Implementing max heap

📚 Max Heap - Manual Implementation in C++

👨‍💻 Code Summary:

We are implementing a **Max Heap** using a static array `arr[50]` and managing the heap with custom methods:

- `push(x)` → Inserts an element
- `pop()` → Removes the root (maximum element)
- `top()` → Returns the root
- `display()` → Displays the current heap
- Heap index starts from `1` for simpler parent-child math.

📦 Class: `max_heap`

```
class max_heap {
  public:
    int arr[50];  // Static array to hold heap elements
    int idx;      // Index tracker, starts from 1
```

🔷 Constructor

```
max_heap() {
    idx = 1;  // Start heap from index 1 (index 0 is unused)
}
```

> Why start from 1?
>
> It simplifies the parent-child relationship:

- `parent(i) = i / 2`
- `left(i) = 2 * i`
- `right(i) = 2 * i + 1`

🔼 Method: `push(int x)`

```
void push(int x){
  arr[idx] = x;
  int i = idx;
  idx++;

  // Heapify up
  while(i != 1){
    int parent = i / 2;
    if(arr[i] > arr[parent]){
      swap(arr[i], arr[parent]);
      i = parent;
    }
    else break;
  }
}
```

✅ Purpose:

- Adds a new element `x` to the heap.
- Restores the max-heap property by **heapifying up** (swapping with parent if larger).

🔽 Method: `pop()`

```
void pop() {
  idx--;
  arr[1] = arr[idx];  // Replace root with last element
```

```
  int i = 1;

  // Heapify down
  while(true) {
    int left = 2 * i;
    int right = 2 * i + 1;

    if (left > idx - 1) break; // No children

    // Only left child exists
    if (right > idx - 1) {
      if (arr[i] < arr[left]) {
        swap(arr[i], arr[left]);
        i = left;
        continue;
      } else break;
    }

    // Both children exist
    if (arr[left] > arr[right]) {
      if (arr[i] < arr[left]) {
        swap(arr[i], arr[left]);
        i = left;
      } else break;
    } else {
      if (arr[i] < arr[right]) {
        swap(arr[i], arr[right]);
        i = right;
      } else break;
    }
  }
}
```

✅ Purpose:

- Removes the root (maximum value) from the heap.

- Restores the heap property by **heapifying down**.

👁️ Method: `display()`

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

> Shows all the elements of the heap in current order (not necessarily sorted).

🔝 Method: `top()`

```
int top() {
  return arr[1]; // Root of the max heap
}
```

> Returns the max element in O(1) time.

🧪 Main Function

```
int main() {
  max_heap h;

  h.push(50);
  h.push(30);
  h.push(40);
  h.push(10);
```

```
  h.push(60);
  h.push(5);
  h.push(35);

  cout << "Heap after inserting elements:\n";
  h.display();

  cout << "\nPopping all elements from heap:\n";
  while(h.idx > 1){
    cout << h.arr[1] << " ";
    h.pop();
  }

  return 0;
}
```

💡 Output:

```
Heap after inserting elements:
60 50 40 10 30 5 35

Popping all elements from heap:
60 50 40 35 30 10 5
```

🧠 Heap Properties Used:

Relationship	Formula
Parent of `i`	`i / 2`
Left child of `i`	`2 * i`
Right child of `i`	`2 * i + 1`

🧩 Time Complexities

Operation	Time
`push`	O(log N)
`pop`	O(log N)
`top`	O(1)

📌 Notes:

- This is a **manual implementation** using array, no STL used.
- Could be enhanced with dynamic array (e.g., `vector`) or size-checking.
- Works great for **educational purpose** or to understand how heaps work internally.

🔁 Code Recap:

```
void push(int x){
  arr[idx] = x;
  idx++;

  int i = idx - 1;
  while(i != 1){
    int parent = i / 2;
    if(arr[i] > arr[parent]){
      swap(arr[i], arr[parent]);
      i = parent;
    } else break;
  }
}
```

💥 Step-by-Step Explanation:

🧱 Heap Initialization:

- Heap is stored in array `arr[ ]` of size 50.

- Indexing starts from **1** (not 0) for easy calculation of parent and child.
- `idx` stores the next **empty** index where a new element will be inserted.

✅ Step 1: Insert the new value at the current index

```
arr[idx] = x;
```

- Suppose `idx = 4` and `x = 50`
- Then it places 50 at `arr[4]`

✅ Step 2: Move index forward for the next insertion

```
idx++;
```

- We increment `idx` to point to the next empty slot.
- This doesn't affect the current insert operation.

✅ Step 3: Rearrangement (Heapify Up)

```
int i = idx - 1;
```

- We now want to **check and fix the heap property**
- `i` points to the **newly inserted element** (i.e., `idx - 1`)

🔁 Loop: While `i != 1` (not root), compare with parent

```
int parent = i / 2;
```

- Parent of node at `i` is always `i / 2` in a binary heap.

⚖️ Check condition:

```
if(arr[i] > arr[parent]){
  swap(arr[i], arr[parent]);
  i = parent; // Go up
}
else break;
```

- If **inserted value is greater than parent**, max-heap property is violated.
- So, we **swap** and continue going **upward** in the tree.

🎯 What this achieves:

This loop ensures that:

- Every parent is **greater than** its children.
- The **max-heap property** is always maintained.
- Time complexity: **O(log n)** because height of heap is log n.

📊 Example Walkthrough:

Initial Heap:

```
Index: 1  2  3
Array: 60 30 40
idx = 4
```

Now, push(50):

- arr[4] = 50
- idx++
- i = 3
- Parent of i=4 is 2 → Compare 50 vs 30 → 50 > 30 → Swap
- Now i = 2, parent = 1 → Compare 50 vs 60 → 50 < 60 → Stop

Final Heap:

```
Index: 1  2  3  4
Array: 60 50 40 30
```

🧠 Final Notes:

- `push` function = insert + fix
- Indexing from `1` makes parent-child relations clean.
- Looping upward ensures **new value bubbles up to right spot.**

🧠 Mera `pop()` function ka kaam kya hai?

> Max Heap me jo sabse bada element hota hai (i.e. arr[1]), use remove karta hai, aur heap ko phir se theek karta hai (heapify down ya percolate down se).

🔍 Step-by-step Explanation

```
void pop(){
   idx = idx-1;                  // Step 1
   arr[1] = arr[idx];             // Step 2
   int i = 1;                   // Step 3
```

Step 1: `idx--`

- Last element remove karne ka signal.
- Heap size kam kar diya.

Step 2: `arr[1] = arr[idx]`

- Last element ko root pe daal diya.
- Ab root pe galat value hai (heap property tooti hai).

Step 3: `int i = 1`

- Start heapify from root (index 1).

```
while(true){
  int left = 2*i;
  int right = 2*i + 1;
```

- Left and right child index calculate kiya.

```
  if (left > idx-1){
    break;
  }
```

Condition 1: Leaf Node

- Agar left child hi nahi hai, to ye leaf node hai.
- Break kar do, kuch karna hi nahi.

```
  if(right > idx-1){
    if(arr[i]<arr[left]){
      swap(arr[i],arr[left]);
      i = left;
      continue;
    }
    else break;
  }
```

Condition 2: Sirf left child hai

- Compare `arr[i]` aur `arr[left]`
- Agar child bada hai, to swap karo aur `i = left`
- `continue` lagaya hai, taaki loop dubara chale updated position ke liye

```
if(arr[left]>arr[right]){
  if(arr[i]<arr[left]){
    swap(arr[i],arr[left]);
    i = left;
  }
  else{
    break;
  }
}
```

Condition 3: Dono child hai, left bada hai

- Left child sabse bada hai → agar root chhota hai, swap karo
- `i = left` kar diya heapify continue karne ke liye

```
else{
  if(arr[i]<arr[right]){
    swap(arr[i],arr[right]);
    i = right;
  }
  else{
    break;
  }
}
```

Condition 4: Right child bada hai

- Same logic as above, but right child ke liye

✅ Example: Let's Visualize

Heap Before `pop()`

```
    60
   /  \
  50   40
 / \   /
30 20 10
```

Array:

`arr = [x, 60, 50, 40, 30, 20, 10]`

`idx = 7`

🔄 Now Call `pop()` :

- `idx-- → 6`
- `arr[1] = arr[6] = 10`

After Replace Root:

```
    10
   /  \
  50   40
 / \
30 20
```

1st Iteration:

- `left = 2` , `right = 3`
- `arr[2] = 50` , `arr[3] = 40`
- `arr[2] > arr[3]` → left is bigger
- `arr[1] = 10` < `arr[2] = 50` → swap

After swap:

```
    50
   /  \
  10   40
```

```
 / \
30  20
```

2nd Iteration:

- `i = 2`
- `left = 4`, `right = 5`
- `arr[4] = 30`, `arr[5] = 20`
- `arr[4] > arr[5]` → left is bigger
- `arr[2] = 10` < `arr[4] = 30` → swap

After swap:

```
    50
   /  \
  30   40
 / \
10  20
```

3rd Iteration:

- `i = 4`
- `left = 8`, `right = 9` → out of bounds
- Exit loop

🏁 Final Heap:

```
    50
   /  \
  30   40
 / \
10  20
```

Array: `[x, 50, 30, 40, 10, 20]`

💡 Summary:

Step	Action
Remove root	Replace with last element
Heapify down	Compare children, swap with bigger child
Repeat	Until heap property is restored
Time Complexity	**O(log n)** (height of the heap)
