The beautiful binary heap.

Weiss has a chapter on the binary heap - chapter 20, pp581-601.

It's a very neat implementation of the binary tree idea, using an array.

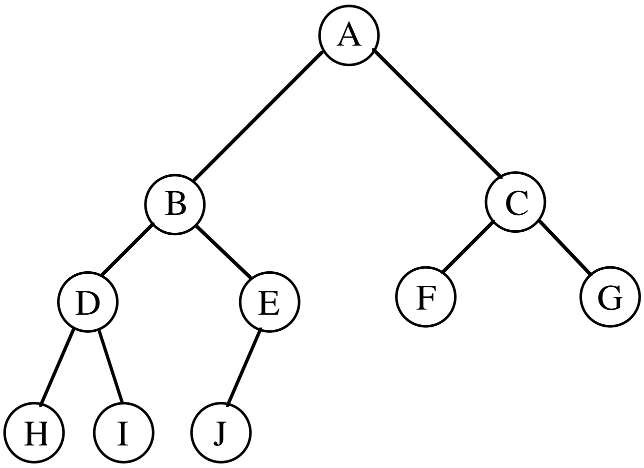
We can use the binary heap to sort an array, and we can use it to make a priority queue (see the Dijkstra and A* algorithms above).

Used for sorting, we get $O(N \lg N)$ performance and we use $O(1)$ space.

Used as a priority queue, we can guarantee $O(\lg N)$ insert / delete performance - far better than an ordered list or binary chop insert / delete.

Richard Bornat
Dept of Computer Science
QUEEN MARY
AND WESTFIELD COLLEGE
UNIVERSITY OF LONDON

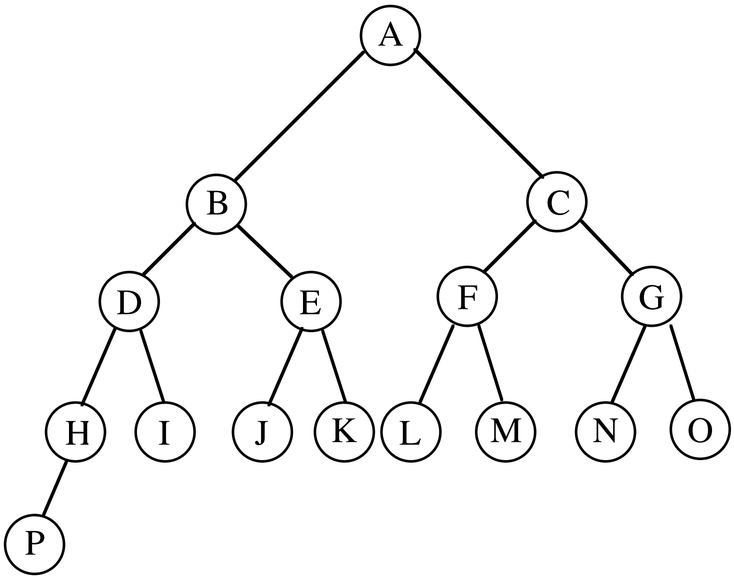
We abandon a distinction between leaves and internal nodes of our binary trees. We read the trees as if they were written in rows (this tree has four rows)



and define a ***complete binary tree*** as one in which every row is filled, except possibly the last, and the last is filled from left to right.

When we add a node to a complete binary tree there is always only one place where it can go: in the picture above it must be the missing right child of *E*. The next node would be added as the left child of *F*, and so on.

Once the fourth row is filled we would start on the fifth and add a left child for *H*:



If we delete a node from a complete binary tree it must be the rightmost node in the last row – to preserve the complete binary tree property.

The reason for the complete binary tree property is that we can write a cbt in an array. The root goes in position 1, the next row in positions 2 and 3, the next in 4-7, the next in 8-15, and so on. We keep a record of the limit of the tree:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
	A	B	C	D	E	F	G	H	I	J									

Richard Bornat
Dept of Computer Science
QUEEN MARY AND WESTFIELD COLLEGE UNIVERSITY OF LONDON

The effect is that the children of the node in position i are always found in position $2i$ and $2i+1$ (provided that those are within the limit of the tree); conversely the parent of the node in position j is always found in position $j \div 2$ (apart from the root, which has no parent).

This gives binary-tree manipulation 'for free', without the overhead of links, references, pointers, whatever.

This array version of a complete binary tree is the ***binary heap***.

I illustrate the examples by using complete binary trees, but write code using the binary heap.

```
class Element { public Key k; public Data d; }

class BinaryHeap {
  private Element[] H; int count; // number of things in the heap
  ...
}
```


Richard Bornat
Dept of Computer Science
QUEEN MARY
AND WESTFIELD COLLEGE
UNIVERSITY OF LONDON

Ordered binary heaps.

In an ordered binary heap, each node comes before (≤) its children, each child comes after its parents.

$$\forall i\left(2 \le i < count \rightarrow H_{i \div 2} \le H_i\right)$$

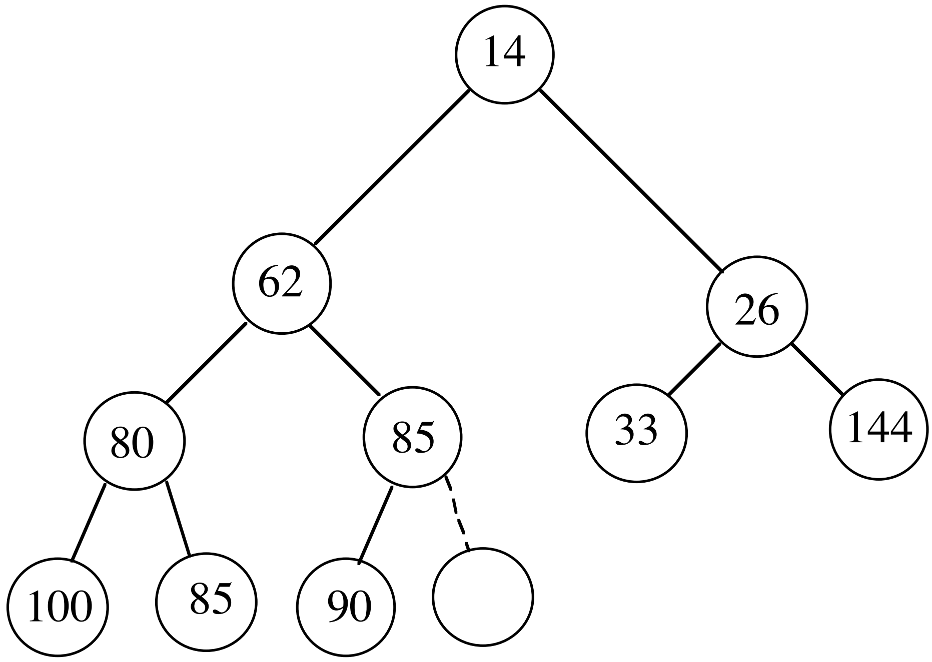
That's the only condition we need – unlike binary dictionary trees, we don't have any ordering between the children or between subtrees.

It follows from this definition that the root of an ordered binary heap is a minimum element of the heap. Access to the minimum element is therefore very fast: $O(1)$ because it is always found in H_1.

Insertion into an ordered binary heap.

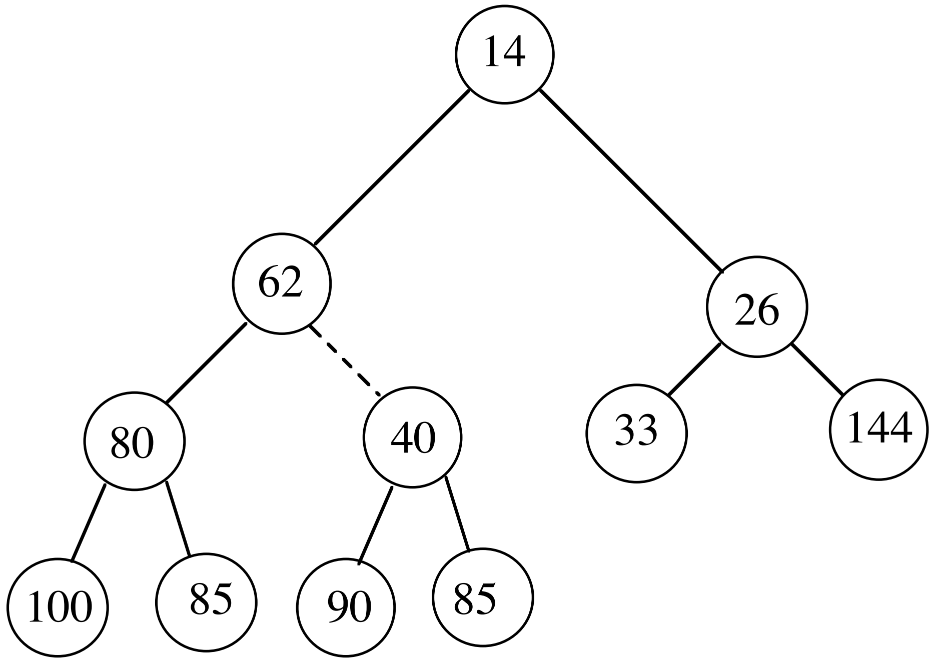
Insertion turns out to be $O(\lg N)$. It has to preserve the ordering property, but that turns out to be easy.

Consider this ordered binary heap, and the problem of inserting a new element into it (at the position marked with an empty circle):

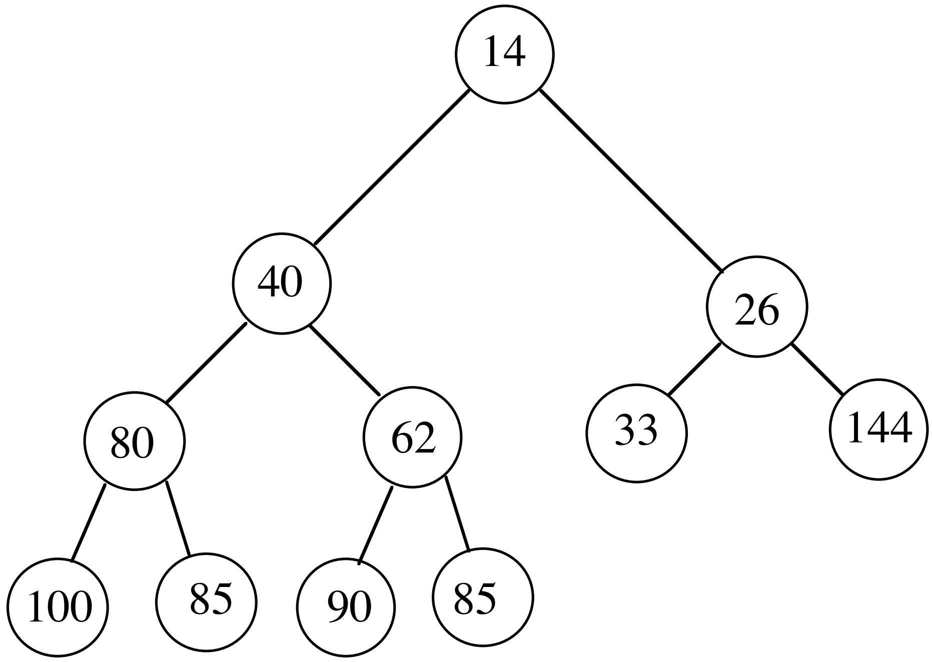


We can insert large numbers (85≤) without moving anything.

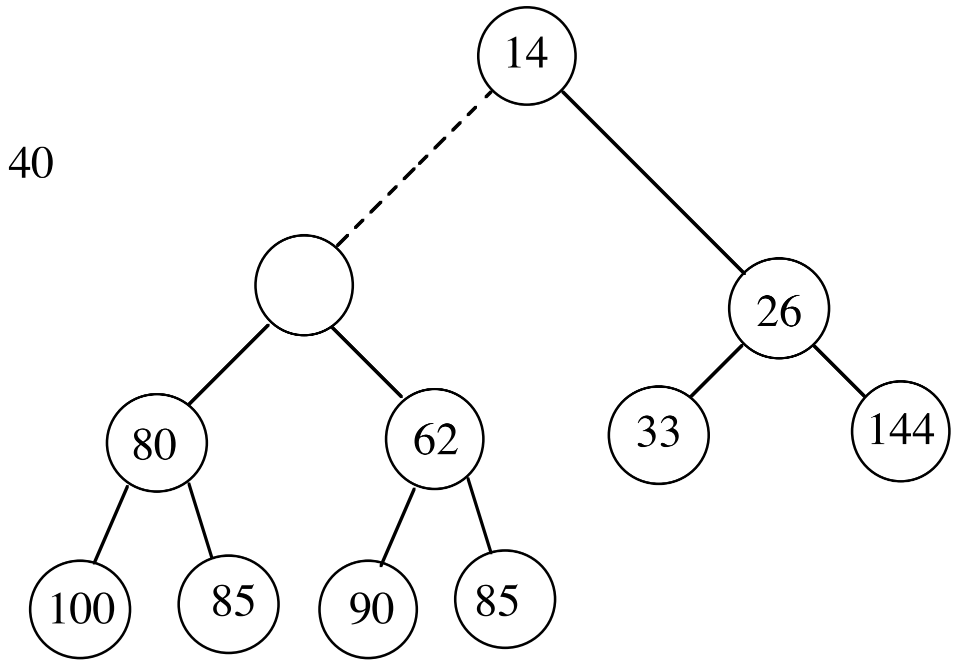
But if we insert 40 at that position, then heap order is destroyed: we must exchange it with its parent:



and then exchange again before heap order is restored:



We can save a bit of time when we realise that we don't have to do any exchanges, just slide things around till the hole is in the right place:



This is called 'bubbling up' a hole in the structure:

```
public void insert(Key k, Data d) {
  count++; bubbleup(k,d,count); // H[1..count] are valid elements
}

private void bubbleup(Key k, Data d, int i) {
  int j;
  for (; j=i/2, i!=1 && !H[j].key.lesseq(k); i=j)
    H[i]=H[j];
  H[i].key=k; H[i].data=d;
}
```

Clearly this takes $O(\lg N)$ time: it halves i repeatedly until it is 1 or it finds a position where H_i's parent is $(\leq k)$. In the worst case this can only take $\lg N$ steps.

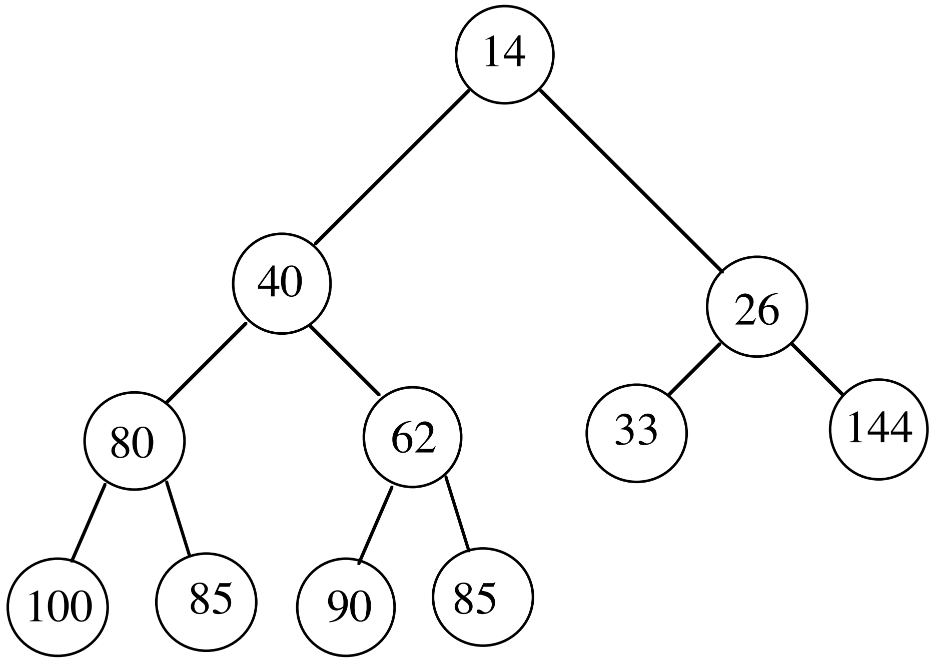
We can make it faster still by the method of sentinels if we put a value $-\infty$ in H_0: then we can avoid the `i!=1` test. It's easy to do that with a special Key value:

```
class Stopper implements Key {
  public boolean equals(Key k) { return k instanceof Stopper; }
  public boolean lesseq(Key k) { return true; }
}
```

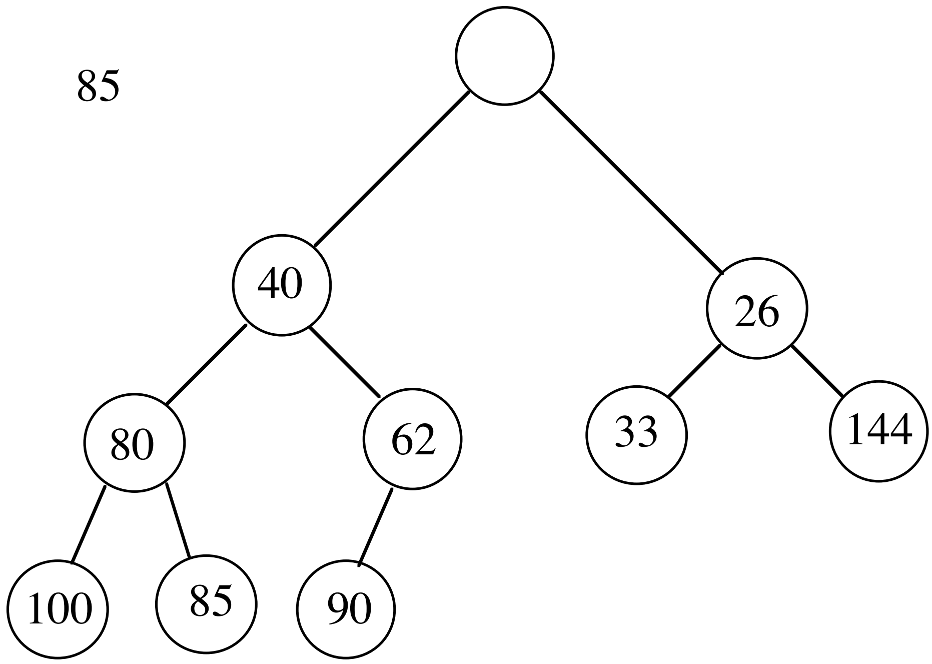
Deleting from an ordered binary heap.

We want, both when sorting and when running a priority queue, to delete the element in H_1. But to keep the heap ordered we actually have to delete the rightmost element on the last row.

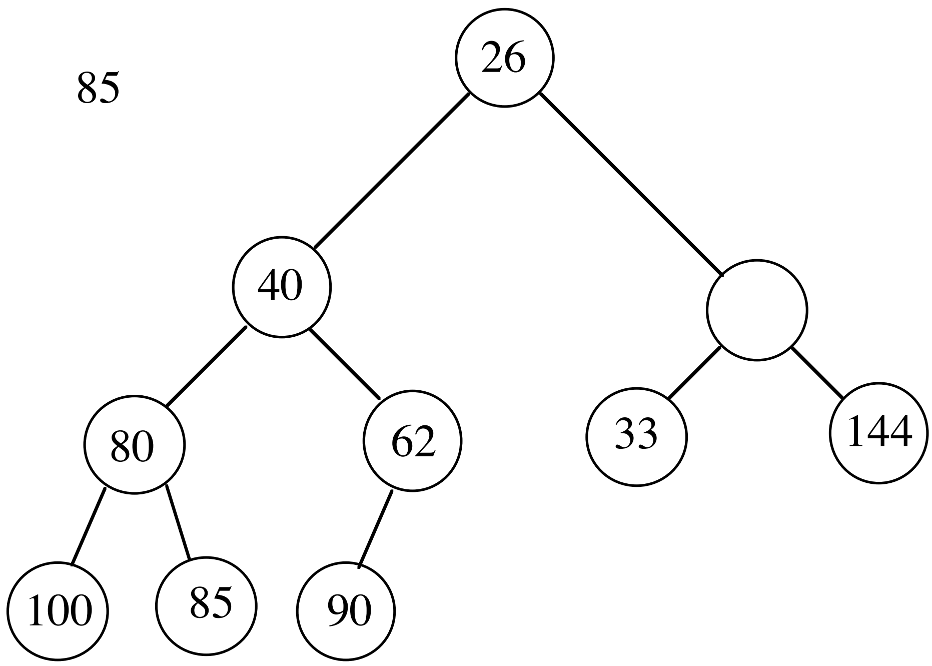
The minimum element in this tree is 14:



We begin by taking away that minimum element, leaving a hole; Then we delete the rightmost element on the last row and try to see if it will fit in the hole:

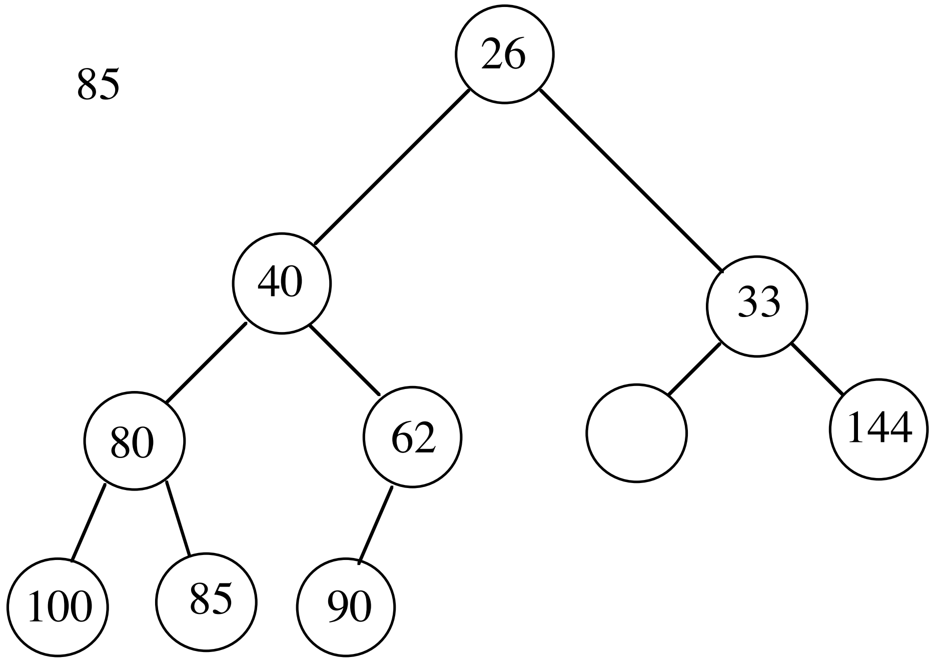


If it won't, we slide the smallest of the children into the hole and try again:



Richard Bornat
Dept of Computer Science

Yet another slide needed:



Now we can put 85 into the hole, and the heap order is restored. This is called ‘bubbling down’:

```
public Data remove() {
  Data d = H[1].d;
  H[1]=H[count--]; bubbledown(1);
  return d;
}

public void bubbledown(int i) {
  Element r = H[i]; int j;
  for (; j=i*2, j<=count; i=j) {
    if (j+1<=count && H[j+1].k.lesseq(H[j].k))
      j++; // pick left or right child
    if (r.k.lesseq(H[j].k)) break; // exit if in order
    else H[i]=H[j];
  }
  H[i]=r;
}
```


Richard Bornat
Dept of Computer Science
QUEEN MARY AND WESTFIELD COLLEGE UNIVERSITY OF LONDON

This is obviously $O(\lg N)$ because it works by repeatedly doubling until it reaches *count* or a position where element r can be assigned whilst preserving heap order.

Now in the Dijkstra algorithm and the A* algorithm we have to use a priority queue: a binary heap does the job perfectly, with optimal efficiency: $O(\lg N)$ insertion, $O(\lg N)$ removal.

Sorting using an ordered binary heap.

We can impose heap order on an unsorted array in $O(N)$ time, by using $N \div 2$ calls of *bubbledown*:

```
for (int i=count/2; i>0; i--) bubbledown(i);
```

Weiss shows (p596, p611) that the sum of the heights of the nodes in a complete binary tree is $O(N)$; it follows that this code must execute in $O(N)$ time.

Once we have heap order, we can extract the minimum element. That moves the last element into position into the tree and leaves an empty slot; we can move the old root element into that slot. It takes $O(\lg N)$ time to do that:

```
Element r = H[1]; remove(); H[count+1]=r;
```

Repeat that N times and we have a sorting algorithm, because $H_{1..count}$ will be in the opposite order to the heap order (minimum element last, maximum element first).

This is an $O(N \lg N)$-time sort which uses $O(1)$ space!

Richard Bornat
Dept of Computer Science
QUEEN MARY AND WESTFIELD COLLEGE UNIVERSITY OF LONDON

You need some arithmetic trickery to deal with the fact that an array starts at position 0 but a heap starts at position 1, and you might want to reverse the notion of heap order to make your result come out in $(\leq)$ order ...

$$\boxed{\text{Truly the binary heap is a wonderful data structure, and its algorithms are wonderful too.}}$$

Richard Bornat
Dept of Computer Science
QUEEN MARY AND WESTFIELD COLLEGE UNIVERSITY OF LONDON