Heaps

A Introduction: What is a Heap?

A **heap** is a specialized tree-based data structure that satisfies the *heap property*. Heaps are fundamental for implementing efficient priority queues and for algorithms such as heapsort. There are two main types:

- **Max Heap:** Every parent node is greater than or equal to its children.
- **Min Heap:** Every parent node is less than or equal to its children.

Heaps are always complete binary trees, meaning all levels are fully filled except possibly the last, which is filled from left to right.
Comment: The heap property ensures fast access to the largest (max heap) or smallest (min heap) element.

B Why Use Heaps?

- **Priority Queues:** Heaps allow for efficient insertion and removal of the highest (or lowest) priority element.
- **Heapsort:** A comparison-based sorting algorithm with $O(n \log n)$ time complexity.
- **Graph Algorithms:** Used in Dijkstra's and Prim's algorithms for efficient minimum/-maximum selection.

C Heap Properties and Structure

- **Complete Binary Tree:** All levels filled except possibly the last.
- **Heap Property:** Parent-child relationship as defined above.
- **Efficient Operations:** Insert, delete, and access max/min in $O(\log n)$ time.

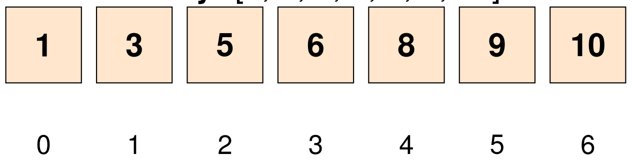
D Array Implementation of Heaps

Heaps are typically implemented as arrays for space and time efficiency. For a node at index i:

- Left child: $2i + 1$
- Right child: $2i + 2$
- Parent: $\lfloor (i - 1)/2 \rfloor$

Comment: This mapping allows for efficient navigation without explicit pointers.

Array: [1, 3, 5, 6, 8, 9, 10]



E Heap Operations

E.1 Insertion

- Add the new element at the end of the array.
- Perform **heapify up** (bubble up) to restore the heap property.

E.2 Extract Min/Max

- Remove the root (min or max).
- Replace it with the last element.
- Perform **heapify down** (bubble down) to restore the heap property.

E.3 Peek

- Access the root element in $O(1)$ time.

F Heapify Algorithms

F.1 Heapify Up (Bubble Up)

Used after insertion to restore the heap property by moving the new element up the tree.

- Insert the new element at the end of the array.
- While the new element is less than its parent (min heap), swap them.
- Repeat until the heap property is restored or the root is reached.

Example: Insert 2 into [1, 3, 5, 6, 8, 9, 10]

- Insert 2 at the end: [1, 3, 5, 6, 8, 9, 10, 2]
- Compare 2 (index 7) with parent 6 (index 3): 2 ¡ 6, swap.
- Array: [1, 3, 5, 2, 8, 9, 10, 6]
- Compare 2 (index 3) with parent 3 (index 1): 2 ¡ 3, swap.
- Array: [1, 2, 5, 3, 8, 9, 10, 6]
- Compare 2 (index 1) with parent 1 (index 0): 2 ¿ 1, done.

Comment: Each swap moves the new element closer to its correct position.

F.2 Heapify Down (Bubble Down)

Used after removing the root to restore the heap property by moving the new root down the tree.

- Replace the root with the last element.
- While the new root is greater than either child (min heap), swap with the smaller child.
- Repeat until the heap property is restored or a leaf is reached.

Example: Remove 1 from [1, 2, 5, 3, 8, 9, 10, 6]

- Replace 1 with 6: [6, 2, 5, 3, 8, 9, 10]
- Compare 6 (index 0) with children 2 (index 1) and 5 (index 2): 2 is smaller, swap.
- Array: [2, 6, 5, 3, 8, 9, 10]
- Compare 6 (index 1) with children 3 (index 3) and 8 (index 4): 3 is smaller, swap.
- Array: [2, 3, 5, 6, 8, 9, 10]
- 6 is now in correct position.

Comment: Each swap moves the out-of-place element closer to its correct position.

Practice Exercise:

Given the array [7, 2, 9, 4, 1, 5], show the array after each insertion as you build a min heap. Draw the tree representation after all insertions. Explain how the heap property is maintained at each step.

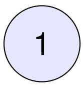
G Visualizing Heaps

G.1 Step-by-Step Example: Building a Min Heap

Below is a step-by-step visualization of building a min heap by inserting the values 1, 3, 5, 6, 8, 9, 10. Each step shows the array and tree representation after an insertion.

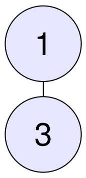
Step 1: Insert 1

Array: [1]



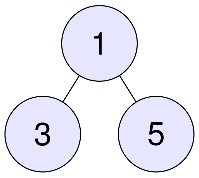
Step 2: Insert 3

Array: [1, 3]



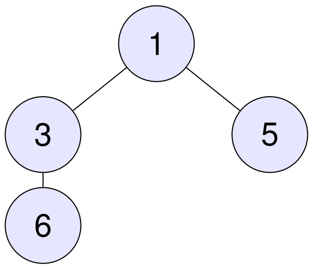
Step 3: Insert 5

Array: [1, 3, 5]



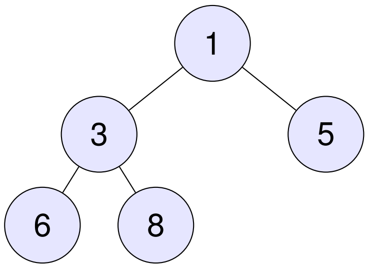
Step 4: Insert 6

Array: [1, 3, 5, 6]



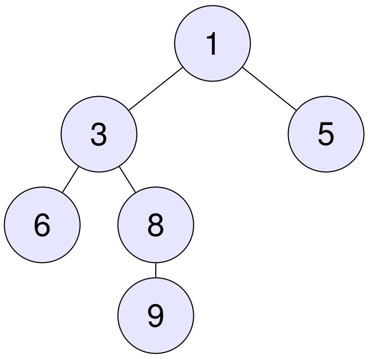
Step 5: Insert 8

Array: [1, 3, 5, 6, 8]

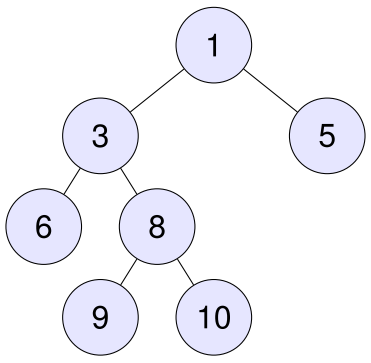


Step 6: Insert 9

Array: [1, 3, 5, 6, 8, 9]



Step 7: Insert 10

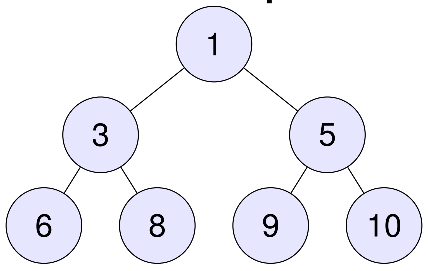


Array: [1, 3, 5, 6, 8, 9, 10]

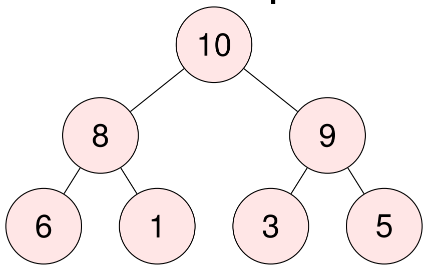
H Min Heap vs Max Heap: Same Data, Different Properties

Given the same data [1, 3, 5, 6, 8, 9, 10], the structure of a min heap and a max heap will differ. Below are both representations:

Min Heap



Max Heap



I Heapsort: Sorting with Heaps

Heapsort is a comparison-based sorting algorithm that uses a heap to sort elements in $O(n \log n)$ time. It is efficient and in-place, making it suitable for large datasets.

I.1 Pseudocode

```
function heapsort(A):
    buildHeap(A)
    for i = n-1 down to 1:
        swap A[0] and A[i]
        heapifyDown(A, 0, i)
```

I.2 Step-by-Step Example

Sort [4, 10, 3, 5, 1] using heapsort (max heap):

1. **Build max heap:**
   - Initial array: [4, 10, 3, 5, 1]
   - After heapify: [10, 5, 3, 4, 1]
2. **Sort:**

- Swap 10 and 1: [1, 5, 3, 4, 10] (10 sorted)
- Heapify: [5, 4, 3, 1, 10]
- Swap 5 and 1: [1, 4, 3, 5, 10] (5 sorted)
- Heapify: [4, 1, 3, 5, 10]
- Swap 4 and 1: [1, 3, 4, 5, 10] (4 sorted)
- Heapify: [3, 1, 4, 5, 10]
- Swap 3 and 1: [1, 3, 4, 5, 10] (3 sorted)
- Heapify: [1, 3, 4, 5, 10]

3. **Result:** [1, 3, 4, 5, 10]

Comment: At each step, the largest element is moved to its correct position at the end.

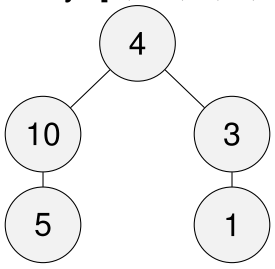
Practice Exercise:

Suppose you have a max heap represented by the array [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]. Show the array after each step of heapsort as the largest elements are removed. Which property of the heap ensures the correctness of the sort?

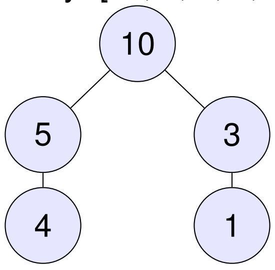
I.3 Visualization of Heapsort

A step-by-step visualization of heapsort for [4, 10, 3, 5, 1] is shown below. Each stage displays the array and tree structure, helping you follow the algorithm visually.

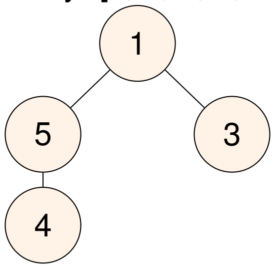
Step 1: Initial Array and Tree
Array: [4, 10, 3, 5, 1]



Step 2: After Building Max Heap
Array: [10, 5, 3, 4, 1]

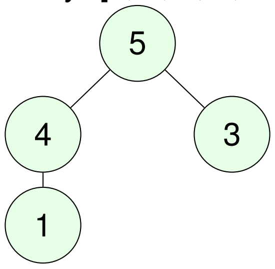


Step 3: After Extracting Max (10)
Array: [1, 5, 3, 4, 10]



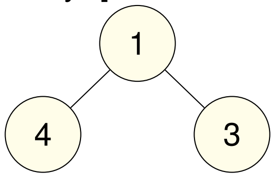
Step 4: Heapify After Extraction

Array: [5, 4, 3, 1, 10]



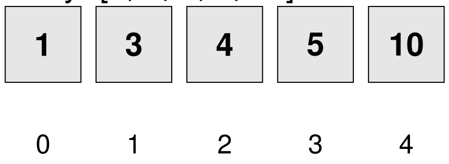
Step 5: Continue Extraction and Heapify

Array: [1, 4, 3, 5, 10] (after extracting 5)



Step 6: Final Sorted Array

Array: [1, 3, 4, 5, 10]



Each step shows the transformation of the heap during sorting. Students are encouraged to follow the swaps and heapify operations visually.