CSCI 4470 Algorithms

Part II Sorting and Order Statistics

- **6 Heapsort**
- 7 Quicksort
- 8 Sorting in Linear Time
- 9 Medians and Order StatisticsHeapsort Algorithms Notes

Chapter 6: Heapsort

```
6 Heapsort
  6.1 Heaps
  6.2 Maintaining the heap property
  6.3 Building a heap
  6.4 The heapsort algorithm
  6.5 Priority queues
```

baeldung.com/cs/binary-tree-max-heapify

Fundamental Concepts in Sorting Algorithms

Given n records, $R_1 ... R_n$ (called a $file$)

- Each record R_i has a key K_i
  - May also contain other (satellite) information
- Keys are objects drawn from a set on which equality is defined
- There is an order relation ($\prec$) defined on keys, which satisfy the following properties:
- ***Trichotomy:*** For any two keys a and b, exactly one of $a \prec b$, $a = b$, or $b \prec a$ is true.
- ***Transitivity:*** For any three keys a, b, and c, if $a \prec b$ and $b \prec c$, then $a \prec c$.
- The relation $\prec$ is a ***TOTAL ORDERING (LINEAR ORDERING)*** on keys.

Basic Definition

- ***Sorting:*** determine a permutation $P = (P_1, ..., P_n)$ of n records that puts the keys in non-decreasing order $K_{p1} \prec ... \prec K * pn.$
- ***Rank:*** Given a collection of n keys, the $rank$ of a key is the number of keys that are $\prec$ than it.
  - That is, $rank(K_j) = |\{K_i | K_i \prec K_j\}|$.

- If the keys are distinct, then the ranks of a key gives its position in the output file.

Terminology

Comparison-based sort, Sorted order is completely determined by performing comparisons between keys

Stable sort, records with equal keys retain their original relative order;

- i.e. $i < j$ & $\mathrm{K_{pi}} = K_{pj} \Rightarrow P_i < P_j$

In-place sort, Needs a constant amount of extra space in addition to that needed to store keys

FULL BINARY TREE

- **Definition**: Every node has 0 or 2 children; no node can have exactly one child.

Properties:

- **Internal Nodes**: Nodes with two children.
- **External Nodes (Leaves)**: Nodes with no children.
- **Relationship**: The number of external nodes is one more than the number of internal nodes.
- **Height of the Tree** h: The number of edges on the longest path from the root node to a leaf.

$$h = Levels - 1$$

- **Number of Nodes**: TWO formulas to find the number of nodes in a full binary tree:
  - a. **Using the number of levels (L):**

$$n = 2^L - 1$$

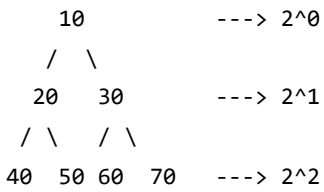
  Here, L is the total number of levels in the tree. 這裡，L 是樹中的總層級數。
  - b. **Using the height of the tree (h):**

$$n = 2^{h+1} - 1$$

  Here, h is the height of the tree, and $L = h + 1$.
- Both formulas give the correct number of nodes but use different parameters: one uses the number of levels, and the other uses the height of the tree.
- **Usage**: Utilized in algorithms and data structure implementations for their optimal and predictable structure.

Examples, Find the total nodes of a full binary tree:

Diagram 1: 3 Levels



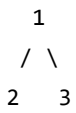
Note: All non-leaf nodes (10, 20, 30) have two children; leaves (40, 50, 60, 70) are on the same level.

> level $(L) = 3$, nodes = $2^3 - 1 = 7$, total 7 nodes, h = level - 1 = 3 - 1 = 2

> Total nodes n of a tree
>
> $$n = 2^{h+1} - 1 = 2^{2+1} - 1 = 8 - 1 = 7$$

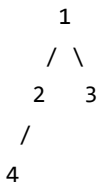
> Total nodes n of a tree
>
> $$n = 2^{L} - 1 = 2^{3} - 1 = 7$$

Diagram 2: 2 Levels



- **Note**: The non-leaf node (1) has two children; leaves (2, 3) are on the same level.
- $L = 2, n = 2^L - 1 = 2^2 - 1 = 3$
- $h = L - 1 = 2 - 1 = 1, n = 2^{h+1} - 1 = 2^{1+1} - 1 = 3$

Diagram 3: Not Full



- **Note**: Node 3 has no children, and node 2 has one child; leaves are not on the same level.

COMPLETE BINARY TREE

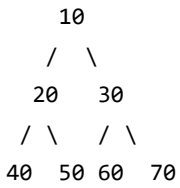
Description:

- **Note**: All levels are fully filled from left to right.

Properties:

- **Balanced**: Ensures optimal performance for operations like insertion, deletion, etc., as the tree is as balanced as possible.
- **Array Representation**: Can be efficiently represented using an array; parent at index i has children at indices $2i + 1$ and $2i + 2$.
- **Usage**: Forms the underlying data structure in heaps and facilitates efficient algorithms due to its balanced nature.

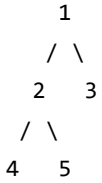
Diagram 1: Full Last Level



General Formula (Fully Filled Tree): $n = 2^{h+1} - 1$

- Maximum nodes for a tree of height h.
- Assumes all levels, including the last, are fully filled.

Diagram 2: Next-to-Last Level Full



- **Note**: The tree is complete; it is full through the next-to-last level, and the last level has leaves as far to the left as possible.

Formula (Half-Filled Last Level): $n = 2^h - 1 + \frac{2^h}{2}$

- For trees where the last level is half filled.
- Can also apply to trees with last level between half and fully filled.

Usage:

- Fully filled tree → Use general formula.
- Last level half or partially filled → Use second formula.

Formula for Total Number of Nodes

- To find the total number of nodes in a complete binary tree where **the last level is half filled**, use the following formula

$$n = 2^h - 1 + \frac{2^h}{2}$$

Where:

- $2^h - 1$ represents the total nodes excluding the last level
- $\frac{2^h}{2}$ represents the nodes in the last level, which is half filled

Calculation

- Sum up the two values to find the total nodes in the tree
- **Definition**: A binary tree that is either full or full through the next-to-last level with the last level filled from left to right.

Diagram 3: Only Root Node

1

- **Note**: The tree is technically complete with only the root node, meeting the criteria of being full through the next-to-last level.

HEAP

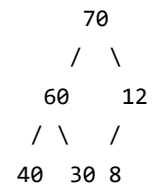
- **Definition**: A ***COMPLETE BINARY TREE*** that satisfies two main heap properties: a ***SHAPE*** property and an ***ORDER*** property.
- ***Shape Property***
  - The shape must be a complete binary tree.
- ***Order Property***
  - Each node's value is greater (max heap) or smaller (min heap) than its children, recursively for all nodes.

Array Representation of a HEAP

Array Representation:

```
Index:   [0] [1] [2] [3] [4] [5] [6]
Value:   70  60  12  40  30   8   -
```

Binary Tree Representation:



Explanation:

- **Array Index and Node Position**:
  - The array index represents the position of the nodes in the binary tree.
  - The root of the binary tree is the first element of the array (index [0]).
- **Node Relationships**:
  - For any given node at position `i`, its:
    - Left Child is at position `2i + 1`.
    - Right Child is at position `2i + 2`.
    - Parent is at position `⌊(i-1)/2⌋`.
- **Leaf Nodes**:
  - The leaf nodes are located from `tree.nodes[numElements/2]` to `tree.nodes[numElements - 1]`.

Is this a heap?

- Yes, it's a max heap. The value of each node is greater than or equal to the values of its children.

Key Points:

- **Efficiency**:
  - A heap can be efficiently represented as an array.
  - The array representation is space-efficient and simplifies heap operations.
- **Parent-Child Relationship**:
  - The parent-child relationship in the binary tree corresponds to specific indices in the array.

Finding the Height of a Max Heap

- **Number of Nodes**:
  - There are 6 nodes in the tree.
- **Finding Height**:
  - The height of a heap is given by the formula:

$$h = \lfloor \log_2(n) \rfloor$$

- Substituting the number of nodes (n = 6) into the formula:

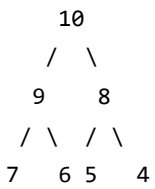
$$h = \lfloor \log_2(6) \rfloor \approx \lfloor 2.585 \rfloor = 2$$

Conclusion:

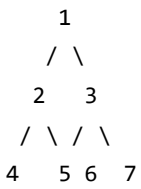
- The height of the max heap is 2.
- **Usage**:
  - **Heap Sort**: Uses heap data structure to sort an array; $O(n \log n)$ time complexity.
  - **Priority Queues**: Implements priority queues to support efficient extraction of minimum or maximum element.

Types of Heaps.

- **Max Heap**
  - In a max heap, for any given node I, the value of I is greater than or equal to the values of its children.



- **Min Heap**
  - In a min heap, for any given node I, the value of I is less than or equal to the values of its children.

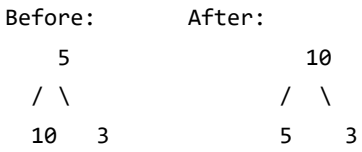


Heap Operations

- **BUILD-MAX-HEAP (or BUILD-MIN-HEAP)**: A procedure that transforms an unordered array into a max-heap or a min-heap in linear time.

Heapify

- **Heapify**: A process to adjust nodes in a binary tree to maintain the heap property.



- **Max-Heapify**: Ensures that the value of a node is greater than or equal to the values of its children. This operation is used to maintain the max-heap property.

```
Max-Heapify(A,i)
1. l = left(i)
2. r = right(i)
3. if l ≤ A.heap - size and A[l] > A[i]
4.    largest = l
5. else largest = i
6. if r ≤ A.heap - size and A[r] > A[largest]
7.    largest = r
8. if largest ‡ i
9.    exchange A[i] with A[largest]
10.   Max-Heapify(A, largest)
```

Max-Heapify Timing Analysis:

- See above `Max-Heapify(A,i)`

Worst Case Number of Nodes, of a HEAP

- Scenario: Largest is **Left Subtree** and has one more row than right subtree
- Let h = tree height, so How many nodes in each group
- n = size of input, to find size of recursive call as a function of n (f(n))

What is A(largest).heap - size as a function of n f(n):

Finding Worst Case Running Time For the given Recurrence Relation $T(n) \leq T(\frac{2n}{3}) + c$

Using simplified Masters method , $T(n) \leq T(\frac{2n}{3}) + c$

$a = 1, b = \frac{2}{3}, f(n)/k = 1$

Since $1 > \frac{2}{3}$, and $a > b^k$, Case 1 applies, **Case 1:** if $a > b^k$, then $T(n) = \Theta(n^{\log_b(a)})$

Thus we conclude that

$$T(n) = \Theta(n^{\log_b(a)}))$$

- $log_{\frac{2}{3}} 1 = 0$

How does a compare to b^k?

Worst Case Number of Nodes in a HEAP

- **Scenario 情境**: The largest element is in the **Left Subtree** which has one more level than the right subtree.

- Let h be the height of the tree.
- n is the size of the input. To find the size of the recursive call as a function of n, we denote it as $f(n)$.
- **Recursive Call Size**, To find the size of the recursive call as a function of n, denoted as $f(n)$.

Recurrence Relation

- Given by $T(n) \leq T\left(\frac{2n}{3}\right) + c$.

Determining $A(\text{largest}).\text{heap-size}$ as a function of n, $f(n)$

- In the worst case, the left subtree contains $\frac{2n}{3}$ of the total nodes, making $f(n) = \frac{2n}{3}$.

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Comparing this with your recurrence relation, we can see that $a = 1$, $b = \frac{3}{2}$, and $f(n) = c$, where c is a constant.

$$T(n) \leq T\left(\frac{2n}{3}\right) + c$$

$$T(n) \leq T\left(\frac{n}{\left(\frac{3}{2}\right)}\right) + c$$

- the fraction rule: $\frac{\frac{a}{b}}{c} = \frac{a \cdot c}{b}$
- $\frac{2n}{3} = \frac{n}{b}, b \cdot \frac{2n}{3} = \frac{n}{b} \cdot b, b \cdot \frac{2n}{3} = n,$
- $b = \frac{n}{\frac{2n}{3}}, b = \frac{3n}{2n}, b = \frac{3}{2}$

Using the Master's theorem, we can determine the case:

- $a = 1, b = \frac{3}{2}, f(n) = c$

Since $a < b^k$ where $k = 0$, Case 2 of the Master's theorem applies.
Thus, the solution to the recurrence relation is:

$$T(n) = \Theta(\log n)$$

- **Min-Heapify**: Ensures that the value of a node is less than or equal to the values of its children. This operation is used to maintain the min-heap property.
- **Time Complexity**: The process of heapifying a node has a time complexity of $O(\log n)$, given that the height of the tree is logarithmic in relation to the number of nodes.

Heap Insertion

- **Insertion**: Adds a node while maintaining the heap properties; $O(\log n)$ time complexity.

```
Before :       After :
    10                10
   /  \              /  \
  5    3            5    3
                   /
                  4
```

Heap Deletion

- **Deletion (Extract-Min/Max)**: Removes a node while maintaining heap properties; $O(\log n)$ time complexity.
- **Deletion (Extract-Min/Max)**: $O(\log n)$ time complexity.

```
Before:       After :
    10                10
   /  \              /
  5    3            5
```

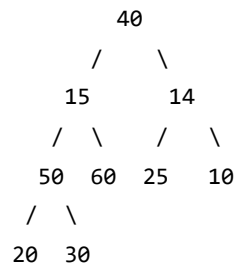
Max Heapify Example and Max-Heapify Code:

```
Max-Heapify(A,i)
1. l = left(i)
2. r = right(i)
3. if l ≤ A.heap - size and A[l] > A[i]
4.    largest = l
5. else largest = i
6. if r ≤ A.heap - size and A[r] > A[largest]
7.    largest = r
8. if largest ‡ i
9.    exchange A[i] with A[largest]
10.   Max-Heapify(A, largest)
```

Step 1: Initial Array

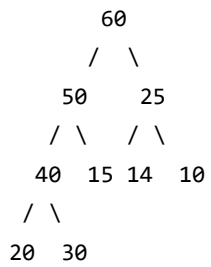
```
Index 索引:   [0] [1] [2] [3] [4] [5] [6] [7] [8]
Value 值:     40  15  14  50  60  25  10  20  30
```

Binary Tree Representation



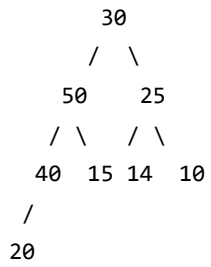
Step 2: Build Max Heap

We start by building a max heap from the initial binary tree.



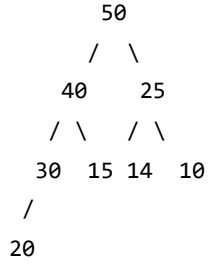
Step 3: Remove 60

After removing 60, we replace it with the last element in the array (30).



Step 4: Max-Heapify

We perform max-heapify to maintain the max heap property.



- In this series of diagrams, we first build a max heap from the initial array. Next, we remove the maximum element (60) and replace it with the last element in the array (30). Finally, we perform max-heapify to maintain

the max heap property, resulting in the final max heap structure.

The `Max-Heapify` algorithm step by step using the given array.

Given:

```
Index:   [0] [1] [2] [3] [4] [5] [6] [7] [8]
Value:   40  15  14  50  60  25  10  20  30
```

The binary tree representation of the above array is:

```
         40
       /    \
     15      14
    / \    /   \
   50  60  25   10
  /  \
 20  30
```

Let's start the `Max-Heapify` process from the root, i.e., `i = 0`.

Step 1: i = 0 (Value: 40)

1. l = left(0) = 1
2. r = right(0) = 2
3. A[1] = 15 and A[0] = 40. Since 15 is not greater than 40, the condition in line 3 is false.
4. So, largest = 0 (from line 5)
5. A[2] = 14, which is also not greater than A[0] = 40. So, the condition in line 6 is false.
6. Since largest is still 0 and is equal to i, we don't swap anything.

However, the root node (40) is not the largest value in the tree. To ensure the max-heap property, we should start the `Max-Heapify` process from the bottom of the tree and work our way up. Let's start with the parent of the last node, which is at index `⌊(8-1)/2⌋ = 3`.

Step 2: i = 3 (Value: 50)

1. l = left(3) = 7
2. r = right(3) = 8
3. A[7] = 20, which is less than A[3] = 50. So, largest = 3.

4. A[8] = 30, which is also less than A[3] = 50. So, largest remains 3.
5. No swap is needed.

Step 3: i = 2 (Value: 14)

1. l = left(2) = 5
2. r = right(2) = 6
3. A[5] = 25, which is greater than A[2] = 14. So, largest = 5.
4. A[6] = 10, which is not greater than A[5] = 25. So, largest remains 5.
5. Swap A[2] and A[5]. The array becomes:

```
Value:     40  15  25  50  60  14  10  20  30
```

6. Since we made a swap, we need to call `Max-Heapify` recursively on the changed index, i.e., i = 5.

Step 4: i = 5 (Value: 14)

1. l = left(5) = 11 (out of bounds)
2. r = right(5) = 12 (out of bounds)
3. No children for i = 5. So, no further action is needed.

Step 5: i = 1 (Value: 15)

1. l = left(1) = 3
2. r = right(1) = 4
3. A[3] = 50, which is greater than A[1] = 15. So, largest = 3.
4. A[4] = 60, which is greater than A[3] = 50. So, largest = 4.
5. Swap A[1] and A[4]. The array becomes:

```
Value:     40  60  25  50  15  14  10  20  30
```

6. Since we made a swap, we need to call `Max-Heapify` recursively on the changed index, i.e., i = 4.

Step 6: i = 4 (Value: 15)

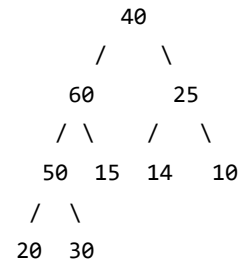
1. l = left(4) = 9 (out of bounds)
2. r = right(4) = 10 (out of bounds)
3. No children for i = 4. So, no further action is needed.

Final Result:

The array after executing `Max-Heapify` from bottom to top is:

```
Value:     40  60  25  50  15  14  10  20  30
```

The binary tree representation is:



Note: The tree is not a max-heap yet. The root node (40) is not the largest value. To convert the entire tree into a max-heap, we would typically use the `Build-Max-Heap` algorithm, which calls `Max-Heapify` for each non-leaf node starting from the bottom.

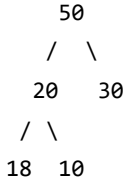
The **BUILD-MAX-HEAP** (or **BUILD-MIN-HEAP**) operation is more efficient than one might initially assume. Its time complexity is $O(n)$, not $O(n \log(n))$. The reason for this efficiency is that when constructing the heap from the bottom up, many nodes are closer to the leaves. These nodes require fewer operations to satisfy the heap property. As we move up the tree, the number of operations required decreases, leading to an aggregate complexity of $O(n)$. This is determined through mathematical deductions that consider the height of the tree and the number of nodes at each level.

Min Heapify Example:

A= {5,8,9,16,12,56,20,18,22,44,13}

Not yet done

Example of a Heap



Analysis

- **Shape Property**

    - Satisfied; the tree is a complete binary tree where all levels, except possibly the last, are completely filled, and all nodes are as far left as possible.
- **Order Property**
    - Satisfied; each node is greater than its children.

Conclusion

- The tree is a heap as it satisfies both the shape and order properties.

Heapsort Algorithm

- A sorting algorithm that works by first building a heap from the values to be sorted, and then repeatedly removing the maximum element from the heap and rebuilding the heap until all values have been removed in sorted order.
- **Heap Sort**: A sorting algorithm with $O(n \log(n))$ time complexity.
- **Priority Queues**: Efficient extraction of minimum or maximum element.

```
Initial:       Sorted:
    3                1
   / \              / \
  1   2            2   3
```

Heapsort Algorithm

- **Step 1**: Build a max-heap from the input array.
- **Step 2**: Repeatedly extract the maximum element from the heap and move it to the sorted part of the array, reducing the size of the heap by one each time.
- **Step 3**: Repeat step 2 until the heap is empty, resulting in a sorted array.
- **Time Complexity**: $O(n \log(n))$, where n is the number of elements to be sorted.

Priority Queues

- **Definition**: A data structure that supports deleting an element with the highest priority (in a max-priority queue) or the lowest priority (in a min-priority queue).
- **Implementation**: Can be efficiently implemented using a heap data structure.

Explanation:

- **Priority Queues**: These are data structures that allow you to manage a set of elements with priorities. The main operations involve inserting elements with a certain priority and extracting elements with the highest or lowest priority.

- **Heap as an Implementation**: Heaps, being a complete binary tree with a specific ordering property, are a natural choice for implementing priority queues as they allow for efficient insertion and extraction operations.
- **Operations**:
  - **INSERT**: This operation adds a new element to the priority queue, maintaining the heap property.
  - **MAXIMUM/MINIMUM**: These operations allow you to peek at the highest or lowest priority element without removing it from the queue.
  - **EXTRACT-MAX/EXTRACT-MIN**: These operations return the highest or lowest priority element and remove it from the queue, maintaining the heap property.
  - **INCREASE-KEY/DECREASE-KEY**: These operations allow you to change the priority of an existing element in the queue, which might involve reorganizing the heap to maintain the heap property.

Feature/Property	Full Binary Tree	Complete Binary Tree	Heap
Definition	Every node has 0 or 2 children. No node has only 1 child.	Every level, except possibly the last, is completely filled, and all nodes are as left as possible.	A Complete Binary Tree with additional ordering properties.
Shape	All levels are fully filled.	All levels, except possibly the last, are fully filled.	Must be a Complete Binary Tree.
Ordering	No specific ordering.	No specific ordering.	Max Heap: Parent ≥ Children; Min Heap: Parent ≤ Children.
Height	For `n` nodes, height is $O(\log n)$.	For `n` nodes, height is $O(\log n)$.	For `n` nodes, height is $O(\log n)$.
Usage	Useful in scenarios where binary outcomes are needed, like Huffman coding.	Useful in scenarios where space optimization is needed, like Heaps.	Priority Queues, Heap Sort.