```
template <typename T>
class PriorityQueue {
public:
        PriorityQueue();
        void enqueue(const T& value);
        bool dequeue(T& value);
        bool isEmpty() const;
        void makeEmpty();
private:
        T elements[20];              // Could also use
                                     // a dynamic array
        int size;
        void heapify(int i);   // Restructures the heap when an
                               // element is removed by 'dequeue'

};
```

```
template <typename T>
bool PriorityQueue<T>::enqueue(const T& value)
{
        bool added = false;
        int child = size;
        int parent = (child – 1)/2;
        while ((child > 0) && (value > elements[parent])) {
                elements[child] = elements[parent];
                child = parent;
                parent = (parent – 1)/2;
        }
        elements[child] = value;
        ++size;
        added = true;
        return added;
}
```

```
template <typename T>
bool PriorityQueue<T>::dequeue(T& value)
{
        bool removed = false;
        if (size > 0) {
                value = elements[0];
                removed = true;
                elements[0] = elements[size-1];
                --size;
                heapify(0);       // Re-structure from element 0
        }                         // Indicating the whole array is to be
        return removed;           // re-structured
}
```

```
template <typename T>
void PriorityQueue<T>::heapify(int i)
{
        int leftChild;
        int rightChild;
        int largest;
        bool stop = false;
        T temp = elements[i];
        leftChild = i*2 + 1;
        while ((leftChild < size) && !stop) {
                rightChild = leftChild + 1;
                if (rightChild >= size) {
                        largest = leftChild;
                }
                else {
                        if (elements[leftChild] >
                            elements[rightChild]) {
                                largest = leftChild;
                        }
                        else {
                                largest = rightChild;
                        }
```

```
            }
            if (elements[largest] > temp) {
                elements[i] = elements[largest];
                i = largest;
                leftChild = i * 2 + 1;
            }
            else {
                stop = true;
            }
        }
        elements[i] = temp;
    }
```