CS 1332 Midterm 1 Study Guide

Krish Katariya

Last updated: **February 04, 2024**

Contents

1. Arrays

Arrays allow you to store data in *contiguous space in memory*

Note 1.1

Pros:

- Arrays are flexible in what they can store (primitives, reference types, etc)
- *Constant time access* when the index is known
  - Accessing when index is not known (searching) -> O(n)

Cons:

- If you run out of space you need to resize the array which is O(n)

2. ArrayLists

ArrayLists are backed by arrays, and are **contiguous**, which means you cannot have null spaces between data elements. This causes us to need to shift data to fill up the null spaces after remove operations.

2.1. ArrayList Big O

Theorem 2.1.1

Adding

Adding to Front: **O(n)** -> need to shift elements over to make space to add

Adding to Back: **amortized O(1)***. There is no need to shift but its amortized because every n operations, you need to perform an O(n) operation by resizing

- Amortized: when an "expensive" operation occurs infrequently so we can "average" it over the runtimes

Removing

Removing from the Front: O(n) -> must shift elements to fill the empty space

Removing from the back: O(1) -> simple set to null

Adding and Removing at a Given Index: O(n) -> shift data around the index

Accessing at a given index: O(1) -> arraylist backed by array

2.2. Pros and Cons

Note 2.2.1

Pros

- Data elements are stored contiguously
- **Dynamic Memory** - even though we resize the backing array behind the scenes, we consider ArrayLists to be dynamic

Note 2.2.2

Cons

- Cannot store primitives
- Still needs O(n) operations for resizing

3. LinkedList

3.1. Singly Linked List

3.1.1. SLL Big O/Methods

Theorem 3.1.1.1

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 3.1.1.2

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

3.2. 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)**

3.3. Doubly Linked List

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

Note 3.3.1

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.

3.3.1. DLL Big O/Methods

Theorem 3.3.1.1

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 3.3.1.2

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)**

3.4. Circularly Singly Linked List

The last node in the list points back to the head

Note 3.4.1

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

3.4.1. CSLL Big O/Methods

Theorem 3.4.1.1

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 (?)

4. Stacks

Definition 4.1

A **stack** is a last in, first out (LIFO) linear data structure, meaning that additions and removals happen on the same side of the structure.

The main operations for stacks include:

- **push(data)** - adds the data to the "top" of the stack
- **pop()** - removes the data at the top of the stack and returns it
- **peek()** - returns data for the top of the list without removing

4.1. SLL-Based Stack

- Does not need a tail pointer

Note 4.1.1

An SLL based stack uses the *front of the SLL as the top of the stack*. Thus, push simply becomes addToFront and pop becomes removeFromFront, both of which are **O(1) operations**

4.2. Array-Based Stack

- Requires a size variable along with the array

Note 4.2.1

In this case, the top of the stack is the back of the array. So we push by adding data to **arr[size]** and pop by removing the value at **arr[size-1]**, both of which are **O(1)** operations.

5. Queues

Definition 5.1

A **queue** is a first in, first out abstract data type. Thus, queue and dequeue operations occur at *opposite* ends of the structure

The main operations for queues include:
- **enqueue(data)** - adds data to the "back" of the queue
- **dequeue()** - removes the data from the front of the queue
- **peek** - returns the data at the front without removing it

5.1. SLL Backed Queue

Note 5.1.1

The SLL-backed queue requires a *tail pointer* in order to get O(1) operations.

The "front" of the queue is the front of the list where data is dequeued from, while the "back" of the queue is the back of the list where data is enqueued

enqueue(data) -> **addToBack(data)**, and **dequeue()** -> **removeFromFront()**

5.2. Array Backed Queue

Note 5.2.1

Array backed queues require a size variable but also a front variable, because *the array behaves circularly*.

arr[front] is the front of the queue, and **arr[(front + size) % arr.length]** is the first empty index at the "back"

For **enqueue**
- Put the element at arr[(front+size) % arr.length] then size++

For **dequeue**
- Remove the element at arr[front], increment front and decrement size
- In this case, front = (front + 1) % arr.length when you increment so that *front never goes out of bounds*

5.3. Dequeue

Note 5.3.1

In Deques (double ended queues), we can add and remove from either side of the deque

The main operations include:
- **addFirst(data)**
- **addLast(data)**
- **removeFirst()**
- **removeLast()**

5.3.1. DLL Backed Queue

Note 5.3.1.1

The DLL backed queue requires a tail.

addFirst(data) -> **addToFront(data)**: O(1)

addLast(data) -> **addToBack(data)**: O(1)

removeFirst() -> **removeFromFront()**: O(1)

removeLast() -> **removeFromBack()**: O(1)

5.3.2. Array Backed Deque

Note 5.3.2.1

Uses a front variable and a size variably (*circular again*)

Important Indices
- arr[(front - 1) % capacity] = addFirst()
- arr[front] = removeFirst()
- arr[(front + size) % capacity] = addLast()
- arr[(front + size - 1) % capacity] = removeLast()

6. Trees

test