# Data Structures/Min and Max Heaps

**Data Structures**
Introduction - Asymptotic Notation - Arrays - List Structures & Iterators
Stacks & Queues - Trees - Min & Max Heaps - Graphs
Hash Tables - Sets - Tradeoffs

Some definitions of a heap follow:

A heap is an array , where there are parent child relationships, and the index of a child is 2 * parent index, or 2* parent index + 1 , and a child has an *order* after the parent, in some concrete ordering scheme injected by a client program of a heap. There is importance in maintaining the ordering n variant after the heap is changed. Some ( Sedgewick and Wayne), have coined the term "swim" and "sink", where maintenance of the invariant involves a invariant breaking item swimming up above children of lower ordering, and then sinking below any children of higher ordering, as there may be two children , so one can swim above a lower ordered child, and still have another child with higher ordering.

A heap is an efficient semi-ordered data structure for storing a collection of orderable data. A min-heap supports two operations:

```
INSERT(heap, element)
element REMOVE_MIN(heap)
```

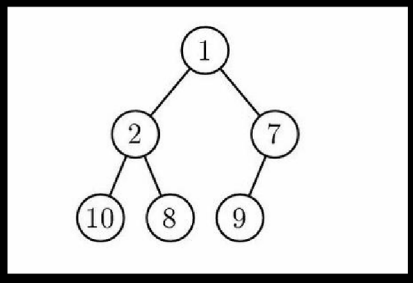
(we discuss min-heaps, but there's no real difference between min and max heaps, except how the comparison is interpreted.)

This chapter will refer exclusively to binary heaps, although different types of heaps exist. The term binary heap and heap are interchangeable in most cases. A heap can be thought of as a tree with parent and child. The main difference between a heap and a binary tree is the heap property. In order for a data structure to be considered a heap, it must satisfy the following condition (heap property):

If $A$ and $B$ are elements in the heap and $B$ is a child of $A$, then $\text{key}(A) \leq \text{key}(B)$.

(This property applies for a min-heap. A max heap would have the comparison reversed). What this tells us is that the minimum key will always remain at the top and greater values will be below it. Due to this fact, heaps are used to implement priority queues which allows quick access to the item with the most priority. Here's an example of a min-heap:

A heap is implemented using an array that is indexed from 1 to N, where N is the number of elements in the heap.




At any time, the heap must satisfy the heap property

```
array[n] <= array[2*n]   // parent element <= left child
```

and

```
array[n] <= array[2*n+1] // parent element <= right child
```

whenever the indices are in the arrays bounds.

## Compute the extreme value

We will prove that `array[1]` is the minimum element in the heap. We prove it by seeing a contradiction if some other element is less than the first element. Suppose `array[i]` is the first instance of the minimum, with `array[j] > array[i]` for all `j < i`, and `i >= 2`. But by the heap invariant array, `array[floor(i/2)] <= array[i]`: this is a contradiction.

Therefore, it is easy to compute `MIN(heap)`:

```
MIN(heap)
   return heap.array[1];
```

## Removing the Extreme Value

To remove the minimum element, we must adjust the heap to fill `heap.array[1]`. This process is called *percolation*. Basically, we move the hole from node i to either node `2i` or `2i+1`. If we pick the minimum of these two, the heap invariant will be maintained; suppose `array[2i] < array[2i+1]`. Then `array[2i]` will be moved to `array[i]`, leaving a hole at `2i`, but after the move `array[i] < array[2i+1]`, so the heap invariant is maintained. In some cases, `2i+1` will exceed the array bounds, and we are forced to percolate `2i`. In other cases, `2i` is also outside the bounds: in that case, we are done.

Therefore, here is the remove algorithm for min heap:

```
#define LEFT(i) (2*i)
```

```
#define RIGHT(i) (2*i + 1)
```

```
REMOVE_MIN(heap)
{
    savemin=arr[1];
    arr[1]=arr[--heapsize];
    i=1;
    while(i<heapsize){
        minidx=i;
        if(LEFT(i)<heapsize && arr[LEFT(i)] < arr[minidx])
```

```
            minidx=LEFT(i);
        if(RIGHT(i)<heapsize && arr[RIGHT(i)] < arr[minidx])
            minidx=RIGHT(i);
        if(minidx!=i){
            swap(arr[i],arr[minidx]);
            i=minidx;
        }
        else
            break;
    }
}
```

Why does this work?

```
   If there is only 1 element ,heapsize becomes 0, nothing in the array is valid.
   If there are 2 elements , one min and other max, you replace min with max.
   If there are 3 or more elements say n, you replace 0th element with n-1th element.
   The heap property is destroyed. Choose the 2 children of root and check which is the minimum.
   Choose the minimum between them, swap it. Now subtree with swapped child is loose heap property.
   If no violations break.
```

# Inserting a value into the heap

A similar strategy exists for INSERT: just append the element to the array, then fixup the heap-invariants by swapping. For example if we just appended element N, then the only invariant violation possible involves that element, in particular if , then those two elements must be swapped and now the only invariant violation possible is between

```
array[floor(N/4)]  and  array[floor(N/2)]
```

we continue iterating until N=1 or until the invariant is satisfied.

```
INSERT(heap, element)
   append(heap.array, element)
   i = heap.array.length
   while (i > 1)
     {
       if (heap.array[i/2] <= heap.array[i])
         break;
       swap(heap.array[i/2], heap.array[i]);
       i /= 2;
     }
```

# TODO

```
Merge-heap: it would take two max/min heap and merge them and return a single heap. O(n) time.
Make-heap: it would also be nice to describe the O(n) make-heap operation
Heap sort: the structure can actually be used to efficiently sort arrays
```

Make-heap would make use a function heapify

```
//Element is a data structure//
   Make-heap(Element Arr[],int size)
   {
       for(j=size/2;j>0;j--)
           {
               Heapify(Arr,size,j);
           }
   }
```

```
   Heapify(Element Arr[],int size,int t)
   {
       L=2*t;
       R=2*t+1;
       if(L<size )
       {
           mix=minindex(Arr,L,t);
           if(R<=size)
               mix=minindex(Arr,R,mix);
       }
       else
           mix=t;
       if(mix!=t)
       {
           swap(mix,t);
           Heapify(Arr,size,mix);
       }
   }
```

minindex returns index of the smaller element

# Applications of Priority Heaps

In 2009, a smaller Sort Benchmark was won by OzSort, which has a paper describing lucidly how to use a priority heap as the sorting machine to produce merged parts of large (internally) sorted sections . If a sorted section took M memory and the sorting problem was k x M big, then take sequential sections of each of the k sections of size M/k , at a time, so they fit in M memory ( k * M/k = M ), and feed the first element of each of k sections to make a k sized priority queue, and as the top element is removed and written to an output buffer, take the next element from the corresponding section. This means elements may need to be associated with a label for the section they come from. When a M/k-sized section is exhausted, load in the next M/k sized minisection from the original sorted section stored on disc. Continue until all minisections in each of the k sections on disc have been exhausted.

(As an example of pipelining to fill up disc operational delays, there are twin output buffers, so that once an output buffer is full one gets written the disc while the other is being filled.)

This paper showed that a priority heap is more straightforward than a binary tree, because elements are constantly being deleted, as well as added, as a queuing mechanism for a k way merge, and has practical application for sorting large sets of data that exceed internal memory storage.

**Data Structures**
Introduction - Asymptotic Notation - Arrays - List Structures & Iterators
Stacks & Queues - Trees - Min & Max Heaps - Graphs
Hash Tables - Sets - Tradeoffs

---