

10.1 List abstract data type (ADT)

List abstract data type

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



PARTICIPATION ACTIVITY

10.1.1: List ADT.



Animation captions:

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

PARTICIPATION ACTIVITY

10.1.2: List ADT.



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

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

 //

Check

[Show answer](#)



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

 //

Check

[Show answer](#)





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

Append(list, 3)

Append(list, 2)

Append(list, 1)

Remove(list, 2)

 //

Check

Show answer

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

Common list ADT operations

Table 10.1.1: Some common operations for a list ADT.

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

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, ©zyBooks 04/06/23 14:24 1369565 77 Gayoung Kim LEHIGHCSE017Spring2023
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

10.1.3: List ADT common operations.



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

4

Fails



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

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

True

False



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



- True
- False

10.2 Singly-linked lists

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

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`, `NUL`, and even the value `0`.

PARTICIPATION ACTIVITY

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



Animation content:

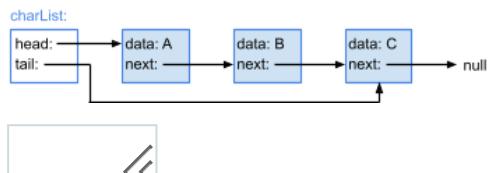
undefined

Animation captions:

1. A new list item is created, with the head and tail pointers pointing to nothing (null), meaning the list is empty.
2. ListAppend points the list's head and tail pointers to a new node, whose next pointer points to null.
3. Another append points the last node's next pointer and the list's tail pointer to the new node.
4. Operations like ListAppend, ListPrepend, ListInsertAfter, and ListRemove, update just a few relevant pointers.
5. The list's first node is called the head. The last node is the tail.



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

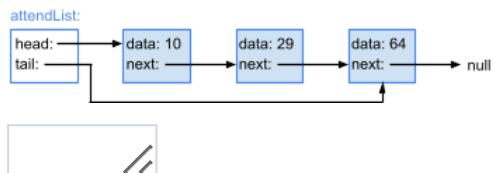


©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

Check

[Show answer](#)

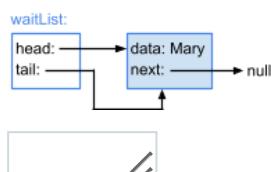
- 2) Given attendList, the head node's
data value is ____.
(Answer "None" if no head exists)



Check

[Show answer](#)

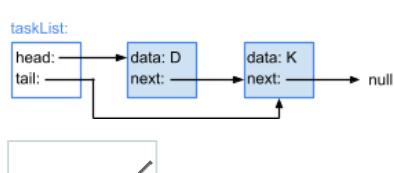
- 3) Given waitList, the tail node's data
value is ____.
(Answer "None" if no tail exists)



Check

[Show answer](#)

- 4) Given taskList, node D is followed by
node ____.



©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

Check

[Show answer](#)

Appending a node to a singly-linked list

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

©zyBooks 04/06/23 14:24 1369565

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

Gayoung Kim

LEHIGHCSE017Spring2023

- *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

10.2.3: Singly-linked list: Appending a node.



Animation content:

undefined

Animation captions:

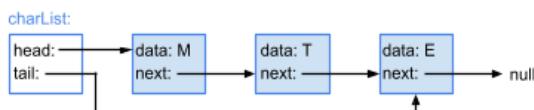
1. Appending an item to an empty list updates both the list's head and tail pointers.
2. Appending to a non-empty list adds the new node after the tail node and updates the tail pointer.

PARTICIPATION ACTIVITY

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



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



- M
- T
- E

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

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



sampleList:

head: null
tail: null

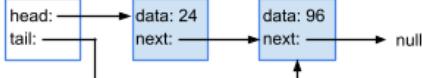
head and tail

head only

tail only

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

ticketList:



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

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

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

10.2.5: Singly-linked list: Prepending a node.



Animation content:

undefined

Animation captions:

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

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

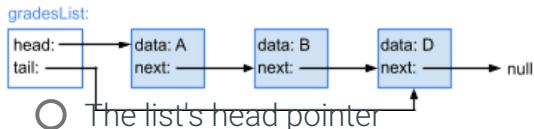
PARTICIPATION
ACTIVITY

10.2.6: Prepending a node in a singly-linked list.



- 1) Prepending C to gradesList updates which pointer?





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

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

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023



- 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

10.2.1: Singly-linked lists.

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

437612.2739130.qx3zqy7

Start

What is numList after the following operations?

```
numList = new List  
ListAppend(numList, node 43)  
ListAppend(numList, node 77)  
ListAppend(numList, node 37)  
ListAppend(numList, node 40)
```

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

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023



10.3 Singly-linked lists: Insert

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

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

PARTICIPATION
ACTIVITY

10.3.1: Singly-linked list: Insert nodes.



©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

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**

10.3.2: Inserting nodes in a singly-linked list.



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

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

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](#)

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

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

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

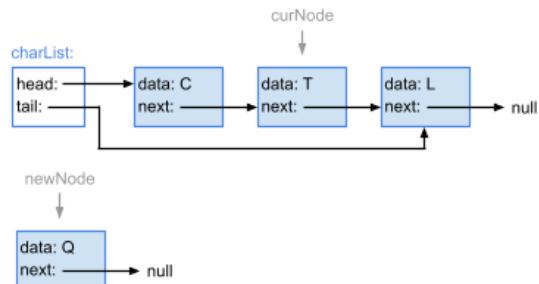
PARTICIPATION ACTIVITY

10.3.3: Singly-linked list insert-after algorithm.



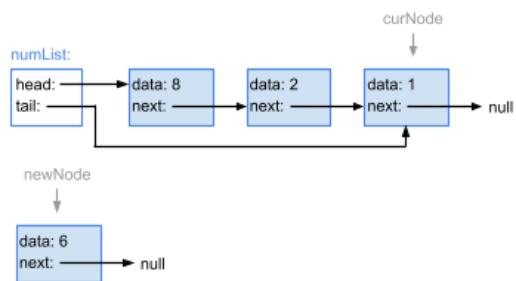
- 1) ListInsertAfter(charList, node T, node Q)
assigns newNode's next pointer with

_____.



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

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



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

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



©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023





curNode
↓
null

newNode
↓

[data: 246
next: → null]

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

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

**CHALLENGE
ACTIVITY**

10.3.1: Singly-linked lists: Insert.



437612.2739130.qx3zqy7

Start

What is numList after the following operations?

numList: 23, 81

ListInsertAfter(numList, node 23, node 34)

ListInsertAfter(numList, node 81, node 20)

ListInsertAfter(numList, node 81, node 57)

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

1

2

3

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

Check

Next

10.4 Singly-linked lists: Remove

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

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

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

- *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

10.4.1: 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

10.4.2: Removing nodes from a singly-linked list.

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023



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](#)

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

3) numsList: 9, 4, 11, 7



ListRemoveAfter(numsList, node
11)

numsList:

//[Check](#)[Show answer](#)

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



ListRemoveAfter(numsList, node
40)

ListRemoveAfter(numsList, node
20)

numsList:

//[Check](#)[Show answer](#)

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



ListRemoveAfter(numsList, node
60)

ListRemoveAfter(numsList, node
77)

ListRemoveAfter(numsList, null)

numsList:

//

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

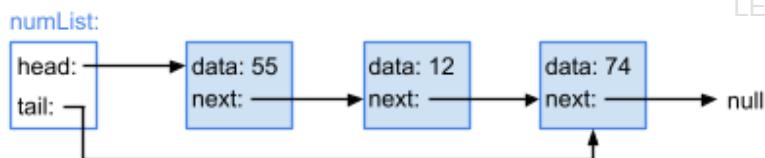
Check**Show answer****PARTICIPATION ACTIVITY**

10.4.3: ListRemoveAfter algorithm execution: Intermediate node.



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

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023



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

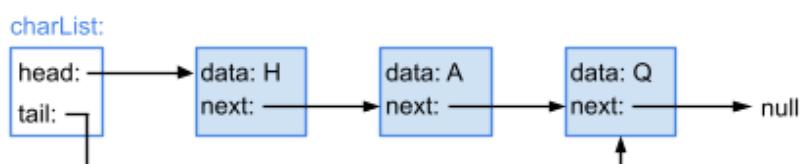
PARTICIPATION ACTIVITY

10.4.4: ListRemoveAfter algorithm execution: List head node.



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

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023



1) $\text{succNode} = \text{list} \rightarrow \text{head} \rightarrow \text{next}$



- Yes
- No

2) $\text{curNode} \rightarrow \text{next} = \text{succNode}$



- Yes
- No

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

3) $\text{list} \rightarrow \text{head} = \text{succNode}$



- Yes
- No

4) $\text{list} \rightarrow \text{tail} = \text{curNode}$



- Yes
- No

CHALLENGE ACTIVITY

10.4.1: Singly-linked lists: Remove.



437612.2739130.qx3zqy7

Start

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

What list results from the following operations?

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

List items in order, from head to tail.

Ex: 25, 42, 12

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

CheckNext

10.5 Linked list search

©zyBooks 04/06/23 14:24 1369565

Gayoung Kim

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

10.5.1: Singly-linked list: Searching.



Animation content:

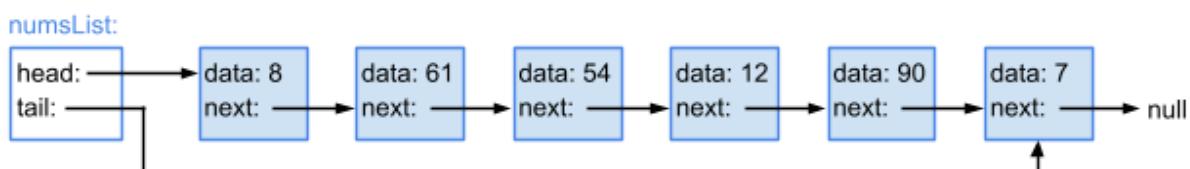
undefined

Animation captions:

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

PARTICIPATION
ACTIVITY

10.5.2: ListSearch algorithm execution.



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

 //

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

CheckShow 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?

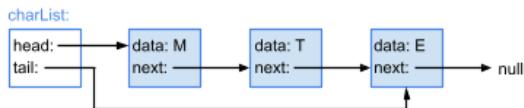
**Check****Show answer**

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

PARTICIPATION ACTIVITY

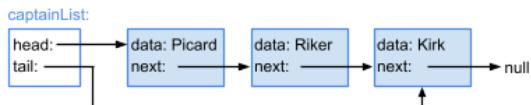
10.5.3: Searching a linked-list.

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



- Node M
- Node T
- Node E

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



- node Riker
- node Kirk

CHALLENGE ACTIVITY

10.5.1: Linked list search.

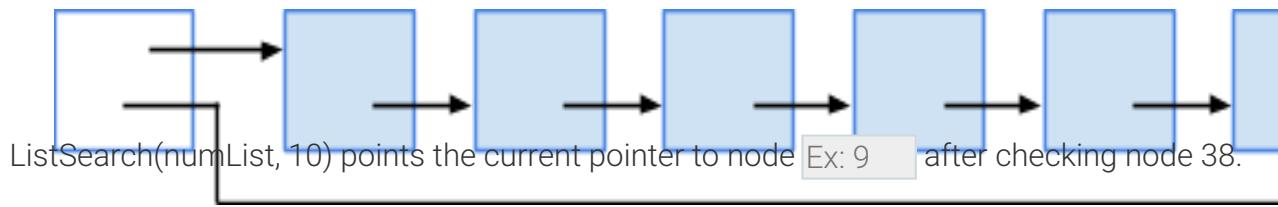
437612.2739130.qx3zqy7

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

Start

numList:

head:	data: 25	data: 26	data: 38	data: 10	data: 24	dat
tail:	next:	next:	next:	next:	next:	ne:



ListSearch(numList, 10) will make _____ comparisons.

©zyBooks 04/06/23 14:24 1369565

Gayoung Kim

2 LEHIGHCSE017Spring2023

1

Check

Next

10.6 Doubly-linked lists

Doubly-linked list

A **doubly-linked list** is a data structure for implementing a list ADT, where each node has data, a pointer to the next node, and a pointer to the previous node. The list structure typically 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

10.6.1: Doubly-linked list data structure.



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



- one
- two

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

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023



- 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

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

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

10.6.2: 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.

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

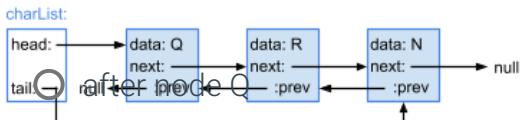
PARTICIPATION ACTIVITY

10.6.3: Doubly-linked list data structure.



- 1) ListAppend(charList, node F) inserts node F ____.

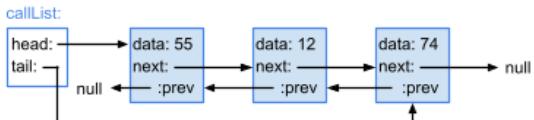




- before node N
 after node N

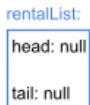
2) ListAppend(callList, node 5) executes which statement?

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023



- 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.

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

PARTICIPATION ACTIVITY

10.6.4: 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.

©zyBooks 04/06/23 14:24 1369565

Gayoung Kim

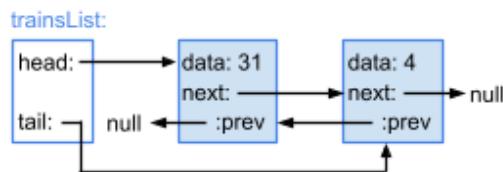
LEHIGHCSE017Spring2023

PARTICIPATION
ACTIVITY

10.6.5: Prepending a node in a doubly-linked list.



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



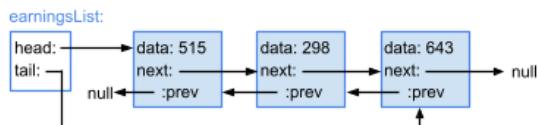
- 4
- 29
- 31

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



- True
- False

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



©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

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

CHALLENGE
ACTIVITY

10.6.1: Doubly-linked lists.



Start

```
numList = new List
ListAppend(numList, node 73)
ListAppend(numList, node 22)
```

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

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

Which node has a null next pointer? Ex: 5

Which node has a null previous pointer? Ex: 5

1

2

3

4

5

Check

Next

10.7 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.

**Animation content:**

undefined

Animation captions:

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

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.



Given weeklySalesList: 12, 30

Show the node order after the following operations:

ListInsertAfter(weeklySalesList, list tail, node 8)

ListInsertAfter(weeklySalesList, list head, node 45)

ListInsertAfter(weeklySalesList, node 45, node 76)

If unable to drag and drop, refresh the page.

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

Position 0 (list's head node)

©zyBooks 04/06/23 14:24 1369565

Gayoung Kim

LEHIGHCSE017Spring2023

Position 1

Position 2

Position 3

CHALLENGE
ACTIVITY

10.7.1: Doubly-linked lists: Insert.



437612.2739130.qx3zqy7

©zyBooks 04/06/23 14:24 1369565

Gayoung Kim

LEHIGHCSE017Spring2023

Start

What is numList after the following operations?

numList: 74, 73

ListInsertAfter(numList, node 73, node 67)

ListInsertAfter(numList, node 67, node 63)

ListInsertAfter(numList, node 74, node 90)

ListInsertAfter(numList, node 74, node 64)

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

What node does node 74's next pointer point to? What node does node 74's previous pointer point to?

1

2

3

4

Check

Next

10.8 Doubly-linked lists: Remove

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim

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.

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

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

10.8.1: 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.

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

PARTICIPATION ACTIVITY

10.8.2: 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

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023



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

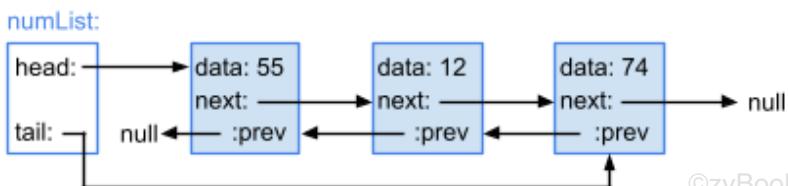
Show answer

PARTICIPATION ACTIVITY

10.8.3: ListRemove algorithm execution: Intermediate node.



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



©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

1) sucNode→prev = predNode



Yes

No

2) predNode→next = sucNode



Yes

No

3) $\text{list} \rightarrow \text{head} = \text{sucNode}$

Yes

No

4) $\text{list} \rightarrow \text{tail} = \text{predNode}$

Yes

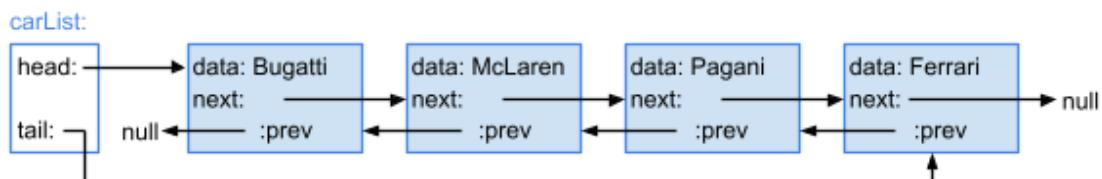
No

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

PARTICIPATION ACTIVITY

10.8.4: ListRemove algorithm execution: List head node.

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



1) $\text{sucNode} \rightarrow \text{prev} = \text{predNode}$

Yes

No

2) $\text{predNode} \rightarrow \text{next} = \text{sucNode}$

Yes

No

3) $\text{list} \rightarrow \text{head} = \text{sucNode}$

Yes

No

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

4) $\text{list} \rightarrow \text{tail} = \text{predNode}$

Yes

No



437612.2739130.qx3zqy7

Start

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

Given numList: 8, 7, 1, 2, 5, 4

What is numList after the following operations?

ListRemove(numList, node 2)

ListRemove(numList, node 8)

ListRemove(numList, node 4)

numList is now: Ex: 25, 42, 12



10.9 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 10.9.1: Linked list traversal algorithm.

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

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

PARTICIPATION
ACTIVITY

10.9.1: 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

10.9.2: 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).

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

- 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.

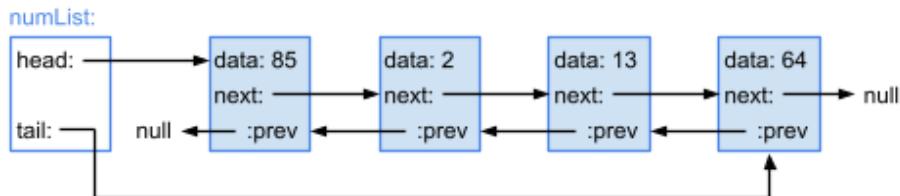
©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

Figure 10.9.2: Reverse traversal algorithm.

```
ListTraverseReverse(list) {  
    curNode = list->tail // Start at  
    tail  
  
    while (curNode is not null) {  
        Print curNode's data  
        curNode = curNode->prev  
    }  
}
```

PARTICIPATION
ACTIVITY

10.9.3: 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 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

10.10 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

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

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023



Animation content:

undefined

Animation captions:

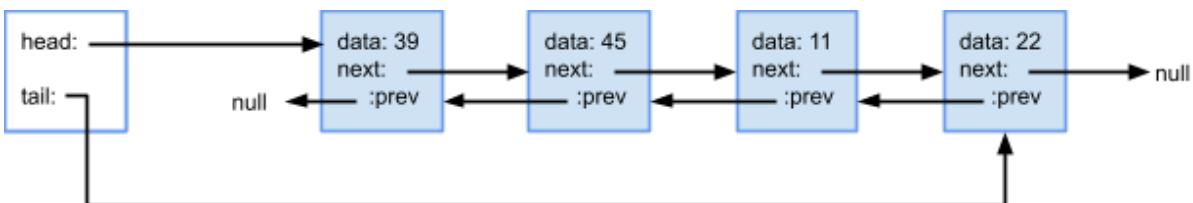
1. The curNode pointer begins at node 91 in the list.
2. searchNode starts at node 81 and does not move because 81 is not greater than 91. Removing and re-inserting node 91 after node 81 does not change the list.
3. For node 23, searchNode traverses the list backward until becoming null. Node 23 is prepended 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

10.10.2: 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

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023



- 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

©zyBooks 04/06/23 14:24 1369565

Gayoung Kim

LEHIGHCSE017Spring2023

Algorithm efficiency

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

Insertion sort for singly-linked lists

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

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

PARTICIPATION
ACTIVITY

10.10.3: Sorting a singly-linked list with insertion sort. 

©zyBooks 04/06/23 14:24 1369565

Gayoung Kim

LEHIGHCSE017Spring2023

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.

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

Figure 10.10.1: 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

10.10.4: 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

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

- 2) How many times is ListPrepend called?



- 0

1

2

- 3) How many times is ListInsertAfter called?

0

1

2

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023



Algorithm efficiency

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

Sorting linked-lists vs. arrays

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

Table 10.10.1: 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.

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

Table 10.10.2: Sorting algorithms difficult to adapt to efficiently sort linked lists.

Sorting algorithm	Challenge
Shell sort	Jumping the gap between elements cannot be done on a linked list, as each element between two elements must be traversed. <small>©zyBooks 04/06/23 14:24 1369565 Gayoung Kim LEHIGHCSE017Spring2023</small>
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

10.10.5: 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?



©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

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

©zyBooks 04/06/23 14:24 1369565

Gayoung Kim

LEHIGHCSE017Spring2023

10.11 Linked list dummy nodes

Dummy nodes

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

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

PARTICIPATION ACTIVITY

10.11.1: Singly-linked lists with and without a dummy node.



Animation captions:

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

PARTICIPATION ACTIVITY

10.11.2: Singly linked lists with a dummy node.



- 1) The head and tail pointers always point to the dummy node.

- True
- False

- 2) The dummy node's next pointer points to the first list item.

- True
- False

©zyBooks 04/06/23 14:24 1369565

Gayoung Kim

LEHIGHCSE017Spring2023





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

- myList->head == null
- myList->tail == null
- myList->head == myList->tail

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

Singly-linked list implementation

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

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

Figure 10.11.1: Singly-linked list with dummy node: append, prepend, insert after, and remove after operations.

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

```

ListAppend(list, newNode) {
    list->tail->next = newNode
    list->tail = newNode
}

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

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

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

        if (sucNode is null) {
            // Removed tail
            list->tail = curNode
        }
    }
}

```

©zyBooks 04/06/23 14:24 1369565
 Gayoung Kim
 LEHIGHCSE017Spring2023

PARTICIPATION ACTIVITY

10.11.4: Singly-linked list with dummy node.



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

©zyBooks 04/06/23 14:24 1369565
 Gayoung Kim
 LEHIGHCSE017Spring2023

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

- `ListRemoveAfter(dataList, null)`
- `ListRemoveAfter(dataList, dataList->head)`

- ListRemoveAfter(dataList,
dataList->tail)

2) Which is a requirement of the ListPrepend function?

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

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

PARTICIPATION ACTIVITY

10.11.5: Singly-linked list with dummy node.

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

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

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

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

2) Suppose the following statement is executed:

```
node19 =
numbersList->head->next->next
```

Which subsequent operations swap nodes 73 and 19?

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

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

Doubly-linked list implementation

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

Figure 10.11.2: Doubly-linked list with dummy node: append, prepend, insert after, and remove operations.

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023


```

ListAppend(list, newNode) {
    list->tail->next = newNode
    newNode->prev = list->tail
    list->tail = newNode
}

ListPrepend(list, newNode) {
    firstNode = list->head->next
    // Set the next and prev pointers for newNode
    newNode->next = list->head->next
    newNode->prev = list->head

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

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

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

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

    sucNode = curNode->next
    predNode = curNode->prev

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

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

    if (curNode == list->tail) { // Removed tail
        list->tail = predNode
    }
}

```

©zyBooks 04/06/23 14:24 1369565

Gayoung Kim
LEHIGHCSE017Spring2023

©zyBooks 04/06/23 14:24 1369565

Gayoung Kim
LEHIGHCSE017Spring2023



- 1) `ListPrepend(list, newNode)` is equivalent to

```
ListInsertAfter(list,  
list->head, newNode).
```

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

- True
- False

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

- True
- False

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

- True
- False

Dummy head and tail nodes

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



Animation content:

undefined

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

Animation captions:

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

Figure 10.11.3: Doubly-linked list with 2 dummy nodes: insert after and remove operations.

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

    sucNode = curNode->next
    newNode->next = sucNode
    newNode->prev = curNode
    curNode->next = newNode
    sucNode->prev = newNode
}

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

    sucNode = curNode->next
    predNode = curNode->prev

    // Successor node is never null
    sucNode->prev = predNode

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

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

Removing if statements from ListInsertAfter and ListRemove

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

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023



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

1) When `list->head == list->tail` is true in _____, the list is empty.

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

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

2) `list->tail` may be null in _____.

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

3) `list->head->next` is always non-null in _____.

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

10.12 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 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

PARTICIPATION ACTIVITY

10.12.1: 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.

©zyBooks 04/06/23 14:24 1369565

Gayoung Kim

LEHIGHCSE017Spring2023

PARTICIPATION ACTIVITY

10.12.2: 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

10.12.3: 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 10.12.1: 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
}

```

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

PARTICIPATION ACTIVITY

10.12.4: 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

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

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

10.12.5: Recursive reverse traversal.



Animation content:

undefined

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

Animation captions:

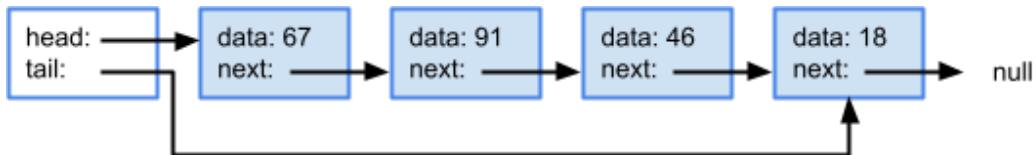
1. ListTraverseReverse is called to traverse the list. Much like a forward traversal, ListTraverseReverseRecursive is called for the list's head.
2. The recursive call on node 19 is made before visiting node 23.
3. Similarly, the recursive call on node 41 is made before visiting node 19, and the recursive 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

10.12.6: 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

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023



- 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 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

10.13 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

10.13.1: 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

10.13.2: 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

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

Push(numStack, 8)

Show answer

Check

- 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](#)

©zyBooks 04/06/23 14:24 1369565

Gayoung Kim

LEHIGHCSE017Spring2023

- 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](#)

©zyBooks 04/06/23 14:24 1369565

Gayoung Kim

LEHIGHCSE017Spring2023



- 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

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

Common stack ADT operations

Table 10.13.1: 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.

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

PARTICIPATION ACTIVITY

10.13.3: 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

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

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**

10.13.1: Stack ADT.



437612.2739130.qx3zqy7

Start

Given numStack: 32, 47, 35 (top is 32)

What is the stack after the operations?

Pop(numStack)

Push(numStack, 66)

Pop(numStack)

Push(numStack, 33)

Ex: 1, 2, 3

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

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

Ex: 5

1	2	3
Check	Next	

©zyBooks 04/06/23 14:24 1369565

Gayoung Kim

LEHIGHCSE017Spring2023

10.14 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

10.14.1: 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

10.14.2: Stack push and pop operations with a linked list.

©zyBooks 04/06/23 14:24 1369565

Gayoung Kim

LEHIGHCSE017Spring2023

Assume the stack is implemented using a linked list.

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

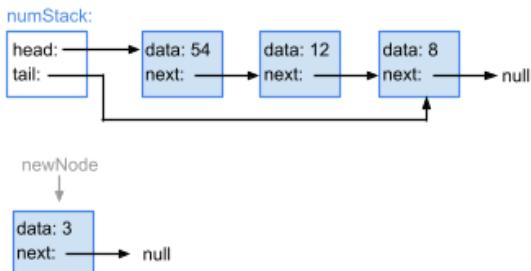
- newNode
- null



Unknown

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

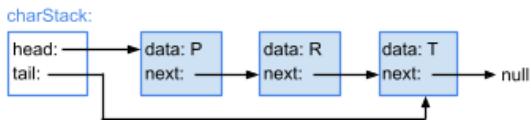
_____.



©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

- 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

10.14.1: Stacks using linked lists.

437612.2739130.qx3zqy7

Start

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

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 operations, which node does the list head pointer point to?

StackPush(numStack, 37)
StackPush(numStack, 89)

Ex: 5 or null

1

2

©zyBooks 04/06/23 14:24 1369565

3 Gayoung Kim

LEHIGHCS017Spring2023

Check

Next

10.15 Array-based stacks

Array-based stack storage

A stack can be implemented with an array. Two variables are needed in addition to the array:

- allocationSize: an integer for the array's allocated size.
- length: an integer for the stack's length.

The stack's bottom item is at `array[0]` and the top item is at `array[length - 1]`. If the stack is empty, length is 0.

PARTICIPATION
ACTIVITY

10.15.1: Array-based stack storage.



Animation content:

undefined

Animation captions:

1. Stack 1's allocationSize and length are both 4. Array element 0 is at the stack's bottom. Array element 3 is at the stack's top.
2. Stack 2's length, 2, is less than allocationSize, 4. So elements at indices 2 and 3 are not part of the stack content.
3. The stack is empty when length is 0, regardless of array content.

Unbounded stack

An **unbounded stack** is a stack with no upper limit on length. An unbounded stack's length can increase indefinitely, so the stack's array allocation size must also be able to increase indefinitely.

PARTICIPATION
ACTIVITY

10.15.2: Unbounded, array-based stack.



Animation content:

undefined

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

Animation captions:

1. The stack's initial allocation size is 1 and length is 0. So pushing 37 assigns array[0] with 37.
2. Pushing 88 occurs when allocationSize and length are both 1. So a new array is allocated, the existing entry is copied, and array[1] is assigned with 88.
3. Pushing 71 occurs when allocationSize and length are both 2. So a new array is allocated, the existing entries are copied, and array[2] is assigned with 71.

PARTICIPATION
ACTIVITY

10.15.3: Unbounded stack.



Refer to the example above.

- 1) In the example above, when the array is reallocated, the size ____.

- increases by 1
- doubles
- decreases by 1

- 2) When does a push operation resize the stack's array?

- When `length == allocationSize`
- When `length == allocationSize - 1`
- Every push operation resizes the stack's array

- 3) In theory, an unbounded stack can grow in length indefinitely. In reality, the stack can ____.

- also grow in length indefinitely

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023



- grow only until all distinct 32-bit integer values are pushed
- grow only until the resize operation fails to allocate memory

Bounded stack

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

A **bounded stack** is a stack with a length that does not exceed a maximum value. The maximum is commonly the initial allocation size. Ex: A bounded stack with allocationSize = 100 cannot exceed a length of 100 items.

A bounded stack with a length equal to the maximum length is said to be **full**.

PARTICIPATION ACTIVITY

10.15.4: Bounded, array-based stack.



Animation content:

undefined

Animation captions:

1. One implementation approach for a bounded stack is to allocate the array to the maximum length upon construction.
2. Three values push successfully.
3. The fourth push fails because length equals maxLength.
4. An alternate bounded stack implementation may distinguish allocation size and max length.
5. The first push assigns array[0] with 53. The second push must resize first, increasing allocationSize from 1 to 2.
6. Pushing 91 can now complete.
7. The next push must also resize. The $\min(\text{allocationSize} * 2, \text{maxLength}) = 3$ is the new allocation size.
8. The next push fails because length equals maxLength.

PARTICIPATION ACTIVITY

10.15.5: Bounded stack - general implementation.

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023



- 1) A bounded stack's maximum length and initial allocation size are always equal.
 - True
 - False



2) A bounded stack implementation may throw an exception if a push operation occurs when full.

- True
- False

3) Array-based stack implementations commonly reallocate when popping.

- True
- False

©zyBooks 04/06/23 14:24 1369535
Gayoung Kim
LEHIGHCSE017Spring2023

PARTICIPATION ACTIVITY

10.15.6: Bounded stack - two implementation approaches.

Consider two bounded stack implementation approaches.

Implementation A: Allocation size is provided at construction time and is the stack's maximum length.

Implementation B: Maximum length is specified at construction time. Default allocation size is 1.

1) Which implementation(s) may need to reallocate the array to push an item?

- Implementation A only
- Implementation B only
- Both
- Neither

2) Which implementation(s) may allow a push when `length == maxLength`?

- Implementation A only
- Implementation B only
- Neither

3) Which implementation(s) guarantee that both push and pop execute in worst-case O(1) time?

- Implementation A only
- Implementation B only
- Both
- Neither

©zyBooks 04/06/23 14:24 1369535
Gayoung Kim
LEHIGHCSE017Spring2023

Single implementation that allows bounded or unbounded

An array-based stack implementation can support both bounded and unbounded stack operations by using `maxLength` as follows:

- If `maxLength` is negative, the stack is unbounded.
- If `maxLength` is nonnegative, the stack is bounded.

©zyBooks 04/06/23 14:24 1369565

The push operation does not allow a push when `length == maxLength`. The stack's length is always nonnegative, so:

LEHIGHCSE017Spring2023

- If `maxLength` is negative, the condition is never true. So all push operations are allowed, and therefore the stack is unbounded.
- If `maxLength` is nonnegative, the condition is true when the stack is full. So push operations are not allowed when full, and therefore the stack is bounded.

PARTICIPATION ACTIVITY

10.15.7: Stack push, resize, and pop operations.



Animation content:

undefined

Animation captions:

1. ArrayStackPush begins with a max length check. $0 \neq 3$, so the stack's length does not equal the maximum.
2. The stack's length, 0, does not equal allocation size, 1. So no resize is needed.
3. Array element 0 is assigned with 79, length is increased to 1, and true is returned.
4. When pushing 16, length and allocationSize are both 1, so ArrayStackResize() is called. The new allocation size is $1 * 2 = 2$.
5. 79 is copied, the stack's array and allocationSize are reassigned, and then execution returns to ArrayStackPush to complete the push.
6. Length and allocation size are both 2, so pushing 54 requires a resize. $\min(2 * 2, 3) = 3$ is the new allocation size.
7. The next push fails because length equals maxLength.
8. Popping 54 decreases the stack's length, but not allocation size. Popping does not resize the stack.

©zyBooks 04/06/23 14:24 1369565

Gayoung Kim

LEHIGHCSE017Spring2023

PARTICIPATION ACTIVITY

10.15.8: ArrayStackInitialize() function.



Assume the push, resize, and pop operations are implemented as in the animation above. Consider an `ArrayStackInitialize()` function that initializes the stack before any operations occur.

1) If `maxLength` is assigned with -1, then
ArrayStackInitialize() may assign
`allocationSize` with ____.

- 1
- 0
- 2

2) If `maxLength` is assigned with 64, then
ArrayStackInitialize() may assign
`allocationSize` with ____.

- any integer
- any nonnegative integer
- any positive integer

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

PARTICIPATION
ACTIVITY

10.15.9: Using ArrayStack functions for an unbounded stack.

Assume a stack is initialized as follows:

```
integerStack->allocationSize = 1
integerStack->array = Allocate array of size 1
integerStack->length = 0
integerStack->maxLength = -1
```

Assume the push, resize, and pop functions are implemented as in the animation above and that the operations below execute in question order.

1) `ArrayStackPush(integerStack,`
21) ____.

- does not change the stack and returns false
- causes an out-of-bounds write in the stack's array
- successfully pushes 21 and returns true

2) `ArrayStackPush(integerStack,`
78) ____.

- does not change the stack and returns false
- causes an out-of-bounds write in the stack's array

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

- resizes, pushes 78, and returns true
- 3) Two more calls to ArrayStackPush() occur. `integerStack`'s allocationSize, length, and maxLength are ____, respectively.



- 2, 2, and -1
- 4, 4, and -1
- 4, 4, and 4

- 4) If each resize operation succeeds, the next 100 ArrayStackPush() operations succeed.



- True
- False

- 5) ArrayStackPop(), ArrayStackPush(), and ArrayStackResize() work for both bounded and unbounded stacks.



- True
- False

PARTICIPATION ACTIVITY

10.15.10: Array-based stack operation complexity.



- 1) For an unbounded stack, what operations take O(1) time in the worst case?



- ArrayStackPop() only
- ArrayStackPush() and ArrayStackPop()
- ArrayStackPush(), ArrayStackPop(), and ArrayStackResize()

- 2) For a bounded stack that does *not* resize, what operations take O(1) time in the worst case?



- ArrayStackPush() only
- ArrayStackPop() only

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

- ArrayStackPush() and
ArrayStackPop()

10.16 Queue abstract data type (ADT)

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

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

10.16.1: 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

10.16.2: 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

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

Enqueue(numQueue, 4)

Show answer

Check

- 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**

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

- 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**

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

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)

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

Check

Show answer

Common queue ADT operations

Table 10.16.1: Some common operations for a queue ADT.

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

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

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

PARTICIPATION ACTIVITY

10.16.3: 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

10.16.1: Queue ADT.

437612.2739130.qx3zqy7

Start

Given numQueue: 23, 84

What are the queue's contents after the following operations?

Dequeue(numQueue)
Enqueue(numQueue, 52)

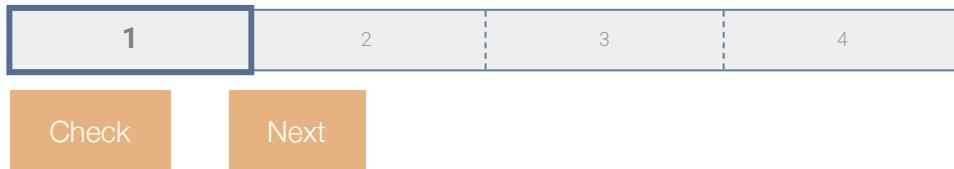
©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

Dequeue(numQueue)
Enqueue(numQueue, 96)

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

Ex: 8

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023



10.17 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. Dequeueing 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

10.17.1: Queue implemented using a linked list.



Animation content:

undefined

Animation captions:

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

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

10.17.2: 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?

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023



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

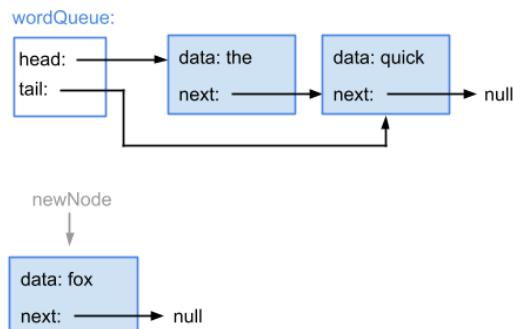
3) For the operation



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

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023



CHALLENGE ACTIVITY

10.17.1: 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?

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

After the following operations:

QueueEnqueue(numQueue, 70)
QueueEnqueue(numQueue, 33)
QueueEnqueue(numQueue, 42)
QueueEnqueue(numQueue, 60)
QueueDequeue(numQueue)

What does the list head pointer point to?

What does the list tail pointer point to?

1

2

Check

Next

10.18 Array-based queues

Array-based queue storage

A queue can be implemented with an array. Three variables are needed in addition to the array:

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

- allocationSize: an integer for the array's allocated size.
- length: an integer for the number of items in the queue.
- frontIndex: an integer for the queue's front item index.

The queue's content starts at `array[frontIndex]` and continues forward through `length` items. If the array's end is reached before encountering all items, remaining items are stored starting at index 0.

**Animation content:**

undefined

Animation captions:

©zyBooks 04/06/23 14:24 1369565

Gayoung Kim

LEHIGHCSE017Spring2023

1. Queue content starts at frontIndex. When frontIndex is 0 and length equals allocationSize, the queue's content matches the array's content: 78, 64, 16, 95.
2. If frontIndex is 2, then the queue's first two items are 16 and 95. Looping back to 0 gives the last two items, 78 and 64.
3. A length of 3 is one less than the allocation size, so 64 is not part of the queue.
4. The queue is empty when length is 0, regardless of array content.



A queue's array is [67, 19, 44, 38] and allocationSize is 4.

- 1) What is the queue's front item if frontIndex is 3 and length is 2?

- 67
- 44
- 38



- 2) What is the queue's back item if frontIndex is 0 and length is 3?

- 67
- 44
- 38



- 3) What is the queue's content if length is 4 and frontIndex is 1?

- (front) 67, 19, 44, 38 (back)
- (front) 19, 44, 38 (back)
- (front) 19, 44, 38, 67 (back)

©zyBooks 04/06/23 14:24 1369565

Gayoung Kim

LEHIGHCSE017Spring2023

Bounded vs. unbounded queue

A **bounded queue** is a queue with a length that does not exceed a specified maximum value. An additional variable, `maxLength`, is needed. `maxLength` is commonly assigned at construction time and does not change for the queue's lifetime. A bounded queue with a length equal to the maximum length is said to be **full**.

An **unbounded queue** is a queue with a length that can grow indefinitely.

PARTICIPATION
ACTIVITY

10.18.3: Bounded vs. unbounded queue.

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023



Animation content:

undefined

Animation captions:

1. Three queues are created: the first unbounded, the second bounded with a maximum length of 2, and the third bounded with a maximum length of 4.
2. Enqueueing 51 then 88 yields the same results for each queue.
3. Enqueueing 73 fails for the full bounded queue, because the length and maximum length are both 2. Enqueueing 73 succeeds for the other two queues.
4. Similarly, enqueueing 47 succeeds for the first and third queues and fails for the second.
5. Both bounded queues now have a length equal to the maximum length. So enqueueing 24 succeeds only for the unbounded queue.

PARTICIPATION
ACTIVITY

10.18.4: Bounded vs. unbounded queue.



1) ____ can be full.

- Only a bounded queue
- Only an unbounded queue
- Both bounded and unbounded queues



2) A bounded queue implementation may throw an exception if an enqueue operation occurs when full.

- True
- False



3) The `maxLength` variable is needed _____.

- only for the bounded queue implementation



©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

- only for the unbounded queue implementation
- for both the bounded and unbounded queue implementations

Flexible implementation and resize operation

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

An array-based queue implementation can support both bounded and unbounded queue operations by using `maxLength` as follows:

- If `maxLength` is negative, the queue is unbounded.
- If `maxLength` is nonnegative, the queue is bounded.

The resize operation allocates an array of larger size, commonly double the existing allocation size. If `maxLength` is nonnegative, the minimum of `maxLength` and double the current allocation size is used. Existing array entries are copied in such a way that the front index is reset to 0.

PARTICIPATION
ACTIVITY

10.18.5: Array-based queue resize operation.



Animation content:

undefined

Animation captions:

1. `maxLength = 4`, so `queue1` is bounded. `length` and `allocationSize` are both 3, so `queue1` is full.
2. `frontIndex` is 1, so `queue1`'s content from front to back is: 16, 53, 97.
3. Resizing `queue1` begins with computing `newSize`. Double the current allocation size is 6. But `maxLength` is 4, which is nonnegative and less than 6. So 4 is used instead.
4. The new array is allocated. The first element is at index 1 in the existing array and is copied to index 0 in the new array.
5. Remaining elements are copied. Then array, `allocationSize`, and `frontIndex` are reassigned.
6. `queue2`'s `maxLength` is -1 and `maxLength` is never reassigned. So when resizing, the condition `queue->maxLength >= 0` is always false.
7. So `queue2` is unbounded, and each resize doubles the allocation size.

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

PARTICIPATION
ACTIVITY

10.18.6: Flexible implementation and resize operation.



Consider `queueA` with variables:

```
allocationSize: 4
array: [73, 91, 87, 62]
frontIndex: 2
length: 4
maxLength: -1
```

1) queueA is ____.

- bounded and full
- bounded and not full
- unbounded



©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

2) After executing

`ArrayQueueResize(queueA),`
queueA's allocation size is ____.

- 1
- 4
- 8



3) After executing

`ArrayQueueResize(queueA),`
queueA.queue_list's first four
elements are ____.

- 73, 91, 87, 62
- 87, 62, 73, 91



4) Can one of queueA's initial variables be
changed such that executing

`ArrayQueueResize(queueA)`
makes queueA's allocation size 6?

- Yes, if `maxLength` were initially 6
- Yes, if `allocationSize` were
initially 6
- No



Enqueue and dequeue operations

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

An enqueue operation:

1. Compares `length` and `maxLength`. If equal, the queue is full and so no change occurs and false is returned.
2. Compares `length` and `allocationSize`. If equal, a resize operation occurs.
3. Computes the enqueued item's index as `(frontIndex + length) % allocationSize` and assigns to the array at that index. Ex: Enqueueing 42 into a queue with `frontIndex` 2, `length` 3, and

allocationSize 8 assigns the queue array at index $(2 + 3) \% 8 = 5$ with 42.

4. Increments `length` and then returns true.

A dequeue operation:

1. Makes a copy of the array item at `frontIndex`.
2. Decrement `length`.
3. Increments `frontIndex`, resetting to 0 if the incremented value equals the allocation size.
©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023
4. Returns the array item from step 1.

PARTICIPATION ACTIVITY

10.18.7: Array-based queue enqueue and dequeue operations.



Animation content:

undefined

Animation captions:

1. `numQueue` is an empty queue with `allocationSize = 1`. Enqueuing 42 starts by comparing `length` against `maxLength`, then `allocationSize`.
2. `array[0]` is assigned with 42, `length` is incremented, and true is returned.
3. When enqueueing 89, a resize occurs first. Then `array[1]` is assigned with 89.
4. Enqueueing 71 resizes again, increasing `allocationSize` to 4. Then 64 is enqueued without resizing.
5. `frontIndex = 0`, so the dequeue operation begins by assigning `toReturn` with `array[0]`. `Length` is decremented to 3, `frontIndex` is reassigned with 1, and 42 is returned.
6. 89 is dequeued similarly.
7. 91 is enqueued at index 0.

PARTICIPATION ACTIVITY

10.18.8: Enqueue and dequeue operations.



1) ____ may resize the queue.



- Only the enqueue operation
- Only the dequeue operation
- Both the enqueue and dequeue operations

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

2) `(frontIndex + length) % allocationSize` yields the index



- of the item to remove during a dequeue operation

- at which to insert a new item during an enqueue operation
 - of the queue's back item
- 3) ArrayQueueEnqueue() returns false if



- the item is successfully enqueued
- the resize operation fails
- the queue is full

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

- 4) An alternate implementation could store the queue's back item index instead of the length.



- True
- False

Worst-case time complexity

Using the implementation from above, worst-case time complexities are the same whether the queue is bounded or unbounded: $O(N)$ for enqueue and $O(1)$ for dequeue.

An alternate implementation of the bounded queue, not presented in this section, uses `maxLength` as the array's initial allocation size. Such an implementation never needs to resize and so the enqueue operation's worst-case time is $O(1)$.

10.19 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.

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023



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, 19.²⁴"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.



Determine the deque contents after the following operations.

If unable to drag and drop, refresh the page.

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

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

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

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 deque and determining the length. A **peek** operation returns an item in the deque without removing the item.

©zyBooks 04/06/23 14:24 1369565

Table 10.19.1: 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 ©zyBooks 04/06/23 14:24 1369565 Gayoung Kim LEHIGHCSE017Spring2023
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

10.19.3: Common deque 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

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

CHALLENGE ACTIVITY

10.19.1: Deque ADT.

437612.2739130.qx3zqy7

Start

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

PushBack(numDeque, 10)
PushFront(numDeque, 56)
PushBack(numDeque, 95)

Ex: 1, 2, 3 (commas between values)

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

Ex: 5

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

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

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

[Check](#)[Next](#)

10.20 Array-based lists

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

Array-based lists

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

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

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

PARTICIPATION
ACTIVITY

10.20.1: 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

10.20.2: Array-based lists.



- 1) The length of an array-based list equals the list's array allocation size.
 True
 False
- 2) 42 is appended to an array-based list with allocationSize = 8 and length = 4. Appending assigns the array at index ____ with 42.

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023



4

8

- 3) An array-based list can have a default allocation size of 0.

True

False

©zyBooks 04/06/23 14:24 1369565

Gayoung Kim

LEHIGHCSE017Spring2023



Resize operation

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

Because all existing elements must be copied from 1 array to another, the resize operation has a runtime complexity of $O(n)$.

PARTICIPATION
ACTIVITY

10.20.3: Array-based list resize operation.



Animation content:

undefined

Animation captions:

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

PARTICIPATION
ACTIVITY

10.20.4: Array-based list resize operation.



©zyBooks 04/06/23 14:24 1369565

Gayoung Kim

LEHIGHCSE017Spring2023

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



1) Which operation causes ArrayListResize to be called?

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

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023



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

- 5
- 7
- 10



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

- 5
- 7
- 10

Prepend and insert after operations

The **Prepend** operation for an array-based list inserts a new item at the start of the list. First, if the allocation size equals the list length, the array is resized. Then all existing array elements are moved up by 1 position, and the new item is inserted at the list start, or index 0. Because all existing array elements must be moved up by 1, the prepend operation has a runtime complexity of $O(n)$.

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 $O(n)$.

Gayoung Kim
LEHIGHCSE017Spring2023

InsertAt operation.

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

InsertAfter to insert after index X - 1.

**PARTICIPATION
ACTIVITY**

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



Animation content:

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

Step 1: Data members for a list are shown: array, allocationSize, and length.

"array:" label is followed by 8 boxes for the array's data. Entries at indices 0 to 4 are: 45, 84, 12, 78, 51.

Entries at indices 5, 6, and 7 are empty.

The other two labels are "allocationSize: 8" and "length: 5".

ArrayListPrepend(list, 91) begins execution. The first if statement's condition is false. Then the for loop executes, moving items up in the array, yielding: 45, 45, 84, 12, 78, 51. Array boxes for indices 6 and 7 remain empty.

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

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

Animation captions:

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

**PARTICIPATION
ACTIVITY**

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



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

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

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)

©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

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



- 5
- 8
- 10

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



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

Search and removal operations

Given a key, the **search** operation returns the index for the first element whose data matches that key, or -1 if not found.

Given the index of an item in an array-based list, the **remove-at** operation removes the item at that index. When removing an item at index X, each item after index X is moved down by 1 position.

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

PARTICIPATION ACTIVITY

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



©zyBooks 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

Animation content:

undefined

Animation captions:

1. The search for 84 compares against 3 elements before returning 2.

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

PARTICIPATION
ACTIVITY

10.20.8: Search and remove-at operations.

©zyBooks 04/06/23 14:24 1369565

Gayoung Kim

LEHIGHCSE017Spring2023

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



- 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 04/06/23 14:24 1369565

Gayoung Kim

LEHIGHCSE017Spring2023

PARTICIPATION ACTIVITY

10.20.9: 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 04/06/23 14:24 1369565
Gayoung Kim
LEHIGHCSE017Spring2023

**CHALLENGE ACTIVITY**

10.20.1: Array-based lists.



437612.2739130.qx3zqy7

Start

numList:

15	52	68	
----	----	----	--

 allocationSize: 4
length: 3

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

Operation	Length	Allocation size
ArrayListAppend(numList, 12)	Ex: 1	Ex:1
ArrayListAppend(numList, 21)		©zyBooks 04/06/23 14:24 1369565 Gayoung Kim LEHIGHCSE017Spring2023
ArrayListAppend(numList, 51)		
ArrayListAppend(numList, 41)		
ArrayListAppend(numList, 49)		

1

2

3

4

Check

Next

©zyBooks 04/06/23 14:24 1369565

Gayoung Kim

LEHIGHCSE017Spring2023

©zyBooks 04/06/23 14:24 1369565

Gayoung Kim

LEHIGHCSE017Spring2023