Min Binary Heap Implementation in C++

Anton Kaminsky, 16 Sep 2014 CPOL

Min Binary Heap Implementation in C++

Introduction

A min binary heap is an efficient data structure based on a binary tree.

A min binary heap can be used to find the `C` (where `C <= n`) smallest numbers out of `n` input numbers without sorting the entire input.

In this tip, I will provide a simple implementation of a min heap using the STL vector.

Background

There are several background topics I need to cover here: binary heap, a binary tree representation using an array and the complexity of constructing the heap.

First the binary heap, a binary heap is a complete binary tree, in which every node is less than its left and right child nodes.

It is easy to see, due to this definition, that the minimum value of the entire heap will always be the root.

The second topic is binary tree representation using an array, basically the rules are as follows:

- The left child of node in index i is: 2*i+1
- The right child of node in index i is: 2*i+2
- The parent of the node in index i is: (int)((i-1)/2)

The root of the tree is in index 0, its left child is in index 1 and its right child is in index 2 and so on.

The third topic is construction of the heap, before explaining how the heap is built, I'll explain two operations, which help performing all the operations available for the heap data structure:

1. `BubbleDown` - `BubbleDown` of a node means: if the node is violating the min heap rule, meaning it is not smaller than both its children, switch it with its minimum child node. Continue doing this recursively until the node is no longer violating the heap rule.
   Each switch of the node lowers its level by 1, so at most there can be log(n) switches in a single `BubbleDown` operation.
2. `BubbleUp` - `BubbleUp` is similar to the `BubbleDown`, only here we test violation of the heap rule between the node and its parent, if such a violation exists, we switch the node with its parent. Once again, continue doing so recursively until the node no longer violates the heap rule. The complexity of the `BubbleUp` is also log(n) since that is the maximum switches that can performed in a single `BubbleUp` operation.

The main operations of the heap are insert and delete min, here is the algorithm for insert:

- Insert the new value at the last position in the array (which means adding a new leaf in the lowest level of the tree). The only violation of the heap can be at the last leaf.
- Perform `BubbleUp` operation of the new leaf. This will fix all violations of the heap.

Therefore, insert operation complexity is log(n).

The algorithm for the delete min operation:

- Remove the root node.
- Move the value in the last index of the array (the last leaf in the lowest level of the tree) to the root position (index 0 in the array). The only violation of the heap is at the root.
- Decrease the array size by 1.
- Perform `BubbleDown` operation on the root. This will fix all violations of the heap.

Therefore, delete min operation complexity is also log(n).

Now it is finally time to talk about heap construction (also called heapify):

The trivial way to construct the heap would be to call insert on every element of the array, this will give us n*log(n) complexity.

There is a much more efficient way to construct the heap:

- Go over all elements of the array from last to first.
- For each element, call `BubbleDown` on the element.

The complexity of this construction in O(n) - Linear, you can see an intuitive and mathematical explanation here: https://www.youtube.com/watch?v=d3qd_wQdYqg.

That's all for the background section, we now have everything we need to implement our heap.

Since I wanted my heap to have a dynamic number of elements, I used the STL vector instead of the array.

Using the Code

The example code was written using Visual Studio 2012.

Here is our *h* file:

Hide Copy Code

```
#include "vector"
using namespace std;

class MinHeap
{
private:
    vector<int> _vector;
    void BubbleDown(int index);
    void BubbleUp(int index);
    void Heapify();

public:
    MinHeap(int* array, int length);
    MinHeap(const vector<int>& vector);
    MinHeap();

    void Insert(int newValue);
    int GetMin();
    void DeleteMin();
};
```

Here is the *cpp* file:

Hide Shrink ▲ Copy Code

```
#include "stdafx.h"
#include "MinHeap.h"

MinHeap::MinHeap(int* array, int length) : _vector(length)
{
    for(int i = 0; i < length; ++i)
    {
        _vector[i] = array[i];
    }

    Heapify();
}

MinHeap::MinHeap(const vector<int>& vector) : _vector(vector)
{
    Heapify();
}

MinHeap::MinHeap()
{
}

void MinHeap::Heapify()
{
    int length = _vector.size();
    for(int i=length-1; i>=0; --i)
    {
        BubbleDown(i);
    }
}

void MinHeap::BubbleDown(int index)
{
    int length = _vector.size();
    int leftChildIndex = 2*index + 1;
    int rightChildIndex = 2*index + 2;

    if(leftChildIndex >= length)
        return; //index is a leaf

    int minIndex = index;

    if(_vector[index] > _vector[leftChildIndex])
    {
        minIndex = leftChildIndex;
    }

    if((rightChildIndex < length) && (_vector[minIndex] > _vector[rightChildIndex]))
    {
        minIndex = rightChildIndex;
    }

    if(minIndex != index)
    {
        //need to swap
        int temp = _vector[index];
        _vector[index] = _vector[minIndex];
        _vector[minIndex] = temp;
        BubbleDown(minIndex);
    }
}

void MinHeap::BubbleUp(int index)
{
    if(index == 0)
        return;

    int parentIndex = (index-1)/2;

    if(_vector[parentIndex] > _vector[index])
    {
        int temp = _vector[parentIndex];
        _vector[parentIndex] = _vector[index];
        _vector[index] = temp;
        BubbleUp(parentIndex);
    }
}

void MinHeap::Insert(int newValue)
{
    int length = _vector.size();
    _vector[length] = newValue;

    BubbleUp(length);
}

int MinHeap::GetMin()
{
    return _vector[0];
}

void MinHeap::DeleteMin()
{
    int length = _vector.size();

    if(length == 0)
    {
        return;
    }

    _vector[0] = _vector[length-1];
    _vector.pop_back();

    BubbleDown(0);
}
```

Here is an example of a main, which finds the 3 smallest numbers of an array:

Hide Copy Code

```
int _tmain(int argc, _TCHAR* argv[])
{
    int array[] = {10, 4, 5, 30, 3, 300};

    MinHeap minHeap(array, 6);

    for(int i=0; i<3; ++i)
    {
        cout << minHeap.GetMin() << "  ";
        minHeap.DeleteMin();
    }

    char x;
    std::cin >> x;
    return 0;
}
```

Points of Interest

The heap data structure has some very useful usages:

- Since we can efficiently extract the min number each time (can also be the max number in a max heap), the heap can be used to implement a priority queue, which always give us the item with the highest priority currently in the heap.
- Another usage of the heap is finding the smallest C numbers out of n numbers, the complexity of doing this with the heap is n + C*log(n) (n for constructing the heap and log(n) for each of the C numbers we want to find). This is a question asked in job interviews occasionally.
- If we want to sort the n numbers, using the heap min we can do it in n+n*log(n) time, that is called the heap sort.

Regarding the second point, if C is a constant, we can use the brute force approach as follows:

- Prepare an array\vector of size C, call it `temp`.
- For each number in `n:`, place it in the correct order in the `temp` array.
- At the end of the loop, the `temp` will hold the C smallest numbers from `n`.

The complexity of this approach is: we pass the `n` numbers once, and for each number we find its place in the array of size C which is O(C) complexity, so total complexity is C*n.

If C is smaller than log(n), the brute force approach is better, however we do need to know C in advance. We also need to allocate a buffer with size C (O(C) space complexity).

The heap solution does not require knowing C in advance, we just keep calling "`GetMin` & `DeleteMin`" as many times as we like, and the complexity is always log(n). Regarding space complexity, while the code example I provided copies all `n` numbers, the code can easily be modified to create the heap in place and not copied to another buffer, therefore the heap solution doesn't require any further space allocation.

History

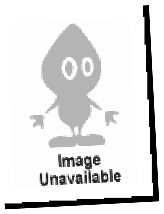
- 16th September, 2014: Initial version

License

Share

EMAIL

About the Author



Anton Kaminsky

Israel

No Biography provided

Comments and Discussions

You must Sign In to use this message board.

Search Comments Go

Profile popups Spacing Relaxed Noise Medium Layout Normal Per page 25 Update

First Prev Next

An FYI	**David O'Neil**	**6-Jan-15 16:10**
Difference to std::map	**_kb_**	**17-Sep-14 8:46**
Re: Difference to std::map	Anton Kaminsky	17-Sep-14 10:01

Last Visit: 31-Dec-99 19:00 Last Update: 24-Mar-15 14:52 Refresh 1

General News Suggestion Question Bug Answer Joke Rant Admin

Use Ctrl+Left/Right to switch messages, Ctrl+Up/Down to switch threads, Ctrl+Shift+Left/Right to switch pages.