Priority Queue

Leo Halfar

October 2023

1 Introduction

This report will discuss different implementations of a priority queue as well as the difference in efficiency between them.

2 Priority Queue Using a List

2.1 Class and Constructor

The "priolist" class uses an ArrayList to create a basic priority queue. The constructor initializes an empty ArrayList called "items" when an object of the class is initiated.

2.2 Add

The "add" method allows you to add an element to the priority queue. When you add an item, it simply appends it to the end of the "items" list. There are no specific rules for ordering or sorting the items in the queue.

2.3 Remove

The "remove" method is used to get and remove the element with the highest priority. It iterates through the "items" list, comparing each item with a variable that holds the highest priority value. If an item has a lower priority than the current highest priority, it updates the highest priority value and its index. After the loop, it removes the item with the highest priority by index. The time complexity is O(n) for this removal operation since it searches the entire list.

3 Priority Queue with Reverse Ordering

3.1 Add

The "add" method allows you to add an element to the priority queue with a time complexity of O(n). It loops through the "items" list to find the correct

position for the new element to maintain priority order. It stops when it finds the correct position and inserts the element there. This method ensures that higher-priority elements come first in the list.

3.2 Remove

The "remove" method efficiently removes the element with the highest priority in O(1) time because the list is maintained in priority order. It directly removes the first element from the list.

4 Linked list Binary tree heap

4.1 Node Class

The "TreeQueue" class implements a priority queue using a linked list binary tree structure. The class includes a nested "Node" class to represent the nodes of the tree.

4.2 Constructor

The constructor initializes an empty tree by setting the root node to null.

4.3 Add

The "add" method inserts an element into the priority queue by navigating the tree to find the appropriate position. First the function checks if the root is null and if it is creates a new node with the element and sets the root to it. Else it calls the "addhelp" function with the root and the new node.

4.4 Addhelp

The function contains varius if statements that ckecks for different scenarios. If the value of the new node is smaller than the root node the values switch what was the root nodes value is compared with the left and right child node. If the value is for example bigger than the the left child node and the right child has the most amount of nodes beneath it. The function is recursively called with the left child node and the node it was compared to. This recursive calling enables the tree to maintain a left-right balance while placing the new node at the right spot.

4.5 Remove

The "remove" method retrieves and removes the highest-priority element from the tree. It utilizes the "remhelp" method, which also employs recursive calls to navigate the tree and adjust its structure as needed.

4.6 Remhelp

This function is pivotal for the "remove" operation. It works as follows:

Checking for Base Cases:

First, it checks if the current node (root of the tree) is a leaf node with no children (base case). If it is, it sets the root to null, effectively removing the node.

If the node has only one child (either left or right), the function directly replaces the current node with that child and terminates the recursion. This is a crucial step in maintaining the tree structure.

Comparing Child Nodes:

If the node has two children, it compares the values of the left and right children to determine which one has the higher priority (smaller value). Updating Priority:

The function updates the value of the current node with the value of the child having higher priority. This effectively promotes the higher-priority child to the current node's position. Recursive Call:

After updating the current node's value, "remhelp" makes a recursive call to the child node where the higher-priority value was moved. This ensures that the process continues down the tree to address the potential violation of the heap's heap property introduced by the replacement. The recursive nature of "remhelp" allows it to navigate the tree and swap elements efficiently while maintaining the heap property. It ensures that the element with the highest priority is removed while preserving the integrity of the data structure.

4.7 Push

Push Method: The "push" method increases the priority of an element by a specified increment. It modifies the element's value and uses a recursive "pushhelp" method to maintain the tree's structure while considering the priority change. If the priority of the element has been increased, "pushhelp" is called to check if the element's position needs to be updated. Depending on whether the left child, right child, or neither has the highest priority, the element is swapped with the corresponding child, and the process may continue recursively. Each swap occurrence a counter is incremented in order to keep track of the depth traversed.

4.8 Pushhelp

The "pushhelp" method is a recursive function that navigates the tree to adjust the position of an element after a priority increase. It uses conditional statements to determine which child of the current node has the highest priority and swaps the elements accordingly. If neither child has a higher priority, the function terminates since the element is already at the correct position.

5 Array binary tree heap

5.1 Class and Constructor

The "ArrayHeap" class implements a heap using a one-dimensional array. The constructor takes a capacity as an argument and initializes the heap by creating an array with the specified capacity, initializing the size to zero, and setting the "heap" field as an array.

5.2 Insert

The "insert" method adds an element to the heap by placing it at the end of the array, and then using the "bubbleUp" method to ensure the heap property. This method ensures that the parent of the newly inserted element has a lower value, which requires potentially swapping elements. The method employs a while loop to repeatedly compare the new element with its parent, and if the parent's value is greater, they are swapped. This process continues up the tree until the heap property is satisfied.

5.3 Remove

The "remove" method retrieves and removes the smallest element (the root) from the heap. It replaces the root with the last element in the array and then uses the "sinkDown" method to reorganize the heap. The "sinkDown" method ensures that the element at the root is less than or equal to its children. The method employs conditional statements and comparisons to select the smaller child and swap with it if necessary. This process continues down the tree to maintain the heap property.

5.4 Swap

The "swap" method is a helper function that exchanges the positions of two elements within the array, typically used within the "bubbleUp" and "sinkDown" methods to maintain the heap property. It involves simple assignments and doesn't require loops or complex conditionals.

6 Benchmarks

The benchmarks for all implementations was achieved by creating a queue of size 1023 with random elements between 1 and 10000. Subsequently 100 elements where either first removed from the queue, incremented by a random value between 1 and 200 and added back to the queue or simply incremented and pushed down the queue. This was done a total of 100 times and the average time consumption and depth was measured.

7 result

	Time(ns)	Depth
Priority Queue Using a List	2200	
Priority Queue with Reverse Ordering	420	
Linked binary tree add and remove	390	
Linked binary tree push	220	2
Array binary tree	170	

Table 1: Time consumption When removing the element with the highest priority incrementing it and adding it back to the queue or incrementing the element and pushing it down the tree. The time is measured in nanoseconds and the depth refers to how many levels of a tree an element has to be pushed down.

8 Discussion

As can be seen in the result the implementation of a priority queue with a list is much slower than the reversed implementation. This outcome is to be expected for the used benchmarks. This is due to the fact that the removed element was incremented by a number between 1 and 200 while the queue contained 1023 numbers between 1 and 10000. This means that the add method in the first implementation will most likely never have to move as far down the queue as the remove function in the reversed implementation due to the fact that the element being removed could be found anywhere in the list. Perhaps a very different result could have been achieved with different benchmarks, however as of now this theory can't be proven. When analyzing the results for the linked list heap implementation one can see that pushing a element is far Superior compared to removing it, incrementing it and subsequently adding it again. This is to be expected due to the fact we are simply computing less operations when pushing. As anticipated the array stack was the most time efficient implementation, this is due to the fact that it has access to every element in the array at all times.

9 conclusion

in conclusion the report compared different priority queue implementations. The array-based implementation proved itself to be the most efficient. The list-based queues where slower and pushing elements in the linked list binary tree heap proved more efficient than removing and reinserting. Choosing which implementation to use simply depends on the task at hand, but arrays will generally offer the best performance.