Heap

Fredrik Berzins

Fall 2023

Introduction

Heap is like a queue with the difference of order when removing where queue works on first in first out, heaps work on a priority. Heaps are useful for a lot of thing for example instructions to a processor where some instructions gets higher priority.

Heap explanation

Heaps can be made from almost any data structure the only requirement is that it can work as a queue with priority for what elements comes out next. This also means theirs a lot of choose when implementing it to get a efficient solution for that particular case. A heap can in many cases be seen as a sorting method and queue combined since most heap implementations are sorted in some way.

Implementation

Heap as linked list

Heap as a linked list is a simple and easy way to work with heaps, it can be done in two main ways. A unsorted linked list or a sorted linked list, the benefits of unsorted is that adding elements has big O of O(1) and removing O(n). If the list is sorted the the big O notation is reversed so it's constant to remove and linear to add. A reason for it being easy to work in is that it inherently needs nodes that can be filled with all relevant information.

Heap as array

Heap as a array is slight more complicated the a list but still easy to work with. An array can be used as a heap in multiple ways one of those is as a representation of a tree by taking an index, multiplying it by 2 and adding 1 or 2 dependent if its the left or right branch/child.

Heap as tree

Heap as a tree is quite complicated but has the benefit of being a inherent node structure like linked list and also being able to efficiently move nodes within. (With a operation like push which only changes priority of a node and moves it directly, instead of removing, changing priority and adding it back in.)

Benchmarks

The testing was done with between 100 and 25600 operations(doubling each time, 100,200,400...). Each time test was redone 100 times witch included adding add removing n elements with random priority, all of this was don another 100 times with other random numbers. This gives a min time, this is preferred for consistency and comparability.

Benchmarks was preformed with 4 different implementations explained above and listed below.

- Linked list (With constant add)
- Linked list (With constant remove)
- Array (Tree as an array)
- Tree (Linked nodes)

For the tree heap the tests was done as the rest with n add and remove operations, it was also done with n push operations, to see efficiency of a specialized push compared to adding and removing operation.

Note: All table show an increasing amount of operations this means a constant time complexity will appear linear and so on.

Result

In table 2 below it's clear that a constant add and constant remove are very different in practice, the difference between these is from having to search thru the hole list when removing to is that's theirs no higher priority while when you sort it from the beginning you don't need to go thru the hole list, just until its sorted. The time difference with this implementation is about one forth of the time for the constant remove(sorted list) this makes it a clear choice it a list implementation is wanted.

When it comes to the array and tree version the array was slight faster but not a significant amount, while the push was a significantly faster the adding and removing separately.

Time to Add or Remove

For all benchmarks the time for add and remove was split to see if the times differed. For the two linked list implementations their was clear difference, for the array implementation both add and remove had a big O notation of O($logn$) but add was still about 20-25% of the time compared to remove. The tree implementation also had the same o notations for add and remove it was O($logn$), for clarity all big o nations are in table 1 below.

Operation	**Add List**	**Remove List**	**Array**	**Tree**
Add	1	n	log(n)	log(n)
Remove	n	1	log(n)	log(n)

Table 1: Big o notation for each operation type and heap implementation. ("Add List" stand for list with constant add and "Remove List" stand for list with constant remove)

n	**Constant Add**	**Constant Remove**
100	16	4
200	60	15
400	228	53
800	905	202
1600	3622	1067
3200	14499	4535
6400	58034	19641
12800	232246	85065
25600	930526	488479

Table 2: Time to add and remove n elements from a linked list in ms

n	**Array**	**Tree**	**Tree(Push)**
100	9	3	3
200	12	13	6
400	29	28	14
800	66	71	31
1600	138	141	72
3200	304	317	172
6400	662	728	393
12800	1466	1601	893
25600	3313	3535	2234

Table 3: Time to add and remove or push n elements from a linked list in ms

Conclusion

In conclusion when selection a approach for a heap the fastest(for both add and remove) was the array implementation. If it's time sensitive when adding or removing the linked lists will be a abetter approach all dependent on use case. even the more complex tree structure as a use if changing priority is a common occurrence.