```
/************************/
/* TREVER WAGENHALS     */
/* PROGRAM 9            */
/* December 6, 2017     */
/************************/

#ifndef HEAP_H
#define HEAP_H

#include <iostream>
#include <climits>
#include <cassert>
#include "CursorCntl.h"
template <typename ElemData, unsigned Capacity>
class Heap
{
public:
    // Constructor
    Heap() : size(0), current(0) { }
    // Return the number of elements in the array.
    unsigned Size() { return size; }
    // Return true if the array is empty.
    bool Empty() { return size == 0; }
    // Return true if the array is full.
    bool Full() { return size >= Capacity; }
    // Insert a new element into the array properly stored in ascending order..
    void Insert(ElemData &data);
    // Perform Heap Sort to sort the array into ascending order.
    void Sort();
    // Call "BinSearch()" to search the sorted array for the entry "data".
    // If found, make this the current entry and return true;
    // otherwise, return false.
    bool Search(ElemData &data);

    // Perform a binary search for "data". Search the index range from
    // "start" to "end". If the item is found, make it the current item and return true.
    // Otherwise, return false.
    bool BinSearch(unsigned start, unsigned end, ElemData &data);

    // Output the array to the stream "os".
    void Output(ostream &os);
    // Show the heap on the right side of the screen.
    void ShowTree() const;
    // Return the current entry.
    ElemData CurrentEntry() { return heap[current]; }
   // Update the current entry.
   void Update() { assert(current != 0); heap[current].Update(); }

    // Standard heap operations
    void PercolateUp();
    void DeleteMax();
    void PercolateDown(unsigned r, unsigned n);
    void Heapify();

private:
    unsigned size;              // The number of items in the heap
    unsigned current;           // The index of the entry found by the last search
    ElemData heap[Capacity+1];  // The heap array

    // Recursive function to show the tree graphics
    void RShowTree(unsigned r, int x, int y) const;
};
```

```
66  const unsigned XRoot = 40;        // Column number for root node
67
68  // Recursive function to display a tree on the right half of the screen
69  // using (crude) character graphics.
70  template <typename ElemData, unsigned Capacity>
71  void Heap<ElemData, Capacity>::RShowTree(unsigned r, int x, int y) const
72  {
73    const unsigned VertSpacing = 7;   // Vertical spacing constant
74    const unsigned HorizSpacing = 10; // Horizontal spacing of tree nodes
75    const unsigned MaxLevels = 4;     // The number of levels that fit on the screen
76
77    // If the tree is not empty display it..
78    if (r <= size && x < MaxLevels)
79      {
80      // Show the left sub-tree.
81      RShowTree(2*r, x+1, y+VertSpacing/(1<<x));
82      // Show the root.
83      gotoxy(XRoot+HorizSpacing*x, y);
84
85      ElemData wc = heap[r];
86
87      wc.Show(cout);
88
89      // Show the right subtree.
90      RShowTree(2*r+1, x+1, y-VertSpacing/(1<<x));
91      }
92  }
93
94  // Display a tree on the right half of the screen using (crude)
95  // character graphics. This function calls RShowTree() which does
96  // the work.
97  template <typename ElemData, unsigned Capacity>
98  void Heap<ElemData, Capacity>::ShowTree() const
99  {
100   const unsigned YRoot = 12;      // Line number of root node
101   const unsigned ScrollsAt = 24;  // Screen scrolls after line 24
102
103 #if (defined _WIN32) && (!defined NoGraphics)
104
105   int xOld;                       // Old cursor x coordinate
106   int yOld;                       // Old cursor y coordinate
107
108   // Save cursor position
109   getxy(xOld, yOld);
110
111   // Has the screen scrolled yet?
112   int deltaY = 0;
113
114   if (yOld > ScrollsAt)
115     deltaY = yOld - ScrollsAt+1;
116
117   // Clear the right half of the screen.
118   for (int y=0; y<ScrollsAt+1; y++)
119   {
120     gotoxy(XRoot,y+deltaY);
121     clreol();
122   }
123   // Show the tree and offset if scrolled.
124   RShowTree(1, 0, YRoot+deltaY);
125
126   // Restore old cursor position.
127   gotoxy(xOld,yOld);
128 #endif
129 }
130
```

```
131 // Sort array into heap
132 template <typename ElemData, unsigned Capacity>
133 void Heap<ElemData, Capacity>::Sort()
134 {
135     unsigned n = size;
136     Heapify();
137 
138     while (n != 0)
139     {
140         ElemData temp = heap[1];
141         heap[1] = heap[n];
142         heap[n] = temp;
143         PercolateDown(1, --n);
144     }
145 
146 }
147 
148 // Go through entire heap and show data
149 template <typename ElemData, unsigned Capacity>
150 void Heap<ElemData, Capacity>::Output(ostream &os)
151 {
152     for (unsigned i = 1; i <= size; i++)
153     {
154         heap[i].Show(os);
155         cout << endl;
156     }
157 }
158 
159 // Convert binary tree to a heap
160 template <typename ElemData, unsigned Capacity>
161 void Heap<ElemData, Capacity>::Heapify()
162 {
163     // start with last leaf node and percolate down
164     // precede on non-leaf node until root is reached
165     for (unsigned r = size/2; r >= 1; r--)
166         PercolateDown(r, size);
167 }
168 
169 // Delete last item
170 template <typename ElemData, unsigned Capacity>
171 void Heap<ElemData, Capacity>::DeleteMax()
172 {
173     // Move last item to root and then percolate
174     heap[1] = heap[size--];
175     PercolateDown(1, size);
176 }
177 
178 // Move child up tree if greater than parent
179 template <typename ElemData, unsigned Capacity>
180 void Heap<ElemData, Capacity>::PercolateUp()
181 {
182     unsigned loc = size;
183     unsigned parent = loc / 2;
184     // Keep moving child if greater than parent
185     while (parent >= 1 && heap[loc] > heap[parent])
186     {
187         ElemData temp = heap[loc];
188         heap[loc] = heap[parent];
189         heap[parent] = temp;
190         loc = parent;
191         parent = loc / 2;
192     }
193 }
194 
195 
```

```
// Insert item in heap at correct location
template <typename ElemData, unsigned Capacity>
void Heap<ElemData, Capacity>::Insert(ElemData &data)
{
    // increment count, store new item at end of heap, then
    // percolate up to find its true location
    size++;
    heap[size] = data;
    PercolateUp();
}

// Binary Search for data
template <typename ElemData, unsigned Capacity>
bool Heap<ElemData, Capacity>::BinSearch(unsigned start, unsigned end, ElemData &data)
{
    // Keep searching while there is still something between start and end
    while (start <= end)
    {
        // check middle, if equal to data, return true, otherwise
        // increment/decrement start/end appropriately and try again
        unsigned middle = (start + end) / 2;
        if (heap[middle] == data)
        {
            current = middle;
            return true;
        }
        else if (heap[middle] > data)
            end = middle - 1;
        else
            start = middle + 1;
    }
    return false;
}

// Move item down tree if smaller than child
template <typename ElemData, unsigned Capacity>
void Heap<ElemData, Capacity>::PercolateDown(unsigned r, unsigned n)
{
    unsigned c = 2*r;
    while (c <= n)
    {
        if (c < n && heap[c] < heap[c+1])
            c++;
        if (heap[r] < heap[c])
        {
            ElemData temp = heap[r];
            heap[r] = heap[c];
            heap[c] = temp;
            r = c;
            c *= 2;
        }
        else
            break;
    }
}

// Call BinSearch
template <typename ElemData, unsigned Capacity>
bool Heap<ElemData, Capacity>::Search(ElemData &data)
{
    return BinSearch(1, size, data);
}

#endif
```