5.1 Doubly-linked lists

Doubly-linked list

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

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

PARTICIPATION ACTIVITY 5.1.1: Doubly-linked list data structure.

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

○ one

○ two

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

○ head

○ tail

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

○ 4

○ 5

4) Given a doubly-linked list with

nodes 8, 12, 7, 3, node 7's next pointer points to node _____.

○ 12

○ 3

Appending a node to a doubly-linked list

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

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

PARTICIPATION ACTIVITY 5.1.2: Doubly-linked list: Appending a node.

Animation content:

`undefined`

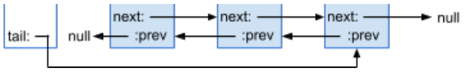
Animation captions:

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

PARTICIPATION ACTIVITY 5.1.3: Doubly-linked list data structure.

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

charList:
head:
data: Q
data: R
data: N

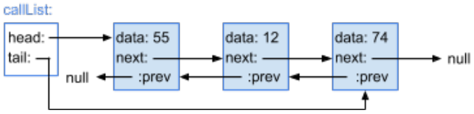


○ after node Q

○ before node N

○ after node N

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

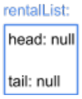


○ `list⇢head = newNode`

○ `list⇢tail⇢next = newNode`

○ `newNode⇢next = list⇢tail`

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



○ `list⇢head = newNode`

○ `list⇢tail⇢next = newNode`

○ `newNode⇢prev = list⇢tail`

Prepending a node to a doubly-linked list

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

- *Prepend to empty list:* If the list's head pointer is null (empty), the algorithm points the list's head and tail pointers to the new node.

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

PARTICIPATION ACTIVITY | 5.1.4: Doubly-linked list: Prepending a node.

Animation content:

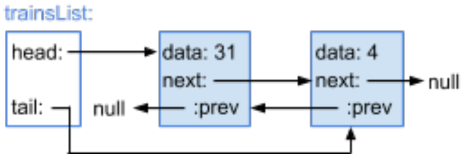
undefined

Animation captions:

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

PARTICIPATION ACTIVITY | 5.1.5: Prepending a node in a doubly-linked list.

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



○ 4

○ 29

○ 31

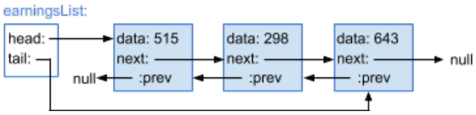
2) ListPrepend(shoppingList, node Milk) updates the list's tail pointer.

head: null
tail: null

○ True

○ False

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



○ `list⇢tail = newNode`

○ `newNode⇢next = list⇢head`

○ `newNode⇢next = list⇢tail`

CHALLENGE ACTIVITY 5.1.1: Doubly-linked lists.

Start

numList = new List
ListAppend(numList, node 35)
ListAppend(numList, node 51)

numList is now: Ex: 1, 2, 3

Which node has a null previous pointer? Ex: 5

Which node has a null next pointer? Ex: 5

1	2	3	4	5

Check Next

5.2 Doubly-linked lists: Insert

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

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

PARTICIPATION ACTIVITY 5.2.1: Doubly-linked list: Inserting nodes.

Animation content:

`undefined`

Animation captions:

1. Inserting a first node into the list points the list's head and tail pointers to the new node.
2. Inserting after the list's tail node points the tail node's next pointer to the new node.

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

PARTICIPATION ACTIVITY 5.2.2: Inserting nodes in a doubly-linked list.

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

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

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

Position 0 (list's head node)

Position 1

Position 2

Position 3

Position 4 (list's tail node)

Reset

CHALLENGE ACTIVITY 5.2.1: Doubly-linked lists: Insert.

Start

numList: 46, 69
ListInsertAfter(numList, node 69, node 27)
ListInsertAfter(numList, node 46, node 33)
ListInsertAfter(numList, node 27, node 60)
ListInsertAfter(numList, node 27, node 23)

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

What node does node 33's next pointer point to? node 23

What node does node 33's previous pointer point to? node 23

1 2 3 4

Check Next

5.3 Doubly-linked lists: Remove

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

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

algorithm points the predecessor's next pointer to the successor node.
- *Removing list's head node:* If curNode points to the list's head node, the algorithm points the list's head pointer to the successor node.
- *Removing list's tail node:* If curNode points to the list's tail node, the algorithm points the list's tail pointer to the predecessor node.

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

PARTICIPATION ACTIVITY 5.3.1: Doubly-linked list: Node removal.

Animation content:

`undefined`

Animation captions:

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

PARTICIPATION ACTIVITY | 5.3.2: Deleting nodes from a doubly-linked list.

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

1) numsList: 71, 29, 54

ListRemove(numsList, node 29)

`numsList:`

Check Show answer

2) numsList: 2, 8, 1

ListRemove(numsList, list tail)

`numsList:`

Check Show answer

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

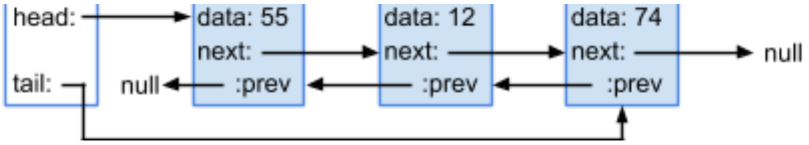
ListRemove(numsList, node 82)
ListRemove(numsList, node 357)
ListRemove(numsList, node 66)

`numsList:`

Check Show answer

PARTICIPATION ACTIVITY | 5.3.3: ListRemove algorithm execution: Intermediate node.

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



1) sucNode⇢prev = predNode

○ Yes

○ No

2) predNode⇢next = sucNode

○ Yes

○ No

3) list⇢head = sucNode

○ Yes

○ No

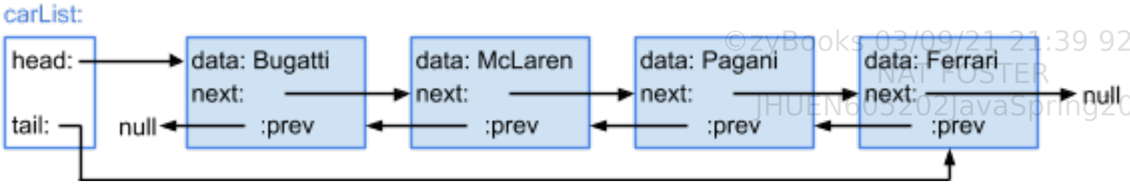
4) list⇢tail = predNode

○ Yes

○ No

PARTICIPATION ACTIVITY 5.3.4: ListRemove algorithm execution: List head node.

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



1) sucNode⇢prev = predNode

○ Yes

○ No

2) predNode⇢next = sucNode

○ Yes

○ No

3) list⇢head = sucNode

○ Yes

○ No

4) list⇢tail = predNode

○ Yes

○ No

CHALLENGE ACTIVITY 5.3.1: Doubly-linked lists: Remove.

Start

Given list: 2, 4, 5, 3, 9
What list results from the following operations?

ListRemoveAfter(list, node 3)
ListRemoveAfter(list, null)
ListRemoveAfter(list, node 4)

List items in order, from head to tail.

Ex: 25, 42, 12

1	2	3	4	5

Check Next

5.4 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 | 5.4.1: 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 | 5.4.2: Singly linked lists with a dummy node.

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

○ True

○ False

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

○ True

○ False

PARTICIPATION ACTIVITY 5.4.3: Condition for an empty list.

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

○ `myList⇢head == null`

○ `myList⇢tail == null`

○ `myList⇢head == myList⇢tail`

Singly-linked list implementation

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

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

Figure 5.4.1: Singly-linked list with dummy node: append, prepend, insert after, and remove after operations.

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


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


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


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

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

PARTICIPATION ACTIVITY 5.4.4: Singly-linked list with dummy node.

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

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

- ○ `ListRemoveAfter(dataList, null)`
- ○ `ListRemoveAfter(dataList, dataList⇢head)`
- ○ `ListRemoveAfter(dataList, dataList⇢tail)`

2) Which is a requirement of the ListPrepend function?

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

PARTICIPATION ACTIVITY 5.4.5: Singly-linked list with dummy node.

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

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

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

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

2) Suppose the following statement is executed:

```
node19 =
numbersList⇢head⇢next⇢next
```

Which subsequent operations swap nodes 73 and 19?

○ ListPrepend(numbersList, node19)

○ ListInsertAfter(numbersList, numbersList⇢head, node19)

○ ListRemoveAfter(numbersList, numbersList⇢head⇢next)
ListPrepend(numbersList, node19)

Doubly-linked list implementation

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

Figure 5.4.2: Doubly-linked list with dummy node: append, prepend, insert after, and remove operations.

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


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

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

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

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


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

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

   sucNode = curNode⇢next
   predNode = curNode⇢prev

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

PARTICIPATION ACTIVITY 5.4.6: Doubly-linked list with dummy node.

1) `ListPrepend(list, newNode)` is equivalent to `ListInsertAfter(list, list⇢head, newNode)`.

○ True

○ False

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

○ True

○ False

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

○ True

○ False

Dummy head and tail nodes

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

PARTICIPATION ACTIVITY 5.4.7: Doubly-linked list append and prepend with 2 dummy nodes.

Animation content:

undefined

Animation captions:

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

Figure 5.4.3: Doubly-linked list with 2 dummy nodes: insert after and remove operations.

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

   sucNode = curNode⇢next
   newNode⇢next = sucNode
   newNode⇢prev = curNode
   curNode⇢next = newNode
   sucNode⇢prev = newNode
}

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

   sucNode = curNode⇢next
   predNode = curNode⇢prev

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

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

Removing if statements from ListInsertAfter and ListRemove

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

PARTICIPATION ACTIVITY | 5.4.8: Comparing a doubly-linked list with 1 dummy node vs. 2 dummy nodes.

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

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

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

2) list⇢tail may be null in _____.

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

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

○ a list with 1 dummy node

5.5 Circular lists

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

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

PARTICIPATION ACTIVITY 5.5.1: Circular list structure and traversal.

Animation content:

`undefined`

Animation captions:

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

PARTICIPATION ACTIVITY 5.5.2: Circular list concepts.

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

○ True

○ False

2) In a circular doubly-linked list with

at least 2 nodes, where does the head node's previous pointer point to?

○ List head

○ List tail

○ null

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

○ List head

○ List tail

○ null

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

○ True

○ False

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

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

○ True

○ False

5.6 Priority queue abstract data type (ADT)

Priority queue abstract data type

A ***priority queue*** is a queue where each item has a priority, and items with higher priority are closer to the front of the queue than items with lower priority. The priority queue ***enqueue*** operation inserts an item such that the item is closer to the front than all items of lower priority, and closer to the end than all items of equal or higher priority. The priority queue ***dequeue*** operation removes and returns the item at the front of the queue, which has the highest priority.

PARTICIPATION ACTIVITY | 5.6.1: Priority queue enqueue and dequeue.

Animation content:

`undefined`

Animation captions:

1. Enqueueing a single item with priority 7 initializes the priority queue with 1 item.
2. If a lower numerical value indicates higher priority, enqueueing 11 adds the item to the end of the queue.
3. Since 5 < 7, enqueueing 5 puts the item at the priority queue's front.
4. When enqueueing items of equal priority, the first-in-first-out rules apply. The 2nd item with priority 7 comes after the first.
5. Dequeue removes from the front of the queue, which is always the highest priority item.

PARTICIPATION ACTIVITY | 5.6.2: Priority queue enqueue and dequeue.

Assume that lower numbers have higher priority and that a priority queue currently holds items: 54, 71, 86 (front is 54).

1) Where would an item with priority 60 reside after being enqueued?

○ Before 54

○ After 54

○ After 86

2) Where would an additional item with priority 54 reside after being enqueued?

○ Before the first 54

○ After the first 54

○ After 86

3) The dequeue operation would return which item?

○ 54

○ 71

○ 86

Common priority queue operations

In addition to enqueue and dequeue, a priority queue usually supports peeking and length querying. A ***peek*** operation returns the highest priority item, without removing the item from the front of the queue.

Table 5.6.1: Common priority queue ADT operations.

Operation	Description	Example starting with priority queue: 42, 61, 98 (front is 42)
Enqueue(PQueue, x)	Inserts x after all equal or higher priority items	Enqueue(PQueue, 87). PQueue: 42, 61, 87, 98
Dequeue(PQueue)	Returns and removes the item at the front of PQueue	Dequeue(PQueue) returns 42. PQueue: 61, 98
Peek(PQueue)	Returns but does not remove the item at the front of PQueue	Peek(PQueue) returns 42. PQueue: 42, 61, 98
IsEmpty(PQueue)	Returns true if PQueue has no items	IsEmpty(PQueue) returns false.
GetLength(PQueue)	Returns the number of items in PQueue	GetLength(PQueue) returns 3.

PARTICIPATION ACTIVITY 5.6.3: Common priority queue ADT operations.

Assume servicePQueue is a priority queue with contents: 11, 22, 33, 44, 55.

1) What does GetLength(servicePQueue) return?

○ 5

○ 11

○ 55

2) What does Dequeue(servicePQueue) return?

○ 5

○ 11

○ 55

3) After dequeuing an item, what will Peek(servicePQueue) return?

○ 11

○ 22

○ 33

4) After calling Dequeue(servicePQueue) a total of 5 times, what will GetLength(servicePQueue) return?

○ -1

○ 0

○ Undefined

Enqueueing items with priority

A priority queue can be implemented such that each item's priority can be determined from the item itself. Ex: A customer object may contain information about a customer, including the customer's name and a service priority number. In this case, the priority resides within the object.

A priority queue may also be implemented such that all priorities are specified during a call to ***EnqueueWithPriority***: An enqueue operation that includes an argument for the enqueued item's priority.

PARTICIPATION ACTIVITY 5.6.4: Priority queue EnqueueWithPriority operation.

Animation content:

```
undefined
```

Animation captions:

1. Calls to EnqueueWithPriority() enqueue objects A, B, and C into the priority queue with the specified priorities.
2. In this implementation, the objects enqueued into the queue do not have data members representing priority.
3. Priorities specified during each EnqueueWithPriority() call are stored alongside the queue's objects.

PARTICIPATION ACTIVITY 5.6.5: EnqueueWithPriority operation.

1) A priority queue implementation that requires objects to have a data member storing priority would implement the _____ function.

○ Enqueue

○ EnqueueWithPriority

2) A priority queue implementation that does not require objects to have a data member storing priority would implement the _____ function.

○ Enqueue

○ EnqueueWithPriority

Implementing priority queues with heaps

A priority queue is commonly implemented using a heap. A heap will keep the highest priority item in the root node and allow access in O(1) time. Adding and removing items from the queue will operate in worst-case O($logN$) time.

Table 5.6.2: Implementing priority queues with heaps.

Priority queue operation	Heap functionality used to implement operation	Worst-case runtime complexity
Enqueue	Insert	O($logN$)
Dequeue	Remove	O($logN$)
Peek	Return value in root node	O(1)
IsEmpty	Return true if no nodes in heap, false otherwise	O(1)
GetLength	Return number of nodes (expected to be stored in the heap's member data)	O(1)

PARTICIPATION ACTIVITY

5.6.6: Implementing priority queues with heaps.

1) The Dequeue and Peek operations both return the value in the root, and therefore have the same worst-case runtime complexity.

○ True

○ False

2) When implementing a priority queue with a heap, no operation will have a runtime complexity worse than O($logN$).

○ True

○ False

3) If items in a priority queue with a lower numerical value have higher

priority, then a max-heap should be used to implement the priority queue.

○ True

○ False

4) A priority queue is always implemented using a heap.

○ True

○ False

CHALLENGE ACTIVITY 5.6.1: Priority queue abstract data type.

Start

Assume that lower numbers have higher priority and that a priority queue numPQue
items: 30, 78, 80 (front is 30).

Where does Enqueue(numPQueue, 93) add an item?

After 30

Where does Enqueue(numPQueue, 79) add an item?

After 30

Where does Enqueue(numPQueue, 30) add an item?

After 30

1	2	3	4	5

Check Next

5.7 Set abstract data type

This section has been set as optional by your instructor.

Set abstract data type

A ***set*** is a collection of distinct elements. A set ***add*** operation adds an element to the set, provided an equal element doesn't already exist in the set. A set is an unordered collection. Ex: The set with integers 3, 7, and 9 is equivalent to the set with integers 9, 3 and 7.

PARTICIPATION ACTIVITY 5.7.1: Set abstract data type.

Animation content:

`undefined`

Animation captions:

1. Adding 67, 91, and 14 produces a set with 3 elements.
2. Because 91 already exists in the set, adding 91 any number of additional times has no effect.
3. Set 2 is built by adding the same numbers in a different order.
4. Because order does not matter in a set, the 2 sets are equivalent.

PARTICIPATION ACTIVITY 5.7.2: Set abstract data type.

1) Which of the following is not a valid set?

○ { 78, 32, 46, 57, 82 }

○ { 34, 8, 92 }

○ { 78, 28, 91, 28, 15 }

2) How many elements are in a set

that is built by adding the element 28 6 times, then the element 54 9 times?

○ 1

○ 2

○ 15

3) Which 2 sets are equivalent?

○ { 56, 19, 71 } and { 19, 65, 71, 56 }

○ { 88, 54, 81 } and { 81, 88, 54 }

○ { 39, 56, 14, 11 } and { 14, 56, 93, 11 }

Element keys and removal

Set elements may be primitive data values, such as numbers or strings, or objects with numerous data members. When storing objects, set implementations commonly distinguish elements based on an element's ***key value***: A primitive data value that serves as a unique identifier for the element. Ex: An object for a student at a university may store information such as name, phone number, and ID number. No two students will have the same ID number, so the ID number can be used as the student object's key.

Sets are commonly implemented to use keys for all element types. When storing objects, the set retrieves an object's key via an external function or predetermined knowledge of which object property is the key value. When storing primitive data values, each primitive data value's key is itself.

Given a key, a set ***remove*** operation removes the element with the specified key from the set.

PARTICIPATION ACTIVITY | 5.7.3: Element keys and removal.

Animation content:

```
undefined
```

Animation captions:

1. Different students at the same university may have the same name or phone number, but each student has a unique ID number.
2. A set for the course roster uses the student ID as the key value, since the exact same student cannot enroll twice in the same course.
3. The call to remove Student C provides only the student ID.

PARTICIPATION ACTIVITY 5.7.4: Element keys and removal.

Refer to the example in the animation above.

1) If the student objects contained a field for GPA, then GPA could be used as the key value instead of student ID.

○ True

○ False

2) `SetRemove(courseRosterSet, "Student D")` would remove Student D from the set.

○ True

○ False

3) SetRemove will not operate properly on an empty set.

○ True

○ False

Searching and subsets

Given a key, a set ***search*** operation returns the set element with the specified key, or null if no such element exists. The search operation can be used to implement a subset test. A set X is a ***subset*** of set Y only if every element of X is also an element of Y.

PARTICIPATION ACTIVITY | 5.7.5: SetIsSubset algorithm.

Animation content:

`undefined`

Animation captions:

1. To test if set2 is a subset of set1, each element of set 2 is searched for in set1. Elements 19, 22, and 26 are found in set1.
2. Element 34 is in set2 but not set1, so set2 is not a subset of set1.
3. The first element in set3, 88, is not in set1, so set3 is not a subset of set1.
4. All elements of set4 are in set1, so set4 is a subset of set1.
5. No other set is a subset of another.
6. But each set is always a subset of itself.

PARTICIPATION ACTIVITY | 5.7.6: Searching and subsets.

1) Every set is a subset of itself.

○ True

○ False

2) For X to be a subset of Y, the number of elements in Y must be greater than or equal to the number of elements in X.

○ True

○ False

3) The loop in SetIsSubset always performs N iterations, where N is the number of elements in subsetCandidate.

○ True

CHALLENGE ACTIVITY | 5.7.1: Set abstract data type.

5.8 Set operations

This section has been set as optional by your instructor.

Union, intersection, and difference

The ***union*** of sets X and Y, denoted as X ∪ Y, is a set that contains every element from X, every element from Y, and no additional elements. Ex: { 54, 19, 75 } ∪ { 75, 12 } = { 12, 19, 54, 75 }.

The ***intersection*** of sets X and Y, denoted as X ∩ Y, is a set that contains every element that is in both X and Y, and no additional elements. Ex: { 54, 19, 75 } ∩ { 75, 12 } = { 75 }.

The ***difference*** of sets X and Y, denoted as X \ Y, is a set that contains every element that is in X but not in Y, and no additional elements. Ex: { 54, 19, 75 } \ { 75, 12 } = { 54, 19 }.

The union and intersection operations are commutative, so X ∪ Y = Y ∪ X and X ∩ Y = Y ∩ X. The difference operation is not commutative.

PARTICIPATION ACTIVITY | 5.8.1: Set union, intersection, and difference.

Animation content:

`undefined`

Animation captions:

1. The union operation begins by adding all elements from set1.
2. Each element from set2 is added. Adding elements 82 and 93 has no effect, since 82 and 93 already exist in the result set.

3. The intersection operation iterates through each element in set1. Each element that is also in set2 is added to the result.
4. The difference of set1 and set2, denoted set1 \ set2, iterates through all elements in set1. Only elements 61 and 76 are added to the result, since these elements are not in set2.
5. Set difference is not commutative. SetDifference(set2, set1) produces a result containing only 23 and 46, since those elements are in set2 but not in set1.

PARTICIPATION ACTIVITY 5.8.2: Union, intersection, and difference.

1) How many elements are in the set { 83, 5 } ∪ { 9, 77, 83 }?

○ 2

○ 4

○ 5

2) How many elements are in the set { 83, 5 } ∩ { 9, 77, 83 }?

○ 1

○ 2

○ 3

3) { 83, 5 } \ { 9, 77, 83 } = ?

○ { 83 }

○ { 5 }

○ { 83, 5 }

4) { 9, 77, 83 } \ { 83, 5 } = ?

○ { 9, 77 }

○ { 9, 77, 83 }

○ { 5 }

5) Which set operation is not commutative?

○ Union

○ Intersection

○ Difference

6) When X and Y do not have any elements in common, which is always true?

○ $X \cup Y = X \cap Y$

○ $X \cap Y = X \setminus Y$

○ $X \setminus Y = X$

7) Which is true for any set X?

○ $X \cup X = X \cap X$

○ $X \cup X = X \setminus X$

○ $X \setminus X = X \cap X$

Filter and map

A ***filter*** operation on set X produces a subset containing only elements from X that satisfy a particular condition. The condition for filtering is commonly represented by a ***filter predicate***: A function that takes an element as an argument and returns a Boolean value indicating whether or not that element will be in the filtered subset.

A ***map*** operation on set X produces a new set by applying some function F to each element. Ex: If X = {18, 44, 38, 6} and F is a function that divides a value by 2, then SetMap(X, F) = { 9, 22, 19, 3 }.

PARTICIPATION ACTIVITY 5.8.3: SetFilter and SetMap algorithms.

Animation content:

```
undefined
```

Animation captions:

1. SetFilter is called with the EvenPredicate function passed as the second argument.
2. SetFilter calls EvenPredicate for each element. EvenPredicate returns true for each

even element, and false for each odd element.

3. Every element for which the predicate returned true is added to the result, producing the set of even numbers from set1.
4. SetFilter(set1, Above90Predicate) produces the set with all elements from set1 that are greater than 90.
5. SetMap is called with the OnesDigit function passed as the first argument. Like SetFilter, SetMap calls the function for each element.
6. The returned value from each OnesDigit call is added to the result set, producing the set of distinct ones digit values.
7. SetMap(set1, StringifyElement) produces a set of strings from a set of numbers.

PARTICIPATION ACTIVITY

5.8.4: Using SetFilter with a set of strings.

Suppose stringSet = { "zyBooks", "Computer science", "Data structures", "set", "filter", "map" }. Filter predicates are defined below. Match each SetFilter call to the resulting set.

```
StartsWithCapital(string) {
   if (string starts with capital letter) {
      return true
   }
   else {
      return false
   }
}

Has6OrFewerCharacters(string) {
   if (length of string <= 6) {
      return true
   }
   else {
      return false
   }
}

EndsInS(string) {
   if (string ends in "S" or "s") {
      return true
   }
   else {
      return false
   }
}
```

SetFilter(stringSet, Has6OrFewerCharacters)

SetFilter(stringSet, StartsWithCapital) **SetFilter(stringSet, EndsInS)**

{ "zyBooks", "Data structures" }

{ "Computer science", "Data structures" }

{ "set", "filter", "map" }

Reset

PARTICIPATION ACTIVITY | 5.8.5: Using SetMap with a set of numbers.

Suppose numbersSet = { 6.5, 4.2, 7.3, 9.0, 8.7 }. Map functions are defined below. Match each SetMap call to the resulting set.

```
MultiplyBy10(number) {
   return number * 10.0
}

Floor(number) {
   return floor(number)
}

Round(number) {
   return round(number)
}
```

SetMap(numbersSet, Floor) **SetMap(numbersSet, Round)**

SetMap(numbersSet, MultiplyBy10)

{ 65.0, 42.0, 73.0, 90.0, 87.0 }

{ 7.0, 4.0, 9.0 }

{ 6.0, 4.0, 7.0, 9.0, 8.0 }

Reset

PARTICIPATION ACTIVITY | 5.8.6: SetFilter and SetMap algorithm concepts.

1) A filter predicate must return true for elements that are to be added to the resulting set, and false for elements that are not to be added.

○ True

○ False

2) Calling SetFilter on set X always produces a set with the same number of elements as X.

○ True

○ False

3) Calling SetMap on set X always produces a set with the same number of elements as X.

○ True

○ False

4) Both SetFilter and SetMap will call the function passed as the second argument for every element in the set.

○ True

○ False

CHALLENGE ACTIVITY 5.8.1: Set operations.

5.9 Static and dynamic set operations

This section has been set as optional by your instructor.

A ***dynamic set*** is a set that can change after being constructed. A ***static set*** is a set that doesn't change after being constructed. A collection of elements is commonly provided during construction of a static set, each of which is added to the set. Ex: A static set constructed from the list of integers (19, 67, 77, 67, 59, 19) would be { 19, 67, 77, 59 }.

Static sets support most set operations by returning a new set representing the operation's result. The table below summarizes the common operations for static and dynamic sets.

Table 5.9.1: Static and dynamic set operations.

Operation	Dynamic set support?	Static set support?
Construction from a collection of values	Yes	Yes
Count number of elements	Yes	Yes
Search	Yes	Yes
Add element	Yes	No
Remove element	Yes	No
Union (returns new set)	Yes	Yes
Intersection (returns new set)	Yes	Yes
Difference (returns new set)	Yes	Yes
Filter (returns new set)	Yes	Yes
Map (returns new set)	Yes	Yes

PARTICIPATION ACTIVITY 5.9.1: Static and dynamic set operations.

1) Static sets do not support union or intersection, since these operations require changing the set.

○ True

○ False

2) A static set constructed from the list of integers (20, 12, 87, 12) would be { 20, 12, 87, 12 }.

○ True

○ False

3) Suppose a dynamic set has N elements. Adding any element X and then removing element X will always result in the set still having N elements.

○ True

○ False

PARTICIPATION ACTIVITY 5.9.2: Choosing static or dynamic sets for real-world datasets.

For each real-world dataset, select whether a program should use a static or dynamic set.

1) Periodic table of elements

○ Static

○ Dynamic

2) Collection of names of all countries on the planet

○ Static

○ Dynamic

3) List of contacts for a user

○ Static

○ Dynamic