Day 8: Data Structures II
Binary Heaps & Disjoint Set Forests

Tinghai Zhang

Jan 9, 2025

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.

Priority queue ADT
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**.

Binary heaps
Definition

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.

Binary heaps
An example
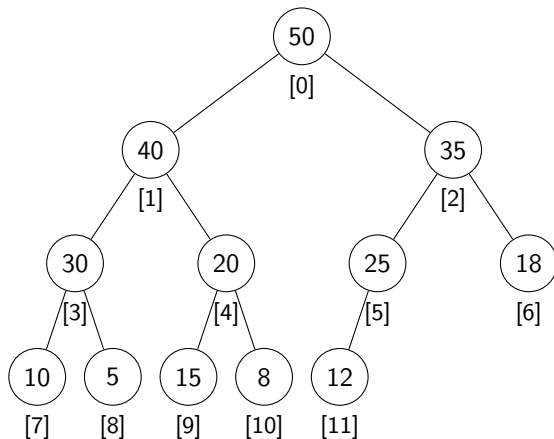
Implementation
Find parent and child nodes

Algorithm 1: Find parent and child nodes

1: **Function** $\mathrm{PARENT}(i)$
2:    | **return** $\lfloor (i-1)/2 \rfloor$
3: **end**

4: **Function** $\mathrm{LEFT\text{-}CHILD}(i)$
5:    | **return** $2i+1$
6: **end**

7: **Function** $\mathrm{RIGHT\text{-}CHILD}(i)$
8: | **return** $2i+2$
9: **end**

Implementation
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.

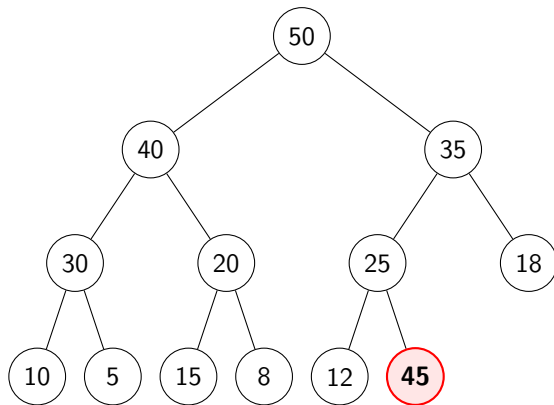The time complexity of inserting a new item is $\mathcal{O}(\log n)$.
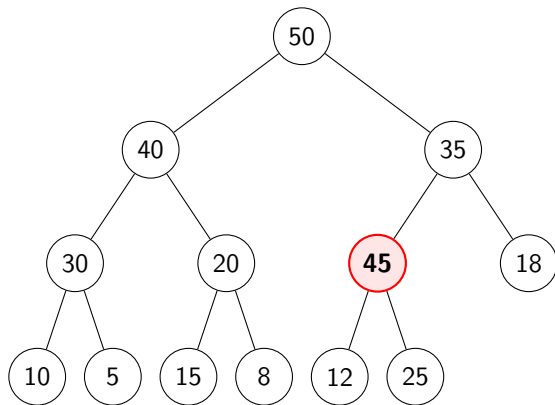
Implementation
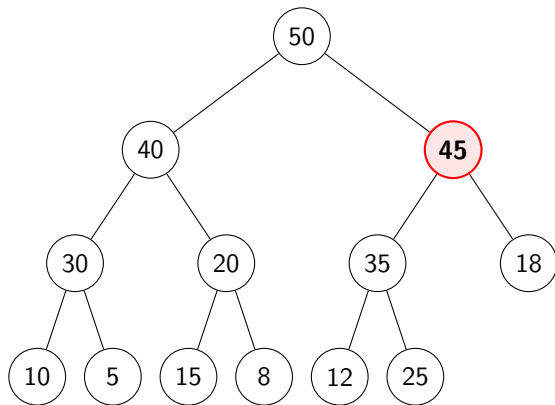## Insert a new item: Example
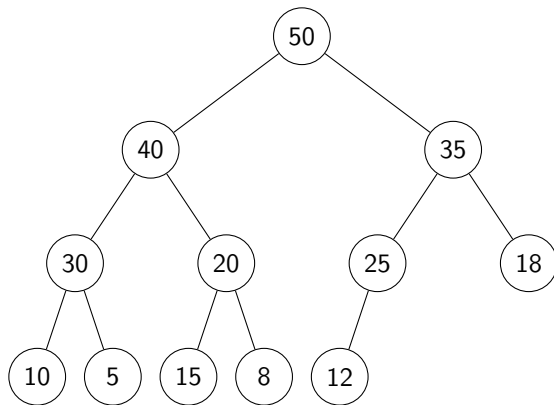
Implementation
Insert a new item: Example

Implementation
Insert a new item: Example

Implementation
Insert a new item: Example

Implementation
Insert a new item: Algorithm

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 $A[i] > A[\text{PARENT}(i)]$ **do**
7:         Swap $A[i]$ and $A[\text{PARENT}(i)]$
8:         $i \leftarrow \text{PARENT}(i)$
9:   **end**
10: **end**

Implementation
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.

The time complexity of deleting the highest-priority item is $\mathcal{O}(\log n)$.

Implementation
Delete the highest-priority item: Example
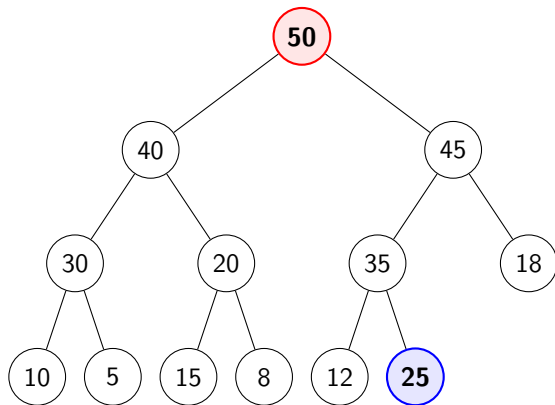
Implementation
Delete the highest-priority item: Example

Implementation
Delete the highest-priority item: Example
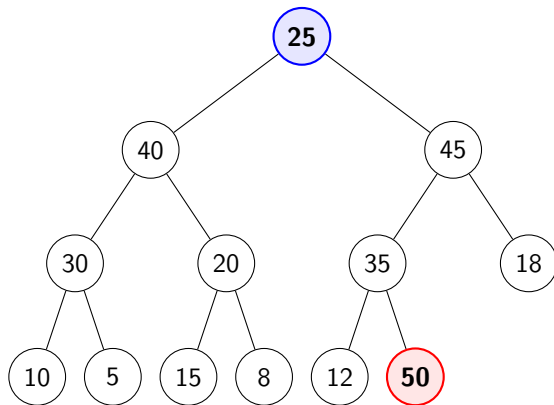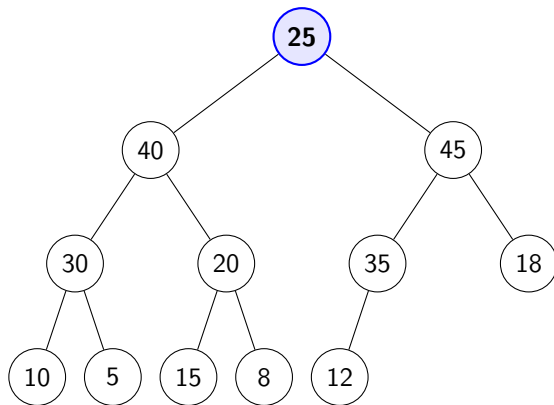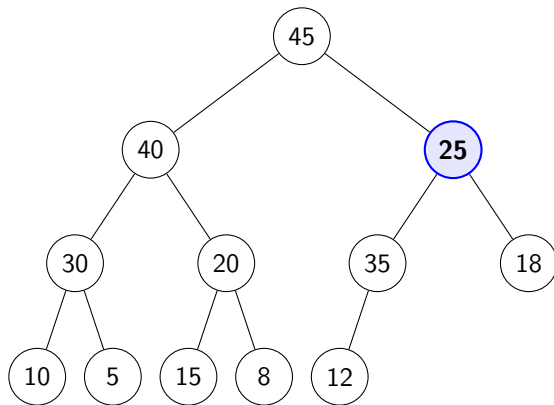
Implementation
Delete the highest-priority item: Example

Implementation
Delete the highest-priority item: Example
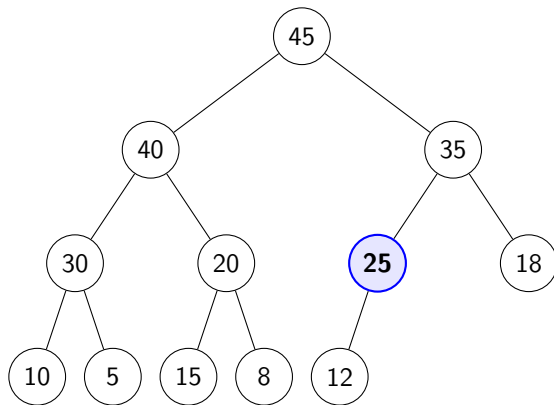
Implementation
Delete the highest-priority item: Algorithm

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**
8:         $j \leftarrow$ LEFT-CHILD(i)
9:         **if** $j + 1 < A.size$ and $A[j+1] > A[j]$ **then** $j \leftarrow j + 1$
10:        **if** $A[i] \geqslant A[j]$ **then return**
11:        Swap $A[i]$ and $A[j]$
12:        $i \leftarrow j$
13:    **end**
14: **end**

Implementation
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

1: **Function** BUILD-HEAP(A)
2:      **for** $i \leftarrow \lfloor A.size/2 \rfloor - 1$ **downto** 0 **do**
3:          SINK-DOWN(A, i)
4:      **end**
5: **end**

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

Implementation
Build a heap from an (unsorted) array: Time complexity

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.

Implementation
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
3  priority_queue<int> pq;                      // max-heap
4  priority_queue<int, greater<int>> pq_min;    // min-heap
```

Implementation
Priority queue in C++ STL

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 Operations of a priority queue in C++ STL

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 ($n \leqslant 2 \times 10^5$) ADD operations and m ($m \leqslant 2 \times 10^5$) 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. $|a_i| \leqslant 10^9$.
- $u_1, u_2, \ldots, u_m$: a GET operation is performed when the u_i-th number is added to the black box. $\{u_i\}$ is non-decreasing. No illegal operation will be contained in the sequence.

Example: [Luogu] P1801
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 $\mathcal{O}((n + m) \log n)$.

Disjoint-set
Definition

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.

Disjoint-set forests
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.

Disjoint-set forests

Example
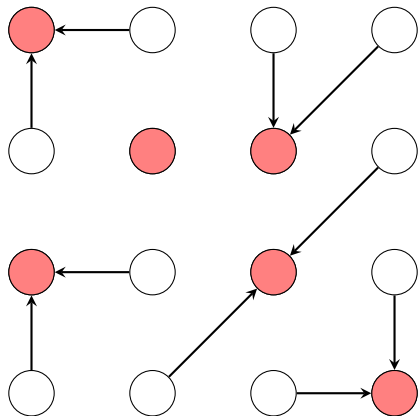
Disjoint-set forests
Heuristics to improve efficiency

- **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.

Disjoint-set forests
Algorithm

Algorithm 5: Disjoint-set forests

1: **Function** $\mathrm{MAKE\text{-}SET}(x)$
2:     $parent[x] \leftarrow x$
3:     $rank[x] \leftarrow 0$
4: **end**

5: **Function** $\mathrm{FIND}(x)$
6:     **if** $x \neq parent[x]$ **then** $parent[x] \leftarrow \mathrm{FIND}(parent[x])$
7:     **return** $parent[x]$
8: **end**

9: **Function** $\mathrm{IS\text{-}SAME}(x, y)$
10:     **return** $\mathrm{FIND}(x) = \mathrm{FIND}(y)$
11: **end**

Disjoint-set forests
Algorithm

Algorithm 6: Disjoint-set forests

1: **Function** LINK(x, y)
2:      **if** $rank[x] > rank[y]$ **then** $parent[y] \leftarrow x$
3:      **else**
4:          $parent[x] \leftarrow y$
5:          **if** $rank[x] = rank[y]$ **then**
6:              $rank[y] \leftarrow rank[y] + 1$
7:          **end**
8:      **end**
9: **end**

10: **Function** UNION(x, y)
11:      $x \leftarrow$ FIND(x), $y \leftarrow$ FIND(y)
12:      **if** $x \neq y$ **then** LINK(x, y)
13: **end**

Application: Kruskal's algorithm
Minimum spanning tree & Kruskal's algorithm

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 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.

Application: Kruskal's algorithm
Algorithm

Algorithm 7: Kruskal's algorithm

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

Thanks for your attention!