Heap(C++)

This C++ file is a comprehensive and practical guide to the **Heap data structure** and its most famous application, **Heap Sort**. It is brilliantly structured to compare different methods for creating a heap, highlighting the trade-offs in their efficiency.

The file begins by demonstrating the intuitive, one-by-one insertion method for building a max heap. This is shown both with a flexible std::vector and a standard C-style array, but it's noted that this approach has a time complexity of $O(n \log n)$.

It then presents the complete **Heap Sort algorithm**. This section is a masterclass in how a heap is used for sorting: it first builds a max heap and then systematically extracts the largest element and places it at the end of the array, repeating the process until the array is fully sorted.

Finally, the file introduces the more advanced and efficient **Heapify algorithm**. This is a "bottom-up" approach that can build a heap in linear time ($O(n)$), which is significantly faster than the one-by-one insertion method. By showcasing these different techniques side-by-side, the file provides an excellent foundation for understanding how heaps work and why they are so powerful.

**Create Heap Using STL Vector**

This section demonstrates how to build a **Max Heap** by inserting elements one at a time. This is a conceptually simple "top-down" approach. Every time a new element is added, it's placed at the end and then "bubbles up" to its correct position to maintain the heap property.

* Insert(vector<int>& vec, int key){ ... }

This function implements the "bubble-up" (or "swim") logic.

* + auto i = vec.size(); vec.emplace\_back(key);

The new key is first inserted at the very end of the vector.

* + while (i > 0 && key > vec[...]){ ... }

This while loop is the core of the insertion. It repeatedly compares the new key with its parent. The complex-looking index i % 2 == 0 ? (i/2)-1 : i/2 is the formula to find the parent's index in a 0-based array. If the key is larger than its parent, the parent is moved down, and the process continues until the heap property is restored.

* InsertInplace(int A[], int n){ ... }

This function does the exact same "bubble-up" logic as the Insert function but operates directly on a C-style array instead of a vector.

* CreateHeap(vector<int>& vec, int A[], int n){ ... }

This is the driver function that builds the entire heap. It simply iterates through the input array A and calls the Insert function for each element. Since each insertion takes O(logn) time, the total time to build the heap this way is O(nlogn).

**Heap Sort**

This section implements the classic **Heap Sort** algorithm. It's an efficient, in-place sorting algorithm that works in two main phases:

1. **Build Max Heap:** The unsorted array is first converted into a max heap.
2. **Sort by Deletion:** The largest element is repeatedly extracted from the heap and moved to its final sorted position at the end of the array.

* int H[] = {0, 10, 20, ...};

This implementation uses 1-based indexing for the array, so the element at index 0 is ignored. This simplifies the math for finding parent/child nodes (parent of i is i/2; children are 2i and 2i+1).

* for (i = 2; i <= 7; i++) Insert(H, i);

Phase 1: This loop builds the max heap using the one-by-one insertion method discussed in the previous section.

* int Delete(int A[], int n)

This function is the core of the sorting phase. It extracts the maximum element and then restores the heap property.

* + val = A[1]; ... A[1] = A[n]; A[n] = val;

The largest element (which is always at the root, A[1]) is swapped with the last element in the current heap. This moves the max element to the end of the array, where it belongs in the final sorted list.

* + while (j <= n - 1) { ... }

After the swap, the new root might be smaller than its children, violating the heap property. This while loop performs the "bubble-down" (or "sift-down") operation. It repeatedly compares the parent with its larger child and swaps them if needed, pushing the smaller element down until the heap property is restored.

* for (i = 7; i > 1; i--) { Delete(H, i); }

Phase 2: This is the main sorting loop. It calls Delete repeatedly. With each call, the largest remaining element is moved to the end, and the effective size of the heap is reduced by one. This process continues until the entire array is sorted.

**Heapify - Faster Method for creating Heap**

This section implements the **Heapify** algorithm, which is a much faster, "bottom-up" method for building a heap. Instead of inserting elements one by one, it starts from the last non-leaf node and works its way up to the root, ensuring the heap property is satisfied at each level. This method has a superior time complexity of **$O(n)$**.

* for (int i=(n/2)-1; i>=0; i--){ ... }

This is the main loop of the Heapify algorithm. In a 0-based array, the parent of the last element is at index (n/2)-1. The loop starts from this last parent and iterates backwards up to the root (i=0).

* while(j < n-1){ ... }

This inner while loop is the "sift-down" logic, which is very similar to the one used in the Delete function of Heap Sort. For each parent i in the outer loop, this code ensures that i is greater than its children, fixing the heap property for the subtree rooted at i.

* if (A[i] < A[j]){ swap(A, i, j); i = j; j = 2 \* i + 1; }

If a parent is smaller than its largest child, they are swapped. Crucially, the check must continue from the new position of the swapped element (so i is updated to j) to ensure the element sifts all the way down to its correct position in that branch. This is what makes the bottom-up approach work.