15.1 Abstract data types

Abstract data types (ADTs)

An ***abstract data type*** (***ADT***) is a data type described by predefined user operations, such as "insert data at rear," without indicating how each operation is implemented. An ADT can be implemented using different underlying data structures. However, a programmer need not have knowledge of the underlying implementation to use an ADT.

Ex: 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. A list ADT is commonly implemented using arrays or linked list data structures.

PARTICIPATION ACTIVITY 15.1.1: List ADT using array and linked lists data structures.

Animation captions:

1. A new list named agesList is created. Items can be appended to the list. The items are ordered.
2. Printing the list prints the items in order.
3. A list ADT is commonly implemented using array and linked list data structures. But, a programmer need not have knowledge of which data structure is used to use the list ADT.

PARTICIPATION ACTIVITY 15.1.2: Abstract data types.

1) Starting with an empty list, what is the list contents after the following operations?

   Append(list, 11)
   Append(list, 4)
   Append(list, 7)

   - 4, 7, 11
   - 7, 4, 11
   - 11, 4, 7

2) A remove operation for a list ADT will remove the specified item. Given a list with contents: 2, 20, 30, what is the list

contents after the following operation?

Remove(list, item 2)

- 2, 30
- 2, 20, 30
- 20, 30

3) A programmer must know the underlying implementation of the list ADT in order to use a list.
   - True
   - False

4) A list ADT's underlying data structure has no impact on the program's execution.
   - True
   - False

Common ADTs

Table 15.1.1: Common ADTs.

Abstract data type	Description	Common underlying data structures
List	A ***list*** is an ADT for holding ordered data.	Array, linked list
Dynamic array	A ***dynamic array*** is an ADT for holding ordered data and allowing indexed access.	Array
Stack	A ***stack*** is an ADT in which items are only inserted on or removed from the top of a stack.	Linked list
Queue	A ***queue*** is an ADT in which items are inserted at the end of the queue and removed from the front of the queue.	Linked list
Deque	A ***deque*** (pronounced "deck" and short for double-ended	Linked list

	queue) is an ADT in which items can be inserted and removed at both the front and back.	
Bag	A ***bag*** is an ADT for storing items in which the order does not matter and duplicate items are allowed.	Array, linked list
Set	A ***set*** is an ADT for a collection of distinct items.	Binary search tree, hash table
Priority queue	A ***priority queue*** is a queue where each item has a priority, and items with higher priority are closer to the front of the queue than items with lower priority.	Heap
Dictionary (Map)	A ***dictionary*** is an ADT that associates (or maps) keys with values.	Hash table, binary search tree

PARTICIPATION ACTIVITY 15.1.3: Common ADTs.

Consider the ADTs listed in the table above. Match the ADT with the description of the order and uniqueness of items in the ADT.

List | Bag | Set | Priority queue

	Items are ordered based on how items are added. Duplicate items are allowed.
	Items are not ordered. Duplicate items are not allowed.
	Items are ordered based on items' priority. Duplicate items are allowed.
	Items are not ordered. Duplicate items are allowed.

Reset

15.2 Applications of ADTs

Abstraction and optimization

Abstraction means to have a user interact with an item at a high-level, with lower-level internal details hidden from the user. ADTs support abstraction by hiding the underlying implementation details and providing a well-defined set of operations for using the ADT.

Using abstract data types enables programmers or algorithm designers to focus on higher-level operations and algorithms, thus improving programmer efficiency. However, knowledge of the underlying implementation is needed to analyze or improve the runtime efficiency.

PARTICIPATION ACTIVITY 15.2.1: Programming using ADTs.

Animation content:

`undefined`

Animation captions:

1. Abstraction simplifies programming. ADTs allow programmers to focus on choosing which ADTs best match a program's needs.
2. Both the List and Queue ADTs support efficient interfaces for removing items from one end (removing oldest entry) and adding items to the other end (adding new entries).
3. The list ADT supports iterating through list contents in reverse order, but the queue ADT does not.
4. To use the List (or Queue) ADT, the programmer does not need to know the List's underlying implementation.

PARTICIPATION ACTIVITY 15.2.2: Programming with ADTs.

Consider the example in the animation above.

1) The _____ ADT is the better match for the program's requirements.
   - ○ queue
   - ○ list

2) The list ADT _____.

- can only be implemented using an array
- can only be implemented using a linked list
- can be implemented in numerous ways

3) Knowledge of an ADT's underlying implementation is needed to analyze the runtime efficiency.
   - True
   - False

ADTs in standard libraries

Most programming languages provide standard libraries that implement common abstract data types. Some languages allow programmers to choose the underlying data structure used for the ADTs. Other programming languages may use a specific data structure to implement each ADT, or may automatically choose the underlying data-structure.

Table 15.2.1: Standard libraries in various programming languages.

Programming language	Library	Common supported ADTs
Python	Python standard library	list, set, dict, deque
C++	Standard template library (STL)	vector, list, deque, queue, stack, set, map
Java	Java collections framework (JCF)	Collection, Set, List, Map, Queue, Deque

PARTICIPATION ACTIVITY 15.2.3: ADTs in standard libraries.

1) Python, C++, and Java all provide built-in support for a deque ADT.
   - True
   -

False

2) The underlying data structure for a list data structure is the same for all programming languages.
   - True
   - False

3) ADTs are only supported in standard libraries.
   - True
   - False

15.3 Algorithm efficiency

Algorithm efficiency

An algorithm describes the method to solve a computational problem. Programmers and computer scientists should use or write efficient algorithms. ***Algorithm efficiency*** is typically measured by the algorithm's computational complexity. ***Computational complexity*** is the amount of resources used by the algorithm. The most common resources considered are the runtime and memory usage.

PARTICIPATION ACTIVITY 15.3.1: Computational complexity.

Animation captions:

1. An algorithm's computational complexity includes runtime and memory usage.
2. Measuring runtime and memory usage allows different algorithms to be compared.
3. Complexity analysis is used to identify and avoid using algorithms with long runtimes or high memory usage.

PARTICIPATION ACTIVITY 15.3.2: Algorithm efficiency and computational complexity.

1) Computational complexity analysis allows the efficiency of algorithms to be compared.
   - True

- False

2) Two different algorithms that produce the same result have the same computational complexity.
   - True
   - False

3) Runtime and memory usage are the only two resources making up computational complexity.
   - True
   - False

Runtime complexity, best case, and worst case

An algorithm's ***runtime complexity*** is a function, T(N), that represents the number of constant time operations performed by the algorithm on an input of size N. Runtime complexity is discussed in more detail elsewhere.

Because an algorithm's runtime may vary significantly based on the input data, a common approach is to identify best and worst case scenarios. An algorithm's ***best case*** is the scenario where the algorithm does the minimum possible number of operations. An algorithm's ***worst case*** is the scenario where the algorithm does the maximum possible number of operations.

Input data size must remain a variable

A best case or worst case scenario describes contents of the algorithm's input data only. The input data size must remain a variable, N. Otherwise, the overwhelming majority of algorithms would have a best case of N=0, since no input data would be processed. In both theory and practice, saying "the best case is when the algorithm doesn't process any data" is not useful. Complexity analysis always treats the input data size as a variable.

PARTICIPATION ACTIVITY 15.3.3: Linear search best and worst cases.

Animation captions:

1. LinearSearch searches through array elements until finding the key. Searching for 26 requires iterating through the first 3 elements.
2. The search for 26 is neither the best nor the worst case.
3. Searching for 54 only requires one comparison and is the best case: The key is found at the start of the array. No other search could perform fewer operations.
4. Searching for 82 compares against all array items and is the worst case: The number is not found in the array. No other search could perform more operations.

PARTICIPATION ACTIVITY 15.3.4: FindFirstLessThan algorithm best and worst case.

Consider the following function that returns the first value in a list that is less than the specified value. If no list items are less than the specified value, the specified value is returned.

```
FindFirstLessThan(list, listSize, value) {
   for (i = 0; i < listSize; i++) {
      if (list[i] < value)
         return list[i]
   }
   return value // no lesser value found
}
```

Worst case **Best case** **Neither best nor worst case**

	No items in the list are less than `value`.
	The first half of the list has elements greater than `value` and the second half has elements less than `value`.
	The first item in the list is less than `value`.

Reset

PARTICIPATION ACTIVITY 15.3.5: Best and worst case concepts.

1) The linear search algorithm's best case scenario is when N = 0.

- ○ True
- ○

False

2) An algorithm's best and worst case scenarios are always different.

- True
- False

Space complexity

An algorithm's ***space complexity*** is a function, S(N), that represents the number of fixed-size memory units used by the algorithm for an input of size N. Ex: The space complexity of an algorithm that duplicates a list of numbers is S(N) = 2N + k, where k is a constant representing memory used for things like the loop counter and list pointers.

Space complexity includes the input data and additional memory allocated by the algorithm. An algorithm's ***auxiliary space complexity*** is the space complexity not including the input data. Ex: An algorithm to find the maximum number in a list will have a space complexity of S(N) = N + k, but an auxiliary space complexity of S(N) = k, where k is a constant.

PARTICIPATION ACTIVITY 15.3.6: FindMax space complexity and auxiliary space complexity.

Animation content:

undefined

Animation captions:

1. FindMax's arguments represent input data. Non-input data includes variables allocated in the function body: maximum and i.
2. The list's size is a variable, N. Three integers are also used, listSize, maximum, and i, making the space complexity S(N) = N + 3.
3. The auxiliary space complexity includes only the non-input data, which does not increase for larger input lists.
4. The function's auxiliary space complexity is S(N) = 2.

PARTICIPATION ACTIVITY 15.3.7: Space complexity of GetEvens function.

Consider the following function, which builds and returns a list of even numbers from the input list.

```
GetEvens(list, listSize) {
   i = 0
   evensList = Create new, empty list
   while (i < listSize) {
      if (list[i] % 2 == 0)
         Add list[i] to evensList
      i = i + 1
   }
   return evensList
}
```


1) What is the maximum possible size of the returned list?
   - listSize
   - listSize / 2

2) What is the minimum possible size of the returned list?
   - listSize / 2
   - 1
   - 0

3) What is the worst case auxiliary space complexity of GetEvens if N is the list's size and k is a constant?
   - S(N) = N + k
   - S(N) = k

4) What is the best case auxiliary space complexity of GetEvens if N is the list's size and k is a constant?
   - S(N) = N + k
   - S(N) = k

15.4 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 15.4.1: 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.

PARTICIPATION ACTIVITY 15.4.2: 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

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 15.4.1: 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 15.4.3: 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

15.5 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 15.5.1: Singly-linked list: Each node points to the next node.

Animation content:

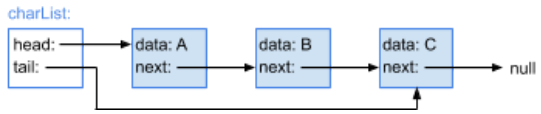
undefined

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 15.5.2: Singly-linked list data structure.

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

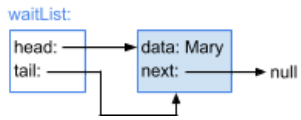


Check Show answer

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

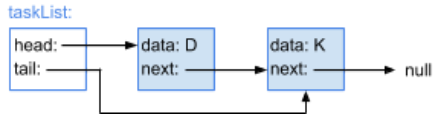
Check Show answer

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 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 15.5.3: Singly-linked list: Appending a node.

Animation content:

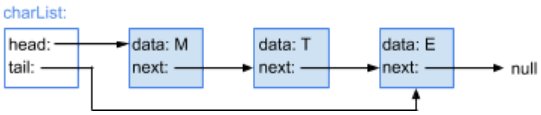
undefined

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 15.5.4: Appending a node to a singly-linked list.

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

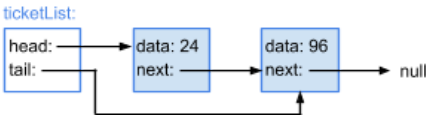


- M
- T
- E

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 15.5.5: Singly-linked list: Prepending a node.

Animation content:

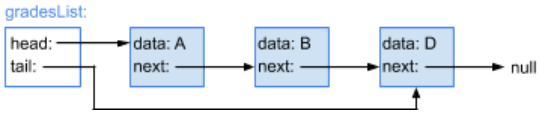
undefined

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.

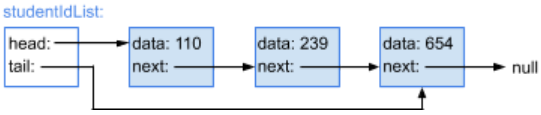
PARTICIPATION ACTIVITY 15.5.6: Prepending a node in a singly-linked list.

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 15.5.1: Singly-linked lists.

Start

numList = new List
ListAppend(numList, node 43)
ListAppend(numList, node 23)
ListAppend(numList, node 87)
ListAppend(numList, node 19)

numList is now: Ex: 1, 2, 3

1	2	3	4	5

Check Next

15.6 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 15.6.1: 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 15.6.2: 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

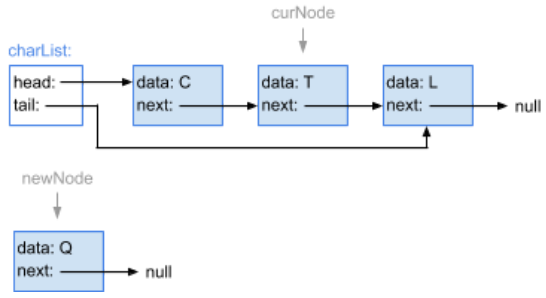
ListInsertAfter(numsList, node 77, node 32)
ListInsertAfter(numsList, node 32, node 50)
ListInsertAfter(numsList, node 32, node 46)

numsList:

Check Show answer

PARTICIPATION ACTIVITY 15.6.3: Singly-linked list insert-after algorithm.

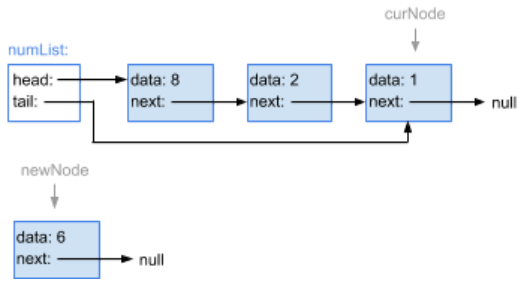
1) ListInsertAfter(charList, node T, node Q) assigns newNode's next pointer with _____.



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

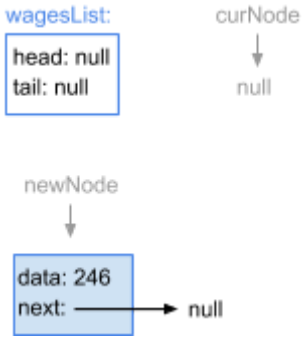
2) ListInsertAfter(numList, node 1, node

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 15.6.1: Singly-linked lists: Insert.

Start

numList: 17, 24
ListInsertAfter(numList, node 17, node 76)

ListInsertAfter(numList, node 76, node 71)
ListInsertAfter(numList, node 76, node 18)

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

1 2 3 4

Check Next

15.7 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 15.7.1: Singly-linked list: Node removal.

Animation content:

`undefined`

Animation captions:

1. If curNode is null, the list's head node is removed.
2. The list's head pointer is pointed to the list head's successor node.

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

PARTICIPATION ACTIVITY 15.7.2: 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

©zyBooks 04/25/21 07:49 488201
xiang zhao
BAYLORCSI14301440Spring2021

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

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

numsList:

Check Show answer

PARTICIPATION ACTIVITY 15.7.3: 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

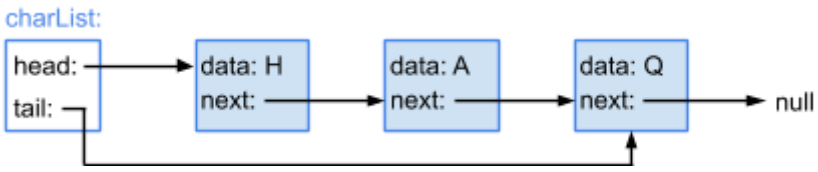
- Yes
- No

4) list⇢tail = curNode

- Yes
- No

PARTICIPATION ACTIVITY 15.7.4: 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

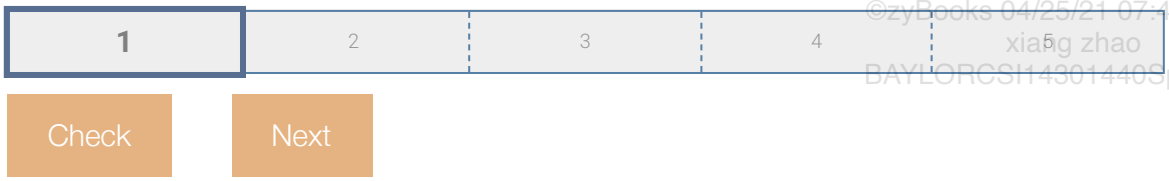
CHALLENGE ACTIVITY 15.7.1: Singly-linked lists: Remove.

Start

Given list: 7, 6, 3, 2, 5
What list results from the following operations?

ListRemoveAfter(list, node 6)
ListRemoveAfter(list, node 2)
ListRemoveAfter(list, null)

Ex: 25, 42, 12 m head to tail.



Check Next

15.8 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 15.8.1: Singly-linked list: Searching.

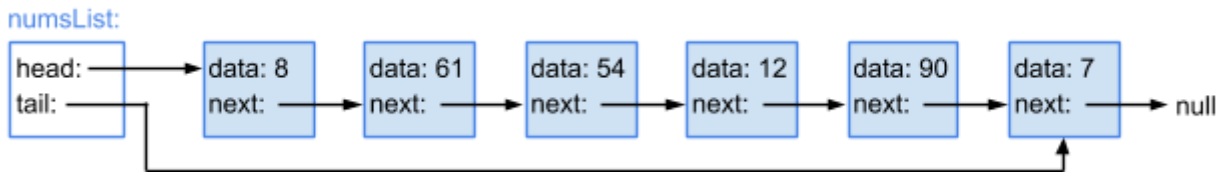
Animation content:

`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 15.8.2: 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?

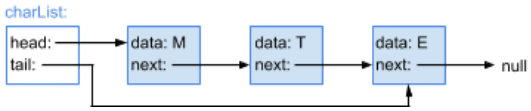
Check Show answer

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

Check Show answer

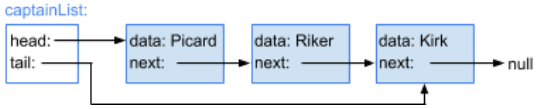
PARTICIPATION ACTIVITY 15.8.3: Searching a linked-list.

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?

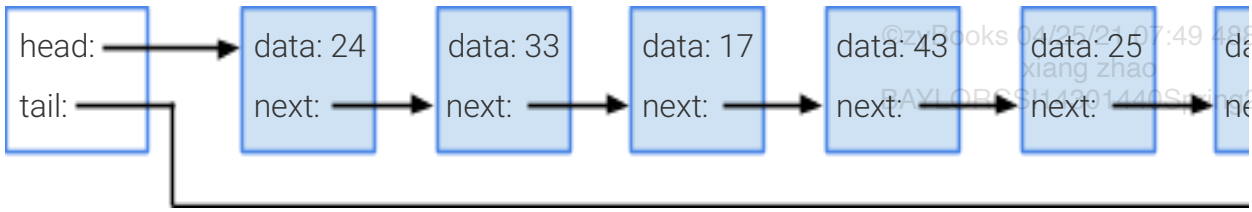


- node Riker
- node Kirk

CHALLENGE ACTIVITY 15.8.1: Linked list search.

Start

numList:



ListSearch(numList, 27) points the current pointer to node Ex: 9 after checking node 33.

ListSearch(numList, 27) will make comparisons.

1 2

Check Next

15.9 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 15.9.1: 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 15.9.2: Doubly-linked list: Appending a node.

Animation content:

undefined

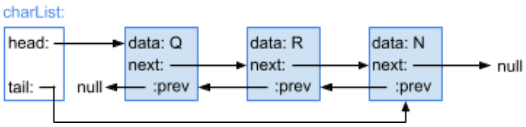
Animation captions:

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.

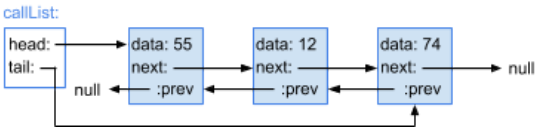
PARTICIPATION ACTIVITY 15.9.3: Doubly-linked list data structure.

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 15.9.4: 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 15.9.5: Prepending a node in a doubly-linked list.

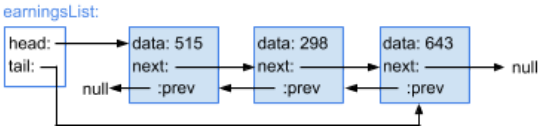
1) Prepending 29 to trainsList updates the list's head pointer to point to node _____.

- 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 15.9.1: Doubly-linked lists.

Start

numList = new List
ListAppend(numList, node 93)
ListAppend(numList, node 19)
ListAppend(numList, node 60)

numList is now: Ex: 1, 2, 3

Which node has a null previous pointer? Ex: 5

Which node has a null next pointer? Ex: 5

1 2 3 4 5

Check Next

15.10 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 15.10.1: 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 15.10.2: Inserting nodes in a doubly-linked list.

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

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

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

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

Reset

CHALLENGE ACTIVITY 15.10.1: Doubly-linked lists: Insert.

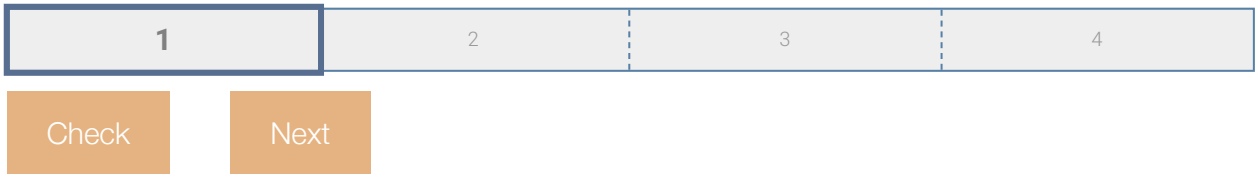
Start

numList: 23, 63
ListInsertAfter(numList, node 23, node 10)
ListInsertAfter(numList, node 63, node 77)
ListInsertAfter(numList, node 10, node 51)
ListInsertAfter(numList, node 10, node 65)

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

What node does node 63's next pointer point to?

What node does node 63's previous pointer point to?



Check Next

15.11 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 15.11.1: 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 15.11.2: 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

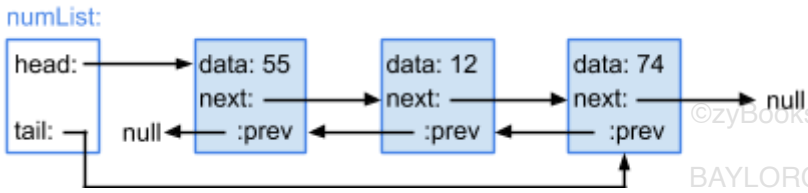
ListRemove(numsList, node 82)
ListRemove(numsList, node 357)
ListRemove(numsList, node 66)

numsList:

Check Show answer

PARTICIPATION ACTIVITY 15.11.3: 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

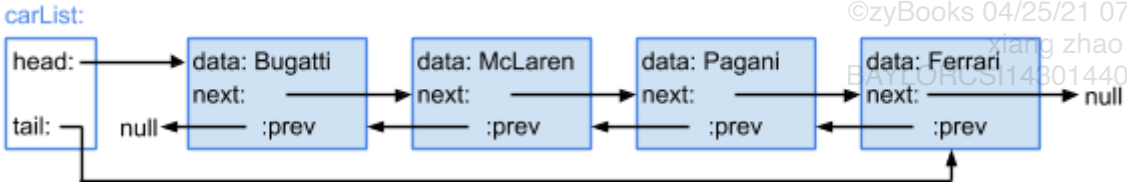
- Yes
- No

4) list⇢tail = predNode

- Yes
- No

PARTICIPATION ACTIVITY 15.11.4: 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 15.11.1: Doubly-linked lists: Remove.

Start

Given list: 4, 2, 6, 5, 9
What list results from the following operations?

ListRemoveAfter(list, node 2)
ListRemoveAfter(list, null)
ListRemoveAfter(list, node 5)

List items in order, from head to tail.

Ex: 25, 42, 12

1 2 3 4 5

Check Next

15.12 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 15.12.1: 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 15.12.1: 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 15.12.2: 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

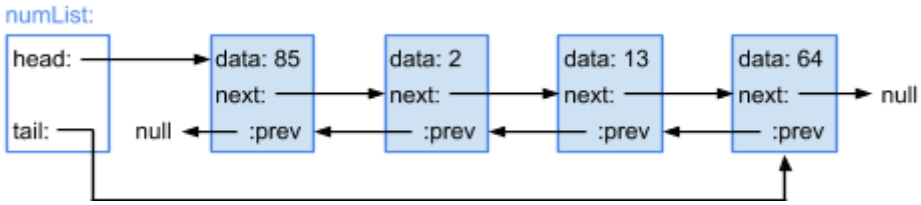
A doubly-linked list also supports a reverse traversal. A ***reverse traversal*** visits all nodes starting with the list's tail node and ending after visiting the list's head node.

Figure 15.12.2: 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 15.12.3: 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

15.13 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 15.13.1: Sorting a doubly-linked list with insertion sort.

Animation content:

`undefined`

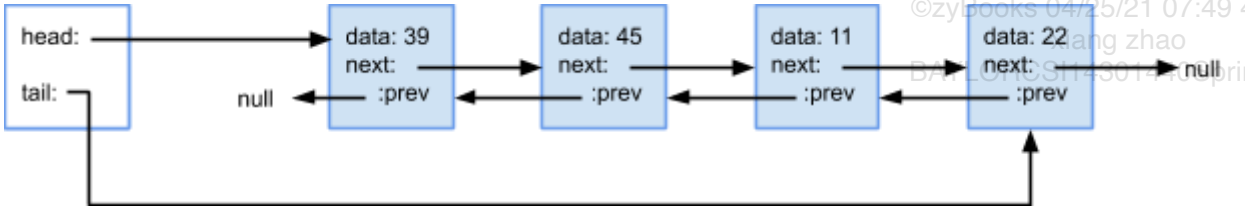
Animation captions:

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 15.13.2: Insertion sort for doubly-linked lists.

Suppose ListInsertionSortDoublyLinked is executed to sort the list below.



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

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

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

Algorithm efficiency

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

Insertion sort for singly-linked lists

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

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

PARTICIPATION ACTIVITY 15.13.3: Sorting a singly-linked list with insertion sort.

Animation content:

`undefined`

Animation captions:

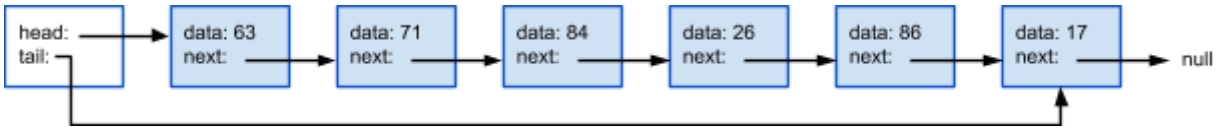
1. Insertion sort for a singly-linked list initializes curNode to point to the second list element, or node 56.
2. ListFindInsertionPosition searches the list from the head toward the current node to find the insertion position. ListFindInsertionPosition returns null, so Node 56 is prepended as the list 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 15.13.1: 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 15.13.4: Sorting singly-linked lists with insertion sort.

Given ListInsertionSortSinglyLinked is called to sort the list below.



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

- null
- Node 63
- Node 71
- Node 84

2) How many times is ListPrepend called?

- 0
- 1
- 2

3) How many times is ListInsertAfter called?

- 0
- 1
- 2

Algorithm efficiency

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

Sorting linked-lists vs. arrays

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

Table 15.13.1: 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 15.13.2: Sorting algorithms that cannot as efficiently sort linked lists.

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

PARTICIPATION ACTIVITY 15.13.5: 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.

15.14 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 15.14.1: 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 15.14.2: 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 15.14.3: Forward traversal concepts.

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

Searching

A recursive linked list search is implemented similar to forward traversal. Each call examines 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 15.14.1: ListSearch and ListSearchRecursive functions.


```
ListSearch(list, key) {
   return ListSearchRecursive(key, list→head)
}

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

PARTICIPATION ACTIVITY 15.14.4: 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 15.14.5: 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 15.14.6: Reverse traversal concepts.

Suppose ListTraverseReverse is called on the following list.

1) ListTraverseReverse passes _____ as the argument to ListTraverseReverseRecursive.

- node 67
-

node 18

- null

2) ListTraverseReverseRecursive has been called for each of the list's nodes by the time the tail node is visited.
   - True
   - False

3) If ListTraverseReverseRecursive were called directly on node 91, the nodes visited would be: _____.
   - node 91 and node 67
   - node 18, node 46, and node 91
   - node 18, node 46, node 91, and node 67

15.15 Stack abstract data type (ADT)

Stack abstract data type

A ***stack*** is an ADT in which items are only inserted on or removed from the top of a stack. The stack ***push*** operation inserts an item on the top of the stack. The stack ***pop*** operation removes and returns the item at the top of the stack. Ex: After the operations "Push 7", "Push 14", "Push 9", and "Push 5", "Pop" returns 5. A second "Pop" returns 9. A stack is referred to as a ***last-in first-out*** ADT. A stack can be implemented using a linked list, an array, or a vector.

PARTICIPATION ACTIVITY 15.15.1: Stack ADT.

Animation captions:

1. A new stack named "route" is created. Items can be pushed on the top of the stack.
2. Popping an item removes and returns the item from the top of the stack.

PARTICIPATION ACTIVITY 15.15.2: Stack ADT: Push and pop operations.

1) Given numStack: 7, 5 (top is 7).

Type the stack after the following push operation. Type the stack as: 1, 2, 3

Push(numStack, 8)

Check Show answer

2) Given numStack: 34, 20 (top is 34)
Type the stack after the following two push operations. Type the stack as: 1, 2, 3

Push(numStack, 11)
Push(numStack, 4)

Check Show answer

3) Given numStack: 5, 9, 1 (top is 5)
What is returned by the following pop operation?

Pop(numStack)

Check Show answer

4) Given numStack: 5, 9, 1 (top is 5)
What is the stack after the following pop operation? Type the stack as: 1, 2, 3

Pop(numStack)

Check Show answer

5) Given numStack: 2, 9, 5, 8, 1, 3 (top is 2).
What is returned by the second pop operation?

Pop(numStack)
Pop(numStack)

Check Show answer

6) Given numStack: 41, 8 (top is 41)
What is the stack after the following operations? Type the stack as: 1, 2, 3

Pop(numStack)
Push(numStack, 2)
Push(numStack, 15)
Pop(numStack)

Check Show answer

Common stack ADT operations

Table 15.15.1: Common stack ADT operations.

Operation	Description	Example starting with stack: 99, 77 (top is 99).
Push(stack, x)	Inserts x on top of stack	Push(stack, 44). Stack: 44, 99, 77
Pop(stack)	Returns and removes item at top of stack	Pop(stack) returns: 99. Stack: 77
Peek(stack)	Returns but does not remove item at top of stack	Peek(stack) returns 99. Stack still: 99, 77
IsEmpty(stack)	Returns true if stack has no items	IsEmpty(stack) returns false.
GetLength(stack)	Returns the number of items in the stack	GetLength(stack) returns 2.

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

PARTICIPATION ACTIVITY 15.15.3: Common stack ADT operations.

1) Given inventoryStack: 70, 888, -3, 2
What does GetLength(inventoryStack) return?
   - 4
   - 70

2) Given callStack: 2, 9, 4
What are the contents of the stack after Peek(callStack)?
   - 2, 9, 4
   - 9, 4

3) Given callStack: 2, 9, 4
What are the contents of the stack after Pop(callStack)?
   - 2, 9, 4
   - 9, 4

4) Which operation determines if the stack contains no items?
   - Peek
   - IsEmpty

5) Which operation should usually be preceded by a check that the stack is not empty?
   - Pop
   - Push

CHALLENGE ACTIVITY 15.15.1: Stack ADT.

Start

Given numStack: 29, 60, 63 (top is 29)

What is the stack after the following operations?

Pop(numStack)
Push(numStack, 45)
Push(numStack, 43)

Ex: 1, 2, 3

After the above operations, what does GetLength(numStack) return?

Ex: 5

Check Next

15.16 Stacks using linked lists

A stack is often implemented using a linked list, with the list's head node being the stack's top. A push is performed by creating a new list node, assigning the node's data with the item, and prepending the node to the list. A pop is performed by assigning a local variable with the head node's data, removing the head node from the list, and then returning the local variable.

PARTICIPATION ACTIVITY 15.16.1: Stack implementation using a linked list.

Animation content:

`undefined`

Animation captions:

1. Pushing 45 onto the stack allocates a new node and prepends the node to the list.
2. Each push prepends a new node to the list.
3. A pop assigns a local variable with the list's head node's data, removes the head node, and returns the local variable.

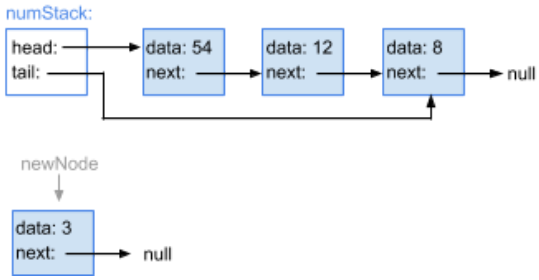
PARTICIPATION ACTIVITY 15.16.2: Stack push and pop operations with a linked list.

Assume the stack is implemented using a linked list.

1) An empty stack is indicated by a list head pointer value of _____.

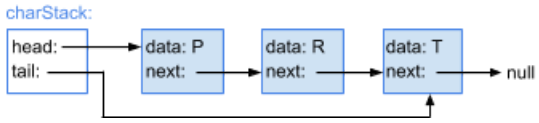
- newNode
- null
- Unknown

2) For StackPush(numStack, item 3), newNode's next pointer is pointed to _____.



- Node 54
- Node 12
- null

3) The operation StackPop(charStack) will remove which node?



- Node P
- Node R
- Node T

4) StackPop returns list's head node.

- True
- False

CHALLENGE ACTIVITY 15.16.1: Stacks using linked lists.

Start

Given an empty stack numStack, what does the list head pointer point to? If the pointer is null, enter null.

Ex: 5 or null

After the following operation, which node does the list head pointer point to?

StackPush(numStack, 34)

Ex: 5 or null

Check Next

15.17 Queue abstract data type (ADT)

Queue abstract data type

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 ***enqueue*** operation inserts an item at the end of the queue. The queue ***dequeue*** operation removes and returns the item at the front of the queue. Ex: After the operations "Enqueue 7", "Enqueue 14", and "Enqueue 9", "Dequeue" returns 7. A second "Dequeue" returns 14. A queue is referred to as a ***first-in first-out*** ADT. A queue can be implemented using a linked list or an array.

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 15.17.1: Queue ADT.

Animation content:

```
undefined
```

Animation captions:

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

PARTICIPATION ACTIVITY 15.17.2: Queue ADT.

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

Enqueue(numQueue, 4)

Check Show answer

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

Enqueue(numQueue, 28)
Enqueue(numQueue, 72)

Check Show answer

3) Given numQueue: 49, 3, 8
What is returned by the following dequeue operation?

Dequeue(numQueue)

Check Show answer

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

What is returned by the second dequeue operation?

Dequeue(numQueue)
Dequeue(numQueue)

Check Show answer

5) Given numQueue: 15, 91, 11
What is the queue after the following dequeue operation? Type the queue as: 1, 2, 3

Dequeue(numQueue)

Check Show answer

6) Given numQueue: 87, 21, 43
What are the queue's contents after the following operations? Type the queue as: 1, 2, 3

Dequeue(numQueue)
Enqueue(numQueue, 6)
Enqueue(numQueue, 50)
Dequeue(numQueue)

Check Show answer

Common queue ADT operations

Table 15.17.1: Some common operations for a queue ADT.

Operation	Description	Example starting with queue: 43, 12, 77 (front is 43)
Enqueue(queue,	Inserts x at end of the queue	Enqueue(queue, 56). Queue:

x)		43, 12, 77, 56
Dequeue(queue)	Returns and removes item at front of queue	Dequeue(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: Dequeue and Peek operations should not be applied to an empty queue; the resulting behavior may be undefined.

PARTICIPATION ACTIVITY 15.17.3: Common queue ADT operations.

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

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

CHALLENGE ACTIVITY 15.17.1: Queue ADT.

Start

Given numQueue: 70, 28, 16
What are the queue's contents after the following operations?

Dequeue(numQueue)
Dequeue(numQueue)
Enqueue(numQueue, 34)

Ex: 1, 2, 3

After the above operations, what does GetLength(numQueue) return?

Ex: 8

1 2 3 4

Check Next

15.18 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. Enqueueing an item is performed by creating a new list node, assigning the node's data with the item, and appending the node to the list. Dequeuing is performed by assigning a local variable with the head node's data, removing the head node from the list, and returning the local variable.

PARTICIPATION ACTIVITY 15.18.1: Queue implemented using a linked list.

Animation content:

undefined

Animation captions:

1. Enqueueing an item puts the item in a list node and appends the node to the list.
2. A dequeue stores the head node's data in a local variable, removes the list's head node, and returns the local variable.

PARTICIPATION ACTIVITY 15.18.2: Queue push and pop operations with a linked list.

Assume the queue is implemented using a linked list.

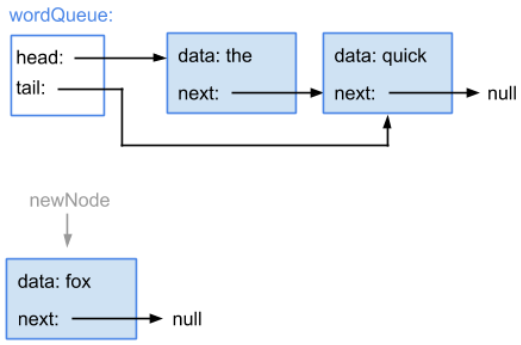
1) If the head pointer is null, the queue _____.
   - is empty
   - is full
   - has at least one item

2) For the operation QueueDequeue(queue), what is the second parameter passed to ListRemoveAfter?
   - The list's head node
   - The list's tail node
   - null

3) For the operation QueueDequeue(queue), headData is assigned with the list _____ node's data.
   - head

- tail

4) For QueueEnqueue(wordQueue, "fox"), which pointer is updated to point to the node?



- wordQueue's head pointer
- The head node's next pointer
- The tail node's next pointer

CHALLENGE ACTIVITY 15.18.1: Queues using linked lists.

Start

Given an empty queue numQueue, what does the list head pointer point to? If the pointer is null, enter null.

Ex: 5 or null

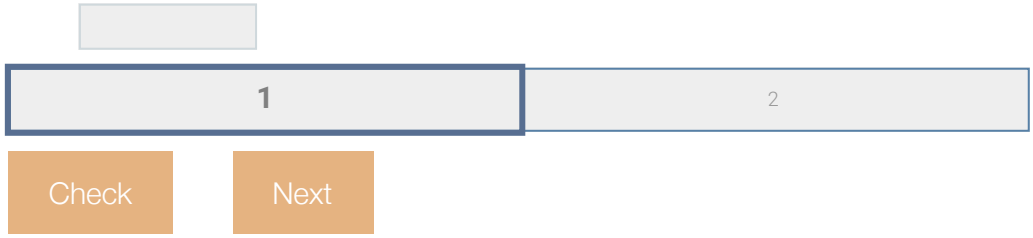
What does the list tail pointer point to?

After the following operations:

QueueEnqueue(numQueue, 66)
QueueEnqueue(numQueue, 99)
QueueEnqueue(numQueue, 12)
QueueDequeue(numQueue)

What does the list head pointer point to?

What does the list tail pointer point to?



15.19 Deque abstract data type (ADT)

Deque abstract data type

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 15.19.1: Deque ADT.

Animation captions:

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 15.19.2: Deque ADT.

Determine the deque contents after the following operations.

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

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

	45, 97, 68
	45, 71, 68
	68, 97, 45

Reset

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 15.19.1: 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	PushBack(deque, 41). Deque: 59, 63, 19, 41
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 15.19.3: 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

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)?
   - 6, 1, 7
   - 3, 6, 1, 7

5) Given that jobsDeque is empty, what does GetLength(jobsDeque) return?
   - -1
   - 0
   - Undefined

CHALLENGE 15.19.1: Deque ADT.

ACTIVITY

Start

Given an empty deque numDeque, what are the deque's contents after the following operatio

PushFront(numDeque, 27)
PushBack(numDeque, 26)

Ex: 1, 2, 3

After the above operations, what does PeekFront(numDeque) return?

Ex: 5

After the above operations, what does PeekBack(numDeque) return?

Ex: 5

After the above operations, what does GetLength(numDeque) return?

Ex: 5

1 2 3

Check Next

15.20 Lab 8 - C Linked List

This section's content is not available for print.

15.21 Lab 9 - Circular Linked List

This section's content is not available for print.