CS 1332 Lecture Notes

Krish Katariya

Last updated: **April 02, 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

Trees are the first **non linear** data structure that we look at.

Note 6.1

Characteristics of Trees

1. No cycles, so you can't reach a node using itself
2. Highly recursive
3. Usually implemented with linked lists rather than arrays (arrays get messy)

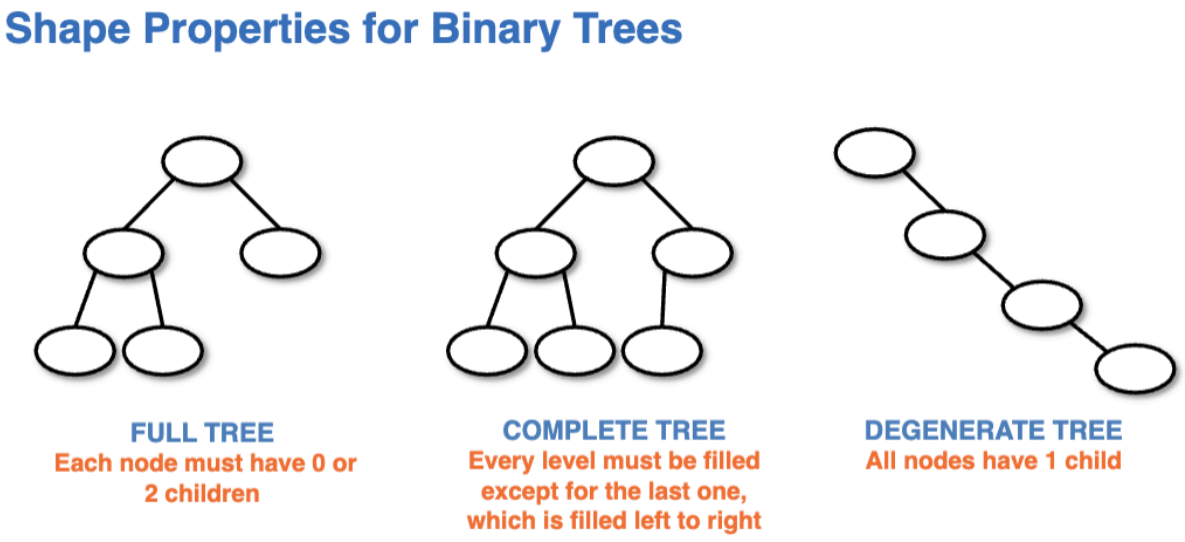
A tree is **full** if each node has exactly 0 or 2 kids, it cannot have just 1.

A tree is **complete** if each level except the last has the maximum number of nodes (2^n w/ root has n = 0)

- The last level must be filled left to right, but doesn't need to be full

A node is **balanced** if its children heights differ by only 0 or 1 (height of missing children = −1). Thus, a tree is **balanced** if all of its nodes are balanced.

- Leaf nodes are always balanced



Theorem 6.1

Important Terminology

Root: the head/entry point of the tree

Children: what the nodes point towards (can have multiple children, each node can only have one parent though)

External/Leaf Nodes: nodes without children

Internal Nodes: nodes with children

Definition 6.1

The **depth** of a node is the distance of it from the root, or how many nodes away it is(?)

The **height** of a node is the distance of a node from the furthest leaf node

- $\text{height} = 1 + \max(\text{height}(\text{children}))$

6.1. Binary Trees

Note 6.1.1

Characteristics of Binary Trees

- Shape: each node can have at most 2 children
- Children are labeled left and right (left precedes right)

Theorem 6.1.1

Iterating through a binary tree

```
public void traverse(Node node) {
  if (node != null) {
    traverse(node.left);
    traverse(node.right);
  }
}
```

6.2. Binary Search Trees

Definition 6.2.1

Binary Search Trees are Binary Trees with the given rule: any child node to its left must be less than that node, and any child note to the right must be greater than it. This applies for all children nodes, not just direct children.

The motivation behind BST's are the **binary search algorithm**, because we are able to split the search space in half each operation, making searching **O(logn)**.

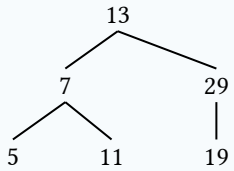
6.2.1. Pre Order Traversal

Definition 6.2.1.1

In **preorder traversal**, we look at each node and then traverse left, then right. If you draw a "glove" around the tree, you would print a node every time you touch the left side of the node

```
preorder (Node node):
  if node is not null:
    look at the data in node
    recurse left
    recurse right
```

Example 6.2.1.1



In this case, our traversal would print *13 7 5 11 29 19*

6.2.2. Post Order Traversal

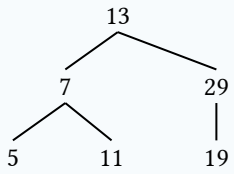
```
ppreorder (Node node):
  if node is not null:
    recurse left
```

```
    recurse right
    look at the data in node
```

Definition 6.2.2.1

In **postorder traversal** we first recurse left and right before looking at the data in the node. So if you wrap around the tree like a glove, every time you reach the right side of a node, you print.

Example 6.2.2.1



In this case, our traversal would print *5, 11, 7, 19, 29, 13*

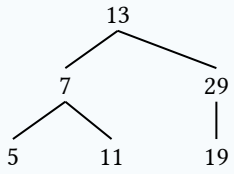
6.2.3. In Order Traversal

Definition 6.2.3.1

In **inorder traversal**, we first recurse left, then look at node, then recurse right. So if you wrap around the tree like a glove, every time you reach the bottom of a node you print it.

```
preorder (Node node):
  if node is not null:
    recurse left
    look at the data in node
    recurse right
```

Example 6.2.3.1



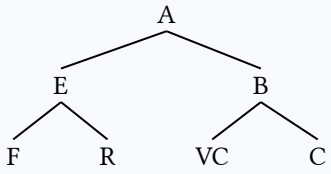
In this case, our traversal would print *5, 7, 11, 13, 19, 29*

6.2.4. Level Order Traversals

Definition 6.2.4.1

Level Order Traversal: print all the nodes at depth 0, then depth 1, then depth 2, etc

Example 6.2.4.1



In this example, we would print *A E B F VC C* in that order

7. Heaps

Definition 7.1

Heaps have the shape property of a binary tree (having 0-2 children), but heaps must also be *complete.*

This property is what makes heaps easy to implement with arrays

7.1. Min Heap

Definition 7.1.1

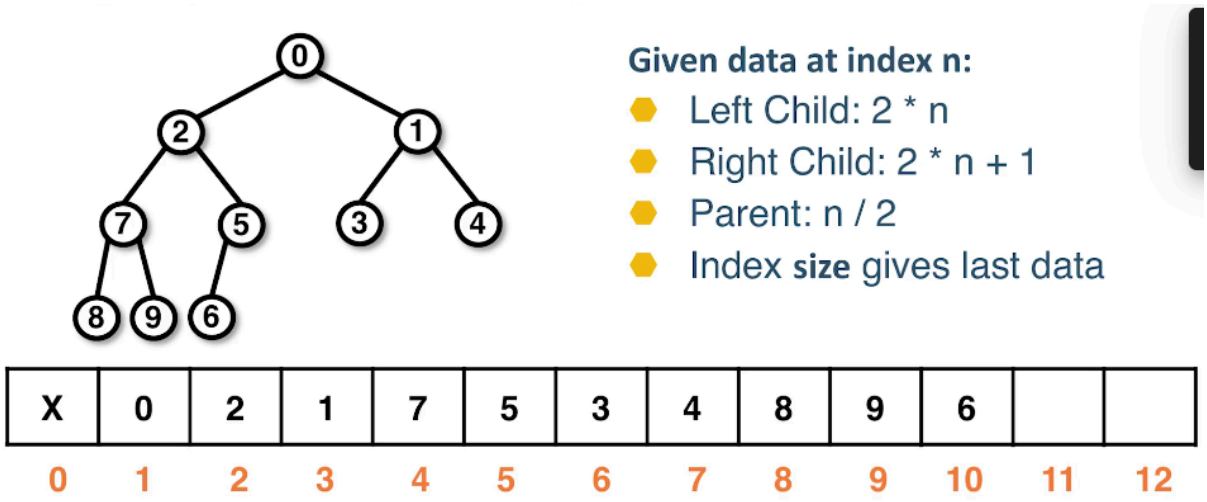
In **minheap**, the smallest data lives in the root and each child is greater than its parents value. There is no relationship between siblings

Heaps can be illustrated and *backed* as an array given their characteristics, displayed in level order.

Definition 7.1.2

Given data at index n:

- Left child: $2 * n$
- Right child: $2 * n + 1$
- Parent: $\frac{n}{2}$
  - Should be truncated if not an integer



Example 7.1.1

Heaps Use Cases

- Not designed for arbitrary searching, mostly designed for accessing root
  - They are no better than just searching an arraylist (O(n))
- Heaps are often used to back priority queues

7.2. Heap Operations

7.2.1. Add Algorithm

- Add to the next spot in the array to maintain completeness
  - this would be index[size]

Note 4.1.1

We do not use **index zero** for heaps

- Up heap starting from the new data to fix order property
  - Compare the data with the parent, and swap the data with the parent as necessary until we read the top or no swap is needed. *This differs for min heap and max heap.*

Theorem 7.2.1.1

Time complexity of adding a new element is **O(logn)**. While adding to end of array is O(1), the up-heap process is O(logn).

7.2.2. Remove Algorithm

- Move the last element of the heap and use it to replace the root (since we want to delete root)
- Down heap starting from the root to fix the order property. If two children, compare data with larger or smaller priority child, depending on if it is a min or max heap.

Theorem 7.2.2.1

Time complexity of removing an element is **O(logn)** due to the down-heap process.

7.2.3. Build Heap Operation

We use the **buildheap operation** for taking an unsorted, unordered data and turning it into a heap.

- First we put the data into an array, which will satisfy the *shape property*
- Then, in order to satisfy order property we look at the sub-heaps and down heap through the valid sub-heaps

We loop starting at `index size/2`, since this is the last element that has a child, and we go up to index 1 and repeatedly calling the down-heap method. For the rest of the array we don't need down heap if its already valid (?)

Theorem 7.2.3.1

There is more data in the bottom half of the tree than the top half, so as we go down each time the data doubles, meaning we have exponential grown in data as we go down.

The down-heap cost is **O(1)** at the bottom of the tree but increases linearly as we go up the tree meaning most of the data at the top is **O(logn)**, so it balances out to be **O(n)** if you do the summation of the series.

8. Hashmaps

Definition 8.1

Hashmaps: array backed data structure that allows us to use "custom" or flexible keys instead of using only indices. It uses a *hashing* function to dictate which indexes corresponds to what data in the backing array.

No 2 entries cannot have the same key, and you cannot go into a map entry and change the key (*immutable*).

Hashmaps generics are denoted as <K, V> for the type for Key, and the type for Value.

Hashmaps are backed by arrays, and we put data into a certain index based on its key value. In order to calculate an index for a certain key, we use a **hashing function**.

Theorem 8.1

Hash Function: represents our key object as an integer value(hashcode)

Maps the hashcode to an integer in the backing array.

`if A.equals(B)` ⟶ `A.hashcode == B.haskcode` If hashcodes are the same, we CANNOT assume that A.equals(B)

We use a compression function by using % on the hashcode in order to get a smaller index. This is useful when hash functions output very large numbers even when there are only a few entries. However, this can cause collisions!

Note 8.1

Where to put Key?

```
index = abs (key.hashCode() & arr.length)
```

How do we avoid collisions?

We do this by controlling the size of the backing array. **For hashmaps, it is very bad to let the backing array get full**, because this causes collisions (from the `%` operator). In order to solve this, we resize while there is still space left in the array.

Theorem 8.2

load factor $= \frac{\text{size}}{\text{capacity}}$

We usually set a maximum allowed load factor in order to avoid collisions. This factor is typically between 0.6 to 0.8, and Java's default is 0.75. After this loadfactor, we resize.

Note 8.2

Small Load Factors has more resizes and fewer collisions, while **Large Load Factors** are more efficient with memory but cause more collisions.

Another way to avoid collisions is to have a *good hashing function*, where different items have different hashes, and a harder goal: similar items have way different hashcodes.

8.1. Collission Handling

External Chaining

Chaining together all the entries that end up at the same index and keeping them in a small linked list. This is a *closed addressing strategy*, because entries are put where they actually belong. Here, array element points to the head of the linked list

Note 8.1.1

We still want to resize periodically so that the chains don't get too long and impact run times. So we need to see each value in each linked list to find the size of the array

Resizing

For each entry in the old backing array:

For each entry in that index: calculate where to put the entry in the resized array: `key.hashCode() % newBackingArray.length`

8.2. Linear Probing

Linear proving is considered **open addressing**, because the index we compute will change. Unlike external chaining, only one key value pair can occupy an index.

To identify a new index for an entry, we use a technique called **probing**, which are open addressing strategies, because the first index we calculate is not necessarily where we might end up putting it.

Definition 8.2.1

Linear Probing: If a collision occurs at a given index, increment the index by one and check again. Do this until a null spot is found.

`index = (h + origIndex) % backingArray.length` h = . number of times probed (0, 1, 2, N).

At first, assume h is zero. If they key value pair isn't found, then add to the index.

9. AVL Trees

Definition 9.1

The **balance factor** of a node is the left child's height - right child's height

- What do different values of the balance factor mean?
  1. 0: Perfectly Balanced
  2. 1: Leaning left but ok
  3. neg 1: Leaning right but ok
  4. >= 2: unlabaced to far left
  5. <= neg 2 unbalanced to far right

Definition 9.2

In **AVL Trees**, the $|\text{node.balancefactor}| \leq 1$. We chose 1 and not 0 because there are some situations where it is straight up impossible to have a perfectly balanced tree.

AVL's help us hit the sweet spot of allowing only a little imbalance while keeping operations efficient.

9.1. Single Rotations

These are used when adding or removing a single node to a tree if it becomes imbalanced.

To perform a rotation when a node has a balance factor of −2: look at the right child first and see if it has a balance factor of 0, −1. Then we can do a simple, single rotations.

Theorem 9.1.1

```
Algorithm leftRotation
1) Node B <- A's right child
2) A's right child <- B's left child
3) B's left child <- A
4) (ALWAYS FIRST BEFORE 5) Update the height & BF of A
5) Update the height & BF of B
6) Return B
```

Right rotations are the mirrored opposite of left rotations.

When to use which rotation?

Use left rotation when

- node.BF = −2
- node.right.BF = −1, 0

Use right rotation when (? not sure)

- node.BF = 2
- node.right.BF = 1, 0

> A nodes balance factor shouldn't go past 2 without a rotation happening

10. Iterative Sorts

10.1. Bubble Sort

Bubble sort inspects pairs of adjacent elements, and swaps as necessary which "bubbles" the maximum element to its correct position.

Pseudocode

```
stopIndex = array.length - 1
while stopIndex != 0:
  while i < stop
    if (arr[i] > arr[i+1]):
      swap values at i and i+1
    i++;
  stopIndex--;
```

Note 10.1.1

One optimization is to make the sort adaptive, meaning it will stop early if necessary. To do this, we have a flag `swapsMade` and if false after an iteration, we can stop early.

Theorem 10.1.1

Time Complexity

Best Case: O(n) (already sorted)

Average Case: O(n^2)

Worst Case: O(n^2) (reverse sorted array)

Bubble sort is *stable, adaptive, and in-place*

11. Cocktail Shaker

Iterative sort that performs bubble sort twice in one iteration, to bubble the largest to the end, and the smallest to the front, shaking the list back and forth

The first stage of cocktail shaker sort loops through the array from left to right, swapping values out of order. At the end of the first iteration, the largest number will reside at the end of the array.

The second stage loops backwards from the end, re-swapping when necessary. However, we will start going backwards at the most recently sorted item, since everything past that should be sorted.

Theorem 11.1

Time Complexity:

Best: O(n) (fully sorted array)

Average Case: O(n^2)

Worst: O(n^2) (reverse sorted array)

Cocktail shaker sort is *stable, adaptive, and in-place*

12. Divide and Conquer Sorts

12.1. Merge Sort

Merge sort recursively divides an array into half, sorts that half, and then merges the two sorted halves back together.

Pseudocode

```
if array.length = base case
  return array

length = arr.length
midIndex = length / 2
left = arr[0:midIndex-1]
right = arr[midIndex:length-1]

merge(left)
merge(right)

initialize i, j
while i and j are not at the end of the left and right arrays:
  if left[i] <= right[i]:
    arr[i+j] = left[i]
    i++
  else:
    array[i+j] = right[j]
    j++

while i < left.length
  arr[i+j] = left[i]
  i++

while j < right.length
  arr[i+j] = right[j]
  j++
```

Theorem 12.1.1

Time Complexities

For all cases, merge sort is O(nlogn).

Merge sort is stable, but not adaptive and not in place.

13. Pattern Matching

13.1. Boyer Moore Algorithm

Definition 13.1.1

Last Occurrence Table: records the index of the last occurrence of the letter. We store it in a pair <letter, index> in a hashmap, and letters not in the alphabet of the pattern as marked as null, or returned as −1 in the functionality

Boyer Moore Last Table(pattern)

```
m = pattern.length
last = HashMap<character, index>
for all i from 0 to m-1
  last = put(pattern[i], i)
end for
return last
```

Theorem 13.1.1

Actual Search Algorithm

1. Create the LSOT to optimize shifts past mismatches
2. Move right to left in pattern
3. If there is a match, continue comparing text and pattern
4. If there is a mismatch, look to see if text character is in the alphabet
   - If the char is in the alphabet, align them
   - If the char is not in the alphabet, then shift past mismatched area algotegher