```
 1: #include <math.h>
 2:
 3: #include "Heap.h"
 4:
 5: Heap::Heap(int nVertex)
 6: :
 7:     m_nVertex(nVertex),
 8:     m_heapVector(NULL),
 9:     m_indexerHeap(NULL),
10:     m_nextAvailableSlot(0)
11: {
12:     allocate();
13: }
14:
15: Heap::~Heap()
16: {
17:     delete m_heapVector;
18:     delete m_indexerHeap;
19: }
20:
21: void Heap::allocate()
22: {
23:     m_heapVector = new std::pair<int, int>[ m_nVertex ];
24:     m_indexerHeap = new int[ m_nVertex ];
25:     for (int i = 0; i < m_nVertex; i++)
26:     {
27:         m_indexerHeap[i] = -1;
28:     }
29: }
30:
31:  bool Heap::removeFromHeap(std::pair<int, int> & root)
32: {
33:     if (m_nextAvailableSlot > 0) {
34:         root = m_heapVector[0];
35:
36:         std::pair<int, int> lastInserted = m_heapVector[m_nextAvailableSlot - 1];
37:
38:         m_heapVector[0] = lastInserted;
39:
40:         // element ja esta alocado no heapvector, para zerar eh so setar zero
41:         m_heapVector[m_nextAvailableSlot - 1].first = 0;// = *(new std::pair<int, int>(0, 0));
42:         m_heapVector[m_nextAvailableSlot - 1].second = 0;
43:         m_nextAvailableSlot--;
44:
45:         // remove vertex from heap's index
46:         m_indexerHeap[root.first] = -1;
47:
48:         bubleDownElement(0);
49:
50:     } else {
51:         fprintf( stderr, "Error removing from empty heap.\n" );
52:         return false;
53:     }
54:
55:     return true;
56: }
57:
58: bool Heap::HasVertex(int iVertex) const
59: {
60:     return m_indexerHeap[iVertex] != -1;
61: }
62:
63: void Heap::insertOnHeap(int iVertex, int degree)
64: {
65:     m_heapVector[m_nextAvailableSlot].first = iVertex;
66:     m_heapVector[m_nextAvailableSlot].second = degree;
67:
68:     // Fill indexer vector
69:     m_indexerHeap[iVertex] =  m_nextAvailableSlot;
70:
71:     // we may need to use our current status in bubbleUpElement,
72:     // so we need to increment before calling the method below
73:     // do not move position
74:     m_nextAvailableSlot++;
75:
76:     bubleUpElement(m_nextAvailableSlot - 1);
77: }
78:
79:
80: void Heap::DecrementDegree( int iVertex )
81: {
82:     m_heapVector[ m_indexerHeap[iVertex] ].second--;
83:     bubleDownElement(m_indexerHeap[iVertex]);
84: }
85:
86: void Heap::bubleUpElement(int iSlotIndex)
87: {
88:     // First element don't have father
89:     if (iSlotIndex <= 0) {
90:         return;
91:     }
92:
93:     // swap element with father
94:     if (swapWithFather(iSlotIndex)) {
```

```
 95:         // if we had an inversion, call recursively for the father
 96:         bubleUpElement((iSlotIndex - 1) / 2);
 97:     }
 98: }
 99:
100: bool Heap::swapWithFather(int iSlotIndex)
101: {
102:     std::pair<int, int> element = m_heapVector[iSlotIndex];
103:
104:     // Compare with parent
105:     if (element.second > m_heapVector[(iSlotIndex - 1) / 2].second) {
106:
107:         // swap with father
108:         std::pair<int, int> father = m_heapVector[(iSlotIndex - 1) / 2];
109:
110:         m_heapVector[(iSlotIndex - 1) / 2] = element;
111:         m_heapVector[iSlotIndex] = father;
112:
113:         m_indexerHeap[ element.first ] = (iSlotIndex - 1) / 2;
114:         m_indexerHeap[ father.first ] = iSlotIndex;
115:
116:         return true;
117:     }
118:
119:     return false;
120: }
121:
122: void Heap::bubleDownElement(int iSlotIndex)
123: {
124:     // Last Nodes already had passed
125:     if (iSlotIndex >= m_nextAvailableSlot) {
126:         return;
127:     }
128:
129:     int swapIndex = swapWithChildren(iSlotIndex);
130:
131:     if (swapIndex == 1) {
132:         bubleDownElement(2 * iSlotIndex + 1);
133:     } else if (swapIndex == 2) {
134:         bubleDownElement(2 * iSlotIndex + 2);
135:     }
136:
137: }
138:
139: int Heap::swapWithChildren(int iSlotIndex)
140: {
141:     int higher = -1;
142:     int result = -1;
143:
144:     int leftChildIndex = 2 * iSlotIndex + 1;
145:     int rightChildIndex = 2 * iSlotIndex + 2;
146:
147:     if((leftChildIndex < m_nextAvailableSlot)  && (rightChildIndex < m_nextAvailableSlot)) {
148:         if (m_heapVector[iSlotIndex].second < m_heapVector[leftChildIndex].second &&
149:             m_heapVector[iSlotIndex].second < m_heapVector[rightChildIndex].second) {
150:             if (m_heapVector[leftChildIndex].second > m_heapVector[rightChildIndex].second) {
151:                 higher = 2 * iSlotIndex + 1;
152:                 result = 1;
153:
154:             } else if (m_heapVector[rightChildIndex].second > m_heapVector[leftChildIndex].second) {
155:                 higher = 2 * iSlotIndex + 2;
156:                 result = 2;
157:             }
158:         }
159:
160:         if (higher != -1) {
161:             // swap with child
162:             std::pair<int, int> child = m_heapVector[higher];
163:             std::pair<int, int> element = m_heapVector[iSlotIndex];
164:
165:             m_heapVector[higher] = m_heapVector[iSlotIndex];
166:             m_heapVector[iSlotIndex] = child;
167:
168:             m_indexerHeap[ element.first ] = higher;
169:             m_indexerHeap[ child.first ] = iSlotIndex;
170:
171:             // 1 indicates that we swapped with left child
172:             return result;
173:         }
174:     }
175:
176:     if (leftChildIndex < m_nextAvailableSlot) {
177:         // Compare with left child
178:         if (m_heapVector[iSlotIndex].second < m_heapVector[leftChildIndex].second) {
179:             std::pair<int, int> element = m_heapVector[iSlotIndex];
180:             // swap with child
181:             std::pair<int, int> child = m_heapVector[leftChildIndex];
182:
183:             m_heapVector[leftChildIndex] = m_heapVector[iSlotIndex];
184:             m_heapVector[iSlotIndex] = child;
185:
186:             m_indexerHeap[ element.first ] = leftChildIndex;
187:             m_indexerHeap[ child.first ] = iSlotIndex;
188:
```

```
189:             // 1 indicates that we swapped with left child
190:             return 1;
191:         }
192:     }
193:
194:     if (rightChildIndex < m_nextAvailableSlot) {
195:         if(m_heapVector[iSlotIndex].second < m_heapVector[rightChildIndex].second) {
196:             std::pair<int, int> element = m_heapVector[iSlotIndex];
197:             // swap with child
198:             std::pair<int, int> child = m_heapVector[rightChildIndex];
199:
200:             m_heapVector[rightChildIndex] = m_heapVector[iSlotIndex];
201:             m_heapVector[iSlotIndex] = child;
202:
203:             m_indexerHeap[ element.first ] = rightChildIndex;
204:             m_indexerHeap[ child.first ] = iSlotIndex;
205:
206:             // 1 indicates that we swapped with left child
207:             return 2;
208:         }
209:     }
210:
211:     return 0;
212: }
213:
214: void Heap::print()
215: {
216:     fprintf(stderr, "Vertex: \n");
217:     for(int i = 0; i < m_nVertex; i++) {
218:         fprintf(stderr, " %d ", m_heapVector[i].first);
219:     }
220:
221:     fprintf(stderr, "\nDegrees: \n");
222:     for(int i = 0; i < m_nVertex; i++) {
223:         fprintf(stderr, " %d ", m_heapVector[i].second);
224:     }
225:     fprintf(stderr, "\n");
226: }
```