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

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.

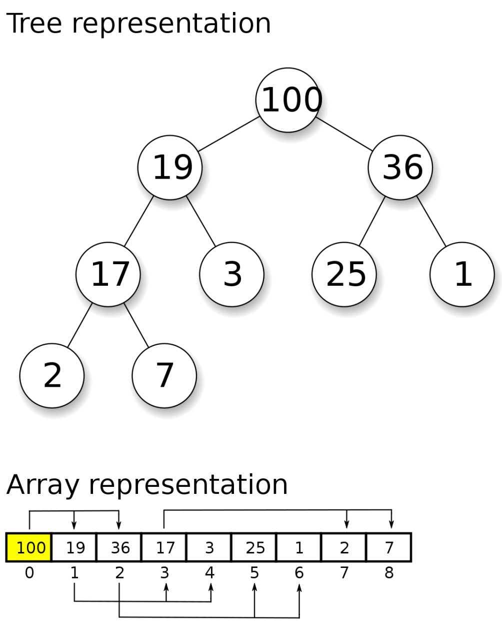

Figure 1:

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.

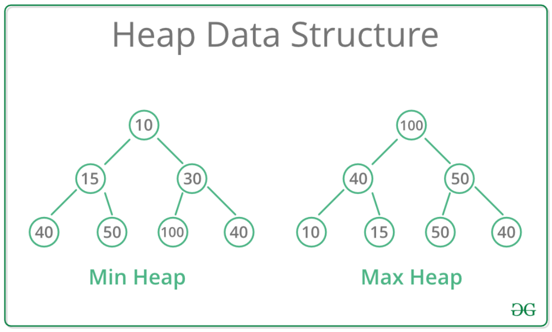

Figure 2: Min Heap and Max Heap

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.

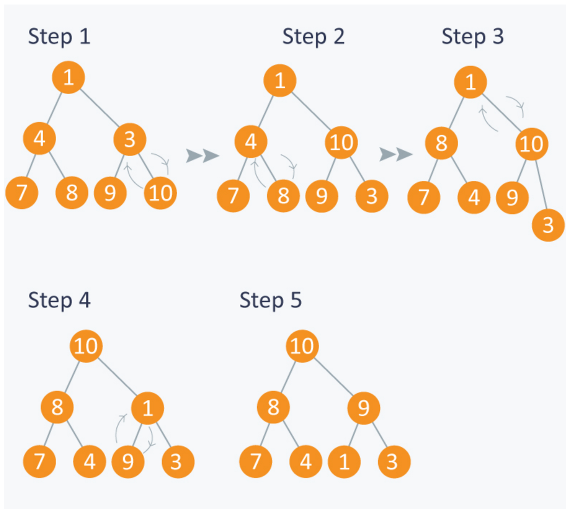

Figure 3:

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

```
CreateMaxHeap(array)
    for i from n/2 down to 1 do:
        MaxHeapify(array, i, n)

CreateMinHeap(array)
    for i from n/2 down to 1 do:
        MinHeapify(array, i, n)
```

Pseudo Code for Heapify function

```
MaxHeapify(array, i, n)
    left = 2i
    right = 2i + 1
    largest = i

    if left <= n and array[left] > array[largest] then:
        largest = left
    if right <= n and array[right] > array[largest] then:
        largest = right
    if largest != i then:
        swap(array[i], array[largest])
        MaxHeapify(array, largest, n)

MinHeapify(array, i, n)
    left = 2i
    right = 2i + 1
    smallest = i

    if left <= n and array[left] < array[smallest] then:
        smallest = left
    if right <= n and array[right] < array[smallest] then:
        smallest = right
    if smallest != i then:
        swap(array[i], array[smallest])
        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

```
// Program for Creating Heaps and Heap Sort
// 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.

#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <algorithm>
using namespace std;


class Heap
{
    vector<int> v;
    int n;
    int left(int i)
    {
        return 2*i+1;
    }
    int right(int i)
    {
        return 2*i+2;
    }
    int parent(int i)
    {
        return (i-1)/2;
    }
    void heapify(int i)
    {
        int l=left(i);
        int r=right(i);
        int largest=i;
        if(l<n && v[l]>v[i])
            largest=l;
        if(r<n && v[r]>v[largest])
            largest=r;
        if(largest!=i)
        {
            swap(v[i],v[largest]);
            heapify(largest);
        }
    }

public:

    Heap()
    {
        n=0;
    }
    void insert(int x)
    {
        v.push_back(x);
        n++;
        int i=n-1;
        while(i>0 && v[parent(i)]<v[i])
        {
            swap(v[i],v[parent(i)]);
            i=parent(i);
        }
    }
    void display()
    {
        for(int i=0;i<n;i++)
            cout<<v[i]<<" ";
        cout<<endl;
    }
    void heapsort()
    {
        for(int i=n-1;i>=0;i--)
        {
            swap(v[0],v[i]);
            n--;
            heapify(0);
        }
    }
    int getmax()
    {
        return v[0];
    }
    int getmin()
    {
        return v[n-1];
    }
    void buildheap()
    {
        for(int i=n/2-1;i>=0;i--)
            heapify(i);
    }
    void swap(int &a,int &b)
    {
        int temp=a;
        a=b;
        b=temp;
    }

};

int main()
{
    Heap h;
    int ch;
    int x;
    do
    {
        cout<<"1.Insert\n2.Display\n3.Heapsort\n4.Get Max\n5.Get Min\n6.Build Heap
    \n7.Exit\n";
        cin>>ch;
        switch(ch)
        {
        case 1:
            cout<<"Enter the element to be inserted: ";
            cin>>x;
            h.insert(x);
            break;
        case 2:
            h.display();
            break;
        case 3:
            h.heapsort();
            break;
        case 4:
            cout<<"Maximum element: "<<h.getmax()<<endl;
            break;
        case 5:
            cout<<"Minimum element: "<<h.getmin()<<endl;
            break;
        case 6:
            h.buildheap();
            break;
        case 7:
            break;
        default:
            cout<<"Invalid Choice\n";
        }
    }while(ch!=7);
    return 0;
}

// #include <iostream>
// #include <algorithm>

// using namespace std;

// class Heap
// {
// private:
//     int *arr;
//     int size;

//     void heapify(int i)
//     {
//         int largest = i;
//         int left = 2 * i + 1;
//         int right = 2 * i + 2;

//         if (left < size && arr[left] > arr[largest])
//         {
//             largest = left;
//         }

//         if (right < size && arr[right] > arr[largest])
//         {
//             largest = right;
//         }

//         if (largest != i)
//         {
//             swap(arr[i], arr[largest]);
//             heapify(largest);
//         }
//     }

// public:
//     Heap(int *arr, int n)
//     {
//         this->arr = arr;
//         this->size = n;

//         // build max heap
//         for (int i = n / 2 - 1; i >= 0; i--)
//         {
//             heapify(i);
//         }
//     }

//     void heapSort()
//     {
//         // sort the array using heap sort
//         for (int i = size - 1; i >= 0; i--)
//         {
//             swap(arr[0], arr[i]);
//              size--;
//              heapify(0);
//          }
//      }

//      int getMin()
//      {
//          return arr[0];
//      }

//      int getMax()
//      {
//          return arr[size - 1];
//      }
// };

// int main()
// {
//      // input marks obtained by students
//      int marks[] = {80, 60, 70, 90, 85};

//      // calculate number of students
//      int n = sizeof(marks) / sizeof(marks[0]);

//      // create heap object
//      Heap heap(marks, n);

//      // perform heap sort to find max and min marks
//      heap.heapSort();

//      // print max and min marks
//      cout << "Maximum marks: " << heap.getMax() << endl;
//      cout << "Minimum marks: " << heap.getMin() << endl;

//      return 0;
// }
```

```
Enter size of table:
4
Enter data for employee 1
Enter name:
Krish
Enter id:
12
Enter age:
21


Employee 1:
Name: Krish
Id: 12
Age: 21

Enter data for employee 2
Enter name:
Part
Enter id:
42
Enter age:
```

```
22


Employee 2:
Name: Part
Id: 42
Age: 22

Enter data for employee 3
Enter name:
Ram
Enter id:
23
Enter age:
32


Employee 3:
Name: Ram
Id: 23
Age: 32

Enter data for employee 4
Enter name:
Ramesh
Enter id:
24
Enter age:
21


Employee 4:
Name: Ramesh
Id: 24
Age: 21

Enter 1 to insert with replacement, 2 to insert without replacement:
1
Hash table now looks like this.
0:-1
1:-1
2:12
3:23
4:24
5:42
6:-1
7:-1
8:-1
9:-1

Inserting data into the file.
Writing employee 1
Employee 1:
Name:
Id: 0
Age: 0
Writing employee 2
Employee 2:
Name:
```

```
Id: 0
Age: 0
Writing employee 3
Employee 3:
Name: Krish
Id: 12
Age: 21
Writing employee 4
Employee 4:
Name: Ram
Id: 23
Age: 32
Writing employee 5
Employee 5:
Name: Ramesh
Id: 24
Age: 21
Writing employee 6
Employee 6:
Name: Part
Id: 42
Age: 22
Writing employee 7
Employee 7:
Name:
Id: 0
Age: 0
Writing employee 8
Employee 8:
Name:
Id: 0
Age: 0
Writing employee 9
Employee 9:
Name:
Id: 0
Age: 0
Writing employee 10
Employee 10:
Name:
Id: 0
Age: 0
Reading data from the file.
Employee 1:
Name:
Id: 0
Age: 0

Employee 2:
Name:
Id: 0
Age: 0

Employee 3:
Name: Krish
Id: 12
Age: 21

Employee 4:
```

```
Name: Ram
Id: 23
Age: 32

Employee 5:
Name: Ramesh
Id: 24
Age: 21

Employee 6:
Name: Part
Id: 42
Age: 22

Employee 7:
Name:
Id: 0
Age: 0

Employee 8:
Name:
Id: 0
Age: 0

Employee 9:
Name:
Id: 0
Age: 0

Employee 10:
Name:
Id: 0
Age: 0

Enter size of table:
2
Enter data for employee 1
Enter name:
krish
Enter id:
124
Enter age:
21


Employee 1:
Name: krish
Id: 124
Age: 21

Enter data for employee 2
Enter name:
Tony
Enter id:
4
Enter age:
23


```

```
Employee 2:
Name: Tony
Id: 4
Age: 23

Enter 1 to insert with replacement, 2 to insert without replacement:
2
Hash table now looks like this.
0:-1
1:-1
2:-1
3:-1
4:124
5:4
6:-1
7:-1
8:-1
9:-1

Inserting data into the file.
Writing employee 1
Employee 1:
Name:
Id: 0
Age: 0
Writing employee 2
Employee 2:
Name:
Id: 0
Age: 0
Writing employee 3
Employee 3:
Name:
Id: 0
Age: 0
Writing employee 4
Employee 4:
Name:
Id: 0
Age: 0
Writing employee 5
Employee 5:
Name: krish
Id: 124
Age: 21
Writing employee 6
Employee 6:
Name: Tony
Id: 4
Age: 23
Writing employee 7
Employee 7:
Name:
Id: 0
Age: 0
Writing employee 8
Employee 8:
Name:
Id: 0
```

```
Age: 0
Writing employee 9
Employee 9:
Name:
Id: 0
Age: 0
Writing employee 10
Employee 10:
Name:
Id: 0
Age: 0
Reading data from the file.
Employee 1:
Name:
Id: 0
Age: 0

Employee 2:
Name:
Id: 0
Age: 0

Employee 3:
Name:
Id: 0
Age: 0

Employee 4:
Name:
Id: 0
Age: 0

Employee 5:
Name: krish
Id: 124
Age: 21

Employee 6:
Name: Tony
Id: 4
Age: 23

Employee 7:
Name:
Id: 0
Age: 0

Employee 8:
Name:
Id: 0
Age: 0

Employee 9:
Name:
Id: 0
Age: 0

Employee 10:
Name:
```

```
Id: 0
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.