CS 213 - Data Structures and Algorithms

January 23, 2021

1 Introduction

Twitter's banner problem - the agenda is to analyze this problem and work towards a solution.

CEO of twitter - Parag Agarwal - his student. Well anyway. What's the problem at hand? People post content on twitter. There's some four people on the banner page of twitter. Those four people will show up to scrapers etc.

Problem: Place "most popular posts on the top page dynamically - updated every five-ish minutes or so."

If some folk says something super-interesting that has the potential to go super-viral, then you want to push that person's post onto the banner page.

So, consider an array of popularity indices. You as a programmer need to process this. We will work with toy models only - no full sized user bases and all.

We want to get the most popular from the list. $\mathcal{O}(n)$ to get `max`, then there's going to be a hole to plug. One approach - sort the list, pop from the stack. This way no holes to be dealt with. Problem with sorting (in this particular situation) - if a super-interesting post comes in - we will not be in a position to dynamically adjust the stack to prioritize it. Also, sorting is something like $\mathcal{O}(n^2)$ - number of cycles doubles in order of magnitude.

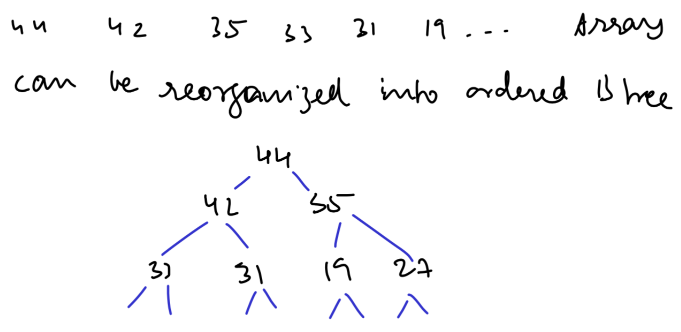

44 42 35 33 31 19 . . . Array
can be reorganized into ordered Btree
44
42
55
33
31
19
27

Figure 1: figures/btree.png

The data can be reorganized into another data structure that is more convenient w.r.t insertions, searching etc. See the above figure. That is a complete, ordered binary tree (ordered $\implies$ left child and right child are distinct for each node, binary $\implies$ two leaves per node, 2^i nodes in level i).

Internally, a binary tree such as this is still stored as an array. If we want the right child of j, we only need realize $j \to 2j+1$., which is like ultra easy for the computer.

- Right child: $j \to 2j+1$
- Parent: $k \to \frac{k}{2}$

The tree has a

- Structure property: it's a complete B tree
- Comparison property: the internal array is sorted - reflects in the B Tree representation

So, how would we go about handling the banner problem? The problem is finding the most popular, next most popular and so on... And the twist is that there are insertions and deletions happening dynamically in the internal array.

So, the first part is *Extract max.* Consider the figure above. This is what

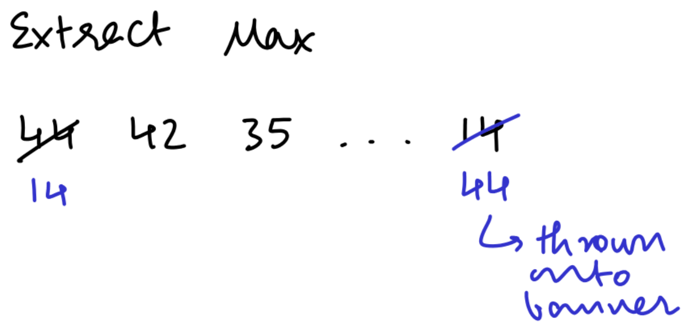


Figure 2: figures/extractmax.png

we're doing when we select the maximum from the array. 44 and 14 have been mutually swapped. But, this destroys the ordering property.

We can then perform a sequence of operations to fix the order. We can keep

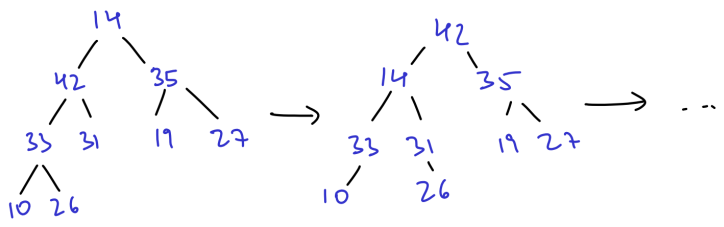


Figure 3: figures/fix.png

performing swaps between parent and child as long as we maintain the invariant that `parent` > `child`. And because. There will be at most $\log_2 n$ comparisons required, where n is the number of elements in the internal array.

Now, if there is some database trigger and the stack gets updated, then in general the comparison property will be destroyed. We can do the same process of sorting them - this time propogating up or down depending on how the comparisons hold up.

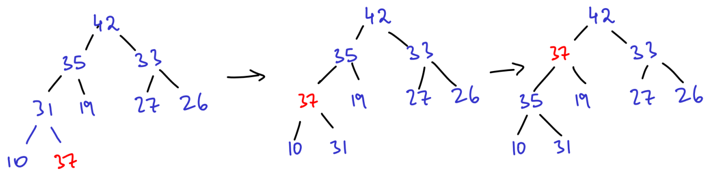


Figure 4: figures/insertion.png

Heap: a complete binary tree with the comparison property. Selecting the right data structure is like choosing a scredriver for a screw that you want to work with. You'd want to choose the best fit, ya? Don't want to be shoving square cross-sections into triangle holes.

2 Recursion

- Basic idea and basic principle
- Parameterization
- Computational complexity
- Tail recursion

Example: Tower of Brahma/Hanoi

Three rods A, B, C, and disks of decreasing size mounted onto the rods. Only one disk can be moved from one rod to another in one move and a larger disk cannot be mounted on top of a smaller disk. The objective is to move all the disks from rod A to rod B while respecting these rules. A recursive solution for this problem is very simple and elegant.

A recursive solution for a problem of size n requires knowing the solution to size $n-1$ and then using it to solve the next higher problem. Also, the program must terminate, so you need a base case. This whole business is very similar to induction.

The **base case** here is $n = 1$, in which case we just move $A \to B$. This is what the base case would look like in our coded function.

```
/* the tower of Hanoi/Brahma
 * For n number of disks,
 * print the move sequence
 * of the solution */
```

```c

#ifndef HANOI_H
#define HANOI_H

void hanoi(int n, char a, char b, char c) {
        if (n == 1)
                printf("move from %c to %c\n", a, b);
```

Listing 1: Base case of tower of Hanoi problem

$n = 2$ is simple,

1. $A \to C$
2. $A \to B$
3. $B \to C$.

Think of the above steps in this way; The first step solves $n = 1$ for rods A and C, then we move the next larger disk from A to B, and finally the last step solves the base case for rods C and B, so that the smaller disk rests on top of the larger one. Take a look at the whole code

```c
/* the tower of Hanoi/Brahma
 * For n number of disks,
 * print the move sequence
 * of the solution */

#ifndef HANOI_H
#define HANOI_H

void hanoi(int n, char a, char b, char c) {
        if (n == 1)
                printf("move from %c to %c\n", a, b);
        else {
                hanoi(n-1, a, c, b);
                printf("move from %c to %c\n", a, b);
                hanoi(n-1, c, b, a);
        }
}
```

Listing 2: Complete solution of tower of Hanoi

Essentially, this solution first does a bulk move $A \to C$, then moves the n^{th} disk $A \to B$, and then moves the $n - 1$ disks from $C \to B$.

How does one calculate the total number of moves? We do not have any explicit loops. Recursive algorithms are best described using recurrence relations like so

$$M(n) = 2M(n-1) + 1 \tag{1}$$

where $M(1) = 1$ is the base case. So, each call does two more calls and a move.

Expanding the recurrence relation, we get

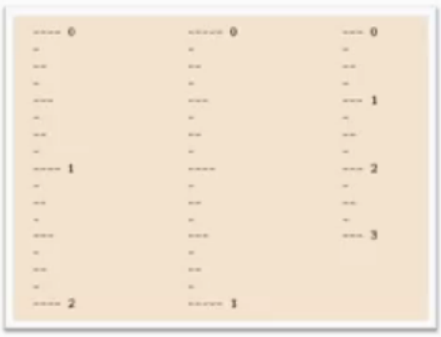
$$\begin{aligned} M(n) &= 2^{n-1} + \sum_{i=0}^{n-2} 2^i \\ &= 2^n - 1 \end{aligned} \tag{2}$$

Essentially, solving a problem using recursion is about finding the recurrence relation. Once you have the base case and the recurrence relation that always reaches the base case, the problem is pretty much done. The question of correctnes remains, as goes without saying.

The base case and its termination is very important. Professor recommends that focus on this first before moving on to the recurrence relation.

Draw Ruler

Print the ticks and numbers of a scale (or ruler)



```
---- 0          ----- 0          --- 0
-               -                -
--              --               --
-               -                -
---             ---              --- 1
-               -                -
--              --               --
-               -                -
---- 1          ----             --- 2
-               -                -
--              --               --
-               -                -
---             ---              --- 3
-               -
--              --
-               -
---- 2          ----- 1
```

Figure 5: Sample output of the ruler problem

So, a ruler is essentially a fractal; zooming in between a pair of ticks, we see the same pattern again. The ruler is a mix between two things

- Numbered inch markings
- Subticks between inch markings, not numbered

So, what are we supposed to do?

- Identify the recursive structure
- identify the base case
- complete the base case
- solve the problem

Our function, `drawTicks(length)` takes a single integer, length of the ticks with inch markings, and prints the ruler of that tick length.

Base case: `length == 0`. In this case we draw nothing. What do the recursive calls look like? We first draw a tick of size `length` with label, then draw an interval of center-tick length `length - 1` and finally draw another tick of size `length` with label.

An interval of a given `center_length` has two smaller intervals before and after it with center-tick length `center_length-1`.

I tried to port the lecture's code to CeeLanguage, but there appears to be some logical error.

```c
#include <stdio.h>
#include <string.h>
#include "ruler.h"

void drawLine(int tick_length, int tick_label) {
        char line[tick_length];
        int i = 0;

        while (i < tick_length)
                line[i++] = '-';


        printf("%s", line);
        if (tick_label + 1 != 0)
                printf(" %d", tick_label);
        putc('\n', stdout);
}

void drawInterval(int center_length) {
        if (center_length > 0){
                drawInterval(center_length-1);
                drawLine(center_length, -1);
                drawInterval(center_length-1);
        }
}

void drawRuler(int num_inches, int major_length) {
        drawLine(major_length, 0);
        for (int j = 1; j < num_inches+1; j++) {
                drawInterval(major_length-1);
                drawLine(major_length, j);
        }
}
```

Listing 3: Recursive and helper functions for the ruler problem