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. 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. The array and list implementation were then compared by benchmarking them.

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 ensure that the remove method runs in constant time, a different strategy was employed. Instead of searching through the entire list or array to delete an element, the process was streamlined. It involved promptly taking the very first element (referred to as the "head" of the list), returning it, and then designating the next element as the new head of the list. This was possible because the list or array had already been organized to be in order of priority during previous operations - 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.

3 Linked list - heap

The next task was to implement a heap (using a tree) with linked lists. For this task, 3 methods were created, an add operation, a remove operation and a push operation. For the add operation, if the heap was empty (null), the element was added as the root. Otherwise the tree was traversed to find to find the correct placement of priority. If the value being examined in the tree was less than the new element, then they were swapped. Otherwise the traversal continued. To keep the tree balanced, the insertion of the node would happen on the branch that had fewer nodes (the number of nodes were kept track of). For the remove method, the root was simple retrieved and returned as it was of the highest priority. To replace it, the element of next highest priority moved up. This was done with several conditions - if the left branch was empty then the element on the right branch was moved up and the opposite would happen if the right branch was empty. Otherwise, if the element at the right branch was less than the one at the left branch, then it was moved and vice versa. The third method was the push method, which had the function to perform operations on an element with the highest priority, give it a new priority, and then return it to the priority queue. This was done by performing an operation on the value of the root element (incrementing) and then swapiing with nodes below in the sub-branches until it is in the right place again.

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.

List size	Remove	Add
100	3000	700
200	5300	900
400	6900	810
800	11058	1100
1600	15400	950

Table 1: Average time taken to remove and add elements with first linked list implementation.

List size	Remove	Add
100	560	5000
200	710	7200
400	530	12800
800	600	27400
1600	590	45700

Table 2: Average time taken to remove and add elements with second linked list implementation

Tree size	Remove	Add
100	5500	3200
200	11300	5100
400	21600	7900
800	49300	16900
1600	113300	34200

Table 3: Average time taken to remove and add elements with linked list implementation of heap

5 Discussion

The two different implementations of the priority queue were bench-marked and the results can be seen above (ref. table 1 and table 2). The results show relatively constant run times for the add method in the first linked list implementation and the remove method in the second. On the other hand, the other add and remove method show time complexities of O(n) as the run-time for both methods varies with the size of the lists. The linked list implementation of the heap was also bench-marked, which can be sen in table 3. Lastly, the remove and add methods for the array implementation of the heap were also conducted, which can be seen in table 4. The results show the run-time for

List size	Remove	Add
100	2200	2900
200	5700	4500
400	15100	10600
800	35300	33400
1600	81200	57400

Table 4: Average time taken to remove and add elements with array implementation of heap

the linked list implementation of the heap to be slightly slower than the array implementation, but overall generated approximately the same run-time.