Data Structures (IIa)

Ethen Yuen {ethening}

2021-04-24

香港電腦奧林匹克競賽
Hong Kong Olympiad in Informatics

Reference

- The slides are mainly adapted from Data Structures (II) slides (2019) by Alex Poon, Graph (II) slides (2019) by Percy Wong

Related Topics in HKOI training

Data Structures (I) (stack, queue, linked list)	Recursion, Divide and Conquer
Data Structures (II) (heap, BST, hash table, DSU)	Advanced C++ STL
Data Structures (III) (sparse table, segment tree, binary indexed tree)	Graph (II) (Shortest Path algorithm, MST)

Agenda

- Introduction to Data Structure
- Binary Heap
- Disjoint Set Union-find (DSU)
- Common Tricks in Data Structure

Data Structure

- A way of organizing some data
- Usually, it is a container to store data which support the following operations:
  - Insert data to the container
  - Erase data from the container
  - Update the existing data
  - Query the existing data
- Different data structures have their specific rules to store & organize the data
  - In order to attain better time complexity in some kinds of operation

Array / Vector

- Operations supported:
  - Insert any element → O(1)
  - Erase any element → O(n)
  - Update an element → O(n)
  - Query an element → O(n)
- Support many kinds of operation but slow
- Note: Array stores element in continuous cell, I am not talking about frequency array e.t.c

Sorted Array / Sorted Vector

- Operations supported:
  - Insert any element → O(n) as you need to make it remain sorted
  - Erase any element → O(n)
  - Any kind of update → O(n)
  - Any kind of query → O(log n) as you can use binary search
- Faster in query but slower in insertion

Data Structures & C++ library

- As you may have already known, some data structures are included in C++ library, so instead of code it, we may just learn how to use it.
- vector == array / sorted array
- priority_queue == Binary Heap
- set, map ~ Binary Search Tree
- unordered_map / unordered_set == Hash Table

Data Structures & C++ library

- Why are we still learning to implement the data structure?
- Knowing how to modifying the data structure for specific uses
- Answering questions in job interviews
- For Fun!

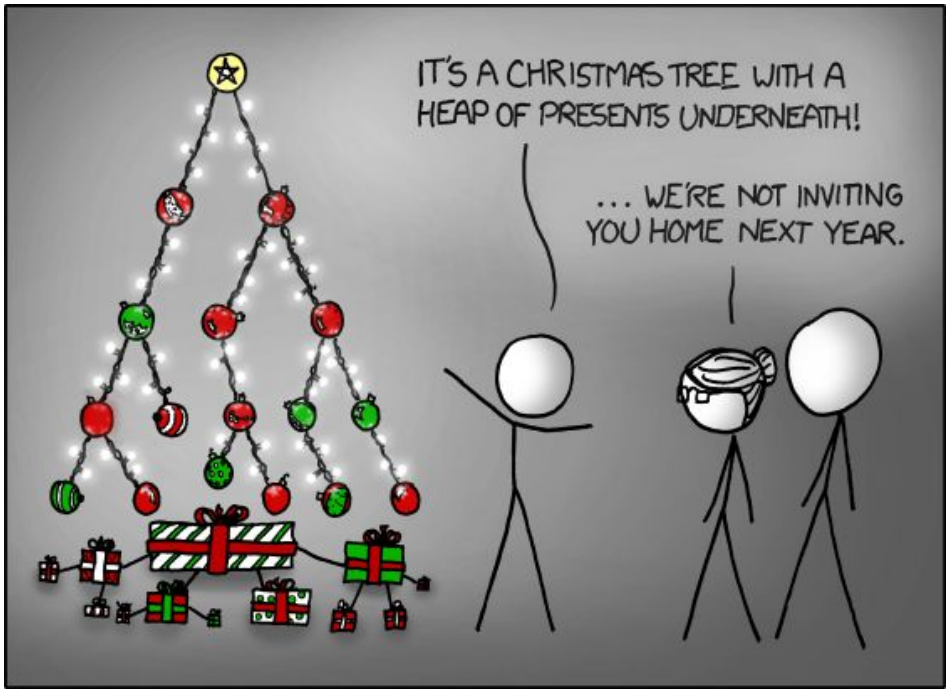
Agenda

- Introduction to Data Structure
- Binary Heap
  - Importance to learn how to code: **
  - Importance to learn how to use: *****
- Disjoint Set Union-find (DSU)
- Common Tricks in Data Structure

Binary Heap

- Operations supported:
  - Insert any number (or element / structure etc) → O(log n)
  - Erase, Update, Query any number → Not supported
  - Erase min → O(log n)
  - Query min → O(1)
- Very good data structure if you need something able to insert & query min/max

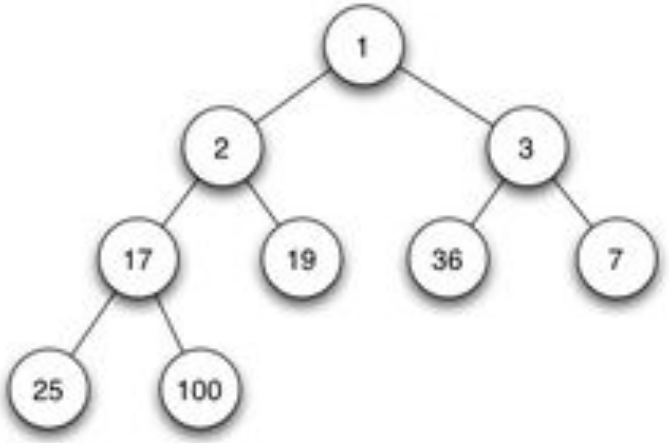
Binary Heap - Format



https://xkcd.com/835/

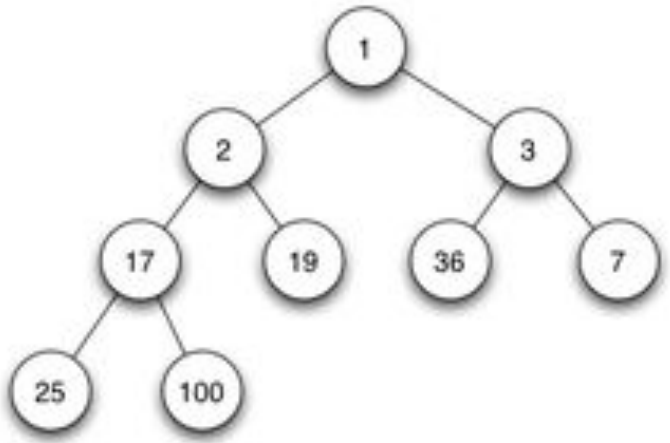
Binary Heap - Format

- In a Binary Heap, each element is stored in the node of the Complete Binary Tree



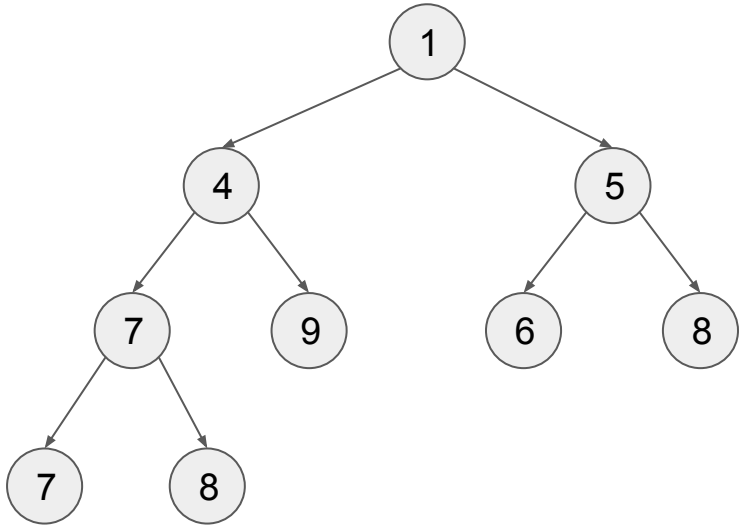
Binary Heap - Property

- Each element should not lesser than its father
- By this constraint, we know that the min is stored in the root
  - Query min. O(1)
- We are able to retain this structure when insert a new element or erase the min very efficiently
  - O(log n) for retain after insert / erase min.



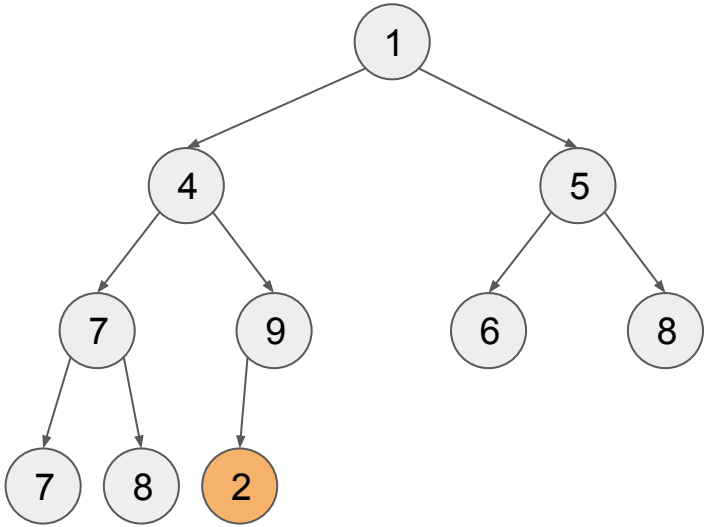
Binary Heap - Insertion

- Add 2 to the existing heap



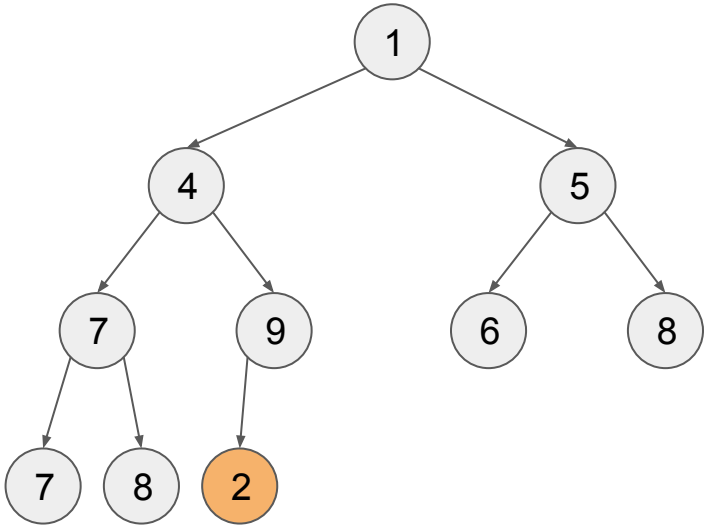
Binary Heap - Insertion

- Add 2 to the existing heap
  - Step 1: Put 2 in the next node in the complete binary tree



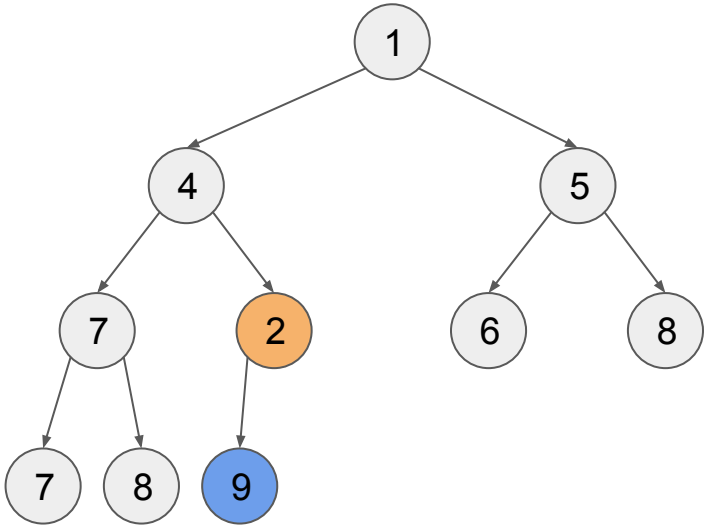
Binary Heap - Insertion

- Add 2 to the existing heap
  - Step 1: Put 2 in the next node in the complete binary tree
  - Step 2: Sift-up the node to maintain the property of heap
    Each element >= father



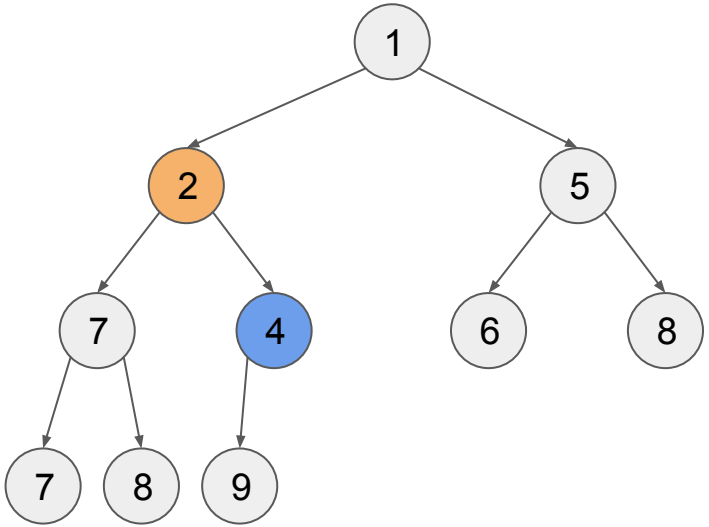
Binary Heap - Insertion

- Add 2 to the existing heap
  - Step 1: Put 2 in the next node
    in the complete binary tree
  - Step 2: Sift-up the node to maintain
    the property of heap
    Each element >= father
    2 < 9, swap



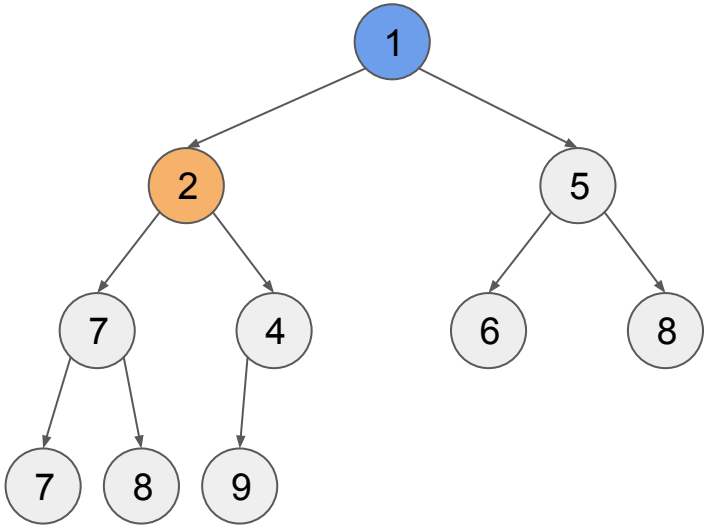
Binary Heap - Insertion

- Add 2 to the existing heap
  - Step 1: Put 2 in the next node
    in the complete binary tree
  - Step 2: Sift-up the node to maintain
    the property of heap
    Each element >= father
    2 < 4, swap



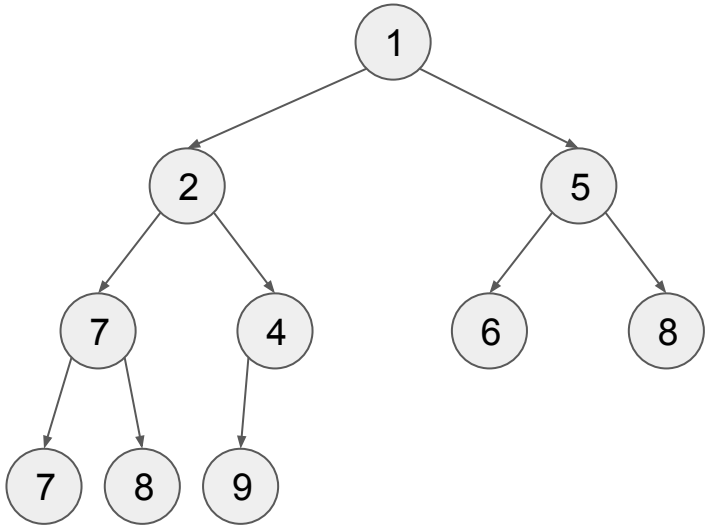
Binary Heap - Insertion

- Add 2 to the existing heap
  - Step 1: Put 2 in the next node
    in the complete binary tree
  - Step 2: Sift-up the node to maintain
    the property of heap
    Each element >= father
    2 >= 1, no need swap, done



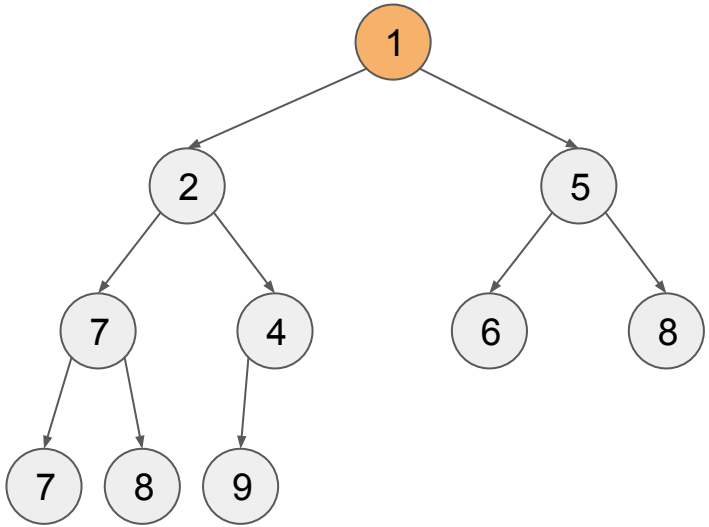
Binary Heap - Insertion

- Time complexity of insert:
  - = number of sift-up needed
  - = level of the current heap
- Level of current heap:
  - K-level heap contains 1+2+4+ ... 2^(k-1)
    = (2^k) - 1 nodes
  - Level of heap = log(number of nodes)
  - O(log n)



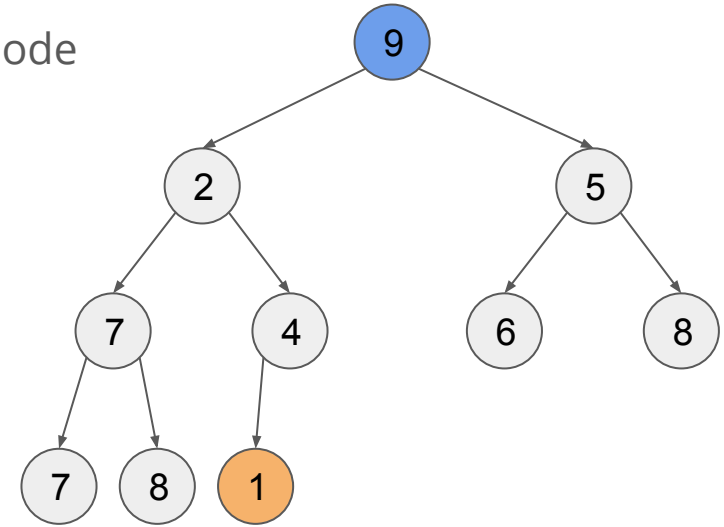
Binary Heap - Query Minimum

- Easy → Just find the root → O(1)



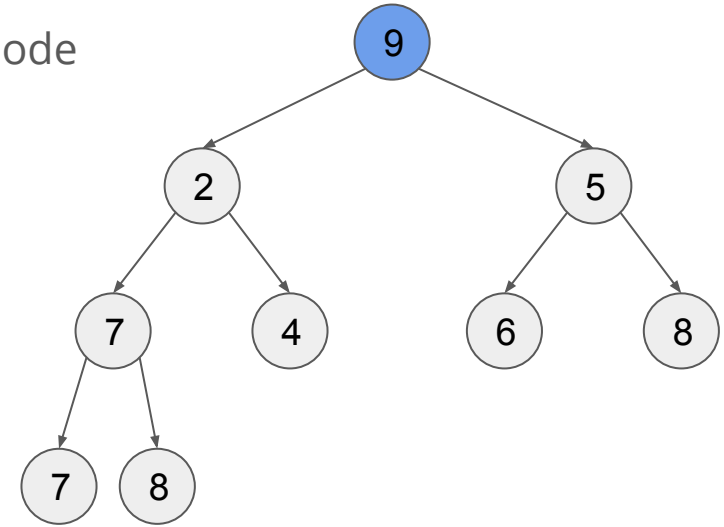
Binary Heap - Remove the Minimum

- Step 1: Swap the minimum (root) and the last node



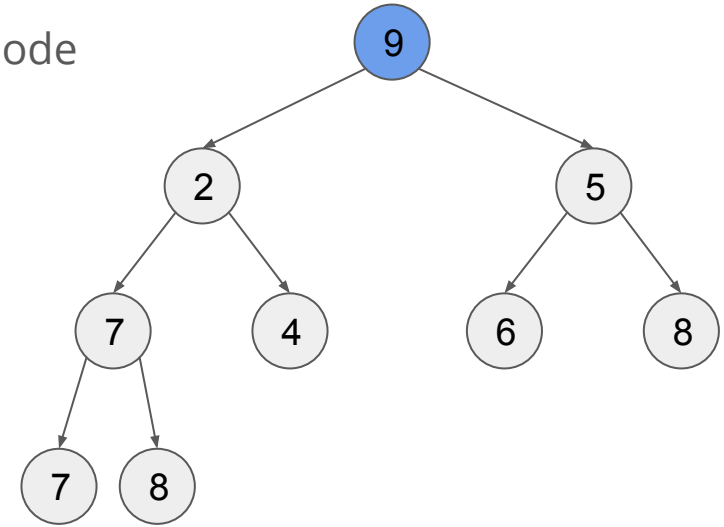
Binary Heap - Remove the Minimum

- Step 1: Swap the minimum (root) and the last node
- Step 2: Erase the last node



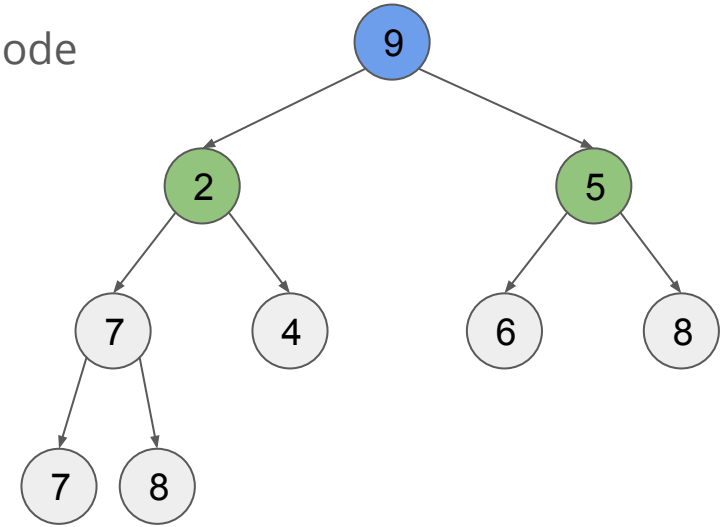
Binary Heap - Remove the Minimum

- Step 1: Swap the minimum (root) and the last node
- Step 2: Erase the last node
- Step 3: Sift-down the root to maintain
  the property of heap
  each element >= father



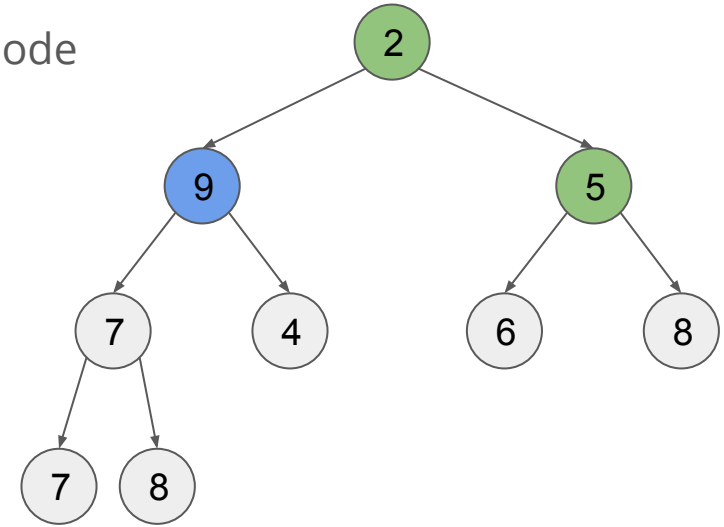
Binary Heap - Remove the Minimum

- Step 1: Swap the minimum (root) and the last node
- Step 2: Erase the last node
- Step 3: Sift-down the root to maintain
  the property of heap
  each element >= father
  - ***Consider 9 and its children (2, 5)***



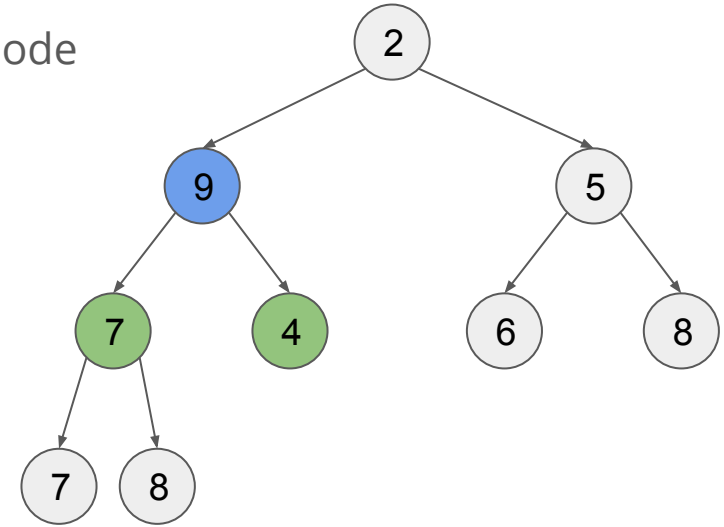
Binary Heap - Remove the Minimum

- Step 1: Swap the minimum (root) and the last node
- Step 2: Erase the last node
- Step 3: Sift-down the root to maintain
  the property of heap
  each element >= father
  - ***Swap 9 with the smallest (2)***



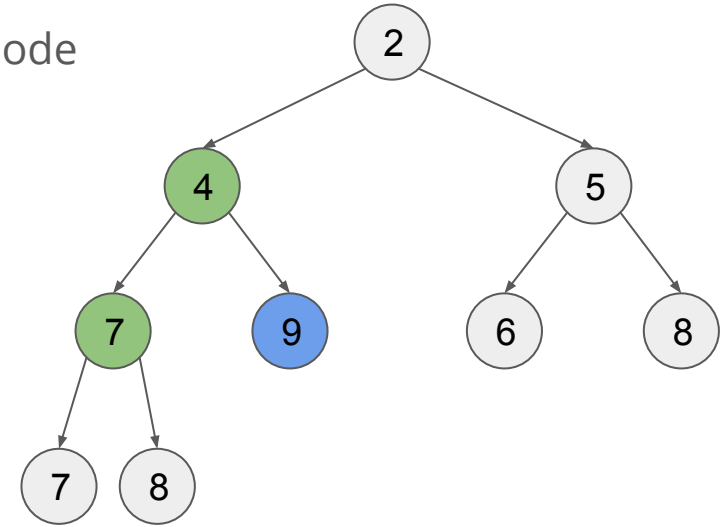
Binary Heap - Remove the Minimum

- Step 1: Swap the minimum (root) and the last node
- Step 2: Erase the last node
- Step 3: Sift-down the node to maintain
  the property of heap
  each element >= father
  - ***Consider 9 and its children (7, 4)***



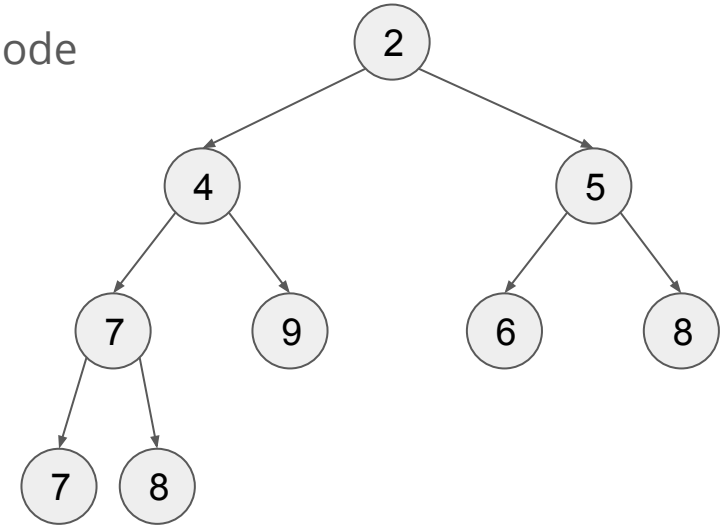
Binary Heap - Remove the Minimum

- Step 1: Swap the minimum (root) and the last node
- Step 2: Erase the last node
- Step 3: Sift-down the node to maintain
  the property of heap
  each element >= father
  - ***Swap 9 with the smallest (4)***



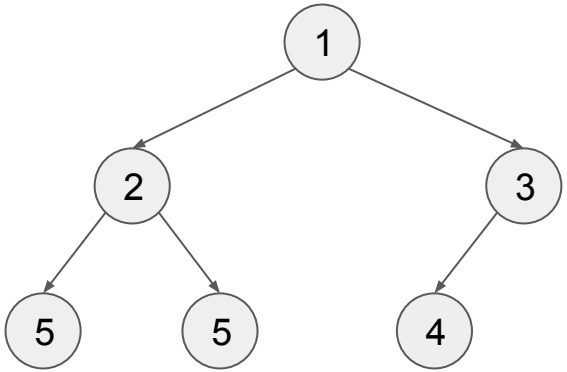
Binary Heap - Remove the Minimum

- Step 1: Swap the minimum (root) and the last node
- Step 2: Erase the last node
- Step 3: Sift-down the node to maintain
  the property of heap
  each element >= father
  - ***Consider 9 and its children, no → done***



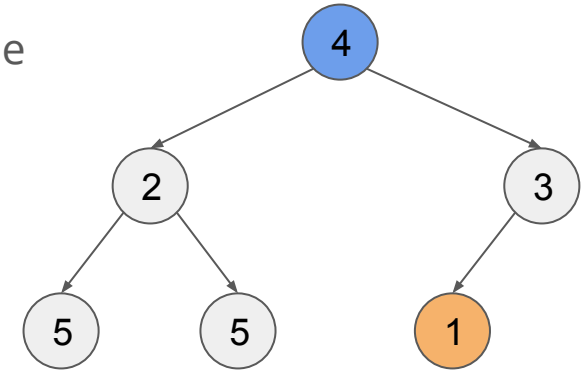
Binary Heap - Remove the Minimum

- Example 2 - Remove min in this heap



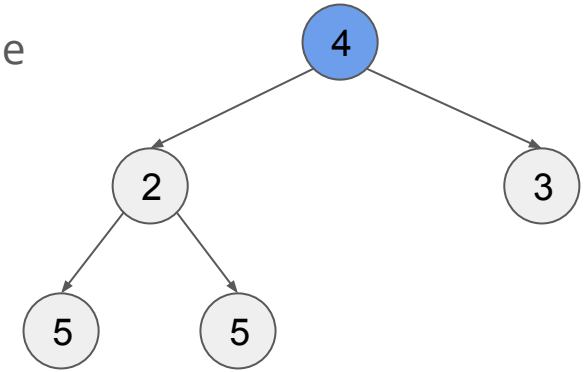
Binary Heap - Remove the Minimum

- Step 1: Swap the minimum (root) and the last node



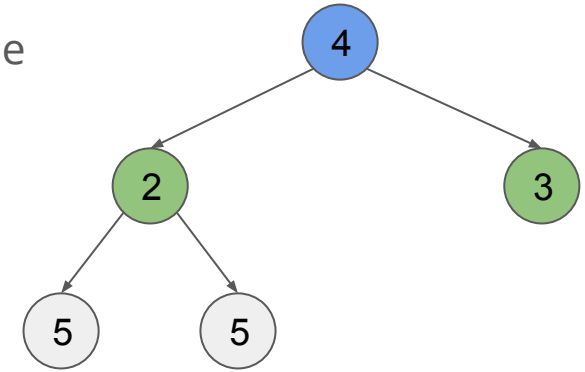
Binary Heap - Remove the Minimum

- Step 1: Swap the minimum (root) and the last node
- Step 2: Erase the last node



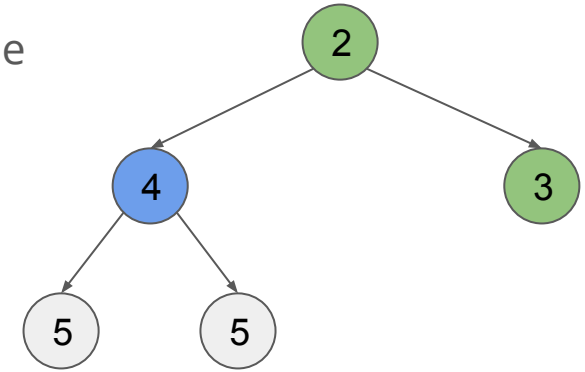
Binary Heap - Remove the Minimum

- Step 1: Swap the minimum (root) and the last node
- Step 2: Erase the last node
- Step 3: Sift-down the root to maintain
  the property of heap
  each element >= father
  - ***Consider 4 and its children***



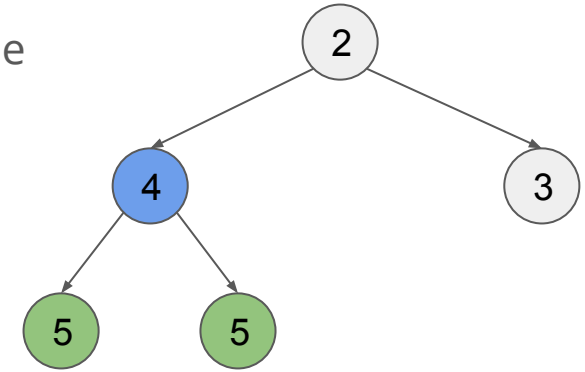
Binary Heap - Remove the Minimum

- Step 1: Swap the minimum (root) and the last node
- Step 2: Erase the last node
- Step 3: Sift-down the root to maintain
  the property of heap
  each element >= father
  - ***Swap 4 with the smallest (2)***



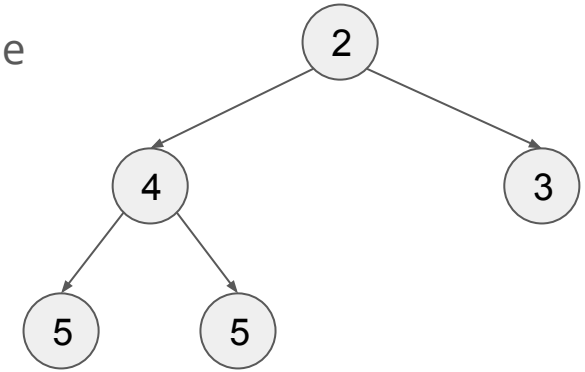
Binary Heap - Remove the Minimum

- Step 1: Swap the minimum (root) and the last node
- Step 2: Erase the last node
- Step 3: Sift-down the node to maintain
  the property of heap
  each element >= father
  - ***Consider 4 and its children***



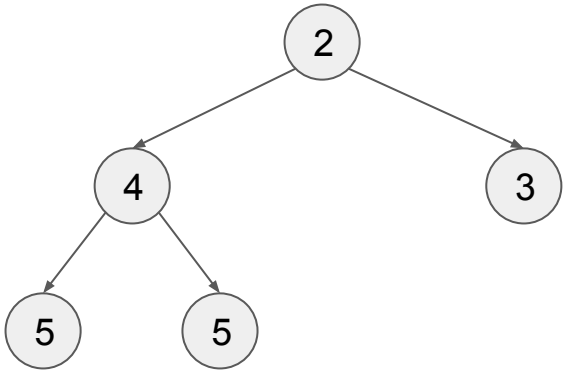
Binary Heap - Remove the Minimum

- Step 1: Swap the minimum (root) and the last node
- Step 2: Erase the last node
- Step 3: Sift-down the node to maintain
  the property of heap
  each element >= father
  - ***Swap 4 with the smallest (4) → no need swap → done***



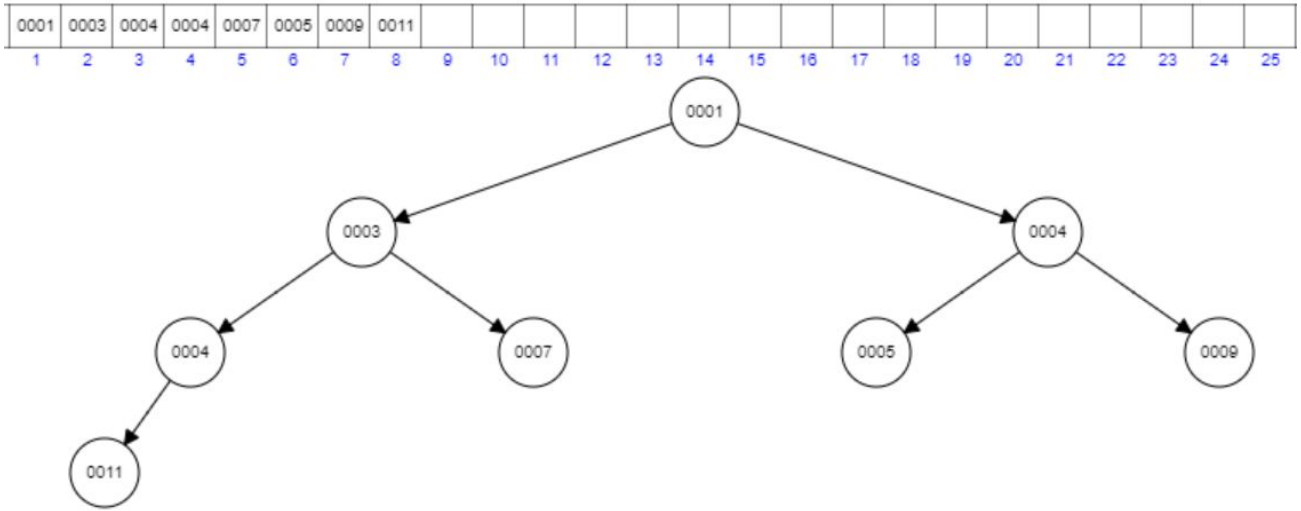
Binary Heap - Remove the Minimum

- Time Complexity: O(log n) again
- Proof same as insert



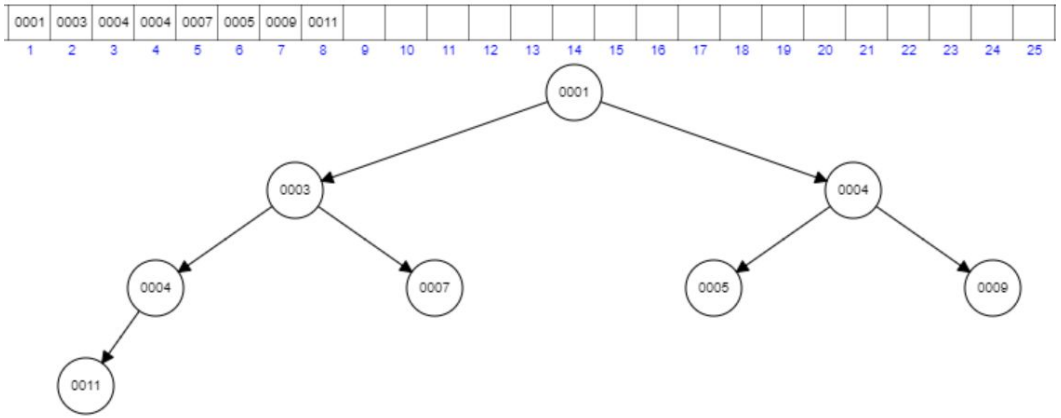
Binary Heap - Code

- Use an array to store the structure of heap



Binary Heap - Code

- Father of node k → store at arr[k/2]
- Children of node k → store at arr[k * 2] and arr[k * 2 + 1]
- Root → arr[1]
- Last node → arr[n]
- Next node → arr[n + 1]



Binary Heap - Summary

	Array / Vector	Sorted Array / Vector	Binary Heap
Insert	O(1)	O(n)	O(log n)
Query min.	O(n)	O(1)	O(1)
Delete min.	O(n)	O(n)	O(log n)
Delete other	O(n)	O(n)	not support
Query other	O(n)	O(log n)	not support

Binary Heap - C++ Library

- Priority queue in C++ is implemented by binary heap
- It support all 3 operations of binary heap

heap.cpp

```cpp
#include<bits/stdc++.h>
using namespace std;

priority_queue<int> pq;

int main () {
    // insert elements to the heap
    pq.push(6);
    pq.push(2);
    pq.push(4);

    // get the max.
    printf("%d\n", pq.top());

    // erase the max.
    pq.pop();
    printf("%d\n", pq.top());
    return 0;
}
```

```
AlexPoon@DESKTOP-9F99TBM: /mnt/c/file/c++
AlexPoon@DESKTOP-9F99TBM:/mnt/c/file/c++$ make heap
g++      heap.cpp    -o heap
AlexPoon@DESKTOP-9F99TBM:/mnt/c/file/c++$ ./heap
6
4
AlexPoon@DESKTOP-9F99TBM:/mnt/c/file/c++$
```

Binary Heap - C++ Library

- Also able to declare your own structure and operator
- You may obtain a min heap or heap for string etc in this way
- priority_queue<int, vector<int>, greater<int>> is another way to implement a min heap

heap.cpp

```cpp
#include<bits/stdc++.h>
using namespace std;

// min heap can obtain my this
struct mystruct {
    int val;
    mystruct(int val) : val(val) {}
    bool operator < (mystruct const & T) const { return T.val < val; }
};

priority_queue<mystruct> minheap;

int main () {
    // insert elements to the heap
    minheap.push(6);
    minheap.push(2);
    minheap.push(4);

    // get the max.
    printf("%d\n", minheap.top().val);

    // erase the max.
    minheap.pop();
    printf("%d\n", minheap.top().val);
    return 0;
}
```

```
AlexPoon@DESKTOP-9F99TBM: /mnt/c/file/c++
AlexPoon@DESKTOP-9F99TBM:/mnt/c/file/c++$ make heap
g++      heap.cpp    -o heap
AlexPoon@DESKTOP-9F99TBM:/mnt/c/file/c++$ ./heap
2
4
AlexPoon@DESKTOP-9F99TBM:/mnt/c/file/c++$
```

Binary Heap - HKOJ 01019 Addition II

- HKOJ 01019 -- Addition II
- Given N integers
- Each time you can merge 2 integers a, b to 1 integer a + b
- The cost of merging a, b == a + b
- Find minimum cost to merge all the integers to a single integer

Binary Heap - HKOJ 01019 Addition II

- {4, 5, 7, 8}
- Optimal way is:
- Merge 4, 5 → {7, 8, 9} → cost = 0 + 9 = 9
- Merge 7, 8 → {9, 15} → cost = 9 + 15 = 24
- Merge 9, 15 → {24} → cost = 24 + 24 = 48
- Non optimal way example:
  {4, 5, 7, 8} → {4, 8, 12} → {4, 20} → {24}, cost = 12 + 20 + 24 = 56

Binary Heap - HKOJ 01019 Addition II

- Solution: Merge the smallest two integers every time
- Step by step:
  - Find the smallest integer and remove it
  - Find the 2nd smallest integer (smallest in the current container) and remove it
  - Insert the sum of the two integers to the container
- What container help you to do the above operations efficiently?
  - Heap

Binary Heap - Last words

- Many tasks in competitive programming required a heap to solve it

- You may not need to remember how to CODE binary heap
- At least you need to know how to USE priority queue in C++

Agenda

- Introduction to Data Structure
- Binary Heap
- Disjoint Set Union-find (DSU)
  - Importance to learn how to code: *****
  - Importance to learn how to use: *****
- Common Tricks in Data Structure

DSU - Introduction

Tracking elements partitioned into a number of disjoint subsets

- One element belongs to exactly one group
- One group may consists of any number of elements

Operations

- Merge two groups together
  - The elements from the two groups are now belonging to the same group
- Find which group does an element belong to
  - So we can also find if some elements belong to the same group

DSU - Naive Implementation

Maintain an array where **p[i]** represents the group id of element **i**

- Find Operation: **find(u)**
  - The answer is simply **p[u]**
- Union Operation: **union(u, v)**
  - Find all elements belong to the group **p[v]**, update them to **p[u]**

```
for (int i = 0; i < n; i++)
    if (p[i] == p[v])
        p[i] = p[u];
```

Worst Case	**union(u, v)**	**find(u)**
1 Operation	O(N)	O(1)
Q Operations	O(NQ)	O(Q)

DSU - Another Implementation

Using tree structure to maintain the groups, group id is represented by the root of each tree

Using an array where **p[i]** represents the parent of element **i**

- Find Operation: **find(u)**
  - The answer is **p[p[p[p[p[...u]]]]]**

```
int find(int u) {
    return p[u] == u ? u : find(p[u]);
}
```

DSU - Another Implementation

Using tree structure to maintain the groups, group id is represented by the root of each tree

Using an array where **p[i]** represents the parent of element **i**

- Union Operation: **union(u, v)**
  - We can simply set the root of u as root of v

```
void union(int u, int v) {
    p[find(u)] = find(v);
}
```

Worst Case	**union(u, v)**	**find(u)**
1 Operation	O(1)	O(N)
Q Operations	O(Q)	O(NQ)

DSU - Optimizations

There are two well-known optimizations on DSU

- Path Compression
  - Optimizing **find(u)** operation
  - **find(u)** operation will have amortized O(logN) time complexity
- Union by Size
  - Optimizing **union(u, v)** operation
  - **union(u, v)** operation will have amortized O(logN) time complexity
- Using them together will have amortized O(α(N)) time complexity

  α(N) is the inverse Ackermann function, α(N) < 4 for N < $2^{2^{65536}} - 3$

DSU - Path Compression

During finding root of element u, also update the parents of visited elements

```
int find(int u) {
    vector<int> visited;
    while (u != p[u]) { // non-root
        visited.push_back(u);
        u = p[u];
    }
    for (int elem : visited)
        p[elem] = u;
    return u;
}
```

DSU - Path Compression

Simpler implementation:

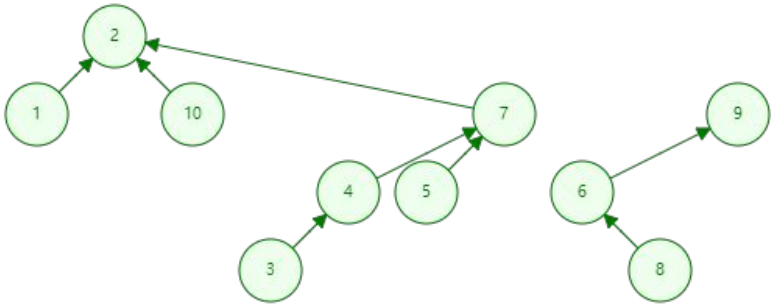
```
int find(int u) {
    if (p[u] == u) return u;
    return p[u] = find(p[u]);
}
```

Simplest implementation:

```
int find(int u) {
    return p[u] == u ? u : p[u] = find(p[u]);
}
```

DSU - Path Compression

Example: union(3, 8)

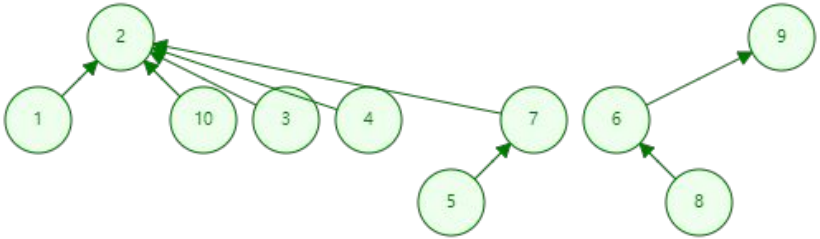


	1	2	3	4	5	6	7	8	9	10
p	2	2	4	7	7	9	2	6	9	2

DSU - Path Compression

Example: union(3, 8)

find(3) → 2



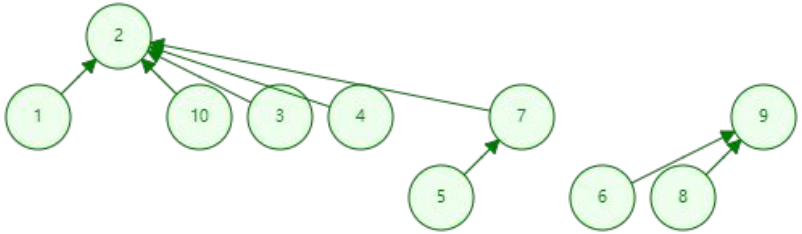
	1	2	3	4	5	6	7	8	9	10
p	2	2	2	2	7	9	2	6	9	2

DSU - Path Compression

Example: union(3, 8)

find(3) → 2

find(8) → 9



	1	2	3	4	5	6	7	8	9	10
p	2	2	2	2	7	9	2	9	9	2

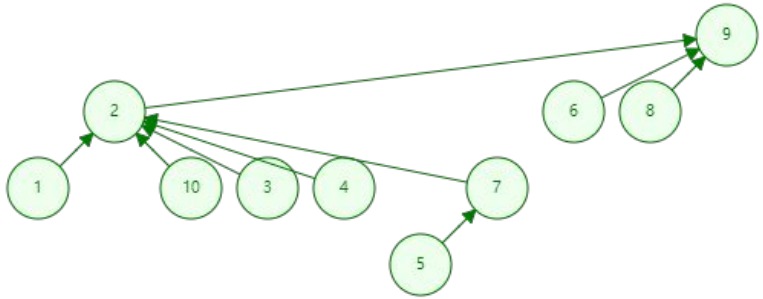
DSU - Path Compression

Example: union(3, 8)

find(3) → 2

find(8) → 9

p[2] = 9



	1	2	3	4	5	6	7	8	9	10
p	2	9	2	2	7	9	2	9	9	2

DSU - Union by Size

We want to make the tree more balance

- To reduce number of steps while calling **find(u)**

Link the subtree with smaller size to that with larger size

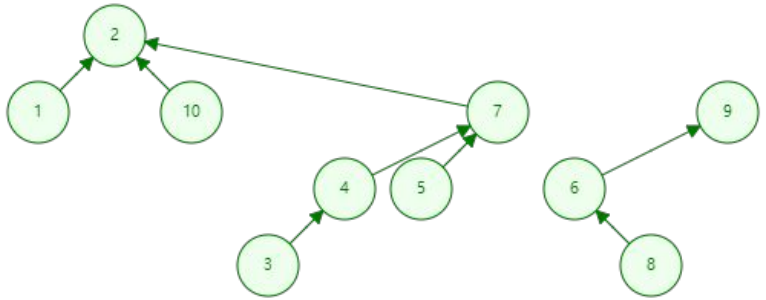
```
void union(int u, int v) {
    int rootu = find(u), rootv = find(v);
    if (rootu == rootv) return;
    if (subtree_size[rootu] < subtree_size[rootv]) {
        p[rootu] = root[v];
    }
    ...
}
```

DSU - Union by Size

```
void union(int u, int v) {
    int rootu = find(u), rootv = find(v);
    if (rootu == rootv) return;
    if (subtree_size[rootu] < subtree_size[rootv]) {
        p[rootu] = rootv;
        subtree_size[rootv] += subtree_size[rootu];
    } else {
        p[rootv] = rootu;
        subtree_size[rootu] += subtree_size[rootv];
    }
}
```

DSU - Union by Size

Example: union(3, 8)



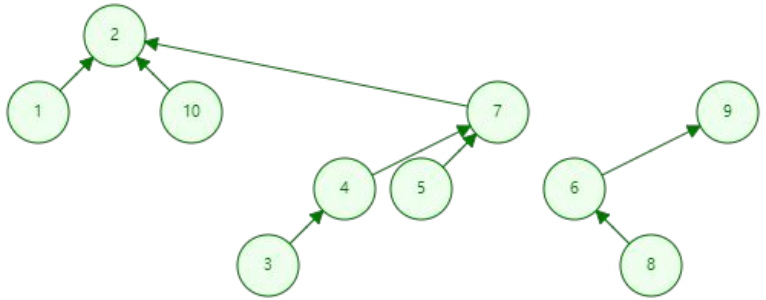
	1	2	3	4	5	6	7	8	9	10
p	2	2	4	7	7	9	2	6	9	2
size	1	7	1	2	1	2	4	1	3	1

DSU - Union by Size

Example: union(3, 8)

find(3) → 2

find(8) → 9



	1	2	3	4	5	6	7	8	9	10
p	2	2	4	7	7	9	2	6	9	2
size	1	7	1	2	1	2	4	1	3	1

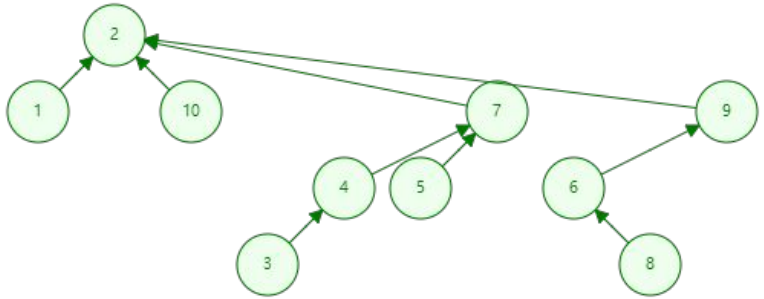
DSU - Union by Size

Example: union(3, 8)

find(3) → 2

find(8) → 9

size(2) > size(9) → p[9] = 2



	1	2	3	4	5	6	7	8	9	10
p	2	2	4	7	7	9	2	6	2	2
size	1	10	1	2	1	2	4	1	3	1

DSU - NOI 2015 Day1 Q1 程序自動分析

- HKOJ N1511 -- 程序自動分析
- Given N mathematical constraints, in the form of
  - `Ai = Aj`
  - `Ai != Aj`
- Determine if all N constraints can be satisfied

- Note that this question is required to use Discretization technique, it will be taught in HKOI training - Optimization and Common Tricks lesson

DSU - NOI 2015 Day1 Q1 程序自動分析

- Solution: Merge variable that must be equal in accordance with the (Ai = Aj) constraint, check if the (Ai != Aj) constraint can be satisfied
- Step by step:
  - For all the (Ai = Aj) constraints, union Ai and Aj
  - For all the (Ai != Aj) constraints, if Ai and Aj have the same root, output “NO”
  - Else output “YES”
- What container help you to do the above operations efficiently?
  - DSU

DSU - Summary

DSU is an useful data structure!

Easy to implement, yet hard to understand why it works (so fast)

Under usual practice, using only path compression itself is fast enough :)

There are more types of implementation (e.g. union by rank): see Wikipedia

Explanation to path compression time complexity: see StackExchange

Agenda

- Introduction to Data Structure
- Binary Heap
- Disjoint Set Union-find (DSU)
- Common tricks in Data structure
  - Lazy propagation
  - Constant K-th element

Data Structure in Competitive Programming

- In Competitive Programming, we seldom really code heap or BST as it is time-consuming
- In most of the cases, we use library
- Sometimes, we may use library Data Structure smartly to perform some operations which is not supported originally to avoid coding the real Data Structure

Heap to support delete - Lazy Propagation

- Let's say we encounter the following problem
  - Insert a number
  - Query Min
  - Remove any number
- For sure we can use BST to solve it, but two heaps also works
- Btw, why heap when we have BST?
  - Heaps is more efficient (in terms of constant)

Lazy Propagation

- We may use two heaps to solve it
- Insert a number → push to heap A
- Erase a number which is minimum → remove in heap A
- Erase a number which is NOT minimum → push in heap B →

  and remove it in heap A later when it becomes the minimum in heap A

Lazy Propagation

Add 5
Add 2
Add 3
Query Min
Remove 3
Add 1
Remove 2
Add 3
Remove 1
Query Min

Heap A: {2, 3, 5}

Heap B: {}

Lazy Propagation

Add 5
Add 2
Add 3
Query Min → 2
Remove 3
Add 1
Remove 2
Add 3
Remove 1
Query Min

Heap A: {2, 3, 5}

Heap B: {}

Lazy Propagation

Add 5
Add 2
Add 3
Query Min
Remove 3 → Not minimum in A, add in B
Add 1
Remove 2
Add 3
Remove 1
Query Min

Heap A: {2, 3, 5}

Heap B: {3}

Lazy Propagation

Add 5
Add 2
Add 3
Query Min
Remove 3
Add 1
Remove 2
Add 3
Remove 1
Query Min

Heap A: {1, 2, 3, 5}

Heap B: {3}

Lazy Propagation

Add 5
Add 2
Add 3
Query Min
Remove 3
Add 1
Remove 2 → Not minimum in A, add in B
Add 3
Remove 1
Query Min

Heap A: {1, 2, 3, 5}

Heap B: {2, 3}

Lazy Propagation

Add 5
Add 2
Add 3
Query Min
Remove 3
Add 1
Remove 2
Add 3
Remove 1
Query Min

Heap A: {1, 2, 3, 3, 5}

Heap B: {2, 3}

Lazy Propagation

Add 5
Add 2
Add 3
Query Min
Remove 3
Add 1
Remove 2
Add 3
Remove 1 → equal to minimum in A
Query Min

Heap A: {2, 3, 3, 5}

Heap B: {2, 3}

Lazy Propagation

Add 5
Add 2
Add 3
Query Min
Remove 3
Add 1
Remove 2
Add 3
Remove 1 → now heap A, B has same min.
Query Min

Same min in heap A, B

Heap A: {2, 3, 3, 5}

Heap B: {2, 3}

→ erase 2 in both heap

Heap A: {3, 3, 5}

Heap B: {3}

Again, same min in heap A, B → Erase

Heap A: {3, 5}

Heap B: {}

Lazy Propagation

Add 5
Add 2
Add 3
Query Min
Remove 3
Add 1
Remove 2
Add 3
Remove 1
Query Min → answer = 3

Heap A: {3, 5}

Heap B: {}

Lazy Propagation

- Why works? → Erasing larger elements does not affect the query result
- So we can delete it later → just before it affects the query result then OK
- This is a common technique in CP → lazy propagation
  - Just label the to-be-deleted or to-be-updated element but not really update it
  - Update it just before it will affect the query result

Constant K-th element

- Problem:
  - Insert an element
  - Delete an element
  - Find the k-th (where k is constant) smallest element

Constant K-th element

- Problem:
  - Insert an element
  - Delete an element
  - Find the k-th (where k is constant) smallest element
- You may solve it by coding your own BST
- But you can also solve it by two C++-library BST (map or set) OR four heaps (priority_queue)

Constant K-th element

- Use one max-heap (heap1) and one min-heap (heap2)
  - Max-heap: able to find max / insert / delete max
  - Min-heap: able to find min / insert / delete min
  - Use lazy propagation to support deleting any element (requires 2 more heaps)

Constant K-th element

- The max-heap stores the smallest K elements at any time
  - (or all elements if current # of element < K)

- The min-heap stores the remaining elements
  - (all elements in the min-heap should >= any element in the max-heap)

- Answer is always the maximum element in the max-heap

Constant K-th element

K = 2
Insert 1
Insert 2
Insert 3
Query
Insert 4
Erase 3
Erase 2
Query
Insert 3
Query

heap 1: {1, 2}

heap 2: {}

Constant K-th element

K = 2
Insert 1
Insert 2
Insert 3 → heap 1 contains K elements now
as 3 >= largest element in heap 1
Push in heap 2
Query
Insert 4
Erase 3
Erase 2
Query
Insert 3
Query

heap 1: {1, 2}

heap 2: {3}

Constant K-th element

K = 2
Insert 1
Insert 2
Insert 3
Query → answer = largest in heap 1 → 2
Insert 4
Erase 3
Erase 2
Query
Insert 3
Query

heap 1: {1, 2}

heap 2: {3}

Constant K-th element

K = 2
Insert 1
Insert 2
Insert 3
Query
Insert 4
Erase 3
Erase 2
Query
Insert 3
Query

heap 1: {1, 2}

heap 2: {3, 4}

Constant K-th element

K = 2
Insert 1
Insert 2
Insert 3
Query
Insert 4
Erase 3 → 3 is in heap 2 → erase 3 in heap 2
Erase 2
Query
Insert 3
Query

heap 1: {1, 2}

heap 2: {4}

Constant K-th element

K = 2

Insert 1

Insert 2

Insert 3

Query

Insert 4

Erase 3

Erase 2 → 2 is in heap 1 → erase 2 in heap 1
→ after that heap 1 contains only 1 element
→ push the smallest element in heap 2 such that heap 1 contains K elements

...

heap 1: {1, 4}

heap 2: {}

Constant K-th element

K = 2
Insert 1
Insert 2
Insert 3
Query
Insert 4
Erase 3
Erase 2
Query → res = maximum in heap 1 = 4
Insert 3
Query

heap 1: {1, 4}

heap 2: {}

Constant K-th element

K = 2

Insert 1

Insert 2

Insert 3

Query

Insert 4

Erase 3

Erase 2

Query

Insert 3 → heap 1 has contains K elements,

→ 3 is smaller than the maximum in heap 1

→ push 3 in heap 1 and re-push 4 in heap 2

Query

heap 1: {1, 3}

heap 2: {4}

Constant K-th element

K = 2
Insert 1
Insert 2
Insert 3
Query
Insert 4
Erase 3
Erase 2
Query
Insert 3
Query → res = 3

heap 1: {1, 3}

heap 2: {4}

Constant K-th element

- Time complexity
  - Insertion / deletion: O(log(n)) each
  - Query: O(1)
  - Number of re-push needed to ensure heap1 contains the smallest K-th elements anytime:
    - Case 1: Use lazy propagation to erase 1 integer from heap1 → re-push the smallest in heap2 to heap1
    - Case 2: Use lazy propagation to erase 1 integer from heap2 → no re-push needed
    - Case 3: Just insert 1 integer to heap1 → re-push the largest in heap1 to heap2
    - Case 4: Just insert 1 integer to heap2 → no re-push needed

Constant K-th element

- Time complexity
  - Insertion / deletion: O(log(n)) each
  - Query: O(1)
  - Number of re-push needed to ensure heap1 contains the smallest K-th elements anytime:
    - In any case, only O(1) operation is needed
    - O(1) per operation → total O(n)
- Total complexity: O(n log n)

Constant K-th element

- Variance of constant K-th element
  - Find constant K-percentile of elements (e.g. find median)
  - Non-constant K-th element but K is monotonic (increasing / decreasing)
- Anyway, 2 heaps to store data is another common trick about data structure

Summary

- More important is to learn how to USE data structure but not CODE
- Learn C++ priority_queue / set / map → Attend Advanced C++ STL lesson
- Practice problem:
  - M0811 Alice's Bookshelf → (2 heaps with lazy propagation or balanced BST)
  - 01019 Addition II → (heaps)
  - N1511 程序自動分析 → (DSU)
  - IOI 2012 Practice Q3 Touristic plan → (Constant K-th element trick)
    http://www.ioi2012.org/wp-content/uploads/2011/12/practice.pdf

Additional Readings

- Data Structures (II) slides (2021)

  Introduction to binary search tree and hash table

Additional Readings

- Data Structures (II) slides (2018)

  Binary Heap:

    How to build binary heap in linear time

    Heuristic Merging (Practice: AP121 Dispatching)

  Binary Search Tree:

    Attaching values to node for lazy deletion

  Hash Table:

    Open Addressing & Separate Chaining

    Determining the performance by Load factor