Deakin University

Data Structures and Algorithms

OnTrack Submission

Programming - Heap

Submitted By:
Peter Stacey
pstacey
2020/09/09 08:35

Tutor:
Maksym Slavnenko

Outcome	**Weight**
Complexity	♦♦♦◊◊
Implement Solutions	♦♦♦◊◊
Document solutions	♦◊◊◊◊

The task involves exploring a Heap data structure and standard as well as some non-standard operations on a heap. As a coding task, this aligns with ULO2. Additionally, the task requires analysis of the space and time complexity of the algorithms used in the non-standard methods, and also requires implementing approaches to meet specific time complexity goals. This aligns with ULO1.

September 9, 2020

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Task_8_1
{
    public class Heap<K, D> where K : IComparable<K>
    {

        // This is a nested Node class whose purpose is to represent a node of a heap.
        private class Node : IHeapifyable<K, D>
        {
            // The Data field represents a payload.
            public D Data { get; set; }
            // The Key field is used to order elements with regard to the Binary Min (Max) Heap Policy, i.e. the key of the parent node is smaller (larger) than the key of its children.
            public K Key { get; set; }
            // The Position field reflects the location (index) of the node in the array-based internal data structure.
            public int Position { get; set; }

            public Node(K key, D value, int position)
            {
                Data = value;
                Key = key;
                Position = position;
            }

            // This is a ToString() method of the Node class.
            // It prints out a node as a tuple ('key value','payload','index')}.
            public override string ToString()
            {
                return "(" + Key.ToString() + "," + Data.ToString() + "," + Position + ")";
            }
        }

        // ---------------------------------------------------------------------
        // Here the description of the methods and attributes of the Heap<K, D> class starts

        public int Count { get; private set; }

        // The data nodes of the Heap<K, D> are stored internally in the List collection.
        // Note that the element with index 0 is a dummy node.
        // The top-most element of the heap returned to the user via Min() is indexed as 1.
```

```
45          private List<Node> data = new List<Node>();
46
47          // We refer to a given comparer to order elements in the heap.
48          // Depending on the comparer, we may get either a binary Min-Heap or a
            ↪  binary  Max-Heap.
49          // In the former case, the comparer must order elements in the ascending
            ↪  order of the keys, and does this in the descending order in the latter
            ↪  case.
50          private IComparer<K> comparer;
51
52          // We expect the user to specify the comparer via the given argument.
53          public Heap(IComparer<K> comparer)
54          {
55              this.comparer = comparer;
56
57              // We use a default comparer when the user is unable to provide one.
58              // This implies the restriction on type K such as 'where K :
                ↪  IComparable<K>' in the class declaration.
59              if (this.comparer == null) this.comparer = Comparer<K>.Default;
60
61              // We simplify the implementation of the Heap<K, D> by creating a dummy
                ↪  node at position 0.
62              // This allows to achieve the following property:
63              // The children of a node with index i have indices 2*i and 2*i+1 (if
                ↪  they exist).
64              data.Add(new Node(default(K), default(D), 0));
65          }
66
67          // This method returns the top-most (either a minimum or a maximum) of the
            ↪  heap.
68          // It does not delete the element, just returns the node casted to the
            ↪  IHeapifyable<K, D> interface.
69          public IHeapifyable<K, D> Min()
70          {
71              if (Count == 0) throw new InvalidOperationException("The heap is
                ↪  empty.");
72              return data[1];
73          }
74
75          // Insertion to the Heap<K, D> is based on the private UpHeap() method
76          public IHeapifyable<K, D> Insert(K key, D value)
77          {
78              Count++;
79              Node node = new Node(key, value, Count);
80              data.Add(node);
81              UpHeap(Count);
82              return node;
83          }
84
85          private void UpHeap(int start)
86          {
87              int position = start;
88              while (position != 1)
```

```
89              {
90                  if (comparer.Compare(data[position].Key, data[position / 2].Key) <
                    ↪  0) Swap(position, position / 2);
91                  position = position / 2;
92              }
93          }
94
95          // This method swaps two elements in the list representing the heap.
96          // Use it when you need to swap nodes in your solution, e.g. in DownHeap()
                ↪  that you will need to develop.
97          private void Swap(int from, int to)
98          {
99              Node temp = data[from];
100             data[from] = data[to];
101             data[to] = temp;
102             data[to].Position = to;
103             data[from].Position = from;
104         }
105
106         public void Clear()
107         {
108             for (int i = 0; i<=Count; i++) data[i].Position = -1;
109             data.Clear();
110             data.Add(new Node(default(K), default(D), 0));
111             Count = 0;
112         }
113
114         public override string ToString()
115         {
116             if (Count == 0) return "[]";
117             StringBuilder s = new StringBuilder();
118             s.Append("[");
119             for (int i = 0; i < Count; i++)
120             {
121                 s.Append(data[i + 1]);
122                 if (i + 1 < Count) s.Append(",");
123             }
124             s.Append("]");
125             return s.ToString();
126         }
127
128         // TODO: Your task is to implement all the remaining methods.
129         // Read the instruction carefully, study the code examples from above as
                ↪  they should help you to write the rest of the code.
130
131
132         /// <summary>
133         /// Deletes the minimum node from the Heap
134         /// </summary>
135         /// <returns>Node as an IHeapifyable, removed from the heap</returns>
136         public IHeapifyable<K, D> Delete()
137         {
138             if (Count is 0) throw new InvalidOperationException();
```

```
139             Node result = data[1];
140             Swap(1, Count);
141             data.RemoveAt(Count);
142             Count--;
143             DownHeap(1);
144             return result;
145         }
146 
147         // Returns the index of a possible left child of an element in the heap
148         private int Left(int start) => start * 2;
149 
150         // Returns the index of a possible right child of an element in the heap
151         private int Right(int start) => start * 2 + 1;
152 
153         // Recursively checks a node against its left and right children and moves
            ↪  the node
154         // downward in the heap as necessary, to ensure the heap property is correct
155         private void DownHeap(int start)
156         {
157             int leftChild = Left(start);
158             if (leftChild > Count) return;
159 
160             int rightChild = Right(start);
161             int position = leftChild;
162 
163             if (rightChild < Count &&
164                 comparer.Compare(data[leftChild].Key, data[rightChild].Key) >= 0)
165                     position = rightChild;
166 
167             if (comparer.Compare(data[start].Key, data[position].Key) < 0) return;
168 
169             Swap(start, position);
170             DownHeap(position);
171         }
172 
173         /// <summary>
174         /// Builds a minimum binary heap using the specified data according to the
            ↪  bottom-up approach
175         /// </summary>
176         /// <param name="keys">Array of keys for each item of data</param>
177         /// <param name="data">Array of data to be added to the heap</param>
178         /// <returns>Array of the items in the heap</returns>
179         /// <exception cref="System.InvalidOperationException">Thrown when the heap
            ↪  is empty, or if the
180         /// number of keys is not the same as the number of data items</exception>
181         public IHeapifyable<K, D>[] BuildHeap(K[] keys, D[] data)
182         {
183             if (Count != 0) throw new InvalidOperationException();
184             if (keys.Length != data.Length) throw new InvalidOperationException();
185 
186             Node[] result = new Node[keys.Length];
187 
188             for(int i = 0; i < keys.Length; i++)
```

```
189             {
190                 Node node = new Node(keys[i], data[i], ++Count);
191                 this.data.Add(node);
192                 result[i] = node;
193             }
194             Heapify();
195 
196             return result;
197         }
198 
199         // Ensures correct Heap property for a bottom-up build
200         private void Heapify()
201         {
202             for (int i = Count / 2; i > 0; i--) DownHeap(i);
203         }
204 
205         /// <summary>
206         ///
207         /// </summary>
208         /// <param name="element">IHeapifyable element to change the key of</param>
209         /// <param name="new_key">New key to be applied to the element</param>
210         /// <exception cref="System.InvalidOperationException">Thrown when the
            ↪ state of the heap, does
211         /// not match the key and data of the passed in element</exception>
212         public void DecreaseKey(IHeapifyable<K, D> element, K new_key)
213         {
214             Node result = element as Node;
215             if (!result.Equals(data[result.Position])) throw new
                ↪ InvalidOperationException();
216             result.Key = new_key;
217             UpHeap(result.Position);
218         }
219 
220         /// <summary>
221         /// Deletes a specific node if it exists, no mater where it is in the heap
222         /// </summary>
223         /// <param name="element">IHeapifyable node to be deleted</param>
224         /// <returns>Node deleted</returns>
225         /// <exception cref="System.InvalidOperationException">Thrown when the
            ↪ element is null, or
226         /// if the state of the heap does not match the values contained in the
            ↪ element</exception>
227         public IHeapifyable<K, D> DeleteElement(IHeapifyable<K, D> element)
228         {
229             if (element is null) throw new ArgumentNullException();
230 
231             Node node = element as Node;
232             if (!data[node.Position].Key.Equals(element.Key)) throw new
                ↪ InvalidOperationException();
233 
234             int position = node.Position;
235 
236             // O(1) complexity
```

```
237             Swap(position, Count);
238 
239             data.RemoveAt(Count);
240             Count--;
241 
242             // O(log n) to repair the heap property
243             DownHeap(position);
244 
245             return node;
246         }
247 
248         /// <summary>
249         /// Returns the Kth minimum element of a heap
250         /// </summary>
251         /// <param name="k">The Kth element to return</param>
252         /// <returns>The Kth minimum element</returns>
253         public IHeapifyable<K, D> KthMinElement(int k)
254         {
255             if (Count is 0) throw new InvalidOperationException();
256             if (k <= 0) throw new ArgumentOutOfRangeException();
257 
258             Heap<K, D> queue = new Heap<K, D>(Comparer<K>.Default);
259 
260             // (k log k) complexity to insert k items
261             for (int i = 1; i <= k; i++) queue.Insert(data[i].Key, data[i].Data);
262 
263             Node min = queue.data[1];
264 
265             // (k) loop to visit each node in the queue once
266             for(int i = 1; i <= queue.Count; i++)
267             {
268                 if(comparer.Compare(min.Key, queue.data[i].Key) < 0) min =
                    ↪  queue.data[i];
269             }
270 
271             return min;
272         }
273 
274     }
275 }
```