LinkedList

Singly Linked List

SLL Big O/Methods

Theorem 0.1

Adding

Adding to Front: O(1)

- Create new node, point next to head, and set the new node to be the new head
  - If list is empty, the head is null which actually works out anyways without edge case (?)

Adding to Back O(n)

- Need to traverse to last node by iterating until curr.next is null (since we need access to last node). Set last mode w/data's next value to the new node.
- If head is null, point head to new node

Theorem 0.2

Removing

Removing from Front: O(1)

- Save data from head node, then set head = head.next

Removing from Back: O(n)

- Need to traverse until curr.next.next is null, then set curr.next to null
- If size is zero, throw exception
- If size is 1, set head to null

Tail Pointer

Having a **tail pointer** makes *adding to back easier*, since you can just set the tails next reference to the new node and update tail. So adding to back is now **O(1)**

Doubly Linked List

Generally doubly linked lists always have both a head and tail pointer, and contain a reference to previous node.

Note 0.1

For a DLL of size 0, both the head and tail point to null. For a DLL of size 1, both the head and tail point to the same node.

DLL Big O/Methods

Theorem 0.3

Adding

Adding to the Front: O(1)
- Set the new nodes next to head, and set the head's previous to new node. Then set head to the new node.
- When size = 0, set head and tail to new node

Adding to the Back: O(1)
- Set the tail's next to the new node, and the new nodes previous to the tail. Then set tail to new node.
- When size = 0, set head and tail to new nodes

Theorem 0.4

Removing

Removing from the Back: O(1)
- Set tail to tail's previous, then set tail next to null.
- When size = 0, set head and tail to null

Removing from the Front: O(1)
- Set head to head's next, then set head.previous to null.
- Size = 0 -> exception
- When size = 1, set head and tail to null.

Having **doubly linked lists** makes *removing from back easier*, since to remove you need to go to the node before the last one, and you need to reset tail. So you can set the second to last node.next to null and reset the tail to the second to last node. So it becomes **O(1)**

Circularly Singly Linked List

The last node in the list points back to the head

Note 0.2

For CSLL, we can't use curr == null to check if we've reached the end of the list. Instead, we must use curr == head to terminate the loop

CSLL Big O/Methods

Theorem 0.5

Adding

Adding to the Front: O(1)
- Create a new, empty node. Connect the new node's next to head's next. Set head's next to the new node. Put the data from head into the new node. Put the data we want to add into the head node.

Adding to the Back: O(1)
- Same steps as add to front, but now set head = head.next

Removing

In general, removing cannot be optimized to be O(1) unless removing from front/edge cases

Removing from Front: O(1)
- Save data from head to return
- Copy data from head's next into head
- Set head's next to head.next.next
- If size = 1, just set to null

Removing from Back: O(n)
- Need to iterate to the end of the array and set the 2nd to last node to point to head (?)