CS217 DSAA Homework-4

HONGLI YE 12311501

October 14^{th} 2024

Contents

1 Question 1

Say whether the following array is a $Max - Heap$ (justify your answer):

34	20	21	16	14	11	3	14	17	13

Table 1: Question 1

Answer: The heap structure is shown below.

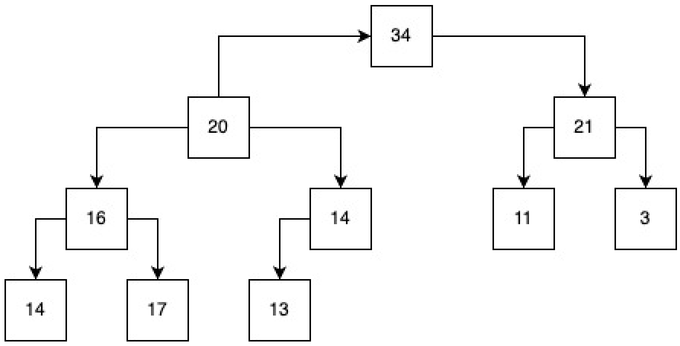

Figure 1: Question 1

Obviously, $A[3] = 16 < 17 = A[8]$, so this is not a max-heap.

2 Question 2

Consider the following input for $HeapSort$:

12	10	4	2	9	6	2	25	8

Table 2: Question 2

Create a heap from the given array and sort it by executing HeapSort. Draw the heap (the tree) after Build-Max-Heap and after every execution of Max-Heapify in line 5 of HeapSort. You don't need to draw elements extracted from the heap, but you can if you wish.

Answer: The picture is shown below

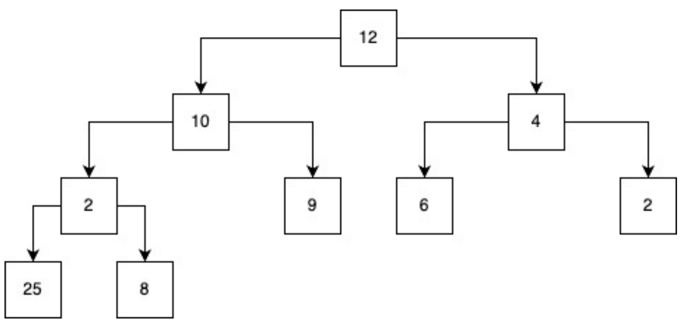

Figure 2: Step 0

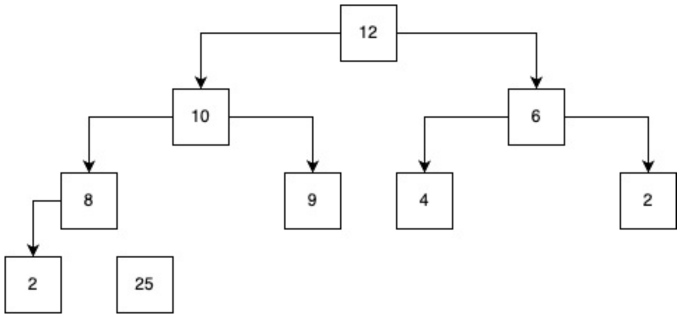

Figure 3: Step 1

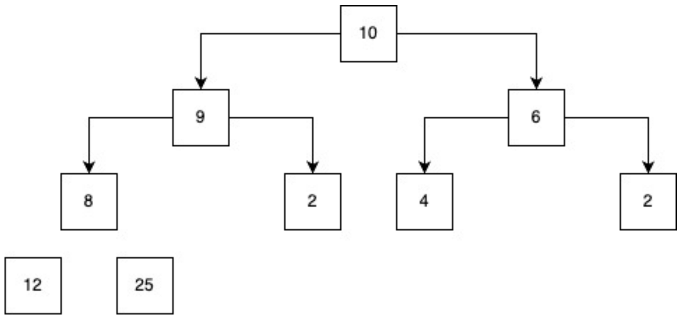

Figure 4: Step 2

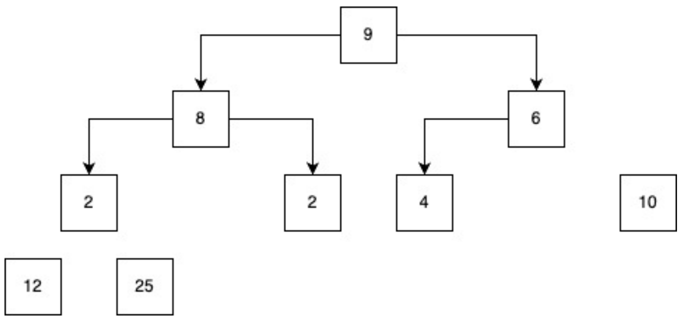

Figure 5: Step 3

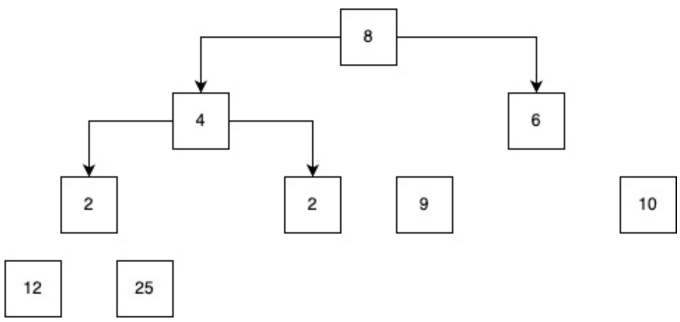

Figure 6: Step 4

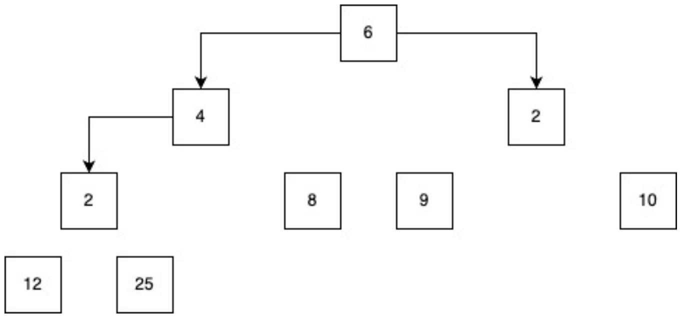

Figure 7: Step 5

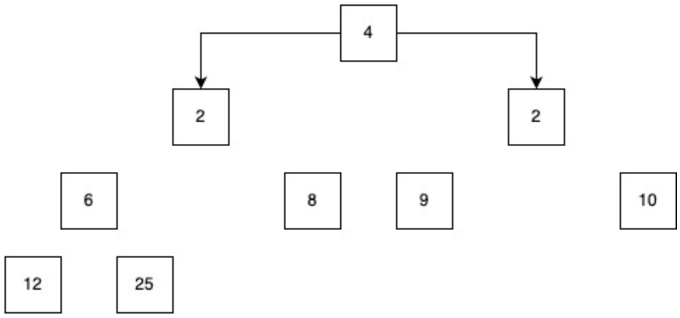

Figure 8: Step 6

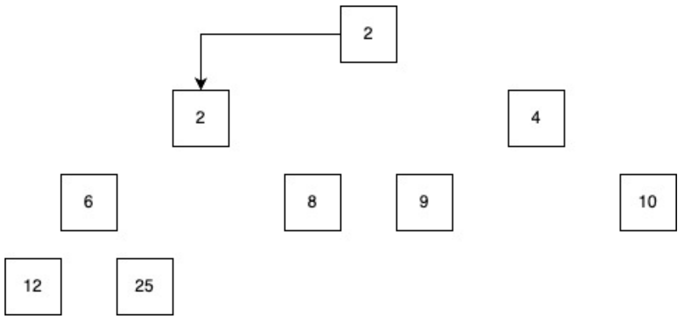

Figure 9: Step 7

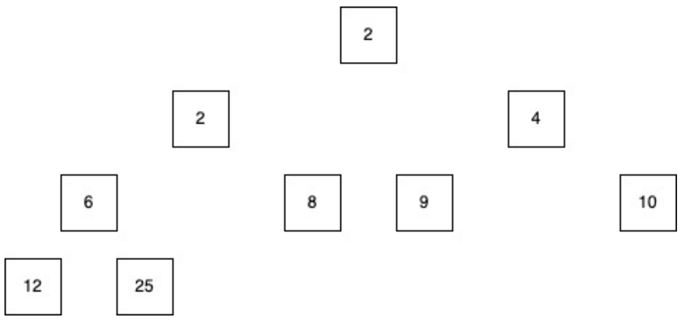

Figure 10: Step 8

3 Question 3

1. Provide the pseudo-code of a $Max-Heapify(A, i)$ algorithm that uses a WHILE loop instead of the recursion used by the algorithm shown at lecture.

2. Prove correctness of the algorithm by loop invariant.

Answer:

1. For question 1:

Algorithm 1 Max-Heapify(A, i, n)

$heapsize \leftarrow n$
while $True$ **do**
  $left \leftarrow 2 * i + 1$
  $right \leftarrow 2 * i + 2$
  $largest \leftarrow i$
  if $left < heapsize$ ***and*** $A[left] > A[largest]$ **then**
  $largest \leftarrow left$
  if $right < heapsize$ ***and*** $A[right] > A[largest]$ **then**
  $largest \leftarrow right$
  if $largest \neq i$ **then**
  swap $A[i]$ **and** $A[largest]$
  $i \leftarrow largest$
  else
  break

2. For question 2:
Before the proof, we need to claim some definitions:
Define the leaf to be the node without children.
Define the height of a node as the longest number of simple downward edges from the node to a leaf.
Loop Invariant: After i iterations of the loop which is taking away the maximum point $A[0]$, at least the left and right subtrees are still max-heap.
Iteration: Since the left and right subtrees are max-heap, so the maximum in the heap $= max(A[0], A[1], A[2])$, suppose the left tree swap with the $A[0]$, then we consider the left subtree as a new independent tree to be sorted. Obviously, the new tree's left and right subtrees are still max-heap.
Termination: There are two possible situations.
First, the algorithm terminated because of the iteration reaches leaf. Then there a no subtrees, the max-heap property is satisfied trivially. Second, the algorithm terminated because of $largest = i$, then it means the subtree satisfies the max-heap property.
All in all, the algorithm is well-designed.

4 Question 4

1. Show that each child of the root of an n-node heap is the root of a sub-tree of at most (2/3)n nodes. (HINT: consider that the maximum number of elements in a subtree happens when the left subtree has the last level full and the right tree has the last level empty. You might want to use the formula seen at lecture: $\Sigma_{i=0}^{k-1}(2^i) = 2^k - 1$

2. As a consequence of (1) we can use the recurrence equation $T(n) \leq T(2n/3) + \Theta(1)$ to describe the runtime of $Max - Heapify(A, n)$. Prove the runtime of $Max - Heapify$ using the Master Theorem.

Answer:

1. Since heap is a maximum binary tree, if the heap has h layers, then all nodes in $[1, h - 1]$ are not empty. So, the largest percent a subtree can hold is when the left subtree's h's layer is full while right is empty.

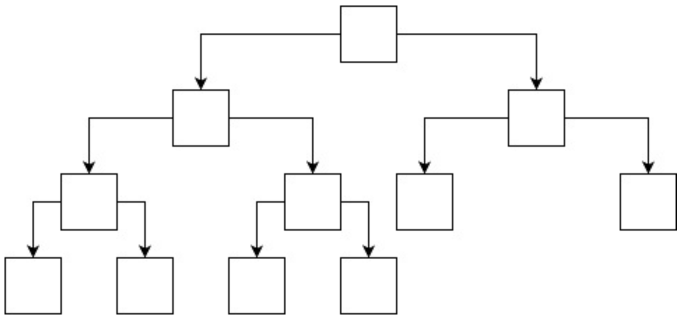

Figure 11: Example

First we calculate the number of all nodes: $\Sigma_{i=1}^{h-1} 2^{i-1} + 2^{h-2} = 3 * 2^{h-2} - 1$
Then we calculate the number of the left subtree: $\Sigma_{i=1}^{h-1} 2^{i-1} = 2^{i-1} - 1$

$$\frac{3 * 2^{h-2} - 1}{2^{h-1} - 1} < \frac{2}{3}$$

2. Watershed function is $n^{log_{\frac{2}{3}}(1)} = n^0 = 1$. Since $\Theta(1) = \Theta(1 \times log^0(b))$
   So: $T(n) = O(logn)$

5 Question 5

Argue that the runtime of $HeapSort$ on an already sorted array of distinct numbers is $\Omega(nlogn)$.

Answer:

Consider an array that is already sorted be:$\{x_1, x_2, \dots, x_n\}$, where $\forall\ i < j, x_i < x_j$.
The initial call is $heap\_sort(arr, n)$.

```
void heap_sort(int arr[], int n)
{
    BuildHeap(arr, n);

    for(int i = n-1; i>=1; i--)
    {
        swap(arr[0], arr[i]);
        Heaplify(arr, 0, i);
    }
}
```

So:

$$T(n) = B(n) + nc_0 + (n-1)c_1 + \Sigma_{i=1}^{n-1}(H(i))$$

where $B(n)$ is the time cost of the function $BuildHeap(arr, n)$, and $H(i) = Heaplify(arr, 0, i)$. Since the array is already sorted. So:$B(n) = \Theta(n)$ and $H(i) = [log(i)] = \Theta(log(i))$ So:

$$T(n) \leq \Theta(n) + (c_1 + c_0)n - c_1 + \Sigma_{i=1}^{n-1}(\Theta(log(i)))$$

$$T(n) = \Theta(n) + \Omega(nlogn) = \Omega(nlogn)$$

6 Question 6

Implement $HeapSort(A, n)$.

Answer:

I Already finish that on OJ.

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

void Heaplify(int arr[], int i, int n)
{
    int largest;
    int l = 2*i + 1;
    int r = 2*i + 2;
    if ( l <=  n-1  && arr[l] > arr[i] )
    {
        largest = l;
    }
    else
    {
        largest = i;
    }
    //for left side
    if ( r <=  n-1 && arr[r] > arr[largest] )
    {
        largest = r;
    }
    //for right side
    if ( largest != i )
    {
        swap(arr[i], arr[largest]);
        Heaplify(arr, largest, n);
    }
}

void BuildHeap(int arr[], int n)
{
    for (int i=n/2-1 ; i>=0 ;i--)
    {
        Heaplify(arr, i , n);
    }
}

void heap_sort(int arr[], int n)
{
    BuildHeap(arr, n);

    for(int i = n-1; i>=1; i--)
    {
        swap(arr[0], arr[i]);
        Heaplify(arr, 0, i);
    }
}

```

```
int main()
{
    int n;
    cin >> n;
    int arr[n];
    for(int i = 0; i < n; i++)
    {
        cin >> arr[i];
    }

    heap_sort(arr,n);

    for(int i = 0; i < n; i++)
    {
        cout << arr[i] << " ";
    }
    return 0;
}
```