[CS203] Design and Analysis of Algorithms

Course Instructor: Dr. Dibyendu Roy — Autumn 2023-24
Scribed by: Prakhar Jain (202251099) — Lecture (Weak 5)

Algorithm 1 Build-Max-Heap

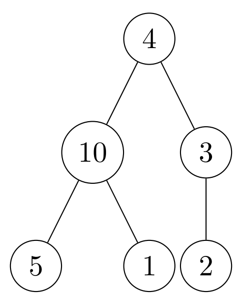
```
procedure BUILD-MAX-HEAP(A)
    A.heapsize ← A.length
    for i ← ⌊A.length/2⌋ downto 1 do
        MAX-HEAPIFY(A, i)
    end for
end procedure
```

Example: Build-Max-Heap

Initial Array:

$$A = [4, 10, 3, 5, 1, 2]$$

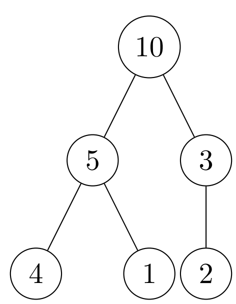
Initial Heap Tree:



After Applying Build-Max-Heap:
Max Heap Array:

$$A = [10, 5, 3, 4, 1, 2]$$

Max Heap Tree:



1 Correctness of Build-Max-Heap Algorithm

1.1 Initialization

- Initially, the array A is an unsorted array.
- The heap size A.heapsize is set to the length of the array.
- The for loop starts from $i = \lfloor A.\text{length}/2 \rfloor$ down to 1. This is the region where all subtrees of A rooted at nodes $i+1, i+2, \ldots, A.\text{length}$ are already max-heaps.

1.2 Maintenance

- The loop maintains the max-heap property of the array. In each iteration, the Max-Heapify procedure is called on node i, which ensures that the subtree rooted at node i satisfies the max-heap property.

1.3 Termination

- After the for loop finishes, all subtrees of A rooted at nodes $1, 2, \ldots, \lfloor A.\text{length}/2 \rfloor$ are max-heaps.
- The entire array A satisfies the max-heap property.
- The algorithm terminates, and A is a max-heap.

2 Time Complexity Analysis

The time complexity of the Build-Max-Heap algorithm is $O(n \log n)$, where n is the number of elements in the array.

- The for loop runs from $i = \lfloor n/2 \rfloor$ down to 1. In each iteration, Max-Heapify is called on node i.
- The Max-Heapify operation takes $O(\log n)$ time, where $\log n$ is the height of the heap.
- The total time complexity is the sum of the Max-Heapify operations for all nodes in the tree.
- The height of a binary heap is $\log n$, and there are n nodes.
- So, the total time complexity is $O(n \log n)$.

3 Deriving Time Complexity

Let's derive the time complexity of calling Max-Heapify at height h.

- Calling Max-Heapify at height h will take $O(h)$ time, represented as c_h.
- The sum of Max-Heapify calls at different heights is:

$$\sum_{h=0}^{\lfloor \log n \rfloor} \lceil \frac{n}{2^h} \rceil \cdot c_h$$

- We want to show that this sum is $O(n)$. Our goal is to derive it to $c_1 \cdot n$.
- The sum of Max-Heapify calls at different heights is:

$$\sum_{h=0}^{\lfloor \log n \rfloor} \lceil \frac{n}{2^h} \rceil \cdot c_h$$

- We want to derive this sum to $c_1 \cdot n$.

To achieve this, let's first simplify the expression by analyzing the terms:

- The term $\lceil \frac{n}{2^h} \rceil$ represents the number of nodes at height h in the heap.
- We know that the height of a binary heap is $\log n$, which means there are at most $\log n + 1$ different heights.
- So, we can simplify the sum to:

$$\sum_{h=0}^{\log n} \lceil \frac{n}{2^h} \rceil \cdot c_h$$

- Notice that c_h is a constant for each height h.
- Let's analyze the term $\lceil \frac{n}{2^h} \rceil$:
  - When $h = 0$, the term is $\lceil \frac{n}{1} \rceil = n$.
  - When $h = 1$, the term is $\lceil \frac{n}{2} \rceil = \lceil \frac{n}{2} \rceil$.
  - When $h = 2$, the term is $\lceil \frac{n}{4} \rceil = \lceil \frac{n}{4} \rceil$.
  - And so on...

To simplify further, we can introduce a new variable j to represent the value of $\lceil \frac{n}{2^h} \rceil$ when $h = j$. Now, let's rewrite the sum with this new variable:

$$\sum_{j=0}^{\log n} j \cdot c_j$$

Now, to show that this is $O(n)$, we need to bound c_j by a constant and analyze the summation. The idea is to show that the growth of the summation is linear in terms of n.

Let's proceed with this simplification:

- We want to show that each c_j is bounded by a constant. This means there exists some k such that $c_j \leq k$ for all j.
- Now, we can bound the summation by the maximum value of j times k:

$$\sum_{j=0}^{\log n} j \cdot c_j \leq k \cdot \sum_{j=0}^{\log n} j$$

- We know that $\sum_{j=0}^{\log n} j = \frac{(\log n)(\log n+1)}{2} \leq \frac{(\log n)(\log n)}{2} = \frac{(\log n)^2}{2}$.

- Therefore, the summation is bounded by $k \cdot \frac{(\log n)^2}{2}$.

The complexity of the Build-Max-Heap algorithm is given by:

$$\sum_{h=0}^{\lfloor \log n \rfloor} \lceil \frac{n}{2^h} \rceil \cdot c_h = O(n)$$

Algorithm 2 Heap Sort

procedure HEAP-SORT(A)
    BUILD-MAX-HEAP(A) ▷ Step 1: Build a max heap
    for $i \leftarrow A$.length **downto** 2 **do** ▷ Step 2: Sort the array
        EXCHANGE($A[1], A[i]$) ▷ Swap root (largest) with last element
        A.heapsize $\leftarrow A$.heapsize $- 1$ ▷ Exclude sorted element
        MAX-HEAPIFY($A, 1$) ▷ Restore max heap property
    end for
end procedure

To analyze the time complexity of Heap Sort, we start with the recurrence relation:

$$T(n) = T(1) + \log(n!)$$

1. **Initialization (Base Case)**: When the input size is 1, we have $T(1)$, which is a constant time operation. This represents the base case.

2. **Recurrence Relation**: The recurrence relation $T(n) = T(1) + \log(n!)$ signifies that we break down the problem into smaller subproblems of size $\frac{n}{2}$, perform some work in merging, and then proceed to the next level of recursion until we reach the base case.

Now, let's derive this expression to $n \log(n) - n + 1$:

$$\begin{aligned} T(n) &= T(1) + \log(n!) \\ &= T(1) + \log(n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 3 \cdot 2 \cdot 1) \\ &= T(1) + \log(n) + \log(n-1) + \log(n-2) + \ldots + \log(3) + \log(2) + \log(1) \\ &\leq T(1) + n\log(n) \quad (\text{since } \log(k) \leq \log(n) \text{ for } k = 1, 2, \ldots, n-1) \\ &= O(n \log(n)) \quad (\text{ignoring the constant } T(1)) \end{aligned}$$

So, the time complexity of the Heap Sort algorithm is $O(n \log(n))$.

4 Priority Queue Operations

A Priority Queue (PQ) is a data structure that maintains a set S, where every element in S is associated with a value called a key. Here are the main operations of a Priority Queue:

4.1 Insert(x)

Adds element x to the set S.

4.2 Maximum(S)

Returns the maximum element in S (commonly found using a max-heap).

4.3 Extract Max

Removes and returns the maximum element from S.

4.4 Heap Max(A)

Returns the maximum element from the max-heap A.

4.5 Increase Key($S, x,$ **key**)

Increases the key associated with element x in S, where key $> S[x]$.

5 Detailed Explanation of Operations

5.1 Heap Extract Max

The operation Heap Extract Max is used to remove and return the maximum element from a max-heap.

5.1.1 Steps:

1. Exchange $A[1]$ with $A[A.\text{heapsize}]$.
2. Decrease A.heapsize by 1.
3. Compare $A[1]$ with its children to maintain the max-heap property.

5.1.2 Example:

Suppose we have a max-heap represented as an array A: $A = [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]$ After performing Heap Extract Max, the max element (16) is removed from the heap, and the max-heap property is restored. The updated heap would be: $A = [14, 8, 10, 4, 7, 9, 3, 2, 1]$

5.1.3 Time Complexity:

The time complexity of Heap Extract Max is $O(\log n)$, where n is the number of elements in the max-heap.

5.2 Heap Increase Key

The operation Heap Increase Key is used to increase the key associated with a specific element in the Priority Queue S.

5.2.1 Steps:

1. Check if the new key is greater than the current key.
2. Update $A[i]$ with the new key.
3. Compare $A[i]$ with its parent $A[\text{parent}[i]]$ to maintain the max-heap property.

5.2.2 Example:

Consider the same max-heap represented as an array A: $A = [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]$ If we want to increase the key associated with an element (e.g., $A[3]$) to 15, we follow these steps and maintain the max-heap property: $A = [16, 15, 10, 8, 7, 14, 3, 2, 4, 1]$

5.2.3 Time Complexity:

The time complexity of Heap Increase Key is $O(\log n)$, where n is the number of elements in the max-heap.

5.3 Heap Insert

The operation Heap Insert is used to add a new element with a key to the max-heap.

5.3.1 Steps:

1. Add the new element to the end of the max-heap (i.e., at $A[A.\text{heapsize} + 1]$).
2. Increase $A.$heapsize by 1.
3. Compare the new element with its parent $A[\text{parent}[A.\text{heapsize}]]$ to maintain the max-heap property.

5.3.2 Example:

Suppose we want to insert a new element with a key of 20 into the max-heap. The updated max-heap would be: $A = [20, 14, 16, 8, 7, 10, 3, 2, 4, 1, 9]$

5.3.3 Time Complexity:

The time complexity of Heap Insert is $O(\log n)$, where n is the number of elements in the max-heap.

Algorithm 3 QuickSort

```
procedure QuickSort(A, p, r)
    if p < r then
        q ← Partition(A, p, r)
        QuickSort(A, p, q − 1)
        QuickSort(A, q + 1, r)
    end if
end procedure
procedure Partition(A, p, r)
    x ← A[r]
    i ← p − 1
    for j ← p to r − 1 do
        if A[j] ≤ x then
            i ← i + 1
            Exchange(A[i], A[j])
        end if
    end for
    Exchange(A[i + 1], A[r])
    return (i + 1)
end procedure
```