NOI.PH Training: DS 2

More Data Structures

Kevin Charles Atienza and Rene Josiah Quinto

Contents

1 Introduction

This is the second data structures module, where we'll be tackling slightly more advanced data structures and techniques. Some of these are new, and some of these are variants of the data structures discussed in the previous data structures module.

2 More on Range Queries

2.1 More Advanced Sqrt Decomposition

In the previous data structures module, we've discussed how sqrt decomposition can be used to do point updates and range queries, all in $\mathcal{O}(\sqrt{n})$ and $\mathcal{O}(n)$ preprocessing. Sqrt decomposition is a very flexible algorithm/paradigm, and for this module we'll discuss more variants.

2.1.1 "Lazy" Sqrt Decomposition

Another way of solving range queries is to withhold updates up to a certain point instead of updating elements directly. As an example, let's look at the RSQ problem, with two operations **sum** and **inc**. If we solve this using a prefix sum, then we can preprocess in $\mathcal{O}(n)$ and solve each query in $\mathcal{O}(1)$.

This works perfectly if our data is static, but if we need to update the elements, then we have to preprocess again in $\mathcal{O}(n)$ before the next query. In the worst case, we have an update before each query, so each query has a worst case complexity of $\mathcal{O}(n)$.

To improve this, what we can do is to not directly add the updates, but instead append them to an array of updates. For this example, we can take note of how much was added to certain elements, and store them in an array. Then for each query, we can first get the range sum before any changes happened, then afterwards loop through each of the updates and check if we need to modify our answer. If an update corresponds to a certain index in our range, then we need to include the update to our answer. If we have u updates, then each query can be solved in $\mathcal{O}(u)$. If u isn't too large, then this can be viable.

However, in the case that u is large, then we can put a "capacity" to our array of updates. Let's say that the capacity of the update array is m. If the update array is full, then we have to "clear" our update array by implementing the updates directly to the problem array, then rerunning the preprocessing. For our example, updating the array is $\mathcal{O}(1)$ per update, and preprocessing takes $\mathcal{O}(n)$. And so, each update has an amortized complexity of $\mathcal{O}(n/m)$. Each query will then have a worst case complexity of $\mathcal{O}(m)$. If we have q queries, and u updates, then we'll have a total complexity of $\mathcal{O}(u \cdot (n/m) + q \cdot m)$. In several problems, we can consider $u = \Theta(q)$, so the total complexity becomes $\mathcal{O}(q \cdot (n/m + m))$. If we try to optimize this complexity through m, we find that the optimal m is $\sqrt{n}$, giving us a final total complexity of $\mathcal{O}(n + q\sqrt{n})$.

> **Exercise 2.1.** Where did the $\mathcal{O}(n)$ bit come from?

The implementation of the above example is shown below. Note that here, we update `A` immediately after every update query, but the prefix sum array `prefix_sums` is corrected only every m updates.

```
class LazySQRTDecompRSQ {
    vector<int> A;
    vector<pair<int,int>> updates;
    vector<int> prefix_sums;
    int m;

public:
    LazySQRTDecompRSQ() {}
```

```
    LazySQRTDecompRSQ(vector<int> &A): A(A), prefix_sums(A.size() + 1) {
        m = int(sqrt(A.size()));
        updates.reserve(m);
        compute_prefix_sums();
    }

    void compute_prefix_sums() {
        for (int i = 0; i < A.size(); i++) {
            prefix_sums[i + 1] = prefix_sums[i] + A[i];
        }
    }

    void inc(int p, int x) {
        if(updates.size() >= m) {  // rebuild everything
            updates.clear();
            compute_prefix_sums();
        }
        A[p] += x;
        updates.push_back({p, x});
    }

    int sum(int a, int b) {
        int total = prefix_sums[b + 1] - prefix_sums[a];
        for (auto& upd: updates) {  // adjust using the list of updates
            if(a <= upd.first && upd.first <= b)
                total += upd.second;
        }
        return total;
    }
};
```

Depending on the problem, we can even lazily update entire ranges (consecutive elements) instead of single elements. One such problem is the RSQ problem, but our update function **inc**(a,b,x) adds a certain value x to all elements from a to b.

> **Exercise 2.2.** Explain how to use lazy sqrt decomposition to handle range **inc**s and range **sum**s in $\mathcal{O}(n + q\sqrt{n})$ time.

2.1.2 Offline Sqrt Decomposition

Let's consider the *static* variant of the problem again. We are only given a bunch of **sum** queries on an unchanging array, and we are tasked to answer them all.

In solving a problem with static range queries, another way to approach it is to take all of the queries first, then provide an answer to all of them at the same time. With this approach, we can solve the queries in an order that is different than the one given. The idea is to make use of the answer of one query to solve another.[1]

Let's denote the range of each query i as $[l_i, r_i]$. First, we want to group them in a way such that all l_is in a group are close enough, say at most a distance m (we'll see later why). One way to do this is to create "buckets" with ranges $[0, m), [m, 2m), [2m, 3m), \ldots$, and we assign each query to one of these buckets according to which bucket their l_i values fall. Then in each bucket, we sort the queries by r_i.

[1] This is also known in the community as Mo's Algorithm, but we prefer to call it *offline sqrt decomposition* or maybe even *sqrt decomposition on queries*.

For example, if we have an array $A = [4, 5, 2, -1, 3, 9, 4]$, and the queries

$$Q = [(0, 3), (4, 5), (6, 6), (1, 2), (5, 6)] .$$

For this example, let's set m to 3. Then in the first bucket, we have the queries $Q_0 = [(1, 2), (0, 3)]$, and in the rest of the buckets we have $Q_1 = [(4, 5), (5, 6)]$ and $Q_2 = [(6, 6)]$. Note that each bucket has already been sorted by r_i.

After this, we can solve our queries by bucket. To solve the queries in a bucket, we start with solving the first query. This will take us $\mathcal{O}(n)$ time. Then for the next query, we make use of our previous answer by adjusting it to fit the range of the query. In the case of the first bucket in our example, our first query is $(1, 2)$ with the answer of 7. To get the answer of the next query $(0, 3)$, we add/subtract the elements concerned – in this case, we add A_1 and A_3. After the last query in the bucket is solved, move on to the next bucket and repeat.

For each bucket, the first query will take $\mathcal{O}(n)$. For the rest, we first observe that l changes at most m for each query, since the l of each query in a bucket is bounded. Since the queries in each bucket are sorted by r, the total number of updates to r is at most n for an entire bucket. Taking account that there are around n/m buckets, the first queries of each bucket contributes $\mathcal{O}(n \cdot (n/m))$ to the total algorithm complexity, and other queries contribute $\mathcal{O}(q \cdot m + n \cdot (n/m))$. This gives us a total complexity of $\mathcal{O}(q \cdot m + n^2/m)$.

Usually, it's fine to choose $m = \Theta(\sqrt{n})$[2] which leaves us a total complexity of $\mathcal{O}((n+q)\sqrt{n})$. If we want to be strict with our complexity analysis, the best choice would be choosing $m = \Theta(\max(1, n/\sqrt{q}))$, leaving us a total complexity of $\mathcal{O}(q + n\sqrt{q})$.[3]

An implementation of the above algorithm is shown below. Note that for this implementation, m is set to around $\sqrt{n}$. Also, since we rearrange the order of the queries for our algorithm, we need to have a reverse mapping to keep the original indices.

```
struct Query {
    int i, j, idx;
};

vector<int> range_sums(vector<int> A, vector<pair<int,int>> Q) {
    // inputs a list of queries, returns RSQ answer for each
    int n = A.size();
    int m = int(sqrt(n));

    vector<int> ans(Q.size());

    // group the queries to their corresponding "blocks" or "buckets"
    vector<vector<Query>> qs(n / m + 1);
    for(int i = 0; i < Q.size(); i++) {
        int bin = Q[i].first / m;
        qs[bin].push_back({Q[i].first, Q[i].second, i}); // remember the original index.
    }

    for (int bin = 0; bin < qs.size(); bin++) {
        vector<Query>& block = qs[bin];

        // sort by j
        sort(block.begin(), block.end(), [](const Query &a, const Query &b) {
            return a.j < b.j;
        });
```

[2] That is to say, a choice of m of the form $m = \sqrt{n} \cdot b + c$, where b and c are constants.

[3] This is not as easy to remember, and if we do forget and if we don't want to go through all the complexity analysis again, we can take comfort in the fact that $m = \Theta(\sqrt{n})$ is usually fine.

```
26
27          // answer the queries
28          int l = bin*m, r = (bin+1)*m;
29          int total = 0, pos = r;
30          for (Query& q: block) {
31              assert(q.i >= l);  // sanity check
32
33              if (q.j < r) {  // it's a small query, so just do manually
34                  ans[q.idx] = 0;
35                  for (int k = q.i; k <= q.j; k++) ans[q.idx] += A[k];
36              } else {
37                  while (pos <= q.j) total += A[pos++];  // move the current range rightward
38
39                  // add the "leftovers" at the left
40                  ans[q.idx] = total;
41                  for (int k = q.i; k < r; k++) ans[q.idx] += A[k];
42              }
43          }
44      }
45
46      return ans;
47  }
```

Finally, though we've used the RSQ problem as an example, we can support any operation as long as it follows certain properties. Let's denote $f(l, r)$ as the answer to the query (l, r). The three following properties must hold:

1. $f(l+1, r)$ can be efficiently derived from $f(l, r)$.
2. $f(l, r+1)$ can be efficiently derived from $f(l, r)$.
3. $f(l-1, r)$ can be efficiently derived from $f(l, r)$. (In some cases, this can be omitted.)

Exercise 2.3. Use offline sqrt decomposition to answer q queries of the following kind (on an array with n elements):

- **.distinct(i,j)**. Find the number of *distinct values* of the array from index i to index j.

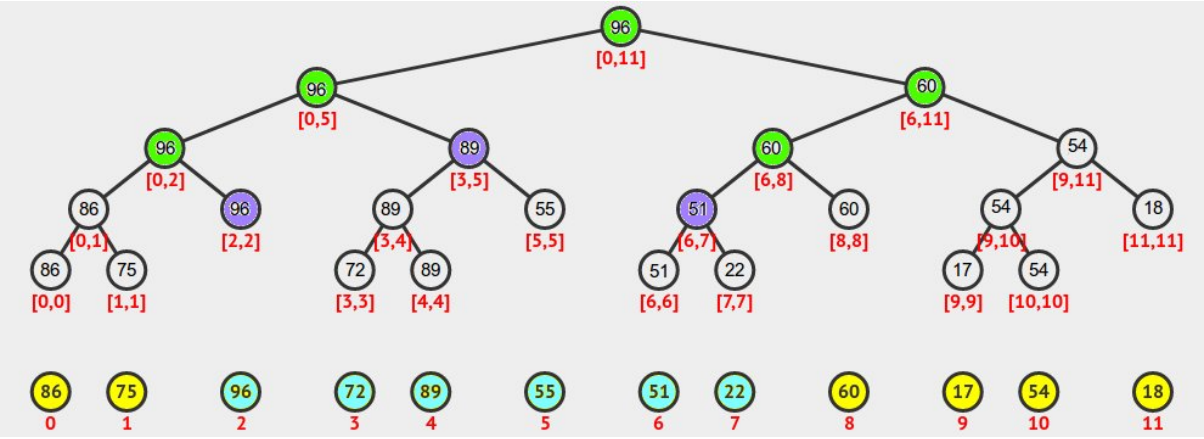
  Feel free to use `set`s.

Here, the word *efficiently* is used somewhat loosely – ideally, these operations should be $\mathcal{O}(1)$, but depending on the problem, a higher complexity can be allowed. Since these operations are the core of our algorithm, the total complexity scales proportionally. If these operations have a worst case complexity of $\mathcal{O}(T(n))$, then the total complexity is $\mathcal{O}(T(n)(n+q)\sqrt{n})$ when $m = \sqrt{n}$ is chosen.

2.2 Segment Tree with Lazy Propagation

In the previous Data Structures module, we've shown that a segment tree can be used to update point values and query ranges. Point update and range queries can both be done in $\mathcal{O}(\lg n)$. However, if we want to update ranges (for example, increase all elements in a certain range by a value x), updating them one-by-one takes $\mathcal{O}(n \lg n)$ time, which is very slow. Even if we try to update them all at the same time, in the worst case we'll have to modify all leaves, which means we'll visit $\mathcal{O}(n)$ nodes.

An image of a segment tree implementing a range maximum query is shown below.



If we're trying to query the range $[2, 7]$, then we'll be traversing down the segment tree until the current node's range is completely inside the query range (marked purple). We stop at these nodes, and return their values. It can be shown that the number of nodes visited is $\mathcal{O}(\lg n)$; it was given as an exercise in the previous Data Structures module. What if we do something similar to our range updates, i.e., stop when we've reached nodes that are completely inside the range? If we just simply update those nodes and not update its children, then we'll have a problem when we have queries that have ranges that only partially covers the range of this node (and so, have to go down to its children).

The solution for this is to mark these nodes as having a *pending update*. Then, every time we visit a node with a pending update, we "propagate" the update down to its two children just in case we have to go down.[4] In this way, we can reduce the update to $\mathcal{O}(\lg n)$ time, but we add some more operations due to propagation. We can prove that these extra operations are also $\mathcal{O}(\lg n)$ per query/update – for each update/query, we visit $\mathcal{O}(\lg n)$ nodes. In the worst case, every node we visit has a pending update, and so we propagate from each visited node. Since we propagate the pending update to at most 2 nodes per node visited, the number of extra operations is also $\mathcal{O}(\lg n)$, and so all updates and queries (point or range) have $\mathcal{O}(\lg n)$ complexity.

This hinges on the important assumption that pending updates can be combined at the internal nodes. For example, let's say that the nodes with range $[3, 5]$ and $[0, 5]$ have pending updates. If we visit the node with range $[0, 5]$, then the update will propagate to $[3, 5]$, and so we need to be able to combine these updates efficiently.[5] In our example, our update adds a constant value to all elements within a certain range. Combining two updates on the same range can then be done in $\mathcal{O}(1)$ (if one update adds x to a node, and another adds y, the combined update is $x + y$), and so we won't have any problems here.

2.2.1 Implementation

The implementation of the above example (with range **max**s and **inc**s) is shown below. Here, we use a helper function `propagate` to implement the update on the current node and to propagate the update to its two children.

[4] This is also where the term "lazy" comes in; we only propagate the updates when we need to. If, for example, we have an update that covers the range $[0, 11]$ in our example, then we'll only need to update one node. After this, when we query, the updates are propagated, but only on those nodes that are concerned.

[5] Ideally $\mathcal{O}(1)$. Otherwise, the total complexity of the algorithm scales with this complexity. This property can also be the deciding factor on whether a segment tree is a viable solution or not.

```cpp
typedef long long ll;
const ll INF = 1LL << 60;

class SegmentTree {
private:
    int i, j;
    ll _max, to_add = 0;
    SegmentTree *left, *right;

    void propagate() {
        if (!to_add) return;

        _max += to_add;
        if (i < j) {  // combine updates in children
            left->to_add += to_add;
            right->to_add += to_add;
        }
        to_add = 0;
    }

    void combine() {
        _max = max(left->_max, right->_max);
    }

public:

    SegmentTree(vector<ll> &A): SegmentTree(A, 0, A.size() - 1) {}

    SegmentTree(vector<ll> &A, int i, int j): i(i), j(j) {
        if (i == j) {  // leaf
            _max = A[i];
            left = right = nullptr;
        } else {
            int k = (i + j) / 2;
            left = new SegmentTree(A, i, k);
            right = new SegmentTree(A, k + 1, j);
            combine();
        }
    }

    void inc(int I, int J, ll v) {
        propagate();
        if (I <= i && j <= J) {  // completely contained
            to_add += v;
            propagate();
        } else if (J < i || j < I) {   // disjoint
            /* do nothing */
        } else {  // partially overlap
            left->inc(I, J, v);
            right->inc(I, J, v);
            combine();
        }
    }

    ll max(int I, int J) {
        propagate();
        if (I <= i && j <= J) {  // completely contained
            return _max;
        } else if (J < i || j < I) {  // disjoint
            return -INF;
        } else {  // partially overlap
```

```
62              return std::max(left->max(I, J), right->max(I, J));
63          }
64      }
65  };
```

It is important here that `propagate()` is called before anything is even done in a node, including on queries. This is to ensure that the current node contains the most up-to-date values.

3 Ancestor Queries on Rooted Trees

3.1 Lowest Common Ancestor

Given a rooted tree, the **lowest common ancestor** (LCA) of two nodes is the lowest node (i.e., largest-depth node) that contain both nodes in its subtree.[6] There are several reasons as to why we would be interested in this node. For one, the LCA will always be one of the nodes in the path between the two nodes. If we're trying to find the distance between two nodes in the rooted tree, then the problem can then be reduced to finding the LCA and getting each node's distance to this LCA.

> **Exercise 3.1.** If x is a node and a is one of its ancestors, describe the distance between x and a in terms of their depths.

Note that for the following implementations, the input is assumed to be a tree that is rooted at node 0, and is represented by an adjacency list.

3.1.1 Naive LCA

A naive way of getting the LCA of two nodes a and b is to do a DFS, then report the path from the root to a and b, and the LCA is the last node in the common prefix of both paths. In the worst case, it's $\mathcal{O}(n)$, since we're traversing the tree for each query.

Another naive method is to gradually traverse up the tree until you find the common ancestor. To do this, we first keep track of the depth of all nodes. Next, assuming without loss of generality that a is further down the tree than b in terms of depth, we go up to the parent of a and continue until we find the node with depth equal to the depth of b, and let's denote this node as a'. Finally, we simultaneously go up from a' and b until they refer to the same node. This is the LCA of a and b. In the worst case, we traverse the entire height of the tree, and so the worst case complexity is $\mathcal{O}(h)$ where h is the height of the tree. Depending on the given tree, this can still be $\mathcal{O}(n)$, but it has a lot less operations than the previous solution, and is simpler to code.

Below is an implementation of the above algorithm.

```
class LCANaiveRMQ {
private:
    vector<int> parent, depth;

    void dfs_init(vector<int> adj[], int u, int parent_, int depth_) {
        depth[u] = depth_;
        parent[u] = parent_;

        for (int v : adj[u])
            if (v != parent_)
                dfs_init(adj, v, u, depth_+1);
    }

public:
    LCANaiveRMQ() {}
    LCANaiveRMQ(vector<int> adj[], int n): parent(n), depth(n) {
        dfs_init(adj, 0, 0 ,0);
```

[6] Note that the LCA can even be one of the two nodes.

```
    }

    int ascend(int i, int d) {  // climb d steps from node i
        while (d --> 0) i = parent[i];
        return i;
    }

    int lca(int a, int b) {
        // WLOG assume a is deeper
        if (depth[a] < depth[b]) swap(a, b);

        // bring a to the same height as b
        a = ascend(a, depth[a] - depth[b]);

        // simultaneously climb until they are on the same node
        while (a != b) {
            a = parent[a];
            b = parent[b];
        }
        return a;
    }
};
```

Our implementations will assume that the root's parent is the root itself, so climbing any number of steps is well-defined.

3.1.2 Square Root Preprocessing (Jump Pointers)

The previous algorithm can be slow since we're traversing up a tree one node at a time. What if we keep track of a certain ancestor with a specific distance from the node, say a distance of m? It can potentially speed up our climbing, since we can skip a couple of nodes.

If we follow the previous process, we have two stages in our algorithm. The first stage is ensuring that the two nodes have the same depth. We do this by ascending from the lower node until they have the same depth, which is $\mathcal{O}(h)$. What if we tried to climb up m nodes at a time, and at the point where climbing another m nodes would exceed the target depth, we climb up one at a time? Then the complexity of this step is reduced to $\mathcal{O}(h/m+m)$, which is potentially lower than our previous solution if m was chosen correctly.

The second stage is when we simultaneously go up each node, until they refer to the same node. If, again, we're to climb m nodes at a time, the complexity can be reduced to $\mathcal{O}(h/m+m)$, using the same reasoning as the first stage. For this stage, we need to jump m nodes at a time until right before they point to the same node, then climb one at a time. This gives us a total complexity of $\mathcal{O}(h/m+m)$ for each query. Here, the optimal m is $\sqrt{h}$, and so gives us a final time complexity of $\mathcal{O}(\sqrt{h})$ for each query.

Preprocessing each node's jump pointer to go up $\sqrt{h}$ times can be computed in $\mathcal{O}(n\sqrt{n})$. An implementation of the above algorithm is shown below.

```
class SqrtJumpPointers {
private:
    vector<int> parent, depth, jump;
    int m;

    void dfs_init(vector<int> adj[], int u, int parent_, int depth_) {
        ... // same as before
    }
```

```
9
10      void dfs_jump(vector<int> adj[], int u) {
11          jump[u] = j;
12          for (int it = 0; it < m; it++) jump[u] = parent[jump[u]];
13
14          for (int v : adj[u])
15              if (v != parent[u])
16                  dfs_jump(adj, v);
17      }
18
19  public:
20      SqrtJumpPointers(){}
21      SqrtJumpPointers(vector<int> adj[], int n): parent(n), depth(n), jump(n) {
22          dfs_init(adj, 0, 0, 0);
23
24          // get the height of the tree
25          int h = 0;
26          for (int d : depth) h = max(h, d);
27
28          m = int(sqrt(h));
29          dfs_jump(adj, 0);
30      }
31
32      int ascend(int i, int d) {
33          for (; d >= m; d -= m) i = jump[i];
34          for (; d >= 1; d -= 1) i = parent[i];
35          return i;
36      }
37
38      int lca(int a, int b) {
39          if (depth[a] < depth[b]) swap(a, b);
40
41          a = ascend(a, depth[a] - depth[b]);
42
43          while (jump[a] != jump[b]) {
44              a = jump[a];
45              b = jump[b];
46          }
47          while (a != b) {
48              a = parent[a];
49              b = parent[b];
50          }
51          return a;
52      }
53  };
```

Exercise 3.2. Show how to compute the mth ancestor of all nodes in $\mathcal{O}(n)$ time (for any fixed m).

3.1.3 Binary Lifting (Jump Pointers)

If we use the same principle, but keep track of multiple jump pointers instead of just one, then we can be more precise with our ascent. If we just select our jump sizes correctly, we can potentially speed up climbing significantly. However, we can't just store *all* ancestors since that would be too many. Instead, we must strike a balance between the number of jump pointers and the speed of climbing.

We can make use of the fact that any positive integer n can be expressed as the sum of

around $\lg n$ powers of two, and have jump pointers that jump nodes in powers of two. More formally, let $f(n, k)$ be the ancestor of node n with distance of 2^k. In the first stage, we know the exact number of nodes we need to climb in order to reach equal depths of the two nodes a and b. Let's denote this number as d. Here, we can use $f(n, k)$ to climb precisely d steps in $\mathcal{O}(\lg h)$. To do this, we can start with a large enough k, and end at $k = 0$. For each k, we check if $2^k \leq d$, and if it is, we jump to $f(n, k)$ and subtrack 2^k from d. For more clarity, refer to the implementation below.

For the second stage, a and b now have the same depths. This time, we need to look for the depth *right before* a and b converges to the same node. We can do this using a similar method as mentioned in the first stage. Once we're done with this, we can simply move up one step to get our LCA. This stage also runs in $\mathcal{O}(\lg h)$ time, and so the total time complexity of this algorithm is $\mathcal{O}(\lg h)$.

In the worst case, h is around n, and so each LCA query is $\mathcal{O}(\lg n)$. In order to simplify everything, we can opt to skip getting the height of the tree, and simply assume the worst case height of n. Though it has a higher complexity, it's somewhat insignificant, and we can still see large speedups when h is large (where it matters).

Initialization can be done in $\mathcal{O}(n \lg n)$ time using the property $f(n, k) = f(f(n, k-1), k-1)$.

An implementation of the above algorithm is shown below. In fact, this implementation hardcodes the maximum possible $\lg n$ to be 20. For larger n, just update this constant. Also, note that n and k are reversed.

```cpp
const int K = 20; // assume n < 2^l

class BinaryLifting {
private:
    vector<int> parent, depth;
    vector<vector<int>> jump;

    void dfs_init(vector<int> adj[], int u, int parent, int depth_) {
        ... // same as before
    }

public:
    BinaryLifting() {}
    BinaryLifting(vector<int> adj[], int n): parent(n), depth(n),
            jump(K+1, vector<int>(n)) {
        dfs_init(adj, 0, 0, 0);

        // compute jump pointers
        jump[0] = parent;
        for (int k = 1; k <= K; k++) {  // compute 2^k jumps from 2^(k-1) jumps
            for (int i = 0; i < n; i++) {
                jump[k][i] = jump[k-1][jump[k-1][i]];
            }
        }
    }

    int ascend(int i, int d) {
        for (int k = K; k >= 0; k--) {
            while (d >= (1 << k)) {
                d -= 1 << k;
                i = jump[k][i];
            }
        }
        assert(d == 0);  // verify that we have climbed exactly d steps
```

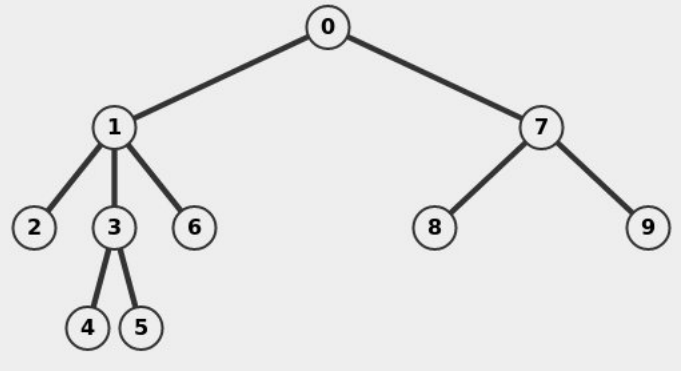
```
35          return i;
36      }
37
38      int lca(int a, int b) {
39          if (depth[a] < depth[b]) swap(a, b);
40
41          a = ascend(a, depth[a] - depth[b]);
42
43          // power-of-two climbs
44          for (int k = K; k >= 0; k--) {
45              while (jump[k][a] != jump[k][b]) {
46                  a = jump[k][a];
47                  b = jump[k][b];
48              }
49          }
50          // final climb
51          while (a != b) {
52              a = parent[a];
53              b = parent[b];
54          }
55          return a;
56      }
57  };
```

Exercise 3.3. What happens in the code above if n is much greater than 2^K?

Exercise 3.4. What happens if we replace **while** with **if** in lines 29 and 45?

3.1.4 LCA through RMQ

Another method of finding the LCA efficiently is to reduce the LCA problem to a simple RMQ problem, which we can solve using our data structure of choice. To do this, we first label each node in the order we visit them in DFS. Such a labeling is shown in the image below.



Next, we keep a list, and for each time we enter and return to a certain node in our DFS, we append the node's label to the end of the list. For example, once we enter node 1, we append "1" to the list, then we enter node 2, and we append "2" to the list, then we go back to node 1 and append "1" to the list again, and so on. For the image above, the final list would be "0121343531610787970". Since for each edge we visit, we append 2 elements to the list, and additionally another at the very start (root), we have exactly $2n - 1$ elements in this list. Let's

call this list the *visit order*.[7]

Now that we have the visit order, we also have to take note when each of the nodes first show up in the list. For example, node 0 showed up at the very beginning (index 0), and node 3 first showed up at index 4. Let's call the index associated with each node the *first visit* of that node.

Finally, to get the LCA of two nodes a and b, we simply have to get the minimum element within the visit order between the first visits of a and b. For example, if we're trying to find the LCA between node 4 and 6, we first get their first visits (index 5 and 10, respectively), then we get the minimum element between (and including) these indices. In this case, it's 1, and that's our LCA.

How it works is this: In the any subtree, the root of that subtree has the lowest label. In other words, any ancestor of any node p has a label that is lower than the label of p. In following the visit order between two nodes a and b, we will also be visiting their LCA as it is certain that it's one of the nodes in the path between a and b. You'll also be visiting other nodes during the DFS traversal, but these other nodes have labels that are higher than the LCA's (DFS traversals exhaustively visit all nodes under a subtree before exiting, and so each node visited between visited before exiting the LCA is part of LCA's subtree), and so the the minimum label between the first visits of a and b is the LCA.

Since we're working on range minimum queries on static elements, we can use a sparse table for $\mathcal{O}(1)$ LCA queries. One such implementation is shown below. Since we produce our own labeling for the algorithm, the LCA query returns the LCA using the original indexing of the nodes by keeping a map to reverse the label. Initialization is $\mathcal{O}(n \lg n)$ due to the sparse table, and each query is $\mathcal{O}(1)$.

```
class SparseTable {
    ... // same as before
};

class LCA_RMQ {
private:
    vector<int> vis_ord, first_vis, label, rev_label;
    int time = 0;

    SparseTable rmq;

    void dfs_init(vector<int> adj[], int u, int parent_) {
        // first time visiting this node
        label[u] = time++;
        rev_label[label[u]] = u;
        first_vis[label[u]] = vis_ord.size();
        vis_ord.push_back(label[u]);

        for (int v : adj[u]) {
            if (v != parent_) {
                dfs_init(adj, v, u);
                vis_ord.push_back(label[u]);  // push label[u] again after every subtree
            }
        }
    }

public:
    LCA_RMQ() {}
```

[7] This is also known as an Euler Tour Tree. For more info, see this link.

```
    LCA_RMQ(vector<int> adj[], int n): vis_ord(2*n-1), label(n), rev_label(n) {
        first_vis.resize(n);
        dfs_init(adj, 0, 0);
        rmq = SparseTable(vis_ord);
    }

    int lca(int a, int b) {
        int avis = first_vis[label[a]], bvis = first_vis[label[b]];

        if (avis > bvis) swap(avis, bvis);

        int lcavis = rmq.min(avis, bvis);
        return rev_label[lcavis];
    }
};
```

Here, we chose to use a sparse table to solve the RMQ, but we can use other data structures, such as segment trees. They have a higher complexity with a $\lg n$ factor in querying, but they can be more comfortable to code for some. Most of the time, that log factor isn't crucial, and so we can sacrifice a little efficiency for comfort.[8]

Exercise 3.5. In the method above, we used the list of *first visit times* as the array in which we do our range minimum queries on.

Explain how we can use the list of *heights* instead, to solve LCA through RMQ.

Hint: You may need a modified **.min** query that returns the *index* of the minimum element, not just the minimum element itself.

3.2 Level Ancestor Problem

Given a rooted tree and an arbitrary node u, the level ancestor problem asks for the ancestor of node u at a certain depth. More formally, let $LA(u, d)$ denote the node v that is an ancestor of u whose depth is d. This is equivalent to finding the sth ancestor of any node u, assuming we have precomputed all depths, because we can compute s as $\text{depth}(u) - d$.

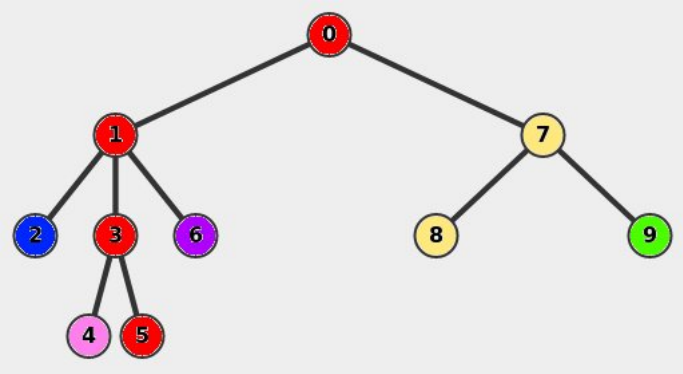
If you've read the LCA section, then this should be familiar – in both the naive and binary lifting solution of LCA, we've basically answered this problem in the first stage. To recall, the naive solution is simply to ascend up one node at a time until you're at the target depth, and the binary lifting solution makes use of jump pointers pointing to the 2^i'th ancestor, and so we can make fast and precise ascents. Those two are valid solutions of the level ancestors problem, the naive solution having a time complexity of $\mathcal{O}(n)$, and the binary lifting solution having a preprocessing time complexity of $\mathcal{O}(n \lg n)$ and query time complexity of $\mathcal{O}(\lg n)$. This seems fast enough for a lot of problems that may require it, such as the LCA problem, but if we somehow need a faster algorithm, we can solve each query in $\mathcal{O}(1)$ by making use of the Ladder Algorithm.

3.2.1 Ladder Algorithm

The ladder algorithm decomposes our rooted tree into a collection of paths. For a given tree, it takes out the longest path from the root to any leaf, and the remaining subtrees will be

[8] If your problem consists of a lot of LCA queries, and the bounds are tight with respect to that, you might want to consider shaving off that log factor just to be sure.

decomposed in the same way, recursively. An example of a valid path decomposition is shown in the image below, where each unique color denotes nodes in the same path.



After this, we'll have a collection of m paths, where each node in the rooted tree belongs to exactly one path. Let's denote the path ID of a certain node u to be $Path(u)$.

By storing each path as an array, we can easily climb to any height in $\mathcal{O}(1)$ with a single lookup in the (appropriate) path array, as long as the target depth is contained in the path as well. However, obviously, not all target nodes belong to the path.

To fix this, for each path, we will extend the length of the path up to twice the original length by adding the ancestors above it. This will result in $2n$ nodes in all of the paths combined. This will also mean that some nodes will belong to several paths, but note that $Path(u)$ is still unique, as it was defined before the extension.

With this, we can solve $LA(u, d)$ by getting $Path(u)$, then using the path to go to the required depth. If we fall short, we instead move to the node at the end of the path, which we denote as v, and repeat from here. Note that since we extended our paths, then $Path(v) \neq Path(u)$. It can be shown that if the node u has height h[9], then v has height at least $2h$ due to the path extension, and how the paths were decomposed.[10] Then the next step will bring us to a height at least $4h$ and so on. In other words, each step starts off slow, and the jumps increase exponentially. With all this, getting $LA(u, d)$ can be done in $\mathcal{O}(\lg n)$ time.

This alone does not provide us an $\mathcal{O}(1)$ solution, but recall that in contrast to the Ladder Algorithm's exponentially increasing jumps, Binary Lifting provides exponentially decreasing jumps. In other words, it starts off with a large step, then each step decreases exponentially until we reach our destination. If we only do the first step of Binary Lifting, it will always bring us to a node m that is at least half of the way to our destination. And since m has a height that is at least half of the height of the destination node, then we can get to the destination from $Path(m)$ in $\mathcal{O}(1)$. In other words, by combining a single step from both Binary Lifting and the Ladder Algorithm, we can solve $LA(u, d)$ in $\mathcal{O}(1)$!

An implementation of the above algorithm is shown below. Note that the input is assumed to be a tree that is rooted at node 0, and is represented by an adjacency list. In the implementation, each path is labeled from the leaf to the root, and so a `deque` was used to represent each path to support `push_front` when we're inserting nodes as we go down the tree. The long path decomposition[11] is done by choosing the child with the largest height as the next node in the

[9] Here, we denote the height of a node as the height of its subtree.

[10] Since we are always taking the longest path within a tree, the length of each path will be at least as large as the height of any node in the path. Therefore, if a node n in a path has height h, it is guaranteed that the path it belongs to has length at least $2h$ after extension.

[11] Not to be confused with Heavy-Light Decomposition. Long Path Decomposition, which was used here, prioritizes the longest path in choosing between children. In Heavy-Light Decomposition, the *heaviest* (most number of nodes in subtree) child is chosen.

path.

```cpp
const int K = 20;

class LevelAncestor {
private:
    vector<int> height, depth, parent, lg;
    vector<vector<int>> jump;

    vector<deque<int>> paths;  // deque so we can push front
    vector<int> path_idx, path_pos;

    void dfs_init(vector<int> adj[], int u, int parent_, int depth_) {
        depth[u] = depth_;
        parent[u] = parent_;

        for (int v : adj[u])
            if (v != parent_) {
                dfs_init(adj, v, u, depth_+1);
                height[u] = max(height[u], height[v]+1);  // compute heights as well
            }
    }

    void get_long_paths(vector<int> adj[], int u) {
        paths.back().push_front(u);
        path_idx[u] = paths.size()-1;

        // find the child with the largest height, and continue the long path
        int next_node = -1;
        for (int v : adj[u]) {
            if (v != parent[u]) {
                if (next_node == -1 || height[next_node] < height[v]) {
                    next_node = v;
                }
            }
        }
        if (next_node != -1) {
            get_long_paths(adj, next_node, u);
        }

        // create paths for the rest of the children
        for (int v : adj[u]) {
            if (v != parent[u] && v != next_node) {
                paths.emplace_back();  // create a new path
                get_long_paths(adj, v);
            }
        }
    }

public:
    LevelAncestor() {}
    LevelAncestor(vector<int> adj[], int n): parent(n), depth(n),
            path_idx(n), path_pos(n), jump(K+1, vector<int>(n)), paths(1), lg(n+1) {

        dfs_init(adj, 0, 0, 0);  // initialize height, depth, parent

        ... // compute jump pointers here (same as before)

        get_long_paths(adj, 0);  // detect the long paths

        for (auto& path : paths) {
```

```
            for (int i = 0; i < path.size(); i++) {  // compute path_pos
                path_pos[path[i]] = i;
            }
            for (int h = path.size(); h > 0; h--) {  // extend paths
                path.push_back(parent[path.back()]);
            }
        }

        // precompute lg
        for (int i = 2; i <= n; i++) lg[i] = lg[i >> 1] + 1;
    }

    int level_anc(int u, int d) {  // find ancestor of u with depth d
        d = depth[u] - d; // convert to 'climb count'

        if (d < 0) throw "invalid arguments";
        if (d == 0) return u;  // already here

        // single jump with jump pointers
        u = jump[lg[d]][u];
        d -= 1 << lg[d];

        // single climb with ladder
        return paths[path_idx[u]][path_pos[u] + d];
    }
};
```

The preprocessing takes $\mathcal{O}(n \lg n)$, dominated by the computation of the jump pointers. The query time is $\mathcal{O}(1)$.

3.3 The theoretically best algorithm

So far, we have looked at two different but closely related problems about ancestors and climbing in rooted trees. Our best algorithms so far can be summarized as follows:

LCA	Preprocessing	Query
Binary lifting	$\mathcal{O}(n \lg n)$	$\mathcal{O}(\lg n)$
LCA with RMQ (segment trees)	$\mathcal{O}(n)$	$\mathcal{O}(\lg n)$
LCA with RMQ (sparse tables)	$\mathcal{O}(n \lg n)$	$\mathcal{O}(1)$
Theoretical best (?)	$\mathcal{O}(n)$	$\mathcal{O}(1)$

Level ancestor	Preprocessing	Query
Binary lifting	$\mathcal{O}(n \lg n)$	$\mathcal{O}(\lg n)$
Ladder	$\mathcal{O}(n)$	$\mathcal{O}(\lg n)$
Ladder + binary lifting	$\mathcal{O}(n \lg n)$	$\mathcal{O}(1)$
Theoretical best (?)	$\mathcal{O}(n)$	$\mathcal{O}(1)$

Looking at these tables, we feel that we ought to be able to find the "theoretically best" solution of $\mathcal{O}(n)$ and $\mathcal{O}(1)$. The question is now: is it possible?

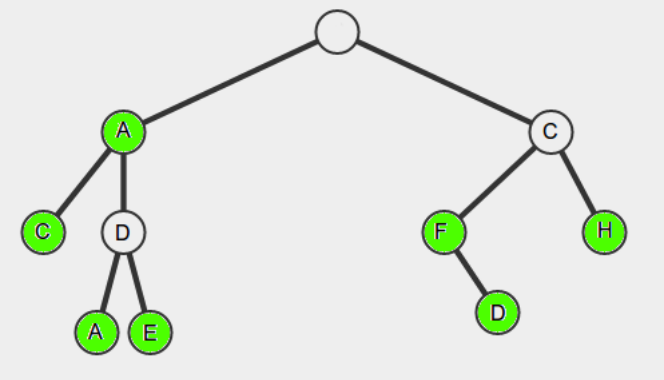
The answer is yes, there is a way to solve both problems with $\mathcal{O}(n)$ preprocessing and $\mathcal{O}(1)$ query time! However, it is a somewhat unsatisfying kind of yes. This is because they are quite complicated to implement, and the constants involved in these "theoretically best" solutions are a bit large that the speedup gains are pretty small, so it's not really worth it for the ns we usually encounter in practice.

If you're interested in reading more about it, read up on the "four Russians trick". It allows us to shave off that final $\lg n$ factor in the preprocessing time for both problems. It is quite a fun read, as long as you don't seriously think of implementing it![12]

[12] though it depends on taste, really. I would find it fun to implement!

3.4 An application: Tries

A **trie** is an information re**trie**val[13] data structure that can compactly store several strings. One such trie is shown below.



The strings stored can be taken by traversing down the tree. Each node colored green in the figure above is an end node of a string. This particular trie stores 7 strings: "A", "AC", "ADA", "ADE", "CF", "CFD", and "CH".

Tries can provide several functionalities apart from compact storage. One of the most common use of trie is as an efficient lookup to check whether a string exists in a set. This can be done in $\mathcal{O}(S)$, where S is the length of the string, by simply traversing down the trie. If we were to do it without a trie, we can simply keep an array of the strings, sort them, then do a lookup through binary searching. This has a complexity of $\mathcal{O}(S \lg n)$. One can argue that they can achieve an average complexity of $\mathcal{O}(S)$ through hashing, but hash functions give no assurance of efficiency, and hash collisions are a lot harder to handle efficiently than coding tries.[14]

One easy way to implement tries is to have each node keep α child pointers, where α is the size of the *alphabet* (the set of letters forming our strings). We also keep a flag indicating whether or not the node is an end node of some string. An implementation of the above data structure is shown below, with a dictionary size of 26 (one for each uppercase character in the English alphabet). It includes a function for inserting a new string into the trie and another for a simple lookup query.

```
1  const int A = 26;
2  class Trie {
3  private:
4      vector<Trie*> child;
5      bool end_node;
6
7      void _insert(string &str, int pos) {
8          if(pos == str.size()) {
9              end_node = true;
10             return;
11         }
12
13         int p = str[pos] - 'A';
```

[13] Officially, it's pronounced the same as "tree", but a lot of people found this ambiguous. It is common to pronounce it as "try".

[14] Hash functions are functions that map a given input to integer outputs bounded by a certain range. More in-depth discussion on hashing will be given in a future module.

```
14          if(child[p] == nullptr)
15              child[p] = new Trie();
16
17          child[p]->_insert(str, pos+1);
18      }
19
20      bool _contains(string &str, int pos) {
21          if(pos==str.size())
22              return end_node;
23
24          int p = str[pos]-'A';
25          if(child[p] == nullptr)
26              return false;
27          return child[p]->_contains(str, pos+1);
28      }
29
30  public:
31      Trie(): child(A, nullptr) {}
32
33      ~Trie() {
34          for (Trie *ch: child)
35              if (ch) delete ch;
36      }
37
38      void insert(string str) {
39          _insert(str, 0);
40      }
41
42      bool contains(string str) {
43          return _query(str, 0);
44      }
45  };
```

Apart from as simple lookups, we can also employ more complex operations using the trie, such as the longest common prefix problem. In the longest common prefix problem, we're given two strings, and we want to know the *length* of the longest common prefix between the two strings efficiently. We can do this naively in $\mathcal{O}(S)$. However, if both strings are stored in the same trie, and we have pointers to where those strings end within the trie, then we can get the end node of the longest common prefix by getting the LCA of the end nodes of the two strings. We can get the LCA more efficiently in $\mathcal{O}(\lg S)$ or even $\mathcal{O}(1)$, as discussed in the LCA section.

Tries also have plenty of uses aside from these. For example, we can also solve queries like this: given s, how many strings in our set start with s? This can be solved with dynamic programming technique on the trie. Later on, we'll cover more advanced uses, in the context of string algorithms.

4 Self-Balancing Trees

Previously, we've discussed binary search trees, and how it makes it faster to find a certain element by traversing down the tree. However, this presupposes that the binary search tree is balanced, and otherwise finding elements becomes $\mathcal{O}(n)$ in the worst case (happens when the BST reduces to a linked list). We can argue that if given random inputs, then the BST becomes somewhat balanced even without balancing, and the height of the tree in the average case become $\mathcal{O}(\lg n)$, but when it comes to competitive programming we have to be certain – the input is not always random, and the testers of the problem have most likely ensured that non-balancing BSTs would fail. And so in self-balancing binary search trees, we want to ensure that every time the tree is modified, the height of the tree still remains around $\lg n$.

There are several techniques to maintain balanced trees – Treaps, AVL Trees, Red-Black Trees, Splay Tree, and so on. For this document, we'll only focus on the first two: Treaps and AVL Trees.

4.1 Treap

A **treap** is a data structure that follows binary search tree ordering (left child is less than root, and right is greater) for its contained values, but has an additional *priority* value wherein it follows the heap ordering (both children are less than root). Let's denote the values as X and the priorities as Y. In other words, a treap is a BST when looking at X, and a binary heap when looking at Y.

The value of a node is the data which we want to be able to access efficiently. Note that if the inserted values in a non-balancing BST is random, then the height of the tree will be $\mathcal{O}(\lg n)$ on average. However, we cannot be assured that this is the case when it comes to inputted values in a problem, and most of the time, the order the elements are inserted is beyond our control. However, if we assign a second sorting criterion (priority) which we can control and assign randomly, then we can somehow ascertain that the average height of our tree is $\mathcal{O}(\lg n)$. This is the basis of the treap.

We also need to implement some functionality to our treap for it to be practical. For now, we will attempt to implement the **set** data structure. The following are our operations:

- **set.create()**. Create a new set, s, initially empty.
- s**.insert(**x**)**. Insert the value x into the set (if it doesn't already exist).
- s**.contains(**x**)**. Checks whether the set contains x.
- s**.remove(**x**)**. Delete the value x from the set.

All of these functions are expected to run in $\mathcal{O}(\lg n)$ time.

What's interesting about treaps is that we can **split** a treap into two treaps *left* and *right* such that, for some given x, the left treap contains all elements with values $x_i < x$, and the right treap contains all elements with the values $x_i \geq x$. We can also **merge** two treaps into a single treap, assuming that all the values in the left treap is less than the values in the right treap (e.g. the result of a split). Both of these can be done in $\mathcal{O}(\lg n)$ on average. These two operations are the core of our treap implementation; we'll be using them to implement other functionalities.

Given all of these, we can create a template for our treap, shown below. Here, we already define our constructor to provide a random value for the priority.

```cpp
class TreapNode {
public:
    int x;
    unsigned int prio;
    TreapNode *left = nullptr, *right = nullptr;
    TreapNode() {}
    TreapNode(int x): x(x), prio(rand()) {}  // random priority

    ~TreapNode() {
        if(left) delete left;
        if(right) delete right;
    }
};
```

4.1.1 Find

The **find** function is the same as normal BST search.

```cpp
TreapNode *find(TreapNode *t, int x) {
    if (t == nullptr) return nullptr;  // not found
    if (t->x == x) return t;  // found!
    return find(x < t->x ? t->left : t->right, x);  // go to the appropriate
    ↪ child
}
```

4.1.2 Split and Merge

To implement the **split** function, notice that if a node contains a value greater than the value x, then all the nodes in its right subtree have values also greater than x, and so we only need to split the left subtree. A similar property can be found for the left subtree when the node contains a value less than x. With this, we can split a treap with the following recursion:

```cpp
void split(TreapNode *t, int x, TreapNode *&l, TreapNode *&r) {
    // splits t into l and r according to x
    if (t == nullptr) {
        l = r = nullptr;
    } else if (t->x > x) {
        r = t;
        split(t->left, x, l, r->left);
    } else {
        l = t;
        split(t->right, x, l->right, r);
    }
}
```

Note that it is also possible to implement a `split` function that returns the resulting pair (so it returns a `pair<TreapNode*,TreapNode*>`). It is a perfectly ok approach. Feel free to implement it that way as well if you wish.

To implement the **merge** function, we have to combine the two treaps without violating our heap property on the priorities, under the assumption that the left treap values are all less than those of the right treap. To do this, at each step, we can choose the node with greater priority as root, and recursively merge the rest while retaining order. If we choose the right treap as the root, then we need to merge the left treap to the left of the new root, and vice-versa.

```
TreapNode *merge(TreapNode *l, TreapNode *r) { // merges l and r
    if (l == nullptr) return r;
    if (r == nullptr) return l;
    if (l->prio > r->prio) {
        l->right = merge(l->right, r);
        return l;
    } else {
        r->left = merge(l, r->left);
        return r;
    }
}
```

With both split and merge implemented, we can now implement insert and remove.

4.1.3 Insert and Remove

For **insert**, we create a new node with value x and an assigned priority y. Starting from the root, if the current node is n, and $y < y_n$, then insert the node down to one of its children (keeping BST ordering). Otherwise, we split this treap by x, and use the generated l and r as the left and right children of our node, and assign our node as the new root.

```
TreapNode *insert(TreapNode *t, TreapNode *new_node) {
    if (t != nullptr && t->prio > nt->prio) {
        if (new_node->x < t->x) {
            t->left = insert(t->left, new_node);
        } else {
            t->right = insert(t->right, new_node);
        }
        return t;
    } else {
        split(t, new_node->x, new_node->left, new_node->right);
        return new_node;
    }
}
```

Alternatively, we could simply split the treap by x, then merge the three treaps (left, right, and x) using two merges. This seems somewhat wasteful, but it's easier to remember, and the complexity remains the same.

```
TreapNode *insert(TreapNode *t, TreapNode *new_node) {
    TreapNode *l, *r, *temp;
    split(t, new_node->x, l, r);
```

```
    return merge(merge(l, new_node), r);
}
```

For **remove**, we find the node with value x, merge its children, replace our node with the new merged treap, and delete our node.

```
TreapNode *remove(TreapNode *t, int x) {
    if (t->x == x) {
        TreapNode *r = merge(t->left, t->right);
        delete t;
        return r;
    } else if (x < t->x) {
        t->left = remove(t->left, x);
        return t;
    } else {
        t->right = remove(t->right, x);
        return t;
    }
}
```

Alternatively, we could simply split the treap by x and get two treaps t_1 and t_2. We then split t_2 by $x+1$ to get the node of x and t_3. Finally, we delete the node of x, and merge t_1 and t_3.

```
TreapNode *remove(TreapNode *t, int x) {
    Treap *t1, t2, t3, nx;
    split(t, x, t1, t2);
    split(t2, x + 1, nx, t3);
    delete nx;
    return merge(t1, t3);
}
```

The expected complexity of all operations here are $\mathcal{O}(\lg n)$ due to the randomized nature of the priorities. The problem setter has no control over these priorities, so in particular, no given sequence operations is guaranteed to trigger the worst case of the algorithm.

> **Exercise 4.1.** In the alternative **remove** implementation, we computed `x + 1` from `x`, so it creates an implicit assumption that `x` is an integer. Explain how to modify the code so that it can still handle any kinds of (comparable) values.
>
> *Hint:* You actually need to create a second kind of `split` function.

4.1.4 Combining everything

Using the functions above, we can now combine everything to implement the **set** ADT with a Treap:

```
class TreapSet {
    TreapNode *root = nullptr;

public:
    void insert(int x) {
        root = insert(root, new TreapNode(x));
    }

    bool contains(int x) {
        return find(root, x) != nullptr;
    }

    void remove(int x) {
        root = remove(root, x);
    }
};
```

4.2 AVL Tree

The **AVL Tree** is binary search tree with the following property: for every node, the heights of its two subtrees differ by at most one. With this, it ensures that the height of the tree is $\mathcal{O}(\lg n)$.

> **Exercise 4.2.** Show that the height of any AVL tree with n nodes is $\mathcal{O}(\lg n)$.
>
> *Hint:* You can show first that for some constant c, $F_k \geq c\sqrt{2}^k$ (for all k), where F_k is the kth Fibonacci number.

This is a relaxed version of "completely balanced binary tree". The height can be a constant factor worse than $\lg n$, but the benefit is that it is far easier to maintain. Indeed, we can always maintain the AVL tree property after insertion with only $\mathcal{O}(\lg n)$ additional work, as we shall see below. Therefore, all operations run in $\mathcal{O}(\lg n)$ worst-case time, an improvement over $\mathcal{O}(\lg n)$ *expected* time.

4.2.1 Heights

To maintain the AVL tree property of every node, it makes sense to keep track of the height of every node. With the height values stored, we can easily detect if we need to rebalance a node by checking the difference between the heights of its two children. If the absolute difference becomes 2, then we say that the node is **unbalanced**, and we have to do some modifications in order to rebalance the tree. These modifications are called *tree rotations*, and there is a corresponding type of rotation depending on the scenario.

Thus, our implementation begins like this:

```
class AVLNode {
    int x, h; // h represents the height
    AVLNode *left = nullptr, *right = nullptr;
```

```
    AVLNode() {}
    AVLNode(int x): x(x) {}
};

int height(AVLNode *t) {
    return t == nullptr ? -1 : t->h;
}

void combine(AVLNode *t, AVLNode *left, AVLNode *right) {
    t->left = left;
    t->right = right;
    t->h = 1 + max(height(left), height(right));
    return t;
}
```

We will use the `combine` function to set the children of some node `t` with the height automatically computed, to make things easier for us.

4.2.2 Find

The **find** function is basically the same.

```
AVLNode *find(AVLNode *t, int x) {
    if (t == nullptr) return nullptr;
    if (t->x == x) return t;
    return find(x < t->x ? t->left : t->right, x);
}
```

4.2.3 Insertion

Insertion starts similarly to the usual insertion, but after inserting the new node, we must ensure that the tree still satisfies the AVL property. We perform this check after insertion, and if we detect a node that is unbalanced, we balance it with some tree rotations (to be described later).

```
AVLNode *balance_if_needed(AVLNode *x) {
    if (abs(height(x->left) - height(x->right)) >= 2) {  // x is unbalanced
        /* perform balancing here */
        return /* balanced version of x */;
    } else {  // no balancing needed
        return x;
    }
}

AVLNode *insert(AVLNode *t, AVLNode *new_node) {
    if (t == nullptr) return new_node;
    if (new_node->x < t->x) {
        t->left = insert(t->left, new_node);
```

```
    } else {
        t->right = insert(t->right, new_node);
    }
    return balance_if_needed(t);
}
```

Once we fill out the details of `balance_if_needed`, we will have a complete AVL tree implementation!

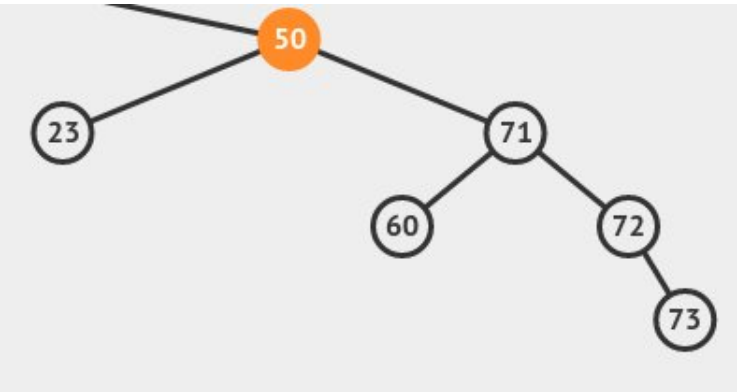
4.2.4 Rebalancing

Let's look at the case where a node `x` is unbalanced, but both of its children (if any) are balanced. Note that since `balance_if_needed` is called bottom-up, this will be true. Also, since `x` is unbalanced, it means one of its subtrees has height exactly 2 greater than the other.

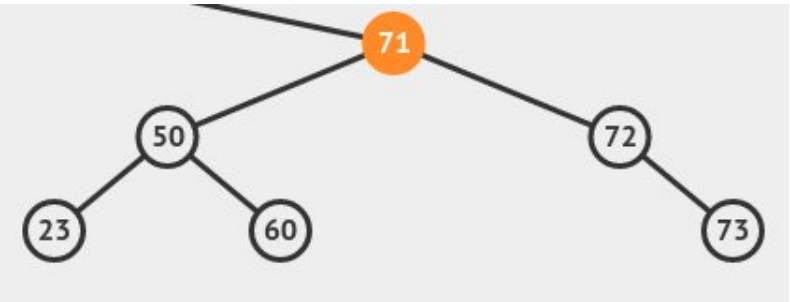
Exercise 4.3. Explain why the absolute difference is at most 2 in our situation. In other words, explain why, after inserting a new node to an AVL tree without rebalancing, the absolute difference between the heights of two subtrees of any node is at most 2.

There are four scenarios, depending on how `x` is unbalanced. Actually, two of them are mirror images of the other two, so there are only two essentially distinct cases. So let's assume for now that the right subtree is taller than the left subtree.

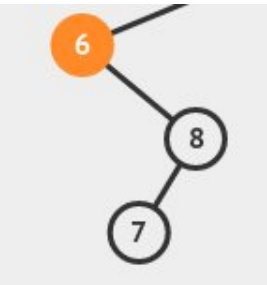
- `height(x->right->right) < height(x->right->left)`. This scenario is depicted as follows, where x is the node containing 50 and z is the node containing 71.



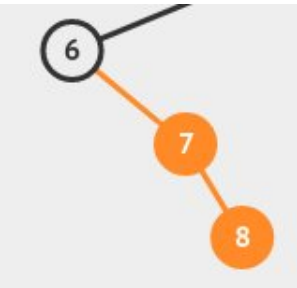
To rebalance this, we have to do what's called a simple left rotation. We let z be the new root of this subtree, z's new left child will be x, and x's new right child will be z's old left child. The result is shown in the image below.



- `height(x->right->right) > height(x->right->left)`. This scenario is depicted as follows, where x is the node containing 6 and z is the node containing 8.



To rebalance this, we first have to transform it into a "Right Right" scenario by doing a simple right rotation on z. The result is shown in the image below.



From here on, we can fix this using a simple left rotation on x. This compound rotation is what's called a right-left double rotation.

In terms of implementation, it's actually quite simple. You simply identify all nodes that need to be repointed to new nodes, and simply assign them to their new children. You simply compare the start and end figures/drawings to know where to point which. For example:

```
AVLNode *balance_if_needed(AVLNode *x) {
    if (height(x->right) >= height(x->left) + 2) {
        if (height(x->right->right) > height(x->right->left)) {

            // left rotation
            // source nodes, in BST order
            AVLNode *a  = x->left;
            AVLNode *ab = x;
            AVLNode *b  = x->right->left;
            AVLNode *bc = x->right;
            AVLNode *c  = x->right->right;
            // point them to their new children
            combine(ab, a, b);
            combine(bc, ab, c);
            return bc; // return the new root

        } else  {

            // right-left rotation
            AVLNode *a  = x->left;
            AVLNode *ab = x;
            AVLNode *b  = x->right->left->left;
            AVLNode *bc = x->right->left;
            AVLNode *c  = x->right->left->right;
```

```
            AVLNode *cd = x->right;
            AVLNode *d  = x->right->right;

            combine(ab, a, b);
            combine(cd, c, d);
            combine(bc, ab, cd);

            return bc;

        }
    } else if (height(x->left) >= height(x->right) + 2) {

        ... /* mirror cases here */

    } else {
        return x;
    }
}
```

For the remaining cases, we have that the left subtree is taller than the right subtree, but there will again be two cases which are basically mirror versions of the above, so we have a left-left and a left-right rotation. The implementation is very similar, so we will leave it up to you to implement.

Exercise 4.4. Finish the implementation of `balance_if_needed` above.

Also, combining everything into a `AVLTree` class is as straightforward as that for the `Treap`.

Now, assuming you have proven that the height of an AVL tree is $\mathcal{O}(\lg n)$, all operations now run in worst-case $\mathcal{O}(\lg n)$, even with all that rebalancing. This is because `balance_if_needed` clearly runs in $\mathcal{O}(1)$, so the amount of additional work needed is proportional to the height of the tree.

5 Problems

Solve as many as you can! Ask me if anything is unclear.[15] In general, the harder problems will be worth more points, but this is only an approximation.

Problems with red points are considered more important.

5.1 Warmup problems

Most of these are straightforward applications of some of the algorithms discussed. You may want to use these problems to test your implementations of the algorithms above.

W1 [50★] **Basic range queries:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/basic-range-queries

5.2 Non-coding problems

No need to be overly formal in your answers; as long as you're able to convince me, it's fine!

Every exercise mentioned in the main text is worth [100★].

Veterans, solve at least [1501★]. First timers, a good personal target would be [801★].

N1 [200★] In long path decomposition, show that the path between any two nodes passes through at most $\Theta(\sqrt{n})$ distinct long paths. Show that this is tight.

N2 [200★] You are given a rooted tree. Perform a preorder traversal, and let $disc(x)$ be the index of node x in the preorder traversal. Let a, b, and c be three nodes (not necessarily distinct) such that $disc(a) \leq disc(b) \leq disc(c)$. Show that

$$disc(lca(a, c)) \leq \min(disc(lca(a, b)), disc(lca(b, c))).$$

N3 [200★] Show that for any three nodes a, b, and c in a rooted tree, the set $\{lca(a, b), lca(b, c), lca(c, a)\}$ has at most two elements.

N4 [300★] Given k nodes $a_1, a_2, \ldots, a_k$ in a rooted tree, what is the maximum size of the set

$$\{lca(a_i, a_j) \mid 1 \leq i, j \leq k\},$$

in terms of k? Show why this is so.

[15] Especially for ambiguities! Otherwise, you might risk getting fewer points even if you *technically* answered the question correctly.

5.3 Coding problems

Class-based implementations are strongly recommended; the idea is to make the implementation easily reusable.[16] Making the implementation self-contained in a class makes it very easy for you to reuse, which is handy for your future contests!

Veterans, solve at least [1201★]. First timers, a good personal target would be [701★].

Partial points will be given for those that don't achieve optimal specs (e.g. achieved $\mathcal{O}(\lg n)$ time when asked to implement it in $\mathcal{O}(1)$ amortized).

C1 [200★] Implement a data structure that takes in a tree (in the form of an adjacency list) with n nodes and an array of n integers. Each integer A_i is associated with node i in the tree. Your data structure must have a query function that takes in two integers i and j and returns the sum of all elements in the unique path between i and j.

Preprocessing should be $\mathcal{O}(n \lg n)$, and each query should be $\mathcal{O}(1)$.

C2 [200★] Implement an $\mathcal{O}(n \lg n)$ preprocessing step that reduces the RMQ problem to the LCA problem.

Hint: Build a tree such that range minimum queries correspond to LCAs. The root of this tree will correspond to the minimum of the array.

Bonus: [100★] Implement an $\mathcal{O}(n)$ preprocessing step.

You can solve as many problems as you want. Submit directly to the specified online judges.

For those that were partially solved (such as Hacker Rank subtasks), you can opt for partial points by submitting your code in the submission bin at the end of the week. Note that the partial points may not be up to scale with the points of the subtask. If you have a partially solved problem that doesn't have subtasks, but you think you were getting near the solution, you can opt to submit it as well for consideration. For all cases, include the details as to how you tried to approach the problem, and any insights you've garnered (you can just add it as a comment in your code).

S1 [200★] **Karen and Cards:** https://codeforces.com/problemset/problem/815/D

S2 [200★] **Chef and his study plans:** https://www.codechef.com/problems/SUBSEG2

S3 [200★] **Best Edge Weight:** https://codeforces.com/problemset/problem/827/D

S4 [200★] **Leha and security system:** https://codeforces.com/problemset/problem/794/F

S5 [200★] **Merciless Chef:** https://www.codechef.com/problems/MLCHEF

S6 [200★] **Pitong Gatang:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/pitong-gatang

S7 [200★] **For Science™:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/2015-for-science-tm

S8 [200★] **Selective Additions:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/selective-additions

S9 [200★] **Expando:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/2017-expando

[16] See also: https://en.wikipedia.org/wiki/Modular_programming.

S10 [500★] **Chef and Guard Towers:** https://www.codechef.com/problems/TRICONT (Use sqrt decomposition)

S11 [500★] **Birjik and Nicole's Tree Game:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/birjik-and-nicoles-tree-game

S12 [500★] **Summing in a Tree:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/summing-in-a-tree

S13 [500★] **GGVV:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/2016-ggvv

Acknowledgment

Most of the material is based on the Data Structures materials written by Josh Quinto.