A heap is a specialized tree-based data structure that satisfies the heap property. There are two types of heaps: **max-heaps** and **min-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.

This makes the largest element always at the root of the heap.

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.

This makes the smallest element always at the root of the heap.

Common Operations

1. **Insert**:
    - Insert the new element at the end of the heap.
    - Heapify-up (or bubble-up) to restore the heap property.
2. **Delete (usually delete the root)**:
    - Replace the root with the last element.
    - Remove the last element.
    - Heapify-down (or bubble-down) to restore the heap property.
3. **Extract-Max or Extract-Min**:
    - Return the root of the heap (max for max-heap, min for min-heap).
    - Replace the root with the last element.
    - Remove the last element.
    - Heapify-down to restore the heap property.

Heapify

- **Heapify-up**: Used when inserting a new element.
    - Start from the inserted node and move up the tree, swapping with the parent if the heap property is violated.
- **Heapify-down**: Used when deleting the root.
    - Start from the root and move down the tree, swapping with the larger (max-heap) or smaller (min-heap) child if the heap property is violated.

Applications

- **Priority Queues**: Heaps are often used to implement priority queues because they allow quick access to the highest (or lowest) priority element.
- **Heapsort**: An efficient sorting algorithm that converts an array into a heap and then repeatedly extracts the maximum (or minimum) element.
- **Graph Algorithms**: Many graph algorithms, such as Dijkstra's shortest path algorithm, use heaps to efficiently manage a set of nodes.

Example of Max-Heap Insertion

Consider inserting the value `15` into the following max-heap:

```
      20
     /  \
   10    5
  /  \  /
 6   2 1
```

1. Insert `15` at the end:

```
      20
     /  \
   10    5
  /  \  / \
 6   2 1  15
```

2. Heapify-up: Swap `15` with `5` (its parent):

```
      20
     /  \
   10   15
  /  \  / \
 6   2 1  5
```

Since `15` is not greater than `20`, the heap property is restored.

Understanding heaps is essential for optimizing many algorithms and solving various computational problems efficiently. If you have any specific questions or need further examples, feel free to ask!

Insertion and Deleletion

Certainly! Let's break down the provided heap class code step by step, explaining each part and its concept.

Class Declaration and Initialization

```
class heap {
public:
    int arr[100];
    int size;

    heap() {
        arr[0] = -1;
        size = 0;
    }
};
```

- **Class** `heap`: Defines a class `heap` that will handle the heap operations.
- `arr[100]`: An array to store the heap elements. The heap is implemented using a fixed-size array of 100 elements.
- `size`: Keeps track of the current number of elements in the heap.
- **Constructor** `heap()`: Initializes the heap. The first element `arr[0]` is set to `-1` (not used in the heap operations), and the size is initialized to `0`.

Insert Function

```
void insert(int val) {
    size = size + 1;
    int index = size;
    arr[index] = val;

    while (index > 1) {
        int parent = index / 2;
        if (arr[parent] < arr[index]) {
            swap(arr[parent], arr[index]);
            index = parent;
        } else {
            return;
        }
    }
}
```

- `size = size + 1`: Increases the heap size by 1.
- `int index = size`: Sets `index` to the current size, indicating the position where the new value will be inserted.
- `arr[index] = val`: Inserts the new value at the end of the heap.
- `while (index > 1)`: Ensures the new value is bubbled up to maintain the heap property. It only runs if there is more than one element.
- `int parent = index / 2`: Calculates the index of the parent node.
- `if (arr[parent] < arr[index])`: Checks if the parent node is smaller than the current node. If true, it swaps them.
- `swap(arr[parent], arr[index])`: Swaps the current node with its parent.
- `index = parent`: Updates `index` to the parent's index, continuing to check up the heap.

Print Function

```
void print() {
    for (int i = 1; i <= size; i++) {
        cout << arr[i] << "  ";
    }
    cout << endl;
}
```

- `for (int i = 1; i <= size; i++)`: Iterates through the heap from the first element to the last.
- `cout << arr[i] << " ";`: Prints each element of the heap.
- `cout << endl;`: Prints a new line after printing all elements.

Delete Function

```
void deleteformHeap() {
    if (size == 0) {
        cout << "Nothing to delete" << endl;
        return;
    }

    arr[1] = arr[size];
    size--;

    int i = 1;
    while (i < size) {
        int leftindex = 2 * i;
        int rightindex = 2 * i + 1;

        if (leftindex < size && arr[i] < arr[leftindex]) {
            swap(arr[i], arr[leftindex]);
            i = leftindex;
        } else if (rightindex < size && arr[i] < arr[rightindex]) {
            swap(arr[i], arr[rightindex]);
            i = rightindex;
        } else {
            return;
        }
    }
}
```

- `if (size == 0)`: Checks if the heap is empty. If true, prints a message and returns.
- `arr[1] = arr[size]`: Moves the last element to the root of the heap.
- `size--`: Decreases the heap size by 1.
- `int i = 1`: Starts the heapify-down process from the root.
- `while (i < size)`: Ensures the loop runs while the index is within the heap.
- `int leftindex = 2 * i`: Calculates the index of the left child.
- `int rightindex = 2 * i + 1`: Calculates the index of the right child.

- `if (leftindex < size && arr[i] < arr[leftindex])`: Checks if the left child exists and is greater than the current node. If true, swaps them.
- `swap(arr[i], arr[leftindex])`: Swaps the current node with the left child.
- `i = leftindex`: Updates `i` to the left child's index.
- `else if (rightindex < size && arr[i] < arr[rightindex])`: Checks if the right child exists and is greater than the current node. If true, swaps them.
- `swap(arr[i], arr[rightindex])`: Swaps the current node with the right child.
- `i = rightindex`: Updates `i` to the right child's index.
- `else`: If neither child is greater than the current node, the heap property is restored, and the function returns.

This class provides basic functionality for inserting elements into a max-heap, printing the heap, and deleting the root element from the heap.

Given and some Input

Certainly! Let's walk through the code step by step, explaining how each insertion and deletion operation works in the heap.

Initial Setup

```
heap h;
```

- This creates an instance `h` of the heap class.

Insertion Operations

Inserting 50

```
h.insert(50);
```

1. **Size Increment**: `size` is increased from `0` to `1`.
2. **Insert at End**: `arr[1] = 50`.
3. **Heapify-Up**: Since `index` is `1` (root), no need to heapify-up.

Heap after insertion:

```
[_, 50]
```

Inserting 54

```
h.insert(54);
```

1. **Size Increment**: `size` is increased from `1` to `2`.
2. **Insert at End**: `arr[2] = 54`.
3. **Heapify-Up**:
    - Parent index of `2` is `1`.
    - `arr[1]` (50) is less than `arr[2]` (54).
    - Swap `arr[1]` and `arr[2]`.
    - Updated `index` is `1` (root), no further heapify-up needed.

Heap after insertion:

```
[_, 54, 50]
```

Inserting 79

```
h.insert(79);
```

1. **Size Increment**: `size` is increased from `2` to `3`.
2. **Insert at End**: `arr[3] = 79`.
3. **Heapify-Up**:
    - Parent index of `3` is `1`.
    - `arr[1]` (54) is less than `arr[3]` (79).
    - Swap `arr[1]` and `arr[3]`.
    - Updated `index` is `1` (root), no further heapify-up needed.

Heap after insertion:

```
[_, 79, 50, 54]
```

Inserting 58

```
h.insert(58);
```

1. **Size Increment**: `size` is increased from `3` to `4`.
2. **Insert at End**: `arr[4] = 58`.
3. **Heapify-Up**:
    - Parent index of `4` is `2`.
    - `arr[2]` (50) is less than `arr[4]` (58).
    - Swap `arr[2]` and `arr[4]`.
    - Updated `index` is `2`.
    - Parent index of `2` is `1`.
    - `arr[1]` (79) is greater than `arr[2]` (58), no further heapify-up needed.

Heap after insertion:

```
[_, 79, 58, 54, 50]
```

Inserting 89

```
h.insert(89);
```

1. **Size Increment**: `size` is increased from `4` to `5`.
2. **Insert at End**: `arr[5] = 89`.
3. **Heapify-Up**:
    - Parent index of `5` is `2`.
    - `arr[2]` (58) is less than `arr[5]` (89).
    - Swap `arr[2]` and `arr[5]`.
    - Updated `index` is `2`.
    - Parent index of `2` is `1`.
    - `arr[1]` (79) is less than `arr[2]` (89).
    - Swap `arr[1]` and `arr[2]`.
    - Updated `index` is `1` (root), no further heapify-up needed.

Heap after insertion:

```
[_, 89, 79, 54, 50, 58]
```

Print Operation

```
h.print();
```

- This prints the elements of the heap:

```
89  79  54  50  58
```

Deletion Operation

Deleting the Root (89)

```
h.deleteformheap();
```

1. **Check Size**: The heap

size is `5`, so proceed with deletion. 2. **Replace Root with Last Element**: `arr[1] = arr[5]` (last element 58 replaces root 89). 3. **Decrement Size**: `size` is decreased from `5` to `4`.

Heap before heapify-down:

```
[_, 58, 79, 54, 50]
```

4. **Heapify-Down**:
    - Starting at root `i = 1`.
    - Calculate left and right children: `leftindex = 2 * i = 2`, `rightindex = 2 * i + 1 = 3`.
    - Compare with left child: `arr[1]` (58) and `arr[2]` (79). Since `arr[2]` is greater, swap them.
    - Swap `arr[1]` (58) and `arr[2]` (79).
    - Update `i = 2`.

Heap after first swap:

```
[_, 79, 58, 54, 50]
```

- Calculate new left and right children: `leftindex = 2 * 2 = 4`, `rightindex = 2 * 2 + 1 = 5`.
- Compare with left child: `arr[2]` (58) and `arr[4]` (50). No swap needed as `arr[2]` is greater than `arr[4]`.
- Right child doesn't exist (`rightindex > size`), so heapify-down stops here.

Heap after deletion and heapify-down:

```
[_, 79, 58, 54, 50]
```

Final Heap State

The final state of the heap after the given operations is:

```
[_, 79, 58, 54, 50]
```

This corresponds to the array:

```
79  58  54  50
```

Here's a detailed step-by-step flow of the insertion and deletion operations:

1. **Insert 50**: Adds 50 to the heap at index 1.

2. **Insert 54**: Adds 54 to the heap, compares with 50, and swaps them.
3. **Insert 79**: Adds 79, compares with 54, swaps them, then compares with 50, and swaps again.
4. **Insert 58**: Adds 58, compares with 50, swaps them, but does not swap further as 79 is greater.
5. **Insert 89**: Adds 89, swaps with 58, then swaps with 79.
6. **Print Heap**: Displays the current heap array: `89 79 54 50 58`.
7. **Delete Root**: Replaces 89 with 58, reduces the size, heapifies down by swapping 58 with 79, resulting in the final heap state: `79 58 54 50`.

Feel free to ask if you have more questions or need further clarification!

Heapify Function

Let's break down the provided code step by step to understand how it works.

Function `heapify`

```
void heapify(int arr[], int n, int i) {
    int largest = i;
    int left = 2 * i;
    int right = 2 * i + 1;

    if (left < n && arr[largest] < arr[left]) {
        largest = left;
    }

    if (right < n && arr[largest] < arr[right]) {
        largest = right;
    }

    if (largest != i) {
        swap(arr[largest], arr[i]);
        heapify(arr, n, largest);
    }
}
```

- **Parameters**:
  - `arr[]`: The array representing the heap.
  - `n`: The number of elements in the heap.
  - `i`: The index of the current element being heapified.
- **Variables**:
  - `largest`: Keeps track of the largest element among the current node and its children.
  - `left`: The index of the left child (`2 * i`).
  - `right`: The index of the right child (`2 * i + 1`).
- **Process**:
  - **Determine the largest**:

        - If the left child exists (`left < n`) and is greater than the current node (`arr[largest] < arr[left]`), update `largest` to be the left child.
        - If the right child exists (`right < n`) and is greater than the current largest node (`arr[largest] < arr[right]`), update `largest` to be the right child.
    - **Swap and recurse**:
        - If the largest element is not the current element (`largest != i`), swap them and recursively call `heapify` on the affected subtree (`heapify(arr, n, largest)`).

Main Function

```
int main() {
    int arr[] = {-1, 54, 53, 55, 52, 50};
    int n = 5;

    for (int i = n / 2; i > 0; i--) {
        heapify(arr, n, i);
    }

    cout << "Printing the array now" << endl;
    for (int i = 1; i < n; i++) {
        cout << arr[i] << "  ";
    }
    cout << endl;

    return 0;
}
```

- **Array Initialization**:
    - `int arr[] = {-1, 54, 53, 55, 52, 50}`: The array to be heapified. Note that `-1` is a placeholder and not part of the heap.
    - `int n = 5`: The number of elements in the heap (excluding the placeholder).
- **Building the Heap**:
    - The loop `for (int i = n / 2; i > 0; i--)` starts from the last non-leaf node and moves upwards to the root.
    - For each node, it calls `heapify(arr, n, i)` to ensure the subtree rooted at that node satisfies the heap property.
- **Printing the Array**:
    - After heapifying the array, it prints the elements from index 1 to `n-1`.

Step-by-Step Execution

1. **Initialization**:
    - `arr[] = {-1, 54, 53, 55, 52, 50}`
    - `n = 5`

2. **Heap Construction**:
    - The loop `for (int i = n / 2; i > 0; i--)` starts with `i = 2` and then `i = 1`.

   Heapify at `i = 2`:
    - `largest = 2`
    - `left = 4`, `right = 5`
    - Compare `arr[2] = 53` with `arr[4] = 52` (no change)
    - Compare `arr[2] = 53` with `arr[5] = 50` (no change)
    - No swap needed as `largest == i`.

   Heapify at `i = 1`:
    - `largest = 1`
    - `left = 2`, `right = 3`
    - Compare `arr[1] = 54` with `arr[2] = 53` (no change)
    - Compare `arr[1] = 54` with `arr[3] = 55`
    - Update `largest = 3`
    - Swap `arr[1] = 54` and `arr[3] = 55`
    - `arr[] = {-1, 55, 53, 54, 52, 50}`
    - Call `heapify(arr, n, 3)`:
        - `largest = 3`
        - `left = 6`, `right = 7` (both out of bounds)
        - No swap needed.

3. **Final Heap**:
    - The array now represents a max-heap: `arr[] = {-1, 55, 53, 54, 52, 50}`

4. **Printing**:
    - The loop prints the elements from index 1 to 4 (since `n = 5`): `55 53 54 52`.

Output

```
Printing the array now
55  53  54  52
```

This code successfully builds a max-heap from the provided array and prints the heap elements.

Step-by-Step Execution of Recursive call

Sure, let's go through a detailed example to understand how the recursion works in the `heapify` function step by step.

Example Array and Initial Call

Consider the array: `arr[] = {-1, 3, 5, 8, 1, 2}`. We'll call `heapify(arr, 6, 1)`.

```
    if (largest != i) {
         swap(arr[largest], arr[i]);
         heapify(arr, n, largest);
     }
```

Initial Call: `heapify(arr, 6, 1)`

1. **Variables Initialization**:
   - `i = 1`
   - `largest = 1`
   - `left = 2 * 1 = 2`
   - `right = 2 * 1 + 1 = 3`
2. **Comparisons**:
   - `arr[1] = 3`, `arr[2] = 5`, `arr[3] = 8`
   - Compare `arr[1]` (3) with `arr[2]` (5): since `5 > 3`, update `largest = 2`.
   - Compare `arr[largest]` (5) with `arr[3]` (8): since `8 > 5`, update `largest = 3`.
3. **Check and Swap**:
   - `largest != i`: `3 != 1`, so swap `arr[1]` and `arr[3]`.
   - Array after swap: `[-1, 8, 5, 3, 1, 2]`.
4. **Recursive Call**:
   - Call `heapify(arr, 6, 3)` to ensure the subtree rooted at index 3 maintains the heap property.

Recursive Call: `heapify(arr, 6, 3)`

1. **Variables Initialization**:
   - `i = 3`
   - `largest = 3`
   - `left = 2 * 3 = 6`
   - `right = 2 * 3 + 1 = 7`
2. **Comparisons**:
   - `arr[3] = 3`, `arr[6] = 2` (no right child, as `right >= n`)
   - Compare `arr[3]` (3) with `arr[6]` (2): no change since `3 > 2`.
3. **Check and Swap**:
   - `largest == i`: `3 == 3`, so no swap is needed.
   - No further recursion as the subtree rooted at 3 is already a valid heap.

Recursion Termination

- The recursive call `heapify(arr, 6, 3)` terminates without any changes.
- The initial call `heapify(arr, 6, 1)` also terminates, as the necessary swaps have been made to restore the heap property.

Final State of the Array

After completing the `heapify` process, the array is: `[-1, 8, 5, 3, 1, 2]`.

Summary

1. **Initial Call**:
    - Compare root with children.
    - Swap with the largest child.
    - Recursively call `heapify` on the affected subtree.
2. **Recursive Call**:
    - Repeat comparisons and possible swaps.
    - Recursion continues until the subtree rooted at the affected node is a valid heap.
3. **Termination**:
    - Recursion terminates when no further swaps are needed, ensuring the entire subtree maintains the heap property.

This recursive process ensures that any violations of the heap property are fixed, starting from the node `i` down to the leaves, resulting in a valid max-heap (or min-heap, depending on the comparison criteria).

MIN Heap

To modify the given max-heap code to create a min-heap, you need to change the comparison conditions in both the `heapify` function and the insertion logic. A min-heap is a binary tree where the parent node is always less than or equal to its children.

Here's how you can modify the max-heap code to implement a min-heap:

Min-Heap `heapify` Function

```
void heapify(int arr[], int n, int i) {
    int smallest = i;
    int left = 2 * i;
    int right = 2 * i + 1;

    if (left < n && arr[smallest] > arr[left]) {
        smallest = left;
    }

    if (right < n && arr[smallest] > arr[right]) {
```

```
        smallest = right;
    }

    if (smallest != i) {
        swap(arr[smallest], arr[i]);
        heapify(arr, n, smallest);
    }
}
```

Main Function for Building Min-Heap

```
int main() {
    int arr[] = {-1, 54, 53, 55, 52, 50};
    int n = 6; // Including the placeholder -1

    for (int i = (n - 1) / 2; i > 0; i--) {
        heapify(arr, n, i);
    }

    cout << "Printing the array now" << endl;
    for (int i = 1; i < n; i++) {
        cout << arr[i] << "  ";
    }
    cout << endl;

    return 0;
}
```

Explanation of Changes

1. **Comparison Conditions**:
    - In the `heapify` function, change the comparison from `arr[largest] < arr[left]` to `arr[smallest] > arr[left]` and similarly for the right child.
    - This ensures that the smallest element bubbles up to the root of the subtree.
2. **Loop for Heap Construction**:
    - The loop `for (int i = (n - 1) / 2; i > 0; i--)` ensures that we start heapifying from the last non-leaf node up to the root.

Complete Min-Heap Code

Here's the complete code for building and displaying a min-heap:

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

void heapify(int arr[], int n, int i) {
```

```
    int smallest = i;
    int left = 2 * i;
    int right = 2 * i + 1;

    if (left < n && arr[smallest] > arr[left]) {
        smallest = left;
    }

    if (right < n && arr[smallest] > arr[right]) {
        smallest = right;
    }

    if (smallest != i) {
        swap(arr[smallest], arr[i]);
        heapify(arr, n, smallest);
    }
}

int main() {
    int arr[] = {-1, 54, 53, 55, 52, 50};
    int n = 6; // Including the placeholder -1

    for (int i = (n - 1) / 2; i > 0; i--) {
        heapify(arr, n, i);
    }

    cout << "Printing the array now" << endl;


```cpp
    cout << "Printing the array now" << endl;
    for (int i = 1; i < n; i++) {
        cout << arr[i] << "  ";
    }
    cout << endl;

    return 0;
}
```

This code will build a min-heap from the array and then print the elements of the heap.

Testing the Min-Heap Code

To verify the code, let's consider the initial array: `arr[] = {-1, 54, 53, 55, 52, 50}`.

1. **Initial Array**:

```
Index:   0   1   2   3   4   5
Values: -1  54  53  55  52  50
```
```

2. **Heapify Process**:

   - Start from the last non-leaf node `(n-1)/2 = 2`:

   Heapify at i = 2:

   - `i = 2`
   - `left = 4`, `right = 5`
   - Compare `arr[2] = 53`, `arr[4] = 52`, `arr[5] = 50`
   - Smallest is `arr[5] = 50`
   - Swap `arr[2]` and `arr[5]`: `[-1, 54, 50, 55, 52, 53]`
   - Recursive call `heapify(arr, 6, 5)` does not change anything (no children for node 5).

   Heapify at i = 1:

   - `i = 1`
   - `left = 2`, `right = 3`
   - Compare `arr[1] = 54`, `arr[2] = 50`, `arr[3] = 55`
   - Smallest is `arr[2] = 50`
   - Swap `arr[1]` and `arr[2]`: `[-1, 50, 54, 55, 52, 53]`
   - Recursive call `heapify(arr, 6, 2)`:
     - `i = 2`
     - `left = 4`, `right = 5`
     - Compare `arr[2] = 54`, `arr[4] = 52`, `arr[5] = 53`
     - Smallest is `arr[4] = 52`
     - Swap `arr[2]` and `arr[4]`: `[-1, 50, 52, 55, 54, 53]`
     - Recursive call `heapify(arr, 6, 4)` does not change anything (no children for node 4).

3. **Final Min-Heap**:

```
Index:   0   1   2   3   4   5
Values: -1  50  52  55  54  53
```

4. **Printed Output**:

```
Printing the array now
50  52  55  54  53
```

Summary

- **Heapify Function**: Adjusted to compare for smallest values instead of largest.
- **Construction Loop**: Starts from the last non-leaf node up to the root.
- **Final Output**: Prints the elements of the min-heap.

This ensures the array maintains the min-heap property, where each parent node is less than or equal to its children.

Let's break down the heap sort algorithm and the provided code step by step.

Heap Sort Algorithm Overview

Heap sort involves two main steps:

1. **Building a max-heap**: Transform the input array into a max-heap.
2. **Sorting the array**: Repeatedly remove the maximum element from the heap and move it to the end of the array. Adjust the remaining heap to maintain the heap property.

Provided Code

Here's the provided code with the heap sort function:

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

void heapify(int arr[], int n, int i) {
    int largest = i;
    int left = 2 * i;
    int right = 2 * i + 1;

    if (left < n && arr[largest] < arr[left]) {
        largest = left;
    }

    if (right < n && arr[largest] < arr[right]) {
        largest = right;
    }

    if (largest != i) {
        swap(arr[largest], arr[i]);
        heapify(arr, n, largest);
    }
}

void heapsort(int arr[], int n) {
    int size = n;

    while (size > 1) {
        swap(arr[size], arr[1]);
        size--;
        heapify(arr, size, 1);
    }
}

int main() {
    int arr[] = {-1, 54, 53, 55, 52, 50};
    int n = 6; // Include the placeholder -1

    for (int i = (n - 1) / 2; i > 0; i--) {
        heapify(arr, n, i);
```

```
    }

    heapsort(arr, n);

    cout << "Printing the sorted heap" << endl;
    for (int i = 1; i < n; i++) {
        cout << arr[i] << "  ";
    }
    cout << endl;

    return 0;
}
```

Step-by-Step Explanation

1. Building the Max-Heap

```
for (int i = (n - 1) / 2; i > 0; i--) {
    heapify(arr, n, i);
}
```

- The loop starts from the last non-leaf node `(n - 1) / 2` and goes up to the root.
- `heapify` is called for each node, ensuring that the subtree rooted at each node satisfies the max-heap property.

2. Heap Sort Function

```
void heapsort(int arr[], int n) {
    int size = n;

    while (size > 1) {
        swap(arr[size], arr[1]);
        size--;
        heapify(arr, size, 1);
    }
}
```

- **Initial Setup**:
  - `int size = n;` initializes the size to the length of the array.
- **Loop**:
  - The loop continues until `size > 1`.
  - **Swap**: The maximum element (`arr[1]`) is swapped with the last element in the current heap (`arr[size]`).

- **Decrement Size**: `size--` reduces the heap size by one, effectively removing the last element (which is now the largest element in the sorted part of the array).
- **Heapify**: `heapify(arr, size, 1)` restores the heap property for the reduced heap.

3. Printing the Sorted Array

```
cout << "Printing the sorted heap" << endl;
for (int i = 1; i < n; i++) {
    cout << arr[i] << "  ";
}
cout << endl;
```

- This loop prints the sorted elements of the array, starting from index 1 (since index 0 is a placeholder).

Execution Example

Let's go through an example with the provided array `arr[] = {-1, 54, 53, 55, 52, 50}`.

1. **Initial Array**:

```
Index:   0   1   2   3   4   5
Values: -1  54  53  55  52  50
```

2. **Building the Max-Heap**:
   - Call `heapify(arr, 6, 2)`:
     - Compare `arr[2] = 53` with `arr[4] = 52` and `arr[5] = 50` -> no swap needed.
   - Call `heapify(arr, 6, 1)`:
     - Compare `arr[1] = 54` with `arr[2] = 53` and `arr[3] = 55`.
     - Swap `arr[1] = 54` and `arr[3] = 55`.
     - Call `heapify(arr, 6, 3)`:
       - No further swaps needed.

3. **Max-Heap Formed**:

```
Index:   0   1   2   3   4   5
Values: -1  55  53  54  52  50
```

4. **Heap Sort**:
   - **First Iteration**:
     - Swap `arr[6]` (50) with `arr[1]` (55).
     - Decrement `size` to 5.
     - Call `heapify(arr, 5, 1)`:
       - Swap `arr[1]` (50) and `arr[3]` (54).

        - Call `heapify(arr, 5, 3)` (no further swaps needed).
    - **Second Iteration**:
        - Swap `arr[5]` (52) with `arr[1]` (54).
        - Decrement `size` to 4.
        - Call `heapify(arr, 4, 1)`:
            - Swap `arr[1]` (52) and `arr[2]` (53).
            - Call `heapify(arr, 4, 2)` (no further swaps needed).
    - **Third Iteration**:
        - Swap `arr[4]` (52) with `arr[1]` (53).
        - Decrement `size` to 3.
        - Call `heapify(arr, 3, 1)` (no swaps needed).
    - **Fourth Iteration**:
        - Swap `arr[3]` (53) with `arr[1]` (52).
        - Decrement `size` to 2.
        - Call `heapify(arr, 2, 1)` (no swaps needed).
    - **Fifth Iteration**:
        - Swap `arr[2]` (53) with `arr[1]` (54).
        - Decrement `size` to 1.

5. **Sorted Array**:

```
Index:   0   1   2   3   4   5
Values: -1  50  52  53  54  55
```

6. **Output**:

```
Printing the sorted heap
50  52  53  54  55
```

Summary

- **Heap Construction**: Transform the array into a max-heap.
- **Sorting**: Repeatedly swap the maximum element with the last element, reduce the heap size, and call `heapify` to maintain the heap property.
- **Final Output**: Print the sorted array.

Let's go through the provided code that uses the C++ Standard Library's `priority_queue` to implement both a max-heap and a min-heap. I'll explain each part of the code step by step.

Code Explanation

Including the Required Header

```
#include <iostream>
#include <queue>
```

```
using namespace std;
```

- `#include <iostream>` is used for input and output operations.
- `#include <queue>` is used to include the `priority_queue` container.

Max-Heap Using `priority_queue`

```
cout << "Priority Queue from Here " << endl;

// Max Heap
priority_queue<int> pq;
pq.push(56);
pq.push(78);
pq.push(54);
pq.push(1);

cout << "The top of the priority queue is: " << pq.top() << endl;
pq.pop();
cout << "The top of the priority queue is: " << pq.top() << endl;
cout << "The size of the priority Queue is: " << pq.size() << endl;
```

1. **Declaration**:
    - `priority_queue<int> pq;` creates a max-heap (default behavior of `priority_queue`).
2. **Insertion**:
    - `pq.push(56);` adds 56 to the heap.
    - `pq.push(78);` adds 78 to the heap.
    - `pq.push(54);` adds 54 to the heap.
    - `pq.push(1);` adds 1 to the heap.
3. **Accessing the Top Element**:
    - `pq.top()` returns the largest element in the heap.
    - After pushing the elements, the largest element is 78.
4. **Removing the Top Element**:
    - `pq.pop()` removes the largest element (78).
    - The new top element becomes 56 after 78 is removed.
5. **Size of the Heap**:
    - `pq.size()` returns the number of elements in the heap.

Min-Heap Using `priority_queue`

```
// Min Heap
priority_queue<int, vector<int>, greater<int>> minheap;
minheap.push(5);
minheap.push(7);
minheap.push(4);
minheap.push(1);

cout << "The top of the priority queue is: " << minheap.top() << endl;
minheap.pop();
cout << "The top of the priority queue is: " << minheap.top() << endl;
cout << "The size of the priority Queue is: " << minheap.size() << endl;
```

1. **Declaration**:
    - `priority_queue<int, vector<int>, greater<int>> minheap;` creates a min-heap.
    - The third template parameter `greater<int>` changes the default behavior to create a min-heap.
2. **Insertion**:
    - `minheap.push(5);` adds 5 to the heap.
    - `minheap.push(7);` adds 7 to the heap.
    - `minheap.push(4);` adds 4 to the heap.
    - `minheap.push(1);` adds 1 to the heap.
3. **Accessing the Top Element**:
    - `minheap.top()` returns the smallest element in the heap.
    - After pushing the elements, the smallest element is 1.
4. **Removing the Top Element**:
    - `minheap.pop()` removes the smallest element (1).
    - The new top element becomes 4 after 1 is removed.
5. **Size of the Heap**:
    - `minheap.size()` returns the number of elements in the heap.

Full Code

Here's the full code combined:

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

int main() {
    cout << "Priority Queue from Here " << endl;

    // Max Heap
    priority_queue<int> pq;
```

```
    pq.push(56);
    pq.push(78);
    pq.push(54);
    pq.push(1);

    cout << "The top of the priority queue is: " << pq.top() << endl;
    pq.pop();
    cout << "The top of the priority queue is: " << pq.top() << endl;
    cout << "The size of the priority Queue is: " << pq.size() << endl;

    // Min Heap
    priority_queue<int, vector<int>, greater<int>> minheap;
    minheap.push(5);
    minheap.push(7);
    minheap.push(4);
    minheap.push(1);

    cout << "The top of the priority queue is: " << minheap.top() << endl;
    minheap.pop();
    cout << "The top of the priority queue is: " << minheap.top() << endl;
    cout << "The size of the priority Queue is: " << minheap.size() << endl;

    return 0;
}
```

Summary

- **Max-Heap**:
  - Implemented using `priority_queue<int>`.
  - The largest element is always at the top.
  - Operations: push, top, pop, size.
- **Min-Heap**:
  - Implemented using `priority_queue<int, vector<int>, greater<int>>`.
  - The smallest element is always at the top.
  - Operations: push, top, pop, size.

This code demonstrates how to use the `priority_queue` container in C++ to create both max-heaps and min-heaps, and perform common heap operations like insertion, top element access, deletion, and size checking.