

# Heap.cs Implementation

SIT221 Data Structures and Algorithms

May 2025

## Heap.cs Source Code

This document contains the complete implementation of the `Heap.cs` file for the SIT221 Data Structures and Algorithms practical task 6.2, implementing a generic binary heap in C#.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace Heap
8  {
9      public class Heap<K, D> where K : IComparable<K>
10     {
11         private class Node : IHeapifyable<K, D>
12         {
13             public D Data { get; set; }
14             public K Key { get; set; }
15             public int Position { get; set; }
16
17             public Node(K key, D value, int position)
18             {
19                 Data = value;
20                 Key = key;
21                 Position = position;
22             }
23
24             public override string ToString()
25             {
26                 return "(" + Key.ToString() + "," + Data.ToString() +
27                     "," + Position + ")";
28             }
29
30         public int Count { get; private set; }
31         private List<Node> data = new List<Node>();
32         private IComparer<K> comparer;
33
34         public Heap(IComparer<K> comparer)
35         {
36             this.comparer = comparer;
37             if (this.comparer == null) this.comparer =
38                 Comparer<K>.Default;
39             data.Add(new Node(default(K), default(D), 0));
39 }
```

```

40
41     public IHeapifyable<K, D> Min()
42     {
43         if (Count == 0) throw new InvalidOperationException("The
44             heap is empty.");
45         return data[1];
46     }
47
48     public IHeapifyable<K, D> Insert(K key, D value)
49     {
50         Count++;
51         Node node = new Node(key, value, Count);
52         data.Add(node);
53         UpHeap(Count);
54         return node;
55     }
56
57     private void UpHeap(int start)
58     {
59         int position = start;
60         while (position != 1)
61         {
62             if (comparer.Compare(data[position].Key, data[position
63                 / 2].Key) < 0) Swap(position, position / 2);
64             position = position / 2;
65         }
66     }
67
68     private void Swap(int from, int to)
69     {
70         Node temp = data[from];
71         data[from] = data[to];
72         data[to] = temp;
73         data[to].Position = to;
74         data[from].Position = from;
75     }
76
77     public void Clear()
78     {
79         for (int i = 0; i <= Count; i++) data[i].Position = -1;
80         data.Clear();
81         data.Add(new Node(default(K), default(D), 0));
82         Count = 0;
83     }
84
85     public override string ToString()
86     {
87         if (Count == 0) return "[]";
88         StringBuilder s = new StringBuilder();
89         s.Append("[");
90         for (int i = 0; i < Count; i++)
91         {
92             s.Append(data[i + 1]);
93             if (i + 1 < Count) s.Append(", ");
94         }
95         s.Append("]");
96         return s.ToString();
97     }

```

```

96
97     public IHeapifyable<K, D> Delete()
98     {
99         if (Count == 0) throw new InvalidOperationException("The
100            heap is empty.");
101
102         Node result = data[1];
103         data[1] = data[Count];
104         data[1].Position = 1;
105         data.RemoveAt(Count);
106         Count--;
107
108         if (Count > 1) DownHeap(1);
109
110         return result;
111     }
112
113     private void DownHeap(int start)
114     {
115         int position = start;
116         while (position <= Count / 2)
117         {
118             int smallest = position;
119             int left = 2 * position;
120             int right = 2 * position + 1;
121
122             if (left <= Count && comparer.Compare(data[left].Key,
123                 data[smallest].Key) < 0)
124                 smallest = left;
125             if (right <= Count &&
126                 comparer.Compare(data[right].Key,
127                     data[smallest].Key) < 0)
128                 smallest = right;
129
130             if (smallest == position) break;
131
132             Swap(position, smallest);
133             position = smallest;
134         }
135     }
136
137     public IHeapifyable<K, D>[] BuildHeap(K[] keys, D[] data)
138     {
139         if (Count != 0) throw new InvalidOperationException("The
140            heap is not empty.");
141         if (keys == null || data == null || keys.Length !=
142             data.Length)
143             throw new ArgumentException("Invalid input arrays.");
144
145         IHeapifyable<K, D>[] result = new IHeapifyable<K,
146             D>[keys.Length];
147         Count = keys.Length;
148
149         for (int i = 0; i < keys.Length; i++)
150         {
151             Node node = new Node(keys[i], data[i], i + 1);
152             this.data.Add(node);
153             result[i] = node;

```

```

147    }
148
149    for (int i = Count / 2; i >= 1; i--)
150        DownHeap(i);
151
152    return result;
153}
154
155 public void DecreaseKey(IHeapifyable<K, D> element, K new_key)
156 {
157     Node node = element as Node;
158     if (node == null || node.Position < 1 || node.Position >
159         Count || data[node.Position] != node)
160         throw new InvalidOperationException("Invalid
161             element.");
162
163     if (comparer.Compare(new_key, node.Key) > 0)
164         throw new ArgumentException("New key is larger than
165             current key.");
166
167     node.Key = new_key;
168     UpHeap(node.Position);
169 }
170 }

```

Listing 1: Heap.cs: Generic Binary Heap Implementation