     my_list.append(item)
```

insert_after 5 3

Run

4.18 LAB: Sorted number list implementation with linked lists

This section's content is not available for print.

©zyBooks 11/20/23 11:00 1048447
Eric Quezada
UTEPCS2302ValeraFall2023