Day 8: Binary Heaps & Disjoint-Set Forests

Tinghai Zhang

January 9, 2025

Contents

1 Binary heaps

1.1 Priority queue ADT

Requirements

There are some items with ***priority***, and we need an ADT which supports:

- The next item to access or remove is the ***highest-priority*** item.
- New items may be added ***any time***.

One of common use cases: hospital emergency department.

Two basic implementation

- (Unsorted) array
  - Enqueue: add new item at the end of the array, $\mathcal{O}(1)$.
  - Dequeue: scan the array to find the highest-priority item, $\mathcal{O}(n)$.
- Sorted array
  - Enqueue: scan the array to find the right position for the new item, $\mathcal{O}(n)$.
  - Dequeue: remove the last item, $\mathcal{O}(1)$.

Entirely unsorted is too chaotic, but entirely sorted is unnecessary. A compromise is to use a ***heap***.

1.2 Binary heaps

Binary heaps store items ***partially sorted***. All the items are stored in a ***binary tree***, which satisfies:

- The tree is ***complete***, i.e. nodes in it are filled left-to-right on each level (row) of the tree.
  - The tree is complete if and only if the array representation of the tree is filled from index 0 to $n - 1$.
- The tree is ***heap-ordered***, i.e. the value of each node is ***greater than or equal to*** the values of its children. We call the property ***max-heap*** property. The ***min-heap*** property is defined similarly.

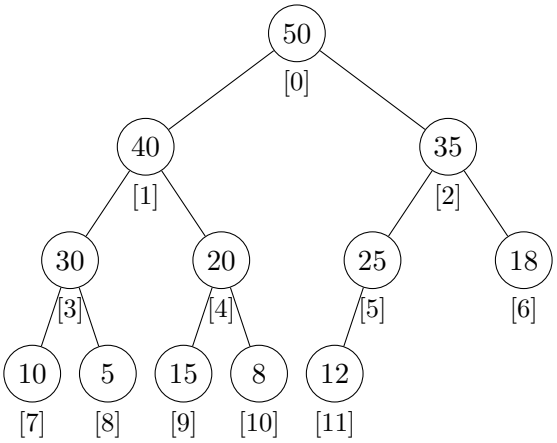


Figure 1 A max-heap binary tree

The numbers in the square brackets are the indices of the array representation of the binary tree.

1.3 Implementation

Find parent and child nodes

The algorithm to find the parent and child nodes of a node at index i in the array representation of a binary heap is shown in Algorithm 1.

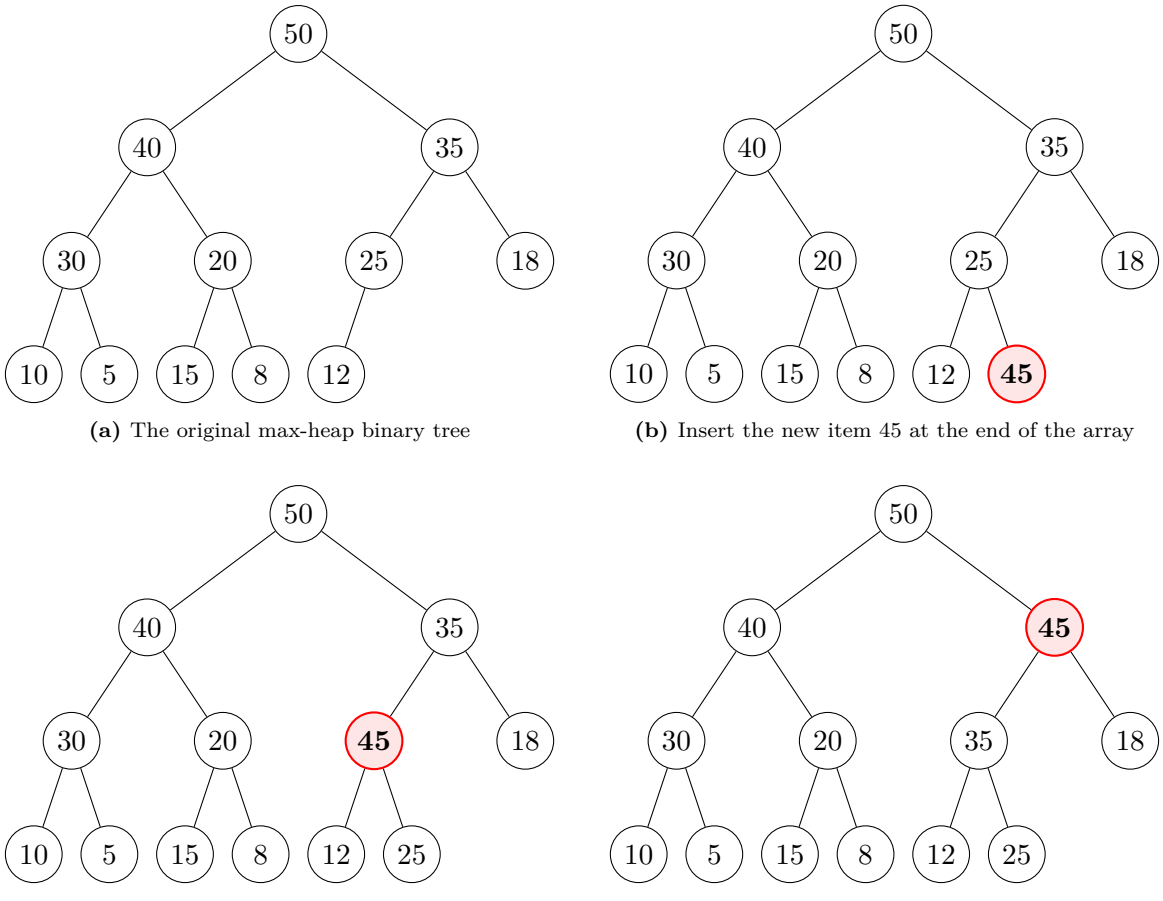
Insert a new item

When inserting a new item into a max-heap, we add it to the end of the array and then ***float*** it up to maintain the max-heap property. During this process, the new item is swapped with its parent until the max-heap property is satisfied.

An example to insert a new item is shown in Figure 2. The time complexity of inserting a new item is $\mathcal{O}(\log n)$. The algorithm is shown in Algorithm 2.

Algorithm 1: Find parent and child nodes

```
Function PARENT(i)
    return ⌊(i − 1)/2⌋
end
Function LEFT-CHILD(i)
    return 2i + 1
end
Function RIGHT-CHILD(i)
    return 2i + 2
end
```



(a) The original max-heap binary tree

(b) Insert the new item 45 at the end of the array

(c) Swap 45 with its parent 25

(d) Swap 45 with its parent 35

Figure 2 An example of inserting a new item into a max-heap binary tree

Algorithm 2: Add a new item to a max-heap

```
1:  Function INSERT(A, x)
2:      A.PUSH-BACK(x)
3:      FLOAT-UP(A.size − 1)
4:  end
5:  Function FLOAT-UP(i)
6:      while i > 0 and A[i] > A[PARENT(i)] do
7:          Swap A[i] and A[PARENT(i)]
8:          i ← PARENT(i)
9:      end
10: end
```

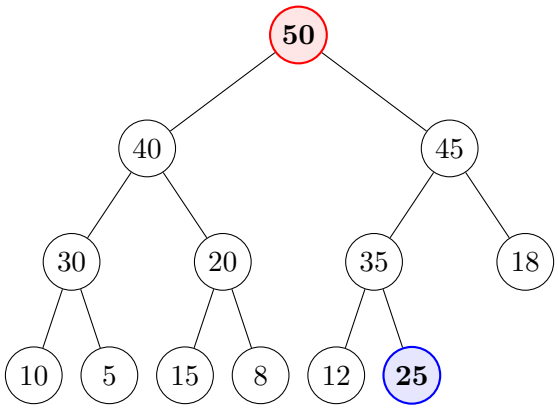
Delete the highest-priority item

When deleting the highest-priority item from a max-heap, we first swap the root with the last item in the array, then ***sink*** the new root down to keep the max-heap property. The ***sink-down*** operation is similar to the ***float-up*** operation, but we swap the current node with its larger child until the max-heap property is satisfied.

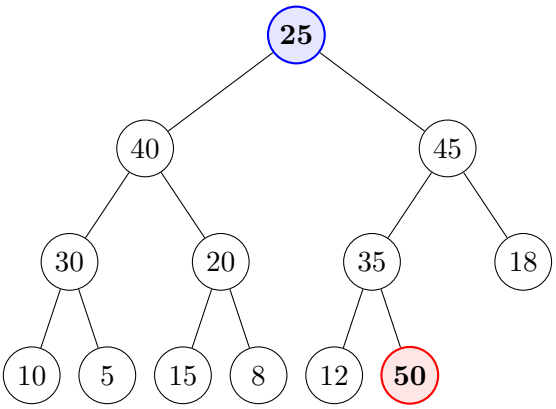
An example to delete the highest-priority item is shown in Figure 3. The time complexity of deleting the highest-priority item is $\mathcal{O}(\log n)$. The algorithm is shown in Algorithm 3.

Algorithm 3: Delete the highest-priority item from a max-heap

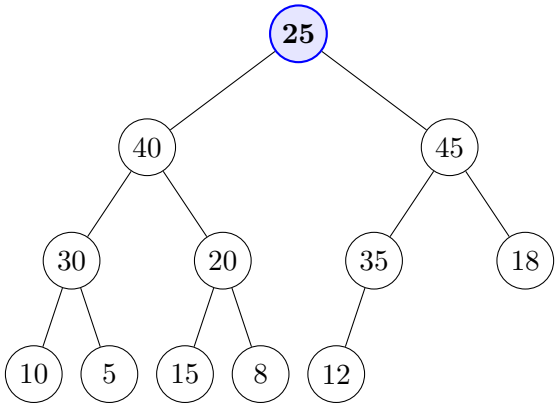
```
1:  Function DELETE(A)
2:      Swap A[0] and A[A.size − 1]
3:      A.POP-BACK()
4:      SINK-DOWN(A, 0)
5:  end
6:  Function SINK-DOWN(A, i)
7:      while LEFT-CHILD(i) < A.size do
            // Find the larger child
8:          j ← LEFT-CHILD(i)
9:          if j + 1 < A.size and A[j + 1] > A[j] then
10:             j ← j + 1
11:         end
12:         if A[i] ⩾ A[j] then return
13:         Swap A[i] and A[j]
14:         i ← j
15:     end
16: end
```



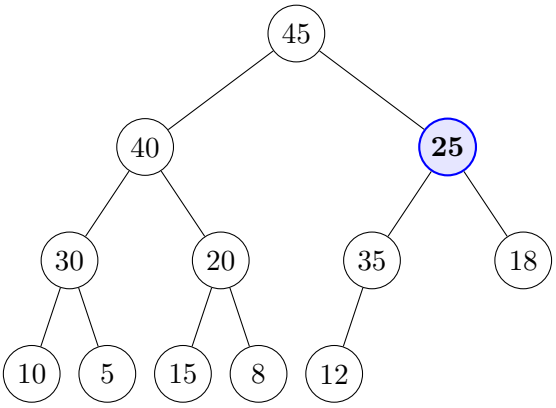
(a) The original max-heap binary tree



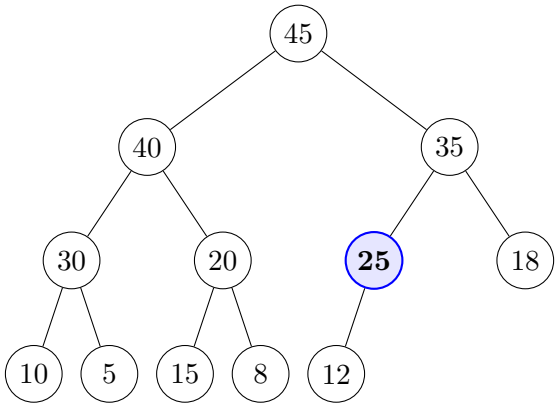
(b) Swap the root with the last item in the array



(c) Delete the last item in the array



(d) Swap the new root with its largest child 45



(e) Swap 25 with its largest child 35

Figure 3 An example of deleting the highest-priority item from a max-heap binary tree

Build a heap from an (unsorted) array

When building a heap from an (unsorted) array, we can perform the ***sink-down*** operation from the last non-leaf node to the root. After each operation, the subtree rooted at the current node satisfies the max-heap property.

Algorithm 4: Build a heap from an (unsorted) array

```
Function BUILD-HEAP(A)
    for i ← ⌊A.size/2⌋ − 1 downto 0 do
        SINK-DOWN(A, i)
    end
end
```

Remark. The node with index i has a child iff LEFT-CHILD(i) $< A.size$, i.e. $2i + 1 < A.size$. Hence $i \leqslant \lfloor A.size/2 \rfloor - 1$.

We consider a complete binary tree with depth N and $n = 2^{N+1} - 1$ nodes. The number of operation we need when building a heap is

$$\sum_{i=0}^{N-1} 2^i (N - i) = 2^{N+1} - N - 2 = \mathcal{O}(n).$$

Therefore, the time complexity of building a heap from an (unsorted) array is $\mathcal{O}(n)$. We should **NOT** build a heap by inserting items one by one, which takes $\mathcal{O}(n \log n)$ time.

1.4 Priority queue in C++ STL

A ***priority queue*** in C++ STL is implemented by a heap. We can define a priority queue of integers by codes below:

```
1 #include <queue>
2 #include <vector>
3 using namespace std;
4
5 priority_queue<int> pq;  // max-heap
6 priority_queue<int, vector<int>, greater<int>> pq_min;  // min-heap
```

Common operations of a priority queue are listed in Table 1.

Operation	Description	Time complexity
`pq.push(x)`	Insert a new item x	$\mathcal{O}(\log n)$
`pq.pop()`	Delete the highest-priority item	$\mathcal{O}(\log n)$
`pq.top()`	Return the highest-priority item	$\mathcal{O}(1)$

Table 1 Operations of a priority queue in C++ STL

1.5 Example: [Luogu] P1801

Description

A ***black box*** is a rudimentary form of database which stores an array of intergers and a special variable i. At the beginning, the array is empty and $i = 0$. The black box supports the following operations:

- `ADD(x)`: Add x to the black box.
- `GET`: i increases by 1, and return the i-th smallest number in the array.

Now you are required to find a best way to process a given sequence of operations. The sequence includes n `ADD` operations and m `GET` operations. Two arrays of interger are used to describe the operation sequence:

- $a_1, a_2, \ldots, a_n$: a sequence of integers to be added to the black box.
- $u_1, u_2, \ldots, u_m$: a `GET` operation is performed when the u_i-th number is added to the black box. No illegal operation will be contained in the sequence.

Constraints

- $1 \leqslant n, m \leqslant 2 \times 10^5$;
- $|a_i| \leqslant 2 \times 10^9$;
- $\{u_i\}$ is non-decreasing.
- Time limit: 0.5s / Memory limit: 500MB.

Solution

We can use a max-heap to maintain the smallest i numbers, and a min-heap to maintain the rest numbers.

- When a new number is added:
  - If the number is smaller than the top of the max-heap, we add it to the max-heap, and move the top of the max-heap to the min-heap.
  - Otherwise, we add it to the min-heap.
- When a `GET` operation is performed, we move the top of the min-heap to the max-heap, and print it.

The time complexity of each operation is $\mathcal{O}(\log n)$, so the total time complexity is $\mathcal{O}((n + m) \log n)$.

2 Disjoint-set forests

2.1 Disjoint-set

A ***disjoint-set*** data structure maintains a collection of disjoint sets, each of which is represented by a unique ***representative***. We can perform the following operations on

disjoint sets:

- MAKE-SET(x): create a new set containing x.
- UNION(x, y): merge the sets containing x and y.
- FIND(x): find the representative of the set containing x.
- IS-SAME(x, y): check whether x and y are in the same component.

2.2 Disjoint-set forests

A fast implementation of disjoint-set is ***disjoint-set forests***. We represent each set as a ***tree***, where the representative is the ***root*** of the tree.

A MAKE-SET operation creates a tree with a single node. A UNION operation merges two trees by making the root of one tree a child of the root of the other tree. A FIND operation returns the root of the tree containing x.

An example of disjoint-set forests is shown in Figure 4.

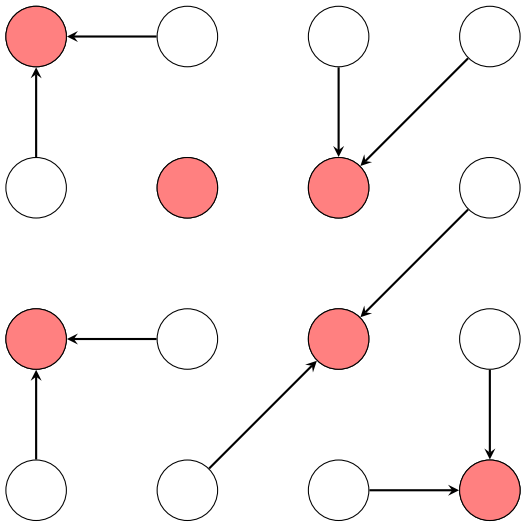


Figure 4 An example of disjoint-set forests

Heuristics to improve efficiency

- **Union by rank**: we attach the tree with smaller ***rank*** to the tree with larger rank. Here the rank of a node is the height of the subtree rooted at the node.
- **Path compression**: when we find the root of the tree containing x, we make all the nodes on the path from x to the root point directly to the root.

With both heuristics, the time complexity of each operation is $\mathcal{O}(\alpha(n))$, where n is the number of elements and $\alpha(\cdot)$ is the ***inverse Ackermann function***. The function grows very slowly, so we can consider it as a constant.

The algorithm is shown in Algorithm 5.

Algorithm 5: Disjoint-set forests

```
1:  Function MAKE-SET(x)
2:      parent[x] ← x
3:      rank[x] ← 0
4:  end
5:  Function FIND(x)
6:      if x ≠ parent[x] then
7:          parent[x] ← FIND(parent[x])
8:      end
9:      return parent[x]
10: end
11: Function IS-SAME(x, y)
12:     return FIND(x) = FIND(y)
13: end
14: Function LINK(x, y)
15:     if rank[x] > rank[y] then
16:         parent[y] ← x
17:     else
18:         parent[x] ← y
19:         if rank[x] = rank[y] then
20:             rank[y] ← rank[y] + 1
21:         end
22:     end
23: end
24: Function UNION(x, y)
25:     x ← FIND(x), y ← FIND(y)
26:     if x ≠ y then
27:         LINK(x, y)
28:     end
29: end
```

2.3 Application: Kruskal's algorithm

Minimum spanning tree

A ***minimum spanning tree*** of a connected, undirected graph is a spanning tree with the smallest possible sum of edge weights. A minimum spanning tree is unique if the edge weights are distinct.

Kruskal's algorithm

Kruskal's algorithm is a greedy algorithm that finds a minimum spanning tree for a connected, undirected graph. In each iteration, the algorithm finds the edge with the smallest weight which links two different components, and adds it to the minimum spanning tree. We repeat this process until all the vertices are connected.

The algorithm is shown in Algorithm 6.

Algorithm 6: Kruskal's algorithm

```
1:  Function KRUSKAL(G)
2:      Sort G.edges by weight in non-decreasing order
3:      for v ∈ G.vertices do
4:          MAKE-SET(v)
5:      end
6:      T ← ∅
7:      for e ∈ G.edges do
8:          if IS-SAME(e.from, e.to) = false then
9:              T ← T ∪ {e}
10:             UNION(e.from, e.to)
11:         end
12:     end
13:     return T
14: end
```