Data Structures (II)

Cheung Cheuk Nam {JosephCCN}

2024-02-24

Agenda

- Binary Heap
- Binary Search Tree
- Hash Table
- Disjoint-set union-find (DSU)

Problem List (If you already attended this lesson last year)

- 01019 - Addition II
- M0811 - Alice's Bookshelf
- 01090 - Diligent
- N1511 - 程序自動分析
- IOI 2012 Practice Q3 - Touristic plan

- AP121 - Dispatching
- M1811 - Almost Constant
- M1533 - Bridge Routing
- M2214 - Fluctuating Market

Hard Problems (can be done in oj.uz):

- BalticOI 2016 D1Q2 - Park
- BalticOI 2018 D2Q2 - Genetics
- JOI Spring Camp 2020 D2Q2 - Making Friends on Joitter is Fun

Data Structure

A data structure is a data organization, management, and storage format that is usually chosen for efficient access to data

Common operations

- Insertion
  - Insert 1 or more data
- Deletion
  - Delete 1 or more data
- Modification
  - Modify the value of an existing data
- Query
  - Check if an element exists
  - Find min/max
  - ...

Question

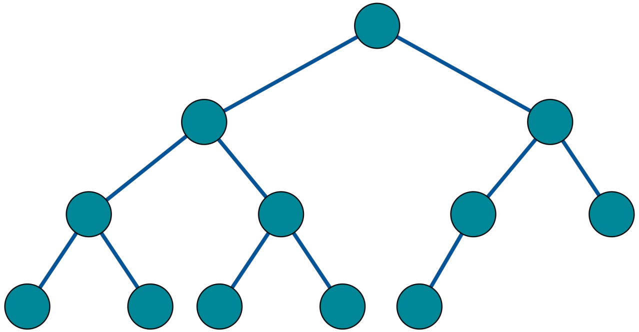
- There are 3 types of operations, with total of **N** operations
  - Delete **X** from the set
  - Insert **X** to the set
  - Find the Min. number of the set

- Find a solution with time complexity O(NlogN)

Binary Heap

Binary Heap

- Binary Heap often used to search for the min/max from a set online
- Binary Heap is a complete binary tree
- A complete binary tree is a perfect binary tree with some rightmost node removed on the last level
- Height of binary heap is ⌊log N⌋



Binary Heap

- Heap supports query and deletion of min/max element
- Operations supported:
  - Insert → O(log N)
  - Delete min → O(log N)
  - Query min → O(1)
  - Delete, Query, Update any number → Not supported
- C++ STL version of heap: priority_queue

- What is the difference between priority_queue and heap?

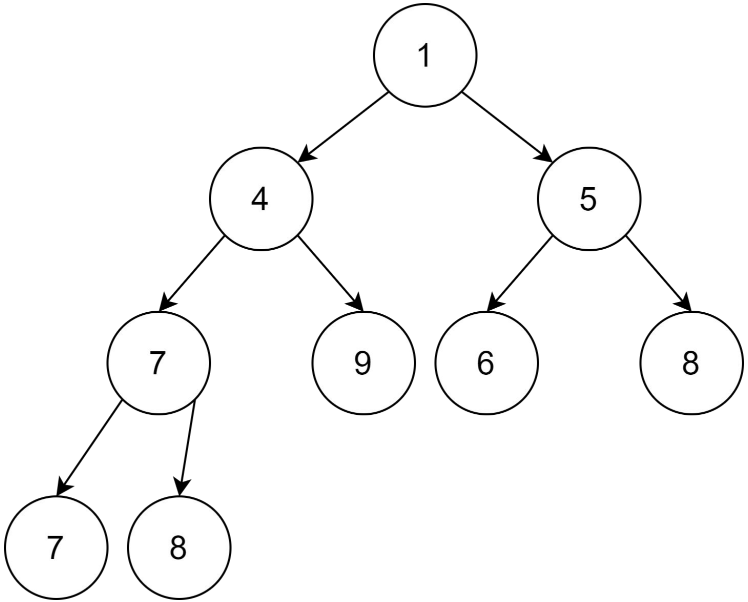
Binary Heap

In a min heap, each element is not less than its parent element

- Key idea:
  - The minimum element is stored in the root
  - Maintain this property during insertion and deletion

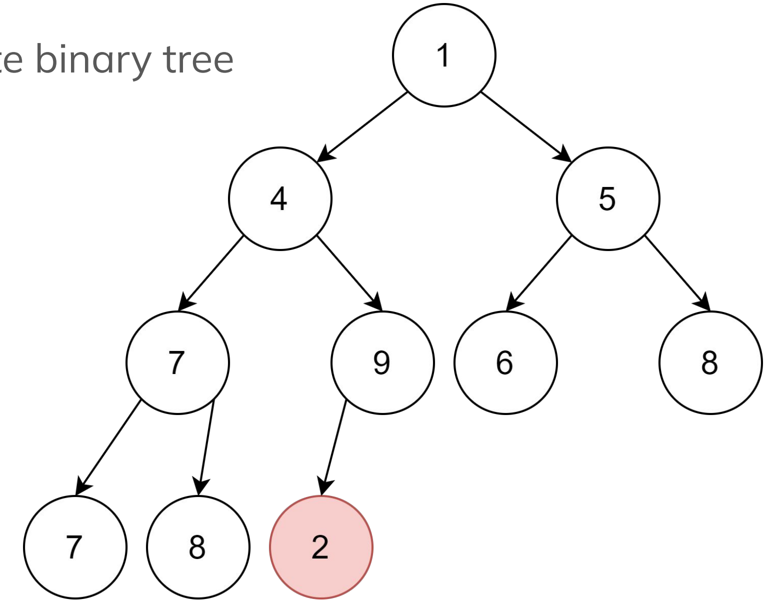
Binary Heap - insertion

- Insert 2 to the existing heap



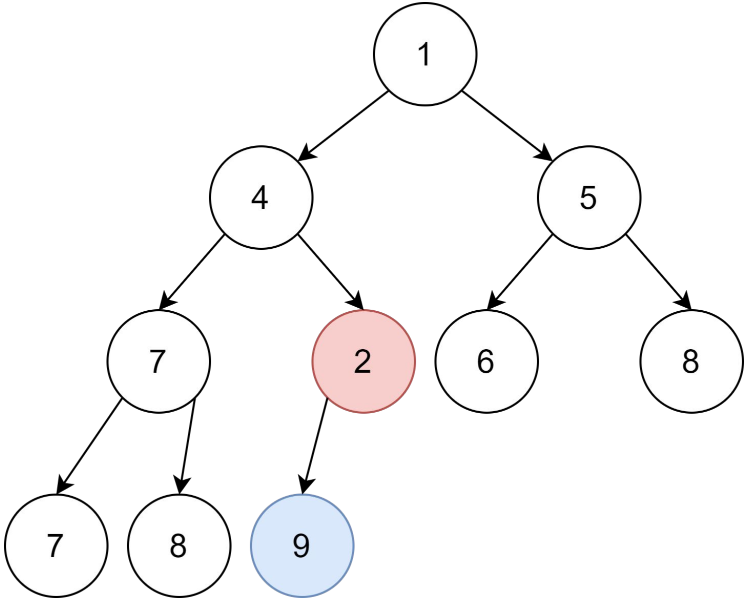
Binary Heap - insertion

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



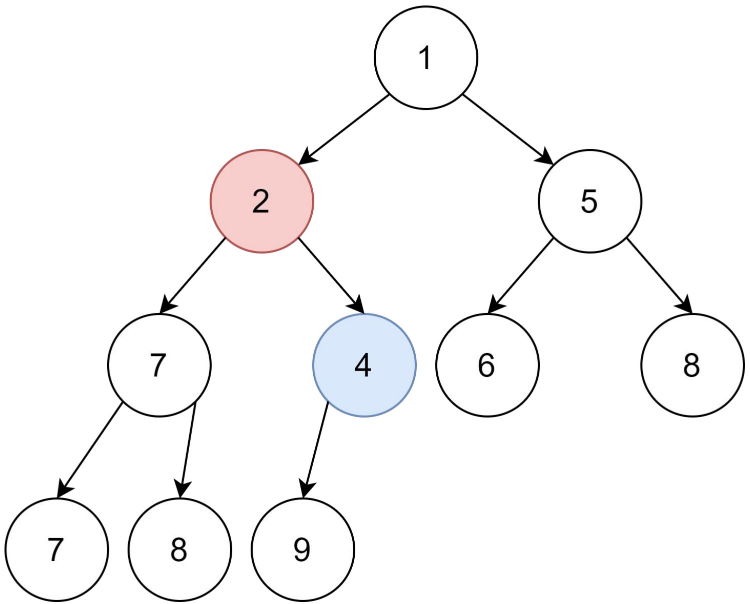
Binary Heap - insertion

- Insert 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 the heap, such that each element >= parent
  - **2 < 9, swap**



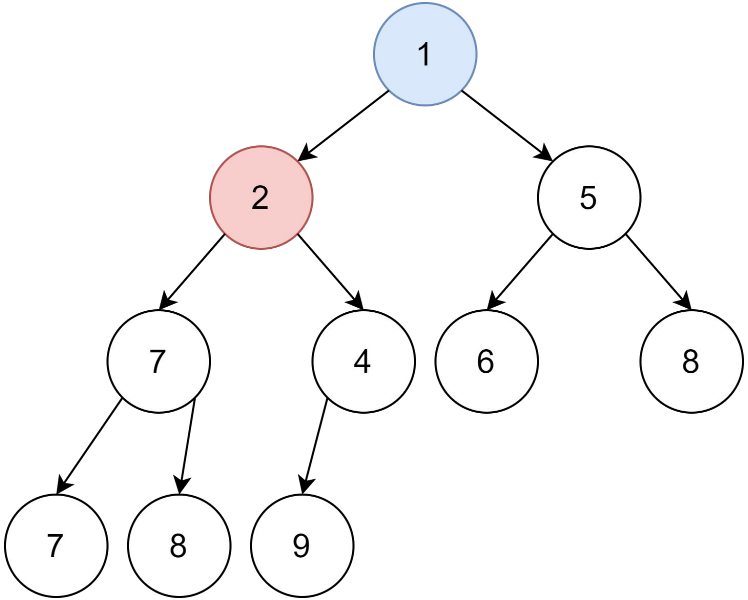
Binary Heap - insertion

- Insert 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 the heap, such that each element >= parent
  - **2 < 9, swap**
  - **2 < 4, swap**



Binary Heap - insertion

- Insert 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 the heap, such that each element >= parent
  - **2 < 9, swap**
  - **2 < 4, swap**
  - **2 >= 1, done**



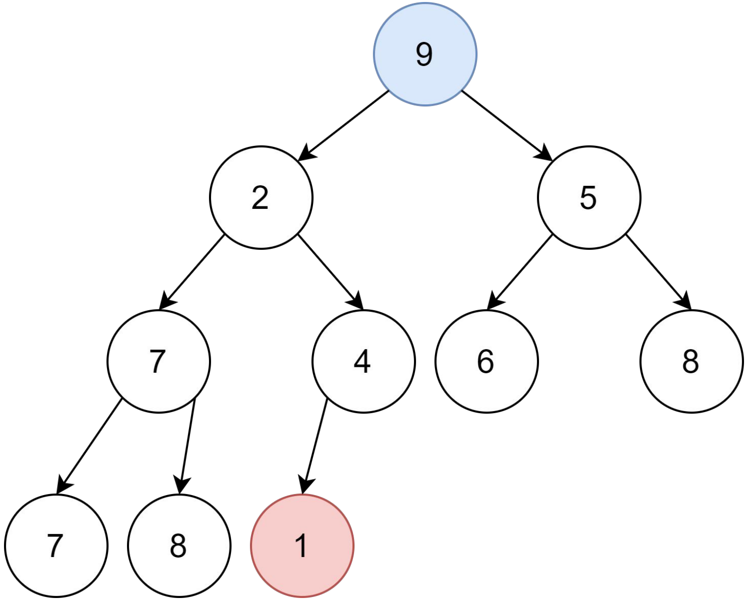
Binary Heap - insertion

Time Complexity:

- Number of lift
- Height of the heap
- O(logN)

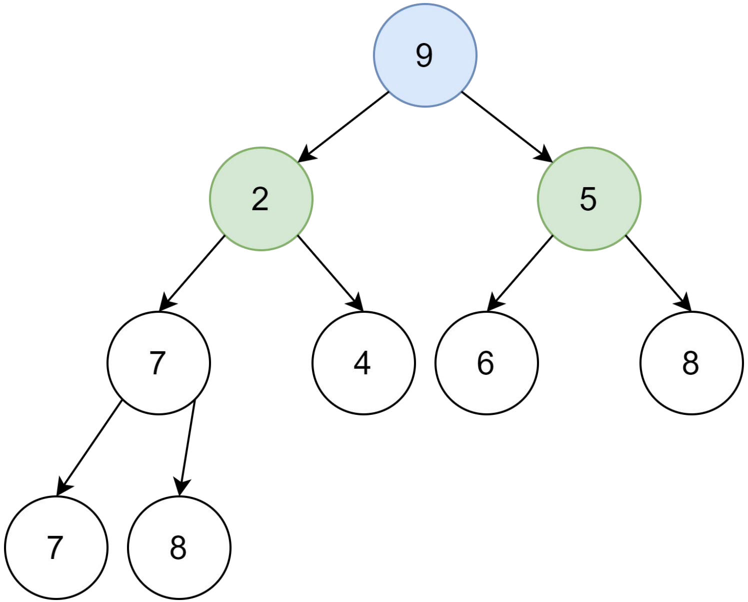
Binary Heap - delete minimum

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



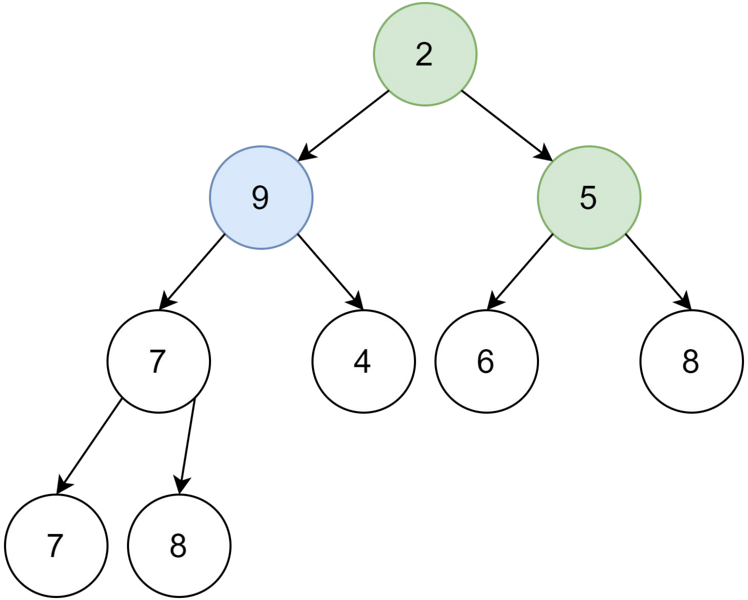
Binary Heap - delete minimum

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



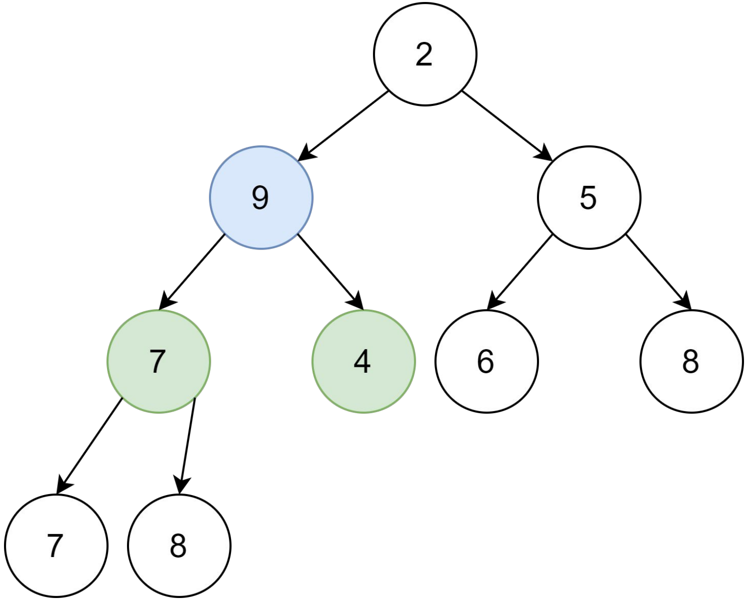
Binary Heap - delete minimum

- Step 1: Swap the minimum (root) and the last node
- Step 2: Delete the last node
- Step 3: Shift-down the root to maintain the property of heap the heap, such that each element >= parent
  - Swap with the smaller children
  - **Swap 9 with 2**



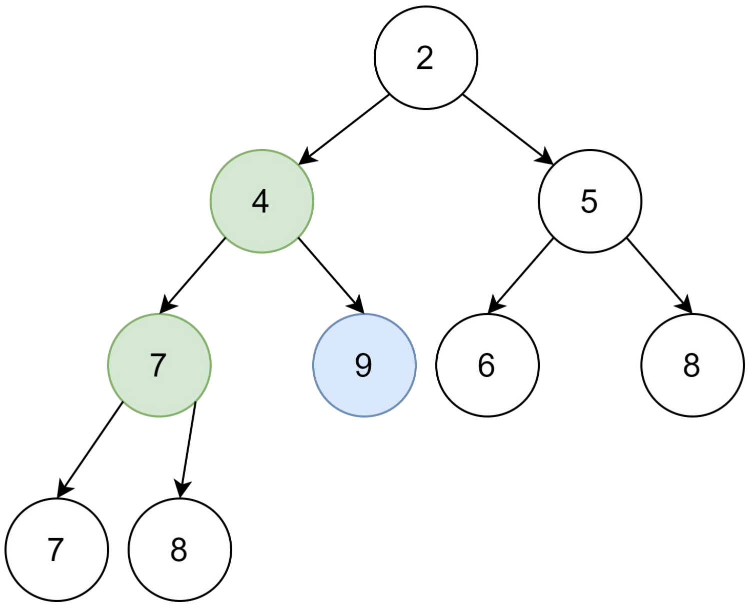
Binary Heap - delete minimum

- Step 1: Swap the minimum (root) and the last node
- Step 2: Delete the last node
- Step 3: Shift-down the root to maintain the property of heap the heap, such that each element >= parent
  - Swap with the smaller children
  - **Consider 9 and its children (7, 4)**



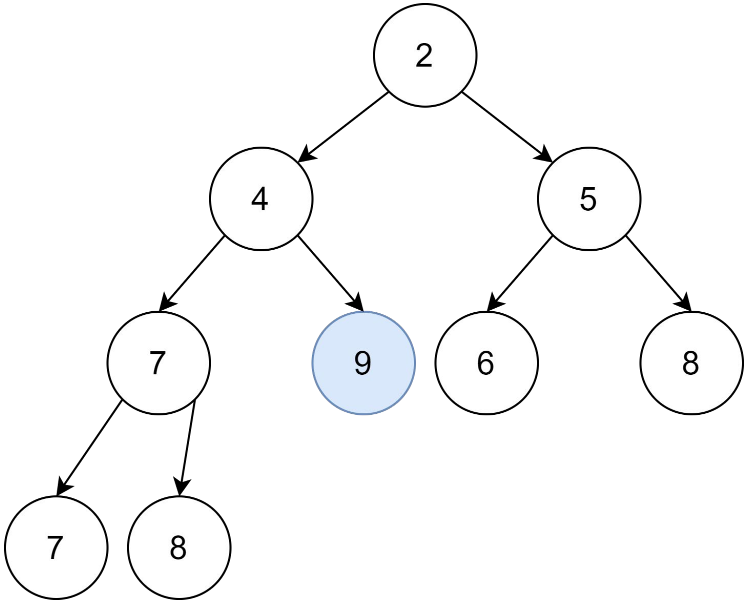
Binary Heap - delete minimum

- Step 1: Swap the minimum (root) and the last node
- Step 2: Delete the last node
- Step 3: Shift-down the root to maintain the property of heap the heap, such that each element >= parent
  - Swap with the smaller children
  - **Swap 9 with 4**



Binary Heap - delete minimum

- Step 1: Swap the minimum (root) and the last node
- Step 2: Delete the last node
- Step 3: Shift-down the root to maintain the property of heap the heap, such that each element >= parent
  - Swap with the smaller children
  - **Consider 9 and its children, no children → done**



Binary Heap - delete minimum

Time Complexity: O(log N)

Same reason for insertion

Binary Heap - query

The min. element always at the root node of the heap

Time complexity: O(1)

Binary Heap - Implementation

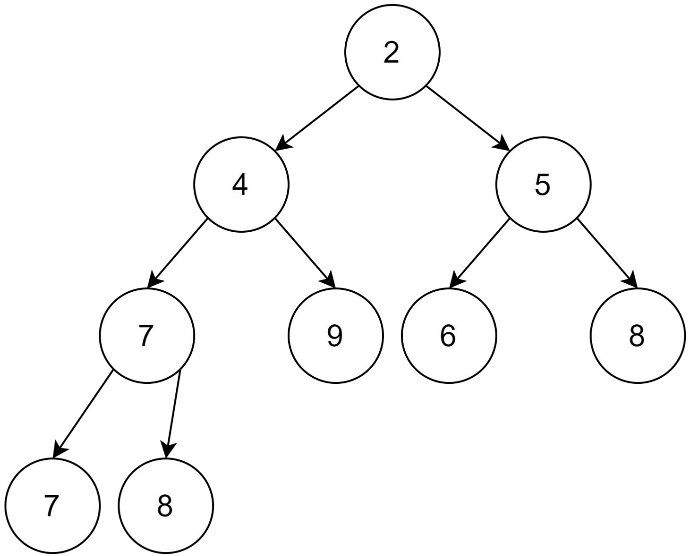
- We still need a way to implement the heap
- Ofc we can construct a class/struct, and implement using pointers

- Annoying to implement, time consuming

Binary Heap - Array Implementation

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

Array representation

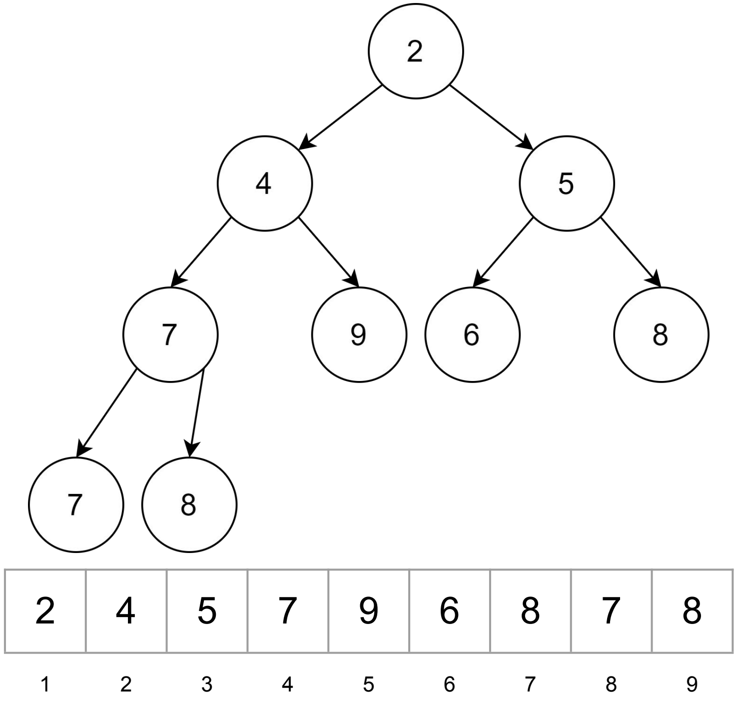


Tree representation

Binary Heap - Array Implementation

In a 1-indexed array:

- Root of heap → arr[1]
- Parent of node arr[k] → arr[(k - 1) / 2]
- Left children of node arr[k] → arr[k * 2]
- Right children of node arr[k] → arr[k * 2 + 1]
- Last node in the heap → arr[N]
- Next node inserted → arr[N + 1]



Binary Heap - C++ Library

- priority_queue supports all 3 operation
- Default max heap

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
  priority_queue<int> pq;
  pq.push(1); // insert
  pq.push(2);
  pq.push(3);
  cout << "Size = " << pq.size() << endl;
  // get max
  cout << "Max = " << pq.top() << endl;

  pq.pop(); // delete max
  cout << "New max = " << pq.top() << endl;
}
```

```
Output:
Size = 3
Max = 3
New max = 2
```

Binary Heap - C++ Library

To declare a min heap:

- Declare `priority_queue<type, container type, compare parameter>` and set the compare parameter instead of `std::less()`
- `priority_queue<int, vector<int>, greater<int>>`
- Declare your own structure and overload the < operator

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
  priority_queue<int, vector<int>, greater<int>>
pq;
  pq.push(1); // insert
  pq.push(2);
  pq.push(3);
  cout << "Size = " << pq.size() << endl;
  // get max
  cout << "Max = " << pq.top() << endl;

  pq.pop(); // delete max
  cout << "New max = " << pq.top() << endl;
}
```

```
Output:
Size = 3
Max = 1
New max = 2
```

Binary Heap - C++ Library

To declare a min heap:

- Declare `priority_queue<type, container type, compare parameter>` and set the compare parameter instead of `std::less()`
- `priority_queue<int, vector<int>, greater<int>>`
- Declare your own structure and overload the < operator

```
#include <bits/stdc++.h>
using namespace std;
struct my {
  int val;
  const bool operator<(const my &e) const { return
val > e.val; }
};
int main() {
  priority_queue<my> pq;
  pq.push({1}); // insert
  pq.push({2});
  pq.push({3});
  cout << "Size = " << pq.size() << endl;
  // get max
  cout << "Max = " << pq.top().val << endl;

  pq.pop(); // delete max
  cout << "New max = " << pq.top().val << endl;
}

Output:
Size = 3
Max = 1
New max = 2
```

Binary Heap - HKOJ 01019 Addition II

- Given N integers
- In each operation, merge 2 integers a, b into an integer (a+b)
- Cost of merging = a + b
- Find the minimum cost of merging all N integers to 1 integer

Binary Heap - HKOJ 01019 Addition II

{4, 5, 7, 8}

- Optimal merge:
  - 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 merge:
  - {4, 5, 7, 8} → {4, 8, 12} → {4, 20} → {24}, cost = 12 + 20 + 24 = 56

Binary Heap - HKOJ 01019 Addition II

- Key Observation: merging the smallest two integers gives the lowest cost
- Repeat the following:
  - Find the smallest element x from container and remove it
  - Find the smallest element y from container and remove it
  - Insert x + y to the container, accumulate answer
  - Repeat above until there is only 1 integer left in the container
- Use a min heap to maintain the above!

Back to the previous question

- There are 3 types of operations, with total of **N** operations
  - Delete **X** from the set
  - Insert **X** to the set
  - Find the Min. number of the set

- Find a solution with time complexity O(NlogN)
- Tip: Maintain 2 heap

Question 2

- There are 4 types of operations, with total of **N** operations
  - Delete **X** from the set
  - Insert **X** to the set
  - Find the Min/Max number of the set

- Find a solution with time complexity O(NlogN)

Binary Search Tree (BST)

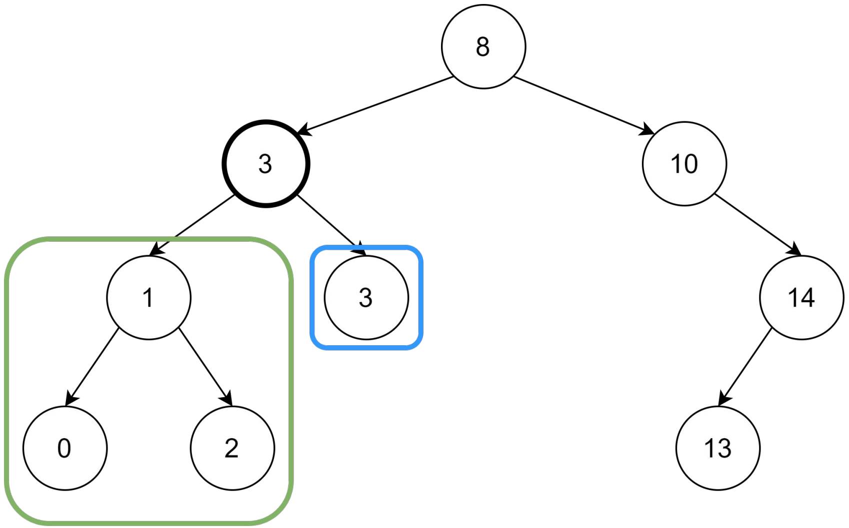
Binary Search Tree (BST)

- Insertion: O(log N)
- Deletion: O(logN)
- Query: O(logN)
- Rank any element: O(logN)

C++ STL: set, multiset, map

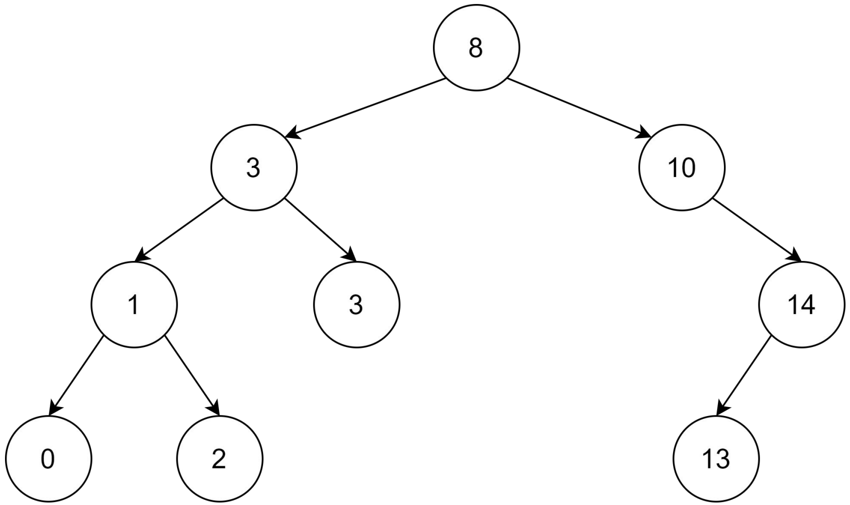
Binary Search Tree

- A directed Binary Tree (each node has 0 - 2 children)
- Each nodes store an element
- Value of all nodes in the left subtree of node k < value of node k
- Value of all nodes in the right subtree of node k >= value of node k



BST - insertion

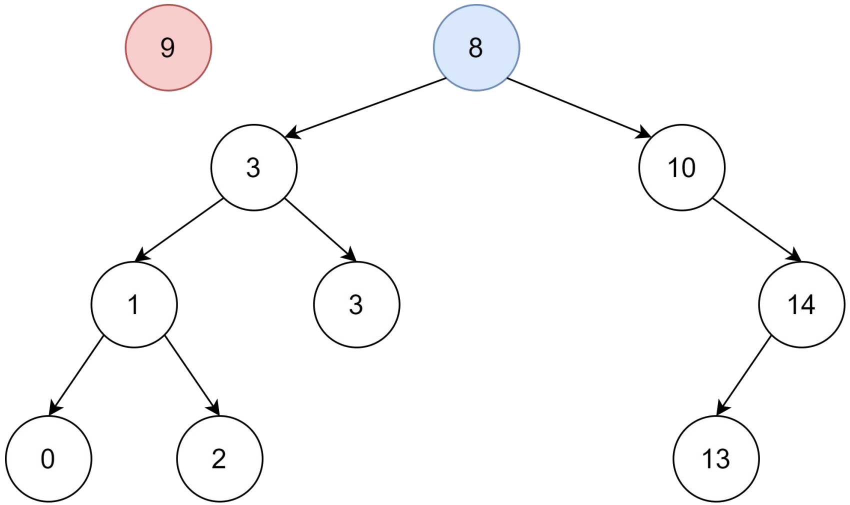
- DFS from root
- Repeatedly travel down the tree -
  If the inserted value < the current node's value → go left
  else → go right
- Until we find a empty space



BST - insertion

Insert 9 to the BST

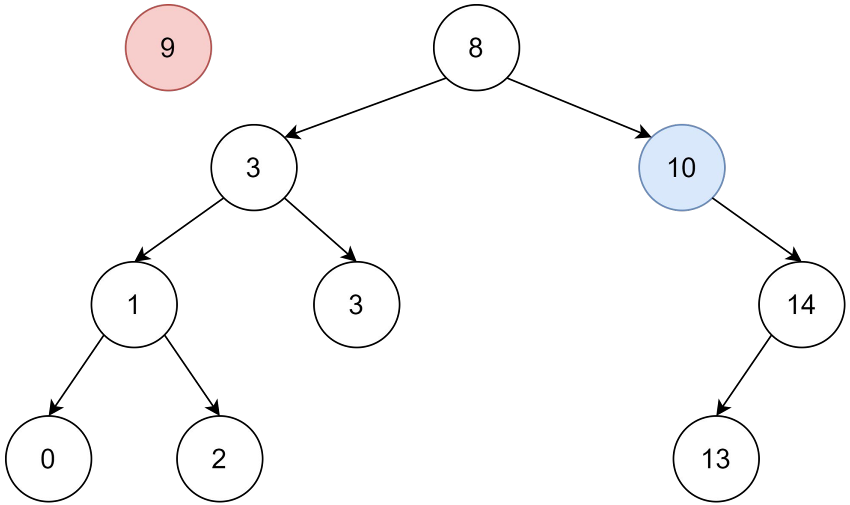
- Current node = root (8)
  9 > 8, go to right subtree
- Current node = 10
  9 < 10, go to left subtree
- Left subtree is empty - place 9 at this empty spot



BST - insertion

Insert 9 to the BST

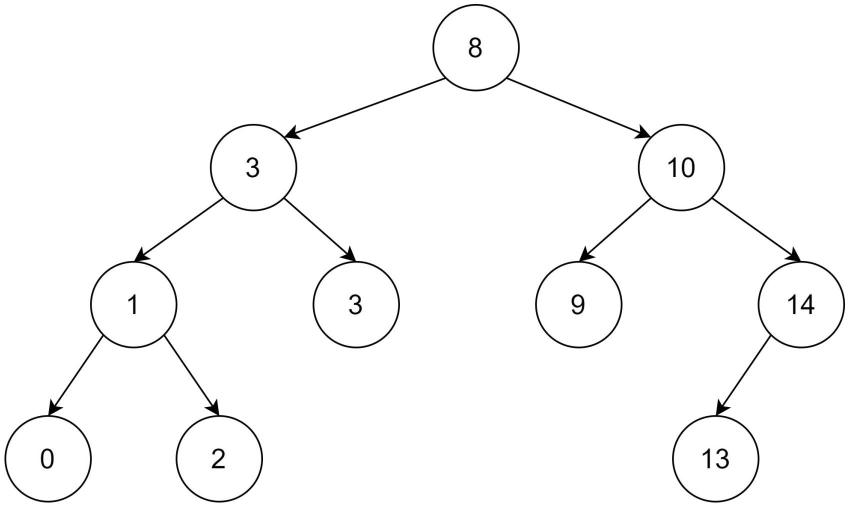
- Current node = root (8)
  9 > 8, go to right subtree
- Current node = 10
  9 < 10, go to left subtree
- Left subtree is empty - place 9 at this empty spot



BST - insertion

Insert 9 to the BST

- Current node = root (8)
  9 > 8, go to right subtree
- Current node = 10
  9 < 10, go to left subtree
- Left subtree is empty - place 9 at this empty spot

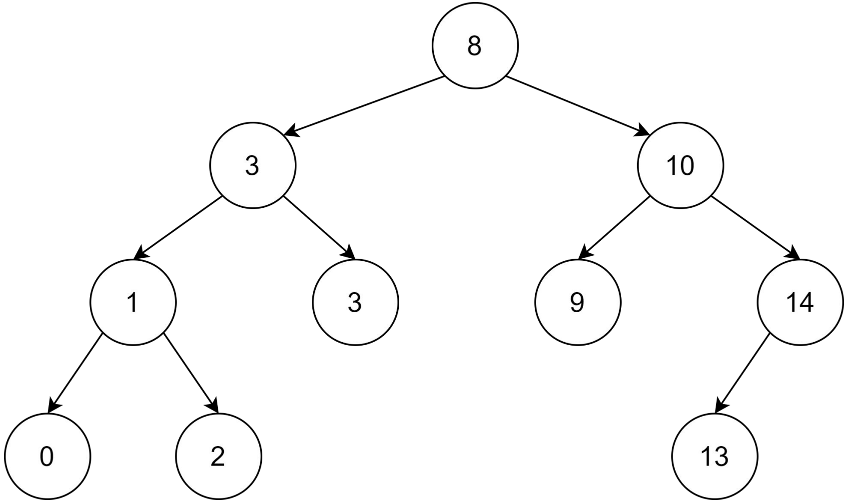


BST - query

Find if a number exists

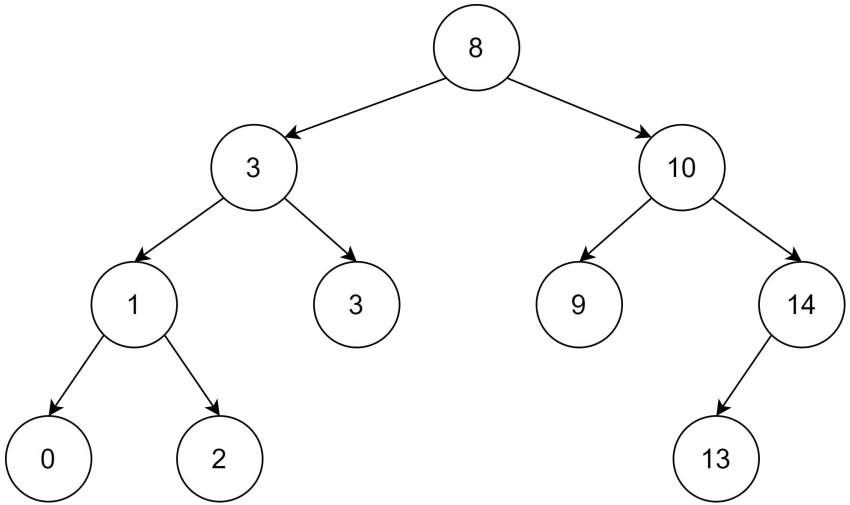
Find the lower-bound / upper-bound

Find min / max



Binary Search Tree - Query

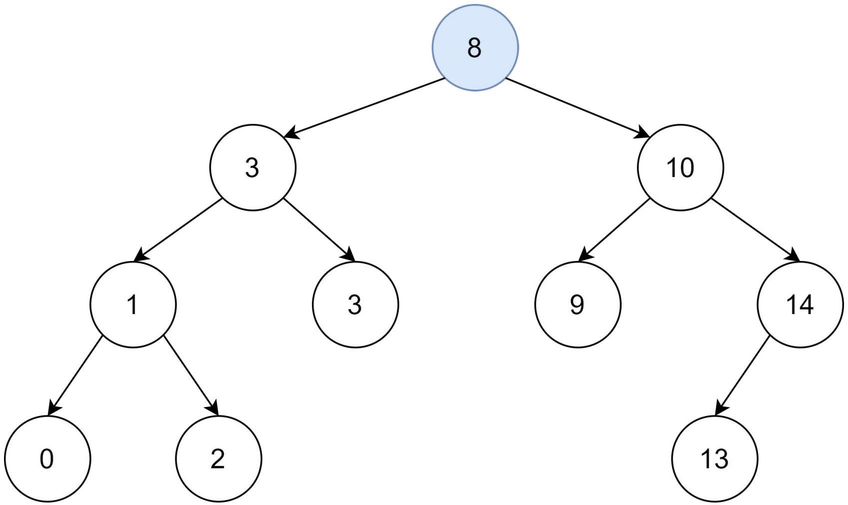
- DFS from the root
- Repeatedly travel down the tree -
  If the inserted value < the current node's value → go left
  else → go right
- Until the value is found



Binary Search Tree - Query if a value exist (1)

Find if 2 exists in a BST

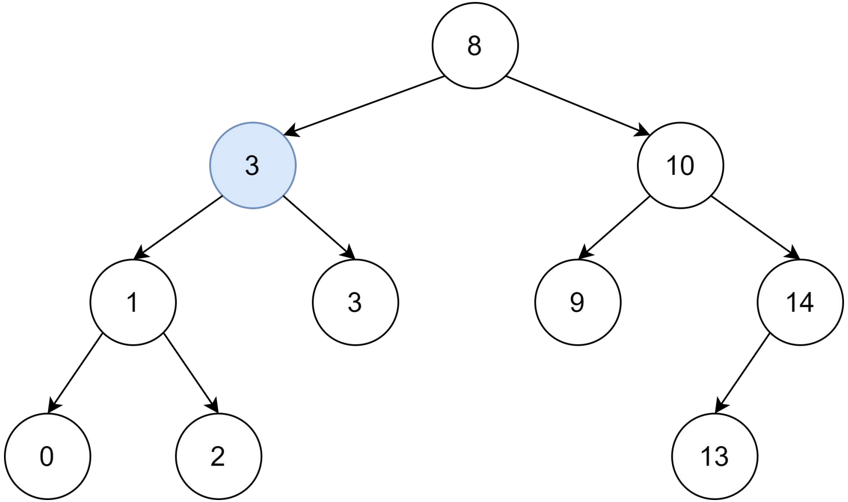
- Current node = root (8)
  2 < 8, go to left subtree
- Current node = 3
  2 < 3, go to left subtree
- Current node = 1
  2 > 1, go to right subtree
- 2 is found!



Binary Search Tree - Query if a value exist (1)

Find if 2 exists in a BST

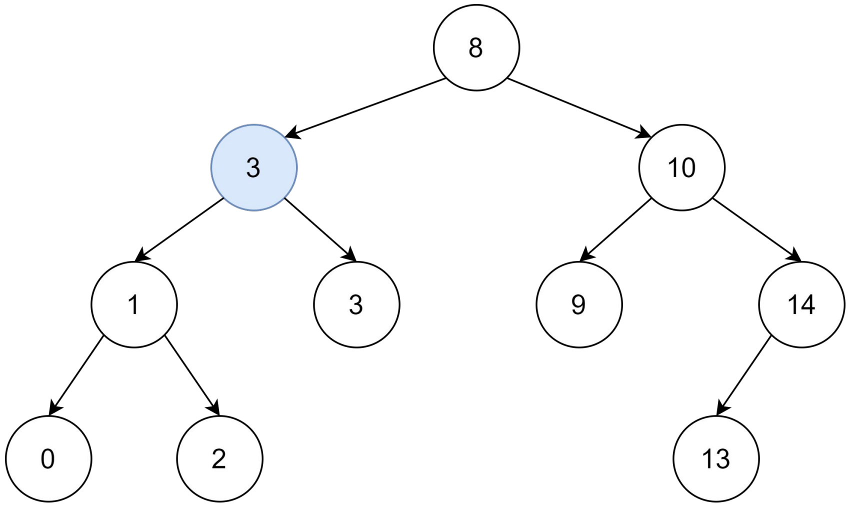
- Current node = root (8)
  2 < 8, go to left subtree
- Current node = 3
  2 < 3, go to left subtree
- Current node = 1
  2 > 1, go to right subtree
- 2 is found!



Binary Search Tree - Query if a value exist (1)

Find if 2 exists in a BST

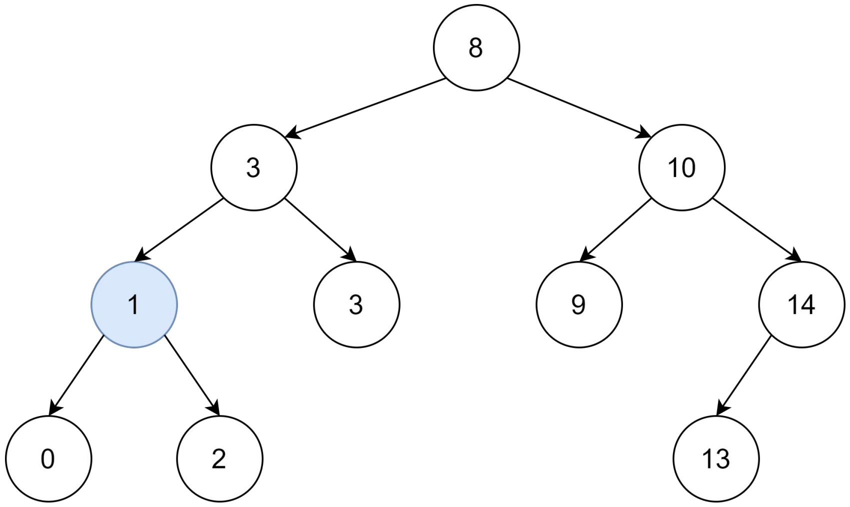
- Current node = root (8)
  2 < 8, go to left subtree
- Current node = 3
  2 < 3, go to left subtree
- Current node = 1
  2 > 1, go to right subtree
- 2 is found!



Binary Search Tree - Query if a value exist (1)

Find if 2 exists in a BST

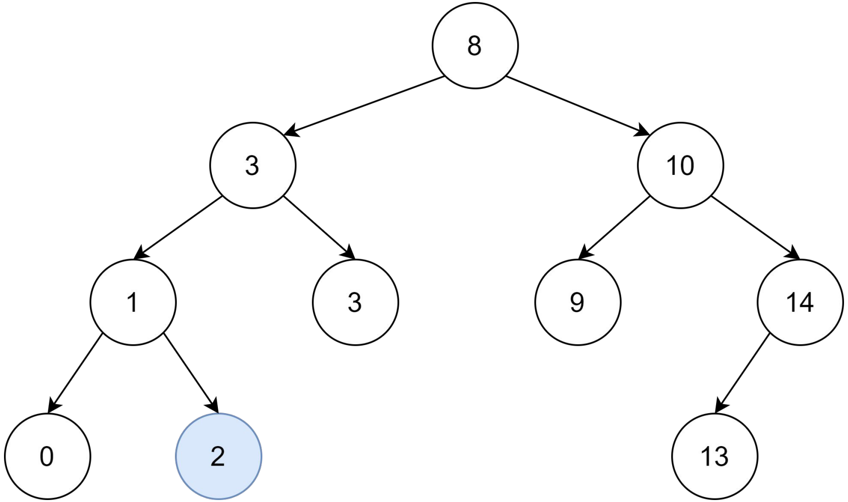
- Current node = root (8)
  2 < 8, go to left subtree
- Current node = 3
  2 < 3, go to left subtree
- Current node = 1
  2 > 1, go to right subtree
- 2 is found!



Binary Search Tree - Query if a value exist (1)

Find if 2 exists in a BST

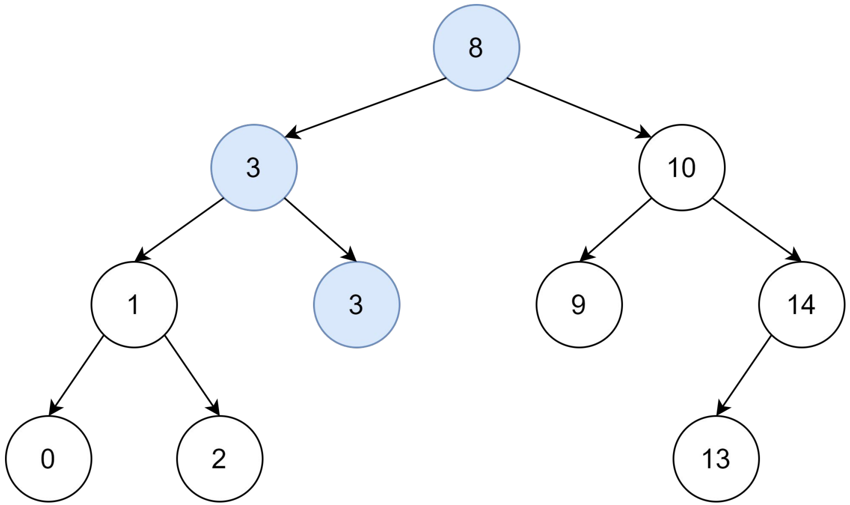
- Current node = root (8)
  2 < 8, go to left subtree
- Current node = 3
  2 < 3, go to left subtree
- Current node = 1
  2 > 1, go to right subtree
- 2 is found!



Binary Search Tree - Query if a value exist (2)

Find if 5 exists in a BST

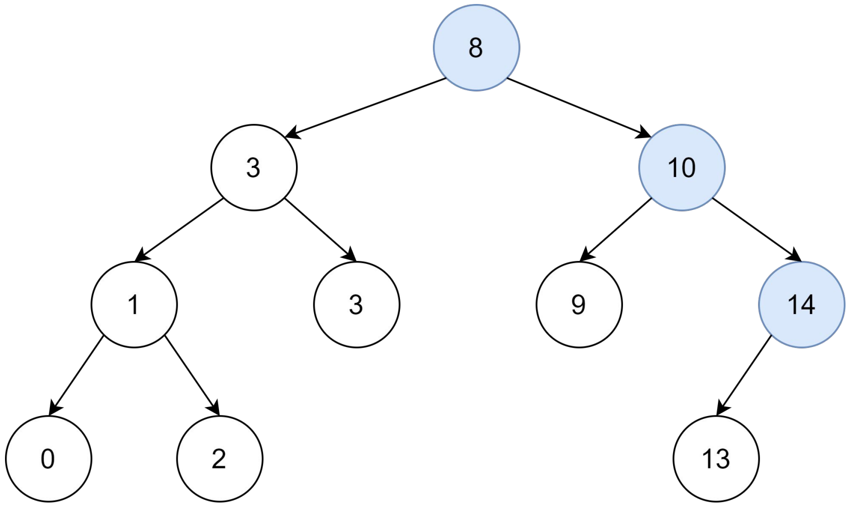
- Current node = root (8)
  5 < 8, go to left subtree
- Current node = 3
  5 > 3, go to right subtree
- Current node = 3
  5 > 3, go to right subtree
- Right subtree is empty, 5 is not in BST



Binary Search Tree - Query extrema

Find the maximum value in BST – rightmost node

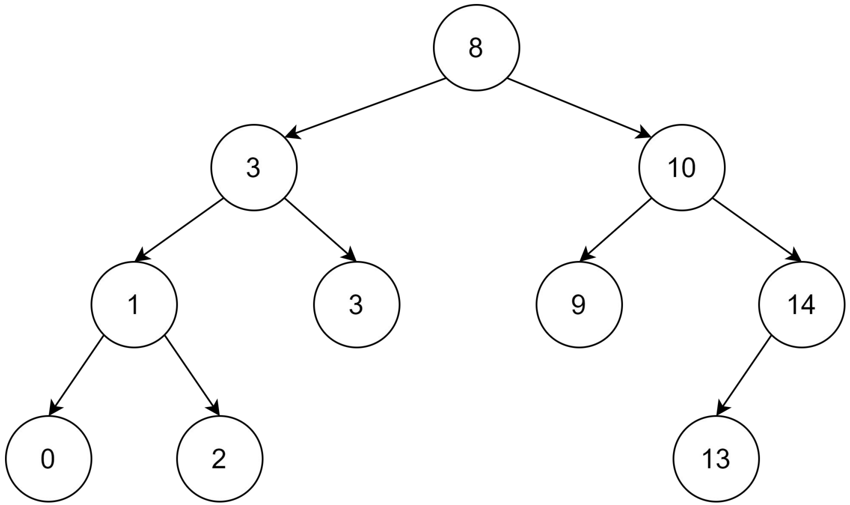
- Current node = root (8)
  Has right subtree → go right
- Current node = 10
  Has right subtree → go right
- Current node = 14
  No right subtree → 14 is the maximum



Binary Search Tree - Query lower bound

Find smallest element which > lower_bound

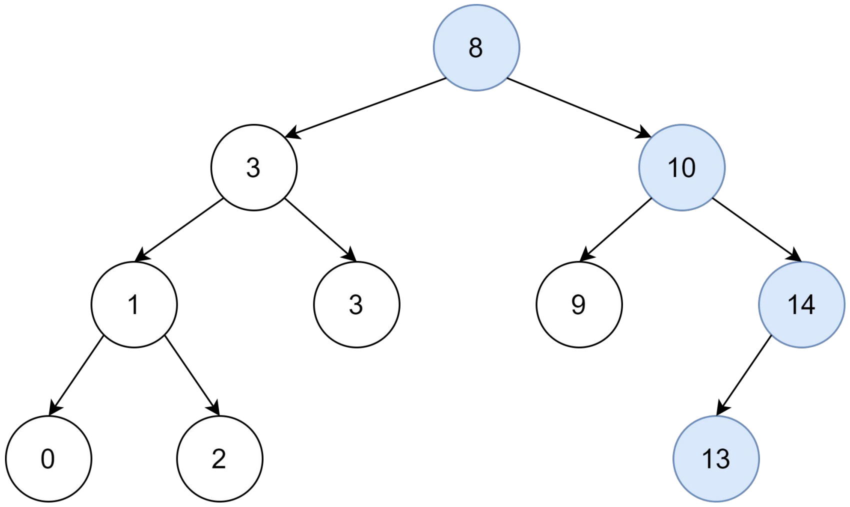
- DFS from the root
- If current node value > lower_bound, save current node value as temporary result and go to left subtree
  Else go right subtree
- The final current node value stored is the value we are looking for



Binary Search Tree - Query lower bound

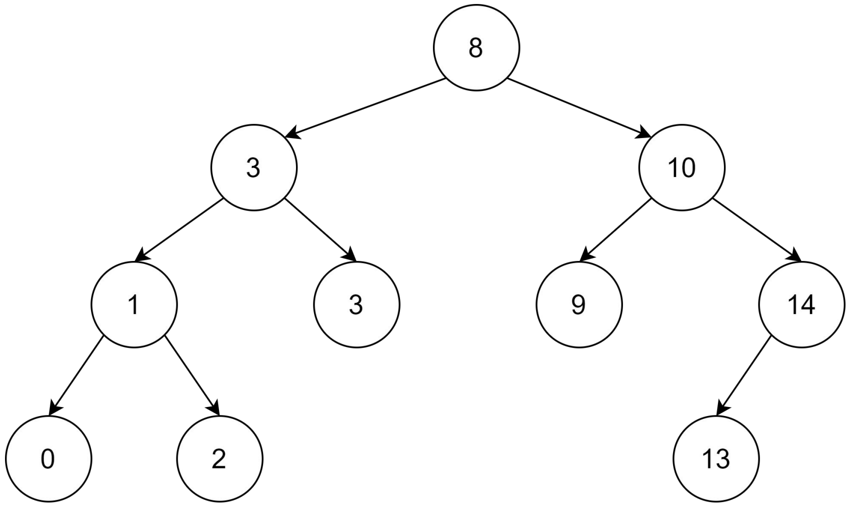
Find the smallest element > 11

- Current node = root (8)
  11 > 8, go to right subtree
- Current node = 10
  11 > 10, go to right subtree
- Current node = 14
  11 < 14, go to left subtree, mark 14 as smallest
- Current node = 13
  11 < 13, go to left subtree, mark 11 as smallest
- Left subtree is empty, 13 is the smallest element > 11



Binary Search Tree - Deletion

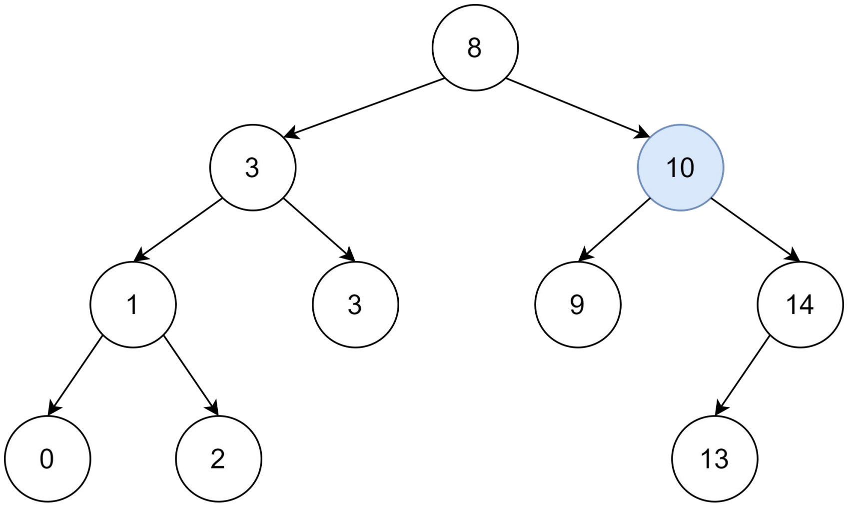
- Locate the to-be-delete element
- Depends on the location of the node:
  - If it is a leaf node, delete directly
  - If it has left subtree, swap it with the largest element in its left subtree
  - If it has right subtree, swap it with the smallest element in its right subtree
- Do it recursively until the to-be-deleted element is a leaf and delete it directly



Binary Search Tree - Deletion

Delete 10

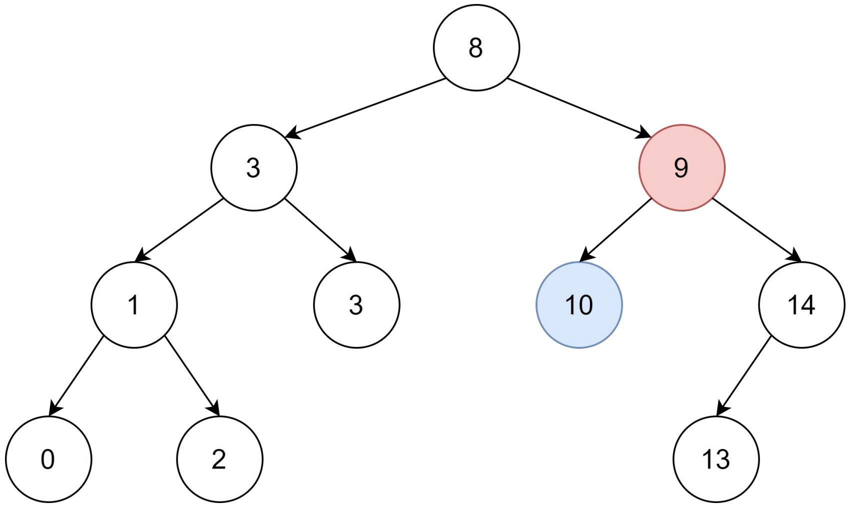
- Step 1: Find 10
- Step 2: As 10 has left subtree, swap it with the largest element in its left subtree (9)
    - This can be done by keep going right in the left subtree
- Step 3: delete 10 as it is a leaf



Binary Search Tree - Deletion

Delete 10

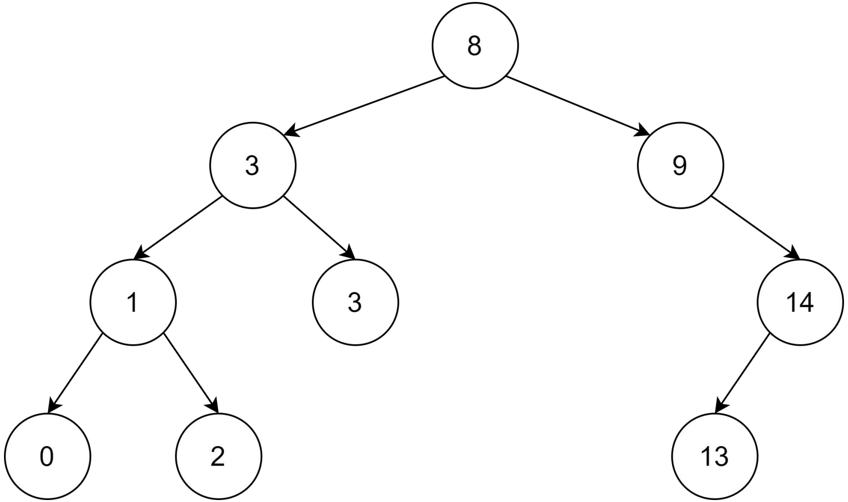
- Step 1: Find 10
- Step 2: As 10 has left subtree, swap it with the largest element in its left subtree (9)
  - This can be done by keep going right in the left subtree
- Step 3: delete 10 as it is a leaf



Binary Search Tree - Deletion

Delete 10

- Step 1: Find 10
- Step 2: As 10 has left subtree, swap it with the largest element in its left subtree (9)
  - This can be done by keep going right in the left subtree
- Step 3: delete 10 as it is a leaf



Binary Search Tree - Time complexity

- In insert, query, delete, we only need to DFS the tree to one of its leaf
- Time complexity = O(height of the BST)
- On average a BST has a height of O(log N)
- In the worst case, the tree forms a chain and time complexity becomes O(N)
- If there are Q operations, it costs O(QN) which can be very slow

Binary Search Tree - Time complexity

To avoid worst case BST:

- Shuffle the element before insertion
- Use self-balancing BST (Red-black tree, AVL Tree, Treap, Splay tree, etc.)
  - Similar to the normal BST but it maintains its height close to O(log N) by self rotation on the subtree
  - Very hard to code
- Use other search tree (Trie, segment tree)

Binary Search Tree - C++ Library

- set and map are implemented by red-black tree
- Support insert, delete, query extrema, lower_bound, exact value operation
- However ranking operation is not supported

```
#include <bits/stdc++.h>
using namespace std;
int main() {
  set<int> s;
  s.insert(4);
  s.insert(6);
  s.insert(9);
  cout << "Size = " << s.size() << "\n";
  for (auto str : s) {
    cout << str << "\n";
  }
  if (s.find(4) != s.end()) {
    cout << "4 is in the BST\n";
  }
  s.erase(6);
  cout << "After deletion: \n";
  for (auto str : s) {
    cout << str << "\n";
  }
}
```

```
Size = 3
4
6
9
4 is in the BST
After deletion:
4
9
```

Binary Search Tree - C++ Library

- Both map and set does not support duplicate keys – use multiset / multimap instead
- std::lower_bound != set::lower_bound / map::lower_bound

```
#include <bits/stdc++.h>
using namespace std;
int main() {
  set<int> s;
  s.insert(4);
  s.insert(6);
  s.insert(9);
  cout << "Size = " << s.size() << "\n";
  // get min
  cout << "Min = " << *s.begin() << "\n";
  // get max
  cout << "Max = " << *s.rbegin() << "\n";
  // lower_bound returns iter to the 1st elem >= 6
  cout << "Lower bound of 6 = " << *s.lower_bound(6) <<
"\n";
  // upper_bound returns iter to the 1st elem > 6
  cout << "Upper bound of 6 = " << *s.upper_bound(6) <<
"\n";
}
```

```
Size = 3
Min = 4
Max = 9
Lower bound of 6 = 6
Upper bound of 6 = 9
```

Binary Search Tree - C++ Library

Multiset / multimap

- Allow inserting same elements multiple times
- Pay attention to erase operation – depends on parameter type
  - Erase by iterator – will erase only one element pointed by the iterator
  - Erase by value – will erase all elements with the same value
- C++ map stores value in the form pair<key, value>, we can use map to store values in form of pair<element value, freq> to replace multiset

Binary Search Tree - C++ Library

- insert(x) in BST (using C++ map)

```
int freq;
freq = mymap.find(x) == mymap.end() ? 0 : mymap.find(x)->second;
mymap.erase(mymap.find(x));
mymap.insert(make_pair(x, freq + 1));
```

- delete(x) in BST (using C++ map)

```
int freq;
freq = mymap.find(x) == mymap.end() ? 0 : mymap.find(x)->second;
mymap.erase(mymap.find(x));
If (freq > 1) mymap.insert(make_pair(x, freq - 1));
```

- More details: HKOI Training - Advanced C++ STL

Back to question 2

- There are 4 types of operations, with total of **N** operations
  - Delete **X** from the set
  - Insert **X** to the set
  - Find the Min/Max number of the set

- Find a solution with time complexity O(NlogN)

Binary Search Tree - HKOJ M0811

Problem: Maintain the following 5 operations

- Insert a number
- Query the minimum number
- Query the maximum number
- Delete the minimum number
- Delete the maximum number

Solution: BST

Hash table

Hash table

- An array that supports the following:
  - Insert any element → O(1)
  - Delete any element → O(1)
  - Query an exact element → O(1)

Hash table vs Frequency array

- Frequency array
  - To insert/update/delete/query an element x → arr[x]
  - arr[x] stores the frequency of x
- Hash table
  - To insert/update/delete/query an element x → arr[h(x)]
  - arr[y] stores the number x_1, x_2, x_3 … where $h(x_1) = h(x_2) = … = y$
  - h(x) is a hash function

Hash table - Hash function

- Hash function is a function that takes an element and maps to an integer which the integer is used as the array index
- Given a wide range of integers $[0, 10^9]$, we want to fit them into an array of size 11
  - Simplest hash function $h(x) = x \% 11$

Hash table - Insertion

Insert 876

- 876 % 11 = 7
- Store 876 in cell 7

							876			
0	1	2	3	4	5	6	7	8	9	10

Hash table - Insertion

Insert 452

- 452 % 11 = 1
- Store 452 in cell 1

	452						876			
0	1	2	3	4	5	6	7	8	9	10

Hash table - Deletion

Delete 452

- 452 % 11 = 1
- Cell 1 contains 452
- Delete 452 from cell 1

							876			
0	1	2	3	4	5	6	7	8	9	10

Hash table - Query

Query 654

- 654 % 11 = 5
- Cell 5 is empty → 654 is not in the table

	452						876			
0	1	2	3	4	5	6	7	8	9	10

Hash table - Query

Query 419

- 419 % 11 = 1
- Cell 1 is not empty but 419 is not found in cell 1 → 419 is not in the table

	452						876			
0	1	2	3	4	5	6	7	8	9	10

Hash table - Query

Query 876

- 876 % 11 = 7
- 876 is stored in cell 7 → 876 is in the table

	452						876			
0	1	2	3	4	5	6	7	8	9	10

Hash table - Collision

Insert 887

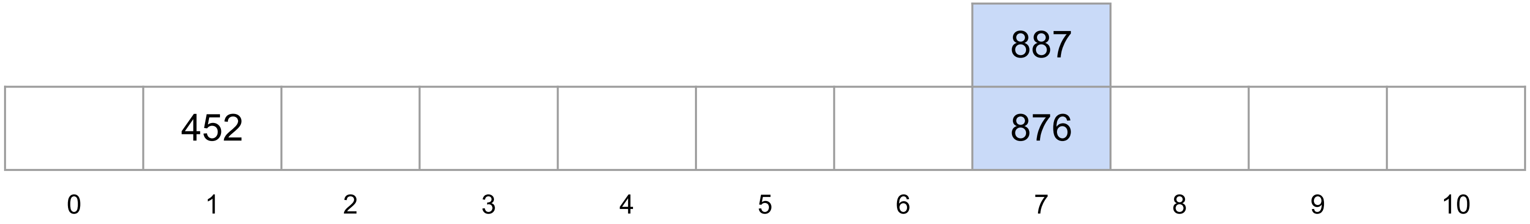
- 887 % 11 = 7
- Cell 7 is already occupied by 876
- Use array of vector instead of frequency array – Open hashing
- Store both 887 and 876 in cell 7

							887			
	452						876			
0	1	2	3	4	5	6	7	8	9	10

Hash table - Collision

Query 865

- 865 % 11 = 7
- We go through the vector of cell 7
- 865 is not found → 865 is not in the table



Hash table - Collision

There are many ways to handle collision

- Closed hashing
    - Linear probing
    - Quadratic probing
    - Double hashing
- Rehashing

One of the most common way to prevent collision is to use a good hash function

Hash table - Good Hash Function

- Goal: avoid collision – distribute the elements evenly in the hash table
  - Bigger hash table
  - Use a prime number modulus (if the data isn’t very random)
- Just use the one provided in STL/library

Hash table - Rolling hash

- How do we hash a string?
- First map the character to a integer, e.g. a = 1, b = 2, c = 3, etc.
- Suppose our string only consists of lowercase letters (a-z)
- Choose a prime modulus M
  - We commonly use $10^9 + 7$, any large prime should work
- The hash value of any string S can be computed by
  - $S[0] + S[1] * 27 + S[2] * 27^2 + \ldots + S[N - 1] * 27^{(N - 1)} \% M$
- So the hash value of "abcd" is
  - $1 + 2 * 27 + 3 * 27^2 + 4 * 27^3 = 80974$
- You can adjust the modulus and character mapping based on the input constraints

Hash table - Time complexity

- Suppose we have a good hash function which is able to distribute n elements evenly, the hash result range from [0..m]
- Each vector is expected to contains n / m elements
- Set m to a large number (10^6) which is comparable to n
  - The expected number of elements in each vector = 1
- Time complexity: O(1) for all insertion, deletion and query

Hash table - C++ Library

- unordered_map / unordered_set in C++ implements hash table
- Support insert, delete, query exact operations in O(1)
- Provides a default hash function for basic data types and string
  - Can ignore hash collision
  - Rehashing when needed
- Supported from C++11 and onwards

Hash table - C++ Library

unordered_map

- Stores elements in a key value combination
- Keys are unordered
- No duplicate keys – use unordered_multimap instead

Output:
Size = 6
Content:
3456 999
64 899
100000 7
32 67
5 2
456 1
314159 is not in the hash table
Size after deletion = 4
Content after deletion:
3456 999
64 899
100000 7
456 1

```
#include <bits/stdc++.h>
using namespace std;
int main() {
  unordered_map<int, int> umap;
  umap[456]++; // insertion by access
  umap[5] = 2;
  umap[32] = 67;
  umap[100000] = 7;
  umap.insert({64, 899}); // insertion by member
function
  umap.insert({3456, 999});

  cout << "Size = " << umap.size() << endl;
  cout << "Content: " << endl;
  for (auto x : umap) {
    cout << x.first << ' ' << x.second << endl;
  }

  if (umap.find(314159) == umap.end()) {
    cout << "314159 is not in the hash table" <<
endl;
  }

  umap.erase(5);              // by key
  umap.erase(umap.find(32)); // by iterator
  cout << "Size after deletion = " << umap.size() <<
endl;
  cout << "Content after deletion: " << endl;
  for (auto x : umap) {
    cout << x.first << ' ' << x.second << endl;
  }
}
```

Hash table - C++ Library

unordered_set

- Keys are hashed into indices of hash table
- Keys are unordered
- Only unique keys are allowed – used unordered_multiset instead

```
#include <bits/stdc++.h>
using namespace std;
int main() {
  unordered_set<string> uset;
  uset.insert("hkoi");
  uset.insert("random string");

  cout << "Size = " << uset.size() << endl;
  cout << "Content: " << endl;
  for (auto x : uset) {
    cout << x << endl;
  }

  if (uset.find("hkoi") != uset.end()) {
    cout << "hkoi is in the hash table" <<
endl;
  }

  uset.erase("hkoi");                        //
by key
  uset.erase(uset.find("random string")); //
by iterator
  cout << "Size after deletion: " <<
uset.size() << endl;
  cout << "Content after deletion: " << endl;
  for (auto x : uset) {
    cout << x << endl;
  }
}
```

Output:
Size = 2
Content:
random string
hkoi
hkoi is in the hash table
Size after deletion: 0
Content after deletion:

Hashtable vs BST

- Sometimes dealing with DP problem, the constraint is too big for using a simple array as DP memo
- Use hashtable instead of BST
- Hashtable has a faster query and insertion time than BST

Hash table - Related topics

- User defined hash functions
- Floating point number as hash table keys
- Anti-hash tests

Disjoint-set union-find (DSU)

Disjoint-set Union Find - Introduction

Tracking elements partitioned into a number of disjoint sets

- One element belongs to exactly one group
- One group may consists of any number of elements
- Example: Given 6 numbers 1, 2, 3, 4, 5, 6
  - {1, 2, 3}, {4, 5}, {6} are disjoint subsets
  - {1, 2, 3}, {2, 4, 5}, {6} are not disjoint subsets

Disjoint-set Union Find - Introduction

Operations

- **Union** - Merge two groups
  - Elements from two groups now belongs to the same group
  - Union({2, 3}, {4, 5, 6}) = {2, 3, 4, 5, 6}
- **Find** - find the group an elements is belong to (usually represented by a group ID)
  - Check if two elements belong to the same group
    - Let {1} be group 1, {2, 3} be group 2 and {4, 5, 6} be group 3
    - Find(2) = 2
    - Find(3) = 2
    - Find(6) = 3
    - 2 and 3 belongs to the same group but 6 is not in the same group with 2 and 3

DSU - Representation

Maintain an array **p[i]** which represents the group ID of element **i**

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

This array p represents disjoint sets

{1, 2, 8, 10} [Group ID = 1]

{3, 4, 5} [Group ID = 4]

{6, 7, 9] [Group ID = 9]

DSU - Naive implementation

Find operation: find(u)

- Group ID is simply p[u]
- Time complexity: O(1)

DSU - Naive implementation

Merge operation: union(u, v)

- Find all elements that belong to group p[v], update them to p[u]

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

- Time complexity: O(N)

DSU - Naive implementation

union(2, 3)

Before:

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

After:

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

DSU - Another implementation

Use tree structure to represent the groups, the group ID is the root of each tree

Using an array p[i] to represent the parent of the element i

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

Disjoint sets:

{1, 2, 3, 4, 7, 9} {5, 6, 8, 10}

DSU - Another implementation

- find(u)
- Recursively find the parent of u until p[u] == u

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

- Time complexity: O(N)
- Worst case the tree is a chain

DSU - Another implementation

union(u, v)

- Simply set the root of u as root of v

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

- Time complexity: O(1)

DSU - Optimization

There are two well-known optimization for DSU

- Path Compression
  - Optimizing find(u) operation
  - find(u) operation will have amortized O(log N) time complexity
- Union by size
  - Optimizing union(u, v) operation
  - union(u, v) will have amortized O(log N) time complexity
- Using both together will have amortized O(α(N)) time complexity, where α(N) is the inverse Ackermann function, $\alpha(N) < 4$ for $N < 2^{\wedge}(2^{65536}) - 3$

DSU - Path compression

- During finding root of element u, also update the parent of the visited nodes to the root of the group.
- Before:
  - ```
    int find(int u) {
      return p[u] == u ? u : find(p[u]);
    }
    ```
- After:
  - ```
    int find(int u) {
      return p[u] == u ? u : p[u] = find(p[u]);
    }
    ```

DSU - Union by size

We want to make the tree more balanced – to reduce number of step during 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] = rootv;
    subtree_size[rootv] += subtree_size[rootu];
  } else {
    p[rootv] = rootu;
    subtree_size[rootu] += subtree_size[rootv];
  }
}
```

DSU - Union by rank

- Similar idea to union by size, but instead we avoid making the tree tall.
- Define the height of tree as the max of distance of root to its leaves

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

DSU - NOI 2015 Day1 Q1 程序自動分析

Given N mathematical constraints, in the form of

- $A_i = A_j$
- $A_i \mathrel{!=} A_j$

Determine if all N constraints can be satisfied.

Note that discretization techniques is used in solving this problem, please refer to Optimization and Common Tricks.

DSU - NOI 2015 Day1 Q1 程序自動分析

- Solution: Merge variable that must be the equal in accordance with the ($A_i = A_j$) constraint, check if the $A_i \neq A_j$ constraints can be satisfied.
- Step by step:
  - For all the $A_i = A_j$, union A_i and A_j
  - For all the A_i != A_j, if A_i and A_j have the same root, output "NO"
  - Else output "YES"
- Exactly the task that could be solved using DSU

Common tricks

Heuristic Merging

- When we merge smaller things into larger things, we are using heuristic merging.
- Union by rank is a kind of heuristic merging.
- The height of tree would only increase by 1 when two tree have the same height. So the height of tree can only increase at most log N number of times. So the resultant tree has a height of log N.
- Such implementation will always have O(N log N).

Lazy Deletion

- Some operation may not affect the succeeding operations immediately & is costly to perform (e.g. deletion)
- Postpone such operations until the operation is necessary

Lazy Deletion - Delete operations on heap

- Consider the following problem
  - Insert a number
  - Query min
  - Remove any number (not supported by heap)
- We can use BST to maintain all of the above
  - Each operation takes O(log N)
- Two heaps also works (and it’s faster with the use of lazy deletion)
  - O(1) deletion (why?), query, O(log N) insertion

Lazy Deletion

- Insert a number → push the number to heap A
- Erase a number which is NOT minimum → push the number to heap B
- Erase a number in A → directly remove from A
  If the number is in B → erase it from heap B too

Lazy Deletion

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 Deletion

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 Deletion

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

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

Lazy Deletion

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 Deletion

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

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

Lazy Deletion

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 Deletion

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 Deletion

Add 5
Add 2
Add 3
Query Min
Remove 3
Add 1
Remove 2
Add 3
Remove 1 → now heap A and B have same min
Query Min

Same min in heap A and 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 both heap, erase 3
Heap A: {3, 5}
Heap B: {}

Lazy Deletion

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

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

Lazy Deletion

- Why does it works? → Erasing larger element does not affect the query result
- Delete it just before it becomes the minimum in A & query result

Adding a Lazy tag is a common technique in CP

- Label the to-be-deleted/updated element without actually performing the operation
- Perform the operation just before they affect the query result

Using 2 BSTs – Constant K-th element

- Problem:
  - Insert an element
  - Delete an element
  - Find the k-th (where k-th is constant) smallest element
- You may solve it by coding your own BST such that the location of the k-th smallest element is easily known
- Alternative: two C++-library BST (set or map)

Using 2 BSTs – Constant K-th element

- Use two maps (able to find max / find min / insert / delete)
- First map: always stores the smallest K-th element
  - Or all elements if # of elements < k
- Second map: always stores the remaining elements
  - All elements in map2 should >= any elements in map1
- Answer is always the maximum element of map1

Using 2 BSTs – Constant K-th element

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

map1: {1, 2}
map2: {}

Using 2 BSTs – Constant K-th element

K = 2

Insert 1

Insert 2

Insert 3

Query

Insert 4

Erase 3

Erase 2

Query

Insert 3

Query

map 1 contains K elements now as 3
>= largest element in map 1
Push to map 2

map1: {1, 2}

map2: {3}

Using 2 BSTs – Constant K-th element

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

map1: {1, 2}
map2: {3}

Using 2 BSTs – Constant K-th element

K = 2

Insert 1

Insert 2

Insert 3

Query

Insert 4

Erase 3

Erase 2

Query

Insert 3

Query

map1: {1, 2}

map2: {3, 4}

Using 2 BSTs – Constant K-th element

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

map1: {1, 2}
map2: {4}

Using 2 BSTs – Constant K-th element

K = 2

Insert 1

Insert 2

Insert 3

Query

Insert 4

Erase 3

Erase 2

Query

Insert 3

Query

2 is in map 1, after erasing map 1 contains only 1 elements, move the smallest element in map 2 to map 1

map1: {1, 4}

map2: {}

Using 2 BSTs – Constant K-th element

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

map1: {1, 4}
map2: {}

Using 2 BSTs – Constant K-th element

K = 2

Insert 1

Insert 2

Insert 3

Query

Insert 4

Erase 3

Erase 2

Query

Insert 3

Query

Since map 1 already contains K elements and 3 < largest element (4) in map 1, move largest element to map 2 and push 3 to map 1

map1: {1, 3}

map2: {4}

Using 2 BSTs – Constant K-th element

K = 2

Insert 1

Insert 2

Insert 3

Query

Insert 4

Erase 3

Erase 2

Query

Insert 3

Query → 3

map1: {1, 3}

map2: {4}

Using 2 BSTs – Constant K-th element

Time complexity

- Insertion / Deletion / Query: O(log N) each
- Number of re-push needed to ensure map 1 contains the K-th smallest elements anytime:
  - Case 1: just erased 1 element from map 1 → re-push the smallest element in map 2 to map 1
  - Case 2: just erased 1 element from map 2 → no re-push needed
  - Case 3: just inserted 1 element to map 1 → re-push the largest element in map 1 to map 2
  - Case 4: just inserted 1 element to map 2 → no re-push needed
- In any case, only O(1) operation is needed

Using 2 BSTs – Constant K-th element

Time complexity

- Insertion / Deletion / Query: O(log N) each
- Number of re-push needed to ensure map 1 contains the K-th smallest elements anytime:
  - O(1) re-push operation * O(log N) per operation = O(log N) for re-push
- Time complexity: O(Q log N) where Q is the number of operations

Using 2 BSTs – Constant K-th element

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

Summary

- Important to learn when and how to use data structure properly in contests
- Learn C++ STL which will ease your work in implementing the data structures → Advanced C++ STL
- Use data structures that supports the operations you need efficiently

Practice problems

- 01019 - Addition II → (heaps)
- M0811 - Alice's Bookshelf → (2 heaps with lazy propagation or balanced BST)
- 01090 - Diligent → (balanced BST or hash table)
- N1511 - 程序自動分析 → (DSU)
- IOI 2012 Practice Q3 - Touristic plan → (Constant K-th element trick)

More practice problems

- AP121 - Dispatching → (Heuristic Merging)
- M1811 - Almost Constant
- M1533 - Bridge Routing
- M2214 - Fluctuating Market

Hard Problems (can be done in oj.uz):

- BalticOI 2016 D1Q2 - Park
- BalticOI 2018 D2Q2 - Genetics
- JOI Spring Camp 2020 D2Q2 - Making Friends on Joitter is Fun

Reference

- Data Structures (II) 2023 lecture notes