# （最大）堆：

* 逻辑上是一种特殊的完全二叉树，在存储上是一个数组
* 完全平衡的二叉树，最后一层叶子节点都在最左侧
* 每个节点的值大于等于其任一子节点的值
* child1 = parent \* 2 + 1;  
  child2 = parent \* 2 + 2;  
  parent = (child - 1) / 2;

# 应用

* 堆排序：最坏时间O(N\*logN)，空间O(1)，不稳定
* 优先队列（e.g., for best-bin-first ANN search）
* 维护最大/小的K个值，提供O(1)的最大值/最小值查询

# 基本操作

* **[O(logN)] restore**从某个节点恢复堆（sift-down）

template <typename T>

void MaxHeap\_siftdown(T\* heap, int heapSize, int begNodeIdx) // O(logN)

{

if (!heap || heapSize < 2 || begNodeIdx < 0) return;

while (true)

{

// find greater child

int childNodeIdx = begNodeIdx \* 2 + 1;

if (childNodeIdx >= heapSize) break;

if (childNodeIdx+1 < heapSize && heap[childNodeIdx+1] > heap[childNodeIdx])

++childNodeIdx;

// compare & swap with greater child

if (heap[begNodeIdx] >= heap[childNodeIdx])

break;

std::swap(heap[begNodeIdx], heap[childNodeIdx]);

begNodeIdx = childNodeIdx;

}

}

* **[O(N)] make**建立堆：依次从最后一个非叶子节点到根节点进行“恢复堆”操作

template <typename T>

void MaxHeap\_make(T\* heap, int heapSize) // O(N)

{

if (!heap || heapSize < 2) return;

int lastNonleafNodeIdx = (heapSize - 1 - 1) / 2;

for (int i = lastNonleafNodeIdx; i >= 0; --i)

MaxHeap\_siftdown(heap, heapSize, i);

}

* **[O(N\*logN)] sort**堆排序：最坏时间O(N\*logN)，空间O(1)，不稳定

template <typename T>

void MaxHeap\_sort(T\* arr, int num)

{

if (!arr || num < 2) return;

MaxHeap\_make(arr, num); // make heap O(N)

for (int i=num-1; i>0; --i) // O(N\*logN)

{

std::swap(arr[0], arr[i]); // get max element from heap top

MaxHeap\_siftdown(arr, i, 0); // restore heap

}

}

* **[O(logN)] pop**取堆顶元素并恢复堆：交换根节点与最后一个叶子节点，从新的根节点“恢复堆”

// (top-down) 取堆顶元素（移至堆尾）并恢复堆

template <typename T>

T\* MaxHeap\_pop(std::vector<T>& heap) // 返回原堆顶元素指针

{

if (heap.empty()) return NULL;

if (heap.size() == 1) return &(heap.front());

std::swap(heap.front(), heap.back()); // 将堆顶元素与堆尾元素互换

// sift-down the new top element

MaxHeap\_siftdown(&heap.front(), (int)heap.size() - 1, 0);

return &(heap.back());

}

* **[O(logN)] push**向堆中添加元素：新元素放在尾部，不断与父节点交换，直到达到根部或者小于等于父节点

// (bottom-up) 向建好的最大堆中添加一个元素

template <typename T>

void MaxHeapPush(std::vector<T>& heap, const T& newElem)

{

heap.push\_back(newElem); // add to the tail

int nodeIdx = (int)heap.size() - 1;

// shift up the node

while (nodeIdx > 0) // while not reach root

{

int parentIdx = (nodeIdx - 1) / 2;

if (heap[nodeIdx] <= heap[parentIdx])

break;

std::swap(heap[nodeIdx], heap[parentIdx]); // swap if greater than parent

nodeIdx = parentIdx;

}

}

# 库函数 #include <algorithm>

* std::make\_heap(a.begin(), a.end(), less<int>()); // 最大堆（默认）
* std::make\_heap(a.begin(), a.end(), greater<int>()); // 最小堆
* std::pop\_heap(a.begin(), a.end()); // 原堆顶放入尾部，恢复前n-1个元素的堆  
  之后通常调用a.pop\_back();
* std::push\_heap(a.begin(), a.end()); // 将尾元素加入原来包含n-1元素的堆  
  之前通常调用a.push\_back(newElem);
* std::sort\_heap(a.begin(), a.end()); // 堆排序，必须预先make\_heap