

# Easiest way of using min priority queue with key update in C++

Asked 7 years, 8 months ago   Active 2 years, 6 months ago   Viewed 32k times



53

Sometimes during programming contests etc., we need a simple working implementation of min priority queue with decrease-key to implement Dijkstra algorithm etc.. I often use `set<pair<key_value, ID> >` and an array (mapping `ID-->key_value`) together to achieve that.



46

- Adding an element to the set takes  $O(\log(N))$  time. To build a priority queue out of  $N$  elements, we simply add them one by one into the set. This takes  $O(N \log(N))$  time in total.
- The element with min `key_value` is simply the first element of the set. Probing the smallest element takes  $O(1)$  time. Removing it takes  $O(\log(N))$  time.
- To test whether some `ID=k` is in the set, we first look up its `key_value=v_k` in the array and then search the element `(v_k, k)` in the set. This takes  $O(\log(N))$  time.
- To change the `key_value` of some `ID=k` from `v_k` to `v_k'`, we first look up its `key_value=v_k` in the array, and then search the element `(v_k, k)` in the set. Next we remove that element from the set and then insert the element `(v_k', k)` into the set. We then update the array, too. This takes  $O(\log(N))$  time.

Although the above approach works, most textbooks usually recommend using binary heaps to implement priority queues, as the time of building the binary heaps is just  $O(N)$ . I heard that there is a built-in priority queue data structure in STL of C++ that uses binary heaps. However, I'm not sure how to update the `key_value` for that data structure.

What's the easiest and most efficient way of using min priority queue with key update in C++?

c++

algorithm

data-structures

edited Mar 20 '17 at 8:24



xuhdev

4,387 1 28 50

asked Feb 9 '12 at 10:40



Chong Luo

431 1 6 7

How about [Boost.Heap](#)? – xuhdev Mar 20 '17 at 8:24

## 3 Answers



43

Well, as Darren already said, [std::priority\\_queue](#) doesn't have means for decreasing the priority of an element and neither the removal of an element other than the current min. But the default `std::priority_queue` is nothing more than a simple container adaptor around a `std::vector` that uses the std heap functions from `<algorithm>` ([std::push\\_heap](#), [std::pop\\_heap](#) and [std::make\\_heap](#)). So for Dijkstra (where you need priority update) I usually just do this myself and use a simple...



By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

```
vec.push_back(item);
std::push_heap(vec.begin(), vec.end());
```

Of course for constructing a queue anew from  $N$  elements, we don't push them all using this  $O(\log N)$  operation (making the whole thing  $O(N \log N)$ ) but just put them all into the vector followed by a simple  $O(N)$

```
std::make_heap(vec.begin(), vec.end());
```

The min element is a simple  $O(1)$

```
vec.front();
```

A pop is the simple  $O(\log N)$  sequence

```
std::pop_heap(vec.begin(), vec.end());
vec.pop_back();
```

So far this is just what `std::priority_queue` usually does under the hood. Now to change an item's priority we just need to change it (however it may be incorporated in the item's type) and make the sequence a valid heap again

```
std::make_heap(vec.begin(), vec.end());
```

I know this is an  $O(N)$  operation, but on the other hand this removes any need for keeping track of an item's position in the heap with an additional data structure or (even worse) an augmentation of the items' type. And the performance penalty over a logarithmic priority update is in practice not that significant, considering the ease of use, compact and linear memory usage of `std::vector` (which impacts runtime, too), and the fact that I often work with graphs that have rather few edges (linear in the vertex count) anyway.

It may not in all cases be the fastest way, but certainly the easiest.

**EDIT:** Oh, and since the standard library uses max-heaps, you need to use an equivalent to `>` for comparing priorities (however you get them from the items), instead of the default `<` operator.

edited Feb 9 '12 at 13:32

answered Feb 9 '12 at 12:16



Christian Rau

40k 9 92 161

---

When you say to *change an item's priority we just need to change it* - do you mean that an  $O(N)$  search is done to find the item within the heap, the item is updated and then the full heap is re-built (at  $O(N)$  again)? – [Darren Engwirda](#) Feb 9 '12 at 12:36

---

@DarrenEngwirda No, I usually have the priority connected to the item anyway (either by being determined from the item directly or by an ID->value array or something the like), which makes updating the priority  $O(1)$ .

- 1 Just because you work with sparse graphs doesn't mean everyone else does. Some of us are doing competitive programming, and the  $O(N)$  and  $O(\log N)$  makes a huge difference. – [nourpyCrazy](#) Jan 18 '18 at 22:41

That's why I didn't claim to provide the fastest and most efficient way for all use-cases. But this certainly is the *easiest* way to do it in standard C++. But even in other use cases it has to be evaluated if some elaborate Fibonacci or whatever heap provides an *actual* performance advantage other than looking cool to a competitive programming jury or in a theoretical complexity analysis. – [Christian Rau](#) Apr 8 '18 at 17:04

- 1 If you're using an  $O(\text{heap size})$  operation on update, shouldn't it become worse than simply re-push elements into heap and throw re-pushed elements when pulling elements from heap? It's  $O(\log(\text{heap size}))$  where heap size is  $O(|E|)$ , which won't exceed  $O(|V|)$ . – [Aplix-Ddr](#) Aug 21 '18 at 14:29



39

Although my response will not answer the original question, I think it could be useful for people who reach this question when trying to implement Dijkstra algorithm in C++/Java (like myself), something that was comment by the OP,



`priority_queue` in C++ (or `PriorityQueue` in Java) do not provide a `decrease-key` operation, as said previously. A nice trick for using those classes when implementing Dijkstra is using "lazy deletion". The main loop of Dijkstra algorithm extracts the next node to be processed from the priority queue, and analyses all its adjacent nodes, eventually changing the cost of the minimal path for a node in the priority queue. This is the point where `decrease-key` is usually needed in order to update the value of that node.

The trick is *not change it* at all. Instead, a "new copy" for that node (with its new better cost) is added into the priority queue. Having a lower cost, that new copy of the node will be extracted before the original copy in the queue, so it will be processed earlier.

The problem with this "lazy deletion" is that the second copy of the node, with the higher bad cost, will be eventually extracted from the priority queue. But that will be always occur after the second added copy, with a better cost, has being processed. So *the very first thing* that the main Dijkstra loop must do when extracting the next node from the priority queue is checking if the node has being previously visited (and we know the shortest path already). It is in that precise moment when we will be doing the "lazy deletion" and the element must be simply ignored.

This solution will have a cost both in memory and time, because the priority queue is storing "dead elements" that we have not removed. But the real cost will be quite small, and programming this solution is, IMHO, easier than any other alternative that tries to simulate the missing `decrease-key` operation.

answered Dec 4 '14 at 22:44



Googol

1,904 16 9

- 2 I considered lazy deletion as well. The problem is future insertions are a function of the size of the existing heap, dead elements included. In some cases the performance could be very bad, other cases not so terrible. – [dromodel](#) Apr 22 '15 at 19:05

@balor123 I'm quite late, but I do wonder: how many elements have you got there?  $O(\log n)$  operations

@domen We need at least  $O(E+V)$  space to represent the original graph, so  $O(E)$  additional elements won't be a problem. – iouvzx Apr 9 '16 at 7:30



I don't think the `std::priority_queue` class allows for an efficient implementation of decrease-key style operations.

19



I rolled my own binary heap based data structure that supports this, basically along very similar lines to what you've described for the `std::set` based priority queue you have:

- Maintain a binary heap, sorted by `value` that stores elements of `pair<value, ID>` and an array that maps `ID`  $\rightarrow$  `heap_index`. Within the heap routines `heapify_up`, `heapify_down` etc it's necessary to ensure that the mapping array is kept in-sync with the current heap position of elements. This adds some extra  $O(1)$  overhead.
- Conversion of an array to a heap can be done in  $O(N)$  according to the standard algorithm described [here](#).
- Peeking at the root element is  $O(1)$ .
- Checking if an `ID` is currently in the heap just requires an  $O(1)$  look-up in the mapping array. This also allows  $O(1)$  peeking at the element corresponding to any `ID`.
- Decrease-key requires an  $O(1)$  look-up in the mapping array followed by an  $O(\log(N))$  update to the heap via `heapify_up`, `heapify_down`.
- Pushing a new item onto the heap is  $O(\log(N))$  as is popping an existing item from the heap.

So asymptotically the runtime is improved for a few of the operations compared with the `std::set` based data structure. Another important improvement is that binary heaps can be implemented on an array, while binary trees are node-based containers. The extra data locality of the binary heap usually translates to improved runtime.

A few issues with this implementation are:

- It only allows integer item `ID`'s.
- It assumes a tight distribution of item `ID`'s, starting at zero (otherwise the space complexity of the mapping array blows up!).

You could potentially overcome these issues if you maintained a mapping hash-table, rather than a mapping array, but with a little more runtime overhead. For my use, integer `ID`'s have always been enough.

Hope this helps.

edited Aug 28 '13 at 19:30

user283145

answered Feb 9 '12 at 11:30



Darren Engwirda


5,820 3 17 38

Thanks! Your method agrees with those described in textbooks. and I believe it has optimal performance.

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

@ChongLuo: Since the mapping array needs to be updated within the `heapify` routines I don't think you can use the `std::` routines, I think you need to write your own. If you were really looking for "the fastest" priority queue for algorithms like Dijkstra's etc it may also be worth checking out some related data structures: Brodal queues, Fibonacci heaps, n-ary heaps, 2-3 heaps to name a few. Some of these data structures are very complex, but offer theoretical improvements to asymptotic complexity... – [Darren Engwirda](#) Feb 9 '12 at 22:38

---

Measuring this on real graphs (road) and comparing with the lazy deletion method would be interesting. I think I can see why the lazy deletion could be faster - you only modify `to_delete` set on decrease and there's a check in extract, only touching one element. Conversely, here you need to modify the (probably larger) map for  $\log(n)$  elements every time you `heapify_down`, which is in extract, decrease and insert. But maybe the heap will blow up due to containing deleted elements takes over. It depends on the number of decreases. Is there data for road networks on decrease op count? – [Adam](#) Aug 24 '18 at 23:08 

---

oh you don't have a `to_delete` set, you just use visited nodes in Dijkstra as the above answer states. So actually it only depends on the number of decreases and the possible blow up – [Adam](#) Aug 25 '18 at 7:04

---

@Adam: Both the `lazy-delete` and `decrease-key` type strategies can be useful I think. Which is better really depends on your particular application. If you're working with very sparse graphs, `lazy-delete` may work well (as the size of the heap won't explode). Denser graphs, on the other hand, are a different story. Imagine you have a graph with mean vertex degree of a few hundred -- `lazy-delete` could lead to a huge space and time blow-up here! Dynamic heaps with a proper `decrease-key` operation lead to  $O(n \log(n))$  Dijkstra-type algorithms. Some other answers here are  $O(n^2)$  ... – [Darren Engwirda](#) Aug 27 '18 at 0:17

---