SEPTEMBER 08,2020

DESIGN AND ANALYSIS OF ALGORITHMS

(QUESTION: 4, HEAP SORT)
EXERCISE 4

SUBMITTED BY-
NAME - RAJ KRISHNA
ROLL NO - 1805233
BATCH - CSE-G1
GROUP - B

1. The objective of the Experiment

The objective of the experiment is to sort the numbers present in the given array using **Heap Sort**.

2. Solution Code

```
#include <iostream>
#include <vector>
using namespace std;

int count = 0;
void print(vector<int> v)
{
  for(int i = 0; i < v.size(); i++)
        cout << v[i] << " ";
  cout << endl;
}
void max_heapify(vector<int>& arr, int i, int size)
{
        int largest, l = (2*i) + 1, r = (2*i) + 2;

        if(l < size && arr[l] > arr[i])
        {
                count++;
                largest = l;
        }
        else
        {
                count++;
                largest = i;
        }
```

```
        if(r < size && arr[r] > arr[largest])
        {
                count++;
                largest = r;
        }

        if(largest != i)
        {
                swap(arr[i], arr[largest]);
                max_heapify(arr, largest, size);
        }

}

void build_max_heap(vector<int>& arr)
{
  int i;
  for(i = (arr.size() / 2); i >= 0; i--)
  max_heapify(arr, i, arr.size());

  cout<<"\nMax-Heap- "<<endl;
  print(arr);
}

void heap_sort(vector<int>& arr)
{
  build_max_heap(arr);
  int size = arr.size();
  for(int i = arr.size() - 1; i > 0; i--)
  {
       swap(arr[0], arr[i]);
       size--;
       max_heapify(arr, 0, size);
```

```
        }
}

int main()
{
        vector<int> arr { 7,9,2,11,19,17,12,5,7,12 };

        cout<<"Array Initialisation- "<<endl;
        print(arr);

        heap_sort(arr);
        cout<<"\nAfter HeapSort- "<<endl;
        print(arr);

        cout<<"\nNo of comparisons- "<<count<<endl;
        return 0;
}
```

3. Summary of the program

Heap Sort is a popular and efficient sorting algorithm. Heap sort works by visualizing the elements of the **array** as a special kind of **complete binary tree** called a **heap**. A complete binary tree has an interesting property that we can use to find the children and parents of any node.

If the index of any element in the array is **i**, the element in the index **2i+1** will become the **left child** and element in **2i+2** index will become the **right child**. Also, the **parent** of any element at index **i** is given by the **lower bound of (i-1)/2**.

The **Heap Data Structure** is used to implement **Heap Sort**.

Heap is a special tree-based data structure. A **binary tree** is said to follow a **heap data structure** if

- ➔ it is a **complete binary tree**
- ➔ All nodes in the tree follow the property that they are **greater** than their children i.e. the largest element is at the root and both its children and smaller than the root and so on. Such a heap is called a **max-heap**. If instead, all nodes are **smaller** than their children, it is called a **min-heap**.

Starting from a **complete binary tree**, we can modify it to become a **Max-Heap** by running a function called **heapify** on all the **non-leaf elements** of the **heap**.

There are two scenarios -

1. The root is the largest element and we don't need to do anything.
2. The root had a larger element as a child and we needed to swap to maintain max-heap property.

We can combine both these conditions in one heapify function.

To build a max-heap from any tree, we can thus start heapifying each sub-tree from the bottom up and end up with a max-heap after the function is applied to all the elements including the root element.

In the case of a complete tree, the first index of a **non-leaf node** is given by **n/2 - 1**. All other nodes after that are leaf-nodes and thus don't need to be heapified.
We start by heapifying the lowest smallest trees and gradually move up until we reach the root element.

How Heap Sort Works?

1. Since the tree satisfies Max-Heap property, then the largest item is stored at the root node.
2. **Swap**: Remove the root element and put at the end of the array (nth position) Put the last item of the tree (heap) at the vacant place.
3. **Remove**: Reduce the size of the heap by 1.
4. **Heapify**: Heapify the root element again so that we have the highest element at root.

The process is repeated until all the items of the list are sorted.

Heap sort algorithms with the following time complexity.
Best Case - $O(n*\log_2 n)$
Average Case - $O(n*\log_2 n)$
Worst Case - $O(n*\log_2 n)$

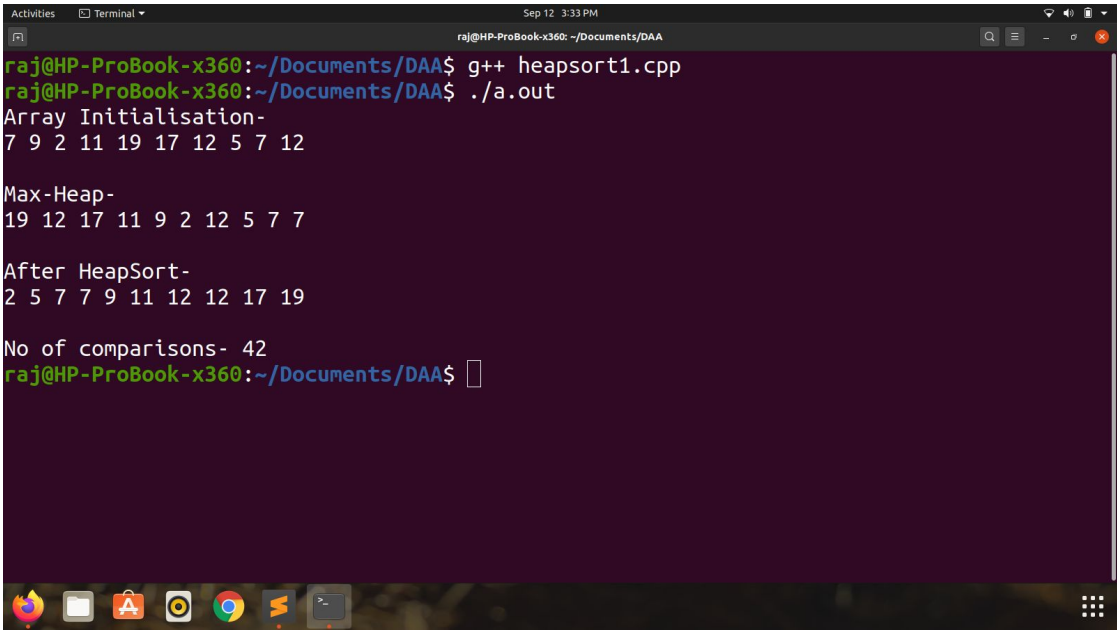
4. Sample Output

Array Initialisation-
7 9 2 11 19 17 12 5 7 12

Max-Heap-
19 12 17 11 9 2 12 5 7 7

After HeapSort-
2 5 7 7 9 11 12 12 17 19

No of comparisons- 42


Activities
Terminal
Sep 12 3:33 PM
raj@HP-ProBook-x360: ~/Documents/DAA
raj@HP-ProBook-x360:~/Documents/DAA$ g++ heapsort1.cpp
raj@HP-ProBook-x360:~/Documents/DAA$ ./a.out
Array Initialisation-
7 9 2 11 19 17 12 5 7 12
Max-Heap-
19 12 17 11 9 2 12 5 7 7
After HeapSort-
2 5 7 7 9 11 12 12 17 19
No of comparisons- 42
raj@HP-ProBook-x360:~/Documents/DAA$