Data Structures

Jaehyun Park

CS 97SI
Stanford University

June 29, 2015

Typical Quarter at Stanford

```
void quarter() {
    while(true) { // no break :(
        task x = GetNextTask(tasks);
        process(x);
        // new tasks may enter
    }
}
```

- `GetNextTask()` decides the order of the tasks

Deciding the Order of the Tasks

- Possible behaviors of GetNextTask():
  - Returns the newest task (stack)
  - Returns the oldest task (queue)
  - Returns the most urgent task (priority queue)
  - Returns the easiest task (priority queue)

- `GetNextTask()` should run fast
  - We do this by storing the tasks in a clever way

Outline

Stack

- Last in, first out (LIFO)
- Supports three constant-time operations
  - `Push(x)`: inserts `x` into the stack
  - `Pop()`: removes the newest item
  - `Top()`: returns the newest item
- Very easy to implement using an array

Stack Implementation

- Have a large enough array `s[]` and a counter `k`, which starts at zero
  - `Push(x)`: set `s[k] = x` and increment `k` by 1
  - `Pop()`: decrement `k` by 1
  - `Top()`: returns `s[k - 1]` (error if `k` is zero)
- C++ and Java have implementations of stack
  - `stack` (C++), `Stack` (Java)
- But you should be able to implement it from scratch

Queue

- First in, first out (FIFO)
- Supports three constant-time operations
  - `Enqueue(x)`: inserts `x` into the queue
  - `Dequeue()`: removes the oldest item
  - `Front()`: returns the oldest item
- Implementation is similar to that of stack

Queue Implementation

- Assume that you know the total number of elements that enter the queue
  - ... which allows you to use an array for implementation
- Maintain two indices `head` and `tail`
  - `Dequeue()` increments `head`
  - `Enqueue()` increments `tail`
  - Use the value of `tail - head` to check emptiness
- You can use `queue` (C++) and `Queue` (Java)

Outline

Priority Queue

- Each element in a PQ has a priority value
- Three operations:
  - `Insert(x, p)`: inserts `x` into the PQ, whose priority is `p`
  - `RemoveTop()`: removes the element with the highest priority
  - `Top()`: returns the element with the highest priority
- All operations can be done quickly if implemented using a heap
- `priority_queue` (C++), `PriorityQueue` (Java)

Heap

- Complete binary tree with the heap property:
  - The value of a node $\geq$ values of its children
- The root node has the maximum value
  - Constant-time `top()` operation
- Inserting/removing a node can be done in $O(\log n)$ time without breaking the heap property
  - May need rearrangement of some nodes

Heap Example

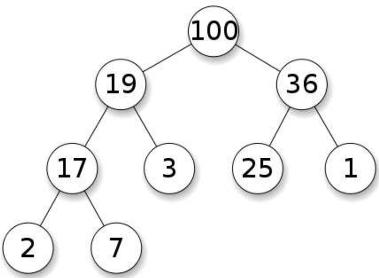
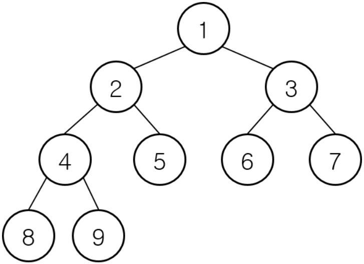


Figure from Wikipedia

Indexing the Nodes



- Start from the root, number the nodes $1, 2, \ldots$ from left to right
- Given a node k easy to compute the indices of its parent and children
  - Parent node: $\lfloor k/2 \rfloor$
  - Children: $2k, 2k+1$

Inserting a Node

1. Make a new node in the last level, as far left as possible
   - If the last level is full, make a new one
2. If the new node breaks the heap property, swap with its parent node
   - The new node moves up the tree, which may introduce another conflict
3. Repeat 2 until all conflicts are resolved

- Running time = tree height = $O(\log n)$

Implementation: Node Insertion

- Inserting a new node with value v into a heap H

```
void InsertNode(int v) {
    H[++n] = v;
    for(int k = n; k > 1; k /= 2) {
        if(H[k] > H[k / 2])
            swap(H[k], H[k / 2]);
        else break;
    }
}
```

Deleting the Root Node

1. Remove the root, and bring the last node (rightmost node in the last level) to the root
2. If the root breaks the heap property, look at its children and swap it with the larger one
   - Swapping can introduce another conflict
3. Repeat 2 until all conflicts are resolved

- Running time $= O(\log n)$
- Exercise: implementation
  - Some edge cases to consider

Outline

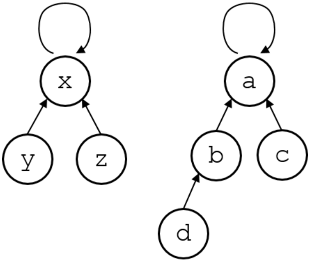
Union-Find Structure

- Used to store disjoint sets
- Can support two types of operations efficiently
  - `Find(x)`: returns the "representative" of the set that `x` belongs
  - `Union(x, y)`: merges two sets that contain `x` and `y`

- Both operations can be done in (essentially) constant time
- Super-short implementation!

Union-Find Structure

- Main idea: represent each set by a rooted tree
  - Every node maintains a link to its parent
  - A root node is the "representative" of the corresponding set
  - Example: two sets $\{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$ and $\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$



Implementation Idea

- `Find(x)`: follow the links from `x` until a node points itself
  - This can take $O(n)$ time but we will make it faster

- `Union(x, y)`: run `Find(x)` and `Find(y)` to find corresponding root nodes and direct one to the other

Implementation

- We will assume that the links are stored in `L[]`

```
int Find(int x) {
    while(x != L[x]) x = L[x];
    return x;
}
void Union(int x, int y) {
    L[Find(x)] = Find(y);
}
```

Path Compression

- In a bad case, the trees can become too deep
  - ... which slows down future operations
- Path compression makes the trees shallower every time `Find()` is called
- We don't care how a tree looks like as long as the root stays the same
  - After `Find(x)` returns the root, backtrack to `x` and reroute all the links to the root

Path Compression Implementations

```
int Find(int x) {
    if(x == L[x]) return x;
    int root = Find(L[x]);
    L[x] = root;
    return root;
}

int Find(int x) {
    return x == L[x] ? x : L[x] = Find(L[x]);
}
```

Outline

Binary Search Tree (BST)

- A binary tree with the following property: for each node ,
  - value of $v \geq$ values in v's left subtree
  - value of $v \leq$ dvalues in v's right subtree

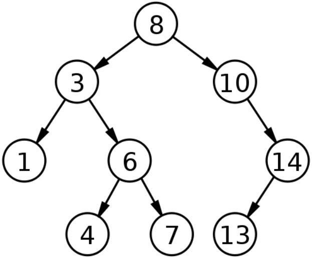


Figure from Wikipedia

What BSTs can do

- Supports three operations
  - `Insert(x)`: inserts a node with value `x`
  - `Delete(x)`: deletes a node with value `x`, if there is any
  - `Find(x)`: returns the node with value `x`, if there is any
- Many extensions are possible
  - `Count(x)`: counts the number of nodes with value less than or equal to `x`
  - `GetNext(x)`: returns the smallest node with value $\geq$ `x`

BSTs in Programming Contests

- Simple implementation cannot guarantee efficiency
  - In worst case, tree height becomes n (which makes BST useless)
  - Guaranteeing $O(\log n)$ running time per operation requires balancing of the tree (hard to implement)
  - We will skip the implementation details
- Use the standard library implementations
  - `set`, `map` (C++)
  - `TreeSet`, `TreeMap` (Java)

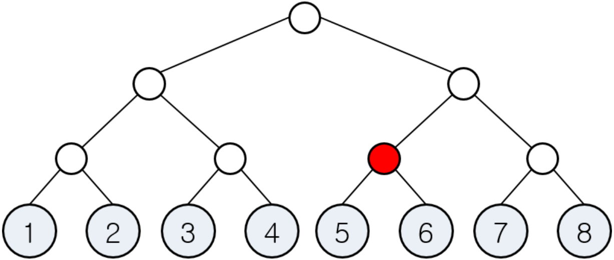
Outline

Fenwick Tree

- A variant of segment trees
- Supports very useful interval operations
  - `Set(k, x)`: sets the value of `k`th item equal to `x`
  - `Sum(k)`: computes the sum of items $1, \dots, \texttt{k}$ (prefix sum)
    - Note: sum of items $\texttt{i}, \dots, \texttt{j} = \texttt{Sum}(\texttt{j}) - \texttt{Sum}(\texttt{i} - 1)$
- Both operations can be done in $O(\log n)$ time using $O(n)$ space

Fenwick Tree Structure

- Full binary tree with at least n leaf nodes
  - We will use $n = 8$ for our example
- kth leaf node stores the value of item k
- Each internal node stores the sum of values of its children
  - *e.g.*, Red node stores `item[5]` $+$ `item[6]`

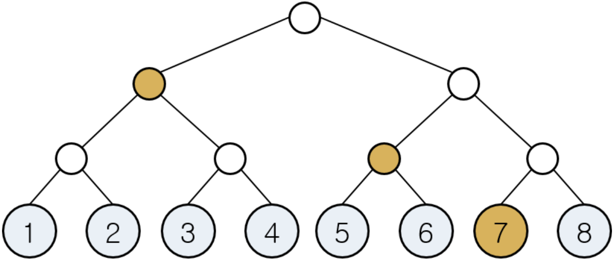


Summing Consecutive Values

- Main idea: choose the minimal set of nodes whose sum gives the desired value
- We will see that
  - at most 1 node is chosen at each level so that the total number of nodes we look at is $\log_2 n$
  - and this can be done in $O(\log n)$ time
- Let's start with some examples

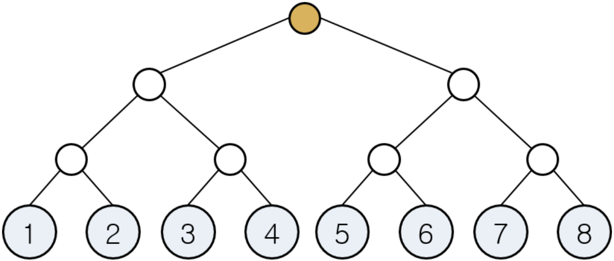
Summing: Example 1

- $\texttt{Sum}(7)$ = sum of the values of gold-colored nodes



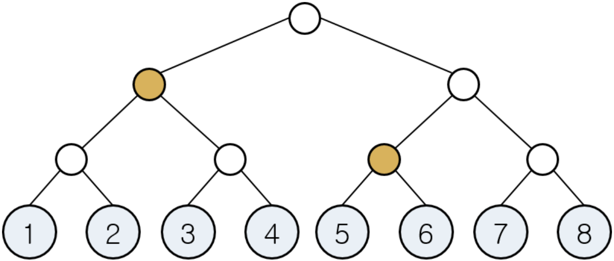
Summing: Example 2

- `Sum(8)`



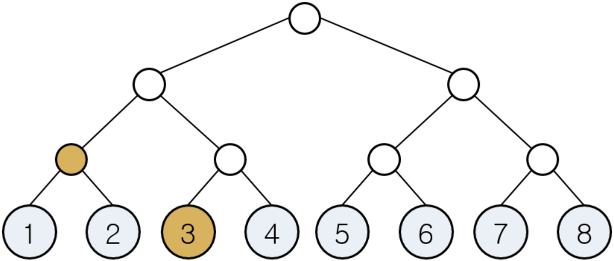
Summing: Example 3

- `Sum(6)`



Summing: Example 4

- `Sum(3)`



Computing Prefix Sums

- Say we want to compute `Sum(k)`
- Maintain a pointer `P` which initially points at leaf `k`
- Climb the tree using the following procedure:
  - If `P` is pointing to a left child of some node:
    - Add the value of `P`
    - Set `P` to the parent node of `P`'s left neighbor
    - If `P` has no left neighbor, terminate
  - Otherwise:
    - Set `P` to the parent node of `P`
- Use an array to implement (review the heap section)

Updating a Value

- Say we want to do `Set(k, x)` (set the value of leaf `k` as `x`)
- This part is a lot easier
- Only the values of leaf `k` and its ancestors change

1. Start at leaf `k`, change its value to `x`
2. Go to its parent, and recompute its value
3. Repeat 2 until the root

Extension

- Make the `Sum()` function work for any interval
  - ... not just ones that start from item 1

- Can support more operations with the new `Sum()` function
  - `Min(i, j)`: Minimum element among items $\texttt{i}, \ldots, \texttt{j}$
  - `Max(i, j)`: Maximum element among items $\texttt{i}, \ldots, \texttt{j}$

Outline

Lowest Common Ancestor (LCA)

- Input: a rooted tree and a bunch of node pairs
- Output: lowest (deepest) common ancestors of the given pairs of nodes

- Goal: preprocessing the tree in $O(n \log n)$ time in order to answer each LCA query in $O(\log n)$ time

Preprocessing

- Each node stores its depth, as well as the links to every 2^kth ancestor
  - $O(\log n)$ additional storage per node
  - We will use `Anc[x][k]` to denote the 2^kth ancestor of node `x`
- Computing `Anc[x][k]`:
  - `Anc[x][0] = x`'s parent
  - `Anc[x][k] = Anc[ Anc[x][k-1] ][ k-1 ]`

Answering a Query

- Given two node indices $\mathtt{x}$ and $\mathtt{y}$
  - Without loss of generality, assume $\mathtt{depth}(\mathtt{x}) \leq \mathtt{depth}(\mathtt{y})$
- Maintain two pointers p and q, initially pointing at $\mathtt{x}$ and $\mathtt{y}$
- If $\mathtt{depth}(\mathtt{p}) < \mathtt{depth}(\mathtt{q})$, bring $\mathtt{q}$ to the same depth as $\mathtt{p}$
  - using `Anc` that we computed before
- Now we will assume that $\mathtt{depth}(\mathtt{p}) = \mathtt{depth}(\mathtt{q})$

Answering a Query

- If p and q are the same, return p
- Otherwise, initialize k as $\lceil \log_2 n \rceil$ and repeat:
  - If k is 0, return p's parent node
  - If Anc[p][k] is undefined, or if Anc[p][k] and Anc[q][k] point to the same node:
    - Decrease k by 1
  - Otherwise:
    - Set p = Anc[p][k] and q = Anc[q][k] to bring p and q up by 2^k levels

Conclusion

- We covered LOTS of stuff today
  - Try many small examples with pencil and paper to completely internalize the material
  - Review and solve relevant problems
- Discussion and collaboration are strongly recommended!