Data Structures II

COMP4128 Programming Challenges

School of Computer Science and Engineering
UNSW Australia

Table of Contents

Problem: Range Queries

- 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.

Problem: Range Queries

- **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: Updates

- We can now receive updates mixed in with the queries. The updates are of the form

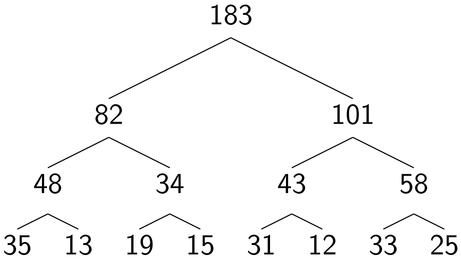
$$a_i := k$$

for given i and k.

- There can still be up to $100,000$ updates and queries in total. 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.
- As we have done in the past, we try to find a solution that slows down our queries but speeds up updates in order to improve the overall complexity.

Let's make a tree

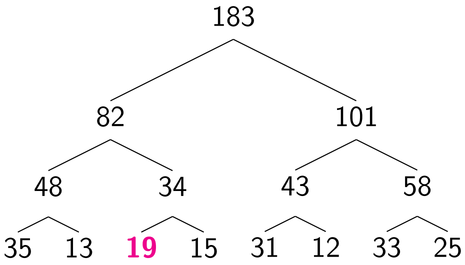
- So far, we've seen trees where each node contains a value we care about. This tree is a little different:



- We only store our data in the leaves. The rest of the tree is intermediate computations.
- Each parent stores the sum of its children.
- Now what? Trees are great, but how do we perform our updates and queries?

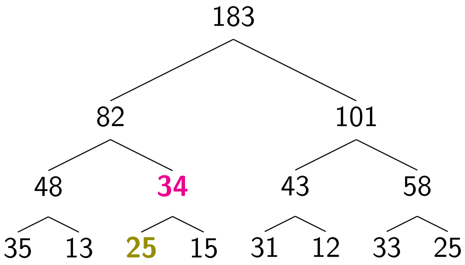
Updates

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

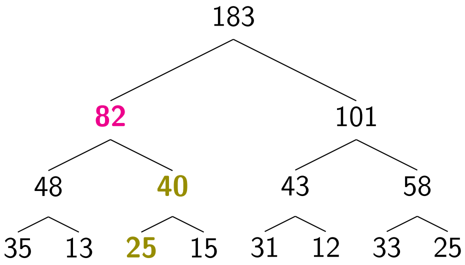


Updates

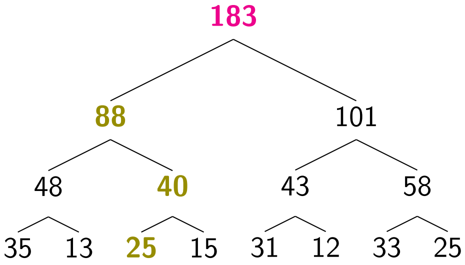
- Let's update the element at index 2 to 25.



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

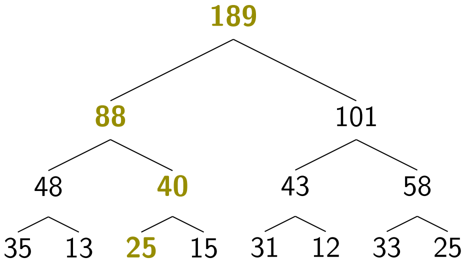


- Let's update the element at index 2 to 25.



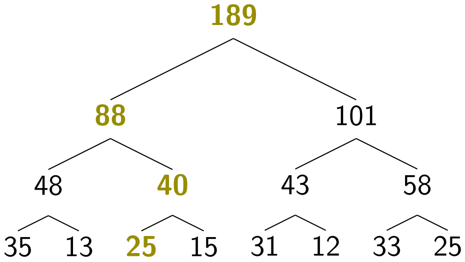
Updates

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



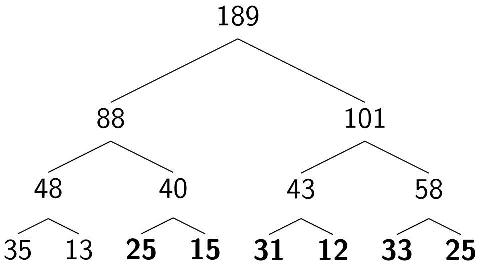
Updates

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



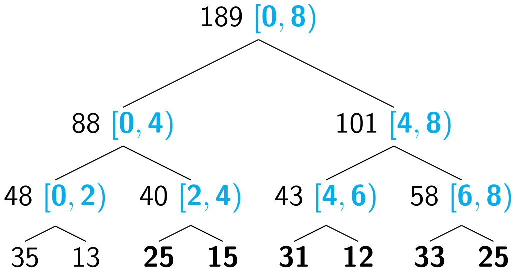
- We always construct the tree so that it's balanced, then its height is $O(\log N)$.
- Thus, updates take $O(\log N)$ time.
- Still, all of this is useless if we can't actually query the tree fast. How do we do that?

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



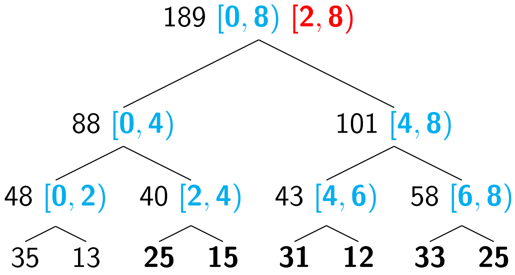
- Each node in the tree has a "range of responsibility".

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



- Each node in the tree has a "range of responsibility", split evenly between its children.

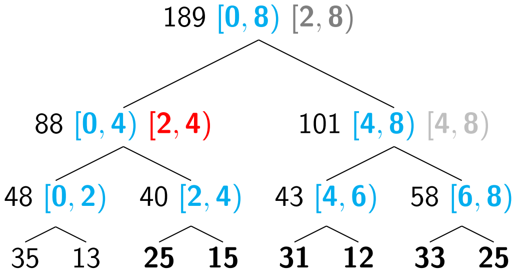
- Let's query the sum of $[2, 8)$ (inclusive-exclusive).



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

Queries

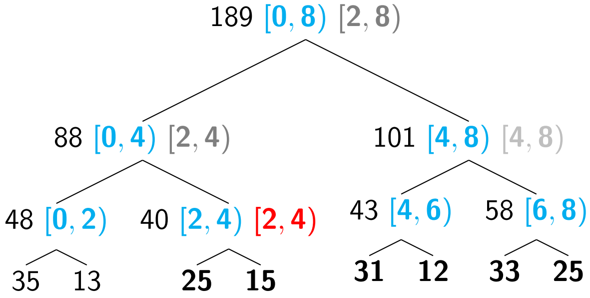
- Let's query the sum of $[2,8)$ (inclusive-exclusive).



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

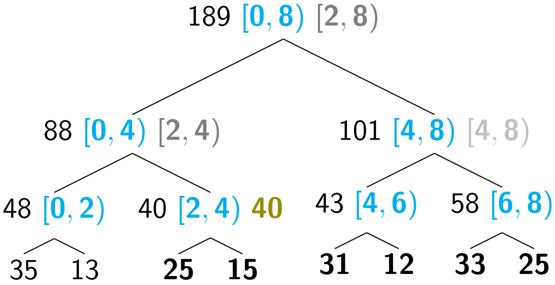
Queries

- 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.

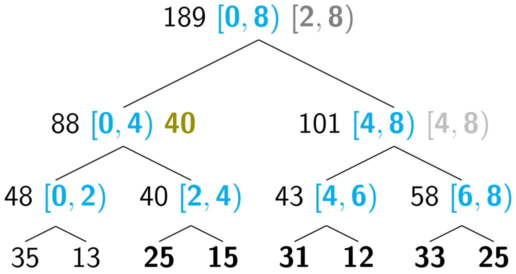
- Let's query the sum of $[2, 8)$ (inclusive-exclusive).



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

Queries

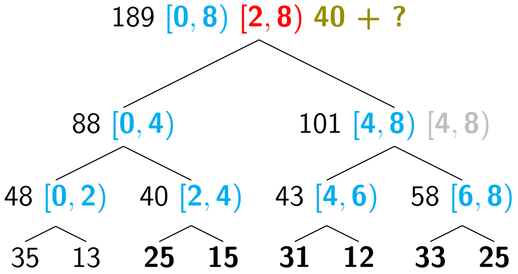
- Let's query the sum of $[2, 8)$ (inclusive-exclusive).



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

Queries

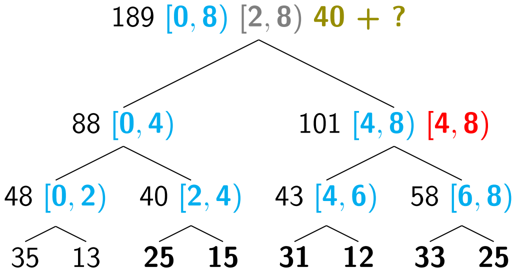
- Let's query the sum of $[2, 8)$ (inclusive-exclusive).



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

Queries

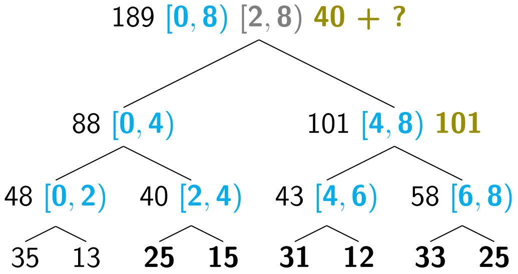
- Let's query the sum of $[2, 8)$ (inclusive-exclusive).



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

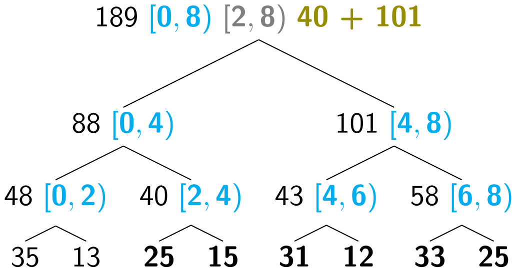
Queries

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



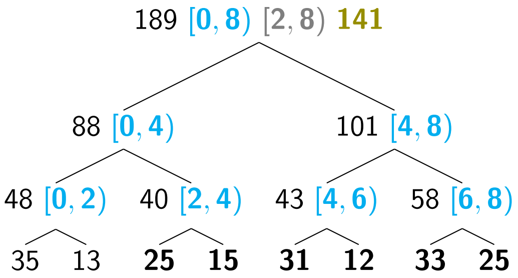
- Here, the query range is equal to the node's range of responsibility, so we're done.

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



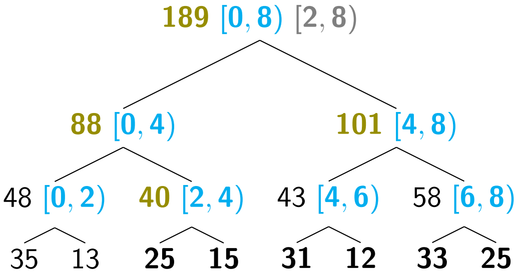
- Here, the query range is equal to the node's range of responsibility, so we're done.

- 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.

Range Tree

- 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.
- All this means is that padding out the data to the nearest power of two is not necessary.

Range Tree

- 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.

Range Tree

- 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.
- Whenever a query or a subsequent update is performed which visits this node, just push the cached update one level further down.
- This is fiddly to implement as updates pile up and often need to combine. Implementation left as an exercise :)

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];
}
```

Data Structures II

Range Queries and Updates

Lowest Common Ancestor

Binary function composition

- **Implementation (queries)**

```
// query the sum in [a, b)
int query(int a, int b, int i = 1, int start = 1, 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** Gilderoy Lockhart has fallen upon hard times, and now finds himself performing magic tricks to entertain Muggles. In his newest trick, he places n cards face down on a table, and turns to face away from the table. He then invites q members of the audience to do either of the following moves:
  1. announce two numbers i and j, and flip all cards between i and j inclusive, or
  2. ask him whether a particular card i is face up or face down.

  True to form, Lockhart is unable to do this trick himself, so write a program to help him!
- **Input** The numbers n and q, each up to 100,000, followed by q lines either of the form `F i j` ($1 \leq i \leq j \leq n$), a flip, or `Q i` ($1 \leq i \leq n$), a query.
- **Output** For each query, print "Face up" or "Face down".

Example problem: Card Trick

- Observe that we can just keep track of how many times each card was flipped; the parity of this number determines whether it is face up or face down.
- We need to handle flips in faster than linear time.
- If we handle flips by adding 1 at the left endpoint and subtracting 1 at the right endpoint, then the sum up to card i (the prefix sum) is the number of times that card i has been flipped.

- **Algorithm** Construct a range tree. For the move `F i j`, increment a_i and decrement a_{j+1}, and for the move `Q i`, calculate $b_i = a_1 + a_2 + \ldots + a_i$ modulo 2.
- **Complexity** Each of these operations takes $O(\log n)$ time, so the time complexity is $O(q \log n)$.

Data Structures II

Range Queries and Updates

Lowest Common Ancestor

Binary function composition

- **Implementation**

```
#include <iostream>
using namespace std;

int main() {
  int n, q;
  for (int i = 0; i < q; i++) {
    char type;
    cin >> type;
    if (type == 'F') {
      int i, j;
      cin >> i >> j;
      update(i, 1);
      update(j + 1, -1);
    }
    else if (type == 'Q') {
      int i;
      cin >> i;
      printf("%s\n",(query(1, i) % 2) ? "Face up" : "Face down");
    }
  }
}
```

Table of Contents

Lowest common ancestor

- **Problem statement** You are given a labelled rooted tree, T, and Q queries of the form, "What is the vertex furthest away from the root in the tree that is an ancestor of vertices labelled u and v?"
- **Input** A rooted tree T $(1 \leq |T| \leq 1,000,000)$, as well as Q $(1 \leq Q \leq 1,000,000)$ pairs of integers u and v.
- **Output** A single integer for each query, the label for the vertex that is furthest away from the root that is an ancestor of u and v

- **Algorithm 1** The most straightforward algorithm to solve this problem involves starting with pointers to the vertices u and v, and then moving them upwards towards the root until they're both at the same depth in the tree, and then moving them together until they reach the same place
- This is $O(n)$ per query, since it's possible we need to traverse the entire height of the tree, which is not bounded by anything useful

- The first step we can take is to try to make the "move towards root" step faster
- Since the tree doesn't change, we can pre-process the tree somehow so we can jump quickly

Table of Contents

Binary function composition

- Let's examine the parent relation parent[u] in the tree
- Our "move towards root" operation is really just repeated application of this parent relation
- The vertex two steps above u is parent[parent[u]], and three steps above is parent[parent[parent[u]]]

Binary function composition

- Immediately, we can precompute the values parent[u][k], which is parent[u] applied k times
- This doesn't have an easy straightforward application to our problem, nor is it fast enough for our purposes

Binary function composition

- If we only precompute parent[u][k] for each $k = 2^\ell$, we only need to perform $O(\log n)$ computations.
- Then, we can then compose up to $\log n$ of these precomputed values to obtain parent[u][k] for arbitrary k
- To see this, write out the binary expansion of k and keep greedily striking out the most significant set bit — there are at most $\log n$ of them.

- **Algorithm 2** Instead of walking up single edges, we use our precomputed parent[u][k] to keep greedily moving up by the largest power of 2 possible until we're at the desired vertex
- We need $O(n \log n)$ time and memory to preprocess the required compositions of our parent relation in the tree, as well as $O(\log n)$ time to handle each query

- **Implementation (preprocessing)**

```
// parent[u][k] is the 2^k-th parent of u
void preprocess() {
  for (int i = 0; i < n; i++) {
    // assume parent[i][0] (the parent of i) is already filled in
    for (int j = 1; (1<<j) < n; j++) {
      parent[i][j] = -1;
    }
  }

  // fill in the parent for each power of two up to n
  for (int j = 1; (1<<j) < n; j++) {
    for (int i = 0; i < n; i++) {
      if (parent[i][j-1] != -1) {
        // the 2^j-th parent is the 2^(j-1)-th parent of the 2^(j-1)-
            th parent
        parent[i][j] = parent[parent[i][j-1]][j-1];
      }
    }
  }
}
```

Lowest common ancestor

- **Implementation (querying)**

```
int lca (int u, int v) {
  // make sure u is deeper than v
  if (depth[u] < depth[v]) swap(u,v);

  // log[i] holds the largest k such that 2^k <= i
  for (int i = log[depth[u]]; i >= 0; i--) {
    // repeatedly raise u by the largest possible power of two until it is the same depth as v
    if (depth[u] - (1<<i) >= depth[v]) u = parent[u][i];
  }

  if (u == v) return u;

  for (i = log[depth[u]]; i >= 0; i--)
    if (parent[u][i] != -1 && parent[u][i] != parent[v][i]) {
      // raise u and v as much as possible without having them coincide
      // this is important because we're looking for the lowest common ancestor, not just any
      u = parent[u][i];
      v = parent[v][i];
    }

  // u and v are now distinct but have the same parent, and that parent is the LCA
  return parent[u][0];
}
```