Struktur Data Lanjutan

Tim Olimpiade Komputer Indonesia

Pendahuluan

Melalui dokumen ini, kalian akan:

- Memahami penggunaan C++ set/map
- Memahami konsep Fenwick Tree
- Memahami konsep Segment Tree
- Memahami konsep Sparse Table

C++ set

- Pada C++ STL (Standard Template Library), terdapat tipe data `std::set<T>` yang dapat digunakan untuk menyimpan sebuah himpunan (*set*) `T`.
  - Sebagai contoh, `set<int>` merupakan himpunan `int`.
- Untuk menggunakan tipe data ini, kita harus menambahkan `include <set>`.
- Pada umumnya, `set` akan menyimpan objek secara terurut menaik.
  - Perilaku ini dapat diubah menggunakan *custom comparator*. Sebagai contoh, pada C++11:

```
auto cmp = [](int a, int b) { ... };
set<int, decltype(cmp)> s(cmp);
```

- `set` diimplementasikan menggunakan *self balancing binary tree*, yang akan dibahas pada beberapa materi selanjutnya.

Operasi C++ set

- Beberapa fungsi/operasi umum yang sering digunakan pada C++ `set`.
  - `set::insert(x)` menambahkan elemen `x` pada `set` dalam waktu $O(\log N)$.
  - `set::erase(x)` menghapus elemen `x` pada `set` dalam waktu $O(\log N)$.
  - `set::size()` mengembalikan banyaknya elemen pada `set` dalam waktu $O(1)$.
  - `set::count(x)` mengembalikan banyaknya elemen x pada `set` dalam waktu $O(\log N)$ (antara 0 atau 1).
  - `set::clear()` menghapus seluruh elemen `set` dalam waktu $O(N)$.

  dengan N adalah banyaknya elemen pada `set`.

Set Iterator

- `set<int>::iterator` adalah tipe data penunjuk sebuah elemen pada `set`.
  - Sebagai contoh, operasi `set<int>::iterator it = s.begin();` akan membuat iterator `it` menunjuk pada elemen pertama pada himpunan `s` (jika `s` tidak kosong).
  - Kemudian operasi `it++` akan membuat iterator `it` menunjuk pada elemen kedua pada himpunan `s` (jika `s` berisi setidaknya dua bilangan).
  - Untuk mendapatkan elemen yang ditunjuk, kita dapat menggunakan `*it`.

Set Iterator (lanj.)

- Beberapa fungsi/operasi umum yang sering digunakan pada C++ `set` yang mengembalikan `set<int>::iterator` dalam waktu $O(\log N)$.
  - `set::lower_bound(x)` mengembalikan penunjuk elemen terkecil yang tidak lebih kecil dari `x` dalam waktu $O(\log N)$. Jika tidak ada elemen yang memenuhi, maka `set::lower_bound(x)` akan mengembalikan `set::end()`.
  - `set::upper_bound(x)` mengembalikan penunjuk elemen terkecil yang lebih besar dari `x` dalam waktu $O(\log N)$. Jika tidak ada elemen yang memenuhi, maka `set::upper_bound(x)` akan mengembalikan `set::end()`.
  - `set::find(x)` mengembalikan penunjuk elemen yang bernilai `x` dalam waktu $O(\log N)$. Jika tidak ada elemen yang memenuhi, maka `set::find(x)` akan mengembalikan `set::end()`.

Operasi C++ multiset

- C++ `std::multiset<T>` merupakan tipe data yang mirip dengan `set` namun dapat menyimpan beberapa elemen yang sama.
- Beberapa perbedaan antara C++ `multiset` dan `set`
  - `multiset::count(x)` dapat mengembalikan bilangan bulat lebih besar dari 1.
  - `multiset::erase(x)` menghapus seluruh elemen `x` pada `multiset`. Jika kita ingin menghapus hanya satu elemen `x`, gunakan `multiset::erase(multiset::find(x))`.

C++ map

- C++ `std::map<K, V>` merupakan tipe data yang digunakan untuk menyimpan pemetaan dari tipe data `K` ke tipe data `V`.
  - Sebagai contoh, `map<string, int>` merupakan pemetaan dari `string` ke `int`.
  - Contoh penggunaan tipe data ini adalah untuk menyimpan pemetaan dari nama murid ke nilai ujian, dan mengakses nilai ujian seorang murid dalam waktu cepat.
- Untuk menggunakan tipe data ini, kita harus menambahkan `include <map>`.
- `map` juga diimplementasikan menggunakan *self balancing binary tree*.

Operasi C++ map

- Misalkan kita memiliki `map<K, V> myMap;`.
- Beberapa fungsi/operasi umum yang sering digunakan pada C++ `map`.
  - `myMap[k]` mengakses nilai pemetaan `k` dalam waktu $O(\log N)$.
  - `myMap[k] = v` menentukan atau mengganti pemetaan dari `k` menjadi ke `v` dalam waktu $O(\log N)$.
  - `myMap.erase(k)` menghapus nilai pemetaan k dalam waktu $O(\log N)$.
  - `myMap.count(k)` mengembalikan 1 jika terdapat nilai pemetaan k, atau 0 jika tidak, dalam waktu $O(\log N)$.
  - `myMap.clear()` menghapus seluruh pemetaan dalam waktu $O(N)$.

  dengan N adalah banyaknya elemen pada `myMap`.

Persoalan Range Sum Query

- Persoalan **Range Sum Query** adalah persoalan menghitung jumlah elemen berurutan pada sebuah *array* A berukuran N.
  - Pada umumnya, diberikan Q pertanyaan yang merepresentasikan sebuah *subarray*.
- Jika *array* yang diberikan tidak dapat berubah, persoalan ini dapat diselesaikan menggunakan *prefix sum* yang menjawab satu pertanyaan dalam waktu $O(1)$.
- Jika *array* yang diberikan dapat berubah, persoalan ini dapat diselesaikan menggunakan *fenwick tree* yang menjawab satu pertanyaan dalam waktu $O(\log N)$ dan mengganti satu elemen *array* dalam waktu $O(\log N)$.

Fenwick Tree

- **Fenwick Tree** (sering disebut juga **Binary Indexed Tree**) adalah data struktur yang menyimpan sebuah *array* berukuran N dengan indeks 1 sampai N dengan setiap elemennya menyimpan jumlah elemen berurutan pada *array* A.
  - $BIT[j]$ menyimpan jumlah elemen $\sum_{i=j-LSBIT(j)+1}^{j} A[i]$, dengan $LSBIT(j)$ adalah nilai dari `j & (-j)` pada C++. Sebagai contoh,
    - $LSBIT(1) = 1, BIT[1] = A[1]$,
    - $LSBIT(2) = 2, BIT[2] = A[1] + A[2]$,
    - $LSBIT(3) = 1, BIT[3] = A[3]$,
    - $LSBIT(4) = 4, BIT[4] = A[1] + A[2] + A[3] + A[4]$, dan
    - $LSBIT(6) = 2, BIT[6] = A[5] + A[6]$.

Fenwick Tree (lanj.)

- Dengan *fenwick tree*, kita dapat menghitung nilai $\sum_{i=1}^{x} A[i]$ dalam $O(\log N)$.
  - Sebagai contoh, menghitung nilai $A[1] + A[2] + A[3] + A[4] + A[5] + A[6]$ dapat disederhanakan menjadi $(A[1] + A[2] + A[3] + A[4]) + (A[5] + A[6]) = BIT[4] + BIT[6]$.
- Secara umum, $\sum_{i=1}^{x}$ dapat dihitung menggunakan rumus berikut:
  - 0, jika $x = 0$,
  - $BIT[x] + \sum_{i=1}^{x-LSBIT(x)}$, jika $x > 0$.

Fenwick Tree (lanj.)

- Jika nilai $A[i]$ berubah, maka kita harus memperbaharui semua nilai $BIT[j]$ yang memenuhi $j - LSBIT(j) < i \leq j$. Terdapat $O(\log N)$ nilai yang harus diperbaharui.
  - Sebagai contoh, jika nilai $A[5]$ berubah, maka kita harus memperbaharui nilai $BIT[5], BIT[6], BIT[8], BIT[16], \cdots$.
- Secara umum, jika nilai $A[i]$ berubah menjadi $A[i] + \delta$, kita dapat memanggil fungsi `update(i)` yang melakukan hal berikut jika $i \leq N$:
  - Perbaharui nilai $BIT[i]$ menjadi $BIT[i] + \delta$.
  - Panggil fungsi `update(i + LSBIT(i))`.

Segment Tree

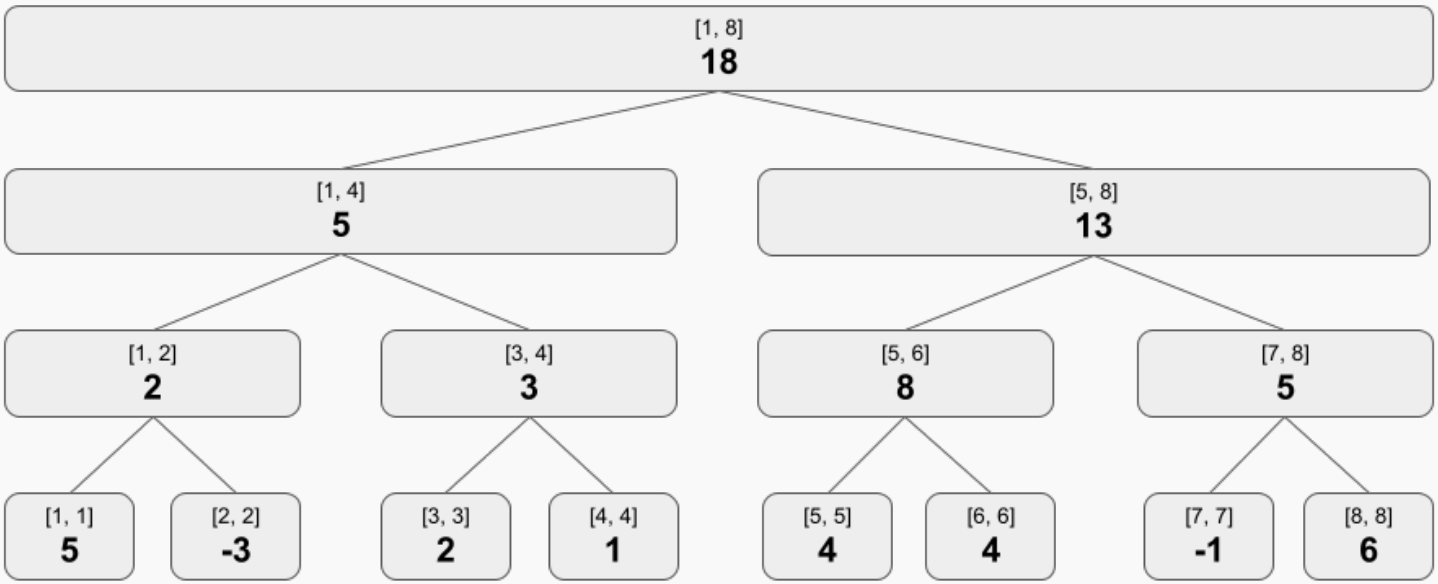
- **Segment Tree** merupakan struktur data alternatif untuk menyelesaikan persoalan *range sum query*.
- *Segment tree* merupakan *binary tree*. Tiap *node* memiliki informasi nilai dari suatu rentang atau segmen. Ukuran segmen dari suatu *node* pada *segment tree* merupakan gabungan segmen dari anak-anaknya.

Ilustrasi Segment Tree

- Misalkan kita memiliki *array* berukuran N. *Segment tree* dari *array* tersebut memiliki $O(\log N)$ tingkat.
- Secara keseluruhan, terdapat maksimal $2N$ segmen. Kompleksitas memori dari *segment tree* adalah $O(N)$.
- Untuk memudahkan implementasi, kita nomori setiap *node* pada *segment tree*. *Root* dari *tree* ini dinomori 1, dengan anak kiri dan kanan dari *node* x dinomori $2x$ dan $2x + 1$ secara berurutan.

Ilustrasi Segment Tree (lanj.)

- Sebagai contoh, berikut merupakan bentuk *segment tree* untuk menghitung *range sum query* dari *array* $[5, -3, 2, 1, 4, 4, -1, 6]$.
- Tiap *node* memiliki informasi jumlah dari segmen yang dilingkupinya.



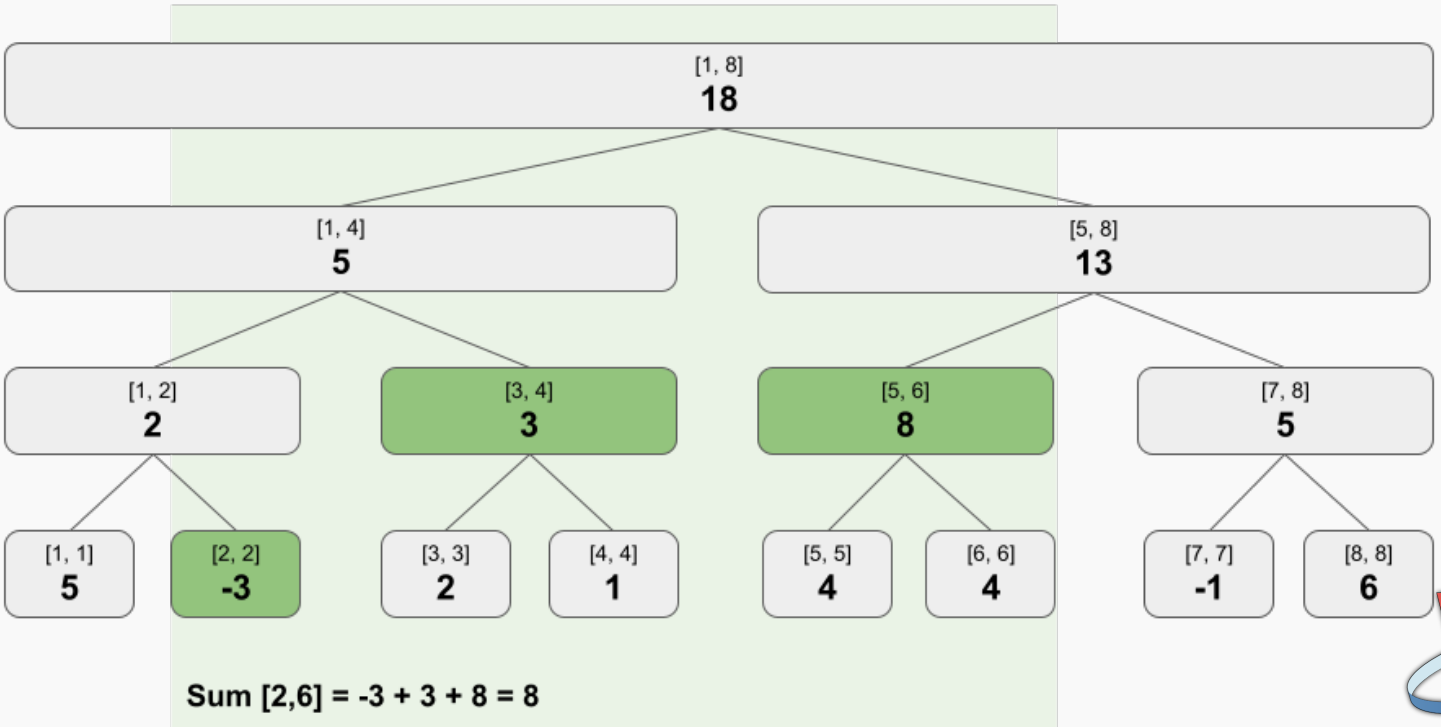
Inisiasi Segment Tree

- Berikut adalah contoh kode untuk menginisiasi *segment tree* untuk menghitung *range sum query*.
- *MAXA* dihitung dengan $2^{\lceil \log_2 N \rceil + 1}$.

```
int st[MAXA];           // informasi jumlah segmen

void build(int idx, int l, int r) {
  if (l == r) {
    st[idx] = val[l];
    return;
  }

  int mid = (l + r) / 2;
  build(idx * 2    , l, mid); // rekursif ke kiri
  build(idx * 2 + 1, mid + 1, r); // rekursif ke kanan

  st[idx] = st[idx * 2] + st[idx * 2 + 1];
}

...
bulid(1, 1, N); // pemanggilan awal
```

Query pada Segment Tree

- Untuk mendapatkan jumlah dari interval L sampai R, kita cukup menjumlahkan segmen-segmen yang membentuk interval L sampai R. Terdapat paling banyak $O(\log N)$ segmen yang harus dijumlahkan.
- Berikut adalah ilustrasi untuk menjumlahkan elemen dari indeks 2 sampai 6.



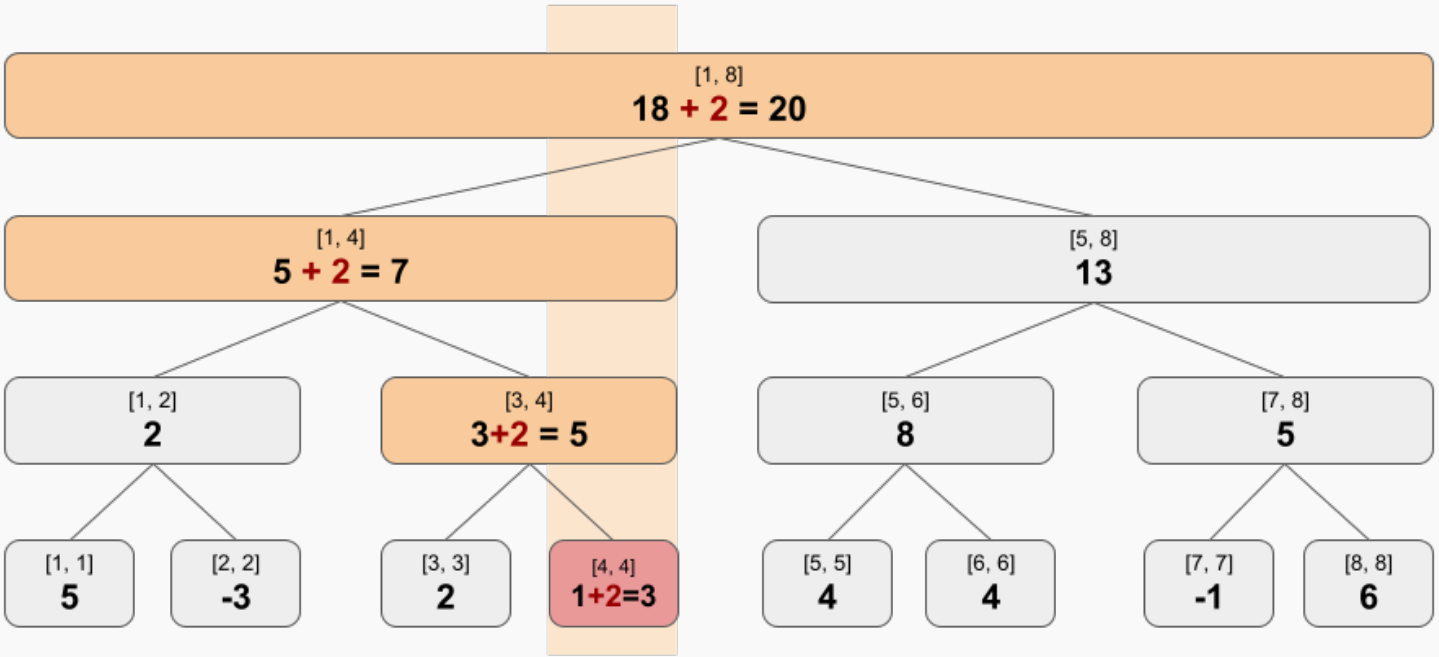
Kode Query pada Segment Tree

- *Query* dapat dicapai dengan melakukan penjelajahan *segment tree* dari segmen terbesar, dan berhenti ketika kita berada di segmen yang sepenuhnya di dalam segmen yang ingin dijumlahkan.

```
int query(int idx, int l, int r, int x, int y) {
  if (x > r || y < l)
    // node ini berada di luar segmen query
    return 0;

  if (x <= l && r <= y)
    // node ini berada di dalam segmen query
    return st[idx];

  int mid = (l + r) / 2;
  return query(idx * 2    , l, mid, x, y) +
         query(idx * 2 + 1, mid + 1, r, x, y);
}
```

Single Point Update pada Segment Tree

- Untuk memperbaharui nilai elemen di suatu indeks, kita cukup memperbaharui nilai seluruh segmen yang melingkupi indeks tersebut.
- Kompleksitas untuk tiap perbaharuan adalah $O(\log N)$.
- Berikut adalah ilustrasi menambahkan nilai 2 pada indeks 4.



Kode Single Point Update

- Berikut adalah contoh kode jika kita ingin menambahkan nilai v pada suatu indeks x.

```
void update(int idx, int l, int r, int x, int v) {
  // perbaharui jumlah nilai pada segmen ini
  st[idx] += v;

  if (l == r)
    // node tanpa anak
    return;

  // kunjungi anak yang benar
  int mid = (l + r) / 2;
  if (x <= mid)
    update(idx * 2    , l, mid, x, v);
  else
    update(idx * 2 + 1, mid + 1, r, x, v);
}
```

Range Update pada Segment Tree

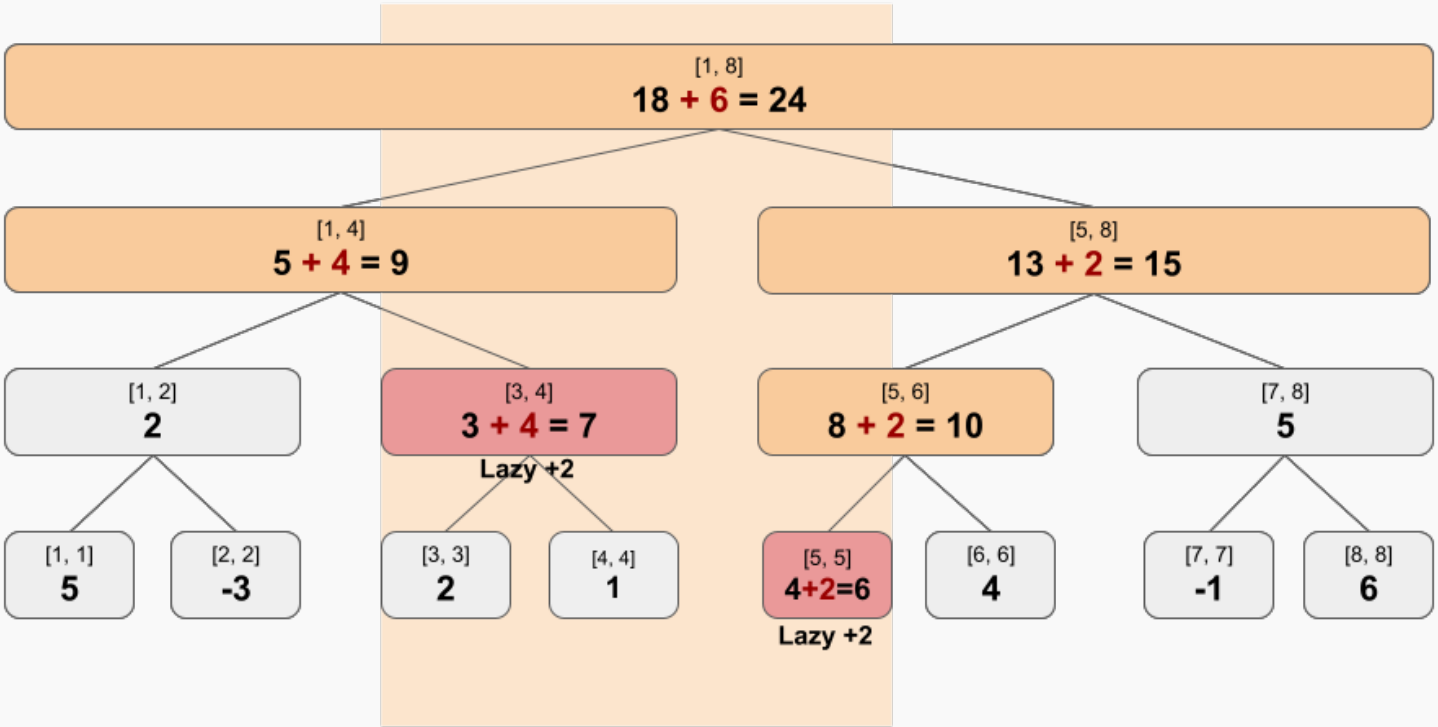
- Bagaimana jika kita harus memperbaharui beberapa elemen sekaligus dalam suatu segmen?
- Misalnya, untuk suatu segmen $[A, B]$, tambahkan nilainya dengan v.
- Kita bisa melakukan penjelajahan pada *segment tree*, dan berhenti di daun yang termasuk dari segmen.
- Namun, kasus terburuknya adalah jika kita harus memperbaharui keseluruhan range, sehingga kita harus memperbaharui keseluruhan $O(N)$ *node* pada *segment tree*.

Lazy Propagation

- Agar dapat memperbaharui segmen dengan lebih efisien, kita cukup berhenti pada segmen yang sudah sepenuhnya di dalam segmen yang ingin diperbaharui.
- Karena kita berhenti di segmen X, kita sebenarnya belum memperbaharui informasi pada segmen-segmen yang merupakan anak dari X.
- Karena itu, kita perlu menyiapkan informasi tambahan yang menyatakan perbaharuan yang belum diselesaikan. Informasi ini akan dipropagasi nanti ketika kita melakukan penjelajahan di segmen tersebut.
- Teknik ini disebut dengan **Lazy Propagation**.

Ilustrasi Lazy Propagation

- Berikut adalah ilustrasi dari *lazy propagation* untuk menambahkan nilai 2 ke indeks 3 sampai 5.



Kode Lazy Propagation

- Berikut adalah kode untuk melakukan *lazy propagation*.

```
void update(int idx, int l, int r, int x, int y, int v) {
  if (x < r || l > y) return; // berhenti jika di luar
                              // segmen
  if (x <= l && r <= y) {
    // perbaharui jumlah segmen ini
    st[idx] += (r - l + 1) * v;
    // perbaharui nilai "lazy" segmen ini
    lazy[idx] += v;
    return;
  }

  int mid = (l + r) / 2;

  propagate(idx, l, r);
  update(idx * 2    , l, mid, x, y, v);
  update(idx * 2 + 1, mid + 1, r, x, y, v);

  st[idx] = st[idx * 2] + st[idx * 2 + 1];
}
```

Kode Propagasi

- Fungsi tambahan `propagate` adalah untuk meneruskan perbaharuan yang belum dieksekusi ke anak segmen.

```
void propagate(int idx, int l, int r) {
  // hanya perlu dilakukan jika nilai "lazy" tidak 0
  if (lazy[idx]) {
    int mid = (l + r)/2, lc = idx * 2, rc = idx * 2 + 1;
    // propagasi ke anak kiri
    lazy[lc] += lazy[idx];
    st[lc] += (mid - l + 1) * lazy[idx];
    // propagasi ke anak kanan
    lazy[rc] += lazy[idx];
    st[rc] += (r - mid) * lazy[idx];

    lazy[idx] = 0; // bersihkan nilai "lazy"
  }
}
```

Query dengan Lazy Propagation

- *Query* pada *segment tree* yang menggunakan *lazy propagation* persis pada *query* biasa, hanya saja cukup ditambahkan fungsi propagasi sebagai berikut.

```
int query(int idx, int l, int r, int x, int y) {
  if (x > r || y < l)
    // node ini berada di luar segmen query
    return 0;

  if (x <= l && r <= y)
    // node ini berada di dalam segmen query
    return st[idx];

  propagate(idx, l, r);

  int mid = (l + r) / 2;
  return query(idx * 2    , l, mid, x, y) +
         query(idx * 2 + 1, mid + 1, r, x, y);
}
```

Sparse Table

- **Sparse Table** merupakan struktur data alternatif untuk menyelesaikan persoalan *range sum query*
- Seluruh bilangan bulat Z dapat direpresentasikan dalam bentuk $\sum_{i=1} 2^{z_i}$ dengan $z_i > z_{i+1}$. Sebagai contoh, $11 = 2^3 + 2^1 + 2^0$.
- Sehingga, seluruh segmen $[L, R)$ dapat dipartisi menjadi paling banyak $O(\log(R - L))$ segmen $[L, L + 2^{z_1}), [L + 2^{z_1}, L + 2^{z_1} + 2^{z_2}), \cdots$. Sebagai contoh, segmen $[2, 13)$ dapat dipartisi menjadi segmen-segmen $[2, 10), [10, 12), [12, 13)$.

Sparse Table (lanj.)

- Pada sebuah *array* A berukuran N, *Sparse table* menyimpan jumlah elemen pada segmen $[i, i + 2^j)$ untuk setiap $0 \leq j \leq \lfloor \log_2 N \rfloor, 1 \leq i < N - 2^j$ ke dalam $S[i][j]$.
- Mengisi seluruh nilai $S[i][j]$ dapat dilakukan dalam waktu $O(N \log N)$ dengan mengiterasikan nilai j dari 0 sampai $\lfloor \log_2 N \rfloor$:
  - Jika $j = 0$, maka $S[i][j] = A[i]$.
  - Jika $j > 0$, maka $S[i][j] = S[i][j-1] + S[i + 2^{j-1}][j-1]$.
- Perhatikan bahwa jika suatu elemen di *array* A diperbaharui, maka terdapat $O(N)$ elemen pada S yang harus diperbaharui, sehingga struktur data ini biasanya tidak digunakan jika elemen pada *array* dapat diperbaharui.

Range Sum Query

- Berikut adalah kode untuk menghitung jumlah elemen indeks $[L, R)$.
- Perhatikan bahwa `1 << j` pada C++ menghitung nilai 2^j. Hal ini akan dibahas lebih lanjut pada bab berikutnya.

```
int query(int l, int r) {
  int res = 0;
  for (int j = floor(log(N)); j >= 0; --j) {
    if (l + (1 << j) <= r) {
      res += S[l][j];
      l += (1 << j);
    }
  }
  return res;
}
```