



DATA STRUCTURES

COMP 321 – McGill University

These slides are mainly compiled from the following resources.

- Professor Jaehyun Park' slides CS 97SI
- Top-coder tutorials.
- Programming Challenges book.

Data Structure

- A way to store and organize data in order to support efficient insertions, queries, searches, updates, and deletions.

Data Structure

- Basic data structures (built-in libraries).
 - Linear DS.
 - Non-Linear DS.
- Data structures (Own libraries).
 - Graphs.
 - Union-Find Structures.
 - Segment Tree.

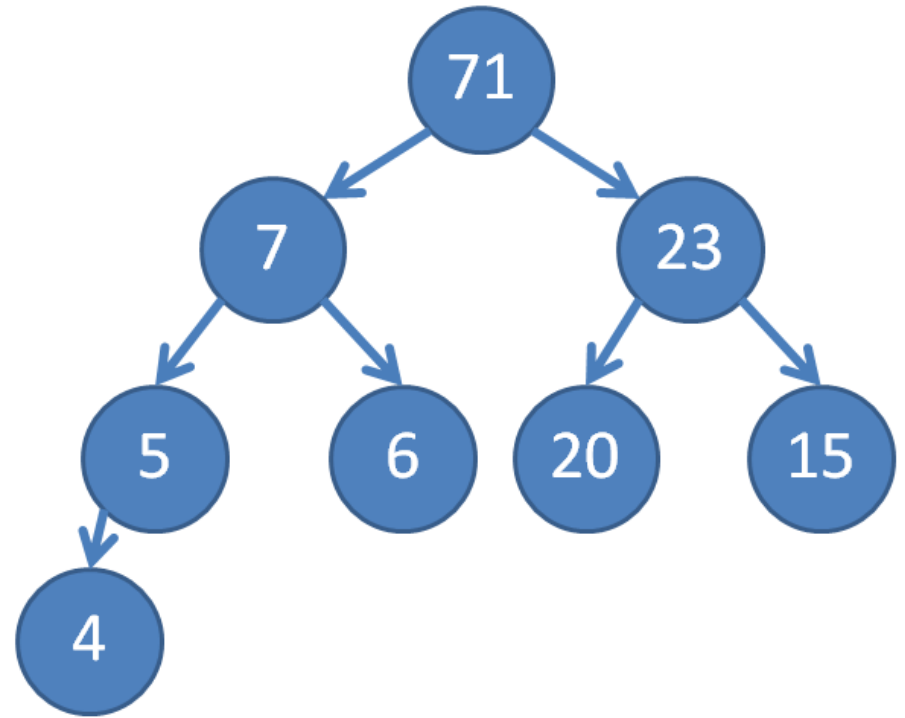
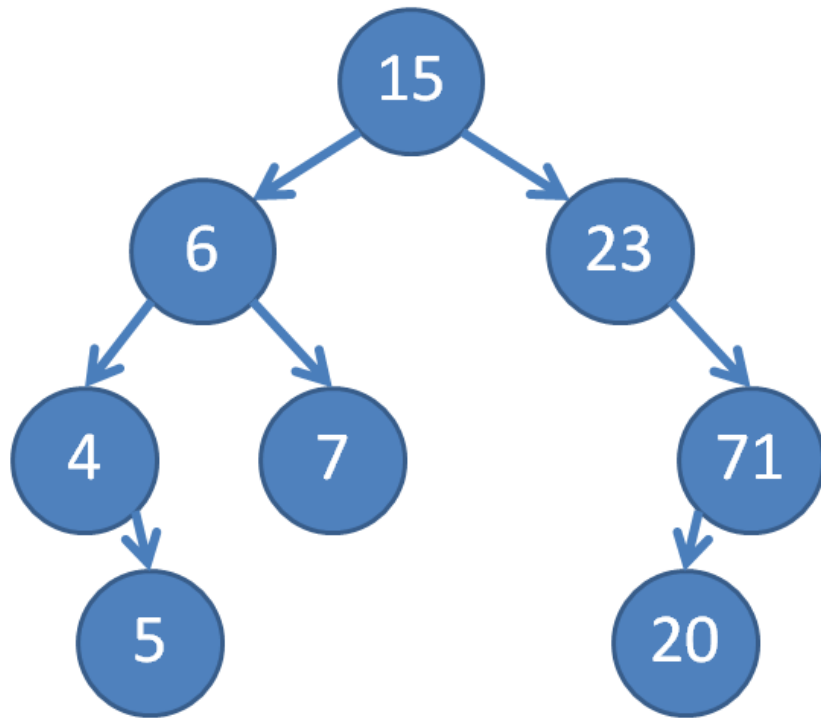
Data Structures

- Basic data structures (built-in libraries).
 - Linear DS (ordering the elements sequentially).
 - Static Array (Array in C/C++ and in Java).
 - Resizeable array (C++ STL<vector> and Java ArrayList).
 - Linked List: (C++ STL<list> and Java LinkedList).
 - Stack (C++ STL<stack> and Java Stack).
 - Queue (C++ STL <queue> and Java Queue).

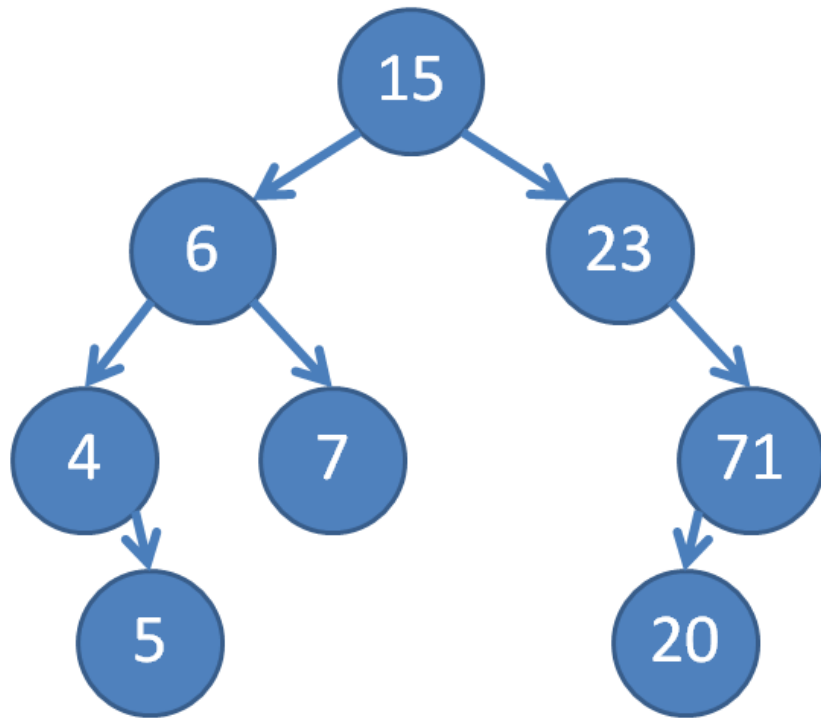
Data Structures

- Basic data structures (built-in libraries).
 - Non-Linear DS.
 - Balanced Binary Search Tree (C++ STL `<map>/<set>` and in Java `TreeMap/TreeSet`).
 - AVL and Red-Black Trees = Balanced BST
 - `<map>` stores (key -> data) VS `<set>` only stores the key
 - Heap (C++ STL `<queue>:priority_queue` and Java `PriorityQueue`).
 - BST complete.
 - Heap property VS BST property.
 - Hash Table (Java `HashMap/HashSet/HashTable`).
 - Non synchronized vs synchronized.
 - Null vs non-nulls
 - Predictable iteration (using `LinkedHashMap`) vs non predictable.

Question for you.

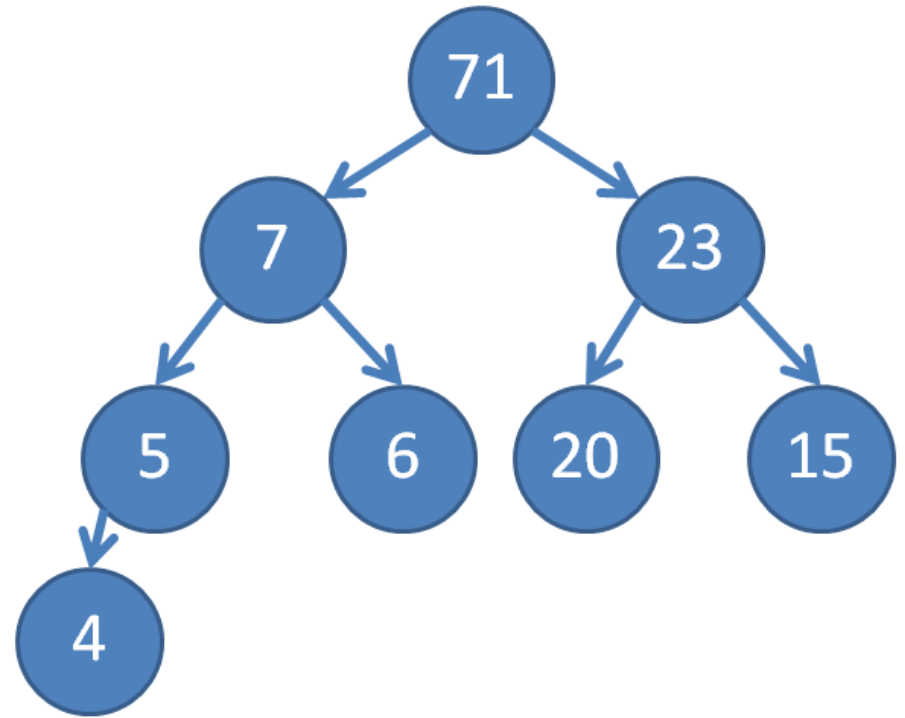


Question for you.



BST

Do you recognize the problem?



HEAP

Deciding the Order of the Tasks

- Returns the newest task (stack)
- Returns the oldest task (queue)
- Returns the most urgent task (priority queue)
- Returns the easiest task (priority queue)

STACK

- Last in, first out (Last In First Out)
- Stacks model piles of objects (such as dinner plates)
- 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

- 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)

STACK

- Useful for:
 - Processing nested formulas
 - Depth-first graph traversal
 - Data storage in recursive algorithms

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

- Assume that you know the total number of elements that enter the queue
 - ... which allows you to use an array for implementation
 - ... If not, you can use linked lists or double linked lists
- Maintain two indices head and tail
 - Dequeue() increments head
 - Enqueue() increments tail
 - Use the value of $\text{tail} - \text{head}$ to check emptiness
- You can use queue (C++) and Queue (Java)

QUEUE

- Useful for
 - implementing buffers
 - simulating waiting lists
 - shuffling cards

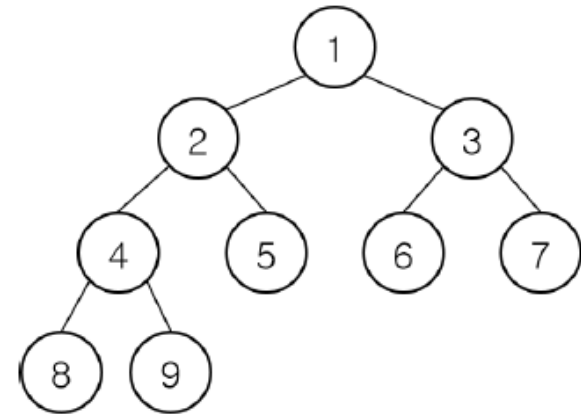
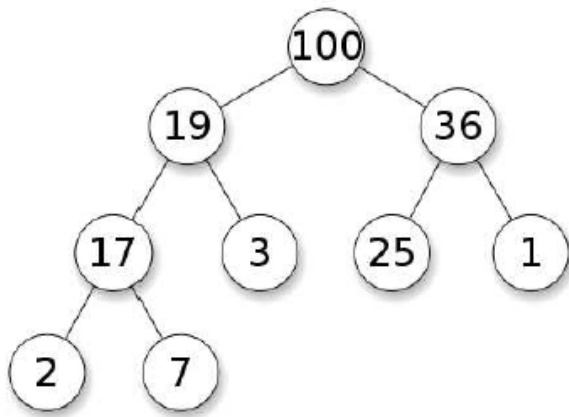
PRIORITY QUEUE

- 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 (if not use a sorted array)
- `priority_queue` (C++), `PriorityQueue` (Java)
- Useful for
 - Maintaining schedules / calendars
 - Simulating events
 - Sweepline geometric algorithms

HEAP

- Complete binary tree with the heap property:
 - The value of a node \geq values of its children
 - What is the difference between full vs complete?
- 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



- Start from the root, number the nodes 1, 2, . . . from left to right
- Given a node k easy to compute the indices of its parent and children
 - Parent node: $\text{floor}(k/2)$
 - Children: $2k, 2k + 1$

Heap – Inserting a Node

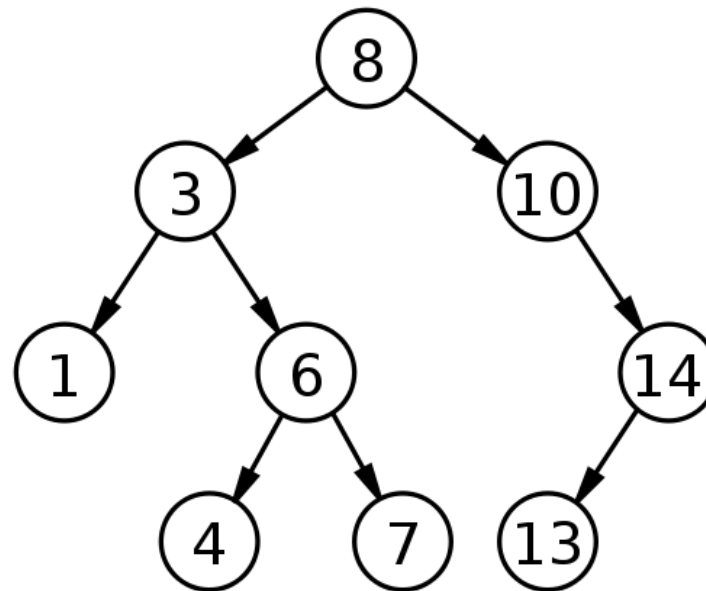
- 1. Make a new node in the last level, as far left as possible
 - If the last level is full, make a new one
- 2. If the new node breaks the heap property, swap with its parent node
 - The new node moves up the tree, which may introduce another conflict
- Repeat 2 until all conflicts are resolved
- Running time = tree height = $O(\log n)$

Heap – Deleting a 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 = $O(\log n)$

BINARY SEARCH TREE (BST)

- The idea behind is that each node has, at most, two children
- A binary tree with the following property: for each node v ,
 - value of $v \geq$ values in v 's left subtree
 - value of $v <$ values in v 's right subtree



BST

- 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 $\geq x$

BST

- 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).
 - For example AVL and Red-Black trees (We will skip the details of these balanced trees, but you should review it.).
 - What does balanced mean??
- Use the standard library implementations
 - set, map (C++)
 - TreeSet, TreeMap (Java)

BST

- 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).
 - For example AVL and Red-Black trees (We will skip the details of these balanced trees, but you should be review it.).
 - What does balanced mean??=> The heights of the two child subtrees of any node differ by at most one.
- Use the standard library implementations
 - set, map (C++)
 - TreeSet, TreeMap (Java)

Question for you

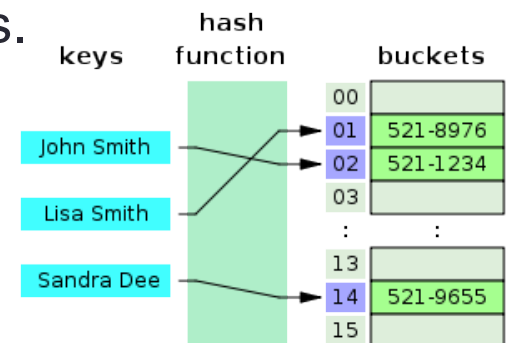
- Why a binary tree is preferable to an array of values that has been sorted?
 - $O(?)$ Finding a given key?

Question for you

- Why a binary tree is preferable to an array of values that has been sorted?
 - $O(\log n)$ to find a given key => traversing BST and binary search.
 - Problem is the adding of a new item.

Hash Tables

- A key is used as an index to locate the associated value.
 - Content-based retrieval, unlike position-based retrieval.
 - Hashing is the process of generating a key value.
 - An ideal algorithm must distribute evenly the hash values => the buckets will tend to fill up evenly = fast search.
 - A hash bucket containing more than one value is known as a “collision”.
 - Open addressing => A simple rule to decide where to put a new item when the desired space is already occupied.
 - Chaining => We associate a linked list with each table location.
- Hash tables are excellent dictionary data structures.



Hash Function

- A function that takes a string and outputs a number
 - A good hash function has few collisions
 - i.e. , If $x \neq y$, $H(x) \neq H(y)$ with high probability
- An easy and powerful hash function is a polynomial mod some prime p .
 - Consider each letter as a number (ASCII value is fine)
 - $H(x_1 \dots x_k) = x_1 a^{k-1} + x_2 a^{k-2} + \dots + x_{k-1} a + x_k \pmod{p}$

Data Structures

- Data structures (Own Libraries).
 - Graph.
 - Lets talk about graphs later.
 - Union-Find Disjoint Sets
 - Segment tree.

Union-Find Structure

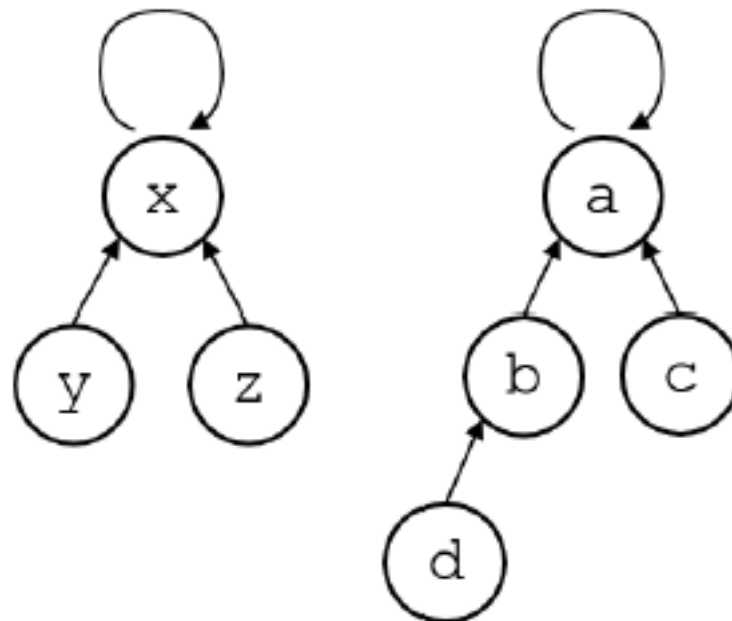
- Used to store disjoint sets
 - What is a disjoint set?
- Can support two types of operations efficiently
 - Find(x) : returns the “representative” of the set that x belongs
 - Union(x, y) : merges two sets that contain x and y
- Both operations can be done in (essentially) constant time
- Super-short implementation!
- Useful for problems involving partitioning.
 - Ex: keeping track of connected components.
 - Kruskal’s algorithm (minimum spanning tree).

Union-Find Structure

- Used to store disjoint sets
 - What is a disjoint set? => sets whose intersection is the empty set.
- 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!
- Useful for problems involving partitioning.
 - Ex: keeping track of connected components.
 - Kruskal’s algorithm (minimum spanning tree).

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 $\{x, y, z\}$ and $\{a, b, c, d\}$



Union-Find Structure

- 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.
- If we assume that the links are stored in L[], then

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

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


Union-Find Structure

- 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



Question for you

- How can you implement the operation `isSameSet(i,j)`?

Question for you

- How can you implement the operation `isSameSet(i,j)`?
 - simply calls `findSet(i)` and `findSet(j)` to check if both refer to the same representative.

Segment Tree

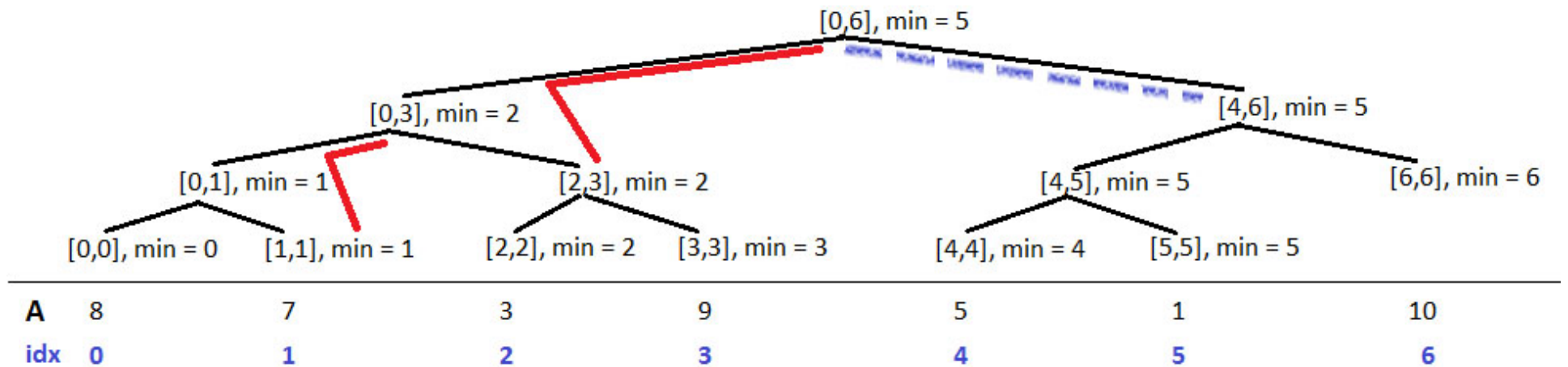
- DS to efficiently answer dynamic range queries.
 - Range Minimum Query (RMQ): finding the index of the minimum element in an array given a range: $[i..j]$.
 - Ex. $\text{RMQ}(1, 3) = 2$, $\text{RMQ}(3, 4) = 4$, $\text{RMQ}(0, 0) = 0$, $\text{RMQ}(0, 1) = 1$, and $\text{RMQ}(0, 6) = 5$.
 - Iterate takes $O(n)$, let make it faster using a binary tree similar to heap, but usually not a complete binary tree (aka segment tree).

| | | | | | | | | | | | | | | |
|---------|---|-------|--|---|--|---|--|---|--|---|--|---|--|----|
| Values | = | 8 | | 7 | | 3 | | 9 | | 5 | | 1 | | 10 |
| Array A | = | ----- | | | | | | | | | | | | |
| Indices | = | 0 | | 1 | | 2 | | 3 | | 4 | | 5 | | 6 |

Segment Tree

- Binary tree.
- Each node is associated with some interval of the array.
- Each non-leaf node has two children whose associated intervals are disjoint.
- Each child's interval has approximately half the size of the parent's interval.

Segment Tree



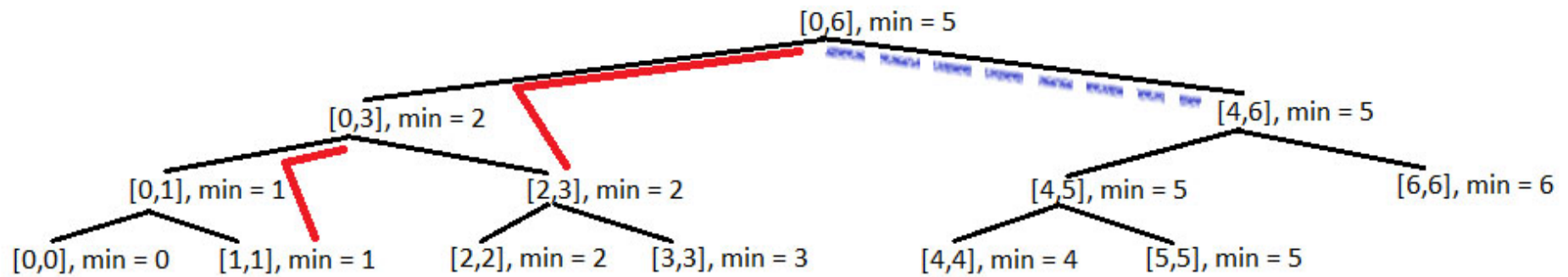
- Root $\Rightarrow [0, N - 1]$ and for each segment $[l, r]$ we split them into $[l, (l + r) / 2]$ and $[(l + r) / 2 + 1, r]$ until $l = r$.

Question for you

- What is the complexity of `built_segment_tree` $O(?)$?
- With segment tree ready, what is the complexity of answering an RMQ?
- Can you give the worst case? $RMQ(?,?)$

Question for you

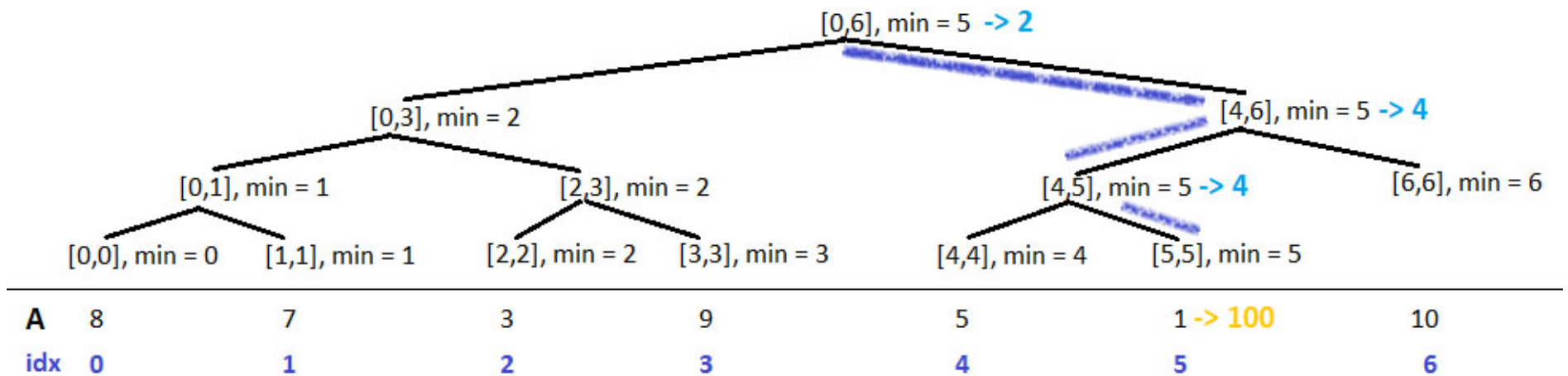
- What is the complexity of `built_segment_tree` $O(n)$
 - There are total $2n-1$ nodes.
- With segment tree ready, what is the complexity of answering an RMQ $\Rightarrow O(\log n)$ (2 root-to-leaf paths)
 - Ex RMQ(4,6) = blue line.
 - Ex RMQ(1,3) = red line.
 - Ex RMQ(3,4) = worst case \Rightarrow one path from $[0,6]$ to $[3,3]$ and another from $[0,6]$ to $[4,4]$.



| | | | | | | | |
|------------|---|---|---|---|---|---|----|
| A | 8 | 7 | 3 | 9 | 5 | 1 | 10 |
| idx | 0 | 1 | 2 | 3 | 4 | 5 | 6 |

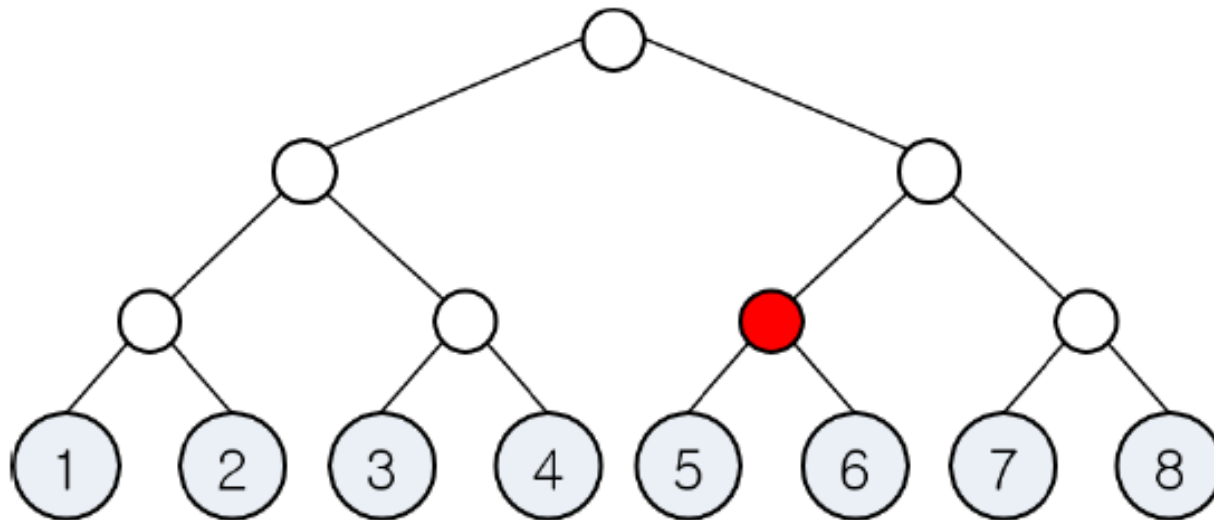
Segment Tree

- If the array A is static, then use a Dynamic Programming solution that requires $O(n \log n)$ pre-processing and $O(1)$ per RMQ.
 - Segment tree becomes useful if array A is frequently updated.
 - Ex. Updating $A[5]$ takes $O(\log n)$ vs $O(n \log n)$ required by DP.



Fenwick Tree

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

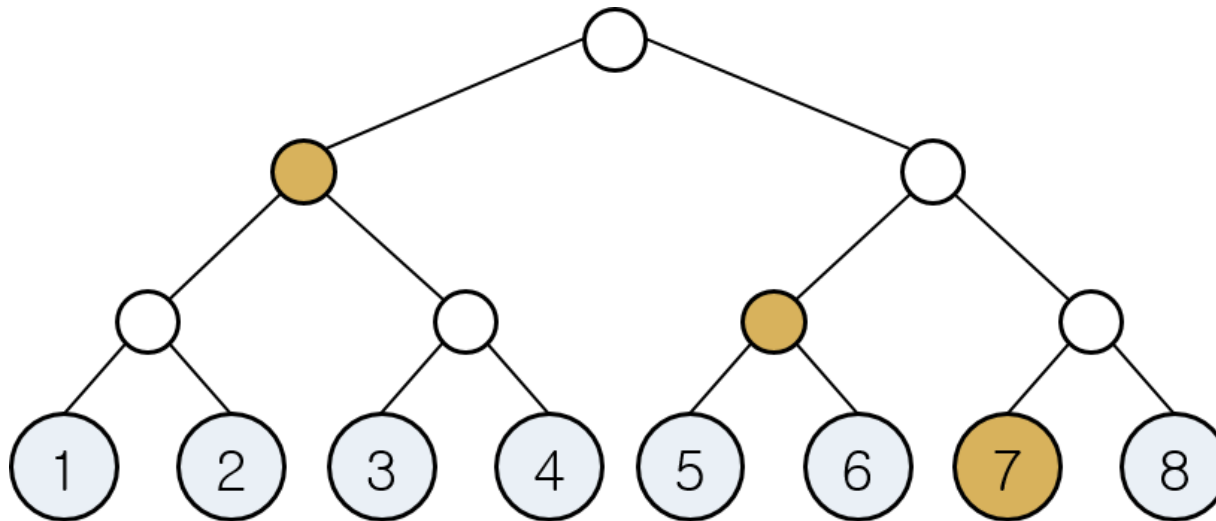


Summing Consecutive Values

- Main idea: choose the minimal set of nodes whose sum gives the desired value
 - 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

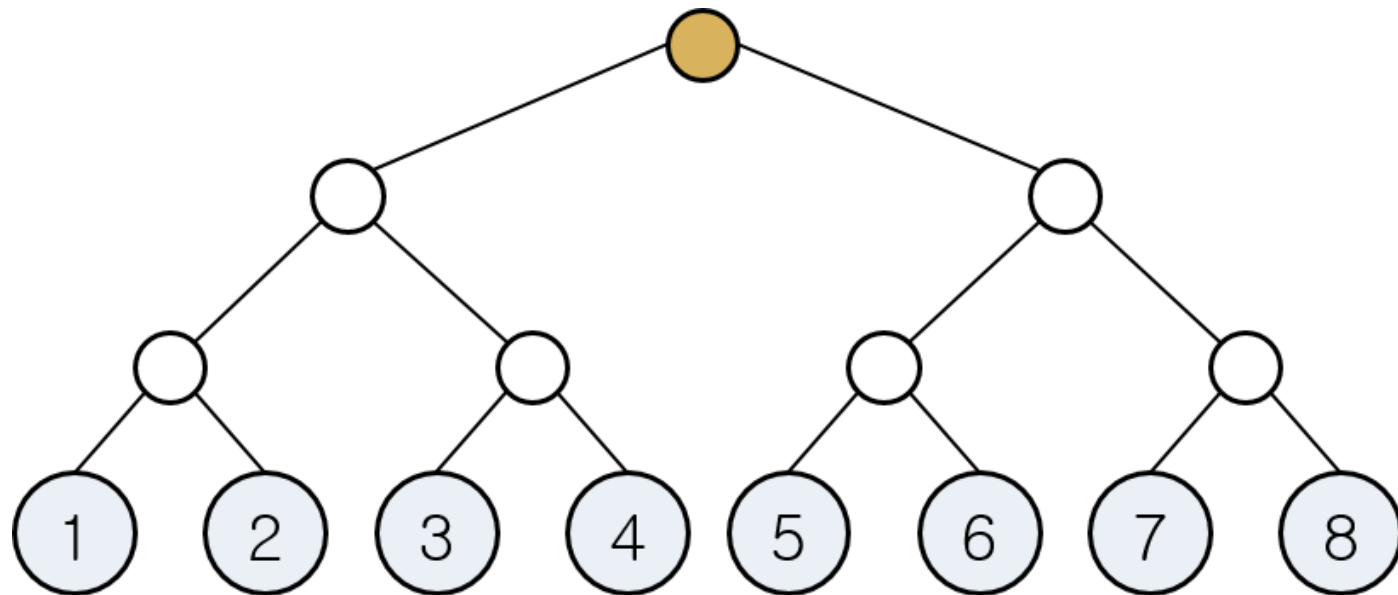
Summing Consecutive Values

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



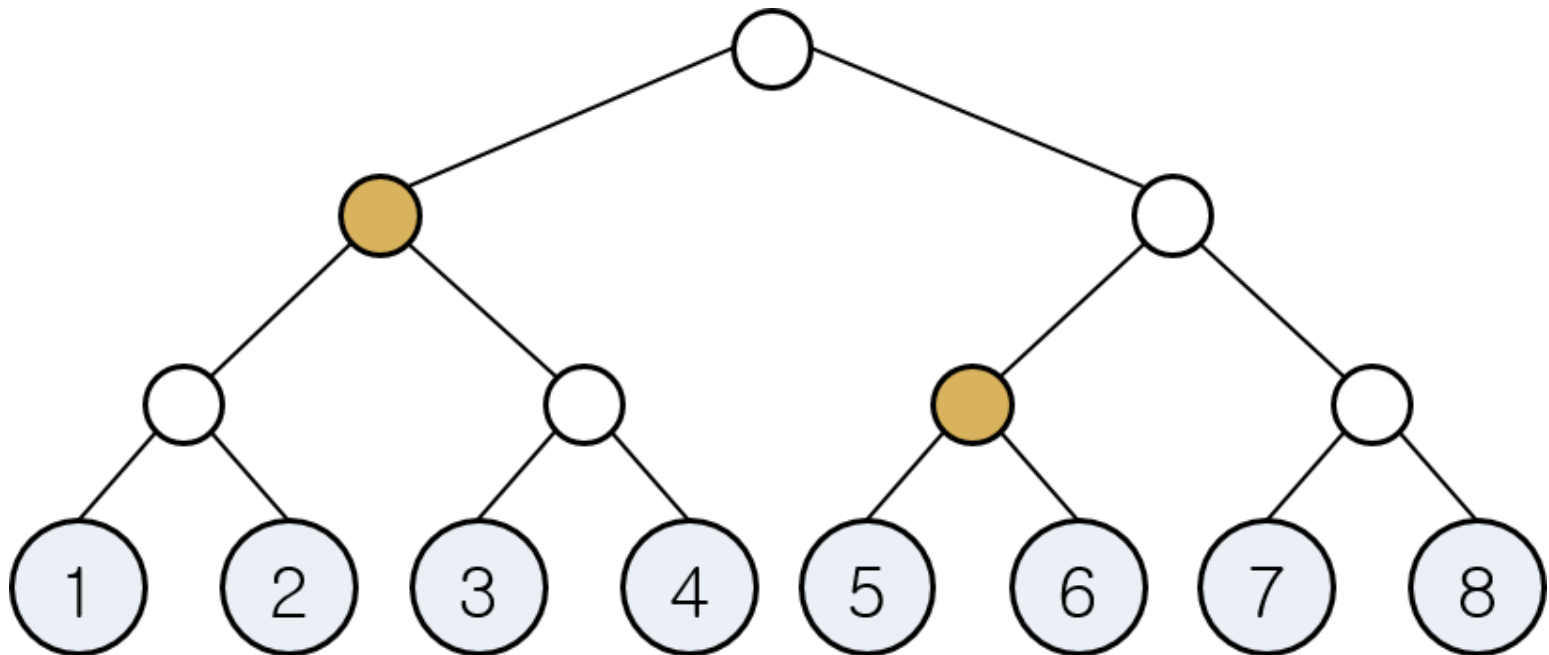
Summing Consecutive Values

- $\text{Sum}(8) =$ sum of the values of gold-colored nodes.



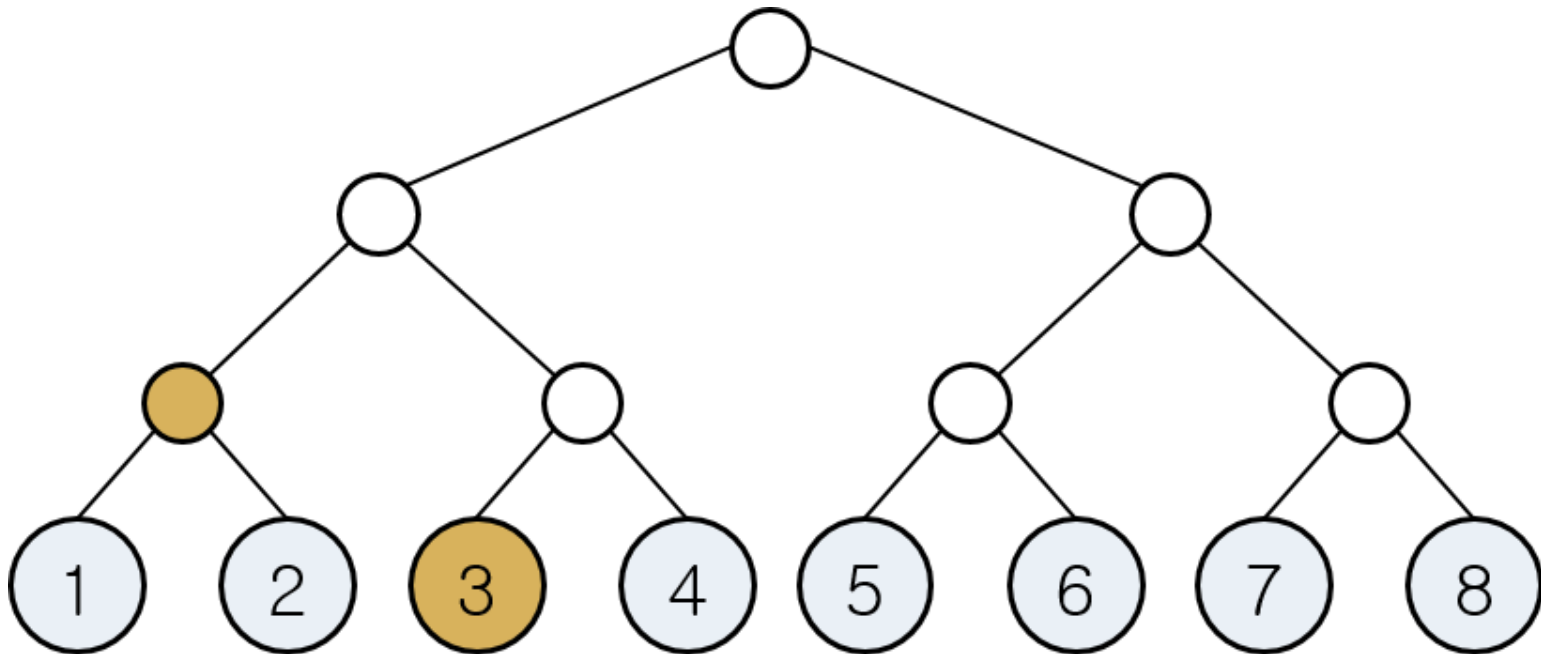
Summing Consecutive Values

- $\text{Sum}(6) =$ sum of the values of gold-colored nodes.



Summing Consecutive Values

- $\text{Sum}(3)$ = sum of the values of gold-colored nodes.



Summing Consecutive Values

- Say we want to compute $\text{Sum}(k)$
 - 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

Updating a Value

- Say we want to do $\text{Set}(k, x)$ (set the value of leaf k as x)
 - 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