8.1 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 8.1.1: 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 8.1.2: Stack ADT: Push and pop operations.

1) Given numStack: 7, 5 (top is 7). Type the stack after the following push operation. Type the stack as: 1, 2, 3

   Push(numStack, 8)

   Check Show answer

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

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

Check Show answer

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

Pop(numStack)

Check Show answer

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

Pop(numStack)

Check Show answer

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

Pop(numStack)
Pop(numStack)

Check Show answer

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

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

Check Show answer

Common stack ADT operations

Table 8.1.1: Common stack ADT operations.

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

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

PARTICIPATION ACTIVITY 8.1.3: Common stack ADT operations.

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

- 4
- 70

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

- 2, 9, 4
- 9, 4

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

- 2, 9, 4
- 9, 4

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

- Peek
- IsEmpty

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

- Pop
- Push

CHALLENGE ACTIVITY 8.1.1: Stack ADT.

502696.2096894.qx3zqy7

Start

Given numStack: 88, 86, 33 (top is 88)

What is the stack after the operations?

Push(numStack, 38)
Pop(numStack)
Pop(numStack)

Ex: 1, 2, 3

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

Ex: 5

Check Next

8.2 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 8.2.1: 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.

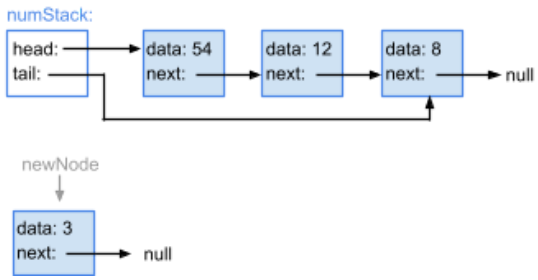
PARTICIPATION ACTIVITY 8.2.2: Stack push and pop operations with a linked list.

Assume the stack is implemented using a linked list.

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

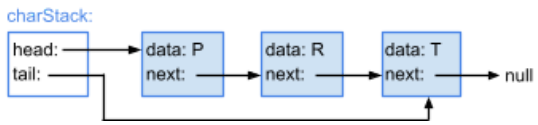
- newNode
- null
- Unknown

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



- Node 54
- Node 12
- null

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



- Node P
- Node R
- Node T

4) StackPop returns list's head node.

- True
- False

CHALLENGE ACTIVITY 8.2.1: Stacks using linked lists.

502696.2096894.qx3zqy7

Start

Given an empty stack numStack,

What does the list head pointer point to? If the pointer is null, enter null.

Ex: 5 or null

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

```
StackPush(numStack, 34)
StackPush(numStack, 31)
StackPush(numStack, 54)
```


Ex: 5 or null

1 2 3

Check Next

8.3 Python: 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 8.3.1: 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 8.3.2: Unbounded, array-based stack.

Animation content:

undefined

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 8.3.3: 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
- grow only until all distinct 32-bit integer values are pushed
- grow only until the resize operation fails to allocate memory

Bounded stack

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 8.3.4: 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(allocationSize * 2, maxLength) = 3 is the new allocation size.
8. The next push fails because length equals maxLength.

PARTICIPATION ACTIVITY 8.3.5: Bounded stack - general implementation.

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

PARTICIPATION ACTIVITY 8.3.6: 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 A and B
   - ○ Neither A nor B

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

- Implementation A only
- Implementation B only
- Neither A nor B

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

- Implementation A only
- Implementation B only
- Both A and B
- Neither A nor B

Implementing a stack with a Python list

Operations presented above work under the assumption that arrays have a fixed size, and so reallocation is needed for some push operations. Fixed size arrays are common in several popular programming languages, and so the approach described above works in each.

The Stack class implementation that follows uses a Python list. A Python list does not have a fixed size, so implementing reallocation is not needed. Therefore, implementation is simpler than an implementation with a fixed size array.

The Stack class has only one attribute: a list named stack_list that stores the stack's data. Stack instance methods push() and pop() implement stack functionality using the list's append() and pop() methods, respectively.

Figure 8.3.1: Stack class using a Python list.

```
class Stack:
    def __init__(self):
        self.stack_list = []

    def pop(self):
        return
self.stack_list.pop()

    def push(self, item):

self.stack_list.append(item)
```

PARTICIPATION ACTIVITY 8.3.7: Implementing a stack with a Python list.

Refer to the Stack class above.

1) The Stack class implements _____ stack.

- a bounded
- an unbounded
- both a bounded and unbounded

2) The Python list class's pop() method has an optional parameter for the index of the list item to remove. Should the Stack class's pop() method have the same optional parameter?

- Yes. The optional parameter makes the implementation more flexible.
- No, because then pop() couldn't be called without parameters.
- No, because the rules of the stack could be violated.

Implementing the Stack class to support bounded or unbounded functionality

The Stack class below can be used as an unbounded stack or a bounded stack. If the parameter passed to the constructor is negative, the stack is unbounded. If the parameter is nonnegative, the stack is bounded.

Figure 8.3.2: Python Stack class that supports bounded or unbounded functionality.

```
class Stack:
    # Initializes the stack. If the optional_max_length argument is omitted
or
    # negative, the stack is unbounded. If optional_max_length is non-
negative,
    # the stack is bounded.
    def __init__(self, optional_max_length = -1):
        self.stack_list = []
        self.max_length = optional_max_length
    #
    # Pops and returns the stack's top item.
    def pop(self):
        return self.stack_list.pop()
    #
    # Pushes an item, provided the push doesn't exceed bounds. Does nothing
    # otherwise. Returns True if the push occurred, False otherwise.
    def push(self, item):
        # If at max length, return false
        if len(self.stack_list) == self.max_length:
            return False
        #
        # If unbounded, or bounded and not yet at max length, then push
        self.stack_list.append(item)
        return True
```


PARTICIPATION ACTIVITY 8.3.8: Python Stack class.

1) Can the constructor's parameter be omitted?

- Yes. Doing so makes the stack bounded.
- Yes. Doing so makes the stack unbounded.
- No.

2) The push() method starts with a check against `max_length`. Must push() be altered to support an unbounded stack?

- Yes. The if statement must change to:
```
if self.max_length > 0 and
   len(self.stack_list) ==
self.max_length:
```
- Yes. The if statement must change to:
```
if len(self.stack_list) >
self.max_length:
```
- No.

zyDE 8.3.1: Python list-based stack implementation.

The following demo constructs a bounded stack with a limit of 4 items and an unbounde stack. Then eight calls to each stack's push() method occur, followed by two pop() calls, t four more push() calls. Then each stack's contents are displayed.

main.py Load default templat

```
class Stack:
    # Initializes the stack. If the optional_max_length argument is c
    # negative, the stack is unbounded. If optional_max_length is nor
    # the stack is bounded.
    def __init__(self, optional_max_length = -1):
        self.stack_list = []
        self.max_length = optional_max_length
    #
    # Gets the length of the stack
    def __len__(self):
        return len(self.stack_list)
    #
    # Pops and returns the stack's top item.
    def pop(self):
        return self.stack_list.pop()
```


Run

8.4 Queue abstract data type (ADT)

Queue abstract data type

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

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

PARTICIPATION ACTIVITY 8.4.1: 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 8.4.2: Queue ADT.

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

Enqueue(numQueue, 4)

Check Show answer

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

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

Check Show answer

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

Dequeue(numQueue)

Check Show answer

4) Given numQueue: 4, 8, 7, 1, 3
What is returned by the second dequeue operation?

Dequeue(numQueue)
Dequeue(numQueue)

Check Show answer

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

Dequeue(numQueue)

Check Show answer

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

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

Check Show answer

Common queue ADT operations

Table 8.4.1: 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.

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

PARTICIPATION ACTIVITY 8.4.3: 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 8.4.1: Queue ADT.

502696.2096894.qx3zqy7

Start

©zyBooks 11/20/23 11:05 1048447
Eric Quezada
UTEPCS2302ValeraFall2023

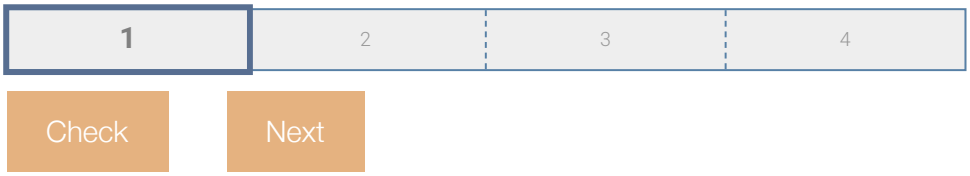
Given numQueue: 33, 82
What are the queue's contents after the following operations?

Dequeue(numQueue)
Enqueue(numQueue, 51)

Ex: 1, 2, 3

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

Ex: 8



Check Next

8.5 Queues using linked lists

A queue is often implemented using a linked list, with the list's head node representing the queue's front, and the list's tail node representing the queue's end. Enqueueing an item is performed by creating a new list node, assigning the node's data with the item, and appending the node to the list. Dequeuing is performed by assigning a local variable with the head node's data, removing the head node from the list, and returning the local variable.

PARTICIPATION ACTIVITY 8.5.1: Queue implemented using a linked list.

Animation content:

undefined

Animation captions:

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

PARTICIPATION ACTIVITY 8.5.2: Queue push and pop operations with a linked list.

Assume the queue is implemented using a linked list.

1) If the head pointer is null, the queue _____.

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

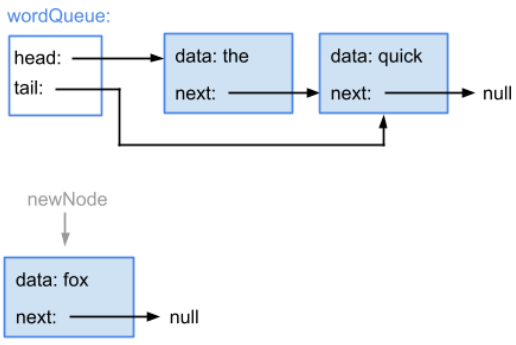
2) For the operation QueueDequeue(queue), what is the second parameter passed to ListRemoveAfter?

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

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

- head
- tail

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



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

CHALLENGE ACTIVITY 8.5.1: Queues using linked lists.

502696.2096894.qx3zqy7

Start

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

Ex: 5 or null

What does the list tail pointer point to?

After the following operations:

QueueEnqueue(numQueue, 90)
QueueEnqueue(numQueue, 47)
QueueEnqueue(numQueue, 41)
QueueEnqueue(numQueue, 12)
QueueDequeue(numQueue)

What does the list head pointer point to?

What does the list tail pointer point to?

1 2

Check Next

8.6 Python: Array-based queues

Array-based queue storage

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

- length: an integer for the queue's length.
- front_index: an integer for the queue's front item index.

The queue's content starts at `array[front_index]` 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.

PARTICIPATION ACTIVITY 8.6.1: Array-based queue storage.

Animation content:

undefined

Animation captions:

1. Queue content starts at front_index. When front_index is 0 and length equals array allocation size, the queue's content matches the array's content: 78, 64, 16, 95.
2. If front_index 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. The allocation size of the queue's array is 4. But the queue's length is 3, 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.

PARTICIPATION ACTIVITY 8.6.2: Array-based queue storage.

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

1) What is the queue's front item if front_index is 3 and length is 2?
   - ○ 67
   - ○ 44
   - ○ 38

2) What is the queue's back item if front_index is 0 and length is 3?
   - ○ 67
   - ○ 44
   - ○ 38

3) What is the queue's content if length is 4 and front_index is 1?
   - ○ (front) 67, 19, 44, 38 (back)
   - ○ (front) 19, 44, 38 (back)
   - ○ (front) 19, 44, 38, 67 (back)

Bounded vs. unbounded queue

A ***bounded queue*** is a queue with a length that does not exceed a specified maximum value. An additional variable, `max_length`, is needed. `max_length` 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 8.6.3: Bounded vs. unbounded queue.

Animation content:

Step 1: A table is shown with four columns. Only column headers are shown, and from left to right are: "Operations", "Result for unbounded queue", "Result for bounded queue with maxLength=2", "Result for bounded queue with maxLength=4".

Step 2: Row 1, below the header row, appears. Leftmost cell lists two operations: "Enqueue 51" and "Enqueue 88". Second to leftmost cell: (front) 51, 88 (back). Second to rightmost cell: (front) 51, 88 (back). Rightmost cell: (front) 51, 88 (back). Each cell in the row except the leftmost has a light green background, indication that operation was successful for each of the three queues.

Step 3: Row 2 appears. Leftmost cell lists one operation: "Enqueue 73". Second to leftmost cell: (front) 51, 88, 73 (back). Second to rightmost cell: (front) 51, 88 (back). Rightmost cell: (front) 51, 88, 73 (back). Second-to-leftmost and rightmost cells have a light green background. Second-to-rightmost cell has a red background, because the operation failed due to a full queue.

Step 4: Row 3 appears. Leftmost cell lists one operation: "Enqueue 47". Second to leftmost cell: (front) 51, 88, 73, 47 (back). Second to rightmost cell: (front) 51, 88 (back). Rightmost cell: (front) 51, 88, 73, 47 (back). Second-to-leftmost and rightmost cells have a light green background. Second-to-rightmost cell has a red background, because the operation failed again due to a full queue.

Step 5: Row 4 appears. Leftmost cell lists one operation: "Enqueue 24". Second to leftmost cell: (front) 51, 88, 73, 47, 24 (back). Second to rightmost cell: (front) 51, 88 (back). Rightmost cell: (front) 51, 88, 73, 47 (back). Second-to-leftmost cell has a light green background. Second-to-rightmost and rightmost cells have a red background, because the operation failed due to a full queue.

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 8.6.4: 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 `max_length` variable is needed _____.
   - only for the bounded queue implementation
   - only for the unbounded queue implementation
   - for both the bounded and unbounded queue implementations

Flexible implementation and resize operation

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

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

The Python implementation uses a list named `queue_list` to store queue items. `len(queue_list)` is the queue's allocation size. The list is used much like a fixed size array. So the resize operation creates a larger list, commonly double the length of the current list. If `max_length` is nonnegative, the new list's

length is the minimum of `max_length` and double the current list's length. Existing list entries are copied in such a way that the front index is reset to 0.

PARTICIPATION ACTIVITY 8.6.5: Array-based queue resize operation.

Animation content:

undefined

Animation captions:

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

PARTICIPATION ACTIVITY 8.6.6: Flexible implementation and resize operation.

Consider queueA with variables:

```
queue_list: [73, 91, 87, 62]
front_index: 2
length: 4
max_length: -1
```

1) queueA is _____.

- bounded and full
- bounded and not full
- unbounded

2) After executing `queueA.resize()`, queueA's allocation size is _____.

- -1
- 4
- 8

3) After executing `queueA.resize()`, queueA's first four array elements are _____.

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

4) Can one of queueA's initial variables be changed such that executing `queueA.resize()` makes queueA's allocation size 6?

- Yes, if `max_length` were initially 6
- Yes, if `len(queueA.queue_list)` were initially 6
- No

Enqueue and dequeue operations

An enqueue operation:

1. Compares `self.length` and `self.max_length`. If equal, the queue is full and so no change occurs and False is returned.
2. Compares `self.length` and `len(self.queue_list)`. If equal, a resize operation occurs.
3. Computes the enqueued item's index as `(self.front_index + self.length) % len(self.queue_list)` and assigns to queue_list at that index. Ex: Enqueueing 42 into a queue with front index 2, length 3, and queue_list length 8 assigns queue_list at index (2 + 3) % 8 = 5 with 42.
4. Increments the `length` attribute and then returns True.

A dequeue operation:

1. Makes a copy of the list item at `front_index`.
2. Decrements `length`.
3. Increments `front_index`, resetting to 0 if the incremented value equals the allocation size.
4. Returns the list item from step 1.

PARTICIPATION ACTIVITY 8.6.7: Array-based queue enqueue and dequeue operations.

Animation content:

undefined

Animation captions:

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

PARTICIPATION ACTIVITY 8.6.8: Enqueue and dequeue operations.

1) _____ may resize the queue.

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

2) `(self.front_index + self.length) % len(self.queue_list)` 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) enqueue() returns False if _____.

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

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 ***`max_length`*** *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).*

zyDE 8.6.1: Python array-based queue implementation.

The following demo constructs a bounded queue with a limit of 4 items and an unbounde queue. Then eight calls to each queue's enqueue() method occur, followed by two dequeu calls, then four more enqueue() calls. Then each queue's contents are displayed.

Current file: **ArrayBasedQueues.py** Load default templat

```
1  from ArrayQueue import ArrayQueue
2
3  # Make two queues, one bounded to 4 items and the other bounded
4  boundedQueue = ArrayQueue(4)
5  unboundedQueue = ArrayQueue()
6
7  # Enqueue 8 items in each
8  print("Enqueueing values 1 through 8 to each queue")
9  for i in range(1, 9):
10     boundedQueue.enqueue(i)
11     unboundedQueue.enqueue(i)
12
13 # Dequeue two items from each queue
14 print("Dequeuing twice")
15 for i in range(2):
```


Run

8.7 Python: Stacks and queues using linked lists

Implementing a stack in Python

A stack can be implemented in Python using a class with a single LinkedList data member. The class has two methods, push() and pop(). push() adds a node to the top of the stack's list by calling LinkedList's prepend() method. pop() removes the head of the stack's list by calling the LinkedList's remove_after() method and then returns the removed node's data.

Figure 8.7.1: Stack implementation using the LinkedList() class.

```
from Node import Node
from LinkedList import LinkedList

class Stack:
    def __init__(self):
        self.list = LinkedList()

    def push(self, new_item):
        # Create a new node to hold the item
        new_node = Node(new_item)

        # Insert the node as the list head (top of stack)
        self.list.prepend(new_node)

    def pop(self):
        # Copy data from list's head node (stack's top node)
        popped_item = self.list.head.data

        # Remove list head
        self.list.remove_after(None)

        # Return the popped item
        return popped_item


num_stack = Stack()
num_stack.push(45)
num_stack.push(56)
num_stack.push(11)

# Output stack
print('Stack after push:', end=' ')
node = num_stack.list.head
while node != None:
   print(node.data, end=' ')
   node = node.next
print()

# Pop 11
popped_item = num_stack.pop()
print('Popped:', popped_item)

# Output final stack
print('Stack after pop:', end=' ')
node = num_stack.list.head
while node != None:
   print(node.data, end=' ')
   node = node.next
print('\n')
```


©zyBooks 11/20/23 11:05 1048447
Eric Quezada
UTEPCS2302ValeraFall2023

```
Stack after push: 11 56 45
Popped: 11
Stack after pop: 56 45
```

PARTICIPATION ACTIVITY 8.7.1: Python Stack class.

1) The Stack class has both a LinkedList and Node data member.

   - True
   - False

2) The Stack class' push() method uses the LinkedList append() method to place elements on a stack.

   - True
   - False

3) The pop() method removes the head of the stack's linked list.

   - True
   - False

Implementing a queue in Python

A queue can also be implemented in Python using a class with a single LinkedList data member and class methods enqueue() and dequeue(). enqueue() adds a node to the end of the queue's list by calling LinkedList's append() method. The dequeue() method removes the queue's head node and returns the removed node's data.

Figure 8.7.2: Queue implementation using the LinkedList() class.

```
from Node import Node
from LinkedList import LinkedList

class Queue:
    def __init__(self):
        self.list = LinkedList()

    def enqueue(self, new_item):
        # Create a new node to hold the item
        new_node = Node(new_item)

        # Insert as list tail (end of queue)
        self.list.append(new_node)

    def dequeue(self):
        # Copy data from list's head node (queue's front node)
        dequeued_item = self.list.head.data

        # Remove list head
        self.list.remove_after(None)

        # Return the dequeued item
        return dequeued_item


num_queue = Queue()
num_queue.enqueue(17)
num_queue.enqueue(24)
num_queue.enqueue(18)

# Output queue
print('Queue after enqueue:', end=' ')
node = num_queue.list.head
while node != None:
   print(node.data, end=' ')
   node = node.next
print()

# Dequeue 17
dequeued_item = num_queue.dequeue()
print('Dequeued:', dequeued_item)

# Output final queue
print('Queue after dequeue:', end=' ')
node = num_queue.list.head
while node != None:
   print(node.data, end=' ')
   node = node.next
print()
```


©zyBooks 11/20/23 11:05 1048447
Eric Quezada
UTEPCS2302ValeraFall2023

```
Queue after enqueue: 17 24 18
Dequeued: 17
Queue after dequeue: 24 18
```

PARTICIPATION ACTIVITY 8.7.2: Python Queue class.

1) The Queue class' enqueue() method uses the LinkedList append() method to insert elements in a queue.

- True
- False

2) Queue's dequeue() method operates differently than Stack's pop() method.

- True
- False

zyDE 8.7.1: Stack and Queue data structures.

The Stack and Queue classes have been placed into separate files and are imported into main.py. Node.py and LinkedList.py are imported into Stack.py and Queue.py.

Current file: main.py Load default templat

```
from Stack import Stack
from Queue import Queue

# Stack operations
num_stack = Stack()
num_stack.push(45)
num_stack.push(56)
num_stack.push(11)

# Output stack
print('Stack after push:', end=' ')
node = num_stack.list.head
while node != None:
   print(node.data, end=' ')
   node = node.next
```

Run

8.8 Deque abstract data type (ADT)

Deque abstract data type

A ***deque*** (pronounced "deck" and short for double-ended queue) is an ADT in which items can be inserted and removed at both the front and back. The deque push-front operation inserts an item at the front of the deque, and the push-back operation inserts at the back of the deque. The pop-front operation removes and returns the item at the front of the deque, and the pop-back operation removes and returns the item at the back of the deque. Ex: After the operations "push-back 7", "push-front 14", "push-front 9", and "push-back 5", "pop-back" returns 5. A subsequent "pop-front" returns 9. A deque can be implemented using a linked list or an array.

PARTICIPATION ACTIVITY 8.8.1: Deque ADT.

Animation captions:

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

PARTICIPATION ACTIVITY 8.8.2: Deque ADT.

Determine the deque contents after the following operations.

If unable to drag and drop, refresh the page.

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

Table 8.8.1: Common deque ADT operations.

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

PARTICIPATION ACTIVITY 8.8.3: 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

CHALLENGE ACTIVITY 8.8.1: Deque ADT.

502696.2096894.qx3zqy7

Start

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

```
PushFront(numDeque, 92)
PushBack(numDeque, 12)
```

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?

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

1 2 3

Check Next

8.9 LAB: Grocery list editor with undo stack

This section's content is not available for print.