

MIT WORLD PEACE UNIVERSITY

Advanced Data Structures  
Second Year B. Tech, Semester 4

---

---

IMPLEMENTATION OF HEAP AS A DATA  
STRUCTURE

---

---

ASSIGNMENT No. 7

Prepared By

Krishnaraj Thadesar  
Cyber Security and Forensics  
Batch A1, PA 20

April 25, 2023

# Contents

<b>1 Objectives</b>	<b>1</b>
<b>2 Problem Statement</b>	<b>1</b>
<b>3 Theory</b>	<b>1</b>
3.1 Heap . . . . .	1
3.2 Types of Heaps . . . . .	2
3.3 Construction of heaps . . . . .	2
3.4 Data Structures Used for Heap Constructions . . . . .	3
3.5 Time and Space Complexities Associated with Heap . . . . .	3
3.6 Heap Vs Binary Search Trees . . . . .	3
3.7 Applications of Heap . . . . .	3
<b>4 Platform</b>	<b>3</b>
<b>5 Test Conditions</b>	<b>4</b>
<b>6 Input and Output</b>	<b>4</b>
<b>7 Pseudo Code</b>	<b>4</b>
<b>8 Time Complexity</b>	<b>5</b>
8.1 Min and Max Heap Creation . . . . .	5
8.2 Min or Max Heap Traversal . . . . .	5
8.3 Heap Sort . . . . .	5
<b>9 Searching in Heap</b>	<b>5</b>
<b>10 Code</b>	<b>5</b>
10.1 Program . . . . .	5
<b>11 Conclusion</b>	<b>15</b>
<b>12 FAQ</b>	<b>16</b>

## 1 Objectives

1. To study the concept of heap
2. To study different types of heap and their algorithms

## 2 Problem Statement

*Read the marks obtained by students of second year in an online examination of particular subject. Find out maximum and minimum marks obtained in that subject. Use heap data structure and Heap sort.*

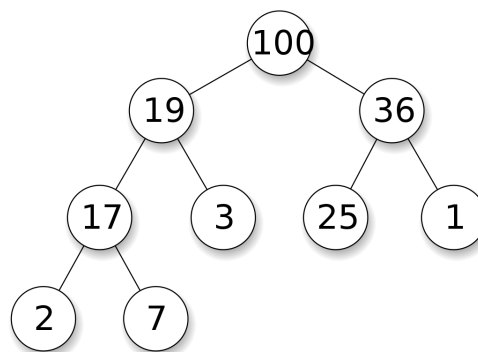
## 3 Theory

### 3.1 Heap

A heap is a specialized tree-based data structure that satisfies the heap property. The heap property is a condition where each node in the tree is greater than or equal to (in a max heap) or less than or equal to (in a min heap) its children. The root node of the heap is the maximum (or minimum) element in a max (or min) heap.

Heaps are often used to implement priority queues, where elements are extracted in order of priority. For example, in a hospital, patients with the most urgent medical needs are given the highest priority for treatment. A priority queue based on a heap can efficiently manage the order of patients by storing their priority level (e.g., critical, urgent, or routine) in each node and maintaining the heap property.

Tree representation



Array representation

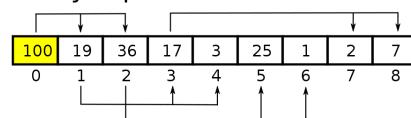


Figure 1:

### 3.2 Types of Heaps

There are two main types of heaps: max heaps and min heaps. In a max heap, the maximum element is always stored at the root, and every node is greater than or equal to its children. In a min heap, the minimum element is stored at the root, and every node is less than or equal to its children.

Both types of heaps have their own advantages and use cases. Max heaps are often used to implement priority queues, where the highest priority element needs to be extracted first. Min heaps are often used in algorithms such as Dijkstra's shortest path algorithm, where the minimum distance to a vertex needs to be determined.

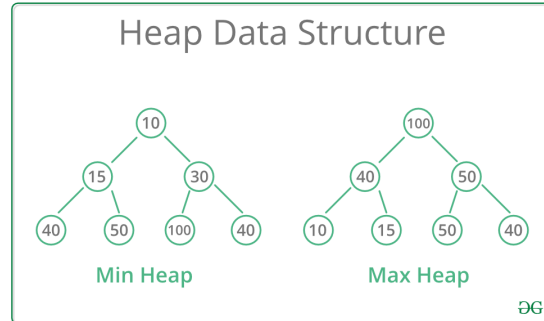


Figure 2: Min Heap and Max Heap

### 3.3 Construction of heaps

Heaps can be constructed using a variety of algorithms, including the bottom-up construction algorithm and the top-down construction algorithm. The bottom-up construction algorithm starts with a partially ordered set of elements and iteratively adds elements to the heap in a way that maintains the heap property. The top-down construction algorithm starts with an empty heap and iteratively adds elements to the heap in a way that maintains the heap property.

Regardless of the algorithm used, the construction of a heap takes  $O(n)$  time, where  $n$  is the number of elements in the heap. This is because each element must be inserted into the heap and the heap property must be maintained at each step.

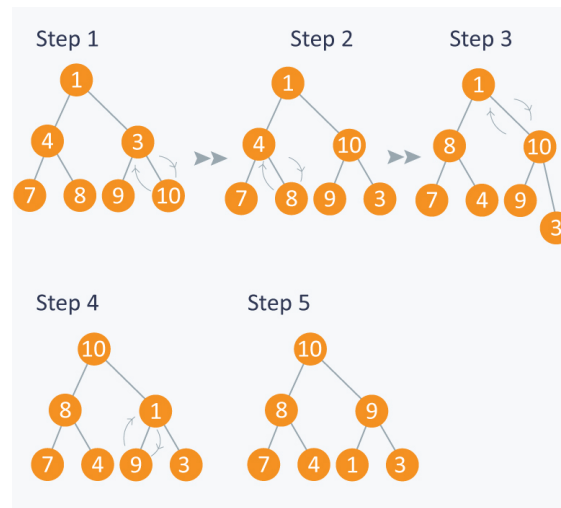


Figure 3:

### 3.4 Data Structures Used for Heap Constructions

The most common data structure used for heap construction is an array. In an array-based heap, the elements are stored in an array in a way that maintains the heap property. The root element is stored at index 0, and the children of a node at index  $i$  are stored at indices  $2i+1$  and  $2i+2$ .

Linked lists can also be used to implement heaps, but they are less commonly used due to their higher overhead.

### 3.5 Time and Space Complexities Associated with Heap

The time complexity of heap operations depends on the height of the heap, which is  $O(\log n)$  for a heap with  $n$  elements. The space complexity of a heap is  $O(n)$ , where  $n$  is the number of elements in the heap.

Inserting an element into a heap takes  $O(\log n)$  time, as the element must be inserted at the bottom of the heap and then sifted up to maintain the heap property. Similarly, extracting the maximum (or minimum) element from a heap takes  $O(\log n)$  time, as the root element must be removed and then the heap must be reorganized to maintain the heap property.

Heap operations are generally more efficient than operations on other data structures such as arrays or linked lists, particularly for large datasets. However, they can be less efficient than other data structures for small datasets due to the overhead associated with maintaining the heap property.

### 3.6 Heap Vs Binary Search Trees

Heaps are often compared to binary search trees (BSTs), another tree-based data structure. However, heaps are not as efficient as BSTs for all operations. For example, searching for an element in a heap takes  $O(n)$  time, as each element must be searched in the worst case. In contrast, searching for an element in a BST takes  $O(\log n)$  time, as the search can be narrowed down to a single subtree.

However, heaps are more efficient than BSTs for operations such as finding the maximum (or minimum) element, as the root element is always the maximum (or minimum) element in a heap. In contrast, finding the maximum (or minimum) element in a BST takes  $O(\log n)$  time, as the maximum (or minimum) element may be located at the bottom of the tree.

### 3.7 Applications of Heap

Heaps are often used to implement priority queues, where elements are extracted in order of priority. For example, in a hospital, patients with the most urgent medical needs are given the highest priority for treatment. A priority queue based on a heap can efficiently manage the order of patients by storing their priority level (e.g., critical, urgent, or routine) in each node and maintaining the heap property.

Heaps are also used in algorithms such as Dijkstra's shortest path algorithm, where the minimum distance to a vertex needs to be determined. In this case, the heap stores the vertices that have not yet been visited, and the minimum distance to each vertex is stored in each node. The heap property is maintained by updating the minimum distance to each vertex as the algorithm progresses.

## 4 Platform

**Operating System:** Arch Linux x86-64

**IDEs or Text Editors Used:** Visual Studio Code

**Compilers :** g++ and gcc on linux for C++

### 5 Test Conditions

1. Input min 10 elements.
2. Display Max and Min Heap
3. Find Maximum and Minimum marks obtained in a particular subject.

### 6 Input and Output

1. The minimum cost of the spanning tree.

### 7 Pseudo Code

#### Pseudo Code for Creation of Min and Max Heap

```
1 CreateMaxHeap(array)
2   for i from n/2 down to 1 do:
3       MaxHeapify(array, i, n)
4
5 CreateMinHeap(array)
6   for i from n/2 down to 1 do:
7       MinHeapify(array, i, n)
```

#### Pseudo Code for Heapify function

```
1 MaxHeapify(array, i, n)
2   left = 2i
3   right = 2i + 1
4   largest = i
5
6   if left <= n and array[left] > array[largest] then:
7       largest = left
8   if right <= n and array[right] > array[largest] then:
9       largest = right
10  if largest != i then:
11      swap(array[i], array[largest])
12      MaxHeapify(array, largest, n)
13
14 MinHeapify(array, i, n)
15   left = 2i
16   right = 2i + 1
17   smallest = i
18
19   if left <= n and array[left] < array[smallest] then:
20       smallest = left
21   if right <= n and array[right] < array[smallest] then:
22       smallest = right
23   if smallest != i then:
24       swap(array[i], array[smallest])
25       MinHeapify(array, smallest, n)
```

## 8 Time Complexity

### 8.1 Min and Max Heap Creation

- **Time Complexity:**  $O(n \log(n))$
- **Space Complexity:**  $O(n)$

### 8.2 Min or Max Heap Traversal

- **Time Complexity:**  $O(n \log(n))$
- **Space Complexity:**  $O(n)$

### 8.3 Heap Sort

- **Time Complexity:**  $O(n \log(n))$
- **Space Complexity:**  $O(n)$

## 9 Searching in Heap

- **Time Complexity:**  $O(n)$

## 10 Code

### 10.1 Program

```
1 // Program for Creating Heaps and Heap Sort
2 // Read the marks obtained by students of second year in an online examination of
3 // particular subject. Find out maximum and minimum marks obtained in that subject
  . Use
4 // heap data structure and Heap sort.
5
6 #include <iostream>
7 #include <fstream>
8 #include <string>
9 #include <vector>
10 #include <algorithm>
11 using namespace std;
12
13
14 class Heap
15 {
16     vector<int> v;
```

```
17     int n;
18     int left(int i)
19     {
20         return 2*i+1;
21     }
22     int right(int i)
23     {
24         return 2*i+2;
25     }
26     int parent(int i)
27     {
28         return (i-1)/2;
29     }
30     void heapify(int i)
31     {
32         int l=left(i);
33         int r=right(i);
34         int largest=i;
35         if(l<n && v[l]>v[i])
36             largest=l;
37         if(r<n && v[r]>v[largest])
38             largest=r;
39         if(largest!=i)
40         {
41             swap(v[i],v[largest]);
42             heapify(largest);
43         }
44     }
45
46 public:
47
48     Heap()
49     {
50         n=0;
51     }
52     void insert(int x)
53     {
54         v.push_back(x);
55         n++;
56         int i=n-1;
57         while(i>0 && v[parent(i)]<v[i])
58         {
59             swap(v[i],v[parent(i)]);
60             i=parent(i);
61         }
62     }
63     void display()
64     {
65         for(int i=0;i<n;i++)
66             cout<<v[i]<<" ";
67         cout<<endl;
68     }
69     void heapsort()
70     {
71         for(int i=n-1;i>=0;i--)
72         {
73             swap(v[0],v[i]);
74             n--;
75             heapify(0);
```



```
76     }
77 }
78 int getmax()
79 {
80     return v[0];
81 }
82 int getmin()
83 {
84     return v[n-1];
85 }
86 void buildheap()
87 {
88     for(int i=n/2-1;i>=0;i--)
89         heapify(i);
90 }
91 void swap(int &a,int &b)
92 {
93     int temp=a;
94     a=b;
95     b=temp;
96 }
97
98 };
99
100 int main()
101 {
102     Heap h;
103     int ch;
104     int x;
105     do
106     {
107         cout<<"1.Insert\n2.Display\n3.Heapsort\n4.Get Max\n5.Get Min\n6.Build Heap\n7.Exit\n";
108         cin>>ch;
109         switch(ch)
110         {
111             case 1:
112                 cout<<"Enter the element to be inserted: ";
113                 cin>>x;
114                 h.insert(x);
115                 break;
116             case 2:
117                 h.display();
118                 break;
119             case 3:
120                 h.heapsort();
121                 break;
122             case 4:
123                 cout<<"Maximum element: "<<h.getmax()<<endl;
124                 break;
125             case 5:
126                 cout<<"Minimum element: "<<h.getmin()<<endl;
127                 break;
128             case 6:
129                 h.buildheap();
130                 break;
131             case 7:
132                 break;
133             default:
```

```
134         cout<<"Invalid Choice\n";
135     }
136     }while(ch!=7);
137     return 0;
138 }
139
140 // #include <iostream>
141 // #include <algorithm>
142
143 // using namespace std;
144
145 // class Heap
146 // {
147 // private:
148 //     int *arr;
149 //     int size;
150
151 //     void heapify(int i)
152 //     {
153 //         int largest = i;
154 //         int left = 2 * i + 1;
155 //         int right = 2 * i + 2;
156
157 //         if (left < size && arr[left] > arr[largest])
158 //         {
159 //             largest = left;
160 //         }
161
162 //         if (right < size && arr[right] > arr[largest])
163 //         {
164 //             largest = right;
165 //         }
166
167 //         if (largest != i)
168 //         {
169 //             swap(arr[i], arr[largest]);
170 //             heapify(largest);
171 //         }
172 //     }
173
174 // public:
175 //     Heap(int *arr, int n)
176 //     {
177 //         this->arr = arr;
178 //         this->size = n;
179
180 //         // build max heap
181 //         for (int i = n / 2 - 1; i >= 0; i--)
182 //         {
183 //             heapify(i);
184 //         }
185 //     }
186
187 //     void heapSort()
188 //     {
189 //         // sort the array using heap sort
190 //         for (int i = size - 1; i >= 0; i--)
191 //         {
192 //             swap(arr[0], arr[i]);
```

```
193 //             size--;
194 //             heapify(0);
195 //         }
196 //     }
197
198 //     int getMin()
199 //     {
200 //         return arr[0];
201 //     }
202
203 //     int getMax()
204 //     {
205 //         return arr[size - 1];
206 //     }
207 // };
208
209 // int main()
210 // {
211 //     // input marks obtained by students
212 //     int marks[] = {80, 60, 70, 90, 85};
213
214 //     // calculate number of students
215 //     int n = sizeof(marks) / sizeof(marks[0]);
216
217 //     // create heap object
218 //     Heap heap(marks, n);
219
220 //     // perform heap sort to find max and min marks
221 //     heap.heapSort();
222
223 //     // print max and min marks
224 //     cout << "Maximum marks: " << heap.getMax() << endl;
225 //     cout << "Minimum marks: " << heap.getMin() << endl;
226
227 //     return 0;
228 // }
```

```
1 Enter size of table:
2 4
3 Enter data for employee 1
4 Enter name:
5 Krish
6 Enter id:
7 12
8 Enter age:
9 21
10
11
12 Employee 1:
13 Name: Krish
14 Id: 12
15 Age: 21
16
17 Enter data for employee 2
18 Enter name:
19 Part
20 Enter id:
21 42
22 Enter age:
```

```
23 22
24
25
26 Employee 2:
27 Name: Part
28 Id: 42
29 Age: 22
30
31 Enter data for employee 3
32 Enter name:
33 Ram
34 Enter id:
35 23
36 Enter age:
37 32
38
39
40 Employee 3:
41 Name: Ram
42 Id: 23
43 Age: 32
44
45 Enter data for employee 4
46 Enter name:
47 Ramesh
48 Enter id:
49 24
50 Enter age:
51 21
52
53
54 Employee 4:
55 Name: Ramesh
56 Id: 24
57 Age: 21
58
59 Enter 1 to insert with replacement, 2 to insert without replacement:
60 1
61 Hash table now looks like this.
62 0:-1
63 1:-1
64 2:12
65 3:23
66 4:24
67 5:42
68 6:-1
69 7:-1
70 8:-1
71 9:-1
72
73 Inserting data into the file.
74 Writing employee 1
75 Employee 1:
76 Name:
77 Id: 0
78 Age: 0
79 Writing employee 2
80 Employee 2:
81 Name:
```

```
82 Id: 0
83 Age: 0
84 Writing employee 3
85 Employee 3:
86 Name: Krish
87 Id: 12
88 Age: 21
89 Writing employee 4
90 Employee 4:
91 Name: Ram
92 Id: 23
93 Age: 32
94 Writing employee 5
95 Employee 5:
96 Name: Ramesh
97 Id: 24
98 Age: 21
99 Writing employee 6
100 Employee 6:
101 Name: Part
102 Id: 42
103 Age: 22
104 Writing employee 7
105 Employee 7:
106 Name:
107 Id: 0
108 Age: 0
109 Writing employee 8
110 Employee 8:
111 Name:
112 Id: 0
113 Age: 0
114 Writing employee 9
115 Employee 9:
116 Name:
117 Id: 0
118 Age: 0
119 Writing employee 10
120 Employee 10:
121 Name:
122 Id: 0
123 Age: 0
124 Reading data from the file.
125 Employee 1:
126 Name:
127 Id: 0
128 Age: 0
129
130 Employee 2:
131 Name:
132 Id: 0
133 Age: 0
134
135 Employee 3:
136 Name: Krish
137 Id: 12
138 Age: 21
139
140 Employee 4:
```

## *Advanced Data Structures - Assignment 7*

---

```
141 Name: Ram
142 Id: 23
143 Age: 32
144
145 Employee 5:
146 Name: Ramesh
147 Id: 24
148 Age: 21
149
150 Employee 6:
151 Name: Part
152 Id: 42
153 Age: 22
154
155 Employee 7:
156 Name:
157 Id: 0
158 Age: 0
159
160 Employee 8:
161 Name:
162 Id: 0
163 Age: 0
164
165 Employee 9:
166 Name:
167 Id: 0
168 Age: 0
169
170 Employee 10:
171 Name:
172 Id: 0
173 Age: 0
174
175 Enter size of table:
176 2
177 Enter data for employee 1
178 Enter name:
179 krish
180 Enter id:
181 124
182 Enter age:
183 21
184
185
186 Employee 1:
187 Name: krish
188 Id: 124
189 Age: 21
190
191 Enter data for employee 2
192 Enter name:
193 Tony
194 Enter id:
195 4
196 Enter age:
197 23
198
199
```

## *Advanced Data Structures - Assignment 7*

---

```
200 Employee 2:
201 Name: Tony
202 Id: 4
203 Age: 23
204
205 Enter 1 to insert with replacement, 2 to insert without replacement:
206 2
207 Hash table now looks like this.
208 0:-1
209 1:-1
210 2:-1
211 3:-1
212 4:124
213 5:4
214 6:-1
215 7:-1
216 8:-1
217 9:-1
218
219 Inserting data into the file.
220 Writing employee 1
221 Employee 1:
222 Name:
223 Id: 0
224 Age: 0
225 Writing employee 2
226 Employee 2:
227 Name:
228 Id: 0
229 Age: 0
230 Writing employee 3
231 Employee 3:
232 Name:
233 Id: 0
234 Age: 0
235 Writing employee 4
236 Employee 4:
237 Name:
238 Id: 0
239 Age: 0
240 Writing employee 5
241 Employee 5:
242 Name: krish
243 Id: 124
244 Age: 21
245 Writing employee 6
246 Employee 6:
247 Name: Tony
248 Id: 4
249 Age: 23
250 Writing employee 7
251 Employee 7:
252 Name:
253 Id: 0
254 Age: 0
255 Writing employee 8
256 Employee 8:
257 Name:
258 Id: 0
```

```
259 Age: 0
260 Writing employee 9
261 Employee 9:
262 Name:
263 Id: 0
264 Age: 0
265 Writing employee 10
266 Employee 10:
267 Name:
268 Id: 0
269 Age: 0
270 Reading data from the file.
271 Employee 1:
272 Name:
273 Id: 0
274 Age: 0
275
276 Employee 2:
277 Name:
278 Id: 0
279 Age: 0
280
281 Employee 3:
282 Name:
283 Id: 0
284 Age: 0
285
286 Employee 4:
287 Name:
288 Id: 0
289 Age: 0
290
291 Employee 5:
292 Name: krish
293 Id: 124
294 Age: 21
295
296 Employee 6:
297 Name: Tony
298 Id: 4
299 Age: 23
300
301 Employee 7:
302 Name:
303 Id: 0
304 Age: 0
305
306 Employee 8:
307 Name:
308 Id: 0
309 Age: 0
310
311 Employee 9:
312 Name:
313 Id: 0
314 Age: 0
315
316 Employee 10:
317 Name:
```



```
318 Id: 0
319 Age: 0
```

### **11 Conclusion**

Thus, we have understood the importance and use of Heaps as a Data structure, and how they are better and more efficient than Binary Search Trees. We have also understood the working of Heap Sort and how it is implemented.

## 12 FAQ

### 1. Discuss with suitable example for heap sort?

Heap sort is a comparison-based sorting algorithm that works by first organizing the data to be sorted into a binary heap. The heap is then repeatedly reduced to a sorted array by extracting the largest element from the heap and inserting it into the output array. The heap is reconstructed after each extraction.

An example of heap sort can be shown using the following array of numbers:

[12, 11, 13, 5, 6, 7]

First, we build a max heap from the given array. The max heap is a binary tree where the parent node is greater than or equal to its children. After building the max heap, the array becomes:

[13, 11, 12, 5, 6, 7]

The first element of the array is the largest number in the heap, so we move it to the end of the array and reduce the heap size by one. The array now becomes:

[7, 11, 12, 5, 6, 13]

We then rebuild the max heap from the remaining elements and repeat the process until the heap is empty. The sorted array is obtained by repeatedly extracting the maximum element from the heap and appending it to the output array. The final sorted array is:

[5, 6, 7, 11, 12, 13]

### 2. Compute the time complexity of heap sort?

The time complexity of heap sort is  $O(n \log n)$  in the worst case, where  $n$  is the number of elements to be sorted. This is because the max heap can be built in  $O(n)$  time and each extraction from the heap takes  $O(\log n)$  time. Therefore, the total time complexity of heap sort is  $O(n \log n)$ . Heap sort is efficient for large datasets and is a good choice when a stable sort is not required.