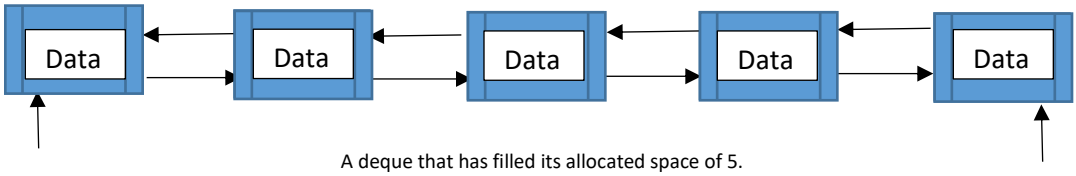
Andrew Taylor

Atayl136

Homework 5: More on Lists

1. A deque (pronounced deck) is an ordered set of items from which items may be deleted at either end and into which items may be inserted at either end. Call the two ends left and right. This is an access-restricted structure since no insertions or deletions can happen other than at the ends. Implement the deque as a doubly-linked list (not circular, no header). Write InsertLeft and DeleteRight.



A deque that has filled its allocated space of 5.

To implement a deque, we will need two pointers, one for left and one for right. Being doubly-linked, each node will have a next pointer and a previous pointer, implemented as entries in a table or array, next to the data. We will insert and delete items at the left and right pointers. Two of the methods that are needed are as follows:

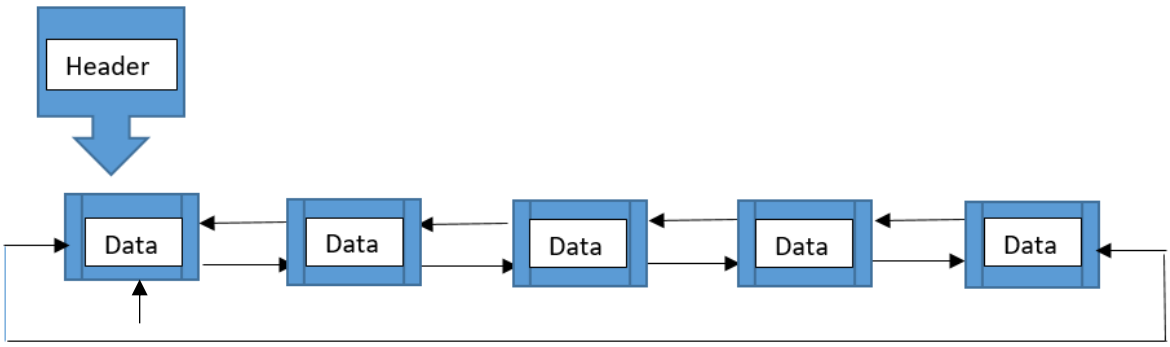
InsertLeft

1) Grab some space for a new node with data and a previous and next pointers.
   First, check if length == allocationsize, then dynamic allocation needed to prevent overflow.
2) check if left == null,
3) then set left and right pointers to new node.
4) If left != null, then newnode.right == left, left.left == newnode, finally set left == newnode.

DeleteRight

1) Check if right != null otherwise there is an underflow error.
2) Store the data in the node pointed by right into a temporary variable.
3) Set right == right.left
4) Check if now right == null, then set left == null because the deque is empty.
5) Otherwise set right.right == null
6) Return the variable and deallocate space

2. Implement a deque from problem 1 as a doubly-linked circular list with a header. Write InsertRight and DeleteLeft.



This circular doubly linked list has node pointers to previous nodes and next nodes but never any null pointers. The last node contains the address of the first node and the first node contains the address of the last node. The variable head contains the address of the first node in the list, and would indicate when one pass through the list has occurred. This list works with one pointer and the head pointer.

InsertRight

1) Initialize a temp variable with the item to insert, this is in the pointer
2) Grab some space for a new node, checking for an overflow (if the allocation is full yet)
3) Stuff the data into the new node with the pointer set pointer == newnode
4) If the head == null, the pointer.next and pointer.prev == head
5) Else set a temp = head, then while temp.next != head:
6) Temp = temp.next, temp.next = pointer, pointer.prev = temp – this joins up the new node
7) Last, head.prev = pointer and pointer.next = head - because it is circular

DeleteLeft

1) Check if head != null otherwise there will be an underflow error
2) Else if head.next == head, they want to delete the list, set head == null and free up space
3) Else temp = head, while temp.next != head:
4) Temp = temp.next, temp.next = head.next – joining up the tail of the list to the head
5) Head.next.prev = temp – retrieving the value to be deleted
6) Head = temp.next – reset the head the circular list is good to go again

3. Write a set of routines for implementing several stacks and queues within a single array. **Hint:** Look at the lecture material on the hybrid implementation.

The hybrid implementation rests on some key principles: order is arbitrary, imposed mostly by the data structures in use, comprised of a data piece and a reference piece to the next item in a single array. The whole structure is managed by a separate list of free space, and random access doesn't give you information about the context of which list information will be in, because you have to access the information sequentially. In this array there is a value, a reference to next, and a list managing free space that contains the number of stacks queues their pointers and free entries locations.

Methods for the queues in the array:

IsFree(Queue)

A pointer, stores an index of the next free memory address

returns the index of the first free entry in the array, -1 if resize needed

QueueBack(Queue)

A pointer stores an index, it starts at -1 … If > or equal = 0

Returns the array index of the back of the queue

updates the back pointer for this queue on the free list, it starts at -1 and -1 when null

else

Returns IsFree because the queue has been created

updates the back pointer for this queue on the free list, it starts at -1 because no queue exists

QueueFront(Queue)

A pointer stores an index, it starts at -1 when null…If > or equal = 0

Returns the array index of the front of the queue

updates the front pointer for this queue on the free list

else

returns IsFree then

updates the front pointer for this queue on the free list, it starts -1 and -1 when null

Enqueue(Queue, Item)

Checks IsFree to avoid overflow

Back = QueueBack

array.IsFree = item, Back # the free space has an item added, next is back of queue

Dequeue(Queue)

Deletedindex = QueueFront

Checks Deletedindex > 0 otherwise underflow

Else temp = array.Deletedindex

If DeletedIndex < IsFree update IsFree, this keeps the pointer moving back, saving space

Cycle sequentially through next values until next == DeletedIndex and set next == null

Array.Deletedindex = -1 or whatever our null value is, free up space, deallocate

Return temp

Methods for Stacks in the Array

getAllocationSize(stack)

Returns size of Allocation for the Stack or Queue

getLength(stack)

Returns current length of stack or queue, this sits on the free space managing list

IsEmpty(stack)

Returns True if a stack is empty

IsFullint(stack)

Returns an integer == (AllocationSize-length)

StackTop(stack, index)

A pointer, it stores an index on the array for the stacks top

Returns that index and updates to the new index, if it is different

Push(stack, item)

Checks IsFullint > 0 otherwise resize/reallocation needed

Checks IsFree to avoid overflow

OldTop = StackTop(IsFree)

Array.IsFree = Item, OldTop

Updates IsFree to the next Free entry in the array, resize if -1

Pop(stack, item)

Checks IsEmpty to avoid underflow

Top = StackTop(next)

Temp = array.Top

Array.Top = -1 or null whatever our value is

If Array.Top < IsFree update IsFree, this keeps the pointer moving back, saving space

Return Temp

If IsFree ever reaches > the size of the array there needs to be a resize of the array.

This is a more complicated method that I'm not sure how to write, it needs to search sequentially for a null value passing through the list at least once, or alternatively, moved back reliably with each deletion and forward after each insertion until the end is reached.

The free list shouldn't have a problem with size.