“What is a HashStackTrieFilterMap?” A Data Structures and Algorithms Refresher for Everyone

Sarah Withee

WARNING: DO NOT ASK THESE QUESTIONS IN JOB INTERVIEWS!!!

(They ask questions of knowledge, it’s just pure memorization, don’t show skill or experience)

Part I: The basics

Data structure: way of storing data and organizing data in a computer so it can be used efficiently

Arrays/lists:

* one continuous chunk in memory

[1][3][7][4][9][ ][ ][ ]

can’t expand once created

Linked Lists:

NOT in one continous chunk of memory

[5] <-> [4] → [2] → [9] → 0/NULL

class Node {

int num;

Node next;

Node prev;

}

Singly-linked list: one link, goes “forward”

Doubly-linked list: two links, goes “forward”/”backward”

Circularly-linked list: end points to beginning

Create:

- arrays: create all elements at same time

- LL: starts off as null

Adding/Delete:

- Arrays: just keep adding, really quick

- LL: add to end means you have to traverse through it

- Arrays: delete in middle, shift elements over or deal with empty holes

- LL: redirect pointers, super quick

Accessing/searching:

- Access in array: just go there (arr[4])

- Access in LL: you have traverse through it

- Search in array: simple for loop

- Search in LL: simple for loop

Space:

- Arrays: always same size, new size means new array then copy it over

- LL: always the same size as elements you have

Decisions:

- Both for simple lists

- Arrays are easy to implement, LL writing some code

- Arrays have fast seek times

- Searching are about the same

- Rearranging in arrays are a pain, LL are nicer

- Larger data choose an LL

Problem:

- A phone book (for lookups)

- List of 100 odd integer above 10

- list of library book titles

- list of library books that are checked out at the moment

- names of seessions at a conference in alphabetical order

Steps for adding to Linked List

0

[5]→0

while currentPtr.next != null: currentPtr=currentPtr.next

[4] [5].next = newNode

[5]→[4]→0

[5]→[4]→[9]→0

deleting:

cp

[5]→[4]→[9]→0

pp cp

[5]→[4]→[9]→0

[5]→ [9]→0

cp = 0

head, tail

[5]→[4]→[9]→0

How might you merge 2 lists together?

[a]→[b]→[c]

[d]→[e]→[f]

[a]→[d]→[b]→[e]...

Stacks/Queues

Stacks:

- like books

- “first in last out” (FILO)

- “last in first out” (LIFO)

- adding “push”, removing is “pop”

Queue:

- like cashier lines

- “first in first out” (FIFO)

- “last in last out” (LILO)

- adding “enqueueing”, removing “dequeueing”

- “peek” “front” “top”

build out of arrays:

- fixed sizes

- stack track the top

- queues track front/back

build out of LL:

- usually easier

- stacks with singly-linked (the “top” is the first element)

[5]→[4]→[8]

top

top

[4]→[5]

[8]→[4]→[5]

queue

front back

[4]→[5]→[6]→[7]

When would we use these?

Stacks:

- balancing things

( [ { ( { }}}}}

- backtracking (undo/redo)

- math parsing

5+4\* (3-6)

- recursive things

queues:

- things shared in order

- asyncronous data

- load balancing

Problems:

- “pull a number”

- reverse a string

- if string had right number of open/close parenthesis

- routing telephone calls in customer support

- determine order of boxes to remove from a shipping truck

Practice problems:

1) How would you implement queue as an array

2) merge two arrays in sorted order

3) Reverse array using stack, then a queue

4) check a string if it’s a palendrome

5) how to change a queue so important items get moved to the front

Trees

- what if you had a list, but instead of linear, it was hierarchical?

- children have parents, parents have parents, children can have children

- in a LL, instead of next/prev, what if we had children links?

head[a]

/ \

[b] [c]

Why we would we want to use trees?

- hierarchial things easier

-Things with some ordering built in

- insertion is quick

- fast to search through

- parent nodes, children nodes

Types of trees:

binary tree: up to 2 children

Binary search tree: insertions are done in order to make them alphabetical

c

/ \

b e

/\ /\

a c b r

Full binary tree: 0 or 2 children

Complete binary tree: every level is full (bottom level is as full as can be)

Perfect binary tree: 2 children (not 1)

pathological tree: all nodes have 1 child (basically a LL)

AVL tree: where all the levels between left/right subtrees are no more than 1

Expression tree: each node corresponds to a math operator

+

/ \

3 \*

/\

4 8 3+ (4\*8)

Decision tree: each node is decision point

Heap: a self-balancing and sorting tree (usually where min or max items rise to the top)

Trie: special tree used to quickly search things

Heap:

- complete binary tree

- max-heap and min-heaps

– both present the root node to the max/min item

- this works recursively throughout

- when a new item is added, it “reheaps” (rearranges items)

8

/ \

5 3

Why a heap?

- it a balanced structure

- easiest to find largest/smallest item

- can be used for sorting

- Can be used to store things of different priority levels

- this is how memory allocation works internally

- Con: when you insert/delete, time/computation involved

hi. who is reading my screen?

Is anyone looking?

Hello? Bueller? Bueller?

Omg so many data structures

Hashmaps/Hash tables/map

What is a hash?

- given an input, returns some unique output

- n % 10

[100][][52][][][][][][][79]

What’s a map/set/pair?

Ties a “key” to a “value”

What’s a hashmap?

- a list of key-values where keys are hashed values

- hopefully stores things uniquely

- hopefully results in instantaneous lookups

- often some sort of array, linked lists, or table

why?

- super fast

- should eliminate searching and give it to you instantly

What if it returns duplicate?

- store multiple things

1 2 3 4 5

| | | | |

4 8

8 6

2 5

- store it in the next available thing

[][][52][33][74][84][][][][][]

[“aaron”][4]

[“jessica”][3]

Examples:

- winners of lottery have phone number stored

- hashed phone number

- stored amount they won

When might we use this?

- counting things

- finding duplicates

- Finding subsets of things

- anythign where our natural instinct is to use something else (maybe)

Don’t use this for…

- add a huge number of items

- with a poor hash function

- if things will collide in a particular spot a lot

Graphs:

- a non-linear data structure with nodes(vertices) and edges

- usually represent some sort of network

Directed/undirected

- undirected ties nodes/vertices together

- directed means traversing goes one way

- bi-directional means traversing goes both ways

[5] –> [6] = [2]

\|/ ||

[4] [9] -> [1]

Weighted/unweighted graphs

- weighted means edges have a value/weight to it

- unweighted is the same value

[5] ~~2~~ [6] ~~5~~ [2]

~~3~~ ~~7~~  ~~4~~

[4] [9] ~~3~~ [1]

cycle = a loop in a graph

(maybe use stacks to prevent cycles)

use a table!

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 1 |  | x |  |  |  |  |  |  | x |
| 2 |  |  |  |  | x | x |  |  |  |
| 3 |  |  |  |  | x |  |  |  |  |
| 4 |  |  |  |  |  |  |  |  |  |
| 5 |  |  |  | x |  | x |  |  |  |
| 6 |  |  |  |  |  |  |  |  | x |
| 7 |  |  |  |  |  |  |  |  |  |
| 8 |  |  |  |  |  |  |  |  |  |
| 9 | x |  |  |  |  | x |  |  |  |

Store edges in table

- weighted use numbers

- directions you remove the item in the table

Why use a graph?

- Trees only work so far, sometimes you need larger/more non-linear

- traversing paths through things

- Trees don’t backtrack

- trees have a starting point, graphs dont

Algorithms

Big O notation

- way to measure the complexity of an algorithm

- how long something might run with larger data sets

- “cost” of an algorithm

- Why “Big O”? order of a function

- little o, theta, omega, maybe others

- this the worst case

- we use “n” data points

print n – 1 action

for (I = 0 to n) … n number of actions

for (I = 0…)

for (j = 0…) n + n actions, 2n

for (I = 0 to n)

for (j = 0 to n) n\*n = n2 actions

for (I = 0 to n)

for (j = 0 to n\*n) n\*n\*n = n3 actions

O(n) O(n2) or O(1)

O(3n^2 + 5n + 6)

drop constants in front of variables

=> O(n^2 + n)

Big-O of…

- search through an array O(n)

- search through 2D array O(nm) = O(n2)

- search one path down a binary tree

Ex: fibonacci function

function f(n)

if n == 1 return 1

else return f(n-1) + f(n-2)

f(5) = f(4) + f(3)

f(4) = f(3) + f(2)

f(3) = f(2) + f(1)

f(2) = f(1) + 1

f(1) = 1

O(n^3)

for (I = n to 0)

sum = I-1 + i-2

O(n \* 3) = O(n)

Arrays/LL:

searching: O(n)

adding: O(1)

deleting: O(1) (from the end)

in the middle: O(n)

insert in middle: O(n)

reverse array: O(n)

Trees:

Add: O(log n)

Search (BST): O(log n)

Sorting

- Bubble sort:

for (I = 0, I < n, i++)

for (j = I, j < n, j++)

if (j < I)

swap(i, j)

83642

38642

28643

26438

26348

987654

O(n2)

Insertion sort

- simple

- tries to insert the current element in the right order

O(n2)

Selection sort

- goes through and “selects” the smaller item to put in place

O(n2)

Merge sort

merging – O(n)

splitting – O(log n)

sorting – O(n log n)

3 5 2 1 4 6 8 7

3521 4687

35 21 46 87

3 5 2 1 4 6 8 7

35 12 46 78

1235 4678

12345678

Quick sort

- pick a pivot point, then sort left and right sides

35214687

352 1 4687

3 5 2 1 4 6 87

2 3 5 1 4678

1 2 3 4 5 6 7 8

O(n log n)

heap sort:

- add things to heap

- heap sorts it

- traverse back out items

O (n log n)

geekygirlsarah.com/algorithms/

@geekygirlsarah