Heap

Laasyasree Desu

Fall 2023

1 Introduction

In this assignment the objective was to implement a priority queue or a heap. This was first implemented using linked lists, with two different versions of an add and remove methods with different time complexities. These two versions were then bench-marked. The second task involved implementing the heap using a binary tree and creating add and remove methods as well as push method to find out the depth. Lastly the assignment involved implementing the heap through arrays.

2 Heap

A heap or a priority queue is a data-structure similar to a queue, however, instead of "first in , last out", it has priority elements in it. This means that the priority element (ex. the smallest integer) is put first, closest to the head and during removal the elements are returned in order of their priority. The priority queue can be implemented using different data-structures, and the name heap is associated with the implementation through a binary tree.

The first task involved the implementation using linked lists where one version was with priority on adding and one with priority on removing. This meant that the version would have a time complexity of O(n) for removing and O(1) for adding, and vice versa for the other version.

2.1 First implementation

2.1.1 Add

To achieve a constant time complexity for the add method, the element was always added to the end of the list. This was done by first checking if the list was empty, in which case the new element would made the head of the list. Otherwise the node last node in the last was kept track off and any new element was added to end of the list by redirecting the pointer to the new node.

2.1.2 Remove

The remove method however had to make sure that the element with highest priority was being removed first, in this case the smallest element in the list. This was done by first taking the head of the list as the current minimum value and traversing through the list to check for smaller integers. If one was found, the current minimum was replaced with the new one. After the whole list was traversed, the smallest integer would be found. The list was then traversed again to the node before the minimum value and the pointer was redirected to the node after it, thereby removing and returning the min element.

2.2 Second implementation

2.2.1 Remove

To achieve constant time on the remove method instead, the opposite was done, which meant that the first element (head) was retrieved and returned and the next element was updated as the head of the list. This was possible as the list was already sorted through the add method.

2.2.2 Add

The add method was done by first checking if the list was empty and if true then the node would be updated as head. Otherwise the node would have to placed in the correct position, where all the nodes before had to smaller and all the nodes after had to be larger. This was done by traversing the list to find the minimum element (similar to the above remove method) while keeping track of all the "previous" nodes. When the minimum was found the new node would be added by updating its pointer to the first element greater than it and updating the "previous" pointer to point to the new node.

2.3 Benchmark

Array size	Time taken (ns)
1000	215
2000	917
4000	3391
8000	11058
16000	45930

Table 1: Average time taken to find duplicates using pointers in sorted array.

The two different implementations of the priority queue were bench-marked and the results can be seen above.

Array size	Time taken (ns)
1000	215
2000	917
4000	3391
8000	11058
16000	45930

Table 2: Average time taken to find duplicates using pointers in sorted array.

3 Push method

3.1 Benchmark

Array size	Time taken (ns)
1000	215
2000	917
4000	3391
8000	11058
16000	45930

Table 3: Average time taken to find duplicates using pointers in sorted array.

4 Array implementation

The third task involved the implementation of the heap using arrays. Since it was tree structure, the array implementation had to incorporate left and right branches. This was done by sectioning the indexes of the array - the indexes for the left branch of a node n would be calculated by 2n+1 and for the right branch it would be 2n+2. The index 0 was used for the root of the tree. The tree also had to be a "complete" tree, meaning that all nodes would contain two "children" except for the lowest level of nodes. This level would have to be filled up starting from the leftmost node of the tree.

4.1 Add

For the add method, the new element is first added at the next available node/index (it the tree is filled to n indices the new element is added to n+1). But since it was not guaranteed that element was in the right place of priority it was compared to the rest of its parent branches. This was done by traversing back - the parent of the nodes with even indices were found by (n-2)/2 and for

odd indices (n-1)/2. Using a while loop, while the parent elements were larger than the new element, the elements would swap places.

4.2 Remove

The remove method other started off by first removing the node at the root, since that was the smallest element in the entire tree. The node was then replaced by the last node in tree. Then comparisons to find out the minimum element in the tree (since the new root is not guaranteed to be the smallest) were carried out by first calculating the "child" indices using the calculations mentioned above and then through if statements comparing the "child" elements to current node (in this case the root). If one of elements were smaller, the elements were swapped. This continues until the tree was returned to a heap.

4.3 Benchmark

Array size	Time taken (ns)
1000	215
2000	917
4000	3391
8000	11058
16000	45930

Table 4: Average time taken to find duplicates using pointers in sorted array.