Data Structure(III)

Ian

Things that we would talk about

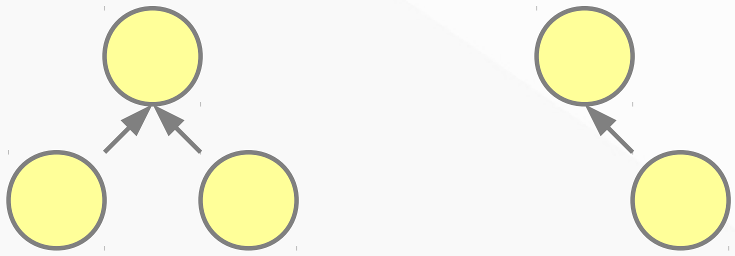
- **Disjoint set**
- **Segment tree**
- **Binary indexed tree**
- **Trie**

Disjoint set

- Keep tracking elements belong to which subset (non-overlapping)
- Support two operation
  - Union (merge two subset into one)
  - Find (see if element x belong to which subset)

Disjoint set

- We use a forest to represent this
- Each node is a element
- Two element is in same subset if they are in the same tree. Which means the root for two element is the same.
- Merging two subset equals to merging to tree into one.
- Below we have two subset.



Implementing Disjoint Set

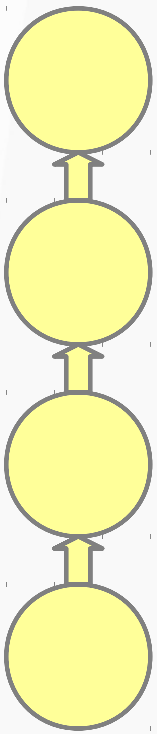
- We define parent(x) as parent of x
- Int find(int x)

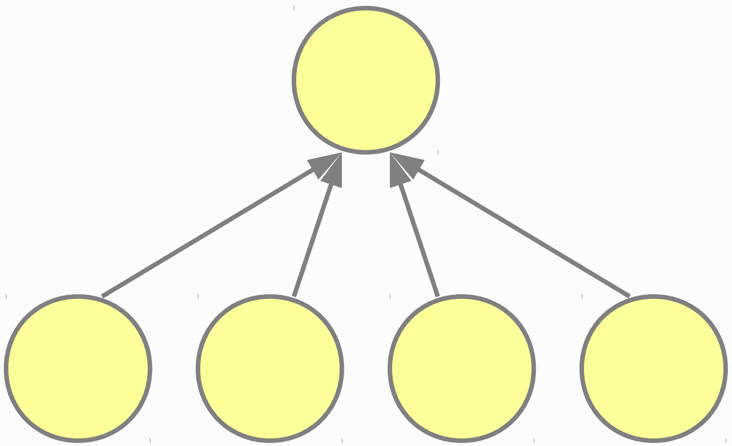
  while (x is not root) x = parent(x)

  return x
- Void merge(int x, int y)

  set parent of x to y
- If x is in the same subset with y, find(x) = find(y)
- Otherwise find(x) <> find(y)
- Time complexity for find is O(N)
- Time complexity for merge is O(1)
- We can make it faster

Path Compression





Path Compression

- Each time we find the root of x, we change the parent all of its ancestor including x to root.
- Int find(int x)

```
if x is root
    return x
int root = find(parent(x))
set root as parent of x
return root
```

- This make find operation O(N log N)

Union by Rank

- Idea if simple. We should avoid making tree tall. So each find operation use less time.
- Define height of the tree as the max of distance of root to its leaves.
- When we union tree S and tree T.
  - If height(S) > height(T)

    set parent of root(T) to root(S)
  - If height(S) < height(T)

    set parent of root(S) to root(T)
- Time complexity O(N log N)

Extra : Tricks About Heuristic Merging

- **Union by rank is a kind of heuristic merging. We merge smaller(shorter thing) into larger thing.**
- **As height of tree would only be increased by 1 when two tree have the same height. Meaning height at most increase log N times. So height of the resultant tree is log N.**

Ackermann

- **A trick which is called Heuristic Merging or merging smaller into larger is basically using the same idea.**
- **We iterate through smaller set and put it in larger set.**
- **Such implementation would always have O(N log N).**
- **Example : NP1612, APIO121**

Disjoint Set

- If we apply union by rank and path compression together, we would get $O(\alpha(N))$ for each operation. Where α is inverse Ackermann function.
- It is very small for large N. So we basically can treat it as constant
- But for most of the times using only path compression is sufficient.

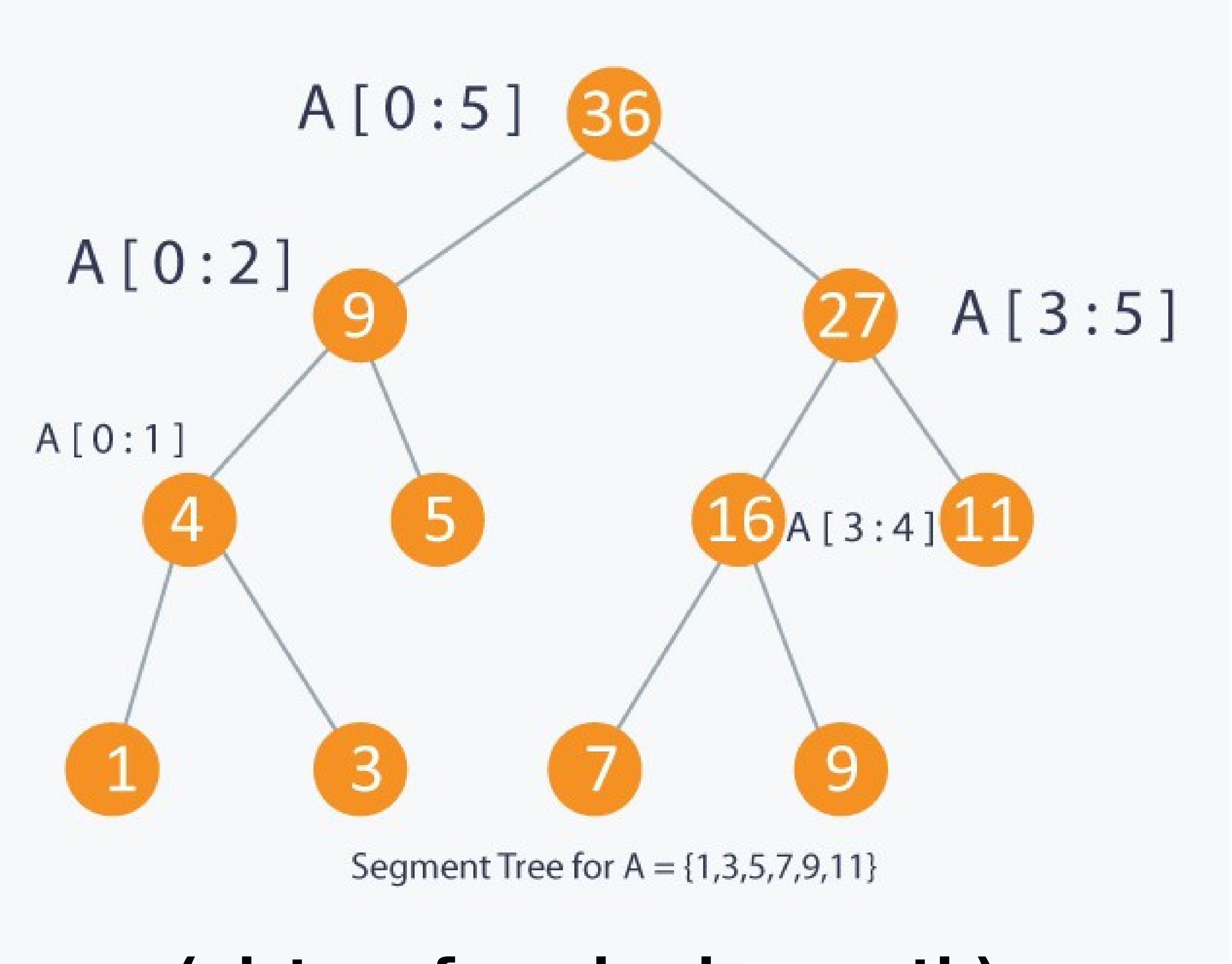
Practice Problems for DSU

- **HKOJ N1511**
- **CF 766D**

Segment Tree

- Segment tree is a data structure that support range query and update
- It is a binary tree
- Each node represent a segment.
- Assume node v maintain data of [l, r]
  - Left child maintain [l, mid]
  - Right child maintain [mid + 1, r]
  - Where mid = (l + r) / 2
- As each time interval is divided by 2, height of tree is log N

Segment Tree



(picture from hacker earth)

Classical Problem for Segment Tree

- Range minimum(maximum) query
  - Given an integer A
  - Query(l, r)
    - Ask for minimum element in [l; r]
  - Update(id, val)
    - Update element in position id to val
- EX : M0921

Classical Problem for Segment Tree

- **Maximum subarray problem**
  - **Given an integer array A**
  - **Query(l, r)**
    - **Find maximum sum of subarray [a; b] such that l <= a <= b <= r**
  - **update(id, val)**
    - **Update element in position id to val**
- **EX : M0923**

Classical Problem for Segment Tree

- **Sweep Line**
- **Given N rectangle, find union area**
- **http://codeforces.com/blog/entry/20377**
- **EX : M1633**

Implementation

- update(id, x, y, pos, val) // update value in pos to val

```
node id maintain [x; y]
If x == y
    a[pos] = val
    return
mid =  (l + r) / 2
If (pos <= mid) update(id * 2, x, mid, pos, val)
else update(id * 2 + 1, mid + 1, y, pos, val)
a[id] = max(a[id * 2], a[id * 2 + 1])
```

Implementation

- **query(int id, int x, int y, int l, int r) // find ans in [l; r]**

```
if (out of range)
    return -1
if (l <= x and y <= r)
    return a[id]
mid = (l + r) / 2
return
max(query(id * 2, x, mid, l, r), query(id * 2 + 1, mid + 1, y, l, r)
```

Practice Problems for Segment Tree

- CF 339D
- CF 380C

Findind LCA using Segment Tree

- https://www.topcoder.com/community/data-science/data-science-tutorials/range-minimum-query-and-lowest-common-ancestor/
- http://codeforces.com/blog/entry/16221

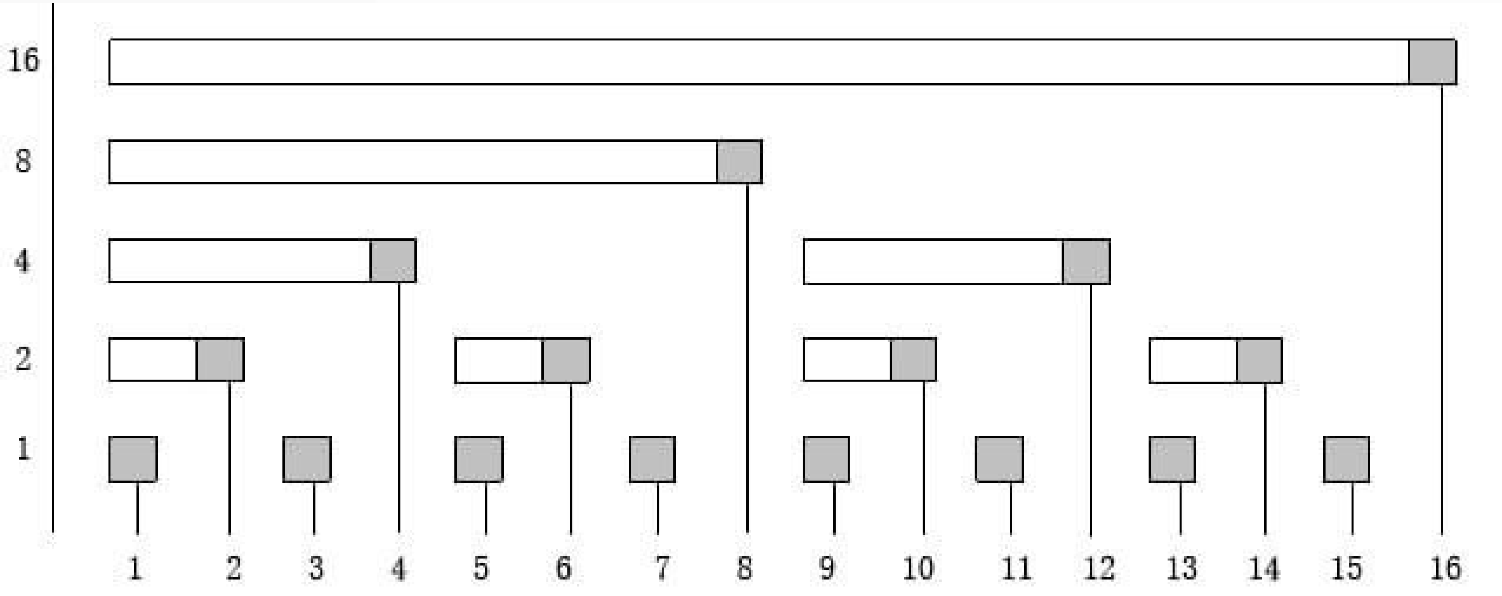
Binary Indexed Tree

- It is a simplified segment tree
- Define lowbit(x) as the value of the rightmost bit in binary representation of x
- Let x = 22 = 10110_2, lowbit(x) = 00010_2 = 2
- Node x maintain information for [x – lowbit(x) + 1, x]
- lowbit(x) = x & -x

Binary Indexed Tree

- Given an array A
- Support two operation
  - update(id, val)
    - Add val to A[id]
  - sum(id)
    - Find sum of A from 1 to id

BIT Visualization



Picture from https://www.hrwhisper.me/binary-indexed-tree-fenwick-tree/

Implementing BIT

- **Add (id, val)**

```
while id <= N
    add val to BIT[id]
    id = id + id & -id    ←--- adding its lowbit
```

- **Sum (id)**

```
ans = 0
while id > 0
    add BIT[id] to ans
    id = id – id & -id    ←--- subtracting its lowbit
```

Binary Indexed Tree

- https://www.topcoder.com/community/data-science/data-science-tutorials/binary-indexed-trees/

BIT Practice Problems

- **CF 830B**
- **CF 369E**

Trie

- **Trie is a data structure that stores string**
- **It is a Tree**
- **Each edge represent an alphabet**
- **The string represented by node v is the path from the root to v**
- **A node also has to store whether the string exist in trie**

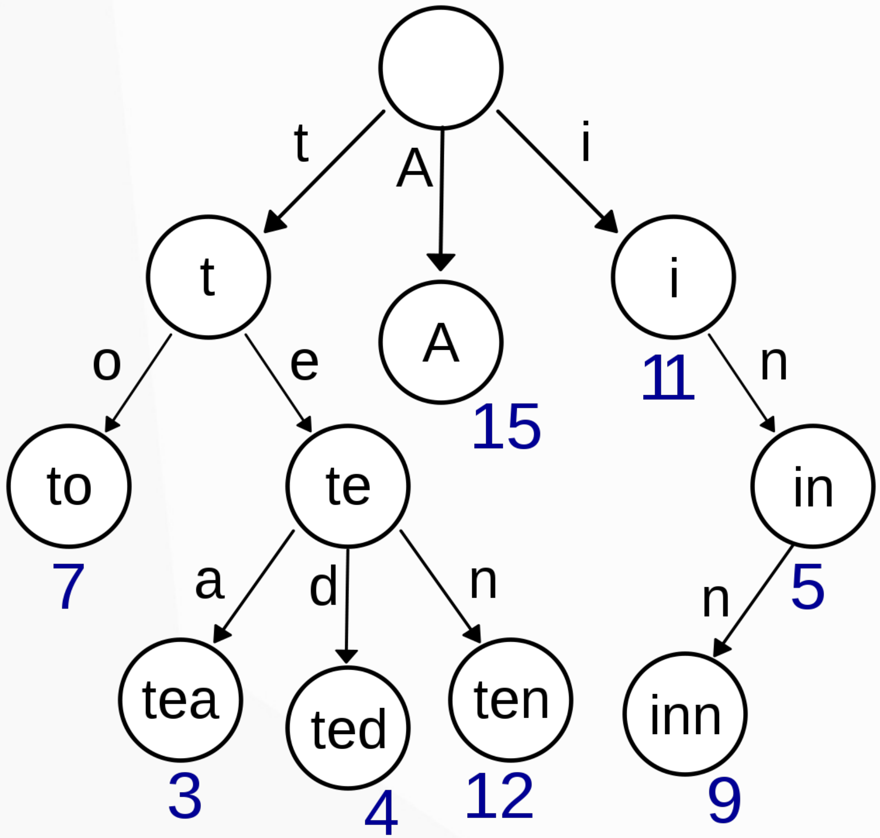
Trie

- Trie supports two operations
  - Adding a string S
  - Finding a string S
- Each of them is O(|S|)

Implementing Trie

- Very easy
- For the add operation
  - Start from root
  - If there isn’t an edge represent current alphabet, add an edge
  - Move to the next node via the edge represent current alphabet
  - Check for the next alphabet
  - If it is the end of the string, add a finish mark to the node
- For the find operation, it is similar to add. But we have to check if there is a finishing mark in the last node

Trie visualization



Picture from wikipedia

Practice Problems for Trie

- CF 514C
- CF 817E