```cpp

////////////////////////////////////////////////////////////////////////////
//
// Authors:          David Hawkins, Terry Griffin
// Email:            david.james@hawkinsonline.us
// Label:            P01
// Title:            Assignment 7 - Heapify Debacle
// Course:           3013
// Semester:         Spring 2020
//
// Description:
//       This program implements a min heap class and demonstrates its
//       functionality in main() by inserting values, then taking a list of
//       unsorted values and "heapifying" it.
//
// Files:
//       heap_working.cpp        : Driver program
//
////////////////////////////////////////////////////////////////////////////
#include <iostream>

using namespace std;

/**
 * Class Heap
 *
 * @methods:
 *       constructors:
 *           Heap        : default constructor
 *           Heap(int)   : overload constructor with heap size
 *       private:
 *           BubbleUp    : you comment this
 *           Left        : you comment this
 *           OnHeap      : you comment this
 *           Parent      : you comment this
 *           Right       : you comment this
 *           Swap        : you comment this
 *           /// Fix These:
 *           SinkDown    : you comment this
 *           PickChild   : you comment this
 *       public:
 *           Insert      : you comment this
 *           Print       : you comment this
 *           Remove      : you comment this
 */
class Heap {
private:
    int size; // size of the array
    int *H;   // array pointer
    int end;  // 1 past last item in array

    /**
   * BubbleUp
   *
   * @description:
   *    This puts one value into its proper
   *    place in the heap.
   *
   * @param  {int} index : index to start bubbling at
   * @return             : void
   */
    void BubbleUp(int index) {
        // check parent is not of beginning of array
        if (Parent(index) >= 1) {
            // index is on array and value is less than parent
            while (index > 1 && H[index] < H[Parent(index)]) {
                // do a swap
                Swap(index, Parent(index));

```

```
                // update index to values new position
                index = Parent(index);
            }
        }
    }

    /**
     * Left
     * @description:
     *      Calculates index of left child.
     *
     * @param  {int} index : index of parent
     * @return {int}       : index of left child
     */
    int Left(int index) {
        return 2 * index;
    }

    /**
     * OnHeap
     * @description:
     *      Checks if an index is on the array (past the end).
     *
     * @param  {int} index : index to check
     * @return {bool}      : 0 = off heap / 1 = on heap
     */
    bool OnHeap(int index) {
        return index < end;
    }

    /**
     * Parent
     * @description:
     *      Calculates parent of a given index.
     *
     * @param  {int} index : index to check
     * @return {int}       : parent index
     */
    int Parent(int index) {
        return index / 2;
    }

    /**
     * Right
     * @description:
     *      Calculates index of right child.
     *
     * @param  {int} index : index of parent
     * @return {int}       : index of right child
     */
    int Right(int index) {
        return 2 * index + 1;
    }

    /**
     * Swap
     *
     * @description:
     *      Swaps to values in the heap
     *
     * @param  {int} i  : index in array
     * @param  {int} j  : index in array
     * @return          : void
     */
    void Swap(int i, int j) {
        int temp = H[i];
        H[i] = H[j];
        H[j] = temp;
    }
```

```
139
140         ////////////////////////////
141         //Fix These Methods
142         ////////////////////////////
143
144         /**
145          * SinkDown
146          * @description:
147          *      Places one heap item into its proper place in the heap
148          *      by bubbling it down to it proper location.
149          *
150          * @param  {int} index  : index to start from in the array
151          * @return              : void
152          */
153         void SinkDown(int index) {
154             int newIndex = 0;                   //Used to hold child index.
155             while ((index <= end) && (H[index] >//Checks if index at end, or if
156             H[Left(index)] || H[index] >        //parent is greater than child.
157             H[Right(index)])){
158
159                 newIndex = PickChild(index);    //Finds index of switched child
160                 if (OnHeap(newIndex)) {
161                     Swap(index, newIndex);      //If index on heap, switch
162                 }
163                 index = newIndex;               //Base case. Index will eventually
164             }                                   //be greater then end.
165         }
166
167         /**
168          * PickChild
169          * @description:
170          *      If one child exists, return it.
171          *      Otherwise, return the smaller of the two.
172          *
173          * @param  {int} index  : index of parent in the array
174          * @return              : index to child
175          */
176         int PickChild(int index) {
177             if (OnHeap(Left(index))) {      //if index not on heap, no children
178                 if (!OnHeap(Right(index))) {
179                     return Left(index);     //if no right child, then left child
180                 }
181                 else {                      //if two children, send smaller one
182                     if (H[Left(index)] <= H[Right(index)]) {
183                         return Left(index);
184                     }
185                     else if (H[Left(index)] >= H[Right(index)]) {
186                         return Right(index);
187                     }
188                 }
189             }
190             else {
191                 return end + 1;             //if no children, return to break loop
192             }
193         }
194
195     public:
196         /**
197        * Heap constructor
198        */
199         Heap() {
200             size = 100;
201             H = new int[size];
202             end = 1;
203         }
204
205         /**
206        * Heap constructor
207        *
```

```
208      * @param  {int} s : heap size
209      */
210      Heap(int s) {
211          size = s;
212          H = new int[s];
213          end = 1;
214      }
215
216      /**
217      * Insert
218      *
219      * @description:
220      *        Add a value to the heap.
221      *
222      * @param  {int} x : value to Insert
223      * @return         : void
224      */
225      void Insert(int x) {
226          H[end] = x;
227          BubbleUp(end);
228          end++;
229      }
230
231      /**
232       * Print
233       *
234       * @description:
235       *      Prints the contents of the heap
236       *
237       * @param       : none
238       * @return      : none
239       */
240      void Print() {
241          for (int i = 1; i <= end - 1; i++) {
242              cout << H[i];
243              if (i < end - 1) {
244                  cout << "->";
245              }
246          }
247      }
248
249      /**
250       * Remove
251       * @description:
252       *      Removes item from top of heap
253       *
254       * @return {int}  : top of heap
255       */
256      int Remove() {
257          int temp = H[1];
258          H[1] = H[end-1];
259          --end;
260          SinkDown(1);
261
262          return temp;
263      }
264
265      /**
266       * Heapify
267       * @description:
268       *      Sorts an unsorted list of values into min-heap order
269       *
270       * @param  {int*} A   :  array pointer with unsorted values to make into a heap
271       * @param  {int} size :  size of new heap
272       */
273      void Heapify(int *A, int size) {
274          for (int i = 0; i < size; i++){
275              Insert(A[i]);
276          }
```

```
277             end = size + 1;
278         }
279     };
280
281     int main() {
282         Heap H;                         //Heap object that uses sorted values
283         Heap J;                         //Heap object that stores unsorted values
284         int size = 15;                  //Size of dynamically allocated array
285         int* A = new int[size];         //Array that stores unsorted values
286
287         for (int i = 1; i <= 15; i++) { //These statements will fill the heap of
288             H.Insert(rand() % 50);      //sorted values
289         }
290         H.Print();
291         cout << endl;
292
293         for (int i = 1; i < 4; i++){    //This demonstrates the functionality of
294             H.Remove();                 //the remove method
295         }
296         H.Print();
297
298         for (int i = 0; i < 15; i++){   //Fills the array of unsorted values with
299             A[i] = rand() % 50;         //random numbers
300         }
301
302         cout << "\nUnsorted values: ";
303         for (int i = 0; i < size; i++){ //Prints the array of unsorted values
304             cout << A[i];
305             if (i < size - 1) {
306                 cout << "->";
307             }
308         }
309
310         J.Heapify(A, size);             //Sorts the values into min-heap order
311
312         cout << "\nSorted values: ";
313         J.Print();                      //Prints the sorted values
314     }
```