Data Structure (III)

GagGuy & Sampson

Overview

- Disjoint sets
- Segment tree
- Binary Indexed Tree
- Trie

Review

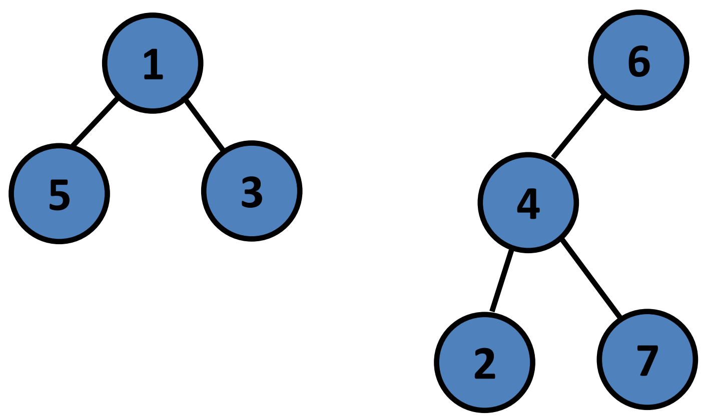
- Binary Search Tree
- Heap (Priority Queue)
- Hash Tables

The Union-Find Problem

- *N* balls initially, each ball in its own bag
  - Label the balls 1, 2, 3, ..., *N*
- Two kinds of operations:
  - Pick two bags, put all balls in bag 1 into bag 2 **(Union)**
  - Given 2 balls, ask whether they belongs to the same bag **(Find)**

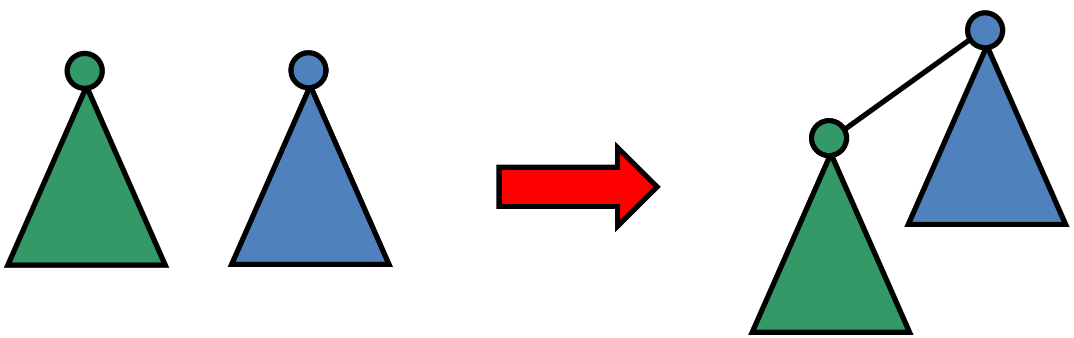
Disjoint Set

- A forest is a collection of trees
- Each bag is represented by a rooted tree, with the root being the representative ball

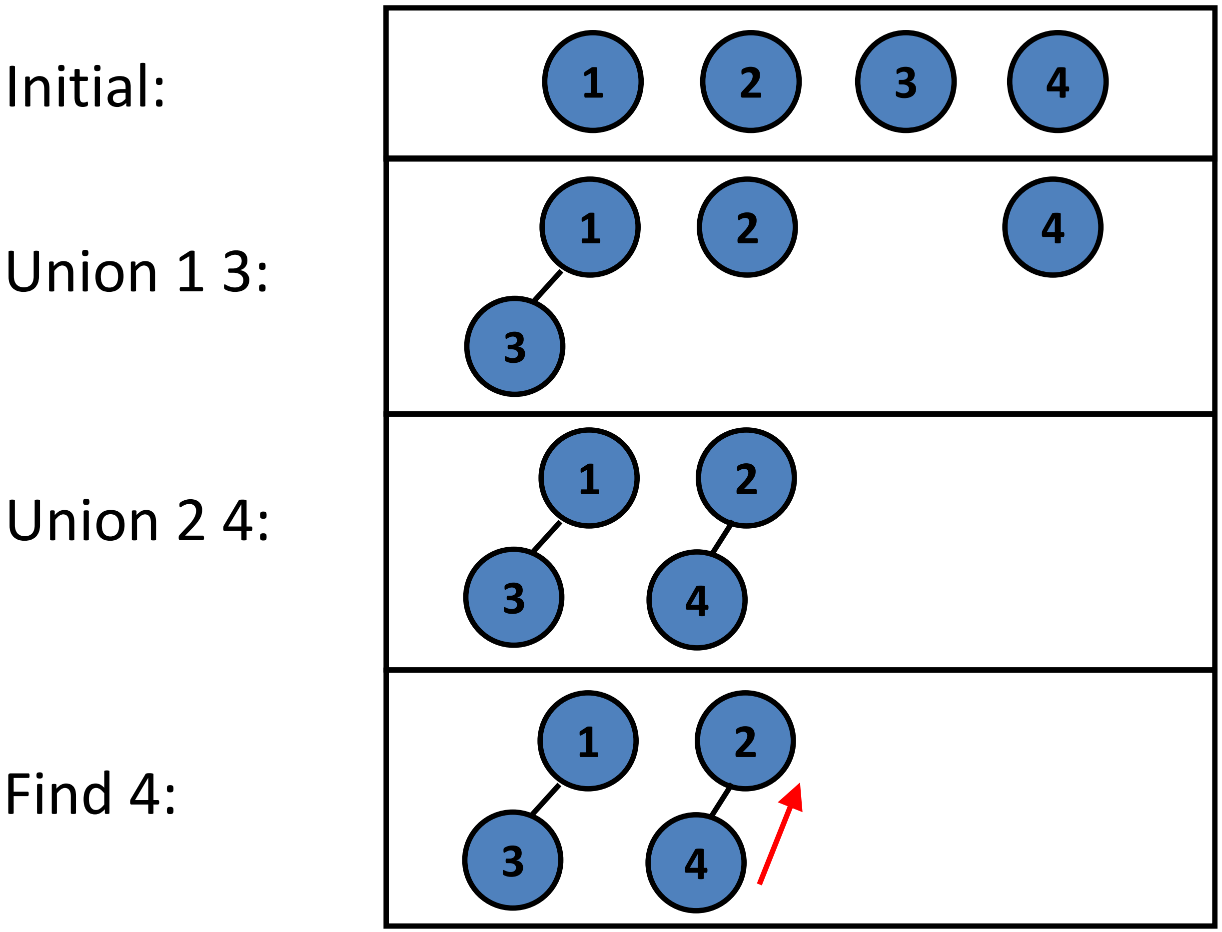


Disjoint Set

- Find(*x*)
  - Traverse from *x* up to the root
- Union(*x*, *y*)
  - Merge the two trees containing *x* and *y*



Disjoint Set



Disjoint Set

- Representing the tree
- Parent array
  - Parent[x] := parent of x
  - If x is a root, then parent[x] = x

Disjoint Set

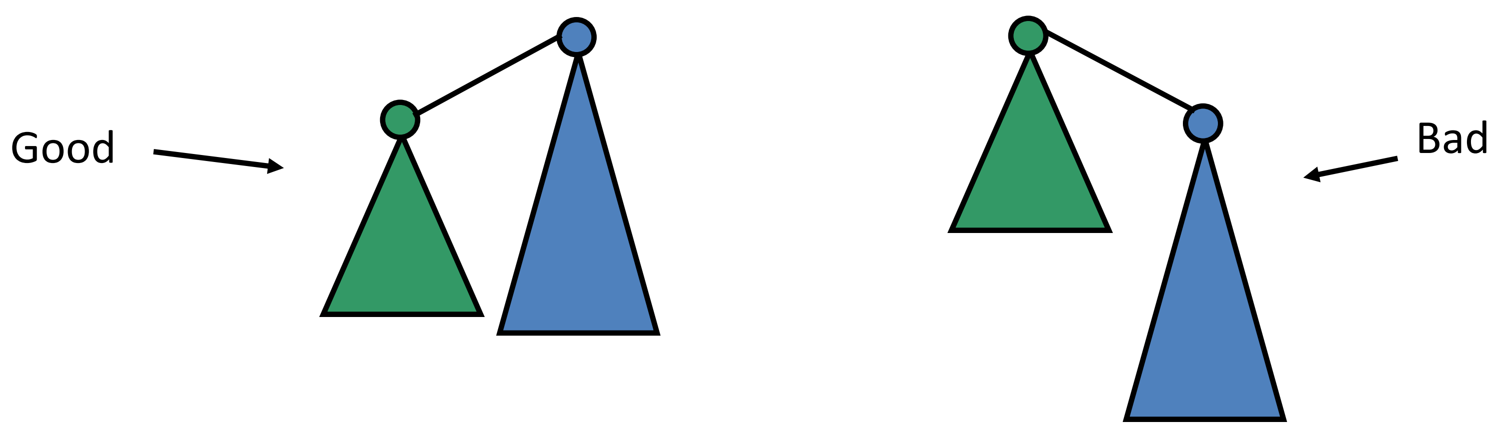
```
int find(int x){
  while (parent[x]!=x) x = parent[x];
  return x;
}

void union(int x,int y){
  parent[y] = x;
}
```

Disjoint Set

- Worst case
  - O(NM)
- Improvements
  - Union by rank
  - Path compressions

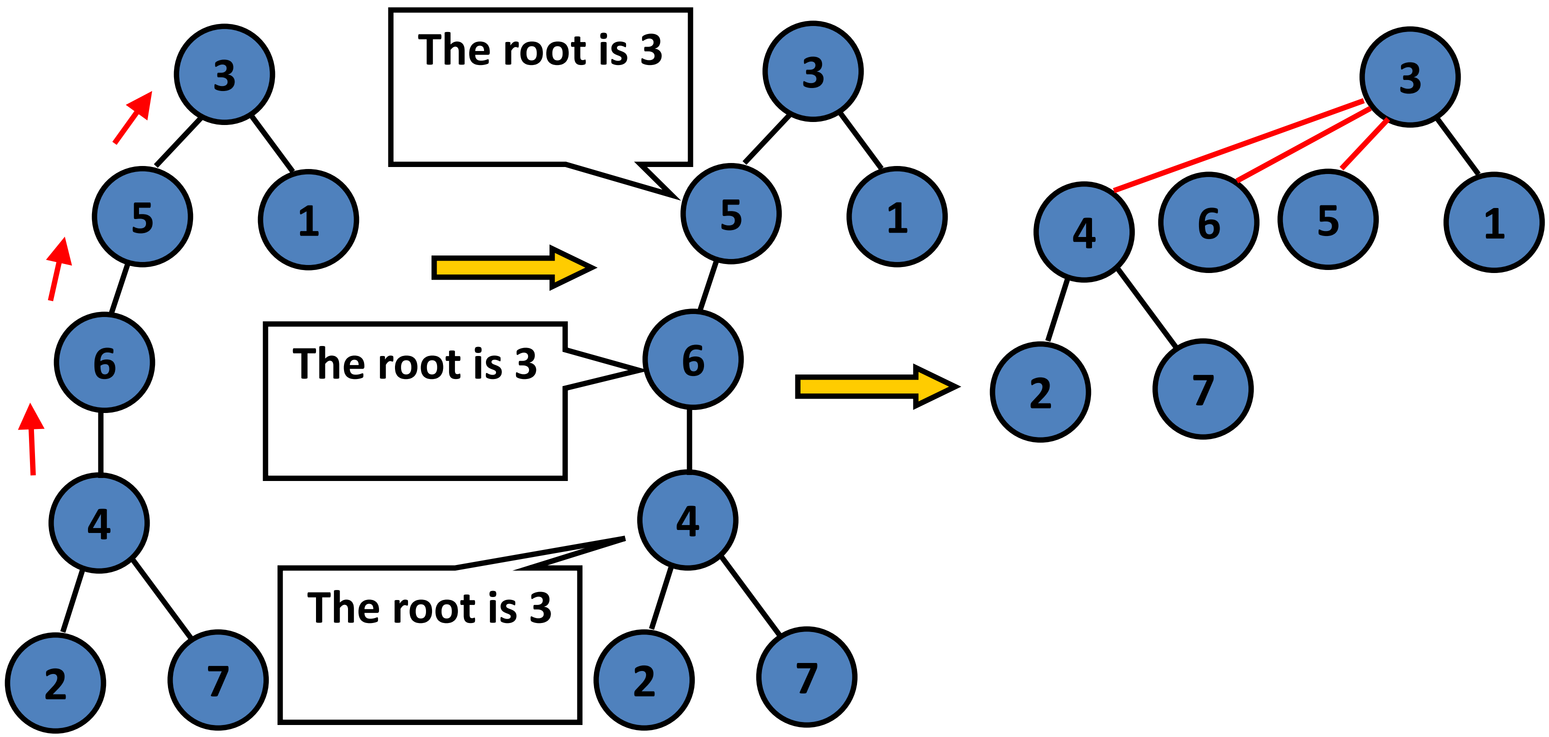
Disjoint Set – Union by rank

- We should avoid tall trees
- Root of the taller tree becomes the new root when union
- So, keep track of tree heights (ranks)



Disjoint Set – Path Compression

- Find(4)



Disjoint Set

- Time complexity using Union by rank + Path compression
- O($\alpha(N)$) for each query
  - Amortized time
  - $\alpha(N) \leq 5$ for practically large N

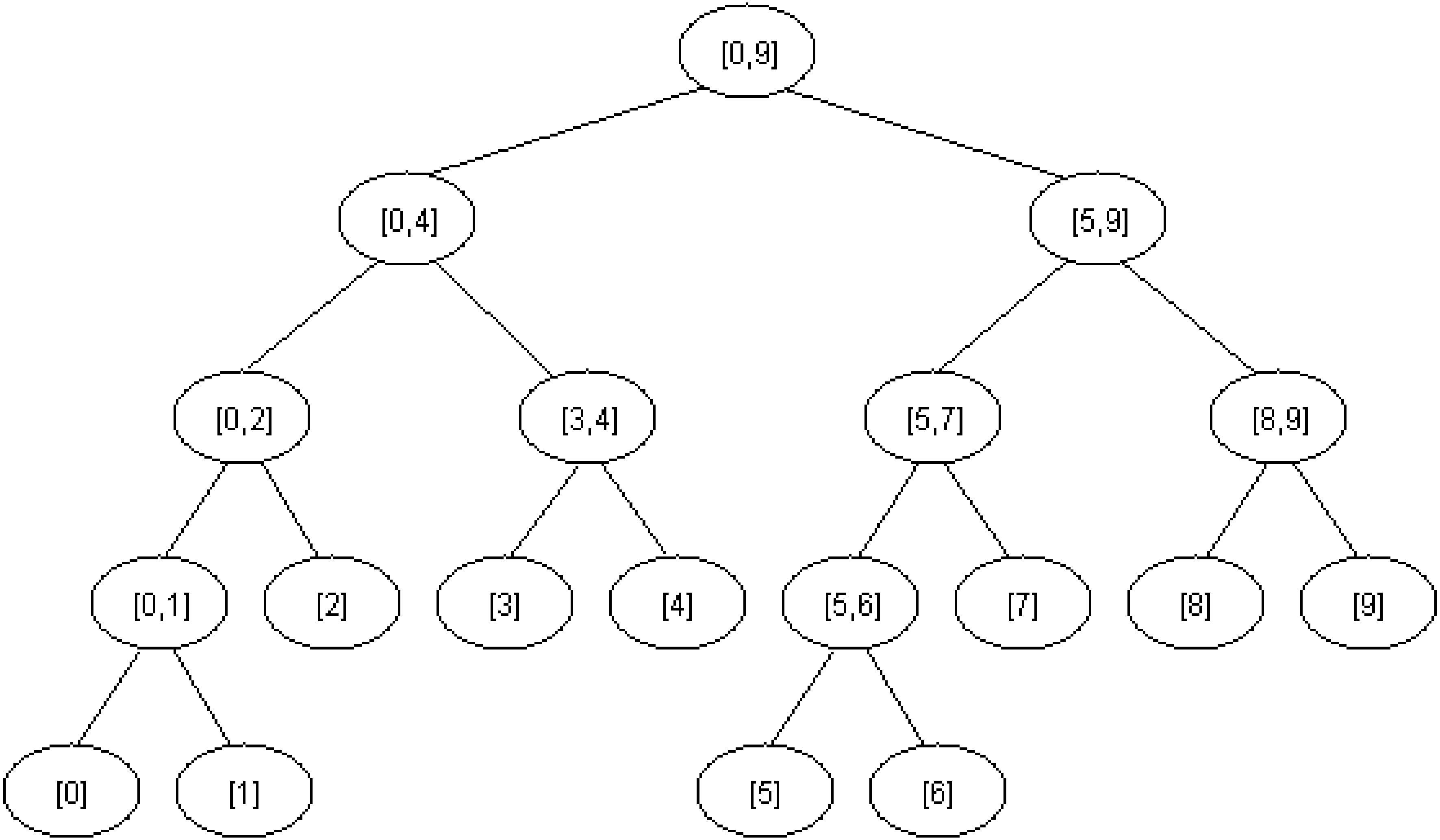
Range Maximum Query

- Given an integer array A
- Query(x,y)
  - Ask for the maximum element in A[x] to A[y]
- Update(x,val)
  - Set A[x] to val

Segment Tree

- Binary Tree
- Each node represent a segment
- Root = [1,N]
- Parent = [x,y]
  - Left child = [x, (x+y)/2]
  - Right child = [(x+y)/2+1, y]
- Tree height = lg N

Segment Tree



Range Maximum Query

- Given an integer array A
- Query(x,y)
  - Ask for the maximum element in A[x] to A[y]

- Each node with interval [l,h]
  - Store the maximum element in A[l] to A[h]

Segment Tree

- Build
  - O(N)
- Query
  - O(log N)
- Update
  - O(log N)

Example 1

- Given an array A with N elements.
- Q operations
  - Update(x,y,v): add v to A[x], A[x+1], .. , A[y]
  - Query(x): find the value of A[x]

Example 2

- Given an array A with N elements.
- Q operations
  - Find(): return the *maximum subsequence sum*
  - Update(x,v): change the value of A[x] to v

Example 3

- There is a 1 x N wall
- Each time the painter will paint color *c* from grid x to grid y
- Return the color of each grid after all paint operations

Example 4

- Given N rectangles on 2D plane
- Find the union area

Implementations

```
struct node{
     int left, right, maxval;
}    Tree[MAX_N];


/*Call build(1, range_x, range_y)*/
void build(int ind, int x, int y){
     Tree[ind].left = x;
     Tree[ind].right = y;
     if (x!=y){
             build(ind*2, x, (x+y)/2);
             build(ind*2+1, (x+y)/2+1, y);
             Tree[ind].maxval = max(Tree[ind*2].maxval, Tree[ind*2+1].maxval);
     }
     else Tree[ind].maxval = a[x];
}

/*Return the largest value in a[x]..a[y]*/
int query(int ind, int x, int y){
     if (Tree[ind].left<=x && y<=Tree[ind].right) return Tree[ind].maxval;
     int leftmax, rightmax;
     leftmax = -INF;
     rightmax = -INF;
     if (x<=Tree[ind*2].right) leftmax = query(ind*2, x, y);
     if (y>=Tree[ind*2+1].left) rightmax = query(ind*2+1, x, y);
     return max(leftmax, rightmax);
}
```

Further reading

- http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=lowestCommonAncestor

Binary Indexed Tree

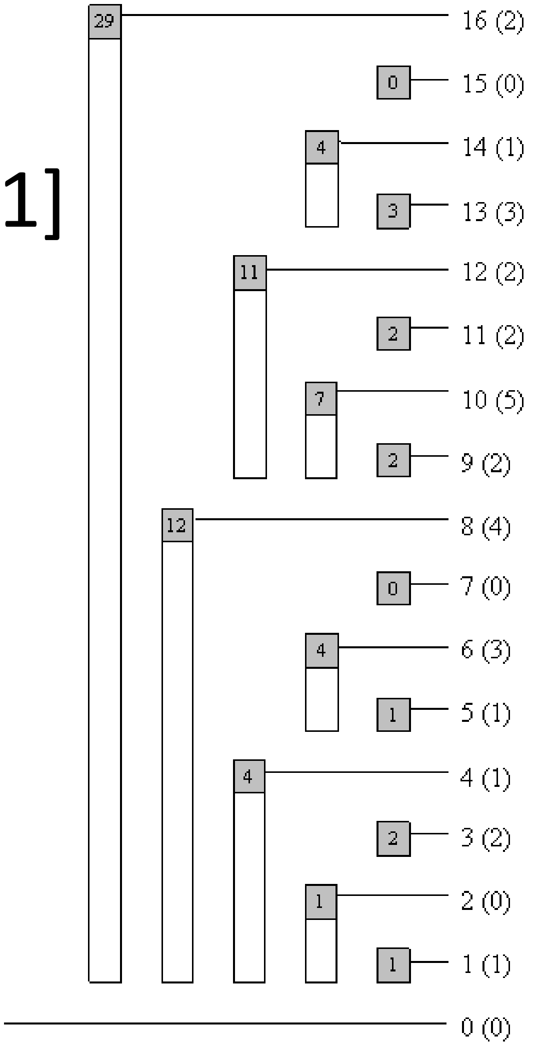
- Simplified segment tree
- Define lowbit(x) = the value of the rightmost 1 in the binary representation of x
- Let x = 22 = 10110_2, lowbit(x) = 00010_2 = 2
- Node x is responsible for [x – lowbit(x) + 1, x]
- An array of size N needs a BIT of size N
- lowbit(x) = x & -x

Binary Indexed Tree

- Given an array A with N elements.
- Q operations
  - Update(x,v): add v to A[x]
  - Sum(x): find the value of A[1] + A[2] + … + A[x]

Binary Indexed Tree

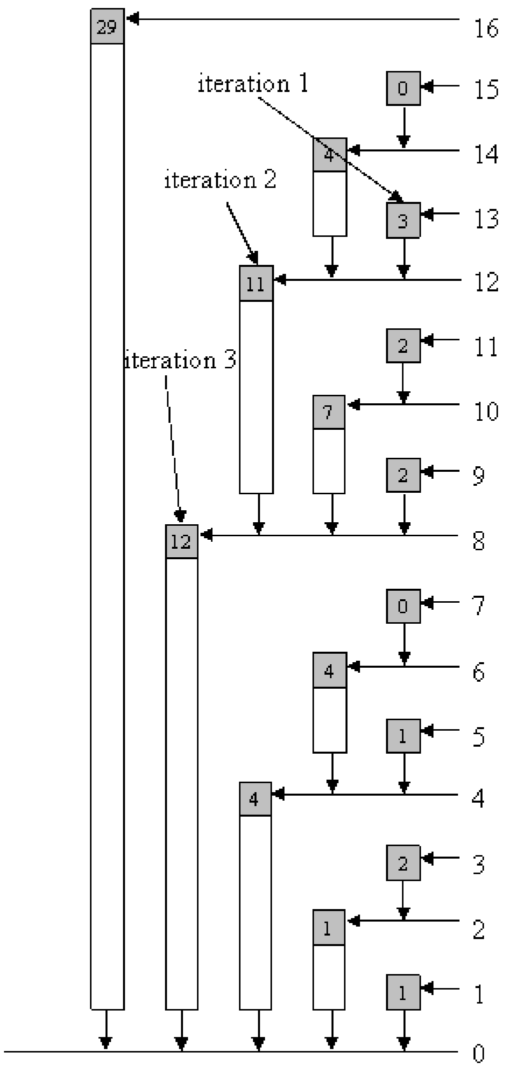
- Node x represents the value of A[x – lowbit(x) + 1] + … + A[x – 1]
+ A[x]



Binary Indexed Tree

- To find sum(13), we iterate through node 13, 12 and 8
- 13: 1011_2
- 12: 1010_2
- 8: 1000_2

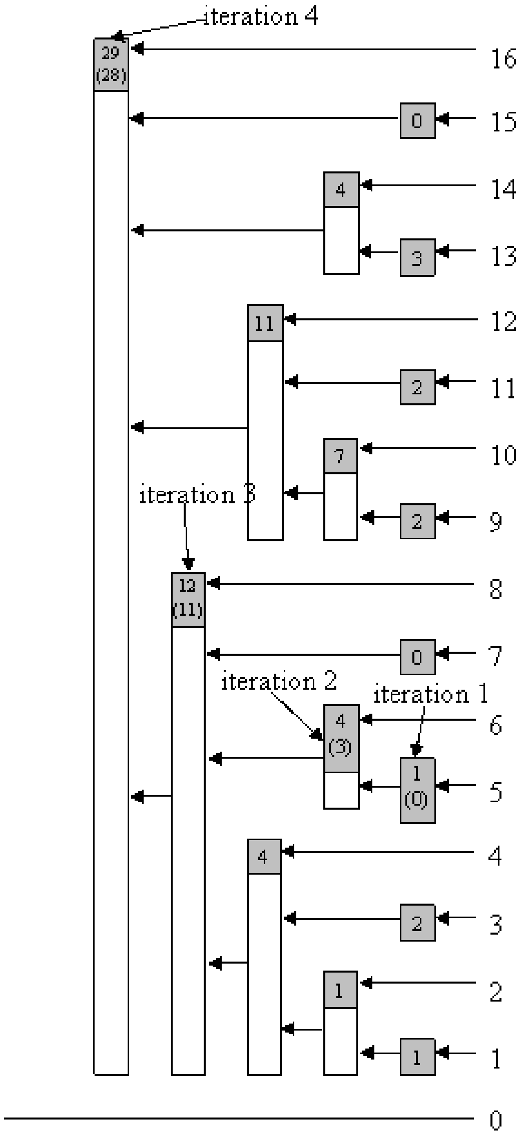
- for(int i = x; i > 0; i -= i & -i)



Binary Indexed Tree

- For Update(5, 1), we iterate through node 5, 6, 8 and 16
- 5: 00101_2
- 6: 00110_2
- 8: 01000_2
- 16: 10000_2

- for(int i = x; i <= n; i += i & -i)



Binary Indexed Tree

- Build
  - O(N)
- Query
  - O(log N)
- Update
  - O(log N)
- Advantage compared to segment tree
  - Shorter code length
  - Smaller constant

Trie

- Given a dictionary of N words
- M queries
- For each query, determine whether the given string is a word in the dictionary

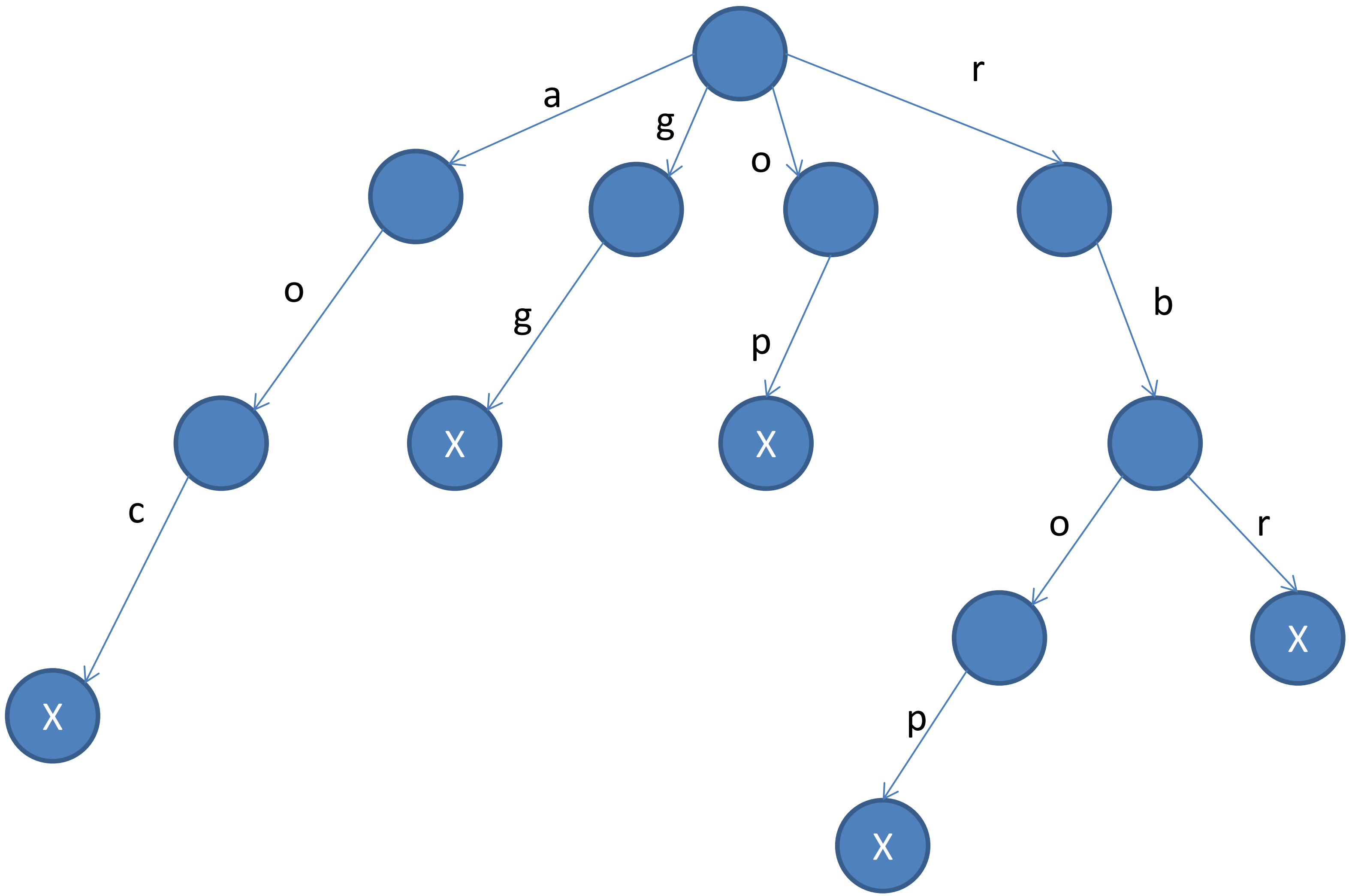
- Solution 1:
- Sort the words in lexicographical order
- Binary search

Trie

- Dictionary: {“rbop”, “rbr”, “op”, “gg”, “aoc”}
- Sorted: {“aoc”, “gg”, “op”, “rbr”, “rbop”}

- Checking whether “aoc” is a word
- Binary searching
- “aoc” < “op”
- “aoc” < “gg”
- “aoc” = “aoc”

Trie



Trie

- If input strings only have ‘a’ to ‘z’
  - Each node has 26 edges corresponding to letters ‘a’, ‘b’, ‘c’, …, ‘z’
- The string represented by node i is the path from the root to node i
- A node also has to store whether the represented string is a valid word

Trie

- Inserting string “op” into trie t

- We start from the root
- Check if the root has an “o” edge
- If not, create the edge and the new node “o”
- Move the current node to “o”

- Check if this node has an “p” edge
- If not, create the edge and the new node “op”
- Move the current node to “op”

- Put a mark in the current node (to mark “op” as a valid word)

Trie

- For searching, the process is similar except that we quit when the corresponding edge is not found
- If we reach the corresponding node in the trie, don't forget to check whether it is a word

- Insertion: O(|S|)
- Searching: O(|S|)
- Memory: O(|S| * Alphabet_size)