Heap Sort Algorithm

# Overview

This document describes approach and implementation of a generic heap sort algorithm. A heap sort is a type of comparative sort that relies on a heap data structure. In order to perform ascending or descending sort we need to respectively use a min or a max heap data structure. The sort can be achieved with the opposite heap as well but in reverse order. This will be explained further in this document. The heap sort is a sorting algorithm of ***O(n\*log[n])*** complexity. That means that sorting an array of size ***n*** takes at most order of ***n\*log[n]*** operations. This is the worst case scenario for this algorithm. Note that for a sorted array this algorithm would be of ***O(n)***. The complexity comes mostly from the ***insert*** and ***heapify*** operations of the heap.

# Description

The heap sort algorithm can be divided into two parts:

1. Building a Heap from the unsorted array.
2. Extracting the max or the min element from the heap until we have a sorted array.

Building a max heap data structure from an array ensures that we have the largest element on top of the heap (0-index) when the heap is built. Correspondingly building a min heap ensures that we have the smallest element on top when the heap is built. Then the heap performs heapify down after deleting the top element and puts the next largest/smallest elements on top. Essentially the heap performs the sorting for us and we rely on the attribute of the max or min heap to keep max or min element on top.

The insert operation inside the heap has the complexity of ***O(log[n])*** and we’re inserting n elements in it. Thus the complexity for inserting the entire array is of ***O(n\*log[n])*** as mentioned above. Each delete operation is also of ***O(log[n])***. We have n delete operations thus again ***O(n\*log[n])***. The total complexity calculation is giving us ***n\*log[n] + n\*log[n]*** or ***2n\*log[n]*** which is ***O(n\*log[n])***.

# Implementation

In order to perform ascending sort on an unsorted array of size n we perform the following operations:

1. We instantiate a new min heap of maximum size larger than n.

Heap<int> heap = Heap<int>();

1. We loop through each value of the input array and we perform insert inside the heap.

for (int i = 0; i < max\_size; i++) {

heap.Insert(new\_array[i]);

}

1. We delete at index 0 of the heap until the heap is empty and populate the deleted value inside a new array in ascending order.

while (!heap.IsEmpty()) {

sorted[index++] = heap.DeleteByIndex(0);

}

1. The new array we populated in step 3 is sorted in an ascending order.

# Heap Implementation

For the heap implementation I would be showing the basic *Max Heap* ***insert*** and ***heapify*** methods. The insert method of the heap inserts and item at the end of the heap container and then ensures that the heap is heapified in such a way that the largest element ends up on top, or at index 0.

/// <summary>

/// Insert at the last position and <see cref="Heap{Type}::HeapifyUp"/>

/// </summary>

/// <param name="item">The item.</param>

template <class Type>

void Heap<Type>::Insert(Type item) {

container[size] = item;

HeapifyUp(size);

size++;

}

The heapify-up method is implemented to swap the element values with its parent’s value if the element has a larger value than its parent:

/// <summary>

/// Heapifies up.

/// </summary>

/// <param name="index">The index.</param>

template <class Type>

void Heap<Type>::HeapifyUp(int index) {

if (index > 0) {

int parent = index / 2;

if (container[index] > container[parent]) {

Type temp = container[parent];

container[parent] = container[index];

container[index] = temp;

HeapifyUp(parent);

}

}

}

The delete operation is similar but in the opposite direction. We delete at a specified index inside the heap which means to cache the value at the index and then to replace it with the last value in the heap. The size of the heap is decremented and the heap performs heapify from the index that we delete at. At the end we return the cached value to provide what was deleted:

/// <summary>

/// Delete at index and <see cref="Heap{Type}::HeapifyDown"/>.

/// </summary>

/// <param name="index">The index.</param>

/// <returns>The value of the deleted item</returns>

template <class Type>

Type Heap<Type>::DeleteByIndex(int index) {

int temp = container[index];

container[index] = container[--size];

HeapifyDown(index);

return temp;

}

Which requires us to also show the heapify-down method which swaps the values of the element and the larger of its children:

/// <summary>

/// Heapifies down.

/// </summary>

/// <param name="index">The index.</param>

template <class Type>

void Heap<Type>::HeapifyDown(int index) {

int child = index \* 2 + 1;

if (child + 1 < size) {

child = container[child] < container[child + 1] ? child + 1 : child;

if (container[child] > container[index]) {

Type temp = container[child];

container[child] = container[index];

container[index] = temp;

HeapifyDown(child);

}

} else if (child < size && container[child] > container[index]) {

Type temp = container[child];

container[child] = container[index];

container[index] = temp;

HeapifyDown(child);

}

}

This implementation provides us with the functionality to be able to retrieve a sorted array in descending order by removing from the heap at its 0 index until the heap is empty.