Data Structures (II)

Ethen Yuen {ethening}
2026-03-07

Agenda

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

Data Structure

A data structure is a way to organize and store data so that we can perform operations on the data **efficiently**.

- How do you measure **efficiency**?
- We can only talk about efficiency based on the **application** of the data structure → there are no way to judge whether a data structure fits if we don’t know what it is used for.
  Let’s look at some examples.

Data Structure

[Problem 0] Given N integers. What is the best data structure to store them?

- You should realize that this is a pointless question.
    - What is the data used for? What query are we answering with these data?

Data Structure

[Problem 0.1] Given N integers. Let's say we need to support two operations with this N integers:

- **Assign(x, v):** Assign value v to the x-th integer.
- **Query():** Query for the sum of all N integers.
- What is a data structure that can perform these operations?
  - A simple array, and another integer maintaining the sum.
  - Assign: O(1), Query: O(1)

Data Structure

[Problem 0.2] Given N integers. Let's say we need to support two operations with this N integers:

- **Assign(x, v):** Assign value v to the x-th integer.
- **Query(v):** Query for the frequency that v appears within the N integers.
- What is a data structure that can perform these operations?
  - An array to maintain the latest N integers, an array maintaining the frequency count of each value.
  - Assign: O(1), Query: O(1), **Memory Complexity: O(value range)**
  - We will have problem when v can be large as 1e9.
    - On the contrary, if v is small like 1e6, this approach is very suitable. e.g. counting the frequency of ages of HK citizens.

Data Structure

[Problem 0.3] Given N integers. Let’s say we need to support an operation with this N integers:

- **Query(l, r):** Query the sum of the l-th to r-th integers (inclusive).
- What is a data structure that can perform these operations?
    - A partial sum array of the N integers.
    - Init: O(N), Query: O(1)

Data Structure

[Problem 0.4] Given N integers. Let’s say we need to support an operation with this N integers:

- **Add(l, r, v):** Add v to the l-th to r-th integers (inclusive).
- **Query(l, r):** Query the sum of the l-th to r-th integers (inclusive).
- What is a data structure that can perform these operations?
  - A segment tree, Init: O(N), Add: O(log N), Query: O(log N).
    (Note: You would not be learning this today)
- What if (r - l) always <= 10?
- What if all Add() always happens before Query()?

Takeaways

- Data structures not only refer to those **typical data structures**: stack, queue, linked list, or those that we will introduce later in this lecture.
- **Any way of storing data** that helps you efficiently perform operations can be considered a data structure.

Takeaways

- Data structures are invented to solve some particular problems.
- When you learn about them, some questions to ask yourself are:
    - What is the problem it aims to solve?
    - What are the operations supported? What’s their time complexity?
    - Any assumptions made on the data stored?
    - **Why does it works?**
        - Once you understand this, you can handle different variations easily as well.

Common operations

Insertion

Deletion

Modification

- Update a slot with value x with new value y

Query

- check if x exists
- min/max

You may notice that in general we can perform these operations in O(N) naively.
We want to aim for **sublinear time** (e.g. O(log N) or O(1), maybe amortized) in order to finish all O(Q) operations within time limit.

Problem 1

Support Q operations of the following 3 types:

- **Insert(x):** Add an integer x to the data structure
- **Delete(x):** Delete an integer x from the data structure
- **QueryMin():** Find the minimum number of the data structure

- Find a solution with time complexity O(Q log Q)

Binary Heap

Problem 1.1

Let’s first tackle a easier version of the previous problem.

Support Q operations of the following 3 types:

- **Insert(x):** Add an integer x to the data structure
- **DeleteMin():** Delete the minimum number from the data structure
- **QueryMin():** Find the minimum number of the data structure

Problem 1.1

- **Insert(x):** Add an integer x to the data structure
- **DeleteMin():** Delete the minimum number from the data structure
- **QueryMin():** Find the minimum number of the data structure

- These operations are exactly what a “heap” would handle.
- But before we go into the details of what a heap is, let’s try to implement these operations ourselves.

Problem 1.1

Suppose we have an array A.

- **Insert()**: we just append the new element to it.
- **QueryMin()**: loop through the whole array to find the minimum number.
- **DeleteMin()**: find the minimum number, then we can delete it and replace it by the last element.
- Problem: Notice that, the bottleneck is the search for the minimum number.

Problem 1.1

Searching the minimum everytime we want to query is too slow.

How about we make sure the minimum is always at the beginning, by keeping the array sorted?

- **QueryMin()**: return first element.
- **DeleteMin()**: discard the first element
- **Insert()**: need to insert in the sorted position → move larger elements one cell to the right.
- Problem: Keeping the array sorted is **very costly** when we need insertion.

Problem 1.1

	Attempt 1	Attempt 2
Insert(x)	O(1)	O(N)
DeleteMin()	O(N)	O(1)
QueryMin()	O(N)	O(1)

- In both attempts, we handle some operations very quickly, but leads to some operations being very slow.
- Can we balance it out to make them all sublinear time?

Problem 1.1

Can we balance it out to make them all sublinear time? Turns out that we can!

- Keeping the whole array sorted is **overkilled** for querying min.
- We only need to be able to achieve two things:
  - Fast access for min element
  - Quick to find the new min element after removing current min

To do this, linear data structure is not enough.

- We need to utilize the structure of a tree.

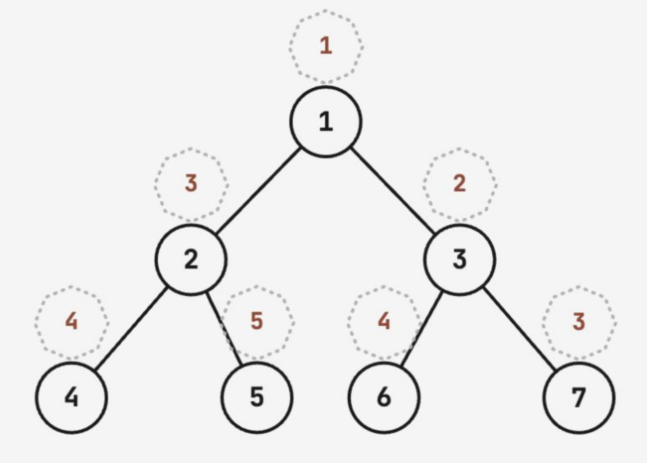
Binary Heap

Binary heap is a complete binary tree that maintain a single property:

- Each element is **not less than** its parent (min heap)

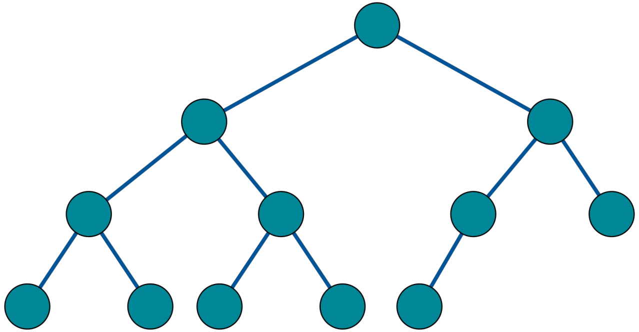
By maintaining this property, we can achieve both.

- Fast access for min element
  - Which is always the root
- Quick to find the new min element after removing current min
  - This tree structure allow us to do this. More details later.



Property of Binary Heap

- A binary tree
  - Each node have at most two children.
  - Usually denoted as Left Child and Right Child
- A complete binary tree
  - Beside the bottom level, every level is fully filled.
  - The bottom level nodes is as left as possible.
- Depth of binary heap: O(log N)
  - Why? (think about complete binary tree)



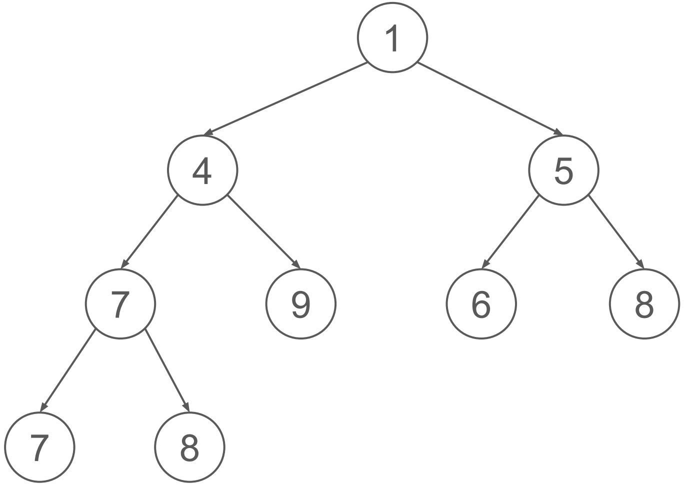
Binary Heap

- This binary heap supports query and deletion of min element
  - You can alter the maintained properties to make a max heap.
- Operations supported:
  - Insert(x): O(log N)
  - DeleteMin(): O(log N)
  - QueryMin(): O(1)
  - Delete, Query, Update any number: Not directly supported
- We will look into each operations and explain each one by one.
  - The main idea is, after every alteration of the heap, we must spend effort to make sure the heap follows the maintained properties that: Each element is not less than its parent.

Binary Heap – Insert

Steps

- Place the new element at the end of the heap
- Repeatedly sift-up the the node until new element's parent < new element

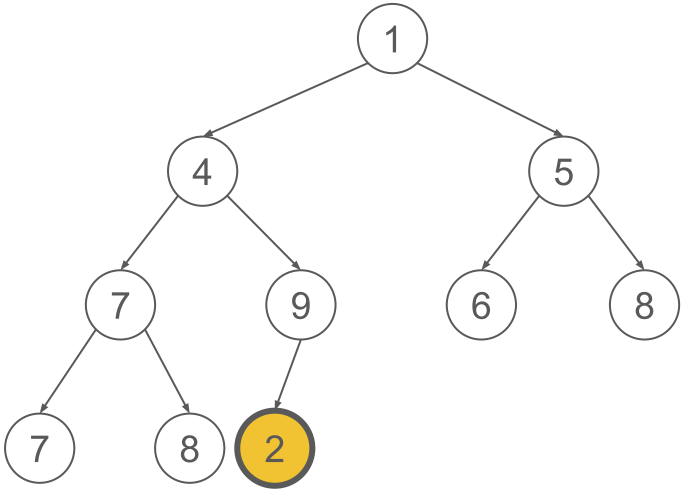


Insert 2 to this heap

Binary Heap – Insert

Steps

- Place the new element at the end of the heap
- Repeatedly sift-up the the node until new element’s parent < new element

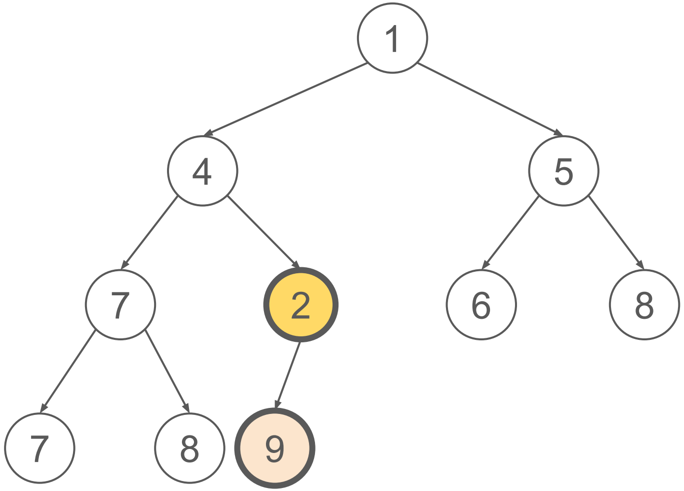


Insert 2 to this heap

Binary Heap – Insert

Steps

- Place the new element at the end of the heap
- Repeatedly sift-up the the node until new element's parent < new element

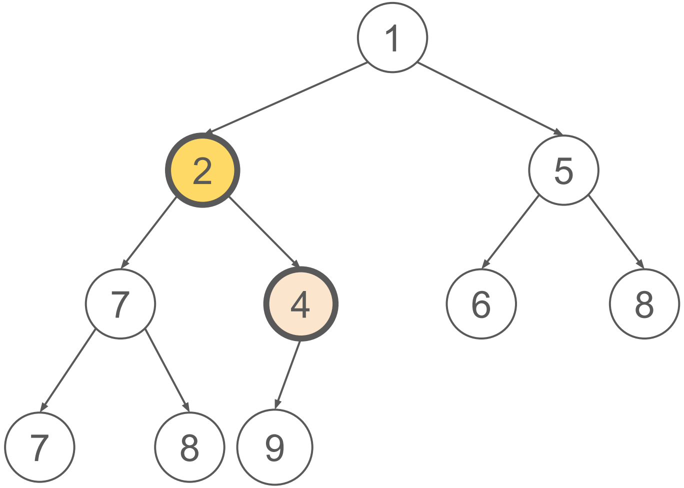


2 <= 9, swap 2 and 9

Binary Heap – Insert

Steps

- Place the new element at the end of the heap
- Repeatedly sift-up the the node until new element's parent < new element

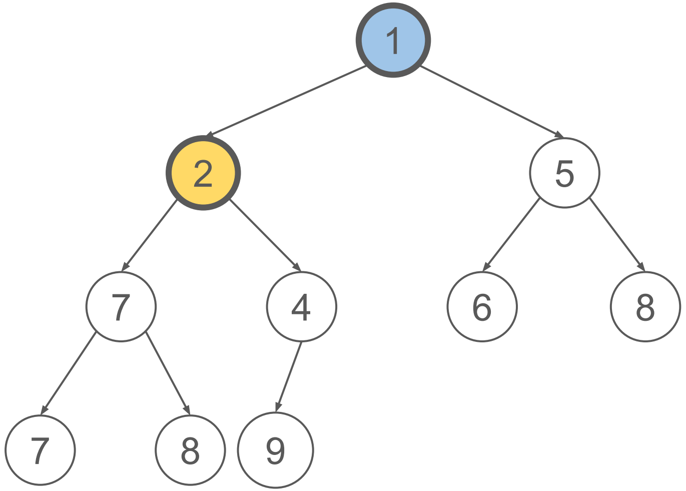


2 <= 4, swap 2 and 4

Binary Heap – Insert

Steps

- Place the new element at the end of the heap
- Repeatedly sift-up the the node until new element's parent < new element



2 >= 1 – Done!

Binary Heap – Insert

Time Complexity:

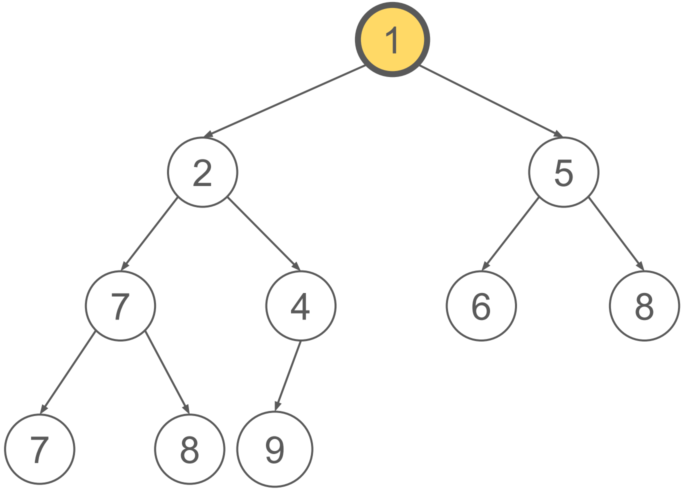
- Number of sift-up → bounded by Height of the heap
- O(log N)

- Notice the difference between this, and when we try to maintain a sorted array after insertion.

Binary Heap – QueryMin

Because of the structure of heap, the minimum element must be at the root

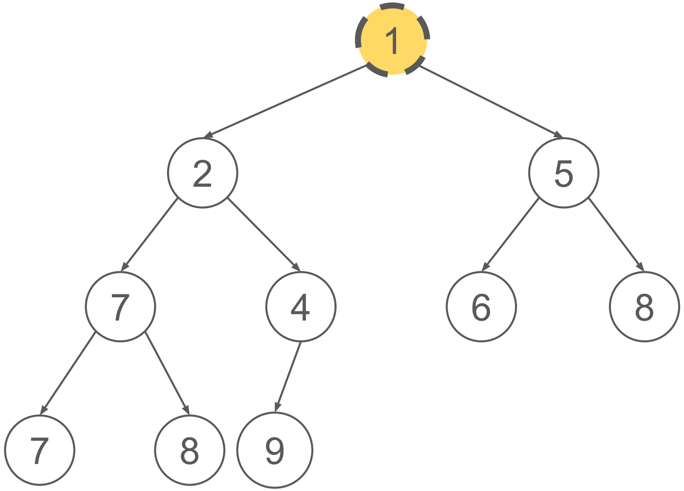
Time complexity: O(1)



Binary Heap – DeleteMin

Steps

- Replace the root node with the last node in heap
- Repeatedly sift-down the new root node (x) until value of x <= its children

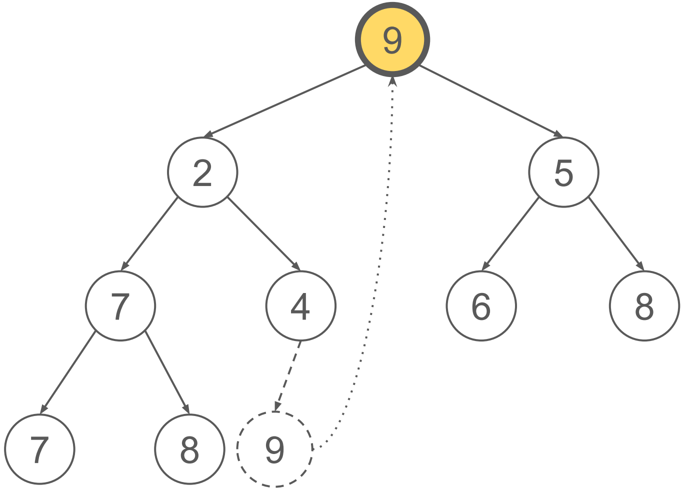


TODO: Delete the minimum (root node)

Binary Heap – DeleteMin

Steps

- Replace the root node with the last node in heap
- Repeatedly sift-down the new root node (x) until value of x <= its children

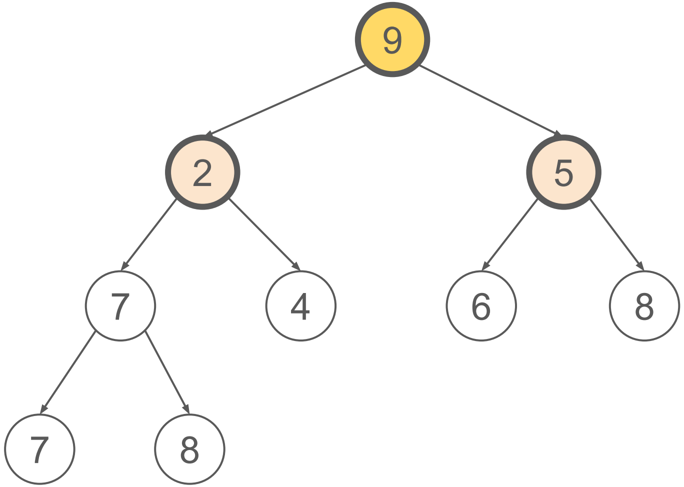


Replace 9 as new root node

Binary Heap – DeleteMin

Steps

- Replace the root node with the last node in heap
- Repeatedly sift-down the new root node (x) until value of x <= its children

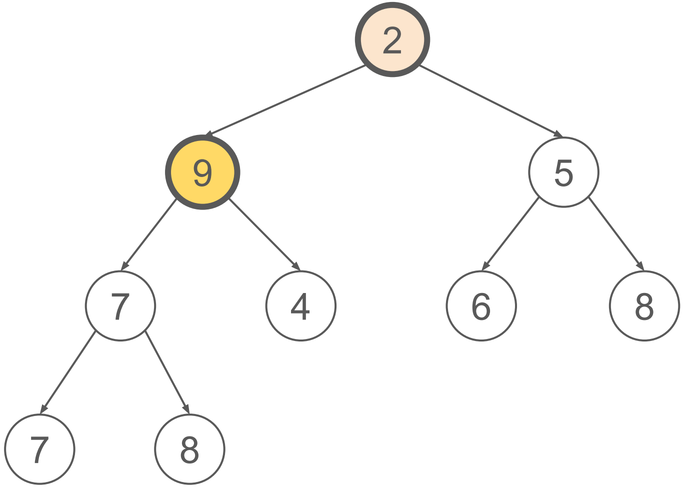


9 >= min(2, 5), swap 9 and min(2, 5) = 2

Binary Heap – DeleteMin

Steps

- Replace the root node with the last node in heap
- Repeatedly sift-down the new root node (x) until value of x <= its children

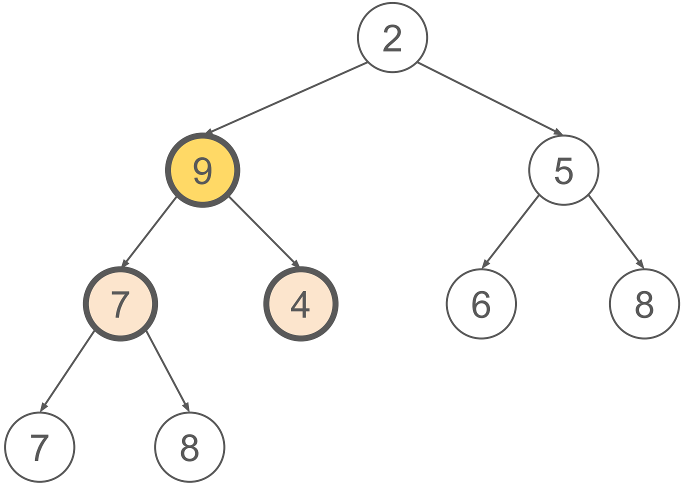


9 >= min(2, 5), swap 9 and min(2, 5) = 2

Binary Heap – DeleteMin

Steps

- Replace the root node with the last node in heap
- Repeatedly sift-down the new root node (x) until value of x <= its children

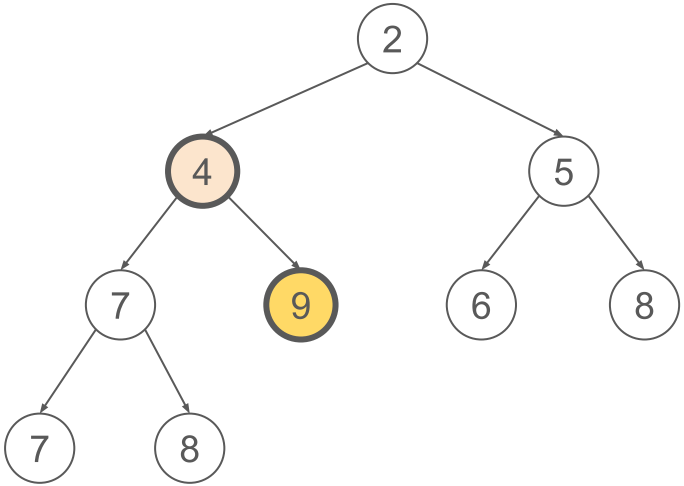


9 >= min(7, 4), swap 9 and min(7, 4) = 4

Binary Heap – DeleteMin

Steps

- Replace the root node with the last node in heap
- Repeatedly sift-down the new root node (x) until value of x <= its children



9 >= min(7, 4), swap 9 and min(7, 4) = 4

Binary Heap – DeleteMin

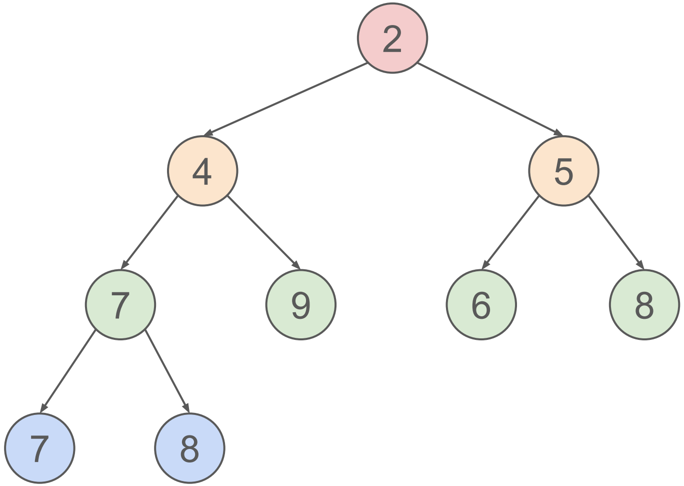
Time Complexity:

- Number of sift-down → bounded by Height of the heap
- O(log N)

Binary Heap – Implementation

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

Array Representation

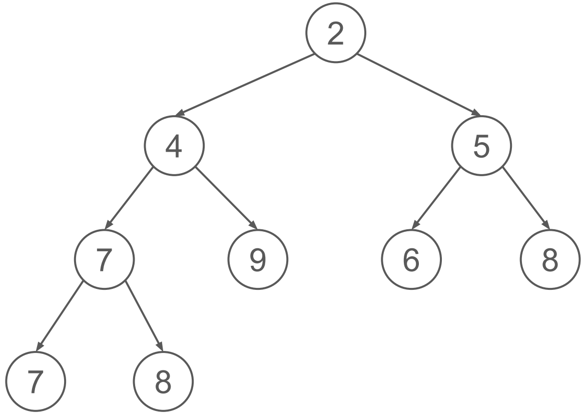


Tree Representation

Binary Heap – Array Implementation

In a 1-based array:

- Root of heap → arr[1]
- Parent of a node (arr[k]) → arr[k / 2]
- Children of a node (arr[k]) → arr[2k] and arr[2k + 1]
- Last node in the heap → arr[N]
- Next node inserted → arr[N + 1]



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

Binary Heap - C++ Library

- std::priority_queue supports all 3 operation
- Default max heap

```
#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

```
#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 min
  cout << "Min = " << pq.top() << endl;

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

```
Output:
Size = 3
Min = 1
New min = 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 min
  cout << "Min = " << pq.top().val << endl;

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

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

Binary Heap Practice Task - HKOJ B100 Binary Heap

- https://judge.hkoi.org/task/B100
- Let’s spend some time to code this task together.
- In contest time, it is rare that you need to code a heap by yourself. Learn to use C++ std library to your advantage!

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 Problem 1

Support Q operations of the following 3 types:

- **Insert(x):** Add an integer x to the data structure
- **Delete(x):** Delete an integer x from the data structure
- **QueryMin():** Find the minimum number of the data structure

- Find a solution with time complexity O(Q log Q)

How do we delete elements that is not the minimum with heap?

- Hint: you need to maintain **2 heaps**
- We will revisit this later.

Problem 2

Support Q operations of the following types:

- **Insert(x):** Add an integer x to the data structure
- **Delete(x):** Delete an integer x from the data structure
- **QueryMin() / QueryMax():** Find the minimum / maximum number of the data structure
- **Find(x):** Check whether the value x exists in the data structure
- **LowerBound(x):** Find the minimum number of the data structure >= x

Find a solution with time complexity O(Q log Q)

Problem 2

Support Q operations of the following types:

- **Insert(x):** Add an integer x to the data structure
- **Delete(x):** Delete an integer x from the data structure
- **QueryMin() / QueryMax():** Find the minimum / maximum number of the data structure
- **Find(x):** Check whether the value x exists in the data structure
- **LowerBound(x):** Find the minimum number of the data structure >= x

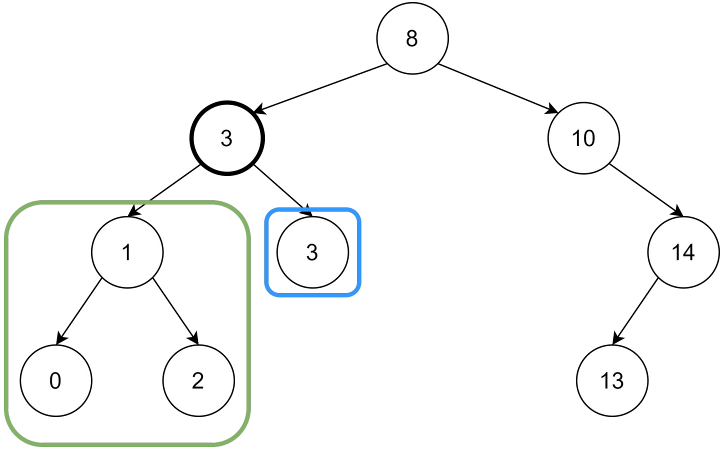
Think about a question: Why Heap cannot support these?

Binary Search Tree (BST)

Binary Search Tree

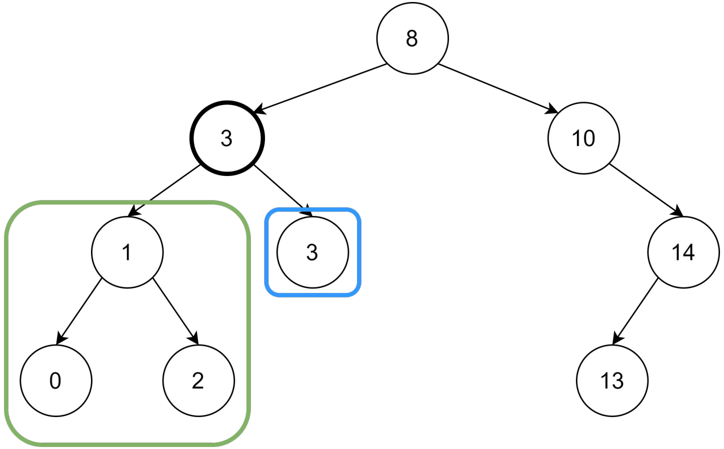
- A binary tree (each node <= 2 children)
- Each nodes has a value
  - Value of all nodes in the **left subtree** of node k < value of node k
  - Value of all nodes in the **right subtree** of node k >= value of node k

- Compare to heap, which we only maintain larger/smaller relationship between parent and child, we maintain the **complete order** in the BST:
  - This mean we know how to locate a value.
  - Think of it as like, putting a sorted array in a tree structure.



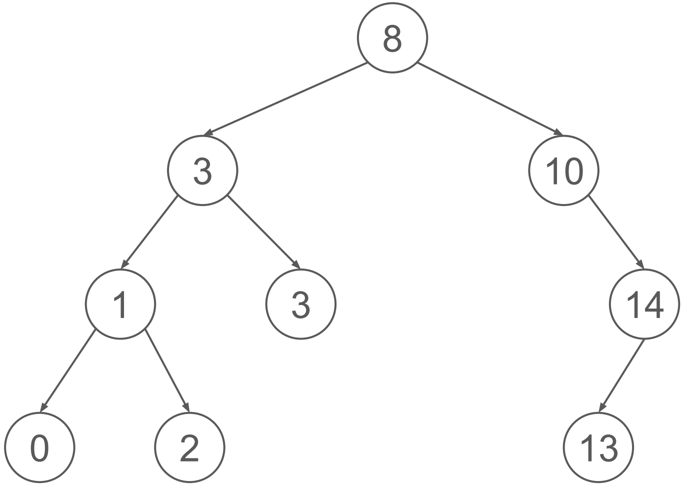
Properties of Binary Search Tree

- Operations supported:
  - **Insert(x):** O(?)
  - **Delete(x):** O(?)
  - **QueryMin() / QueryMax():** O(?)
  - **Find(x):** O(?)
  - **LowerBound(x):** O(?)



Binary Search Tree – Insert

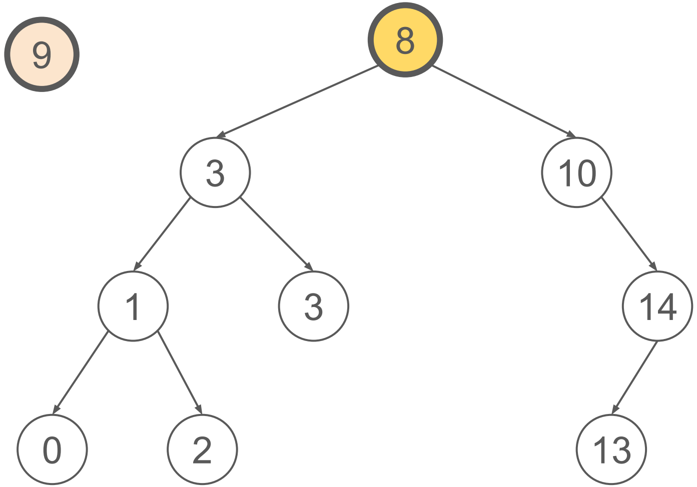
- DFS from the root
- Repeatedly travel down the tree -
  If the inserted value < the current node's value, go left;
  go right otherwise
- Until we find a empty space



Binary Search Tree – Insert

Insert 9 to the BST

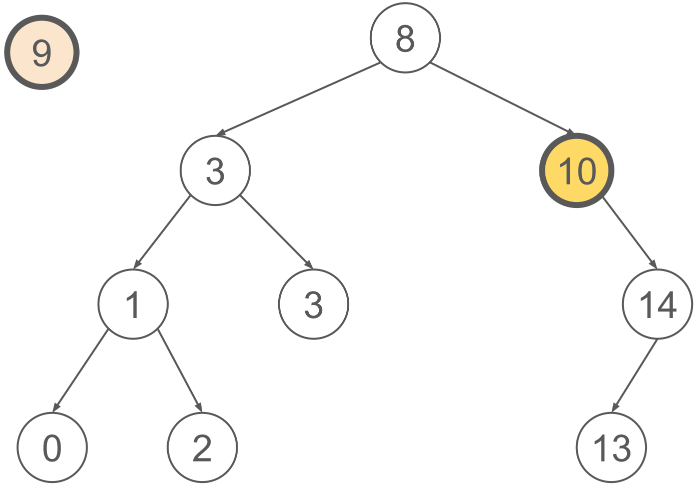
- Current node = root (8)
  9 > 8, go to right subtree
- Current node = 10
  9 < 10, go to left subtree
- Left subtree is empty – place 9 at this empty spot



Binary Search Tree – Insert

Insert 9 to the BST

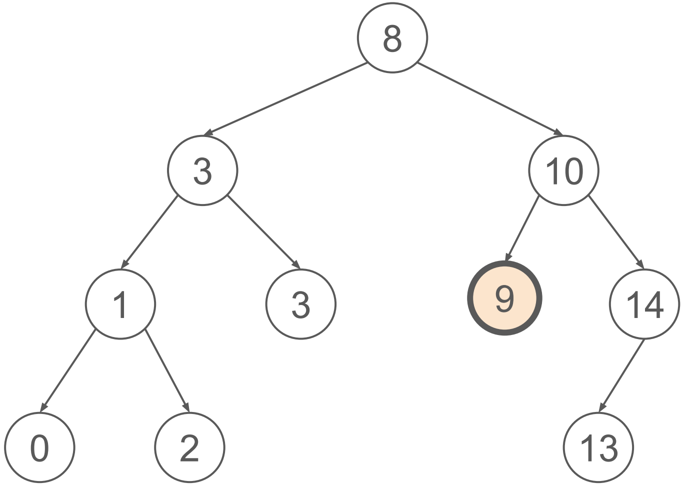
- Current node = root (8)
  9 > 8, go to right subtree
- Current node = 10
  9 < 10, go to left subtree
- Left subtree is empty – place 9 at this empty spot



Binary Search Tree – Insert

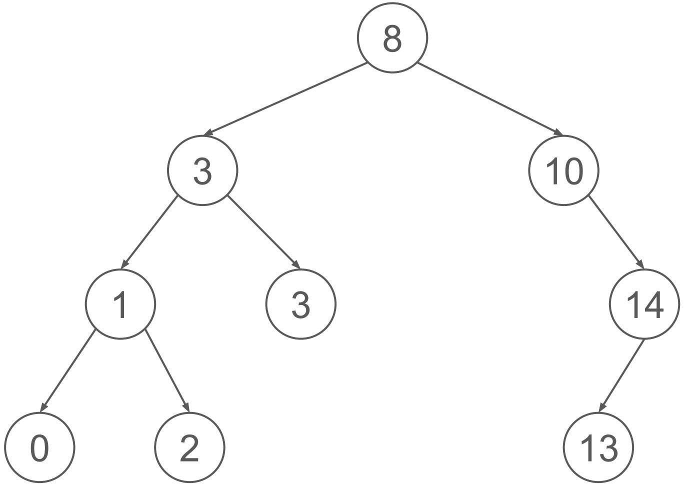
Insert 9 to the BST

- 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



Binary Search Tree – Find

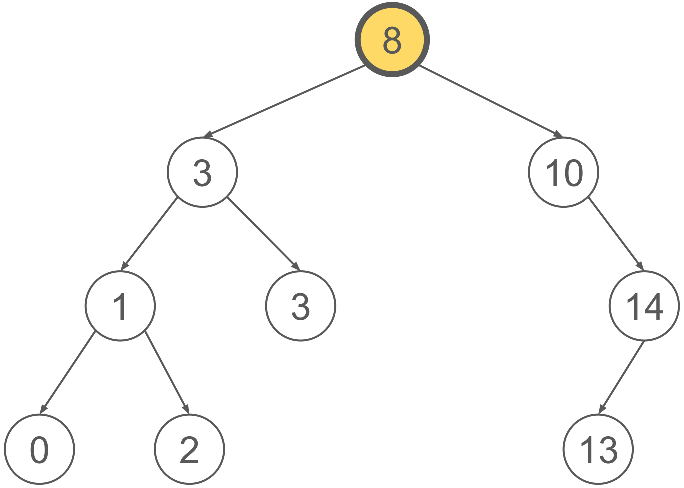
- DFS from the root
- Repeatedly travel down the tree -
  If the inserted value < the current node's value, go left;
  go right otherwise
- Until the value is found



Binary Search Tree – Find (1)

Find if 2 exists in a BST

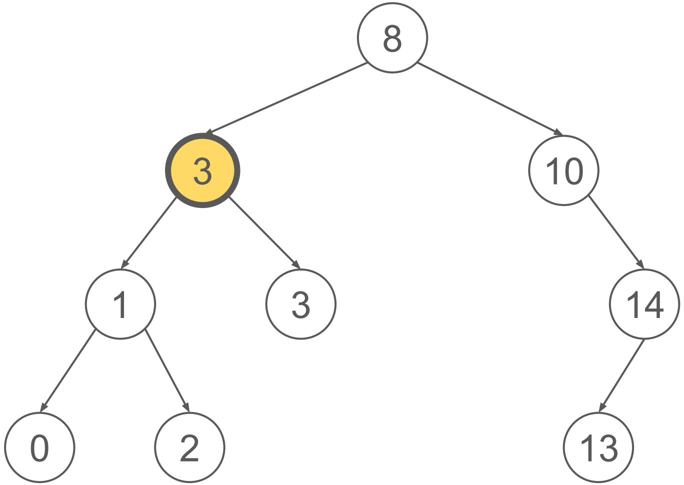
- Current node = root (8)
  2 < 8, go to left subtree
- Current node = 3
  2 < 3, go to left subtree
- Current node = 1
  2 > 1, go to right subtree
- 2 is found!



Binary Search Tree – Find (1)

Find if 2 exists in a BST

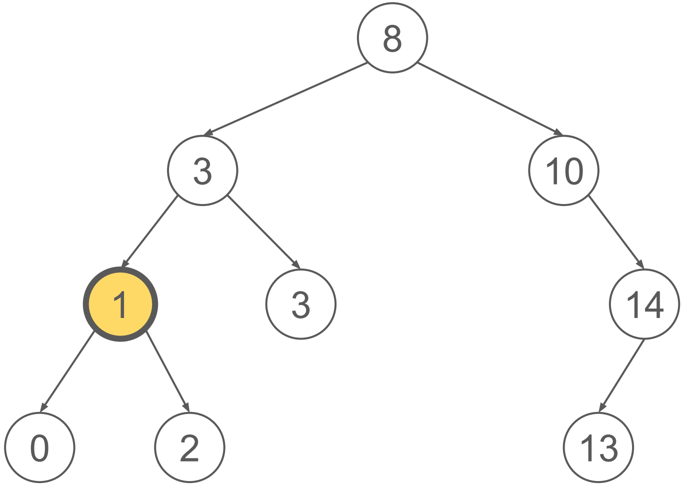
- Current node = root (8)
  2 < 8, go to left subtree
- Current node = 3
  2 < 3, go to left subtree
- Current node = 1
  2 > 1, go to right subtree
- 2 is found!



Binary Search Tree – Find (1)

Find if 2 exists in a BST

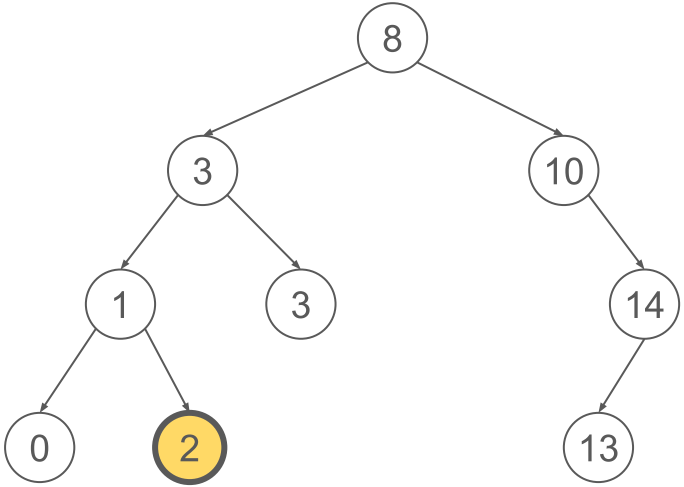
- Current node = root (8)
  2 < 8, go to left subtree
- Current node = 3
  2 < 3, go to left subtree
- Current node = 1
  2 > 1, go to right subtree
- 2 is found!



Binary Search Tree – Find (1)

Find if 2 exists in a BST

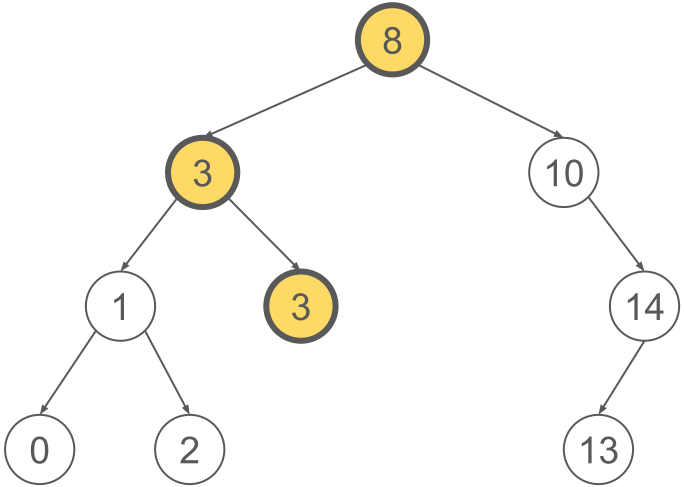
- Current node = root (8)
  2 < 8, go to left subtree
- Current node = 3
  2 < 3, go to left subtree
- Current node = 1
  2 > 1, go to right subtree
- 2 is found!



Binary Search Tree – Find (2)

Find if 5 exists in a BST

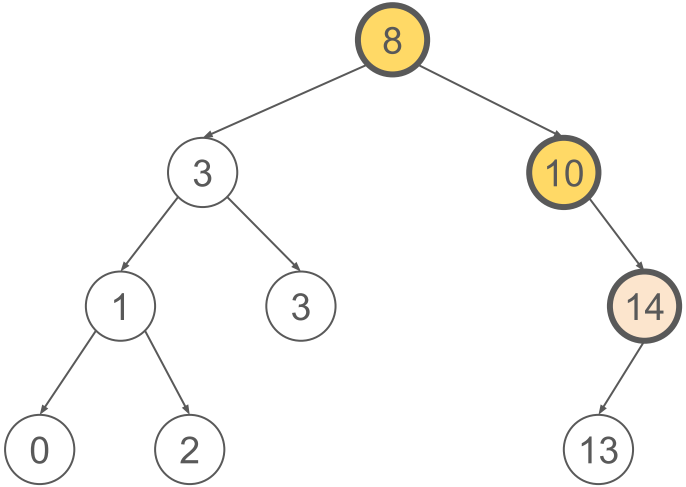
- Current node = root (8)
  5 < 8, go to left subtree
- Current node = 3
  5 > 3, go to left subtree
- Current node = 3
  5 > 4, go to right subtree
- Right subtree is empty – 5 is not in the BST



Binary Search Tree – Query Extrema

Find maximum value in a 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

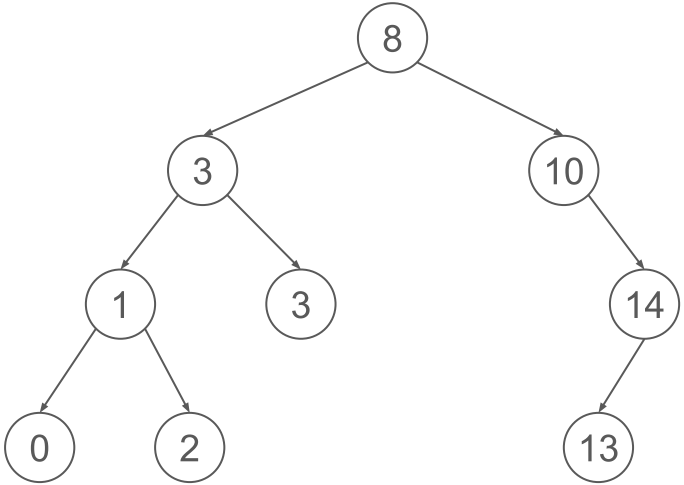


Binary Search Tree – LowerBound

Find smallest element which >= lower_bound

- DFS from the root
- If current_value >= lower_bound

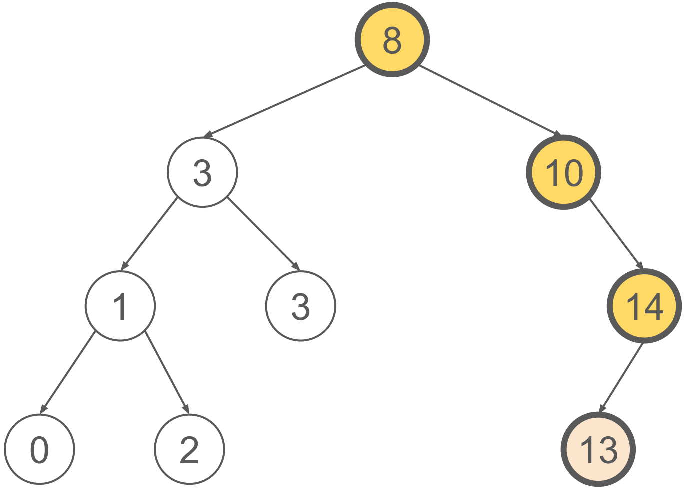
```
If current_value >= lower_bound
        Res = current_value
        gotoLeftSubtree;
Else
        gotoRightSubtree;
```



Binary Search Tree – LowerBound

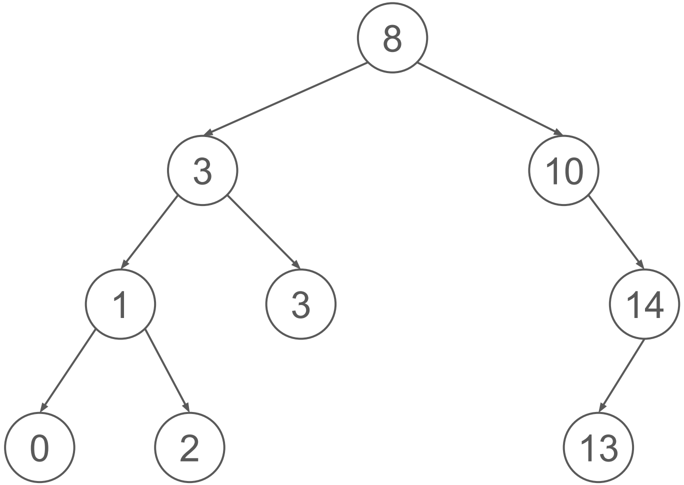
Find smallest element >= 11

- 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 & set lower bound = 14
- Current node = 13,
  11 < 13, set lower bound = 13 and done (no children)



Binary Search Tree – Delete

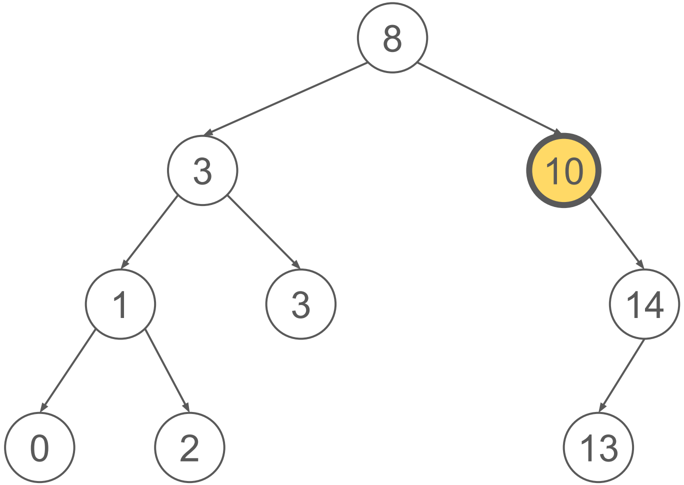
- Locate the to-be-delete element
- Depends on the location of the node:
  - If it is a leaf, delete directly
  - If it has a left subtree, swap it with the largest element **in its left subtree**
  - If it has a right subtree, swap it with the smallest element **in its right subtree**
- Do it recursively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



Binary Search Tree – Delete

Delete 10

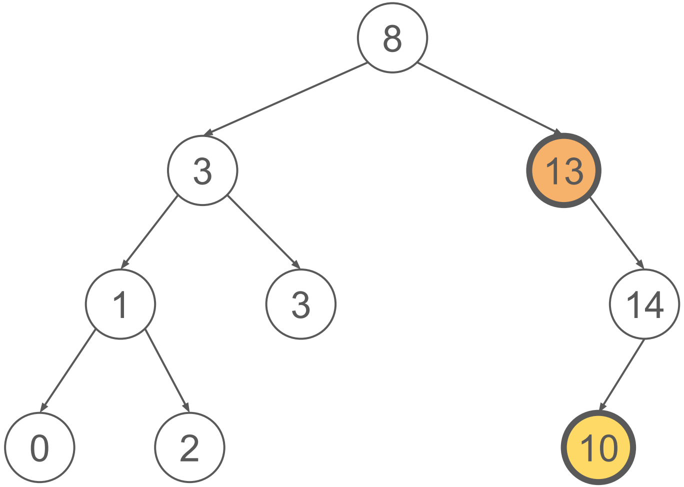
- Step 1: Find 10
- Step 2: As 10 only has right subtree, swap 10 with the smallest element in its right subtree (13)
    - It can be done by keep going left from the right subtree of 10
- Step 3: delete 10 as it is a leaf



Binary Search Tree – Delete

Delete 10

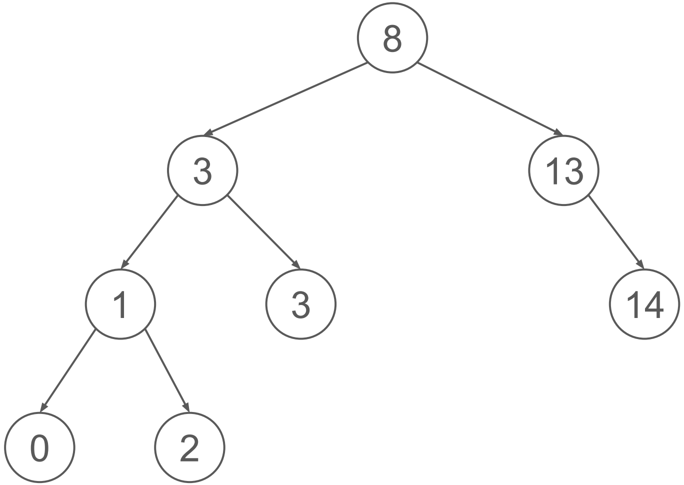
- Step 1: Find 10
- Step 2: As 10 only has right subtree, swap 10 with the smallest element in its right subtree (13)
    - It can be done by keep going left from the right subtree of 10
- Step 3: delete 10 as it is a leaf



Binary Search Tree – Delete

Delete 10

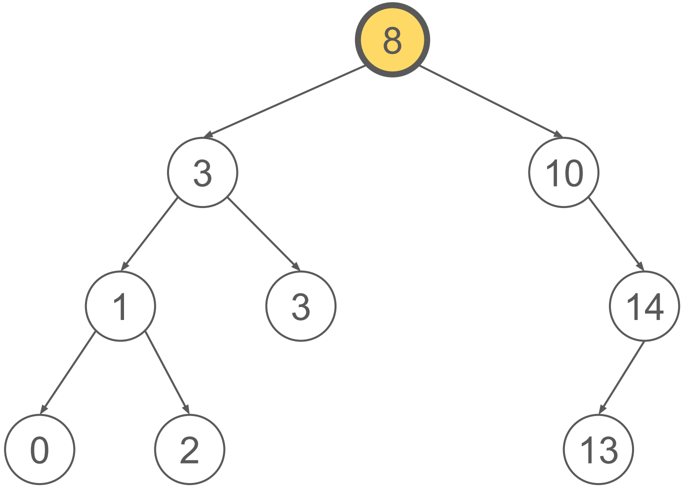
- Step 1: Find 10
- Step 2: As 10 only has right subtree, swap 10 with the smallest element in its right subtree (13)
    - It can be done by keep going left from the right subtree of 10
- Step 3: delete 10 as it is a leaf



Binary Search Tree – Delete

Delete 8

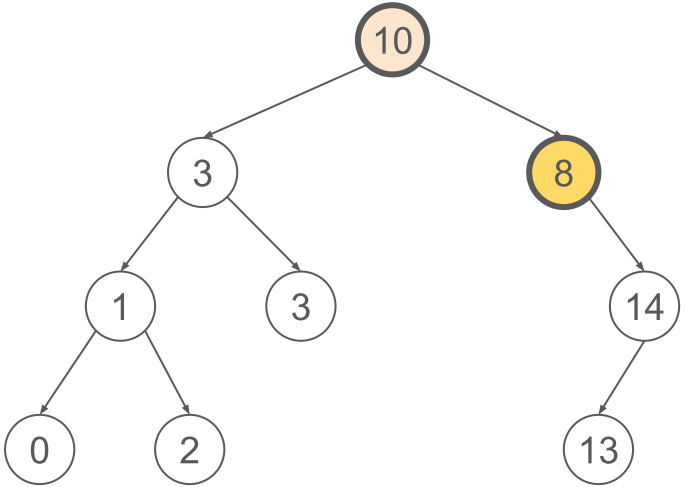
- Step 1: Find 8
- Step 2: Find the smallest element in the right subtree of 8 (10) and swap 8 with it
    - (you can choose left/right subtree in case of both exist)
- Step 3: Continue to swap 8 with the smallest element in the right subtree of 8 (13)
- Step 4: Delete 8 as it becomes a leaf



Binary Search Tree – Delete

Delete 8

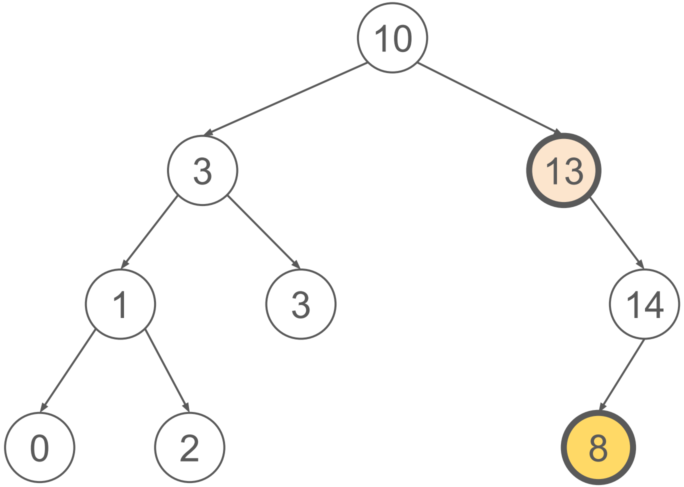
- Step 1: Find 8
- Step 2: Find the smallest element in the right subtree of 8 (10) and swap 8 with it
    - (you can choose left/right subtree in case of both exist)
- Step 3: Continue to swap 8 with the smallest element in the right subtree of 8 (13)
- Step 4: Delete 8 as it becomes a leaf



Binary Search Tree – Delete

Delete 8

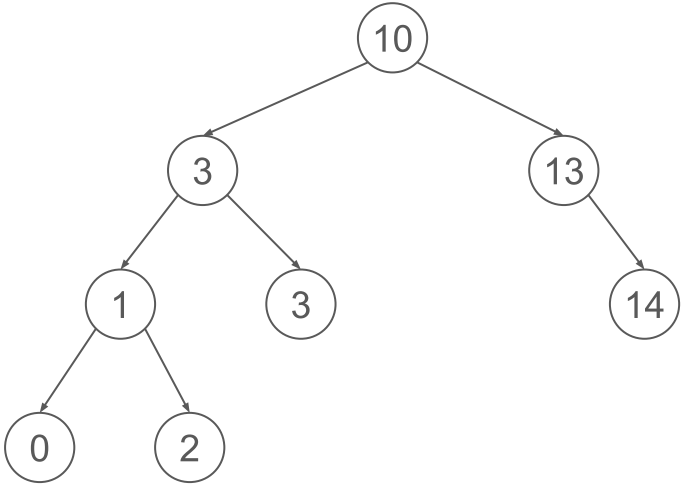
- Step 1: Find 8
- Step 2: Find the smallest element in the right subtree of 8 (10) and swap 8 with it
  - (you can choose left/right subtree in case of both exist)
- Step 3: Continue to swap 8 with the smallest element in the right subtree of 8 (13)
- Step 4: Delete 8 as it becomes a leaf



Binary Search Tree – Delete

Delete 8

- Step 1: Find 8
- Step 2: Find the smallest element in the right subtree of 8 (10) and swap 8 with it
    - (you can choose left/right subtree in case of both exist)
- Step 3: Continue to swap 8 with the smallest element in the right subtree of 8 (13)
- Step 4: Delete 8 as it becomes a leaf



Binary Search Tree - Time complexity

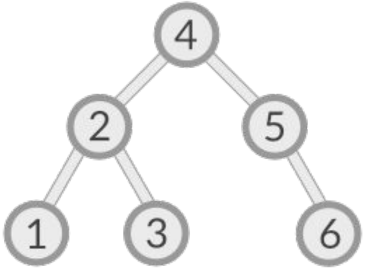
- In insert & query, we only need to DFS the tree from root to one of its node.
    - Time complexity = O(height of the BST)
- In delete, we may need to search multiple times, but we always walks downward. Hence,
    - Time complexity = O(height of the BST)

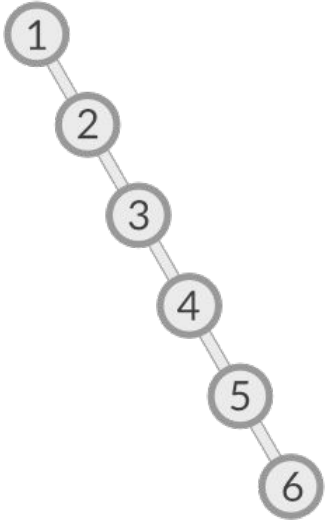
Binary Search Tree - Time complexity

- Time complexity = O(height of the BST)
  - On average a BST has a height of O(log N)
- In the worst case, the BST's height can be O(N), which degrades the operations' efficiency.
  - How?

Binary Search Tree - Time complexity

- The tree forms a chain due to its imbalance structure.





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, if it is suitable (Trie, segment tree)

Binary Search Tree - C++ Library

- **std::set** and **std::map** are implemented by red-black tree
  - **std::set**: key only
  - **std::map**: key used for ordering, value per each key
- Support insert, delete, query extrema, lower_bound, exact value operation
- However ranking operation is not supported.
  - If you code your own BST, you can support this by maintaining additional info like subtree sizes.

```cpp
#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

std::multiset / std::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
- std: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 std::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 std::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 - Programming in C++

BST Practice Task - HKOJ B103 Binary Search Tree

- https://judge.hkoi.org/task/B103
- We will not code this together during the lesson.
- Also note that this BST without self-balancing mechanism is not efficient enough. In general you should use C++ STL.

Binary Search Tree - HKOJ M0811 Alice's Bookshelf

Problem: Support the following 5 operations

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

Solution: BBST (std::multiset)

Problem 3

Support Q operations of the following types:

- **Insert(x):** Add an integer x to the data structure
- **Delete(x):** Delete an integer x from the data structure
- **Find(x):** Check whether the value x exists in the data structure

- These can be handled by a BBST in O(Q log Q)
- However, the task this time is to find a solution with time complexity O(Q)

Problem 3

Support Q operations of the following types:

- **Insert(x):** Add an integer x to the data structure
- **Delete(x):** Delete an integer x from the data structure
- **Find(x):** Check whether the value x exists in the data structure

- What is the main difference between this set of operations and the set of operations of Heap and BST?
- **Order is not required to maintain.**
  - **However, locating the element is still needed.**

Hash table

Hash table

- Operations supported:
    - Insert(x): O(1)
    - Delete(x): O(1)
    - Find(x): O(1)
- Additionally, we can treat the previous element as key, and store a value for each key.
    - Hash table can support editing value of a key directly in 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 - Insert

Insert 876

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

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

Hash table - Insert

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 - Delete

Delete 452

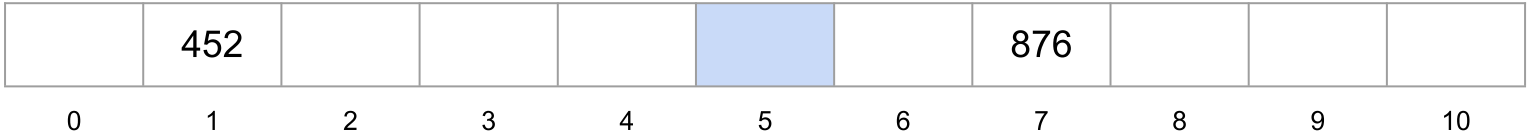
- 452 % 11 = 1
- Cell 1 contains 452
- Delete 452 from cell 1

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

Hash table - Find

Find 654

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



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

Hash table - Find

Find 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 - Find

Find 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 Collision

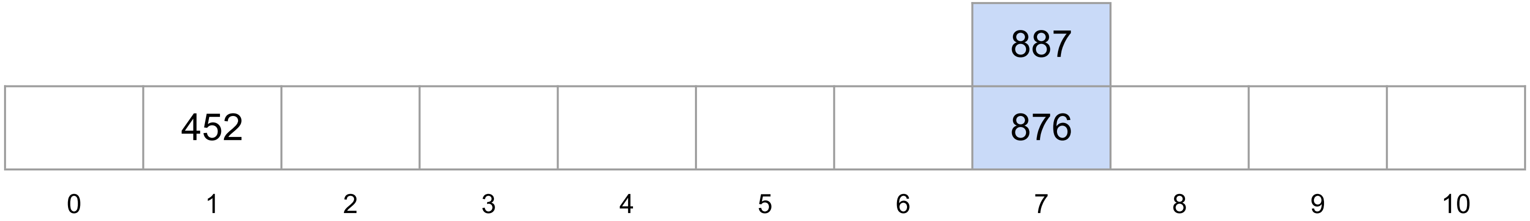
In the hash table example, we map all integer in the wide range of integers [0, 10^9], into an array of size 11.

- How can we fit that large amount of keys in such a small range?
- The answer is that we cannot.

Hash table - Collision

Insert 887

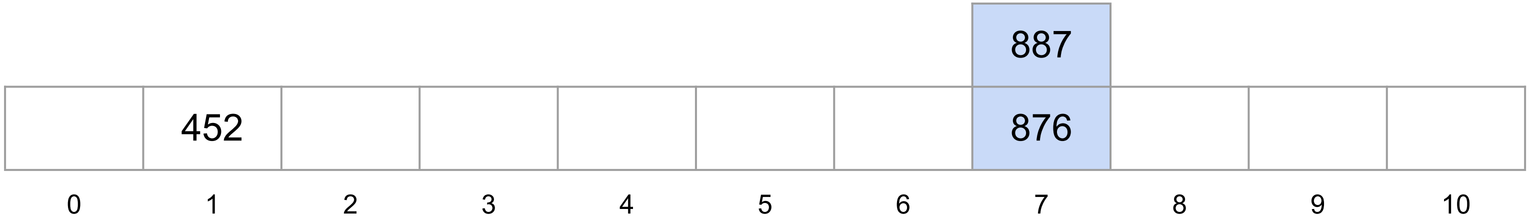
- 887 % 11 = 7
- Cell 7 is already occupied by 876
- Use array of vector instead of simple array – Separate chaining
- Store both 887 and 876 in cell 7



Hash table - Collision

Find 865

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



887

452 876

0 1 2 3 4 5 6 7 8 9 10

Hash table - Collision

There are many ways to handle collision

- Closed hashing
    - Linear probing
    - Quadratic probing
    - Double hashing
- Separate chaining

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

There is also a scenario that every insert will lead to collision - the table is full.

We can handle that by:

- Rehashing

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)

Hash table - Good Hash Function

Given a list of strings (consisting of ‘a’ - ‘z’ only), we want to fit them in a finite-sized array (size = n)

Some possible hash functions:

- [(s[0] - ‘a’) + (s[1] - ‘a’) + (s[2] - ‘a’) + … + (s[len(s) - 1] - ‘a’)] % n
- [(s[0] - ‘a’) + (s[1] - ‘a’) * 27 + (s[2] - ‘a’) * 272 … + (s[len(s) - 1] - ‘a’) * 27(len(s) - 1)] % n

Which one is better?

- Hint: compute the hash values of “listen” and “silent” using the above functions

- The hash value of “abcd” in the 2nd function is
    - $1 + 2 * 27 + 3 * 27^2 + 4 * 27^3 = 80974$

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 (this value is named **Load Factor**)
- Generally we can set m as around 1e6 (or comparable to n)
  - the expected number of values stored in each vector = 1
- Time Complexity: O(1) in general

Hash table - C++ Library

- std::unordered_map / std::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:
```

Hash table - Related topics

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

Hash Table Practice Task - HKOJ B101 Hash Table

- https://judge.hkoi.org/task/B101
- Let’s spend some time to code this task together.
- In contest time, there may be cases which this is useful.

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(x, y)** - 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(x)** - 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 - Naive Implementation (1): 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 (1)

Find operation: **find(u)**

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

Merge operation: **union(u, v)**

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

```
for (int i = 1; i <= n; i++)
    if (p[i] == p[v]) p[i] = p[u];
```

- Time complexity: O(N)

DSU - Naive Implementation (1)

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

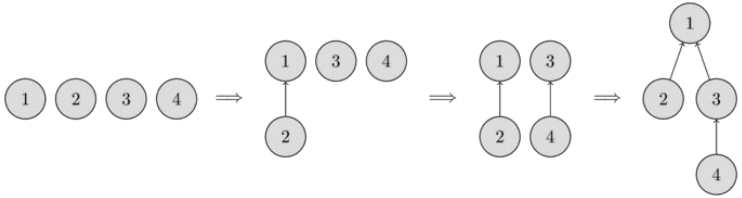
DSU - Naive Implementation (2): Representation

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 - Naive Implementation (2)

Find operation: **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

Merge operation: **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)

- It turns out that this implementation could be extended to produce a efficient-enough data structure.

DSU - Optimization

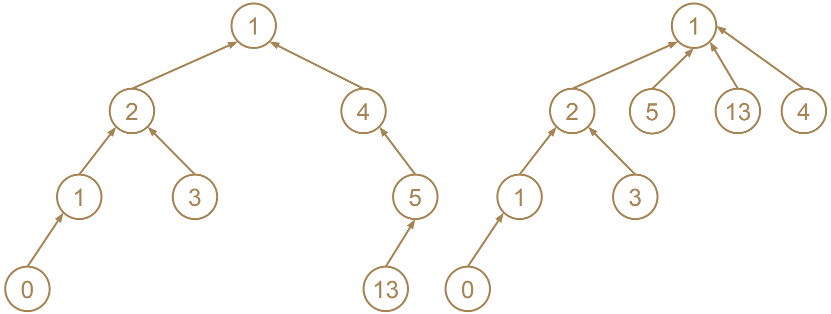
There are two well-known optimization for DSU

- **Path Compression**
  - Optimizing **find(u)** operation
  - **find(u)** will have amortized O(log N) time complexity
- **Union by Size**
  - Optimizing **union(u, v)** operation
  - **find(u)** 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, 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:

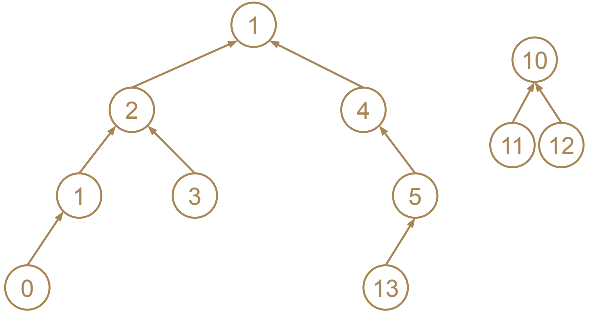
```
int find(int u) {
    return p[u] == u ? u : p[u] = find(p[u]);
}
```

DSU - Union by Size

We want to make the tree more balanced – to reduce number of step during **find(u)**

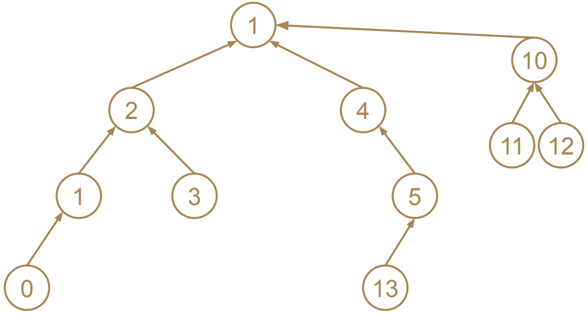
- Link the subtree with smaller size to that with larger size

```
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] = 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 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 Practice Task - HKOJ B102 Disjoint Set Union

- https://judge.hkoi.org/task/B102
- Let’s spend some time to code this task together.

DSU - NOI 2015 Day1 Q1 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 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 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”

- Exactly the task that could be solved using DSU

Common tricks

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

- Let’s revisit Problem 1
    - **Insert(x)**: Add an integer x to the data structure
    - **Delete(x)**: Delete an integer x from the data structure ← **Not support by heap**
    - **QueryMin()**: Find the minimum number of the data structure
- 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 to heap A
- Delete a number → push to heap B
- Every time, before an operation is executed.
    - While minimum of A == minimum of B, remove from both
- Such minimum of A should be **deleted already**, we just handling it a bit late.
    - Each element in B acts as a “Promise”. We will execute the deletion some time.

Lazy Deletion

Add 5
Add 2
Add 3
Query Min
Delete 3
Add 1
Delete 2
Add 3
Delete 1
Query Min

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

Lazy Deletion

Add 5

Add 2

Add 3

Query Min → 2

Delete 3

Add 1

Delete 2

Add 3

Delete 1

Query Min

Heap A: {2, 3, 5}

Heap B: {}

Lazy Deletion

Add 5
Add 2
Add 3
Query Min
Delete 3 → Add to B
Add 1
Delete 2
Add 3
Delete 1
Query Min

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

Lazy Deletion

Add 5
Add 2
Add 3
Query Min
Delete 3
Add 1
Delete 2
Add 3
Delete 1
Query Min

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

Lazy Deletion

Add 5

Add 2

Add 3

Query Min

Delete 3

Add 1

Delete 2 → Add to B

Add 3

Delete 1

Query Min

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

Heap B: {2, 3}

Lazy Deletion

Add 5
Add 2
Add 3
Query Min
Delete 3
Add 1
Delete 2
Add 3
Delete 1
Query Min

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

Lazy Deletion

Add 5

Add 2

Add 3

Query Min

Delete 3

Add 1

Delete 2

Add 3

Delete 1 → Add to B

Query Min

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

Heap B: {1, 2, 3}

Lazy Deletion

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

Same min in heap A and B
Heap A: {1, 2, 3, 3, 5}
Heap B: {1, 2, 3}
Erase 1 in both heap
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
Delete 3
Add 1
Delete 2
Add 3
Delete 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 & we need to 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(x)**: Add an integer x to the data structure
    - **Delete(x)**: Delete an integer x from the data structure
    - **Query()**: Find the k-th (where k is given 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++ STL BST (set or map)

Using 2 BSTs – Constant K-th element

- Use two std::maps (able to find max / find min / insert / delete)
- First std::map: always stores the smallest K-th element
  - Or all elements if # of elements < k
- Second std::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

Delete 3

Delete 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
Delete 3
Delete 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

Delete 3

Delete 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

Delete 3

Delete 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

Delete 3

Delete 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

Delete 3

Delete 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

Delete 3

Delete 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

Delete 3

Delete 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

Delete 3

Delete 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

- Variants 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

Heuristic Merging (Small to Large Merging)

- Problem: Each element is a heap with 1 element at the beginning. Need to support the following operations.
  - Heap Operations on a particular heap
    - **Insert(x):** Add an integer x to the data structure
    - **DeleteMin():** Delete the minimum number from the data structure
    - **QueryMin():** Find the minimum number of the data structure
  - Merge 2 heaps into 1 heap
    - **Merge(x, y)**: Merging heap x and heap y
    - Doing this naively will induce large transfer cost.

Heuristic Merging (Small to Large Merging)

- Merge 2 heaps into 1 heap
  - **Merge(x, y)**: Merging heap x and heap y
  - Doing this naively will induce large transfer cost.
- Keypoint: We always merge small heap into large heap.
  - Naively, pop everything out of the small heap and push into the large heap.
  - We call this technique **heuristic merging**.
- Calculate the time complexity of this small to large idea:
  - Each element will only be transferred when it is in the **SMALL side** of the merge.
  - Hence, each element can only be transferred O(log N) time. (Why?)
  - The total time complexity would be O(N log^2(N))
- This trick is very useful and appear everywhere.

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 → Programming in C++ / 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