Data Structures I

COMP4128 Programming Challenges

School of Computer Science and Engineering
UNSW Australia

Table of Contents

How does <vector> work?

- Vectors are dynamic arrays
- Random access is $O(1)$, like arrays
- A vector is stored *contiguously* in a single block of memory
- Supports an extra operation `push_back()`, which adds an element to the end of the array

How does `push_back()` allocate memory?

- Do we have enough space allocated to store this new element? If so, we're done: $O(1)$.
- Otherwise, we need to allocate a new block of memory that is big enough to fit the new vector, and copy all of the existing elements to it. This is an $O(N)$ operation when the vector has N elements. How can we improve?
- If we double the size of the vector each reallocation, we perform $O(N)$ work once, and then $O(1)$ work for the next $N - 1$ operations, an average of $O(1)$ per operation.
- We call this time complexity *amortised* $O(1)$.
- How is this different from average case complexity? When we quote 'average case' complexity (e.g. for a hash table), it is usually possible to construct a case where N consecutive operations will each take $O(N)$ time, for a total time of $O(N^2)$. This is not possible with amortised complexity.

Example Vector Usage


```cpp
#include <cassert>
#include <vector>
using namespace std;


int main() {
    vector<int> v;
    for (int i = 0; i < 10; i++) v.push_back(i*2);
    v[4] += 20;
    assert(v[4] == 28);

    return 0;
}
```

Table of Contents

- Supports `push()` and `pop()` operations in $O(1)$
- LIFO (last in, first out)
- STL implements a templated stack in `<stack>`
- Equivalently, you can use an array or vector to mimic a stack, with the advantage of allowing random access

- Supports `push()` and `pop()` operations in $O(1)$
- FIFO (first in, first out)
- STL implements a templated queue in `<queue>`
- Equivalently, you can use an array or vector to mimic a queue, with the advantage of allowing random access

Example Usage


```cpp
#include <cassert>
#include <queue>
#include <stack>
using namespace std;

int main() {
    stack<int> stk;
    queue<long long> que;

    stk.push(1);
    stk.push(2);
    assert(stk.top() == 2);
    stk.pop();
    assert(stk.top() == 1);
    assert(stk.size() == 1);
    assert(!stk.empty());

    que.push(1);
    que.push(2);
    assert(que.front() == 1);

    return 0;
}
```

Table of Contents

- STL's `<set>` is a set with $O(\log n)$ random access
- Internally implemented as a red/black tree of set elements
- Unfortunately doesn't give you easy access to the underlying tree - iterator traverses it by infix order
- C++11 adds `<unordered_set>`, which uses hashing for $O(1)$ average case ($O(n)$ worst case) random access
- Main advantage of `<set>` is it keeps the data ordered, hence has `lower_bound(x)` and `upper_bound(x)` which returns the next element not less than (resp. greater than) x.
- `<multiset>` and (C++11) `<unordered_multiset>` are also available

- STL's `<map>` is a dictionary with $O(\log n)$ random access
- Internally implemented with a red/black tree of (key,value) pairs
- Unfortunately doesn't give you access to the underlying tree - iterator traverses it by infix order
- C++11 adds `<unordered_map>`, which uses hashing for $O(1)$ average case ($O(n)$ worst case) random access
- Main advantage of `<set>` is it keeps the data ordered, hence has `lower_bound(x)` and `upper_bound(x)` which returns the next element not less than (resp. greater than) x.
- `<multimap>` and (C++11) `<unordered_multimap>` are also available

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

set<int> s;
map<int, char> m;

int main() {
    s.insert(2); s.insert(4); s.insert(1);
    m = {{1,'a'}, {4,'c'}, {2,'b'}};
    // Check membership:
    cout << (s.find(2) != s.end()) << ' ' << (s.find(3) != s.end()) << '\n';
          // 1 0
    // Access map:
    cout << m[1] << '\n'; // 'a'
    // WARNING: Access to non-existent data just silently adds it, avoid
         this.
    // cout << m[3] << '\n';
    // Lower and upper bounds:
    cout << *s.lower_bound(2) << '\n'; // 2
    cout << *s.upper_bound(2) << '\n'; // 4
    auto it = m.lower_bound(2);
    cout << it->first << ' ' << it->second << '\n'; // 2 b
    // Move around with prev/next or increment/decrement
    cout << prev(it)->first << '\n'; // 1
    ++it;
    cout << it->first << '\n'; // 4
    return 0;
}
```

- One of the main problems with set and map is they don't track index information.
- So you can't query what the k-th number is or how many numbers are $< x$.
- Most SBBSTs can be modified to track this metadata. But we do not want to implement a SBBST.

Order Statistic Trees

- There is a fix in GNU C++. So it is not a C++ standard but pretty widespread.
- Contained in an extension called "Policy Based Data Structures".
- In headers:

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>

using namespace __gnu_pbds;
```

- Details are pretty technical, fortunately we don't need to know them.

New data structure:

```
typedef tree<int, null_type, less<int>, rb_tree_tag,
             tree_order_statistics_node_update>
             ordered_set;
```

- Key type: int
- No mapped type (a set not a map)
- Comparison: `less<int>`
- `rb_tree_tag`: Implemented as a red-black tree, guarantees O(log n) performances
- `tree_order_statistics_node_update`. The magic. Tells it to update order statistics as it goes.

Order Statistic Trees

Essentially a set/map with 2 extra operations:

- `find_by_order(x)`: Find the x-th element, 0-indexed.
- `order_of_key(x)`: Output the number of elements that are $< x$.
- Both are $O(\log n)$ still!
- Furthermore, in other regards they still behave like a set/map!

```cpp
#include <bits/stdc++.h>
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>

using namespace __gnu_pbds;
using namespace std;

typedef tree<int, null_type, less<int>, rb_tree_tag,
             tree_order_statistics_node_update> ordered_set;

ordered_set myset;
int main() {
    myset.insert(2);
    myset.insert(4);
    myset.insert(1);
    printf("%d\n", *(myset.find_by_order(0))); // 1
    printf("%d\n", (int)myset.order_of_key(3)); // 2
    printf("%d\n", (int)myset.order_of_key(4)); // 2
    printf("%d\n", (int)myset.size()); // 3
}
```

```cpp
#include <bits/stdc++.h>
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>

using namespace __gnu_pbds;
using namespace std;

typedef tree<int, char, less<int>, rb_tree_tag,
             tree_order_statistics_node_update> ordered_map;

ordered_map mymap;
int main() {
    mymap[2] = 'a';
    mymap[4] = 'b';
    mymap[1] = 'c';
    pair<int, char> pic = *mymap.find_by_order(0);
    printf("%d %c\n", pic.first, pic.second); // 1 c
    printf("%d\n", (int)mymap.order_of_key(3)); // 2
    printf("%d\n", (int)mymap.order_of_key(4)); // 2
    printf("%d\n", (int)mymap.size()); // 3
}
```

Table of Contents

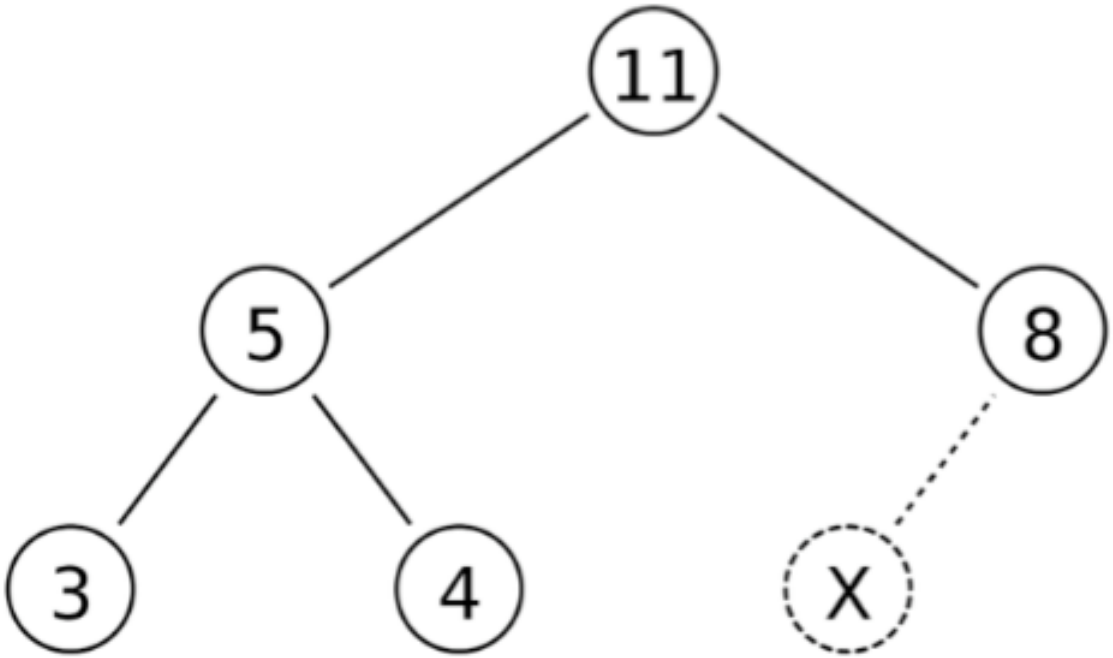
- Supports `push()` and `pop()` operations in $O(\log n)$ Supposes `top()` in $O(1)$.
- `top()` returns the value with highest priority
- Is usually used to implement a priority queue data structure
- STL implements a templated priority queue in <queue>
- The default is a max heap - often we want a min heap, so we declare it as follows:

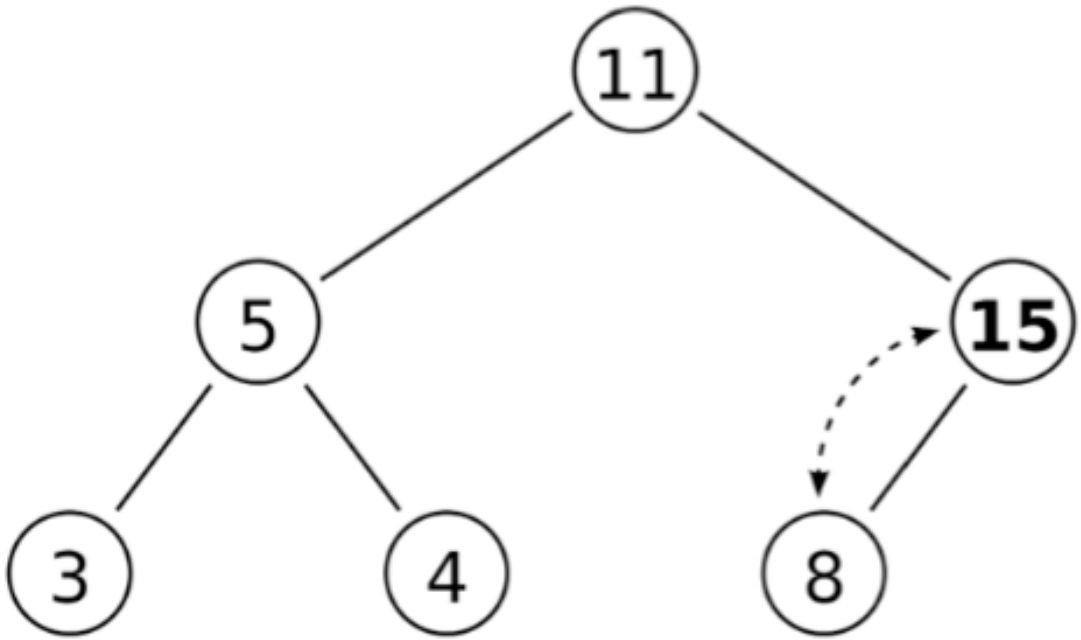
```
priority_queue <T, vector<T>, greater<T>> pq;
```

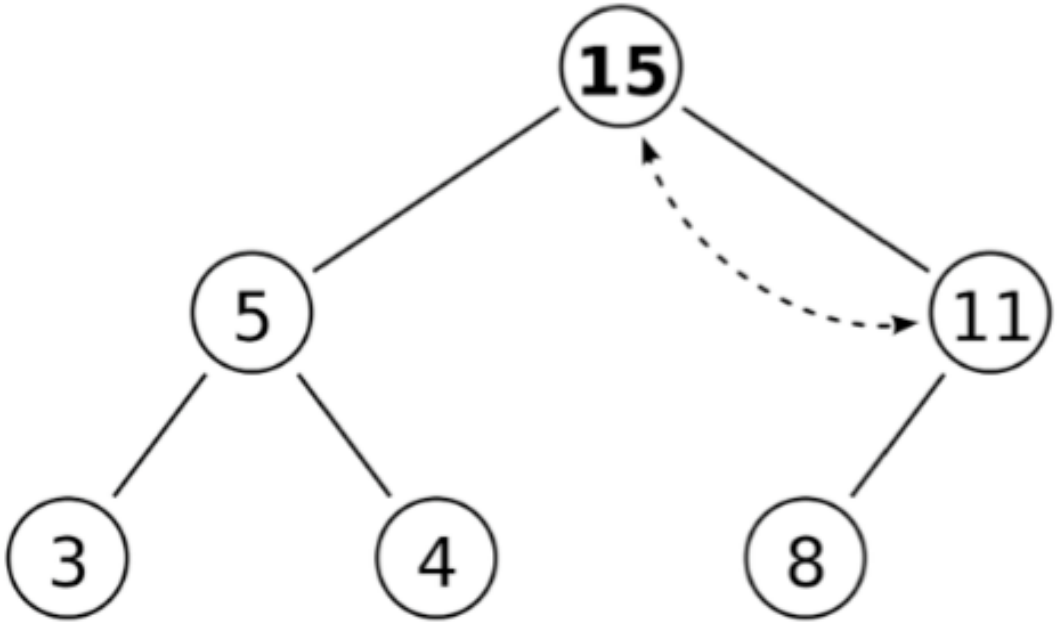
- It's significantly more code to write a heap yourself, as compared to writing a stack or a queue, so it's usually not worthwhile to implement it yourself

- The type of heaps usually used is more accurately called a *binary array heap* which is a binary heap stored in an array.
- It is a binary tree with two important properties:
  - **Heap property:** the value stored in every node is greater than the values in its children
  - **Shape property:** the tree is as close in shape to a complete binary tree as possible

- Operation implementation
  - `push(v)`: add a new node with the value v in the first available position in the tree. Then, while the heap property is violated, swap with parent until it's valid again.
  - `pop()`: the same idea (left as an exercise)







- **Implementation**

```
#include <queue>
```

Table of Contents

Basic Uses

- Most uses fall out naturally from the use case.
- Vectors: Use everywhere.
- Stacks: When you need a LIFO structure. Generally when the most recent thing you've seen is most important or should be processed first.
- E.g: basic parsers, dfs, bracket matching.
- Queues: When you need a FIFO structure. Generally when you want to process events in order of occurrence.
- E.g: event processing, bfs.

- Heap: When you find yourself asking how I can get the "largest/smallest" item.
- E.g: Dijkstras, problem from yesterday.
- Set: Seen array on unbounded keys. Also when you need to dynamically maintain a sort order.
- E.g: Recognizing duplicates, find closest key to x.
- Map: As above but with keyed data.
- E.g: Count duplicates, find index of the closest key to x.

Recall this problem boiled down to:

- Process countries in any order.
- For each, seat delegates at restaurant with **most** seats, then second **most**, etc.
- Sounds like a max heap to me!

Example problem: Restaurants


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

const int MAXN = 2005, MAXM = 2005;
int N, numDelegates[MAXN], M;
priority_queue<int> restaurants;

int main() {
    cin >> N;
    for (int i = 0; i < N; i++) cin >> numDelegates[i];
    cin >> M;
    for (int i = 0; i < M; i++) {
        int s; cin >> s; restaurants.push(s);
    }
    int starved = 0;
    for (int i = 0; i < N; i++) {
        vector<int> poppedRestaurants;
        int delegatesRemaining = numDelegates[i];
        while (delegatesRemaining && !restaurants.empty()) {
            // seat a delegate at the restaurant with the most seats.
            delegatesRemaining--;
            int seatsRemaining = restaurants.top() - 1;
            restaurants.pop();
            poppedRestaurants.push_back(seatsRemaining);
        }
        for (int r : poppedRestaurants)
            if (r > 0) restaurants.push(r);
        starved += delegatesRemaining;
    }
    cout << starved << '\n';
}
```

New complexity?

- Let A be the maximum number of delegates per country.
- $O(A \cdot N \cdot \log M) \approx O(2000 \cdot 20 \cdot 11)$.

- **Problem statement** You are given an array of n numbers, say $a_0, a_1, \ldots, a_{n-1}$. Find the number of pairs (i, j) with $0 \leq i < j \leq n$ such that the corresponding contiguous subsequence satisfies

$$a_i + a_{i+1} + \ldots + a_{j-1} = S$$

  for some specified sum S.
- **Input** The size n of the array ($1 \leq n \leq 100,000$), and the n numbers, each of absolute value up to 20,000, followed by the sum S, of absolute value up to 2,000,000,000.
- **Output** The number of such pairs

- **Algorithm 1** Evaluate the sum of each contiguous subsequence, and if it equals S, increment the answer.
- **Complexity** There are $O(n^2)$ contiguous subsequences, and each takes $O(n)$ time to add, so the time complexity is $O(n^3)$.
- **Algorithm 2** Compute the prefix sums

$$b_i = a_0 + a_1 + \ldots + a_{i-1}.$$

Then each subsequence can be summed in constant time:

$$a_i + a_{i+1} + \ldots + a_{j-1} = b_j - b_i.$$

- **Complexity** This solution takes $O(n^2)$ time, which is an improvement but still too slow.

- We need to avoid counting the subsequences individually.
- For each $1 \leq j \leq n$, we ask: how many $i < j$ have the property that $b_i = b_j - S$?
- If we know the frequency of each value among the b_i, we can add all the answers involving j at once.
- The values could be very large, so a simple frequency table isn't viable - use a map!

- **Algorithm 3** Compute the prefix sums as above. Then construct a map, and for each b_j, add the frequency of $b_j - S$ to our answer and finally increment the frequency of b_j.
- **Complexity** The prefix sums take $O(n)$ to calculate, since

$$b_{i+1} = b_i + a_i.$$

Since map operations are $O(\log n)$, and each b_j requires a constant number of map operations, the overall time complexity is $O(n \log n)$.

- **Implementation**

```
const int N = 100100;
int a[N];
int b[N];

int main() {
  int n, S;
  cin >> n;
  for (int i = 0; i < n; i++) {
    cin >> a[i];
    b[i+1] = b[i] + a[i];
  }
  cin >> S;

  long long ret = 0;
  map<int,int> freq;
  for (int i = 0; i <= n; i++) {
    ret += freq[b[i]-S];
    freq[b[i]]++;
  }
  cout << ret << endl;
}
```

Largest Interval

- **Problem statement** You have $M \leq 1,000,000,000$ chairs, initially all empty. There are $U \leq 100,000$ updates, in each a person comes in and takes an unoccupied chair c_i. After each update, what is the longest range of unoccupied chairs?
- **Input** First line, M then U. Next U lines, each contains one integer, $1 \leq c_i \leq M$. Guaranteed no integer appears more than once.
- **Output** For each update, an integer, the longest range of unoccupied chairs after the update.

- **Sample Input:**

```
12 3
5
7
10
```

- **Sample Output:**

```
7
5
4
```

- **Observation 1:** We only care about maximal ranges. Assuming chair 0 and chair $M + 1$ are occupied, we only care about ranges starting and ending with occupied chairs.
- So we will maintain for each chair, what is the length of the range to its right.
- How does an update change the intervals?
- It breaks one apart and adds 2.

- What data do we need to store to handle updating intervals (i.e: to determine what the 2 new intervals are when we insert a chair)?
- For each update, we need to find the closest chair in both directions.
- We need to maintain a *sorted* list of chairs *associating* with each chair the length of the range starting at that chair.
- Map!
- Figuring out the new range lengths is basic maths, just be careful with off-by-1s!

- Now we know how to track length of each range. Remains to track the *largest* of the ranges.
- Heap!
- But wait, heaps can not do arbitrary deletions... (which we need when we delete an interval)
- Set!

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

int M, U;
map<int, int> chairToRange;
multiset<int> allRanges;

void addRange(int start, int length) {
    chairToRange[start] = length;
    allRanges.insert(length);
}

void updateRange(int start, int length) {
    allRanges.erase(allRanges.find(chairToRange[start]));
    chairToRange[start] = length;
    allRanges.insert(length);
}
```

```
int main() {
    cin >> M;
    // Pretend the 2 ends are occupied. This reduces case handling.
    addRange(0, M); addRange(M+1, 0);
    cin >> U;
    for (int i = 0; i < U; i++) {
        int q; cin >> q;
        auto it = chairToRange.lower_bound(q);
        int qLength = it->first - q - 1;
        --it;
        int updatedLength = q - it->first - 1;
        addRange(q, qLength);
        updateRange(it->first, updatedLength);
        cout << *allRanges.rbegin() << '\n';
    }
    return 0;
}
```

Area Under Histogram

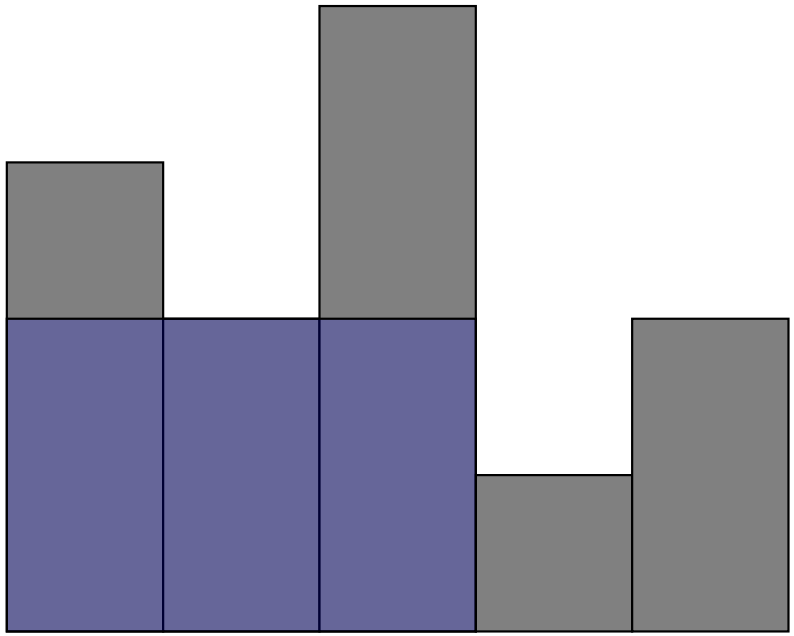
- Many of our data structures work best if data is sorted.
- E.g: we can then chuck them into a set and use `lower_bound`
- Or we can chuck them into a vector and binary search.
- Sometimes we have to work a bit to get this!

- **Problem statement** Given a histogram with n unit-width columns, the i-th with height h_i. What is the largest area of a rectangle that fits under the histogram.
- **Input** The integer $1 \leq n \leq 100,000$ and n numbers, $0 \leq h_i \leq 1,000,000,000$.
- **Output** The largest area of a rectangle that you can fit under the histogram.

- **Sample Input:**

  5

  3 2 4 1 2

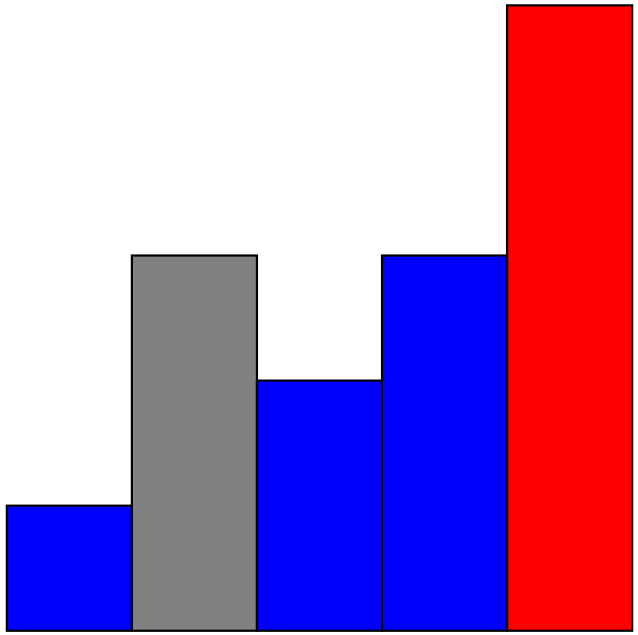


- **Sample Output:** 6

- **Observation 1:** We only care about "maximal" rectangles.
- More formally, they hit some column's roof and can not be extended to the left or right.
- Many angles to approach this problem. Let us focus on one specific column's roof. We now want to find the largest histogram that hits that column's roof.
- **Claim:** We just need to know the first column to its left (and right) that has lower height than it.
- But we need this for all choices of our "specific column". So we will try to do this in a linear sweep and maintain some sort of data structure that can answer this.

- Queries: what is the first column with height $< h$.
- Updates: add a new column (pos, h_{pos}) where pos is greater than all previous positions.
- Multimap? But what can we search on...?
  - If our key is height then we can find a column lower than us. But it is not guaranteed to be the closest one.
  - If our key is position then we can't do anything.
- Heap? Again, same problem (our heap can't do anything a set can't do).

- **Key Observation:** Out of all added columns, we only care about columns that have no lower columns to their right!



- So if we only keep the blue columns in our map then now, the first column in our map lower than us is also the closest column lower than us.

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

int N, h[100005];

// height -> column
map<int, int> importantColumns;

int main() {
    cin >> N;
    for (int i = 0; i < N; i++) cin >> h[i];
    // Reduces cases. Important -2 has lower height than all input heights.
    importantColumns[-2] = -1;
    for (int i = 0; i < N; i++) {
        // find closest column to i's left with lower height.
        auto it = prev(importantColumns.lower_bound(h[i]));
        cout << it->second << '\n';
        // update importantColumns
        while (importantColumns.rbegin()->first >= h[i]) {
            importantColumns.erase(importantColumns.rbegin()->first);
        }
        importantColumns[h[i]] = i;
    }
    return 0;
}
```

- **Complexity?** Dominated by map operations. $O(n)$ calls to `lower_bound`. How about calls to push and pop?
- Each item is pushed and popped once so $O(n)$ calls to both.
- Amortizes to $O(n \log n)$.
- Do this in reverse and add a bit of maths to solve original problem regarding largest rectangle under histogram.
- Could have sped this up to $O(n)$ by using a vector instead of a set and binary searching.
- **Challenge:** There is a beautiful algorithm that does it in one stack sweep in $O(n)$. Essentially the same idea except process a rectangle not at the column where it attains its maximum but at the right end.
- Another famous problem using a similar idea is Longest Increasing Subsequence.

Table of Contents

- A tree is a connected, undirected graph with a unique simple path between any two vertices.
- A *rooted* tree is one with a designated root. All other vertices have a parent, for v, $par[v]$ is the next node in the unique path from v to the root.
- An easy way to represent a rooted tree is to just store this parent array.
- Example: **(Drawn in class)**

Union-Find

- Also called a *system of disjoint sets*
- Used to represent disjoint sets of items.
- Given some set of elements, support the following operations:
  - union(A, B): union the disjoint sets that contain A and B
  - find(A): return a canonical representative for the set that A is in
    - More specifically, we must have find(A) = find(B) whenever A and B are in the same set.
    - It is okay for this answer to change as new elements are joined to a set. It just has to remain consistent across all elements in each disjoint set at a given moment in time.

- **Strategy:** Represent each disjoint set as a rooted tree. For this, we just need to store the parent of each element
- The representative of each rooted tree is the chosen root
- A **find** is just walking up parent edges in the tree until the root is found
- A **union** is just adding an edge between two separate rooted trees.
- Due to how we store our rooted tree, we will add the edge between $root(A)$ and $root(B)$, not directly between A and B
    - The number of edges traversed is equal to the height of the tree, so this is $O(h)$, where h is the maximum height of the tree, which is $O(n)$

- When adding an edge for a union, find the representative for both sets first
- Then, set the parent of the representative for the smaller set to the representative of the larger set
- The maximum height of this tree is now $O(\log n)$
  - When we traverse the edges from any particular element to its parent, we know that the subtree rooted at our current element must at least double in size, and we can double in size at most $O(\log n)$ times
- **find** and **union** are now $O(\log n)$ time, since the height is now $O(\log n)$

- When performing a find operation on some element A, instead of just returning the representative, we change the parent edge of A to whatever the representative was, flattening that part of the tree
- Alone gives an amortised $O(\log n)$ per operation complexity. Proof is nontrivial, omitted.
- Combined with the size heuristic, we get a time complexity of amortised $O(\alpha(n))$ per operation, but the proof is very complicated.

- What is $\alpha(n)$? $\alpha(n)$ is the inverse Ackermann function, a very slow growing function which is less than 5 for $n < 2^{2^{2^{2^{16}}}}$.
- As mentioned, the above two optimisations together bring the time complexity down to amortised $O(\alpha(n))$.
- **Warning:** due to a low-level detail, the path compression optimisation actually significantly slows down the find function, because we lose the tail recursion optimisation, now having to return to each element to update it. This may overshadow the improvement from $O(\log n)$ to $O(\alpha(n))$ depending on bounds of the problem.

Implementation

```
int parent[N];
int subtree_size[N];

void init(int n) {
  for (int i = 0; i < n; i++) {
    parent[i] = i; subtree_size[i] = 1;
  }
}

int root(int x) {
  // only roots are their own parents
  // otherwise apply path compression
  return parent[x] == x ? x : parent[x] = root(parent[x]);
}

void join(int x, int y) {
  // size heuristic
  // hang smaller subtree under root of larger subtree
  x = root(x); y = root(y);
  if (x == y) return;
  if (subtree_size[x] < subtree_size[y]) {
    parent[x] = y;
    subtree_size[y] += subtree_size[x];
  } else {
    parent[y] = x;
    subtree_size[x] += subtree_size[y];
  }
}
```

Table of Contents

The main application of union find.

- Given a graph with N vertices and no edges. Support Q queries of two forms
  - Support updates of **add** an edge between a and b
  - Support queries of, is a connected to b?
- N, Q up to 100,000
- Add updates will be of the form `add a b`
- Connectedness queries will be of the form `q a b`, output 1 if they are connected, 0 otherwise.

Problem: Basic Dynamic Connectivity


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

// TODO: insert your union find implementation here
int root (int u);
void join (int u, int v);

int main() {
    int N, Q;
    cin >> N >> Q;
    for (int q = 0; q < Q; q++) {
        string queryType;
        int a, b;
        cin >> queryType >> a >> b;
        if (queryType == "add") {
            join(a,b);
        } else {
            cout << (root(a) == root(b)) << '\n';
        }
    }
    return 0;
}
```

- **When is it useful?** When you need to maintain which items are in the same set.
- **Main limitation:** You **can not** delete connections, only add them. However, in a lot of natural contexts, this is not a restriction since items in the same set can be treated as the same item.

Table of Contents

- Last main topic is data structures that support operations on a range
- Why do we care about this?
  - Pragmatic answer: Impossible to just support arbitrary queries and updates. Meanwhile there is a lot of interesting stuff we can do with ranges.
  - But also, a lot of useful applications by itself. Naturally, a lot of what we care about comes in the form of ranges. E.g: ranges of numbers, a range in an array, linear sweeps often result in caring about ranges...

- Given N integers $a_0, a_1, \ldots, a_{N-1}$, answer queries of the form:

$$\sum_{i=l}^{r-1} a_i$$

  for given pairs l, r.
- N is up to $100,000$.
- There are up to $100,000$ queries.
- We can't answer each query naïvely, we need to do some kind of precomputation.

- **Algorithm** Construct an array of prefix sums.
- $b_0 = a_0$.
- $b_i = b_{i-1} + a_i$.
- This takes $O(N)$ time.
- Now, we can answer every query in $O(1)$ time.
- This works on any "reversible" operation. That is, any operation $A \star B$ where if we know $A \star B$ and A, we can find B.
- This includes addition and multiplication, but *not* max or gcd.

Problem: Range Max Queries

- Given N integers $a_0, a_1, \ldots, a_{N-1}$, answer queries of the form:

  $$\max(a[l, r))$$

  for given pairs l, r.
- N is up to $100,000$.
- There are up to $100,000$ queries.
- We can't answer each query naïvely, we need to do some kind of precomputation.
- Also can't take prefix sums as knowing $\max([a_0, \ldots, a_l))$ and $\max([a_0, a_r))$ says almost nothing about $\max([a_l, a_r))$.

- Max is not "reversible" but it does have a nice property: *idempotence*. This just means $\max(x, x) = x$, i.e: I can apply max as many times as I want to the same element, it does not do anything.
- This translates to the following property. If $l \leq s \leq t \leq r$ then

$$\max([a_l, a_r)) = \max(\max([a_l, a_t)), \max([a_s, a_r)))$$

- So we want to precompute the max of a bunch of intervals, such that $[a_l, a_r)$ is always the union of two of these intervals.

- **Key Idea:** Instead of precomputing prefix sums, precompute all intervals whose lengths are powers of 2.
- This can be done quickly since an interval of length 2^k is the union of two intervals of length 2^{k-1}.

Precomp Implementation

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

const int MAXN = 100000;
const int LOGN = 18;

// sparseTable[l][i] = max a[i...i+2^l)
int sparseTable[LOGN][MAXN];

int main() {
    // Input the initial array
    for (int i = 0; i < MAXN; i++) cin >> sparseTable[0][i];
    for (int l = 1; l < LOGN; l++) {
        int prevp2 = 1 << (l-1);
        for (int i = 0; i < MAXN; i++) {
            int intEnd = i + prevp2;
            if (intEnd < MAXN)
                sparseTable[l][i] = max(sparseTable[l-1][i],
                                        sparseTable[l-1][intEnd]);
            else
                sparseTable[l][i] = sparseTable[l-1][i];
        }
    }
}
```

- Suppose we now want $\max[a_l, \ldots, a_{l+s})$.
- Let $p = 2^t$ be the largest power of 2 that is $\leq s$.
- **Key Observation:**

$$l \leq (l + s) - p \leq l + p \leq l + s$$

since $p \leq s \leq 2p$.

- **Hence:**

$$\max([a_l, a_{l+s})) = \max(\max([a_l, a_{l+p})), \max([a_{l+s-p}, a_{l+s})))$$

- But we've precomputed these internal maxes since they have length a power of 2!

Query Implementation

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

const int MAXN = 100000, LOGN = 18;
// sparseTable[l][i] = max a[i...i+2^l)
int N, sparseTable[LOGN][MAXN], log2s[MAXN];

void precomp() {
    // TODO: Insert precomp of sparse table here.
    for (int i = 2; i < MAXN; i++) log2s[i] = log2s[i/2] + 1;
}

int main() {
    // Input the initial array
    cin >> N;
    for (int i = 0; i < N; i++) cin >> sparseTable[0][i];
    precomp();

    int Q; cin >> Q;
    for (int q = 0; q < Q; q++) {
        int l, r; cin >> l >> r;
        // Problem: Find max of a[l...r)
        int l2 = log2s[r-l];
        int res = max(sparseTable[l2][l], sparseTable[l2][r-(1<<l2)]);
        cout << res << '\n';
    }
    return 0;
}
```

Sparse Tables

- **Complexity?** $O(N \log N)$ precomp, $O(1)$ per query.
- **Warning:** You need your operation to be idempotent. E.g: this will double count for sum, multiply, count, etc...
- Works for max, min, gcd, lcm.
- Practically, don't see it too often. But a nice idea and the data structure for Lowest Common Ancestor is similar.

Problem: Updates

- **Problem** with earlier data structures: They do not support updates.

Problem: Updates

New problem:

- Given N integers $a_0, a_1, \ldots, a_{N-1}$, answer queries of the form:

$$\sum_{i=l}^{r-1} a_i$$

  for given pairs l, r.
- **But** there are now also updates of the form, set:

$$a_i := k$$

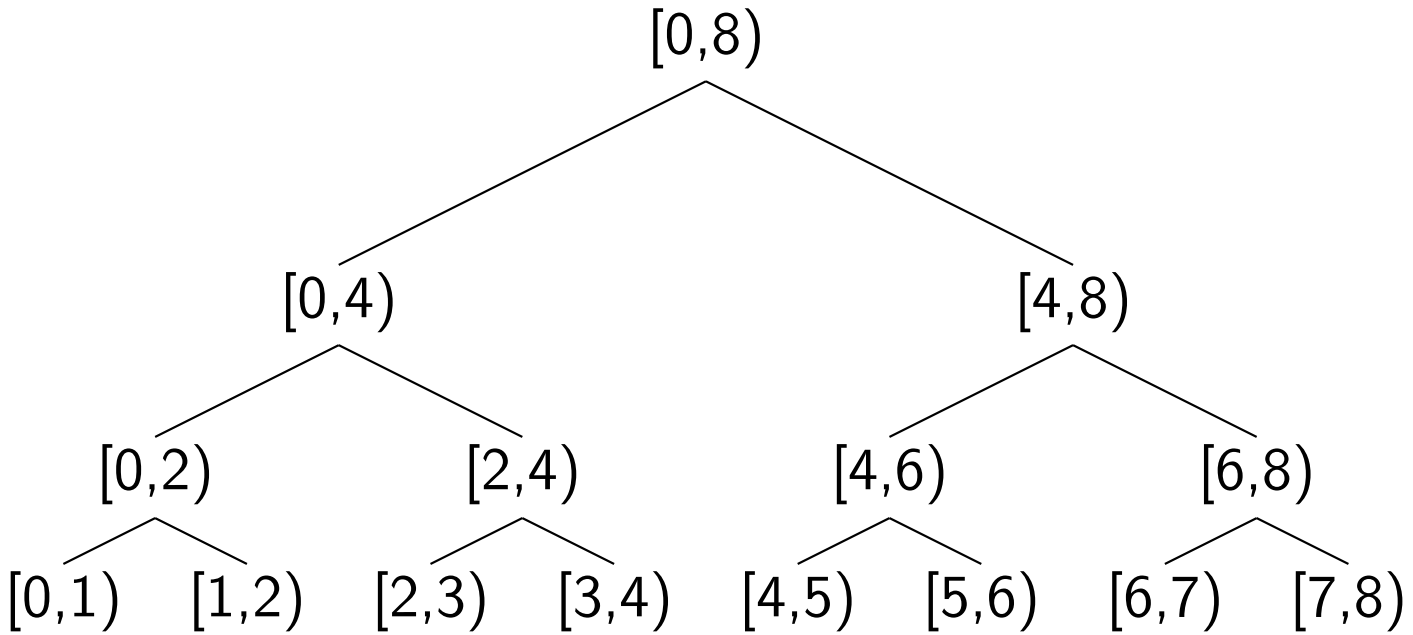
- N is up to $100,000$.
- There are up to $100,000$ queries and updates total.

Problem: Updates

- Recomputing the prefix sums will take $O(N)$ time per update, so our previous solution is now $O(N^2)$ for this problem, which is too slow.
- Let's try to find a solution that slows down our queries but speeds up updates in order to improve the overall complexity.

- The problem with our earlier data structures is there are too many ranges containing any given value, updating all of them is $O(N)$ per update.
- The problem with just storing the array is there are not enough ranges, querying is $O(N)$.
- We need a new way to decompose $[1, N]$ into ranges that is a healthy balance.

- We will make a tree. Each node in the tree is responsible for a range.

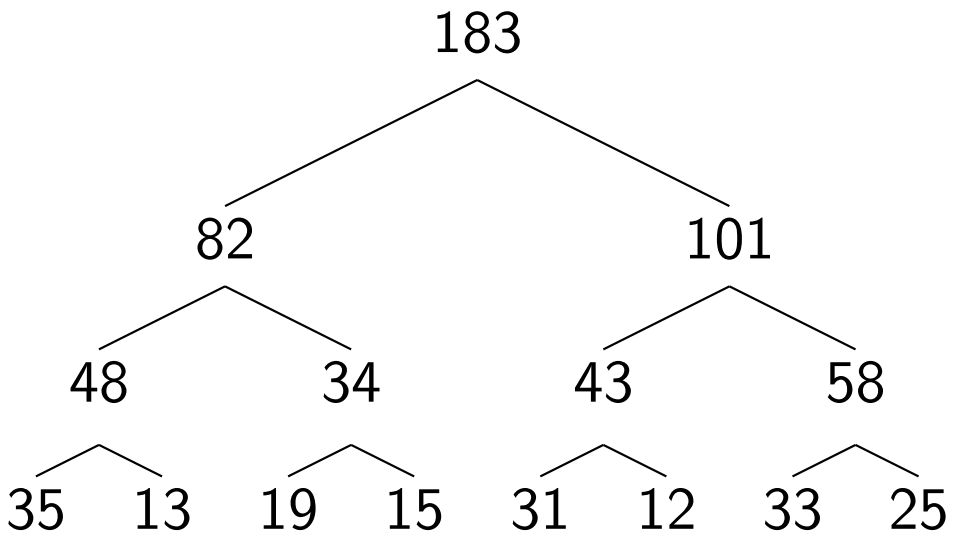


- The array itself goes into the leaves.
- The internal nodes store information on the range.
- This depends on problem. For our earlier problem, we would want each node to store the sum of that range.

- Consider the array

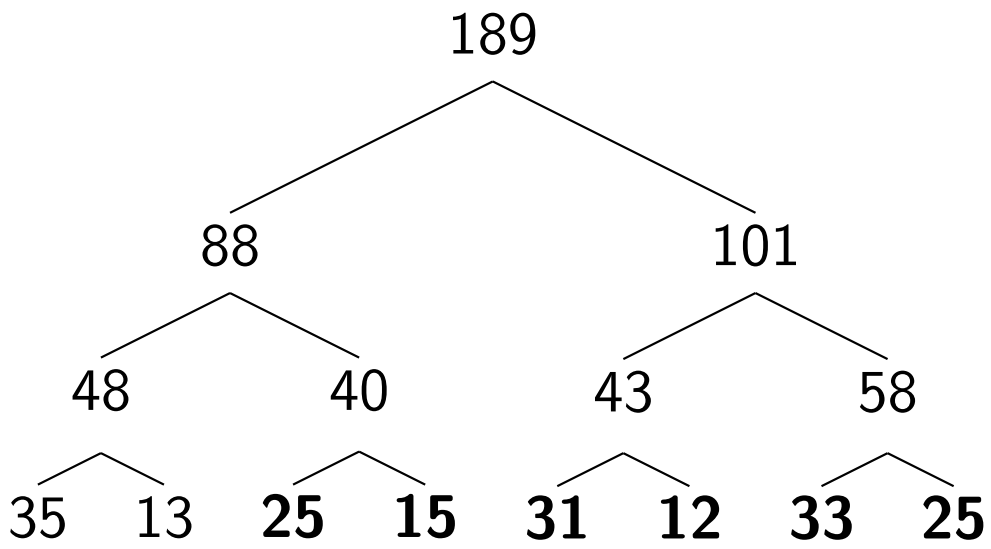
$$[35, 13, 19, 15, 31, 12, 33, 23]$$

- We would get the tree



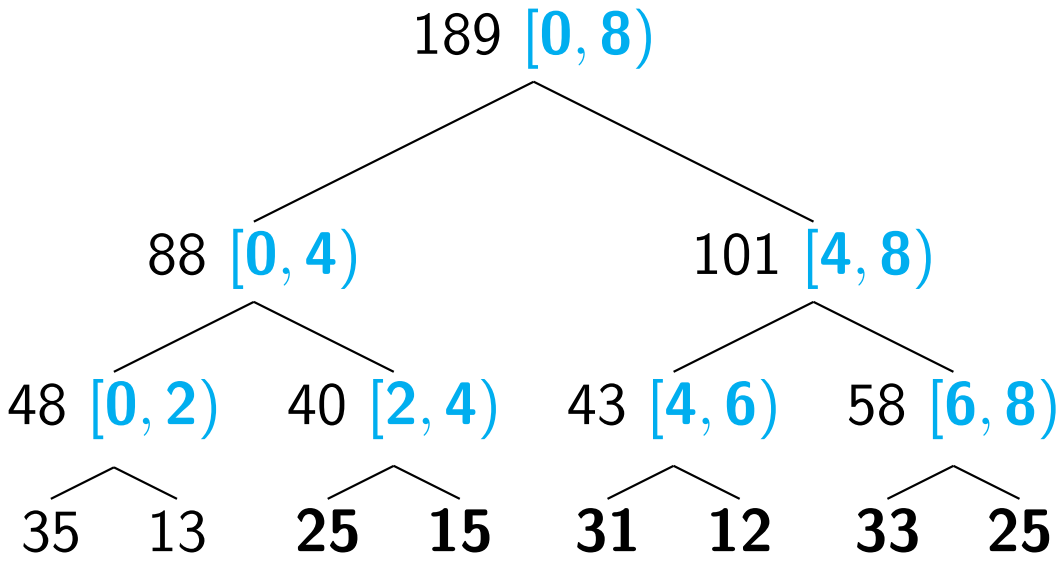
- Note, the leaves store the array and every other node is just the sum of its two children.

- Let's query the sum of $[2, 8)$ (inclusive-exclusive).



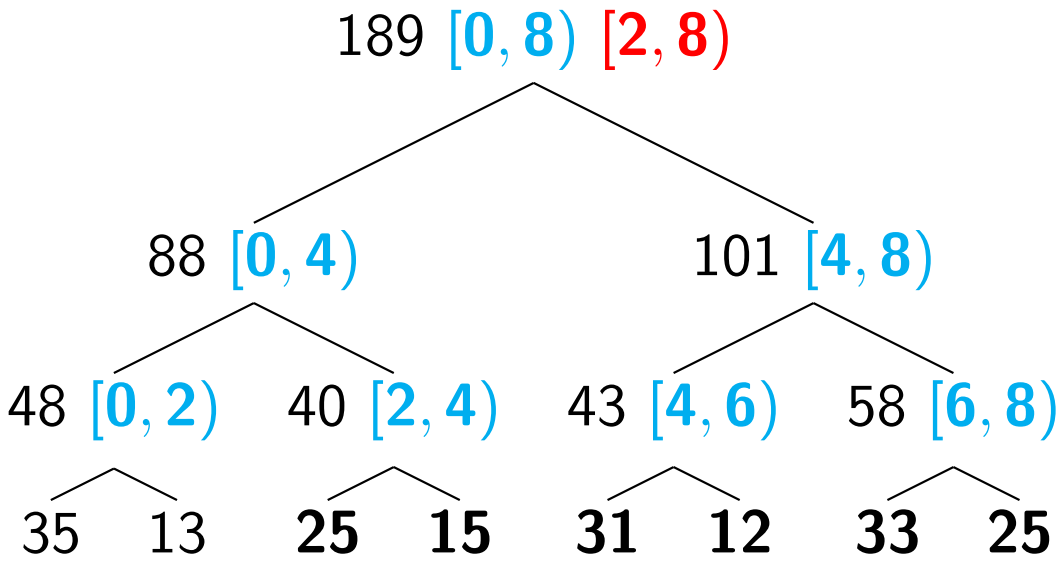
- Recall each node in the tree has a "range of responsibility".

- Let's query the sum of $[2, 8)$ (inclusive-exclusive).



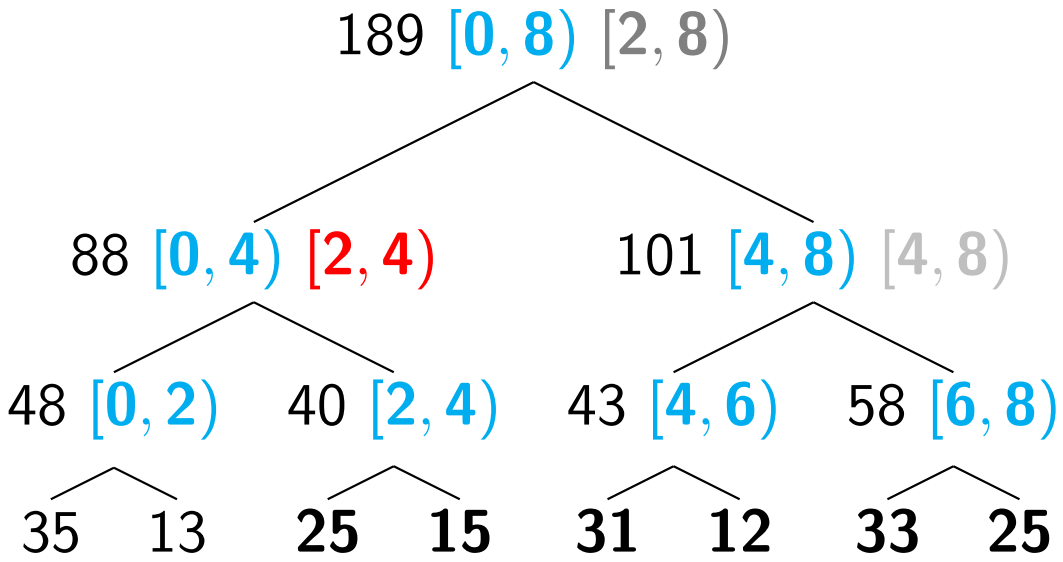
- Our goal is the same as in the sparse table, find a set of ranges whose disjoint union is $[2, 8)$. Then taking the sum of those nodes gives us the answer.

- Let's query the sum of $[2, 8)$ (inclusive-exclusive).



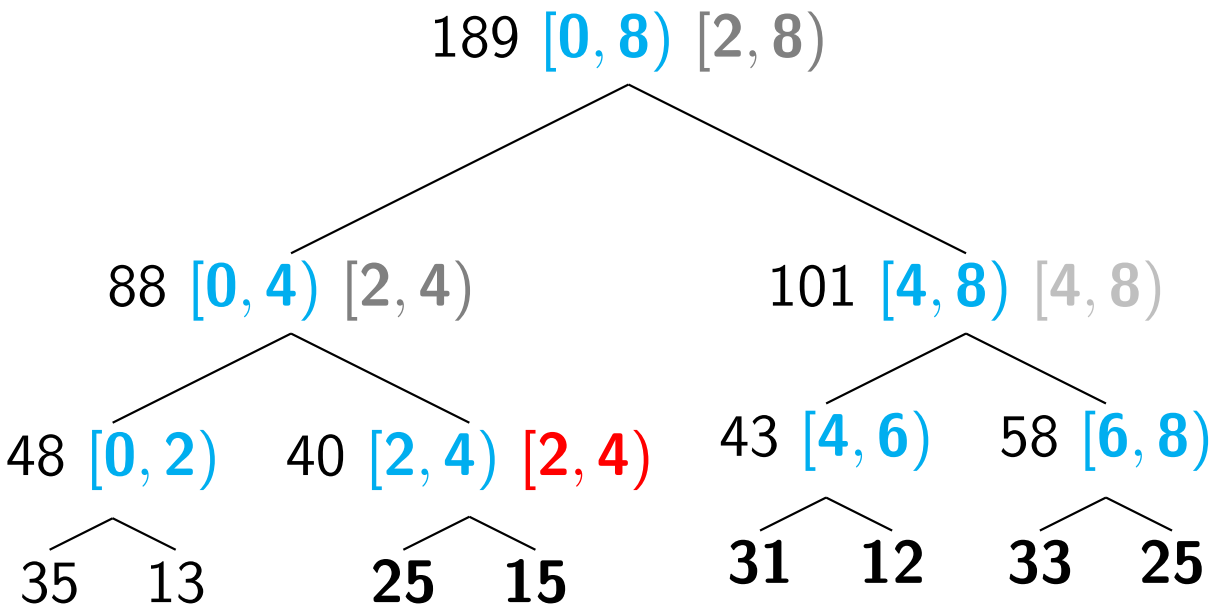
- We start at the top of the tree, and 'push' the query range down into the applicable nodes.

- Let's query the sum of $[2, 8)$ (inclusive-exclusive).



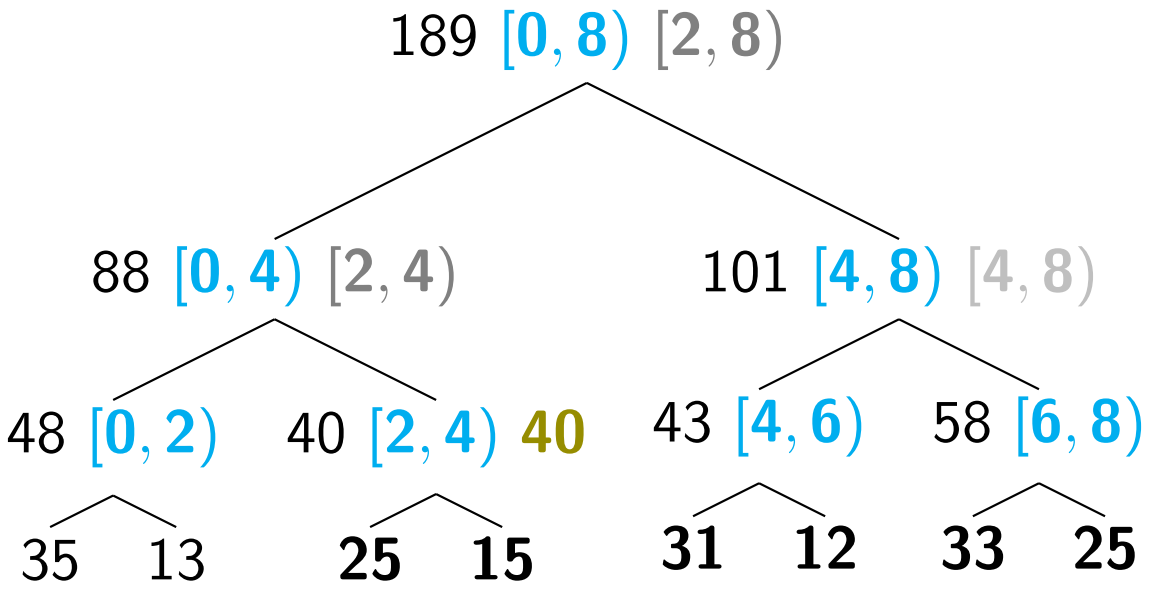
- This is a recursive call, so we do one branch at a time. Let's start with the left branch.

- Let's query the sum of $[2, 8)$ (inclusive-exclusive).



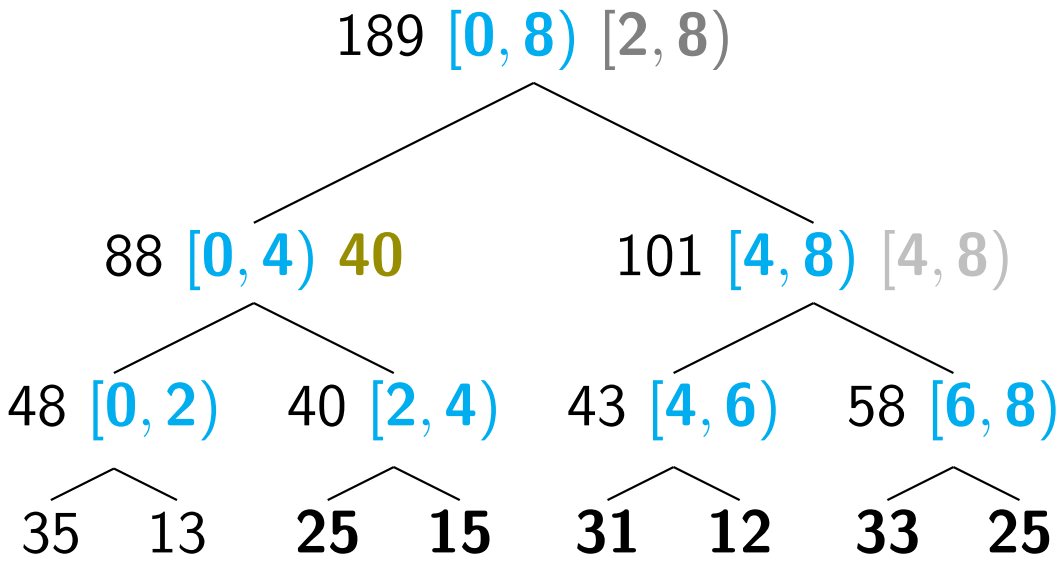
- There is no need to continue further into the left subtree, because it doesn't intersect the query range.

- Let's query the sum of $[2, 8)$ (inclusive-exclusive).



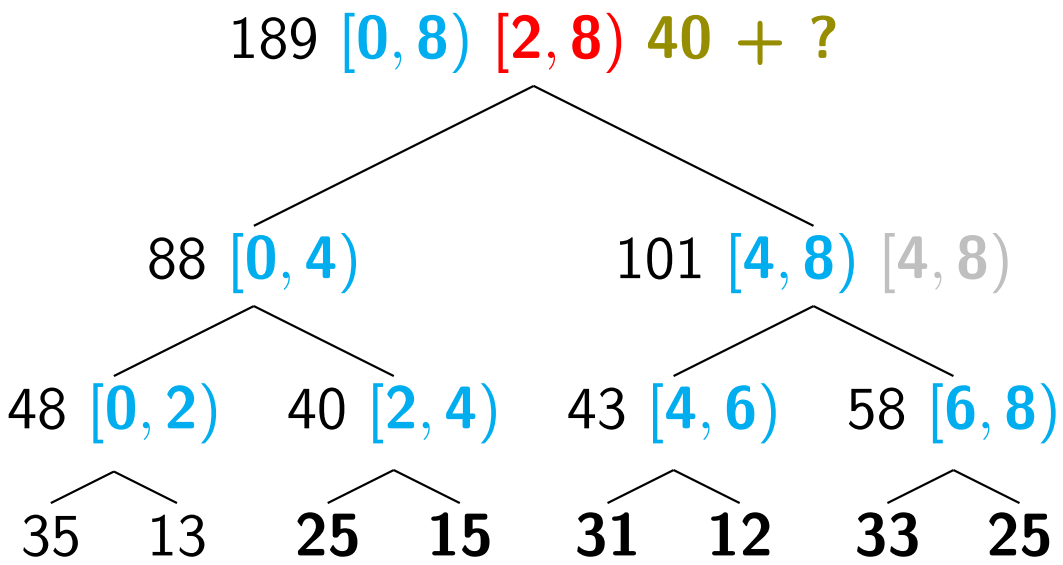
- There is also no need to continue further down, because this range is equal to our query range.

- Let's query the sum of $[2, 8)$ (inclusive-exclusive).



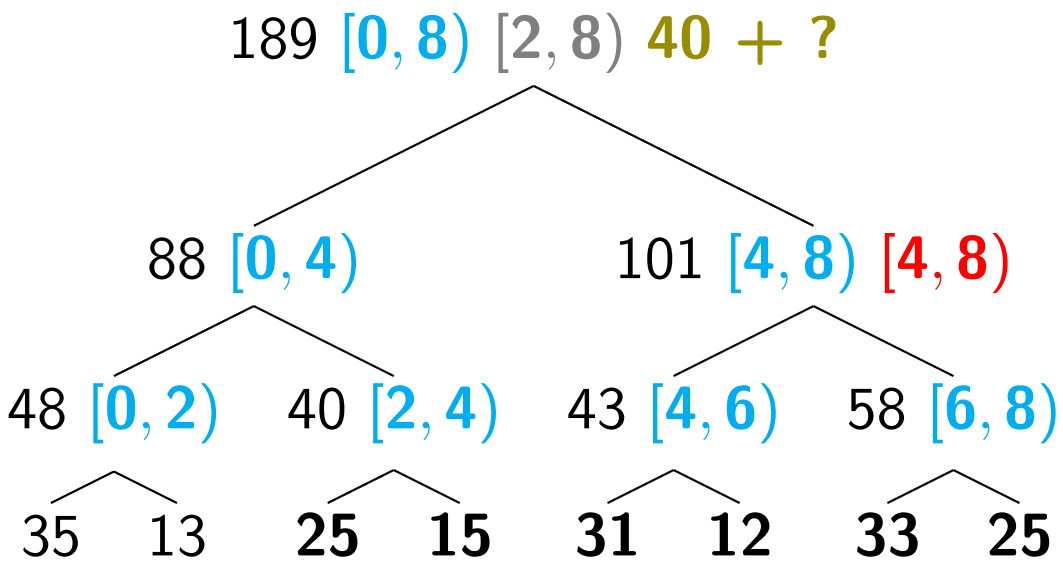
- We return the result we have obtained up to the chain, and let the query continue.

- Let's query the sum of $[2, 8)$ (inclusive-exclusive).



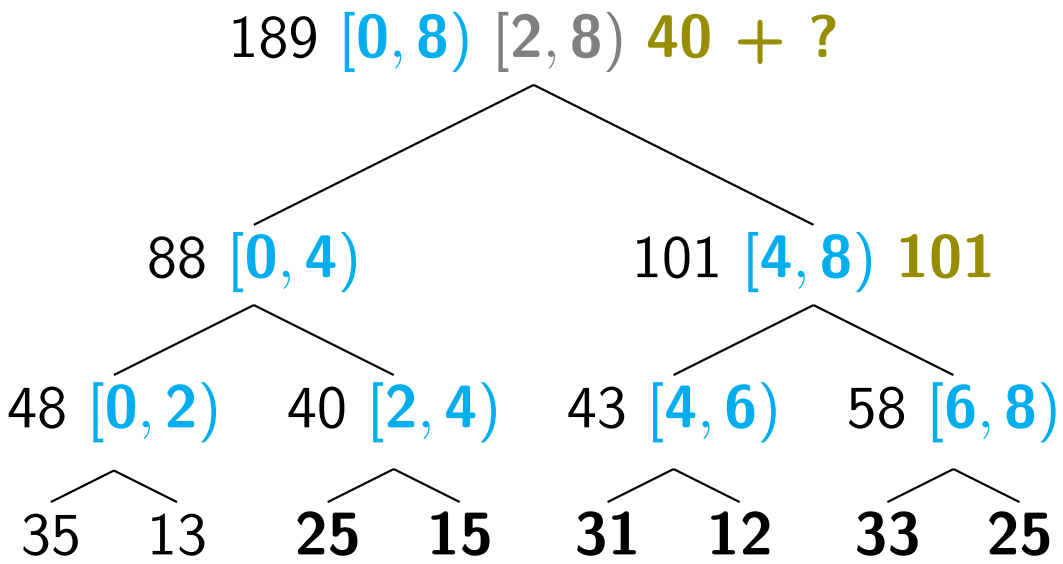
- We return the result we have obtained up to the chain, and let the query continue.

- Let's query the sum of $[2, 8)$ (inclusive-exclusive).



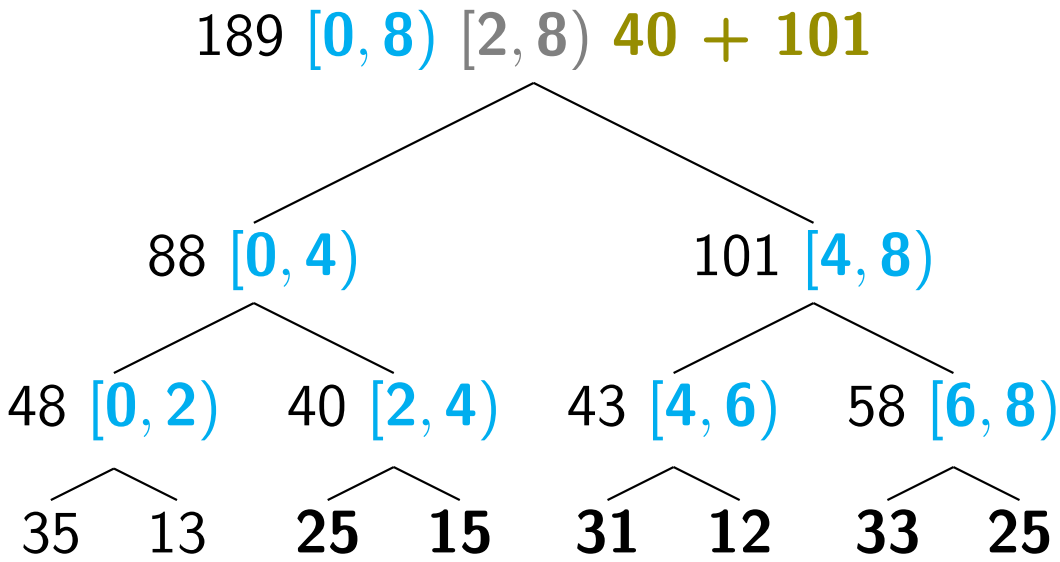
- Now, it is time to recurse into the other branch of this query.

- Let's query the sum of $[2, 8)$ (inclusive-exclusive).



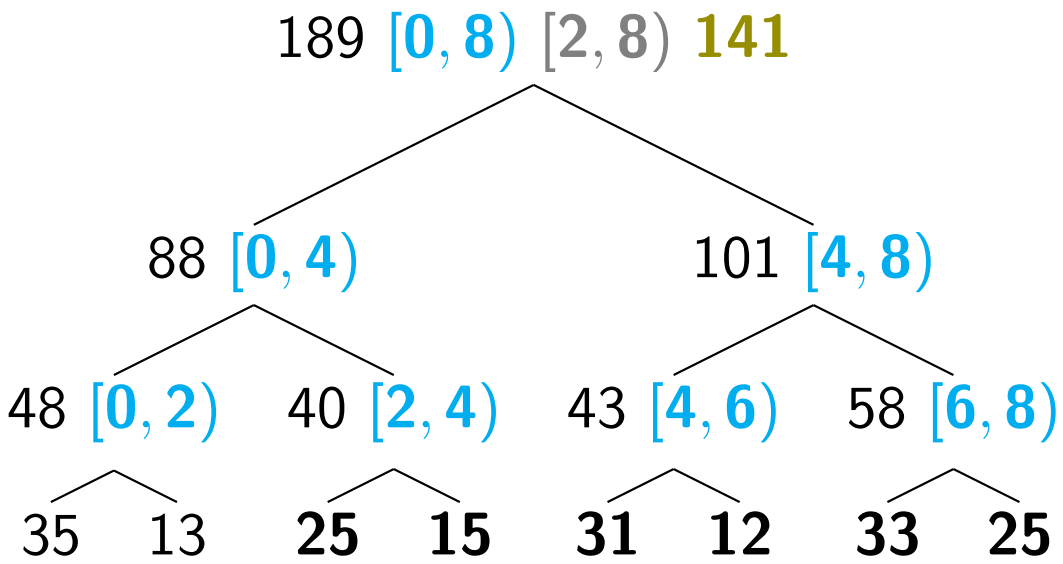
- Here, the query range is equal to the node's range of responsibility, so we're done.

- Let's query the sum of $[2, 8)$ (inclusive-exclusive).



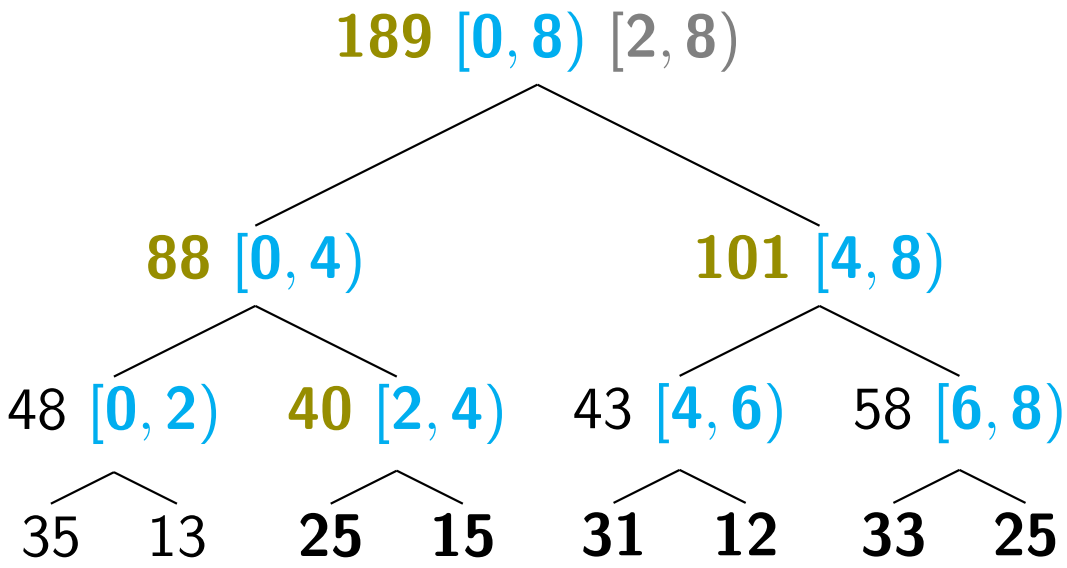
- Here, the query range is equal to the node's range of responsibility, so we're done.

- Let's query the sum of $[2, 8)$ (inclusive-exclusive).



- Now that we've obtained both results, we can add them together and return the answer.

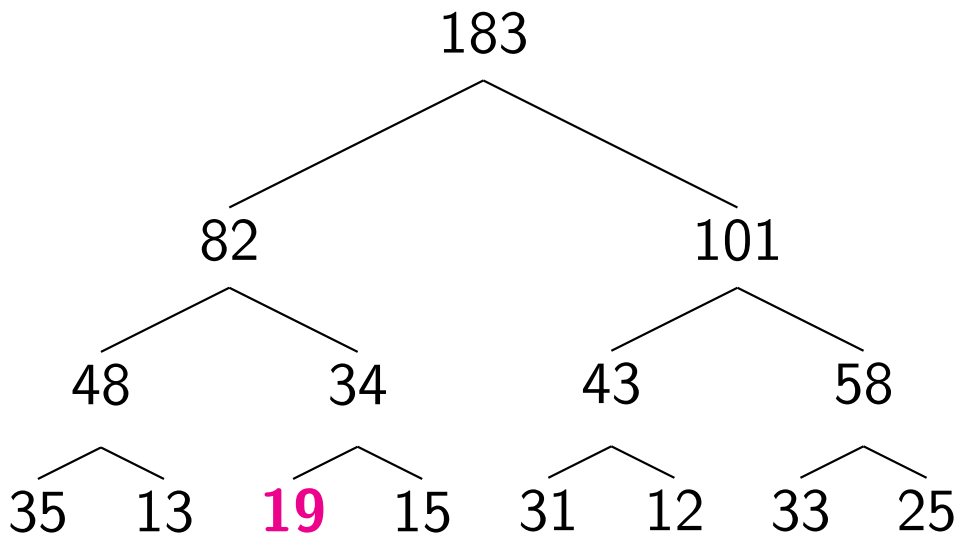
- We didn't visit many nodes during our query.



- In fact, because only the left and right edges of the query can ever get as far as the leaves, and ranges in the middle stop much higher, we only visit $O(\log N)$ nodes during a query.

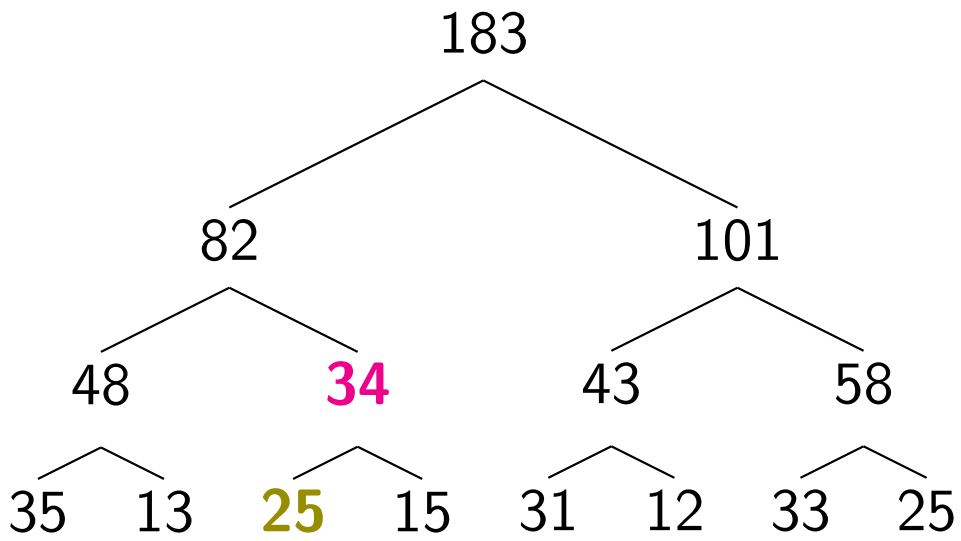
- One way to see this is consider cases based on if the query range shares an endpoint with the current node's range of responsibility.
- Another way is to consider starting with the full range from the bottom and going up.
- Probably easiest if you play around a bit and convince yourself of this fact.

- Let's update the element at index 2 to 25.

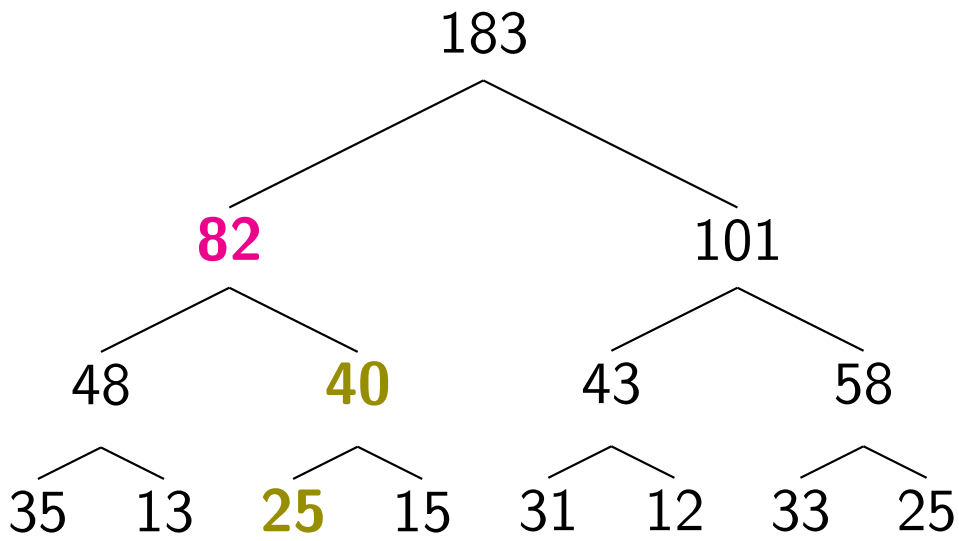


Updates

- Let's update the element at index 2 to 25.

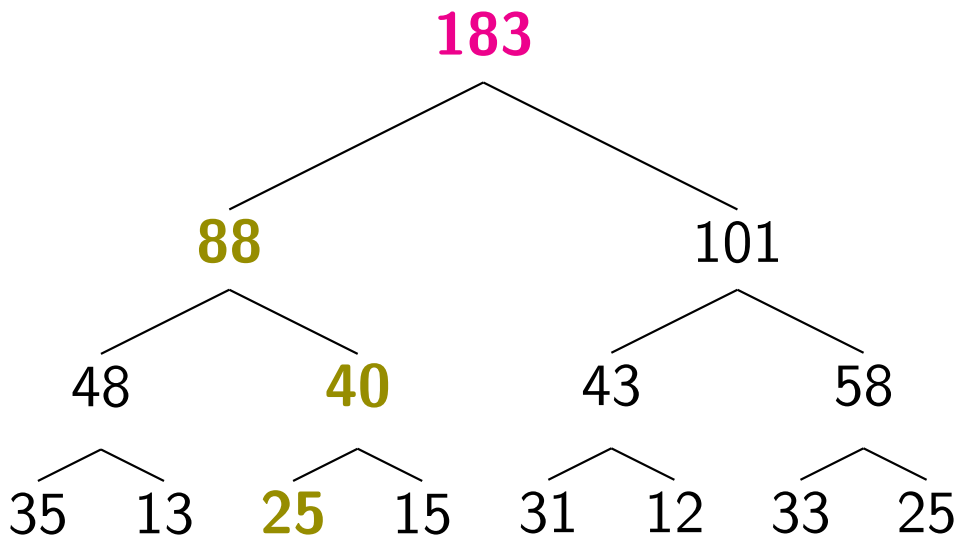


- Let's update the element at index 2 to 25.

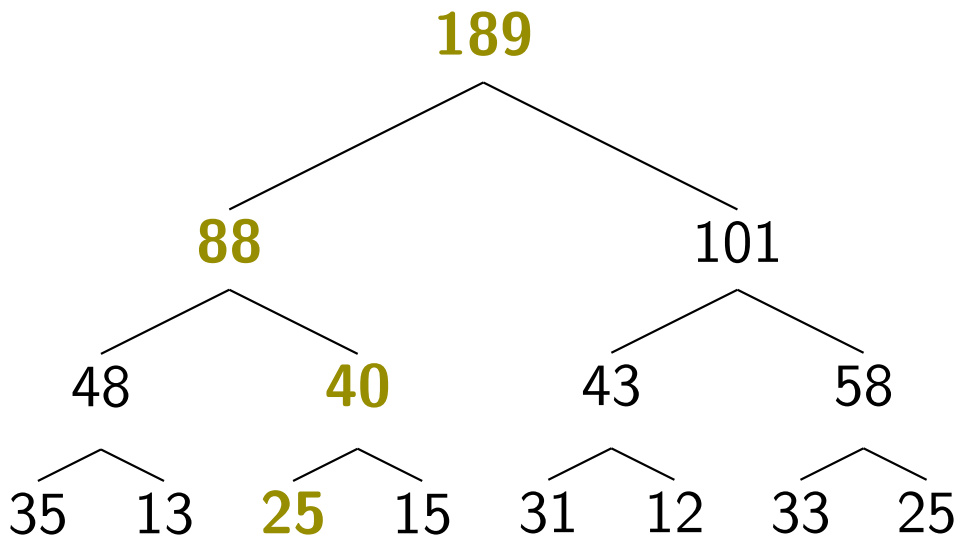


Updates

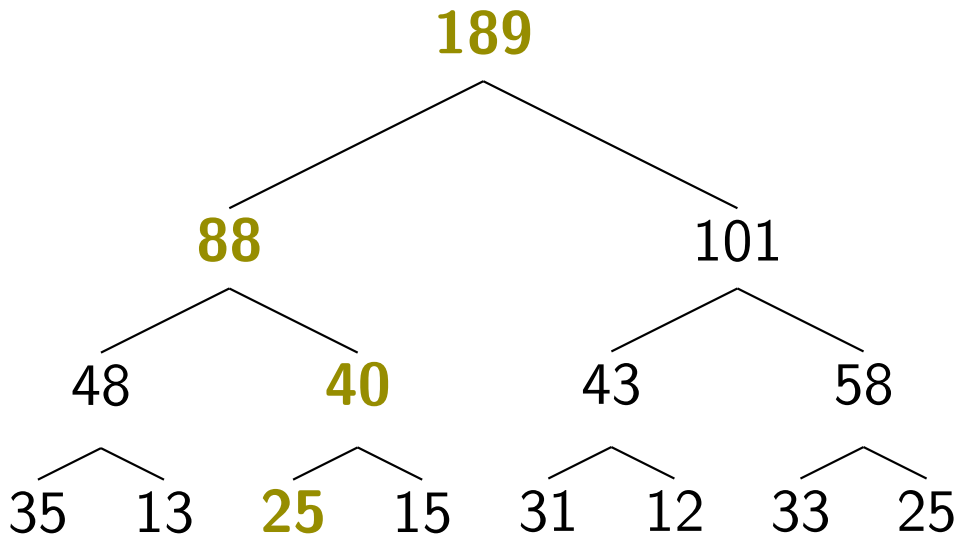
- Let's update the element at index 2 to 25.



- Let's update the element at index 2 to 25.



- Let's update the element at index 2 to 25.



- We always construct the tree so that it's balanced, then its height is $O(\log N)$.
- Thus, updates take $O(\log N)$ time.

- Thus we have $O(\log N)$ time for both updates and queries.
- This data structure is commonly known as a range tree, segment tree, interval tree, tournament tree, etc.
- The number of nodes we add halves on each level, so the total number of nodes is still $O(N)$.
- For ease of understanding, the illustrations used a *full* binary tree, which always has a number of nodes one less than a power-of-two. This data structure works fine as a *complete* binary tree as well (all layers except the last are filled). This case is harder to imagine conceptually but the implementation works fine, for each internal node just split the range of responsibility by the average.
- All this means is that padding out the data to the nearest power of two is not necessary.

- Since these binary trees are complete, they can be implemented using the same array-based tree representation as with an array heap
  - Place the root at index 0. Then for each node i, its children (if they exist) are $2i + 1$ and $2i + 2$.
  - Alternatively, place the root at index 1, then for each node i the children are $2i$ and $2i + 1$.
- This works with any binary associative operator, e.g.
  - min, max
  - sum
  - gcd
  - merge (from merge sort)
    - For a non-constant-time operation like this one, multiply the complexity of all range tree operations by the complexity of the merging operation.

- We can extend range trees to allow range updates in $O(\log N)$ using *lazy propagation*
- The basic idea is similar to range queries: push the update down recursively into the nodes whose range of responsibility intersects the update range.
- However, to keep our $O(\log N)$ time complexity, we can't actually update every value in the range.
- Just like we returned early from queries when the query range matched a node's entire range, we cache the update at that node and return without actually applying it.
- When a query or a subsequent update is performed which visits this node you might need to push the cached update one level further down.
- Will talk more about this in 2 weeks time.

- **Implementation (updates)**

```
#define MAX_N 100000
// the number of additional nodes created can be as high as the next
      power of two up from MAX_N (131,072)
int tree[266666];

// a is the index in the array. 0- or 1-based doesn't matter here, as
      long as it is nonnegative and less than MAX_N.
// v is the value the a-th element will be updated to.
// i is the index in the tree, rooted at 1 so children are 2i and 2i
      +1.
// instead of storing each node's range of responsibility, we
      calculate it on the way down.
// the root node is responsible for [0, MAX_N)
void update(int a, int v, int i = 1, int start = 0, int end = MAX_N) {
  // this node's range of responsibility is 1, so it is a leaf
  if (end - start == 1) {
    tree[i] = v;
    return;
  }
  // figure out which child is responsible for the index (a) being
        updated
  int mid = (start + end) / 2;
  if (a < mid) update(a, v, i * 2, start, mid);
  else update(a, v, i * 2 + 1, mid, end);
  // once we have updated the correct child, recalculate our stored
        value.
  tree[i] = tree[i*2] + tree[i*2+1];
}
```

- **Implementation (queries)**

```
// query the sum in [a, b)
int query(int a, int b, int i = 1, int start = 0, int end = MAX_N) {
  // the query range exactly matches this node's range of
       responsibility
  if (start == a && end == b) return tree[i];
  // we might need to query one or both of the children
  int mid = (start + end) / 2;
  int answer = 0;
  // the left child can query [a, mid)
  if (a < mid) answer += query(a, min(b, mid), i * 2, start, mid);
  // the right child can query [mid, b)
  if (b > mid) answer += query(max(a, mid), b, i * 2 + 1, mid, end);
  return answer;
}
```

- **Implementation (construction)**
  - It is possible to construct a range tree in $O(N)$ time, but anything you use it for will take $O(N \log N)$ time anyway.
  - Instead of extra code to construct the tree, just call `update` repeatedly for $O(N \log N)$ construction.

- **Problem statement** Given an array of integers, find the longest (strictly) increasing (not necessarily contiguous) subsequence.
- **Input** N, the size of the array, followed by N integers, a_i. $1 \leq N, a_i \leq 100,000$.
- **Output** A single integer, the length of the longest increasing subsequence.

- **Example Input**

  5
  4 0 3 2 8

- **Example Output** 3
- **Explanation**: Either the subsequence $0, 3, 8$ or $0, 2, 8$.

Example problem: LIS

- We will compute this iteratively, for each index find the longest increasing subsequence that ends at that index.
- Let's say we store this in an array $best[0 \ldots N)$.
- Let us try to sweep from left to right, pretty natural order for subsequences.
- How do we compute $best[i]$?
- Either $best[i] = 1$ or it extends an existing subsequence.
- The subsequence we extend must be the longest one ending at some $j < i$ where $\mathbf{a[j]} < \mathbf{a[i]}$.
- Note: the **bolded** part is a range requirement (the other condition is also a range requirement but is handled by the sweep order).

- So we want to query, out of all j where $\mathbf{a[j]} < \mathbf{a[i]}$, what is the **max** value of $best[j]$.
- **Solution:** Range tree over the values $a[j]$!
- As we sweep, we maintain a range tree, over the array

$$bestWithEnd[h]$$

  where $bestWithEnd[h]$ is defined to be the longest subsequence we can make so far whose last value is h.
- Then to process a new index, i, the longest subsequence ending at i is

$$\max([bestWithEnd[1], bestWithEnd[a[i]]]) + 1$$

- Finally, we update $bestWithEnd[a[i]]$ with $best[i]$.

Example problem: LIS


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

const int MAXVAL = 100000;
// TODO: Add your range tree code here.
// The range tree should initially start with all values 0 and support:
void update(int index, int val);
int query(int a, int b); // Returns the minimum of the range [a,b)

int main() {
    int N;
    cin >> N;
    for (int i = 0; i < N; i++) {
        int cVal; cin >> cVal;
        int best = query(0, cVal);
        update(cVal, best + 1);
    }
    cout << query(0, MAXVAL + 1) << endl;
    return 0;
}
```

- **Complexity?** N range tree queries and updates, each $O(\log N)$. Total: $O(N \log N) \approx O(100,000 \cdot 17)$.
- **Moral:** When trying to solve a problem, be on the lookout for suboperations that might be sped up by data structures. Often take the form of needing to support simple queries.
- Also it is useful to consider range trees over values, not just indices.
- The bound $h_i \leq 100,000$ was not necessary, only the relative order of the h_i values mattered. So we could have sorted them and replaced each with its rank in the resulting array. Sometimes called "coordinate compression".

- Alternatively, instead of doing it from left to right, one can do it in increasing order of values in the array. Then your range tree is over indices not values, and your queries become "what is the largest value in $best[0, \ldots, i)$.
- There is also a really elegant solution with a left to right sweep and a sorted stack. Let $minEnd[i]$ store the minimum end value for a subsequence of length i. This is a sorted array and we can update it in $O(\log N)$ time with binary search.