

# 4.1 Queue abstract data type (ADT)

## Queue abstract data type

©zyBooks 02/06/20 14:41 652770

JEFFREY WAN

A **queue** is an ADT in which items are inserted at the end of the queue and removed from the front of the queue. The queue **push** operation inserts an item at the end of the queue. The queue **pop** operation removes and returns the item at the front of the queue. Ex: After the operations "Push 7", "Push 14", and "Push 9", "Pop" returns 7. A second "Pop" returns 14. A queue is referred to as a **first-in first-out** ADT. A queue can be implemented using a linked list, an array, or a vector.

A queue ADT is similar to waiting in line at the grocery store. A person enters at the end of the line and exits at the front. British English actually uses the word "queue" in everyday vernacular where American English uses the word "line".

PARTICIPATION  
ACTIVITY

4.1.1: Queue ADT.



### Animation captions:

1. A new queue named "wQueue" is created. Items are pushed to the end of the queue.
2. Items are popped from the front of the queue.

PARTICIPATION  
ACTIVITY

4.1.2: Queue ADT.



- 1) Given numQueue: 5, 9, 1 (front is 5)  
What are the queue contents after the following push operation? Type the queue as: 1, 2, 3

Push(numQueue, 4)

Check

Show answer

©zyBooks 02/06/20 14:41 652770

JEFFREY WAN

JHUEN605202Spring2020

- 2) Given numQueue: 11, 22 (the front is 11)  
What are the queue contents after the following push operations? Type the queue as: 1, 2, 3



Push(numQueue, 28)  
Push(numQueue, 72)

**Check****Show answer**

©zyBooks 02/06/20 14:41 652770

JEFFREY WAN

JHUEN605202Spring2020



- 3) Given numQueue: 49, 3, 8

What is returned by the following  
pop operation?

Pop(numQueue)

**Check****Show answer**

- 4) Given numQueue: 4, 8, 7, 1, 3

What is returned by the second pop  
operation?

Pop(numQueue)

Pop(numQueue)

**Check****Show answer**

- 5) Given numQueue: 15, 91, 11

What is the queue after the following  
pop operation? Type the queue as: 1,  
2, 3

Pop(numQueue)

**Check****Show answer**

©zyBooks 02/06/20 14:41 652770

JEFFREY WAN

JHUEN605202Spring2020



- 6) Given numQueue: 87, 21, 43

What are the queue's contents after  
the following operations? Type the  
queue as: 1, 2, 3

Pop(numQueue)

Push(numQueue, 6)  
 Push(numQueue, 50)  
 Pop(numQueue)

©zyBooks 02/06/20 14:41 652770

JEFFREY WAN

JHUEN605202Spring2020

## Common queue ADT operations

Table 4.1.1: Some common operations for a queue ADT.

Operation	Description	Example starting with queue: 43, 12, 77 (front is 43)
Push(queue, x)	Inserts x at end of the queue	Push(queue, 56). Queue: 43, 12, 77, 56
Pop(queue)	Returns and removes item at front of queue	Pop(queue) returns: 43. Queue: 12, 77
Peek(queue)	Returns but does not remove item at the front of the queue	Peek(queue) return 43. Queue: 43, 12, 77
IsEmpty(queue)	Returns true if queue has no items	IsEmpty(queue) returns false.
GetLength(queue)	Returns the number of items in the queue	GetLength(queue) returns 3.

Note: Pop and Peek operations should not be applied to an empty queue; the resulting behavior may be undefined.

**PARTICIPATION ACTIVITY**

4.1.3: Common queue ADT operations.

©zyBooks 02/06/20 14:41 652770  
 JEFFREY WAN  
 JHUEN605202Spring2020

- 1) Given rosterQueue: 400, 313, 270, 514, 119, what does GetLength(rosterQueue) return?

- 400  
 5



2) Which operation determines if the queue contains no items?

- IsEmpty
- Peek

3) Given parkingQueue: 1, 8, 3, what are the queue contents after Peek(parkingQueue)?

- 1, 8, 3
- 8, 3

©zyBooks 02/06/20 14:41 652770  
JEFFREY WAN  
JHUEN605202Spring2020



4) Given parkingQueue: 2, 9, 4, what are the contents of the queue after Pop(parkingQueue)?

- 9, 4
- 2, 9, 4



5) Given that parkingQueue has no items (i.e., is empty), what does GetLength(parkingQueue) return?

- 1
- 0
- Undefined



## 4.2 Queues using linked lists

A queue is often implemented using a linked list, with the list's head node representing the queue's front, and the list's tail node representing the queue's end. Pushing an item to the queue is performed by appending the item to the list. Popping is performed by pointing a local variable to the list's head node, removing the head node from the list, and returning the local variable.

©zyBooks 02/06/20 14:41 652770  
JEFFREY WAN  
JHUEN605202Spring2020

PARTICIPATION  
ACTIVITY

4.2.1: Queue implemented using a linked list.



### Animation captions:

1. Pushing an item to the queue appends the item to the list, which inserts the item after the list's tail node.
2. A pop points a local variable to the list's head node, removes the list's head node, and returns the local variable.

**PARTICIPATION ACTIVITY**
**4.2.2: Queue push and pop operations with a linked list.**

©zyBooks 02/06/20 14:41 652770

JEFFREY WAN

JHUEN605202Spring2020

Assume the queue is implemented using a linked list.

- 1) If the head pointer is null, the queue

\_\_\_\_\_.

- is empty
- is full
- has at least one item

- 2) For the operation QueuePop(queue), what is the second parameter passed to ListRemoveAfter?

- The list's head node
- null
- poppedItem

- 3) For the operation QueuePop(queue), poppedItem contains a pointer to the \_\_\_\_\_.

- list head
- list tail

- 4) For QueuePush(charQueue, item 3), which pointer is updated to point to the newItem?



- List's head pointer

©zyBooks 02/06/20 14:41 652770

JEFFREY WAN

JHUEN605202Spring2020

- Node S' next pointer
- Node R's next pointer

## 4.3 List data structure

©zyBooks 02/06/20 14:41 652770  
JEFFREY WAN  
JHUEN605202Spring2020

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

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

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

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

PARTICIPATION  
ACTIVITY

4.3.1: Linked lists can be stored with or without a list data structure.



### Animation content:

undefined

### Animation captions:

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

PARTICIPATION  
ACTIVITY

4.3.2: Linked list data structure.

©zyBooks 02/06/20 14:41 652770  
JEFFREY WAN  
JHUEN605202Spring2020



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

- True
- False



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

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

- True
- False

©zyBooks 02/06/20 14:41 65270  
JEFFREY WAN  
JHUEN605202Spring2020

## 4.4 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

4.4.1: 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.

©zyBooks 02/06/20 14:41 65270  
JEFFREY WAN  
JHUEN605202Spring2020

PARTICIPATION ACTIVITY

4.4.2: 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**

©zyBooks 02/06/20 14:41 652770  
JEFFREY WAN  
JHUEN605202Spring2020

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 4.4.1: Some common operations for a list ADT.

©zyBooks 02/06/20 14:41 652770  
JEFFREY WAN

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 ©zyBooks 02/06/20 14:41 652770 JHUEN605202Spring2020
Sort(list)	Sorts the list's 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**
**4.4.3: 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

©zyBooks 02/06/20 14:41 652770  
JEFFREY WAN  
JHUEN605202Spring2020

# 4.5 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 use 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

4.5.1: 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

4.5.2: Array-based lists.



- 1) The length of an array-based list equals the list's array allocation size.
  - True
  - False
- 2) An item can be appended to an array-based list, provided the length is less than the array's allocated size.
  - True
  - False
- 3) An array-based list can have a

©zyBooks 02/06/20 14:41 652770  
JEFFREY WAN  
JHUEN605202Spring2020

default allocation size of 0.

- True
- False

## Resize operation

©zyBooks 02/06/20 14:41 652770

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  $O(N)$ .

PARTICIPATION  
ACTIVITY

4.5.3: Array-based list resize operation.



### 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

4.5.4: 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)

array: 

81	23	68	39	
----	----	----	----	--

  
 allocationSize: 5  
 length : 4

©zyBooks 02/06/20 14:41 652770  
 JEFFREY WAN  
 JHUEN605202Spring2020

- 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

©zyBooks 02/06/20 14:41 652770  
JEFFREY WAN  
JHUEN605202Spring2020

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 the 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  $O(N)$ .

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  $O(1)$  and a worst case runtime complexity of  $O(N)$ .

### 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 `ArrayListInsertAfter` to insert after index  $X - 1$ .

©zyBooks 02/06/20 14:41 652770  
JEFFREY WAN  
JHUEN605202Spring2020

PARTICIPATION  
ACTIVITY

4.5.5: Array-based list prepend and insert after operations.



## Animation captions:

1. To prepend 91, every array element is first moved up 1 index.
2. Item 91 is assigned at index 0.
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.

©zyBooks 02/06/20 14:41 652770

JEFFREY WAN

JHUEN605202Spring2020

PARTICIPATION  
ACTIVITY

4.5.6: 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)

array: 

22	16		
----	----	--	--

  
 allocationSize: 4  
 length : 2

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

©zyBooks 02/06/20 14:41 652770

JEFFREY WAN

JHUEN605202Spring2020

## 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  $O(N)$ .

©zyBooks 02/06/20 14:41 652770

JEFFREY WAN

JHUEN605202Spring2020

PARTICIPATION  
ACTIVITY

4.5.7: Array-based list search and remove-at operations.



### 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

4.5.8: Search and remove-at operations.



array: 

94	82	16	48	26	45
----	----	----	----	----	----

allocationSize: 6

length : 6

- 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**



©zyBooks 02/06/20 14:41 652770  
JEFFREY WAN  
JHUEN605202Spring2020

- 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**

©zyBooks 02/06/20 14:41 652770  
JEFFREY WAN  
JHUEN605202Spring2020

**PARTICIPATION ACTIVITY****4.5.9: 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

©zyBooks 02/06/20 14:41 652770  
JEFFREY WAN  
JHUEN605202Spring2020

## 4.6 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

©zyBooks 02/06/20 14:41 652770  
JEFFREY WAN  
JHUEN605202Spring2020

**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

4.6.1: Singly-linked list: Each node points to the next node.



### 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.

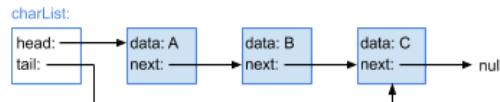
PARTICIPATION  
ACTIVITY

4.6.2: Singly-linked list data structure.



- 1) Given `charList`, C's next pointer value  
is \_\_\_\_.

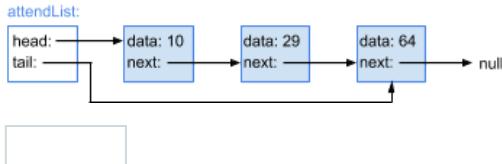
©zyBooks 02/06/20 14:41 652770  
JEFFREY WAN  
JHUEN605202Spring2020



**Check**

**Show answer**

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

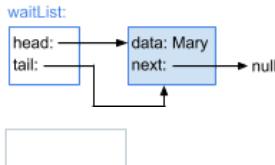


©zyBooks 02/06/20 14:41 652770  
JEFFREY WAN  
JHUEN605202Spring2020

**Check**

**Show answer**

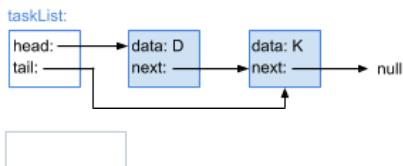
- 3) Given `waitList`, the tail node's data value is \_\_\_\_.  
(Answer "None" if no tail exists)



**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 newly created node or an existing node in a list 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**
**4.6.3: Singly-linked list: Appending a node.**


©zyBooks 02/06/20 14:41 652770

JEFFREY WAN

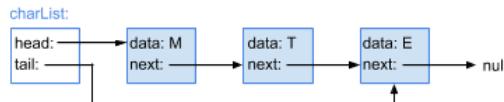
JHUEN605202Spring2020

**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**
**4.6.4: Appending a node to a singly-linked list.**


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



- M  
 T  
 E

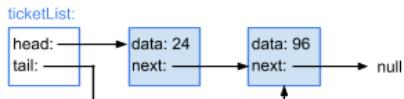
- 2) Appending node W to sampleList  
updates which of sampleList's  
pointers?



- head and tail  
 head only  
 tail only

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

©zyBooks 02/06/20 14:41 652770  
JEFFREY WAN  
JHUEN605202Spring2020



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

©zyBooks 02/06/20 14:41 652770  
JEFFREY WAN  
JHUEN605202Spring2020

## 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

4.6.5: Singly-linked list: Prepending a node.



### Animation captions:

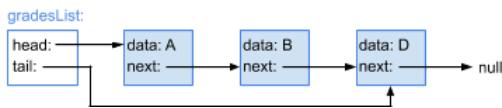
1. Prepending an item to an empty list points the list's head and tail pointers to new node.
2. Prepending to a non-empty list points the new node's next pointer to the list's head node.
3. Prepending then points the list's head pointer to the new node.

PARTICIPATION  
ACTIVITY

4.6.6: 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

©zyBooks 02/06/20 14:41 652770  
JEFFREY WAN  
JHUEN605202Spring2020

- 2) Prepending node 789 to



studentIdList updates the list's tail pointer.



- True
- False

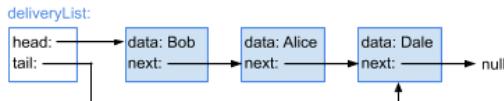
3) Prepending node 6 to parkingList updates the list's tail pointer.

©zyBooks 02/06/20 14:41 652770  
JEFFREY WAN  
JHUEN605202Spring2020



- True
- False

4) Prepending Evelyn to deliveryList executes which statement?



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

## 4.7 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. 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

4.7.1: Singly-linked list: Insert nodes.



©zyBooks 02/06/20 14:41 652770  
JEFFREY WAN  
JHUEN605202Spring2020

### 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

4.7.2: 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](#)

©zyBooks 02/06/20 14:41 652770  
JEFFREY WAN  
JHUEN605202Spring2020

3) numsList: 1



ListInsertAfter(numsList, node 1,  
node 6)

ListInsertAfter(numsList, node 1,  
node 4)

numsList:

**Check**

**Show answer**

- 4) numsList: 77

©zyBooks 02/06/20 14:41 652770  
JEFFREY WAN  
JHUEN605202Spring2020



ListInsertAfter(numsList, node 77,  
node 32)

ListInsertAfter(numsList, node 32,  
node 50)

ListInsertAfter(numsList, node 32,  
node 46)

numsList:

**Check**

**Show answer**

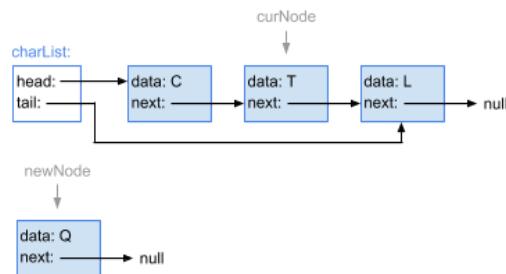
**PARTICIPATION**  
**ACTIVITY**

4.7.3: Singly-linked list insert-after algorithm.



- 1) ListInsertAfter(charList, node T, node Q) points newNode's next pointer to

\_\_\_\_\_.

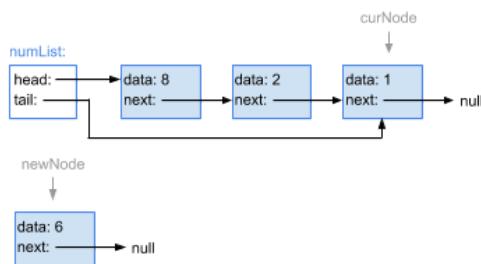


- curNode → next
- list → head
- null

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

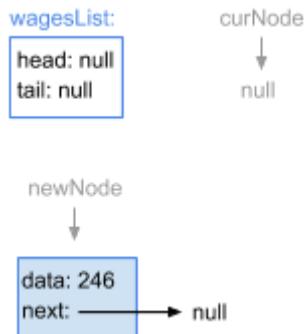
©zyBooks 02/06/20 14:41 652770  
JEFFREY WAN  
JHUEN605202Spring2020





- `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`

## 4.8 Singly-linked lists: Remove

©zyBooks 02/06/20 14:41 652770  
JEFFREY WAN  
JHUEN605202Spring2020

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.

curNode points to an existing list node. Because the list's head node is not after another node, if curNode is null, RemoveAfter implements a special case that removes the list's head 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**

4.8.1: Singly-linked list: Node removal.



**Animation captions:**

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

**PARTICIPATION  
ACTIVITY**

4.8.2: 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**

©zyBooks 02/06/20 14:41 652770  
JEFFREY WAN  
JHUEN605202Spring2020

2) numsList: 3, 57, 28, 40



ListRemoveAfter(numsList, null)

numsList:

**Check**

**Show answer**

**Check****Show answer**

3) numsList: 9, 4, 11, 7



ListRemoveAfter(numsList, node 11)

numsList:

**Check****Show answer**©zyBooks 02/06/20 14:41 652770  
JEFFREY WAN  
JHUEN605202Spring2020

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)

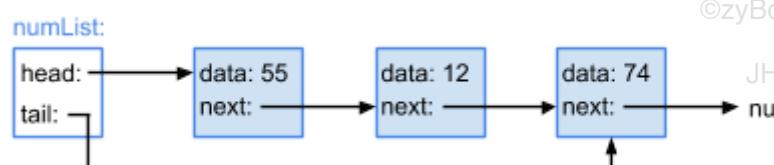
numsList:

**Check****Show answer****PARTICIPATION  
ACTIVITY**

4.8.3: ListRemoveAfter algorithm execution: Intermediate node.



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

©zyBooks 02/06/20 14:41 652770  
JEFFREY WAN  
JHUEN605202Spring2020

1) sucNode = list → head → next

 Yes

No2)  $\text{curNode} \rightarrow \text{next} = \text{sucNode}$  Yes No3)  $\text{list} \rightarrow \text{head} = \text{sucNode}$ 

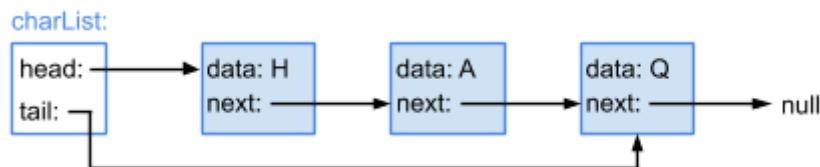
©zyBooks 02/06/20 14:41 65270

JEFFREY WAN

JHUEN605202Spring2020

 Yes No4)  $\text{list} \rightarrow \text{tail} = \text{curNode}$  Yes No**PARTICIPATION ACTIVITY****4.8.4: ListRemoveAfter algorithm execution: List head node.**

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

1)  $\text{sucNode} = \text{list} \rightarrow \text{head} \rightarrow \text{next}$  Yes No2)  $\text{curNode} \rightarrow \text{next} = \text{sucNode}$  Yes No3)  $\text{list} \rightarrow \text{head} = \text{sucNode}$ ©zyBooks 02/06/20 14:41 652770  
JEFFREY WAN  
JHUEN605202Spring2020 Yes No4)  $\text{list} \rightarrow \text{tail} = \text{curNode}$  Yes

No

## 4.9 Linked list traversal

©zyBooks 02/06/20 14:41 652770  
JEFFREY WAN  
JHUEN605202Spring2020

### 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 4.9.1: Linked list traversal algorithm.

```
ListTraverse(list) {  
    curNode = list->head // Start at head  
  
    while (curNode is not null) {  
        Print curNode's data  
        curNode = curNode->next  
    }  
}
```

PARTICIPATION  
ACTIVITY

4.9.1: Singly-linked list: List traversal.



### 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.

©zyBooks 02/06/20 14:41 652770  
JEFFREY WAN  
JHUEN605202Spring2020

PARTICIPATION  
ACTIVITY

4.9.2: List traversal.





1) ListTraverse begins with \_\_\_\_.

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

2) Given numList is: 5, 8, 2, 1.

ListTraverse(numsList) visits \_\_\_\_ node(s).

©zyBooks 02/06/20 14:41 652770  
JEFFREY WAN  
JHUEN605202Spring2020

- 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 4.9.2: Reverse traversal algorithm.

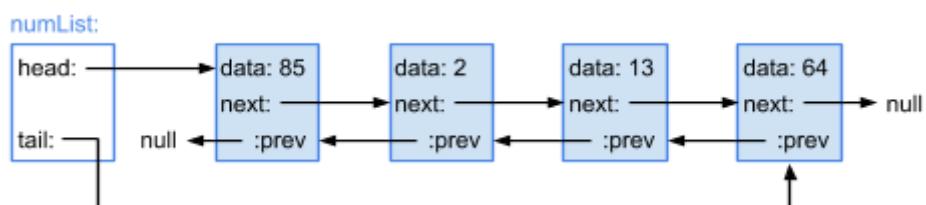
```
ListTraverseReverse(list) {
    curNode = list->tail // Start at tail

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

©zyBooks 02/06/20 14:41 652770  
JEFFREY WAN  
JHUEN605202Spring2020

PARTICIPATION  
ACTIVITY

4.9.3: Reverse traversal algorithm execution.



- 1) ListTraverseReverse visits which node second?

- Node 2
- Node 13

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

- True
- False

©zyBooks 02/06/20 14:41 652770  
JEFFREY WAN  
JHUEN605202Spring2020

## 4.10 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 current node is null, the algorithm returns null (matching node was not found).

PARTICIPATION  
ACTIVITY

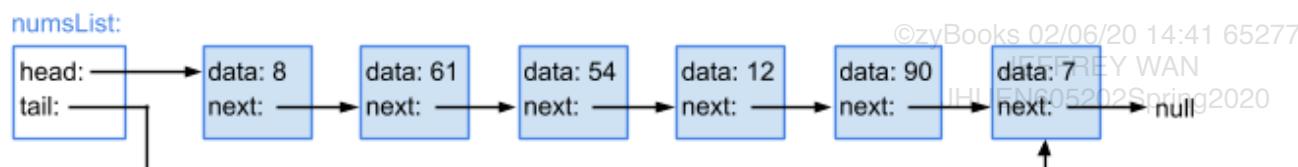
4.10.1: Singly-linked list: Searching.

### 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

4.10.2: ListSearch algorithm execution.



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

**Check**

**show answer**

**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**

©zyBooks 02/06/20 14:41 652770  
JEFFREY WAN  
JHUEN605202Spring2020

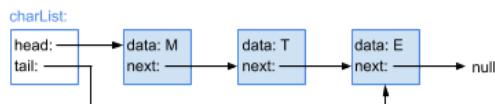


**PARTICIPATION ACTIVITY**

4.10.3: Searching a linked-list.

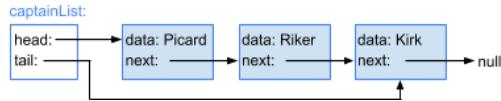


- 1) ListSearch(charList, E) first assigns curNode to \_\_\_\_.



- 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

©zyBooks 02/06/20 14:41 652770  
JEFFREY WAN  
JHUEN605202Spring2020



## 4.11 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.

©zyBooks 02/06/20 14:41 652770  
JEFFREY WAN  
JHUEN605202Spring2020

PARTICIPATION  
ACTIVITY

4.11.1: Recursive forward traversal.



### 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. Node 19 is visited and an additional recursive call visits node 41. The last recursive call encounters a null node and stops.

PARTICIPATION  
ACTIVITY

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



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

- 9
- 10
- 11

PARTICIPATION  
ACTIVITY

4.11.3: Forward traversal concepts.



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

©zyBooks 02/06/20 14:41 652770  
JEFFREY WAN  
JHUEN605202Spring2020

empty list.

- True
- False

## Searching

©zyBooks 02/06/20 14:41 652770

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 4.11.1: ListSearch and ListSearchRecursive functions.

```

ListSearch(list, key) {
    return ListSearchRecursive(key, list->head)
}

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

```

**PARTICIPATION ACTIVITY**

4.11.4: 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

©zyBooks 02/06/20 14:41 652770  
JEFFREY WAN  
JHUEN605202Spring2020

ListSearchRecursive.

- 1
- 2
- 10
- 11

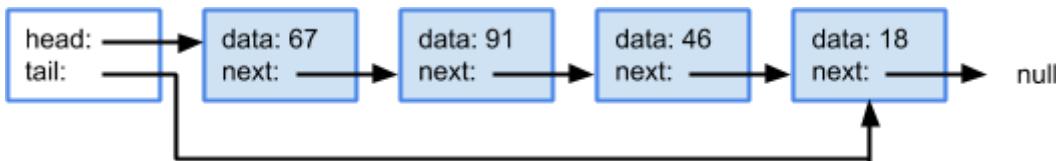
- 3) When the key is not found,  
ListSearch returns \_\_\_\_.

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

©zyBooks 02/06/20 14:41 652770  
JEFFREY WAN  
JHUEN605202Spring2020



&lt;p



- 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

©zyBooks 02/06/20 14:41 652770  
JEFFREY WAN  
JHUEN605202Spring2020

## 4.12 Deque abstract data type (ADT)

### Deque abstract data type

©zyBooks 02/06/20 14:41 652770  
JEFFREY WAN  
JHUEN605202Spring2020

A **deque** (pronounced "deck" and short for double-ended queue) is an ADT in which items can be inserted and removed at both the front and back. The deque push-front operation inserts an item at the front of the deque, and the push-back operation inserts at the back of the deque. The pop-front operation removes and returns the item at the front of the deque, and the pop-back operation removes and returns the item at the back of the deque. Ex: After the operations "push-

back 7", "push-front 14", "push-front 9", and "push-back 5", "pop-back" returns 5. A subsequent "pop-front" returns 9. A deque can be implemented using a linked list or an array.

**PARTICIPATION ACTIVITY**
**4.12.1: Deque ADT.**

**Animation captions:**

©zyBooks 02/06/20 14:41 652770

JEFFREY WAN

JHUEN605202Spring2020

1. The "push-front 34" operation followed by "push-front 51" produces a deque with contents 51, 34.
2. The "push-back 19" operation pushes 19 to the back of the deque, yielding 51, 34, 19. "Pop-front" then removes and returns 51.
3. Items can also be removed from the back of the deque. The "pop-back" operation removes and returns 19.

**PARTICIPATION ACTIVITY**
**4.12.2: Deque ADT.**


Determine the deque contents after the following operations.

**push-front 97,  
push-back 71,  
pop-front,  
push-front 45,  
push-back 68**

**push-back 45,  
push-back 71,  
push-front 97,  
push-front 68,  
pop-back**

**push-front 71,  
push-front 68,  
push-front 97,  
pop-back,  
push-front 45**

45, 97, 68

45, 71, 68

68, 97, 45

**Reset**

©zyBooks 02/06/20 14:41 652770

JEFFREY WAN

JHUEN605202Spring2020

**Common deque ADT operations**

In addition to pushing or popping at the front or back, a deque typically supports peeking at the front and back of the deck and determining the length. A **peek** operation returns an item in the deque without removing the item.

Table 4.12.1: Common deque ADT operations.

Operation	Description	Example starting with deque: 59, 63, 19 (front is 59)
PushFront(deque, x)	Inserts x at the front of the deque	PushFront(deque, 41). Deque: 41, 59, 63, 19
PushBack(deque, x)	Inserts x at the back of the deque	©zyBooks 02/06/20 14:41 652770 PushBack(deque, 41). Deque: 59, 63, 19, 41 JHUEN605202Spring2020
PopFront(deque)	Returns and removes item at front of deque	PopFront(deque) returns 59. Deque: 63, 19
PopBack(deque)	Returns and removes item at back of deque	PopBack(deque) returns 19. Deque: 59, 63
PeekFront(deque)	Returns but does not remove the item at the front of deque	PeekFront(deque) returns 59. Deque is still: 59, 63, 19
PeekBack(deque)	Returns but does not remove the item at the back of deque	PeekBack(deque) returns 19. Deque is still: 59, 63, 19
IsEmpty(deque)	Returns true if the deque is empty	IsEmpty(deque) returns false.
GetLength(deque)	Returns the number of items in the deque	GetLength(deque) returns 3.

**PARTICIPATION ACTIVITY**

4.12.3: Common queue ADT operations.

- 1) Given rosterDeque: 351, 814, 216, 636, 484, 102, what does GetLength(rosterDeque) return?

- 351
- 102
- 6

- 2) Which operation determines if the deque contains no items?

- IsEmpty
- PeekFront

©zyBooks 02/06/20 14:41 652770  
JEFFREY WAN  
JHUEN605202Spring2020



- 3) Given jobsDeque: 4, 7, 5, what are the deque contents after PeekBack(jobsDeque)?
- 4, 7, 5
  - 4, 7

- 4) Given jobsDeque: 3, 6, 1, 7, what are the contents of the deque after PopFront(jobsDeque)?

©zyBooks 02/06/20 14:41 652770  
JEFFREY WAN  
JHUEN605202Spring2020

- 6, 1, 7
- 3, 6, 1, 7

- 5) Given that jobsDeque is empty, what does GetLength(jobsDeque) return?



- 1
- 0
- Undefined

©zyBooks 02/06/20 14:41 652770  
JEFFREY WAN  
JHUEN605202Spring2020