Data Structures and Algorithms

Table of Contents

[AVL Trees 2](#_Toc160148444)

[AVL Concept 2](#_Toc160148445)

[Insert 2](#_Toc160148446)

[Complexity 4](#_Toc160148447)

[Delete 4](#_Toc160148448)

[Height – Proofs 5](#_Toc160148449)

[Augmented Data Structures 5](#_Toc160148450)

[Rank 5](#_Toc160148451)

[Select 6](#_Toc160148452)

[Weight-Balanced Trees 7](#_Toc160148453)

[Insert and Rebalance 8](#_Toc160148454)

[Rebalance Cases 8](#_Toc160148455)

[Heap 8](#_Toc160148456)

[Heap insert algorithm: 9](#_Toc160148457)

[Heap extract-max 9](#_Toc160148458)

[Heaps in arrays/vectors 11](#_Toc160148459)

[Proving height of heap 12](#_Toc160148460)

[Depth-First Search 12](#_Toc160148461)

[Concept 12](#_Toc160148462)

[Pseudo code 13](#_Toc160148463)

[Complexity 13](#_Toc160148464)

[Timestamps 14](#_Toc160148465)

[Cycle Detection 15](#_Toc160148466)

[Pseudo-code: 15](#_Toc160148467)

[Directed Graphs 15](#_Toc160148468)

[Cycle Detection 17](#_Toc160148469)

[Strongly Connected Component 17](#_Toc160148470)

[Transpose of a graph 17](#_Toc160148471)

[Computing SCCs 18](#_Toc160148472)

[Pseudocode 18](#_Toc160148473)

[Proof Skeleton 20](#_Toc160148474)

AVL Trees

🡪 When BSTs go wrong

🡪 Depending on the insert order a BST can become so unbalanced that the complexity is as bad a linked list aka O(n)

Solution: Balanced Tree

🡪 They follow a rule such that the tree will always be somewhat balanced and maintain a worst complexity of log(n)

AVL Concept

Maintains the height of log(n)

Balance factor = height(left subtree) – height(right subtree). Single node has balance 0

Height = number of nocdes/edges on a root-to-lead path. Single node has height 1

🡪 Maintains the balance factor between -1 and 1 for all nodes in the tree

A diagram of a tree

Description automatically generated

Insert

A couple of circles with numbers

Description automatically generated

We have multiple nodes that now have a balance factor outside of -1 0 or 1

🡪 To solve this case we make a single clockwise rotation and update the factors

A diagram of numbers and circles

Description automatically generated

62 gets a new parent!

How to know which rotation to do?

*2 main cases with subcases*

1. **Right-heavy tree:**

* The right subtree is left heavy aka height is bigger.
  + Right Left rotation:
* The right subtree is not left heavy
  + Left Rotation:

A diagram of mathematical equations

Description automatically generated

1. **Left-heavy tree:**

* The left subtree is right heavy.
  + Left Right rotation
* The right subtree is not left heavy
  + Right Rotation:

A diagram of a diagram

Description automatically generated

A screenshot of a computer code

Description automatically generated

Example of Double rotation. We inserted 46 and got issues.

A diagram of a tree

Description automatically generated

Complexity

🡪 We only update on the leaf-to-root path (we only look at the nodes we had to go through to insert). If one update is O(1) then insert is

Find position for insert 🡪

Insert node 🡪

Rebalance + update height 🡪

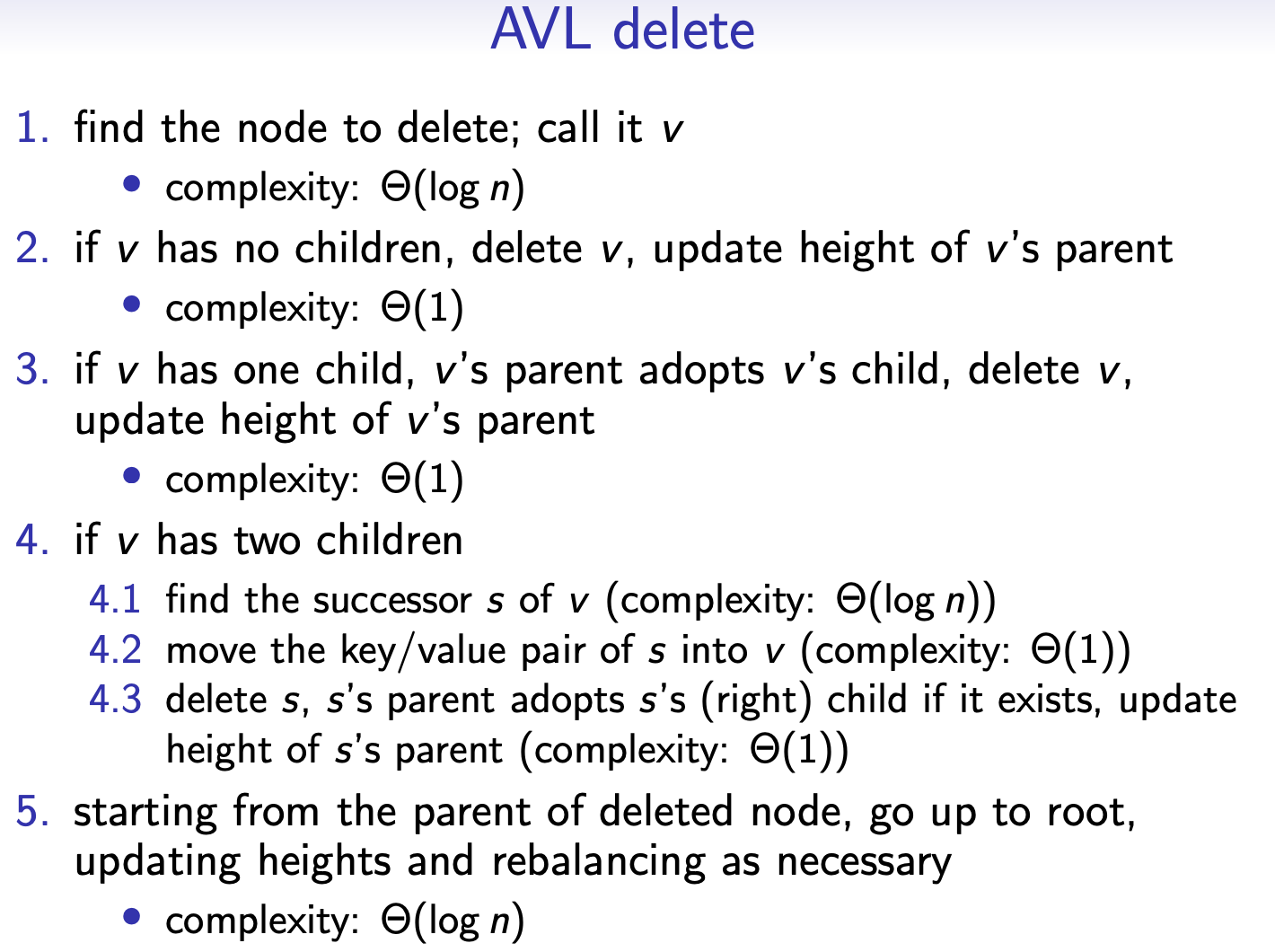
**Total =**

Delete

Idea: Delete Node just like a normal BST then rebalance as needed

Rebalance from the deleted node’s parent then upward

Complexity:



Height – Proofs

🡪 What’s the max possible height for a tree with n nodes?

🡪 What the minimum number n of nodes for a height h ?

Those two questions are equivalent

Let minsize(h) denote the minimum size (number of nodes) of a tree of height (number of nodes of the longest root-to-leaf path) h

Base Case:

Induction Hypothesis:

Augmented Data Structures

🡪 An existing data structure that is modified to store additional information and/or perform additional operation

Rank

Rank(n, n.key) = 1 + size(n.left)

🡪 Given a key compute its rank

🡪 1 + the number of nodes smaller than n

A diagram of a number and a number

Description automatically generated with medium confidence

A white paper with black text and numbers

Description automatically generated

Pseudo Code:

A white screen with black text

Description automatically generated

Select

🡪 Given a rank find the corresponding key

🡪 At each node n compare given rank to size(n.left)+1

A screenshot of a computer program

Description automatically generated

Pseudo Code:

A screenshot of a computer code

Description automatically generated

Union

🡪 Given two AVL trees T1 and T2 make one tree that combines both trees

* If decide which vertice gets saved into the union

🡪 We insert the smallest tree into the biggest tree (compare using num of nodes)

Idea:

1. Split T1 into smaller pieces (aka smaller problems to deal with)
2. Split T2 into smaller pieces (aka smaller problems to deal with)
3. Appy algorithm on smaller pieces
4. Merge Solution

Split:

which are both balanced AVLs (gieven that T2 has more nodes than T1)

* 🡪 All keys in T1 smaller than the root k of T2
* 🡪 All keys in T1 greater than the root k of T2

BOTH are AVL Trees!

A diagram of a triangle

Description automatically generated

Given this tree and the key 16 to split

16<25

🡪 Right Subtree is Join ({17},25,{27,30})

🡪 Left subtree is {12,14}

A white text with black letters

Description automatically generated

Join:

For (L,k,G)

L contains nodes < k anf G contains nodes > k

We could just insert them like that but we need to maintain balance.

**3 cases:**

1. If L much taller than G (height(L) − height(G) > 1)

🡪 insert k and G as subtree into L

A diagram of a child

Description automatically generated with medium confidence

Complexity Θ(n log( m n + 1))

1. If G much taller than L (height(G) − height(L) > 1)

🡪 insert k and L as subtree into G

1. If L and G differ by ≤ 1 (abs(height(L) − height(G)) ≤ 1)

🡪 make a tree with k in root, L as left subtree, and G as right subtree

Examples:

Weight-Balanced Trees

🡪 Another way to keep a BST balanced

Idea: at every node we have 2 possible cases

Where the weight(n)=size(n)+1

Size(n)= the number of nodes in the subtree (counting the root)

Equivalently,:

How should the tree be augmented to hold the size?

🡪 Add a weight node?

Insert and Rebalance

A diagram of a diagram

Description automatically generated with medium confidence

Weight node 51. right = 1 , size = 0

Weight node 51.left = 4 , size = 3

🡪 Need to rebalance

A diagram of a number flow

Description automatically generated with medium confidence

Rebalance Cases

Case 1 🡪 v is right heavy aka weight(v.right) > weight(v.left) × 3

1. AND weight(v.right.left) < weight(v.right.right) × 2

🡪 Single Left/Counter-clockwise rotation

A diagram of a diagram of a diagram

Description automatically generated with medium confidence

1. IF weight(v.right.left) weight(v.right.right) × 2

🡪 Double Rotation Right Left

A diagram of a diagram

Description automatically generated

Case 2 🡪 v is left-heavy aka aka weight(v.left) > weight(v.right) × 3

1. weight(x.right) < weight(x.left) × 2

🡪 Single Right/Clockwise rotation

1. weight(x.right) weight(x.left) × 2

🡪 Double rotation Left Right

Pseudo-Code:

A screenshot of a computer code

Description automatically generated

Union

Heap

🡪 One way to store a priority queue

🡪 A binary tree (aka every node has 2 children max)

For each node n

**🡪** This way we always know that rood not is the biggest node in the tree/subtree

Example:

A diagram of a diagram

Description automatically generated

Now let’s insert 15

A diagram of a network

Description automatically generated

**For node 7**

i**s false** so we need to rearrange the tree

A diagram of a number

Description automatically generated

We swap 15 with the node parent node until 15 no longer bigger than its parent

A diagram of a number

Description automatically generated

Now we’re done

Heap insert algorithm:

A black text on a white background

Description automatically generated

Height is log(n)

Heap extract-max

🡪 Means we have extract the root

🡪 Successor is the leftist right child

🡪 This means we have to rearrange the tree

A diagram of a network

Description automatically generated

We removed 16 and replaced it with 6

Now to fix the rule break we swap 7 with the largest child until the rule is no longer broken

A diagram of a number

Description automatically generated

We swapped 7 with 15 then with 14 and now the rule is no longer broken and we are **done. All of the tree respects the rule**

**Algorithm:**

A screenshot of a computer program

Description automatically generated

Heaps in arrays/vectors

A diagram of a diagram

Description automatically generated

How to use:

* Insert\remove things at the end
* Left child of node at index i is at index
* Right child of node at index i is at index
* Parent of node at index i is at index

Pros:

* Saves space 🡪 No need to store bunch of pointers
* Easy to use 🡪 No weird pointer swapping or things like that just insert and remove things like in a normal array/vector

Cons:

* ?

Proving height of heap

A math equations and numbers

Description automatically generated

Depth-First Search

Concept

1. Visit the start index, A
2. Choose one adjacent, unvisited vertex of the previous and visit it (B, a neighbor of A)
3. Choose one adjacent unvisited vertex of the previous and visit it (C, a neighbour of B)
4. … Repeat
5. When there is no choice left, backtrack to the last time you had a choice and choose another one

When also call it Choose – Explore – Unchoose or Backtracking recursion

A diagram of a breadcrumb

Description automatically generated

DFS can help us find:

* Whether a vertex is reachable from start (aka a path)
* All reachable vertices from start
* With some modifications whether a cycle exists

Pseudo code

A white screen with black text

Description automatically generated

Complexity

|V| the number or nodes |E| the number of edges

1. **Visiting each vertex** -

🡪 Happens only once since we are marking them

1. **Each edge is considered twice**  -

🡪 From origin to neighbour - Choose

🡪 From neighbour back to origin - Unchoose

1. **We find each vertex’s adjancency list once** -

🡪 Right after line 7 in the pseudo code.

🡪 For each node in the adjency list go chose

1. **Check v’s colour deg(v) times** -

🡪 Every time it is in the adjency list of another node

🡪 Degree = the number of edges that point to that node

Total 🡪 =

Assuming marking/checking a vertex’s color and finding it’s adjency list is

Timestamps

🡪 Provide important information about the structure of the graph and are helpful in reasoning about the behavior of the depth-first search

🡪 Each vertex gets two timestamps

1. When v was first discovered
2. When the search finishes examining the adjacency list aka when we blacken it aka when we have visit all the neighbors and all neighbors of neighbors aka when we have bactracked to v

A black background with white text

Description automatically generated

v. 🡪 marks that u is the predecessor of v

time is a global variable in this case 🡪 Why it does not need to be passed

How does it tell us things about the tree’s structure?

3 possible cases:

In any depth-first search of a (directed or undirected) graph G = (V, E), for any two vertices u and v, exactly one of the following three conditions holds:

1. [u.d, u.f] and [v.d, v.f] are entirely disjoint, and neither u nor v is a descendant of the other in the depth-first forest,
2. [u.d, u.f] is contained entirely within the interval [v.d, v.f], and u is a descendant of v in a depth-first tree, or
3. [v.d, v.f] is contained entirely within the interval [u.d, u.f], and v is a descendant of u in a depth-first tree. »

Cycle Detection

During a DFS if something like this happens

A black and white image of a person

Description automatically generated

When u has an edge to a gray vertex that is not its direct predecessor (aka the node we were at right before this one).

Then it must be a cycle ! If we never has this case then this must mean that there is no cycle in the tree

Pseudo-code:

A screenshot of a computer code

Description automatically generated

Directed Graphs

A directed graph G is a pair (V,E) of:

🡪 V — a set of vertices

🡪 E — a set of edges, where an edge is a pair of vertices (usually, we disallow edges from a vertex to itself)

🡪 **Each edge specifies one direction.**

* (a, b) lets you go from a to b, if present.
* (b, a) lets you go from b to a, if present.

Storing a graph: Adjacency lists

A diagram of a diagram

Description automatically generated

From the adjacency list we can tell that c is adjacent to a but a is not adjacent to c

* **degree**: out-degree + in-degree
  + **out-degree**: how many edges go out of a vertex
  + **in-degree**: how many edges go into a vertex
* path, reachable: must comply with edge directions  
  path ⟨v0,...,vk⟩ requires (v0,v1) ∈ E, ..., (vk−1,vk) ∈ E
* cycle: must comply with edge directions  
  cycle ⟨v0,...,vk−1,v0⟩ requires (v0,v1) ∈ E, ..., (vk−1,v0) ∈ E  
  Note: ⟨b, d, b⟩ is a simple cycle this time: (b, d) and (d, b) are two different edges.
* BFS, DFS: no change needed because:

“for each v in u’s adjacency list” already complies with edge direction (u,v)

BFS/DFS Depend on the choice of the start vertex:

* Will visit every vertex if start at: a, b, c
* Will not visit every vertex if start at: d, e

🡪 But for undirected graph all vertex will be visited regardless of where we start

Cycle Detection

A white paper with black text

Description automatically generated

Strongly Connected Component

🡪 Maximal subset of vertices reachable from each other

🡪 For every pair u 🡪 v and v 🡪 u i.e both vertices are reachable from each other

A diagram of a diagram

Description automatically generated

{f,g,h,k} are strongly connected components

🡪 **To find these we need the Transpose of the graph**

Transpose of a graph

🡪 Graph with the same vertices as the original graph except all the edges are reversed

A diagram of a triangle and a triangle with arrows

Description automatically generated

* The SCC are the same

Computing SCCs

🡪 u and v in a tree G are reachable from each other (i.e in the same SCC) iff they are also in the same SCC in

1. DFS on G
   * Visit all vertices
   * Store all finish times
   * Accumulate vertices in reverse finish time order
2. Compute adjency list of
3. DFS on
   * Use the above order to pick start/restart vertices

🡪 Meaning we go in reverse order of finish times

1. Each tree found has the vertices of one strongly connected component

Complexity:

Pseudocode

A white sheet with black text

Description automatically generated

A white sheet with black text

Description automatically generated

Proof Skeleton

A white text with black text

Description automatically generated

A screenshot of a computer code

Description automatically generated

Minimum Spanning Trees

Edge weighted graphs consist of:

* Set of vertices V
* Set of edges E
* Weights a map (usually )
  + If undirected graph (u,v) and (v,u) have the same weight
  + If directed (u,v) and (v,u) may have different weights

Storing a weighted graph

A close-up of a table

Description automatically generated

Minimum Spanning Trees on weighted graphs

Minimum 🡪 Sum of all the weight is the smallest possible

Spanning 🡪 Covers all the vertices (every vertex is an endpoint at least one edge)

Tree 🡪 No cycles

A diagram of a triangle with circles and letters

Description automatically generated

We get: such that this sum is the smallest possible

Usually, MST are used on undirected connected graphs

**How do we compute an MST?**

Kruskal’s Algorithm

A screenshot of a computer program

Description automatically generated

1. Each vertex is it’s own cluster/tree/set
2. Find an edge of minimum weight (the set of edges is sorted so if we go in order, we automatically pick the smallest possible)

🡪 Ignore cycles!

1. Use the edge to merge two clusters/tree/set into one
2. Do it again until all edges have been checked

***How do we know if adding an edge to the cluster will form a cycle??***

**A diagram of a network

Description automatically generated**

In this iteration of the algorithm we see that the edge i-g would cause a cycle.

We know it will because i and g are already in the cluster. Aka there is already a path to access both nodes so we don’t need to add another path. **In short, to avoid cycles check if both nodes are in the same cluster**

Storing Clusters

🡪 Each cluster is a linked list

v. cluster is pointer to v’s owning linked list

u.cluster v.cluster means that we just compare pointers =  **time** ☺

Merging Clusters

Always choose to move the smallest cluster into the bigger one:

🡪 Best case: smallest has 1 one

🡪 Worst case: smallest node has almost as mnay nodes as larger list

Size of cluster roughly doubles

How many times can we double the size of the cluster?

times (how many times can you divide the number of nodes in 2)

Time Complexity

Let n = |V| and m = |E|.

🡪 Collecting and sorting edges: Θ(m log m)

🡪 v.cluster updates: O(log n) per vertex, so O(n log n) total

🡪 the rest is Θ(1) per vertex or edge

Total: O(n log n + m log m) time.

But lets look at n and m:

🡪 maximum number of edges in a (undirected) graph with n vertices: n(n − 1)/2

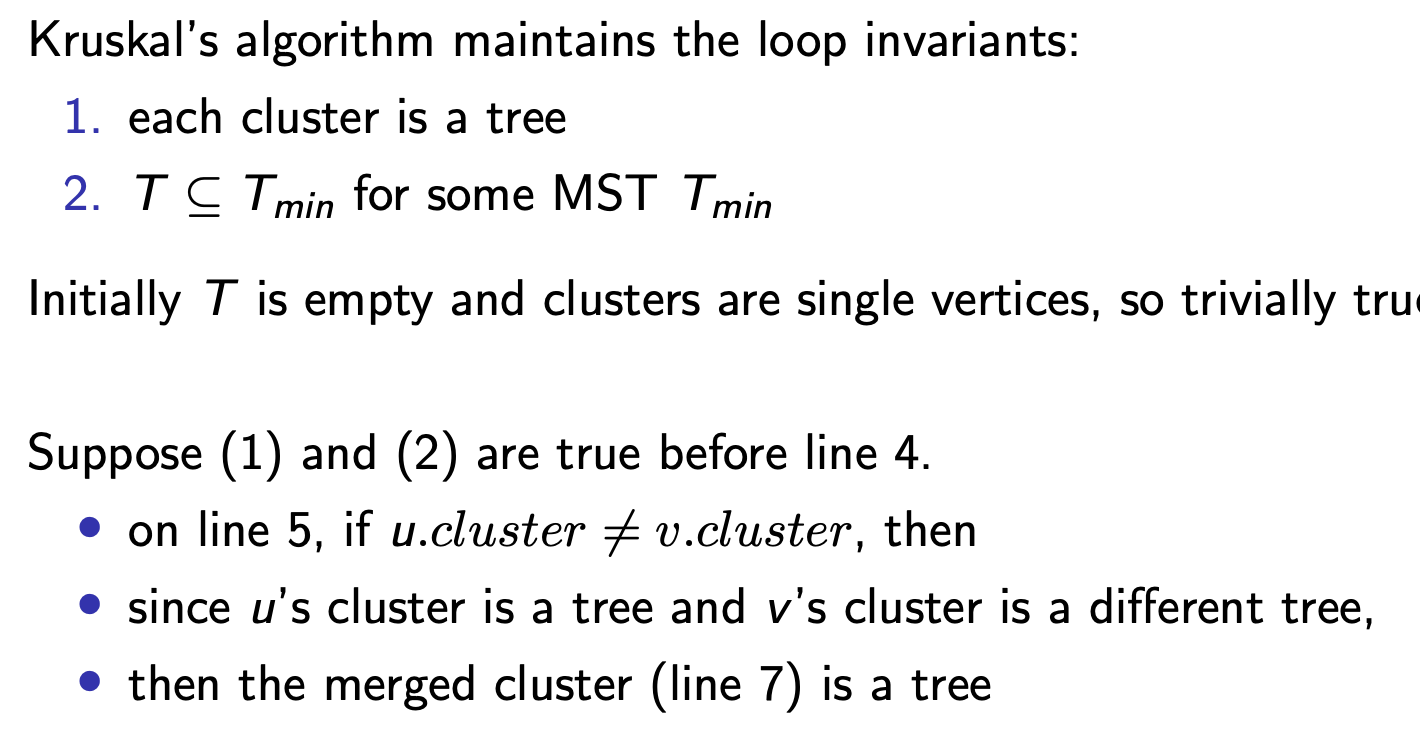
• then m ≤ n(n − 1)/2 ≤

∴ ≤ log() =

∴ log m ∈ O(log n)

Then total time is **O((n + m) log n).**

Correctness



A math equations on a white background

Description automatically generated

Prim’s Algorithm

Also a special BFS search

🡪 We use minimum priority queue

We will need decrease-priority(vertex, new-priority)  
Given