Introduction to Algorithms

Science Honors Program (SHP)

Session 2

Christian Lim

Saturday, February 24, 2024

Slide deck in github

- You may find the slide deck from:
  - https://github.com/yongwhan/yongwhan.github.io/blob/master/columbia/shp/Spring%202024%20Introduction%20to%20Algorithms_%20Session%201.pdf
- You may get to the link by:
  - https://github.com/yongwhan/
  - => yongwhan.github.io
  - => columbia
  - => shp
  - => session 2 slide

Overview

- **Built-in data structures** (vector; stack; queue; priority_queue; set; map)
- **Break #1 (5-minute)**

- **Custom data structures** (disjoint set union; Fenwick/Segment tree; ordered set)
- **Break #2 (5-minute)**

- **Some programming exercises**

Attendance

- Let's take a quick attendance before we begin!

CodeForces Columbia SHP Algorithms Group

- While I take the attendance, please join the following group: **https://codeforces.com/group/lfDmo9iEr5**

- We will be using them in the last portion of the session today!

Built-in Data Structures

- vector
- queue
- stack
- deque
- priority_queue
- set
- map

vector

- **Array, but resizable (Dynamic Array)**

- [], push_back, pop_back, clear, size, empty

practice with vector

queue

- **FIFO (First In First Out)**

- push, front, pop, clear, size, empty

- Typically used for Breadth First Search (BFS)
  - we will see them later!

practice with queue

stack

- **Last In First Out (LIFO)**

- push, top, pop, clear, size, empty

- Typically, used for Depth First Search (DFS)
  - we will see them later!

practice with stack

deque

- **Double-ended**

- [], push_back, push_front, pop_back, pop_front, clear, size, empty

- Typically used for 0/1 Breadth First Search (0/1 BFS)
  - we will see them later!

practice with deque

priority_queue

- **queue with priority**

- push, top, pop, clear, size, empty

- Typically used for Dijkstra
  - we will see them later!

practice with priority_queue

Graph Traversal Summary

Algorithm	**Container**
Breadth First Search (BFS)	queue
Depth First Search (DFS)	stack
0-1 BFS	deque
Dijkstra	priority_queue

set

- store unique elements following a specific order.

- insert, find, clear, erase, size

- Typically used for de-duplicating elements.
- Typically used for find elements quickly (logarithmic).

practice with set

map

- {key, value} pairs, following a specific order.
- [], find, clear, erase, size
- Typically used for counting **frequency** of each elements.

practice with map


BREAK #1

Custom Data Structures

- disjoint set union
- Fenwick/Segment tree
- ordered set

Disjoint Set Union: Intuition

- `make_set(v)`
- `find_set(v)`
- `union_sets(a, b)`

Disjoint Set Union: Intuition

- path compression
- union by size/rank

Disjoint Set Union: Implementation

```
void make_set(int v) {
  parent[v] = v;
  size[v] = 1;
}

int find_set(int v) {
  if (v == parent[v])
    return v;
  return find_set(parent[v]);
}
```

Disjoint Set Union: Implementation

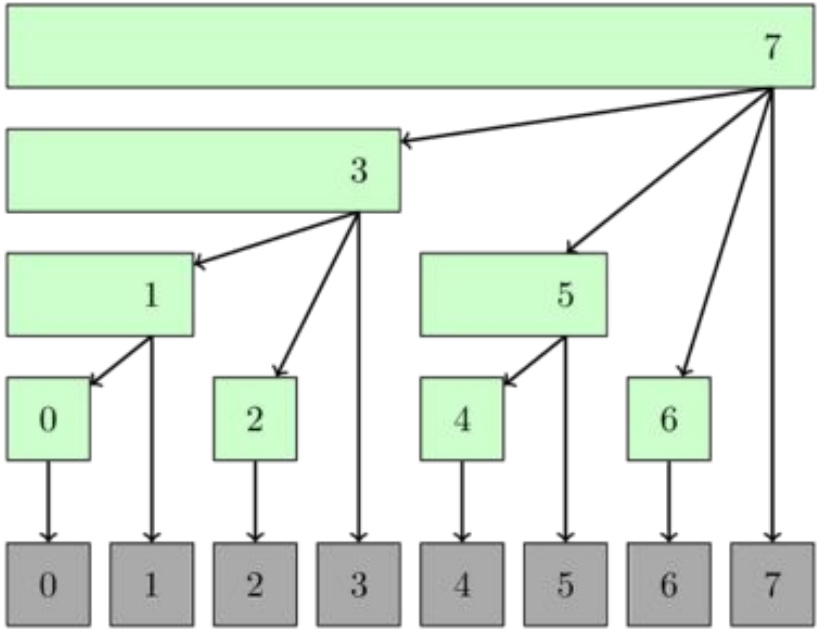
```
void union_sets(int a, int b) {
  a = find_set(a);
  b = find_set(b);
  if (a != b) {
    if (size[a] < size[b])
      swap(a, b);
    parent[b] = a;
    size[a] += size[b];
  }
}
```

Disjoint Set Union: Example

- Minimum Spanning Tree: **Kruskal's Algorithm**
  - we will see them later!

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);
}
```

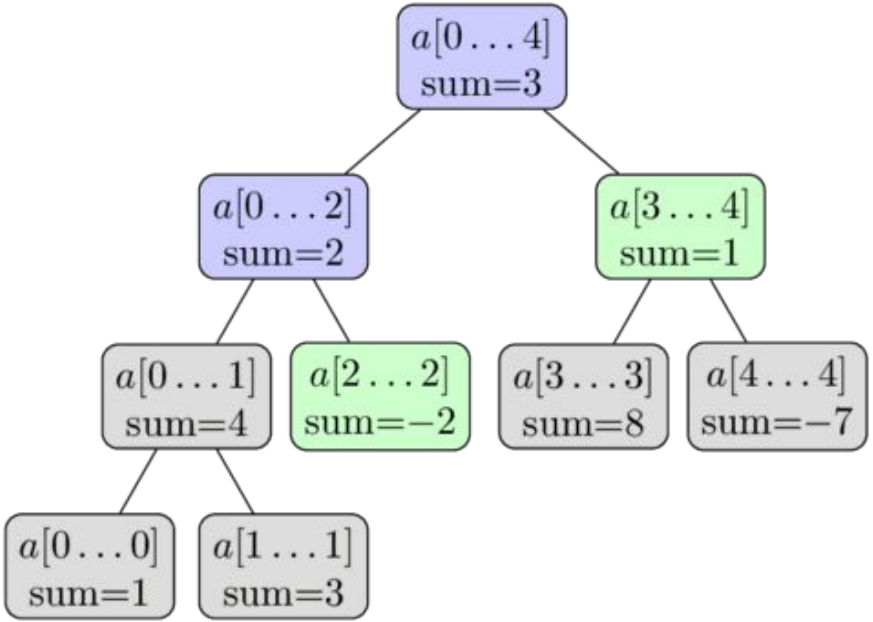
Binary Indexed Tree / Fenwick Tree

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

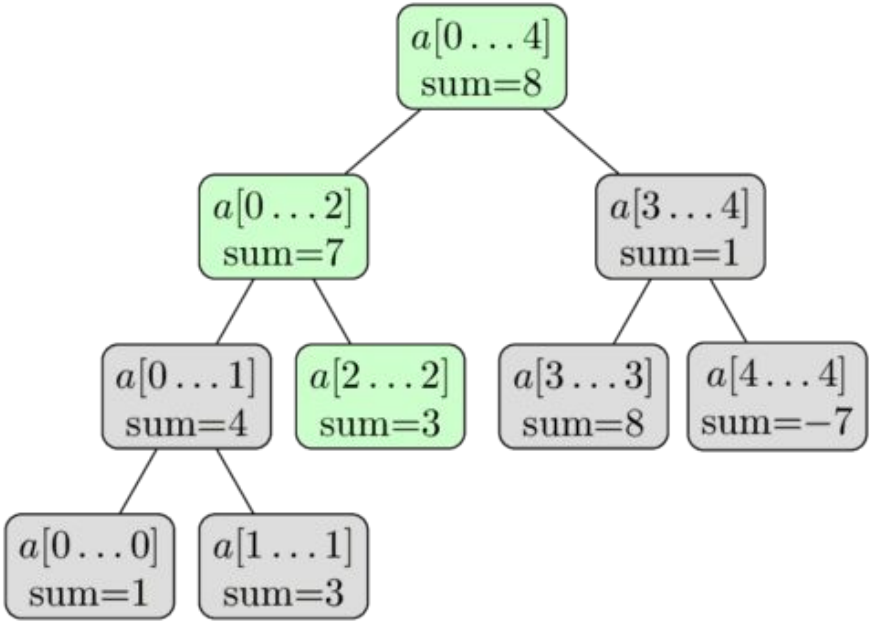
Segment Tree: Point Update & Range Query

- `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!**

Ordered Set

- A policy based data structure in g++ that keeps the unique elements in sorted order.
- It performs all the operations as performed by the set data structure in STL in **`log(n)`** complexity.

- In addition, it performs two additional operations also in logarithmic complexity.
  - **`order_of_key(k)`**: Number of items strictly smaller than k.
  - **`find_by_order(k)`**: k^{th} element in a set (counting from zero).

Ordered Set: Implementation (header)

```
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;
template <class T>
using Tree = tree<T, null_type, less<T>, rb_tree_tag,
tree_order_statistics_node_update>;
// find_by_order
// order_of_key
```


BREAK #2

Again, CodeForces Columbia SHP Algorithms Group

- Please join the following group: **https://codeforces.com/group/lfDmo9iEr5**

Practice Problems

- **Binary Indexed/Fenwick Tree**
  - https://codeforces.com/contest/597/problem/C
  - https://codeforces.com/problemset/problem/704/A
  - https://codeforces.com/problemset/problem/296/C
  - https://codeforces.com/contest/276/problem/C
  - https://codeforces.com/contest/1354/problem/D

Practice Problems

- **Segment Tree**
  - https://codeforces.com/problemset/problem/920/F
  - https://codeforces.com/problemset/problem/242/E
  - https://codeforces.com/contest/474/problem/F
  - https://codeforces.com/problemset/problem/52/C
  - https://codeforces.com/contest/438/problem/D

Practice Problems

- **Ordered Set**
  - https://codeforces.com/contest/1676/problem/H2
  - https://codeforces.com/contest/1915/problem/F
  - https://codeforces.com/contest/652/problem/D

Next Week!

- Next week, we will cover some algorithm paradigms!
  - **Complete Search**
  - **Divide and Conquer**

Slide Deck

- You may **always** find the slide decks from:
  - https://github.com/yongwhan/yongwhan.github.io/blob/master/columbia/shp


THANK YOU