A heap or priority queue

Algorithms and data structures ID1021

Johan Montelius

Fall term 2022

Introduction

A heap, or priority queue, is a queue but where the order is determined not by when an item was added to the queue but some other factor. Items with a high priority should be closer to the head of the queue and when removing an item one should always receive the item with the highest priority.

A priority queue is often implemented as a tree structure and is then called a *heap*. In this assignment you will use different implementations and you task is to determine the pros and cons of each.

A list of items

The simplest way of implementing a priority queue is to use a regular list so let's start there and look at the pros and cons.

Implement a priority queue that holds integers, smaller numbers have higher priority. Your first implementation should have an `add` method with a time complexity of $O(1)$ and a `remove` time complexity of $O(n)$ where n is the number of elements in the queue.

Now make a second implementation where the situation is the reversed i.e. it's expensive to add new elements but removal can be don in $O(1)$. Benchmark the two implementations and show the difference.

Are there any situations where you would prefer one over the other?

A very special tree

Having an $O(n)$ time complexity is of course something that we should avoid and the solution is of course to use a tree. We know that operations in trees often can be done in $O(lg(n))$ time and this is also true when we implement a priority queue.

The idea is to use the priority to order the nodes in the tree but if we do as usual and order the tree in a left to right order we might not meet

our goals. Finding the smallest element in an ordered tree is done simply by finding the leftmost node which is easy and we can do this in $O(lg(n))$ time. Adding a new entry is also straight forward and also this operation can be done in $O(lg(n))$ time. It turns out that we can do slightly better than this and that we also have a very efficient implementation to do so.

a heap

A *heap* is a tree where the root node holds the element with the highest priority and both the left and the right branches are heaps. Assume that smaller integers are higher priority then a heap could be a tree with 5 in the root node. The only requirement now is that 5 is the smallest number in the whole tree and that both branches are heaps. If the left branch has a node with the number 8 then we know that all nodes below this node have higher numbers. The right branch could hold any number (higher then 5) and has no specified relationship with the numbers in the left branch. The right node could have 7 or 42 as its value and this would be perfectly OK.

The main advantage of the heap is that we can find the smallest item in $O(1)$ time. If we want to remove it we need to do some work that gives us a $O(lg(n))$ operation but finding it is done in no time.

Adding an element to a tree is at first glance simple. The only thing you need to do is to look at the root node and determine if the new element should be pushed further down the tree or if it should replace the root element and push the root element further down.

Removing a node is equally simple, the element that should be returned is always in the root of the tree (since we implement a priority queue). When this element is removed the question is which of the root elements of the branches should be promoted and of course the one with the highest priority is selected. When this element is promoted it is in similar manner replaced by another element further down the tree etc.

balancing the tree

One problem we will have is to keep the tree balanced. If we always add nodes in the same way the tree will be unbalanced to say the least. We will end up with one long path to the left much like a linked list. In order to keep the tree balanced we can keep track of the total number of elements in each sub-tree and always insert in the branch with the least number of nodes.

Implement a heap as a linked binary tree where each node in the tree holds, apart from the element and the branches, the total number of elements in the sub-tree. When adding an element you should always go down the branch with the fewest elements. When removing an element you do not

have any choice but must of course update the information as you go down the tree.

add

Adding an elements is fairly easy follow these steps and you will be fine. If the root is *null* then of course we simply add a new node to the heap. If not we keep track of the *current node*, the *current value* and work your way down.

- Increment the size by one.
- If current value is less than the value of the current node then swap the values.
- If the left branch is empty then add a new node to the left and return.
- If the right branch is empty then add a new node to the right and return.
- Let the current node be either the left or right node and start again.

remove

Removing an item is slightly more problematic since we have many cases to consider. If the root of the heap is null the we return null (we could throw an exception) but otherwise the value of the root node is the value that we in the end should return. If the size of the root node is 1 then we have removed the last entry in the heap so we set root to null and return the value. Otherwise we keep track of the *current node*, decrement its size by one and do as follows.

- If the left branch is empty then *promote* the right value. If this was the last value in the right branch then replace the branch with null and return, otherwise adopt the right branch as the current node, decrement it's size and continue.
- Otherwise, if the right branch is empty then do the same for the left branch.
- Otherwise, if the right value is less than the value of the left branch, then promote the right value and as before either remove the branch or adopt the right branch as the current node, decrement its size and continue.
- Otherwise, promote the left branch and either move it or adopt is as the current node, decrement its size and continue.

Removing an element will not keep the tree balanced so we might end up with a slightly unbalanced tree. In average the tree should be fairly balanced even if we can find sequences of add and remove operations that will behave very badly (we will soon solve this problem).

increment a element

Apart from finding the smallest element we will be able to push an element further down the tree. This might seam like a strange operation but in the way priority queues are use it turns out to be a quite frequent operation. Often you want to find the element with the highest priority, do some operation in the element, give it a new priority and return it to the queue. In many application the new priority will only be slightly less than its previous priority so the element will only be pushed a few levels down.

Even if the new priority is so low that the element should sink to the bottom of the tree, this is still a cheaper solution than first removing it and then adding it.

Implement a method `push(Integer incr)` that increments the root element by `incr` and then pushes it (by swapping values) either to the left or right branch. For the benchmark the method should return the depth the operation needs to go to.

benchmark

Set up a benchmark where you first add 64 elements with random values (for example [0..100]) to a heap. Then run a sequence of push operations where you increment by a random number (for example [10..20]). Collect some statistics on the depth that the push operation needs to go down to.

Adapt the `add()` method so that it also returns the depth (it will always go down all the way to a leaf). Run the same benchmark but now collect statistics on the add method.

An array implementation

It turns out that an array is very well suited for implementing a heap. It might be that this is the main reason for using a heap when implementing a priority queue.

The trick we will use is that we can represent a binary tree in an array in a quite efficient way if we organize it well. The rule is that a node at position n will have its left and right branch at $n * 2 + 1$ and $n * 2 + 2$. The root of the tree is always at position 0 so the left branch is at 1 and the right branch at 2. The node at position 1 has the left branch at 3 and the right branch at 4.

If the tree that we wan to represent have both short and long branches the array representation will be very sparse i.e. there will be many positions with a null value since the node that should be there does not exist. If the tree is dense i.e. almost all nodes are there, then the array will be almost completely filled up with nodes.

A *complete* tree is a tree where all levels are completely filled apart from the lowest level of leafs. If the lowest level is filled from left to right the tree becomes ideal to represent using an array. This is of course what we want and it turns out that in a heap we can choose to construct the tree in this way.

adding an element - bubble

Assume that we have a heap that is complete tree with the lowest level filled from left to right. The array that represents the tree will have all slots filled up to a position k. When we add an element we simply write this at position $k + 1$, the problem is of course that the tree might not fulfill the requirements of a heap any longer.

The reason why the tree might not be a heap any more is that the newly added node could have a lower value compared to its parent node. If the new node has a higher value then all is fine but if not we have to do something about it.

Assume that n is an even number, let's say 8. All even numbers have their parent node at $(n - 2)/2$. The node at 8 thus has its parent at 3. All odd nodes have their parent at $(n - 1)/2$ so the node at 3 has its parent at 1 and the parent of 1 is a at 0. We can thus trace the nodes from the new leaf to the root of the tree.

When we add a new value we identify the parent node and if the new value is lower we swap the values. The tree might still not be a heap so we repeat this process looking at the newly swapped node and its parent. We will either reach a node whose parent is not greater or the root of the tree and in both cases the tree is again a heap.

removing an element - sink

Removing an element is equally simple, we let a value sink down to its right position. The value that we should return is of course the value of the root node. We remove this value and replace it with the last value in the array (at k). The tree is now probably not a heap since we replaced the smallest element with an element that is found in a leaf.

If we let this value sink i.e. be swapped with either the value of the left or right branch (swapping with the smallest). We will eventually come to a position where it's either a leaf or the branches below it have higher values.

Note that if the last element was at position k before we did the remove operation, the last element is now found at $k - 1$.

Increment a value is done by updating the root value then let it sink to its right position.

benchmark

Implement the heap as an array and run some benchmarks. How does it compare with the implementation as a linked structure?