Competitive Training Camp

Lecture 9

Christian Yongwhan Lim

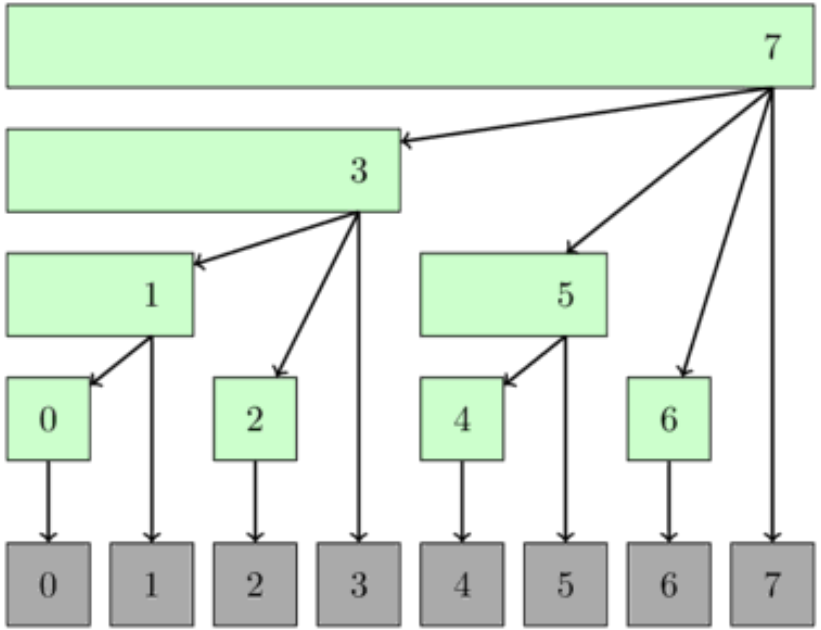
Wednesday, July 23, 2025

Overview: Data Structures

- Binary Indexed Tree
- Segment Tree
- Sparse
- Treap

Binary Indexed Tree / Fenwick Tree

- `sum(l, r)`
- `add(idx, delta)`



Binary Indexed Tree / Fenwick Tree

```
struct FenwickTree {
  vector<int> bit;
  int n;
  FenwickTree(int n) {
    this->n = n;
    bit.assign(n, 0);
  }
  int sum(int r);
  int sum(int l, int r);
  void add(int idx, int delta);
};
```

Binary Indexed Tree / Fenwick Tree

```
int sum(int r) {
  int ret = 0;
  for (; r >= 0; r = (r & (r + 1)) - 1)
    ret += bit[r];
  return ret;
}

int sum(int l, int r) {
  return sum(r) - sum(l - 1);
}
```

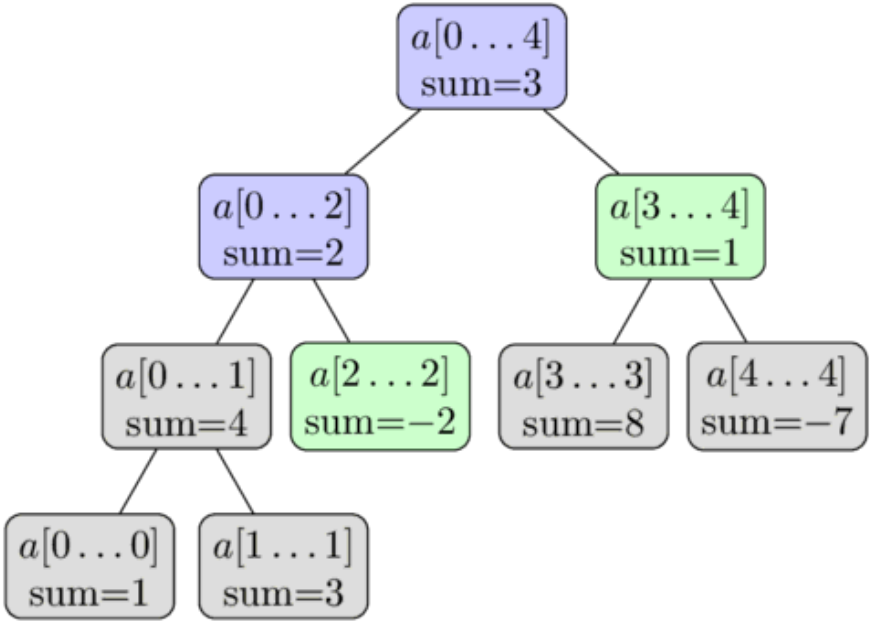
Binary Indexed Tree / Fenwick Tree

```
void add(int idx, int delta) {
  for (; idx < n; idx = idx | (idx + 1))
    bit[idx] += delta;
}
```

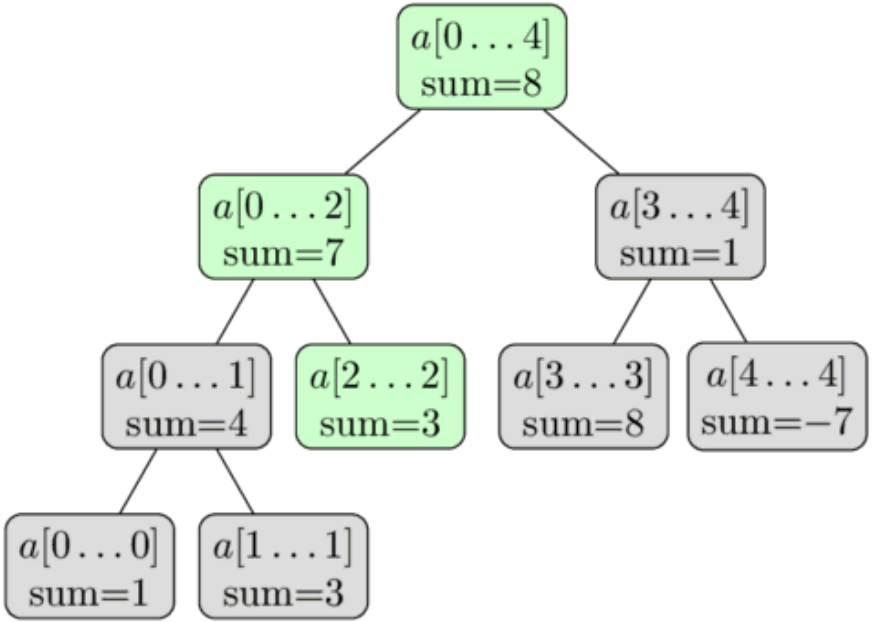
Segment Tree: Point Update & Range Query

- `update(i, x)`
- `sum(l, r)`

Segment Tree: Example: sum(2,4)



Segment Tree: Example: update(2,3)



Segment Tree: Implementation

```
int n, t[4*MAXN];
void build(int a[], int v, int tl, int tr) {
  if (tl == tr) {
    t[v] = a[tl];
  } else {
    int tm = (tl + tr) / 2;
    build(a, v*2, tl, tm);
    build(a, v*2+1, tm+1, tr);
    t[v] = t[v*2] + t[v*2+1];
  }
}
```

Segment Tree: Implementation

```
int sum(int v, int tl, int tr, int l, int r) {
  if (l > r)
    return 0;
  if (l == tl && r == tr) {
    return t[v];
  }
  int tm = (tl + tr) / 2;
  return sum(v*2, tl, tm, l, min(r, tm))
         + sum(v*2+1, tm+1, tr, max(l, tm+1), r);
}
```

Segment Tree: Implementation

```
void update(int v, int tl, int tr,
            int pos, int new_val) {
  if (tl == tr) t[v] = new_val;
  else {
    int tm = (tl + tr) / 2;
    if (pos <= tm) update(v*2, tl, tm, pos, new_val);
    else update(v*2+1, tm+1, tr, pos, new_val);
    t[v] = t[v*2] + t[v*2+1];
  }
}
```

Lazy Segment Tree: Range Update & Range Query

- `update(l, r, add)`
- `max(l, r)`

- **update, only when you need to!**

Lazy Segment Tree: Implementation

```
void push(int v) {
  t[v*2] += lazy[v];
  lazy[v*2] += lazy[v];
  t[v*2+1] += lazy[v];
  lazy[v*2+1] += lazy[v];
  lazy[v] = 0;
}
```

Lazy Segment Tree: Implementation

```
void update(int v, int tl, int tr,
            int l, int r, int add) {
  if (l > r) return;
  if (l == tl && tr == r) t[v] += add, lazy[v] += add;
  else {
    push(v);
    int tm = (tl + tr) / 2;
    update(v*2, tl, tm, l, min(r, tm), add);
    update(v*2+1, tm+1, tr, max(l, tm+1), r, add);
    t[v] = max(t[v*2], t[v*2+1]);
  }
}
```

Lazy Segment Tree: Implementation

```
int query(int v, int tl, int tr, int l, int r) {
  if (l > r)
    return -INF;
  if (l == tl && tr == r)
    return t[v];
  push(v);
  int tm = (tl + tr) / 2;
  return max(query(v*2, tl, tm, l, min(r, tm)),
             query(v*2+1, tm+1, tr, max(l, tm+1), r));
}
```

Persistent Segment Tree: Save History

- `update(l, r, new_val, k)`
- `sum(l, r, k)`

- **Use vertex struct!**

Sparse Table

- `st[i][j]` stores answer for the range `[j, j+2^i-1]` of length 2^i.

- most range queries can be answered in **O(log n)**
- **range minimum query** can be in **O(1)**(!).

- the caveat is the data must be **immutable**!

Sparse Table: Range Sum Queries

```
long long st[K + 1][MAXN];

std::copy(array.begin(), array.end(), st[0]);

for (int i = 1; i <= K; i++)
  for (int j = 0; j + (1 << i) <= N; j++)
    st[i][j] = st[i - 1][j]
               + st[i - 1][j + (1 << (i - 1))];
```

Sparse Table: Range Sum Queries: sum(L,R)

```
long long sum = 0;
for (int i = K; i >= 0; i--) {
  if ((1 << i) <= R - L + 1) {
    sum += st[i][L];
    L += 1 << i;
  }
}
```

Sparse Table: Range Minimum Queries

- The range minimum of [1, 6] is clearly the same as the minimum of the range minimum of [1, 4] and the range minimum of [3, 6].

- Generally, we can compute the minimum of the range `[L, R]` with:

$$\texttt{min(st[i][L], st[i][R-2}^{\texttt{i}}\texttt{+1])} \text{ where } i=\log_2(R-L+1)$$

Sparse Table: Range Minimum Queries: Pre-computation

```
int lg[MAXN+1];
lg[1] = 0;
for (int i = 2; i <= MAXN; i++)
  lg[i] = lg[i/2] + 1;
```

Sparse Table: Range Minimum Queries: Implementation

```
int st[K + 1][MAXN];
std::copy(array.begin(), array.end(), st[0]);
for (int i = 1; i <= K; i++)
  for (int j = 0; j + (1 << i) <= N; j++)
    st[i][j] = min(st[i - 1][j],
                   st[i - 1][j + (1 << (i - 1))]);
```

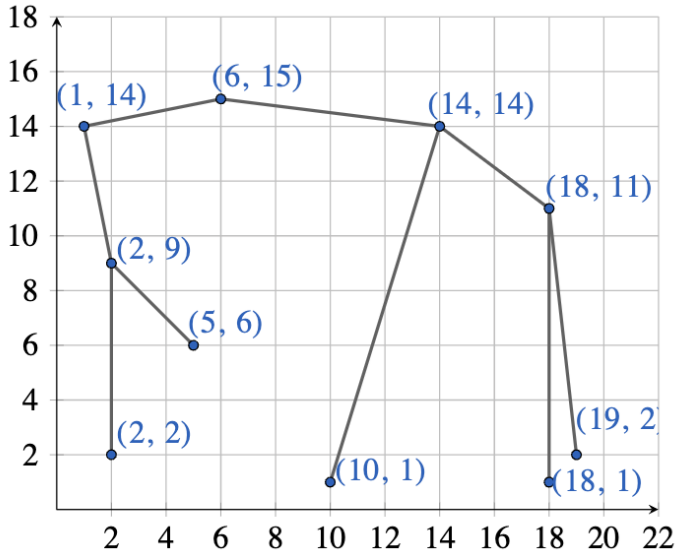
Sparse Table: Range Minimum Queries: min(L,R)

```
int i = lg[R - L + 1];
int minimum = min(st[i][L], st[i][R - (1 << i) + 1]);
```

Treap (Cartesian Tree)

- Combines **binary tree** and **binary heap**
- Hence, the name: **tree** + **heap** => **treap**

- Stores pairs (X, Y) in a binary tree in such a way that it is a binary search tree by X and a binary heap by Y.



Treap

- Split(T,X)
- Merge(T_1,T_2)
- Insert(X,Y)
- Search(X)
- Erase(X)
- Build(X_1,..., X_N)
- Union(T_1,T_2)
- Intersect(T_1,T_2)