Week 1

Computational problem

> Defines a computational task, specifies what the input is and what the output should be.

Algorithm

> A step-by-step recipe to go from input to out put, different from implementation.

Correctness and complexity analysis

> A formal proof that the algorithm solves the problem, analytical bound on the resources it uses.

Efficiency

> An algorithm is efficient if it runs in polynomial time.
>
> On any instance of size n, it performs no more than $p(n)$ steps for some polynomial $p(x) = a_d x^d + \ldots + a_1 x + a_0$.

O-notation

$T(n) = O(f(n))$ if there exists n_0, $c > 0$ such that $T(n) \leq cf(n)$ for all $n > 0$.

Ω-notation

$T(n) = \Omega(f(n))$ if there exists n_0, $c > 0$ such that $T(n) \geq cf(n)$ for all $n > 0$.

Θ-notation

$T(n) = \Omega(f(n))$ if $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$.

- If $T(n) = \Theta(p(n))$ where p is a polynomial of degree d, then doubling the size of the input should roughly increase the running time by a factor of 2^d.

Example of asymptotic growth:

- Constant, $\Omega(1)$
  - Assignments, Comparisons, Boolean operations, Basic mathematical operations, Constant sized combinations of the above.
- Polynomial, $O(n^c)$.
  - Linear, $O(n)$
  - Quadratic, $O(n^2)$
  - Cubic, $O(n^3)$
- Logarithmic, $O(\log n)$
- Quasi-linear, $O(n \log n)$
- Exponential, $O(2^n)$

Properties of asymptotic growth

Transitivity:

- If $f = O(g)$ and $g = O(h)$ then $f = O(h)$.
- If $f = \Omega(g)$ and $g = \Omega(h)$ then $f = \Omega(h)$.
- If $f = \Theta(g)$ and $g = \Theta(h)$ then $f = \Theta(h)$.

Sums of functions:

- If $f = O(h)$ and $g = O(h)$ then $f + g = O(h)$.
- If $f = \Omega(h)$ and $g = \Omega(h)$ then $f + g = \Omega(h)$.

A note on style

For assessments:

- Describe you algorithm.
- Prove its correctness.
- Analyze its time complexity.

Week 2

ADT (Abstract Data Type)

- Type defined in terms of its data items and, associated operations, not its implementation.

In `python`:

- ADT is given as an abstract base class.
- It declares methods, without providing code.
- Can be inherited by a **data structure implementation**, provides code for all the required methods and has a constructor.

Array-based List

Just a static array, nothing to mention.

Positional Lists

Just a linked list, nothing to mention.

Doubly Linked List

nothing to mention

Performance

nothing to mention

Iterators in `Python`

- `iter(obj)` returns an iterator of the object collection.
- implement `__inter__(self)` and return `self`.
- implement `__next__(self)` to define the behavior of iterator, raise `StopIteration` to tell the end of the iteration.

Stack and Queue

- Stacks follow last-in-first-out (LIFO).
- Queues follows first-in-first-out (FIFO).

Double-ended queue

Allows insertions and deletions at both ends

Amortized analysis

A sequence of n operation has $O(f(n))$ amortized time complexity if in the worst-case the total amount of work done by then n operations is no more than $O(nf(n))$

Week 3

Tree

> A tree is an abstract model of a hierarchical structure

- A tree consists of nodes with a parent-child relation.
- A node has at most one parent in a tree.
- A node can have zero, one or more children.

About node

Root: node without parent.

Internal node: node with at least one child.

External/leaf node: node without children.

Ancestors: parent, grandparent, great-grandparent, etc.

Descendant: children, grandchildren, great-grandchildren, etc.

Siblings: two nodes with the same parent.

About Tree

Depth of a node: number of ancestors **not including itself**.

Level: set of nodes **with given depth**.

Height of a tree: maximum depth, **counting from 0**.

Subtree: tree made up of some node and its descendants.

Edge: pair of nodes (u, v) such that one is the parent of the other.

Path: sequence of nodes such that 2 consecutive nodes in the sequence have an edge.

Ordered Tree

> There is a prescribed order for each node's children.

Traversing Tree

- pre-order
- post-order
- in-order (for binary trees)

Binary Trees

> An ordered tree with:
>
> - Each internal node has at most two children.
> - Each child node is labeled as a left child or a right child.
> - Child ordering is left followed by right.
> - Tree is **proper** if every internal node has two children.

- Sometime the method may call itself on all children, the total cost is linear in the number of nodes.
- Sometime the method may call itself on at most one child, the total cost is linear in the height of the tree.

Week 4

Binary Search Tree

> For any node, it's left child is always small than it, and it's right child is always bigger then it.

Complexity

- Space used: $O(n)$.
- Get, put, remove $O(h)$, h is hight, the best one can take $O(\log n)$

B-Trees

- Group tree into chunks of size B and layout each chunk in its own external memory block.
- Number of I/Os equal number of chunks we need to fetch.

For chunks has close to B nodes, so each chunk has height close to $\log_2 B$.

Balanced BST

> BST with height $O(\log n)$ at all times.

Rank-balanced Trees

> Keeping a **"rank"** for every node, where r(v) acts as a proxy measure of the size of the subtree rooted at v.

AVL Tree definition

r(v) is its height of the subtree rooted at v.

The height of an AVL tree storing n keys is at most $O(\log n)$

- Balance constraint: The ranks of the two children of every internal node differ by at most 1.

Insertion

1. If k is in the tree, do nothing.
2. If k is not in the tree, insert it.
    - Some ancestors may have increased their height by 1.
3. Re-arrange tree to re-establish AVL property.

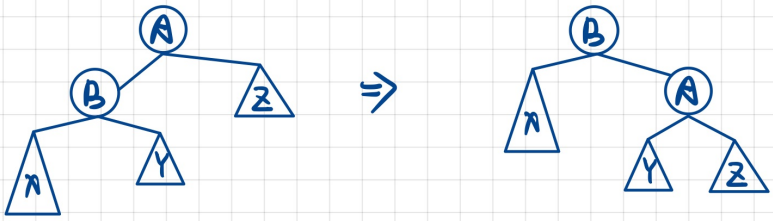
Removal

1. If k is not in the tree, do nothing.
2. If k is in the tree, remove it.
    - Some ancestors may have decreased their height by 1.
3. Re-arrange tree to re-establish AVL property.

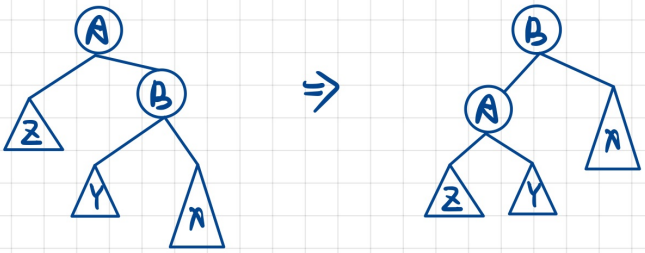
Re-establish

There are only 4 types of unbalance state could appear in AVL tree.

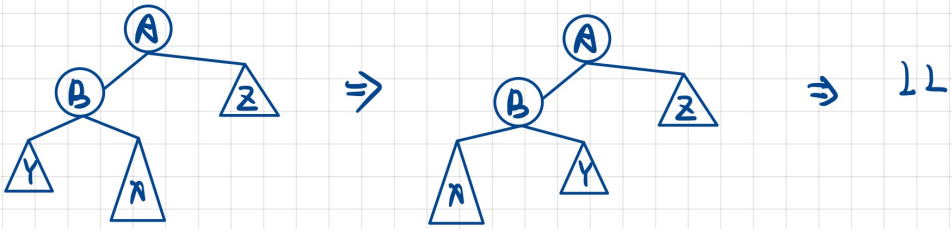
LL: Left child's left child is higher than left child's right child.



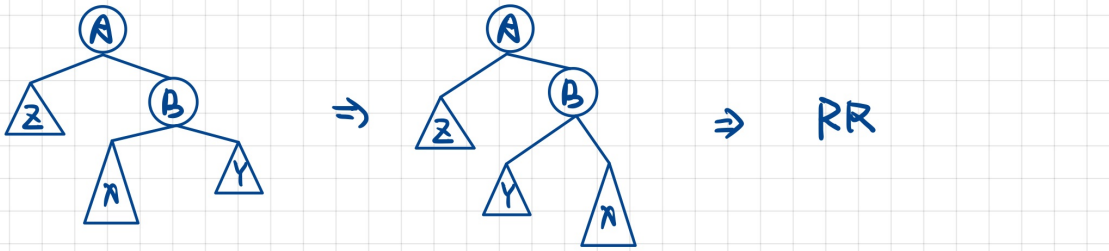
RR: Right child's right child is higher than right child's left child.



LR: Left child's right child is higer than left child's left child



RL: Right child's left child is higher than Right child's right child



Evaluation

- Space: $O(n)$
- Height: $O(\log n)$
- Searching: $O(\log n)$
- Insertion: $O(\log n)$
- Removal: $O(\log n)$

The Map

- List-Based (unsorted) Map
- Tree-Based (sorted) Map

Week 5

Priority Queue

> Special type of ADT map to store a collection of key-value items where we can only remove smallest key.

- Unsorted list implementation

  `insert` in $O(1)$

  `remove_min` and `min` in $O(n)$

- Sorted list implementation

  `insert` in $O(n)$

  `remove_min` and `min` in $O(1)$

Selection-Sort and Insertion-Sort

nothing to mention.

Heap data structure (min-heap)

A heap is a binary tree storing items at its nodes:

- Heap-Order: for every node $m \neq root$, $key(m) \geq key(parent(m))$
- Complete Binary Tree: let h be the height
  - Every level i < h is full.
  - Remaining nodes take leftmost positions of level h.

Fact:

- The root always holds the smallest key in the heap.
- A heap storing n keys has height log n.

Implement:

- Since heap is a complete binary tree, it is better to use an **array** to store it.

Insertion into a Heap

1. Create a new node with given key.
2. Find location for new node.
3. Restore the heap-order property (bottom up).

- $O(\log n)$

Removal

1. Replace the root key with the key of the last node
2. Delete the last one.
3. Restore the heap-order property (Top down).

- $O(\log n)$

Heap sort

Using an array to store heap:

- Parent of i: $(i-1)/2$
- Left child of i: $i*2+1$
- Right child of i: $i*2+2$

Main idea:

1. Construct a heap:
    - Max heap for ascending order
    - Min heap for descending order
2. Exchange the value of the `array[0]` and `array[i]`
    - `i = size - 1` at the beginning.
    - Decrease by 1 each time.
3. Re-establish max heap.

Heap in array

- A heap on n keys can be constructed in $O(n)$ time.
- The n remove_min still take $O(nlogn)$ time

Heapify

Building a priority queue into a max or min heap only take $O(n)$.

For dequeuing, just remove the minimum number in the heap.

Week 6

Hash

- A **hash function h** is used to map keys to corresponding indices in an array A.
    - h is a mathematical function.
    - h is fairly efficient to compute.
- A **hash table** for a given key type K consists of:
    - Hash function h: $K \to [0, N-1]$
    - Array of size N
    - Ideally, item (x, o) is stored at $A[h(x)]$

Hash Functions

A hash function h is usually the composition of two functions:

- Hash code:

  Transform keys to integers.
- Compression function:

  Transform integers to indices.

Example

Common Hash Codes

- view the key k as a tuple of integers $(x_1, x_2, \dots x_d)$ with each being an integer in the range $[0, M-1]$ for some M.
- view the key k as nonnegative integer.

Summing components

Used for keys $k = (x_1, x_2, \dots x_d)$:

- $h(k) = \sum_i x_i$.
- $h(k) = \sum_i x_i \ mod \ p$ where p is a prime.
- $h(k) = \oplus_i x_i mod p$
- $h(k) = x_1 a^{d-1} + x_2 a^{d-2} + \dots + x_{d-1} a + x_d$

Modular division

Used on keys k that are positive integers $h(k) = k \ mod \ N$

Universal hash functions

Let H be a family of hash functions $[0, M] \to [0, N-1]$.

H is **2-universal** if picking h unifordsxmly at random (UAR) from H yields that for any two keys i and j:

$$Pr[h(i) = h(j)] <= 1/N$$

Fact: Let h be a function chosen UAR from a 2-universal family then the expected number of collision for a given key k in a set S of n keys is n/N.

Random Linear Hash Function

Used on keys k that are positive integers:

$$h(k) = ((ak + b) mod \ p) mod \ N$$

For some primer number p, and a and b are chosen UAR from [0, p-1] with $a \neq 0$.

Collision Handling

- Separate chaining
- Linear probing
- Cuckoo hashing

Separate chaining

If two or more element hash into the same location, put they into the list in that location.

Load Factor

Load factor is the ratio of **the number of element** in the hash table and **the size** of the hash table:

$$\alpha = \frac{n}{N}$$

In Separate chaining, The expected time for hash table operations is $O(1 + \alpha)$, but when all the items collide into a single chain, it will become $O(n)$.

- In Java, $\alpha < 0.75$
- In Python, $\alpha < 0.66$

Open addressing using Linear Probing

Open addressing: the colliding item is placed in a different cell of the table.

Linear probing: handles collisions by placing the colliding item in the next available cell.

Updates with Linear Probing

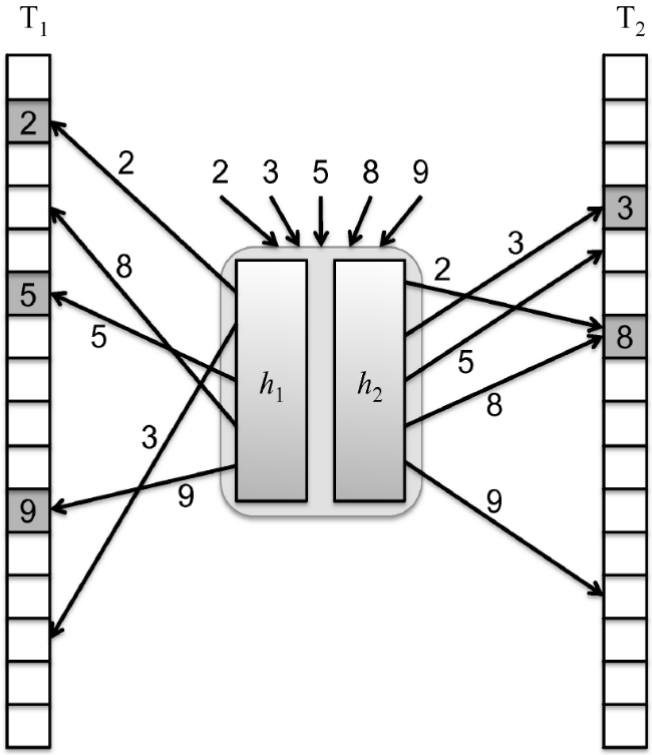
To handle insertions and deletions, use `DEFUNCT` to replaces deleted elements. If target element is deleted directly, `get()` function might be interrupted.

- `get()`: Pass over cells with `DEFUNCT` and keep probing until the element is found or reaching an empty cell.
- `remove()`: If element is found, replace it with the special item `DEFUNCT`.
- `put()`: If element is found, update it. If element is not found, insert element into the first cell we find that has `DEFUNCT` or empty.

Evaluation:

- In the worst case, all operation take $O(n)$ time.
- Expected number of probes for each get and put is $\frac{1}{1-\alpha}$, see more: Bernoulli trial.

Cuckoo hashing



- Use two lookup, T_1 and T_2, each of size N.
- Use two hash functions for two lookup respectively.
- For key k, there are two possible places to store.
- The worst-case expected $O(1)$ for put and remove.

Get

If it's in $T1$ or $T2$, return that.

remove

If it's in $T1$ or $T2$, remove that.

put

If k is already in $T1$ or $T2$, update that.

- If k is not in hash table:
  - If there has a empty cell in $T1$ or $T2$, insert k into that cell.
  - If there has not empty cell, do:

    ```
    kickout = T1[h(k)] or T2[h(k)]
    T1[h(k)] or T2[h(k)] = k
    k = kickout
    ```

    until all elements get a cell.

Eviction cycles

> Repeat a previous eviction.

To detect eviction cycles we can count the time we enter the loop, if we consider it's a eviction cycles, hash table should be expanded.

Performance

- Expected work of n put operations is $O(n)$ provided $N > 2n$
- Worst-case $O(1)$ time for lookups and removals.

Set

> A set is an unordered collection of elements, without duplicates.

Operations:

- `addAll(T)`
- `retainAll(T)`
- `removeAll(T)`

using HashMap those operations can be performed in `O(1)` time.

Multiset

> A set that allows duplicates, counting occurrences of e by operation `count(e)`, implement by Map.

Week 7

Graph

> A graph G is a pair (V, E), where:
>
> - V is a set of nodes, called vertices.
> - E is a collection of pairs of vertices, called edges.

Edge Types

- Directed edge:
  - ordered pair of vertices (u, v)
  - u is the origin/tail.
  - v is the destination/head.
- Undirected edge:
  - unordered pair of vertices (u, v)

Terminology

Undirected graphs

- Edges connect **endpoints**.
- Edges are incident on endpoints.
- **Adjacent** vertices are connected.
- **Degree** is the number of edges on a vertex.
- **Parallel edges** share same endpoints.
- **Self-loop** have only one endpoint.
- **Simple graphs** have no parallel or self-loops.

Directed graphs

- Edges go from **tail** to **head**.
- **Out-degree** is the number of edges out of a vertex.
- **In-degree** is the number of edges into a vertex.
- **Parallel edges** share tail and head.
- **Self-loop** have same head and tail.
- **Simple directed graphs** have no parallel or self-loops, but anti-parallel loops are allowed.

- A **path** is a sequence of vertices, a **simple path** is one where all vertices are distinct.
- A **cycle** is defined by a path that starts and ends at the same vertex, a **simple cycle** is one where all vertices are distinct.
- **Subgraphs**.
- A graph is **connected** if there is a path between every pair of vertices.
- A **connected component** of a graph is a maximal connected subgraph.
- An unrooted **tree** is a graph such that:
  - It is connected.
  - It has no cycles.
- A **forest** is a graph whose connected components are trees.
- **Spanning tree**:
  - A spanning tree is a connected subgraph on the same vertex set.
  - A spanning tree is not unique unless the graph is a tree.

Properties

- $\sum_{vinV} deg(v) = 2m.$
- In a simple undirected graph $m \leq n(n-1)/2.$
- In a simple directed graph $m \leq n(n-1)$

m is the number of edges.

n is the number of vertices.

Implement

Edge List Structure

This structure maintain two sequence:

Vertex sequence:

- Sequence of vertices.
- Vertex objects keeps track of its position.

Edge sequence:

- Sequence of edges.
- Edge objects keeps track of its position.
- Edge object points to two vertices it connects.

Adjacency List

Base on edge list structure but additionally:

- Vertex keeps a sequence of edges incident on it.

Adjacency Matrix Structure

Vertex induces an index from 0 to n-1 for each vertex.

- Reference to edge object for adjacent vertices.
- Null for nonadjacent vertices.

DFS

- $O(n)$

Cut edge:

In a connected graph, an edge is cut edge if this edge is removed then the graph is not connected.

Finding a cut edge in $O(n+m)$ time:

To find a cut edge between (u, v) in a tree :

1. Calculate the level of u.
2. Use DFS to find if there is a route started at v, ending at any vertex that level is higher than or equal to u (for example, 0 is higher than 1).

BFS

For a graph with n vertices and m edges, BFS takes $O(n+m)$, when using adjacency matrix structure, it takes $O(n^2)$.

Topological sort

nothing to mention

Week 8

Weighted Graphs

- Each edge has an associated numerical value.

Greedy algorithms

nothing to mention

Shortest Paths

To find a shortest paths, we use **Dijkstra's Algorithm**, which is similar to UCS.

Dijkstra's Algorithm

Find the smallest path from start vertex to each vertex by BFS.

Using a priority queue, the algorithm spends `O(m)` time on everything except queue operation:

- insert: `n`
- decrease_key: `m`
- remove_min: `n`

Using a heap for priority queue, Dijkstra runs in $O(m \log n)$.

Minimum Spanning Tree

Giving a connected graph, MST is a subset of the edges belongs to a spanning tree whose sum of edge weights is minimized.

- If all edge cost c_e are distinct, only one MST exists.
- Cut property: divide graph into two part, if there exist a min cost edge with exactly one endpoint in one of the subset, MST must contains this edge.
- Cycle property: If there is a cycle in graph, and there is a edge in this cycle with the max cost, then MST must not contain f.

Prim's Algorithm

1. Base on an arbitrary vertex, put it into known list.
2. do until all vertices are known:
    1. Find the min cost edge which the opposite vertex is not in known list.
    2. Add the opposite vertex into known list.

Evaluation:

- While using heap, it takes $O(m \log n)$.
- While using Fibonacci heap, it takes $O(m + n \log n)$.

Kruskal's Algorithm

This algorithm is using a union-find structure which:

> stores a collection of disjoint (non-overlapping) sets.

It has two basic operation:

1. `find(a)` return an id for the set element a belongs to.
2. `union(a, b)`: union the sets elements a and b belong to.

The reason that union-find structure is used:

- There should not be any circle inside a tree.
- In arbitrary subset of union-find structure, if `a` and `b` is already inside, it means there is a route from `a` to `b`, then any others edge from `a` to `b` should not be add into return array

anymore.

Procedure of Algorithm:

1. Sort edges in increasing order base on their cost.
2. Divide each vertex into different subset of union-find structure.
3. For each edge:
    1. If it's two endpoints is already in the same subsets, do nothing.
    2. If it's not, add this edge into the return array and merge two subsets.

Union-find Set

For each node in the set, it structure consisted of:

```
struct Node{
    int data;
    int size;// number of descendant
    int parent;
}
```

For:

1. `find(a)` return an id for the set element a belongs to, compress the route by set current (each time we call `find()`) node's father as the root.
2. `union(a, b)`: chose the node with smaller size, set this node's father as the other node. It takes `O(min(|A|, |B|))` if we always insert smaller set to a bigger set, a sequence of n union operations takes at most $O(n \log n)$

Week 9

Greedy algorithms

> Keep taking as much as possible of best item.

Fractional Knapsack Algorithm

Best item:

- highest benefit.
- smallest weight.
- benefit/weight (best in this case).

This algorithm takes $O(n \log n)$

Interval Partitioning

Check at most how much course is overlapped at each course beginning.

When using priority queue, it takes $O(n \log n)$.

Huffman encoding

Main idea:

- Use a tree structure to store the string, the character with more frequent appearance will be in a higher level node in the tree.
- For encoding, the code of each character is the route from the root it. Go left is 0, go right is 1.

Implement:

- Merge the two nodes with the minimum value to one blank node, which value is the sum of these two nodes, until all nodes are merge together.
- Traverse though the whole tree and assign the Huffman code to each node.

Evaluation:

- It runs in time $O(n + d \log d)$ when n is the size of string, d is the number of distinct characters.
- The minimizes bits needed to encode string is

$$\sum_{c\ in\ C} f(c) * depth_t(c)$$

  - $f(c)$ means the frequency.
  - $depth_t(c)$ means the depth of the char, which is same length as Huffman code.

Week 10

Randomized algorithms

> Randomized algorithms are algorithms where the behavior doesn't depend solely on the input. It also depends (in part) on random choices or the values of a number of random bits.

- For all permutations to be equally likely, we want that every permutation is generated by the same number of possible executions.

Fisher Yates Algorithm

```
def FisherYates(A)
  n = length of array A
  for i in [0:n] do
    j = pick_uniformly_at_random([i:n])
    A[i], A[j] = A[j], A[i]
  return A
```

In this algorithm, number of possible executions is:

$$1 * 2 * 3 * \ldots * n = n!$$

Number of possible permutations is:

$$1 * 2 * 3 * \ldots * n = n!$$

Every execution leads to a different permutation.

Treap

Given a collection $(v_0, p_0), \ldots, (v_{n-1}, p_{n-1})$:

- If we look at the v-value, T is binary search tree.
- If we look at the p-value, T is heap.

Insert

1. Insert the node as binary search tree.
2. If the properties of heap is broken (children's p-value is smaller than parent's), left rotate (see RR in AVL-tree) or right rotate (see LL in AVL-tree) the tree.

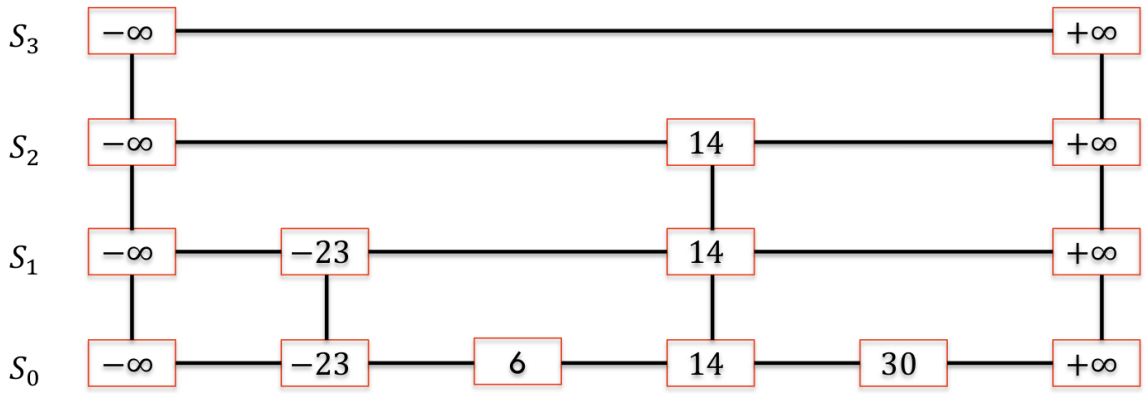
Remove

1. Find the node we want to remove.
2. If one of it's node is empty, replace the node with another existed node.
3. If both two children exist, left rotate or right rotate the node depends on the priority of the children, and go back to step 1.

Skip lists

Skip lists is a group of linked list, the node inside have 5 variables:

- `Before`.
- `After`.
- `Below`.
- `Above`.
- `Value`.



Search

1. Starting at the highest level.
2. If the node after the current node is smaller than target value, set that node as current node. If not, go to the lower level and repeat 2 until it reaches the bottom of the skip lists.
3. Return the last node we find, no matter it is equal to target value or not.

Insert

1. Use search to find a position for value we need to insert.
2. Use a random number to decide the level of the new node.
3. Connect the node into the skip list.

Removal

1. Search the position of the target value, if return node is not equal to target value, stop.
2. Cut off the connection between the skip list and the target node from bottom to top.

Evaluation

- The expected height of a skip list is $O(\log n)$, the probability that an element is present at height i is $\frac{1}{2^i}$.
- The expected search, insertion, deletion time of a skip list is $O(\log n)$.
- The expected space used by a skip list is $O(n)$.

Week 11

Divide and Conquer algorithms

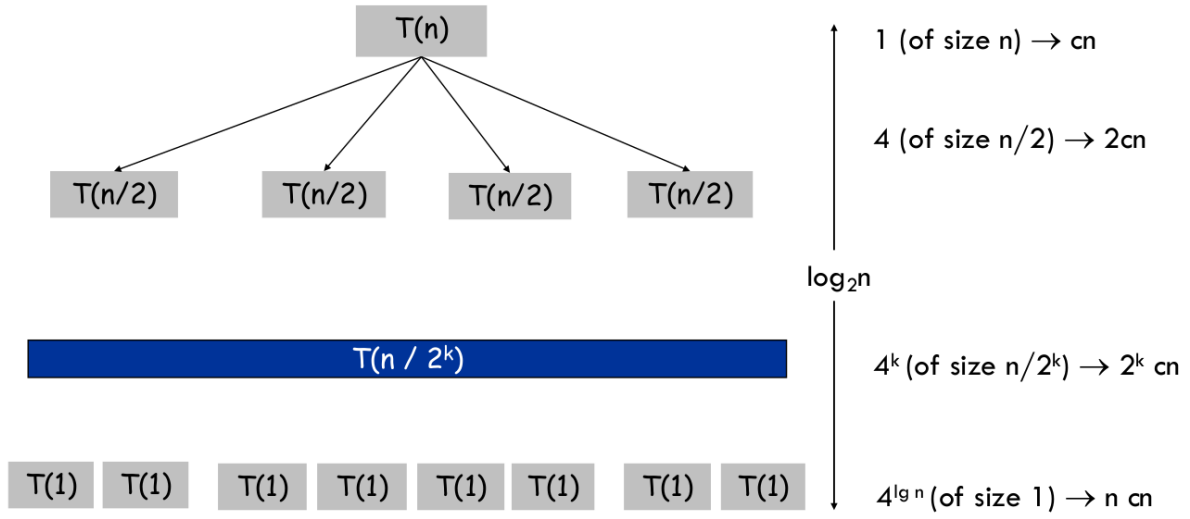
1. Divide: If it is a base case, solve directly, otherwise break up the problem into several parts.
2. Recur: Recursively solve each part.
3. Conquer: Combine the solutions of each part into the overall solution.

To calculate the time complexity of these algorithms $T(n)$, we need to find out:

- Divide step cost.
- Recur step cost in terms of $T(smallervalues)$.
- Conquer step cost.

$$T(n) = \begin{cases} Divide\ step\ +\ Recur\ step\ +\ Conquer\ step & \text{for} \quad n > 1 \\ O(1) & \text{for} \quad n = 1 \end{cases}$$

Proof by unrolling



In `i`th depth, there will be 4^k step, and each of them go though $n/2^k$ data. So when $n/2^k = 1$, we can know how much step at the last depth eventually.

Some recurrence formulas with solutions

Recurrence	Solution
$T(n) = 2T(n/2) + O(n)$	$T(n) = O(n \log n)$
$T(n) = 2T(n/2) + O(\log n)$	$T(n) = O(n)$
$T(n) = 2T(n/2) + O(1)$	$T(n) = O(n)$
$T(n) = T(n/2) + O(n)$	$T(n) = O(n)$
$T(n) = T(n/2) + O(1)$	$T(n) = O(\log n)$
$T(n) = T(n-1) + O(n)$	$T(n) = O(n^2)$
$T(n) = T(n-1) + O(1)$	$T(n) = O(n)$

Quick Sort

1. **Divide** Choose a random element from the list as the pivot (usually the first one is used) Partition the elements into 3 lists:
   - less than the pivot.
   - equal to the pivot.
   - greater than the pivot.
2. **Recur** Recursively sort the less than and greater than lists.
3. **Conquer** Join the sorted 3 lists together.

Quick sort takes $O(n \log n)$.

Maxima-set

1. **Preprocessing** Sort the points by increasing x coordinate.
2. **Divide** sorted array into two halves.
3. **Recur** recursively find the MS of each half.
   - Every points in right MS can be saved.
   - Every points in left MS that have larger y than the largest y in right MS can be saved.
4. **Conquer** compute the MS of the union of Left and Right MS

Evaluation:

$$T(n) = 2T(n/2) + O(n) = O(n \log n)$$

Integer multiplication

Main idea: For $x * y$, we have:

$$x = x_1 * 2^{n/2} + x_0$$
$$y = y_1 * 2^{n/2} + y_0$$
$$x * y = x_1 y_1 * 2^n + (x_1 y_0 + x_0 y_1) * 2^{n/2} + x_0 y_0$$

for $x_1 y_1$, $x_1 y_0 + x_0 y_1$, $x_0 y_0$ we can apply the same algorithm.

1. **Divide step** (produce halves) takes $O(n)$.
2. **Recur step** (solve subproblems) takes $4T(n/2)$.

3. **Conquer step** (add up results) takes $O(n)$.

$x_1y_0 + x_0y_1$ also equal to $(x_1 + x_0)(y_1 + y_0) - x_1x_0 - y_1y_0$.

Quick Sort

Same as quick sort nothing to mention
