DATE/ PAGE/

WORSE CASE COMPLEXITY

ADTs, DATA STRUCTURE, RUN TIME, COMPLEXITY 06/01/2020

ADT - Abstract Data Type

concept, set of objects with set of operations

what the data is and what can it do

why important?

specification

modularity - independent of implementation => implementation can change without affecting rest of the program

reusability - implemented once, used in many differed programs

Data Structures

implementation of ADT

a way to represent the object and algorithm of operations

how objects are implemented and how operations are performed

Complexity - amount of resources and algorithm uses used

how to quantify?

a function with input as size of

-running time - space (memory) -number of logic gates

-area of chip -messages or bits communicated

DATE/ PAGE/

Running Time $T(n)$ of algorithm with input size n.

The number of primitive operations executed

Worst-case complexity

Let $t(x)$ be # steps algorithm A takes on input x

worst-case time complexity of A of input size n

$$T_{wc}(n) = \max_{|x|=n} \{t(x)\}$$

i.e. considers the input x of size n that causes the algorithm to take the most steps.

BIG O, BIG Ω AND BIG Θ 09/01/2020

Big O

Let g be a function. $O(g)$ is the set of functions $f \in \mathcal{F}$ s.t.

$$\exists C \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \forall n \in \mathbb{N}, n \geq n_0 \rightarrow f(n) \leq C g(n)$$

$f \in O(g)$ if $f(n) \leq C g(n)$ after n exceeds n_0 i.e. $f(n)$ has an upper bound.

Big Ω (omega)

Let $g \in \mathcal{F}$, $\Omega(g)$ is the set of functions $f \in \mathcal{F}$ s.t.

$$\exists b \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \forall n \in \mathbb{N}, n \geq n_0 \rightarrow f(n) \geq b g(n) \geq 0$$

$f \in \Omega(g)$ if $f(n) \geq b g(n)$ with large enough n

i.e. $f(n)$ has a lower bound.

Big Θ (theta)

Let $g \in \mathcal{F}$, $\Theta(g)$ is the set of functions $f \in \mathcal{F}$ s.t.

DATE/ PAGE/

$f(n) \in O(g) \wedge \Omega(g) \Leftrightarrow \exists b \in \mathbb{R}^+, \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \forall n \in \mathbb{N}, n \geq n_0 \rightarrow bg(n) \leq f(n) \leq cg(n)$

Summary:

$O(g(n))$: The set of functions that grow no faster than $g(n)$ when n is sufficiently large

$\Omega(g(n))$: The set of functions that grow at least as fast as $g(n)$ when n is sufficiently large.

$\Theta(g(n))$: The set of functions that grow at the same rate as $g(n)$ when n is sufficiently large

Using limits to prove Big O

Assume $\exists n_0, \forall n \geq n_0, f(n) \geq 0$ ~~and~~ and $g(n) \geq 0$.

$\lim_{n \to \infty} \frac{f(n)}{g(n)}$ exist and is finite $\rightarrow f(n) \in O(g(n))$

↳ $\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty \rightarrow f(n) \notin O(g(n))$

limit does not help when the function is disjoint

BALANCED TREES

RED BLACK TREES

13/01/2020

Like a BST but properties allow it to always have height $O(\log n)$

DATE/ PAGE/

PROPERTIES

Root:

The root and NULL leaves are black

Red:

The children are black i.e. no consecutive red nodes

Black:

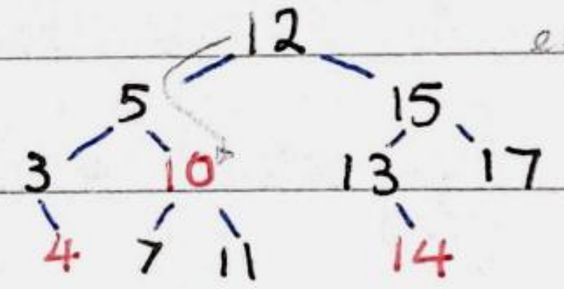
The children can be either colour

For each node with at most one child, # of black nodes along path between the node's NULL child and root is the same

↳ ensures tree is balanced and has height $O(\log n)$

OPERATIONS

Search: same way as BST

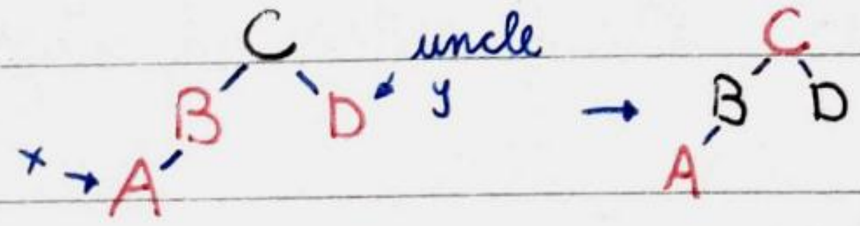


eg 10

Insert

1. Use search to locate position to insert
2. Insert as a red node and fix the tree accordingly:

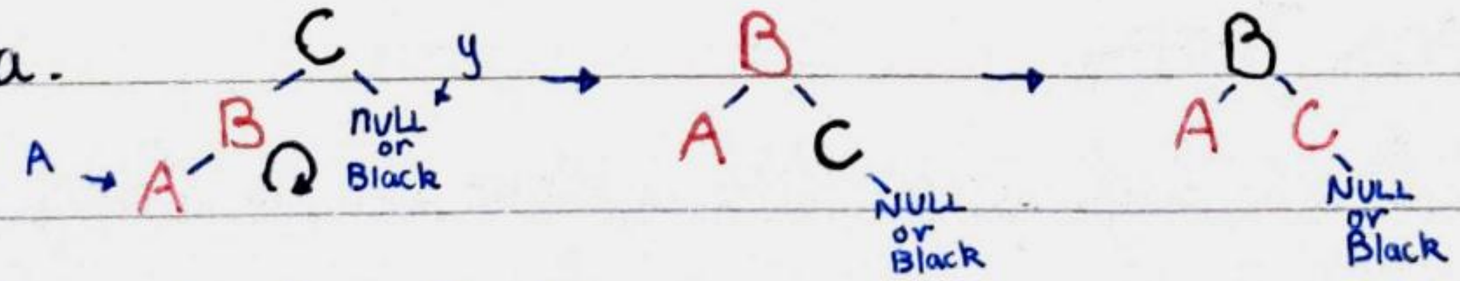
Case 1:



recolour as same fashion if C has same problem

double red and uncle red

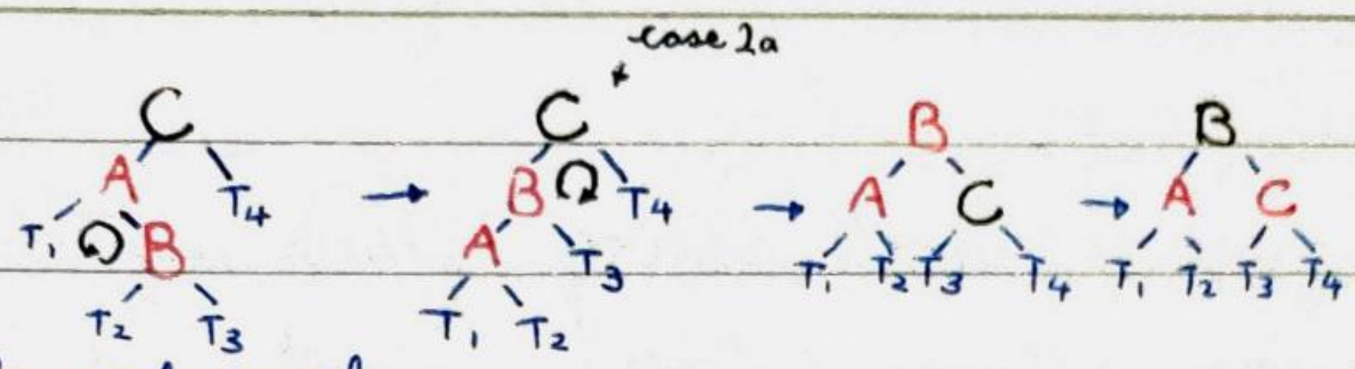
Case 2a.



double red straight line

DATE/ PAGE/

case 2b:



double red bend

case 3 : empty tree → i.e. newly inserted ~~red~~ root, change to black

root becomes ~~red~~ due to prior cases → change to black

↳ only case "height" of tree changes, works because affects all nodes

DELETE IN RED-BLACK TREES 16/01/2020

Delete:

1. Use search to locate the node to be deleted

Red nodes

leaf just delete, no problem

non-leaf

copy the values in successor and delete successor.

if successor is to black, recolour / other cases

Black nodes

with two internal children

replace with successor and delete successor

with one internal child

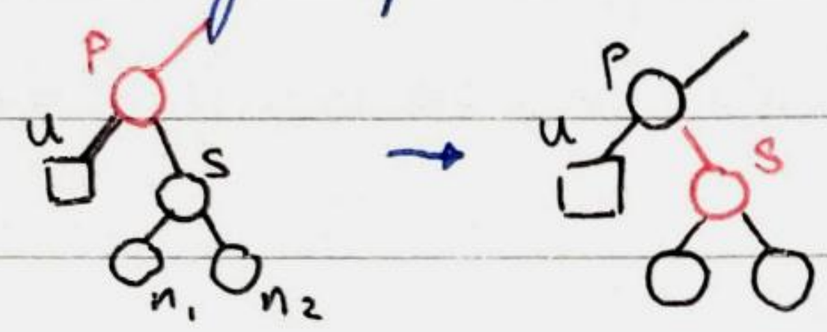
child must be red, ~~same case~~ replace with child and delete child

leaf

red parent black sibling black nephews

swap colours between parent and sibling

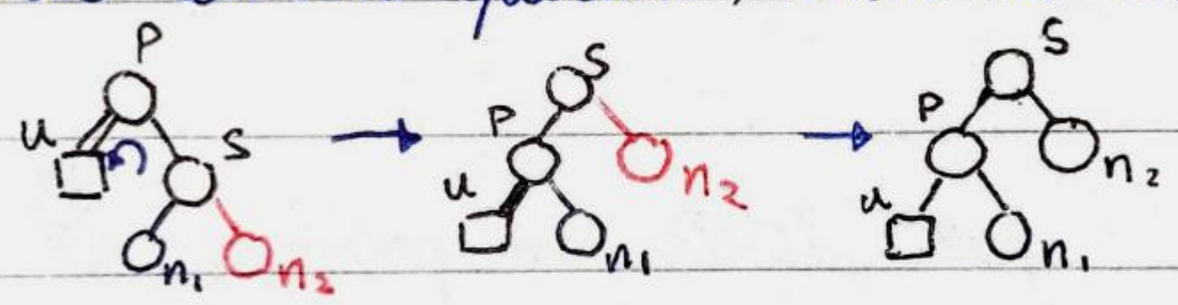
(pushing up black



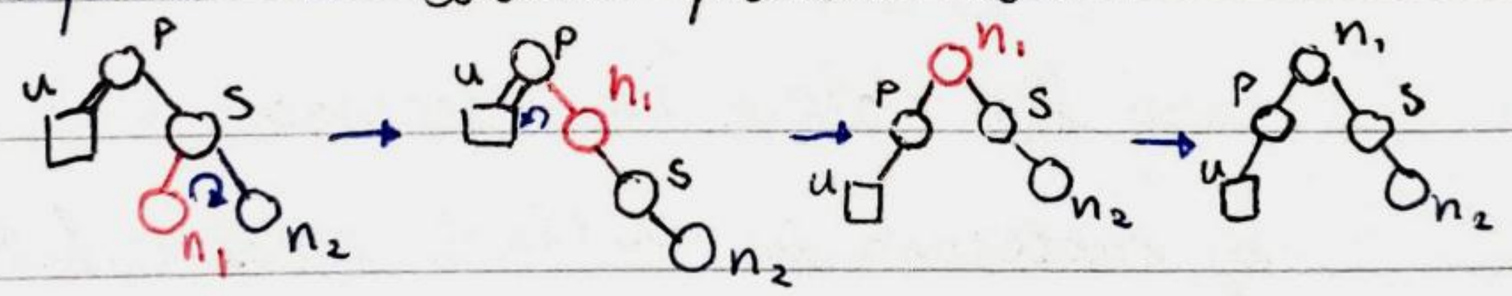
black parent black sibling red nephews

red nephew to black parent straight:

rotate about parent, make rotated nephew black



red nephew to black parent bend

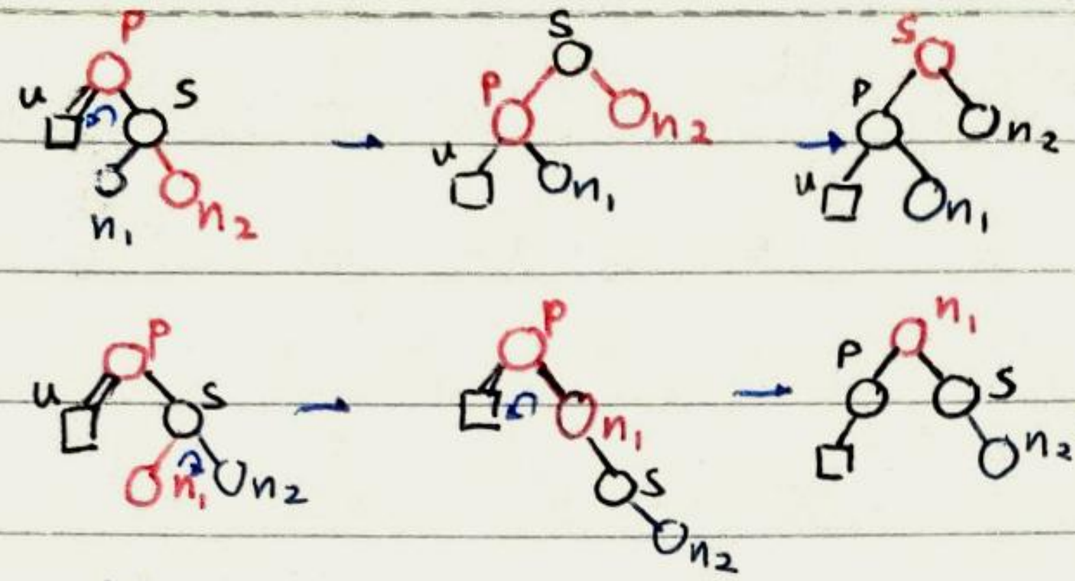


rotate about sibling, then opposite direction about parent, make rotated nephew black

RED-BLACK TREES CONT. 20/01/2020

red parent black sibling red nephews

same as above but root of the subtree stays red

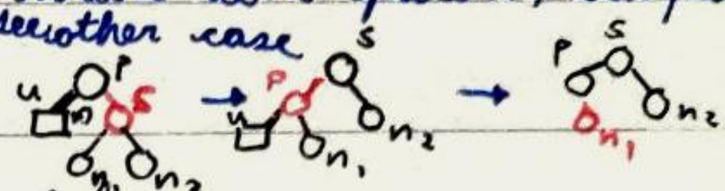


black parent red sibling

black nephews

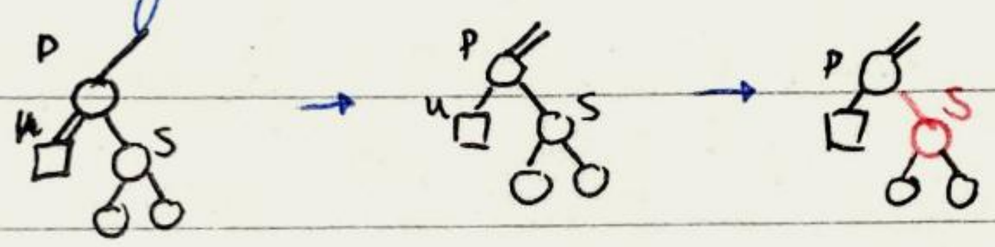
rotate about parent, old sibling b, old parent R

see other case



black parent black sibling black nephews

push the double black up the tree, make sibling red, see other cases



Tree Height

black height = number of black nodes on the path from the root to a NIL leaf (NIL node doesn't count)

height = longest path from root to NIL node

* A red-black tree with n-internal nodes has height h such that

$$h \leq 2 \log(n+1)$$

AUGMENTED TREES AND INTERVAL TREES 23/01/2020

~~ac~~ Augmented Data Structure

An existing data structure modified to store more info

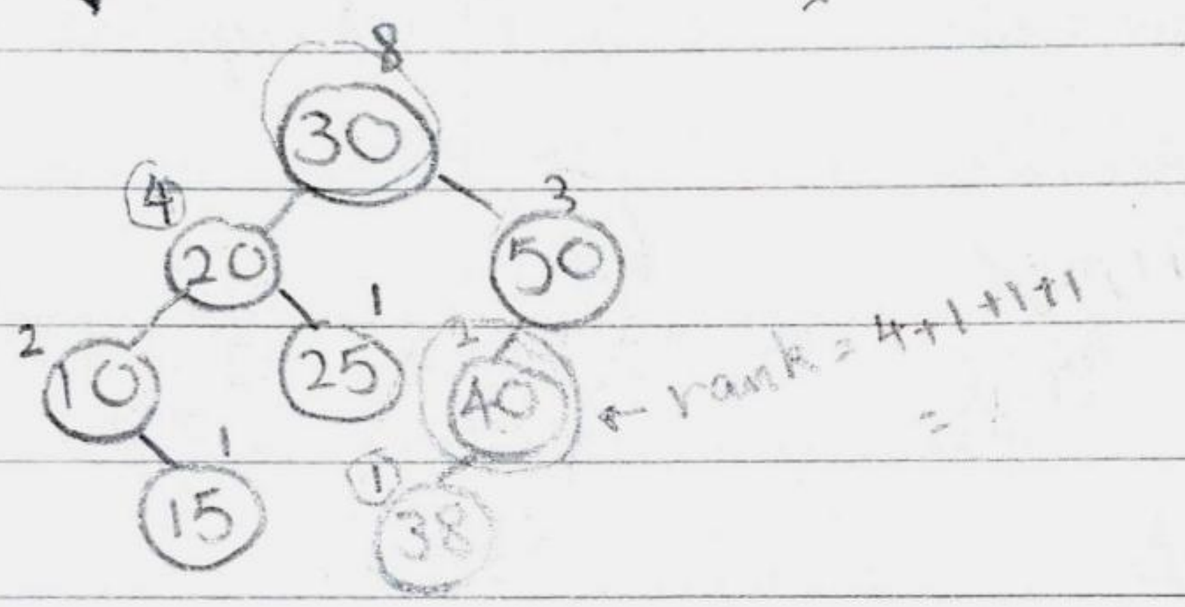
e.g. has, insert, delete, search, rank(k), select(r)

rank(k): return "position" of key

select(r): return key of "position"

DATE/ PAGE/

	delete / insert / search	select . rank
Normal RB Tree	O(log n)	O(N)
RB Tree with rank field	O(N)	O(log n)
" " " size field (# of keys in subtree + itself)	O(log n)	O(log n)



INTERVAL TREES

has closed intervals $\{x \in \mathbb{R} \mid l \leq x \leq h\} = [l, h]$

represented with l and h.

insert (l, h) : Store $[l, h]$, delete $[l, h]$, search (l, h) ↓ return a stored OVERLAPS interval that with $[l, h]$

Comparing:

$l < l' \Rightarrow [l, h] < [l', h']$

$l = l'$ * and $h < h'$, $\Rightarrow [l, h] < [l', h']$

Problem: search ~~tool~~ compares l first so if root and right subtree does not overlap with $[l, h]$

but some in left subtree does, ~~search~~ search can't find it

Solution: add max hi for each node that tells us the highest value of hi in the left subtree

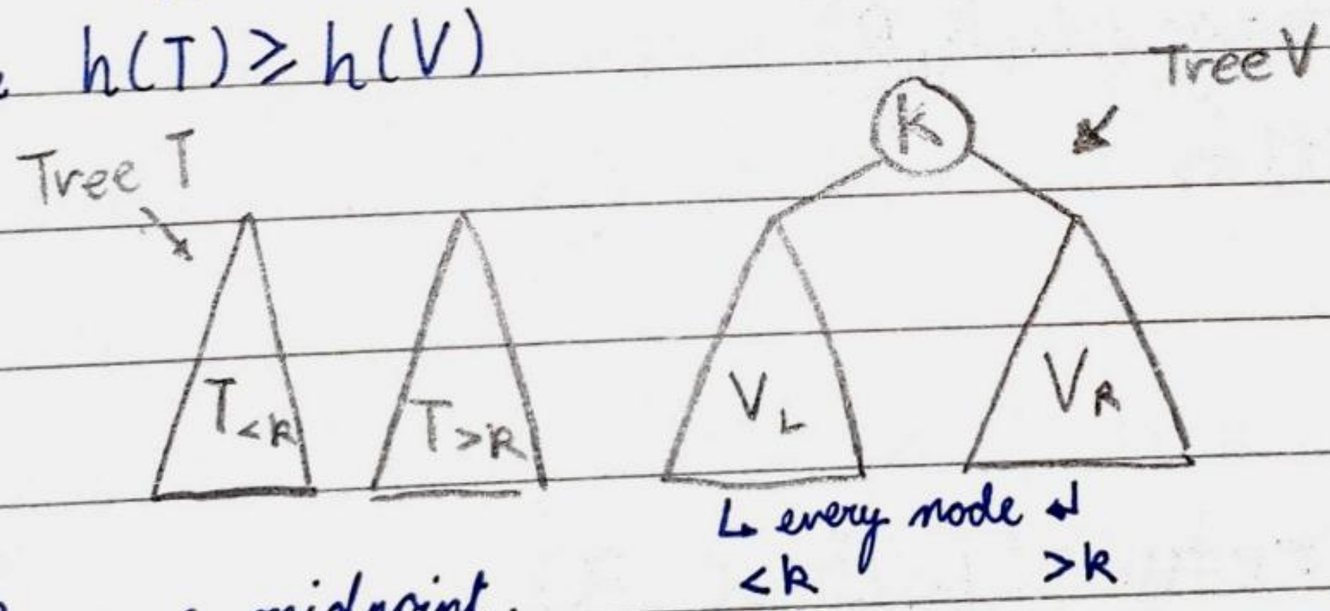
UNION OF BALANCED TREES 27/01/2020

$O(m \log(\frac{n}{m}+1))$ if $n \approx m$ ~~then~~ then is just $\Theta(m)$

A(T, V)

└ union / difference / intersection of tree

assume $h(T) \geq h(V)$



using k as a midpoint,

split T into $T_{<k}$ and $T_{>k}$, V into V_L, k and V_R

compute $L \leftarrow A(T_{<k}, V_L)$ and $R \leftarrow A(T_{>k}, V_R)$

Split

until you hit k / NULL, function calls recursively to split tree into pieces that <k and >k, then merges the pieces as a tree <k and a tree >k, also returns bool has-k

DATE/ PAGE/

Union Algorithm

Given T and V, return a balanced tree with all the keys in T and V

```
union (T, V)
  k = root(V)
  (T<k, has_k, T>k) = split (T, k)   # has_k: bool for k ∈ T
  V_L = k.left
  V_R = k.right
  L = union (T<k, V_L)
  R = union (T>k, V_R)
  return merge (L, k, R)
```

Split Algorithm

```
split (T, k)
  r = root(T)
  if r==k: return (L, True, R)
  elif r==NULL: return (NULL, False, NULL)
  elif r<k:
    go along right, include r and r's left child L to T<k.
    (T<k, b, T>k) = split (R, k)
    return (merge (L, r, T<k), b, T>k)
  else: r>k
```

go along left path, include r and r's right child R to $T_{>k}$

$(T_{<k}, b, T_{>k}) = split(L, k)$

return $(T_{<k}, b, merge(T_{>k}, r, R))$

PIORITY QUEUES

PIORITY QUEUES, HEAP, HEAP SORT

30/01/2020

Queue → first in, first out.

With priority, item of highest priority exits first

insert (p, j): j = job, p = priority max(): read job of max piority

extract-max(): read and remove job of max piority

increase-priority (j, p'): increase j's priority to p'

Data Structure: Heap (ADT)

- binary tree
- "nearly complete" every level; has 2^i nodes, bottom not necessary (nodes all ~~on~~ "aligned" to left)
- priority ≥ child's piority

INSERT

1. put ~~to~~ new node at bottom ~~leve~~ leftmost NULL position
2. if parent priority < new piority, swap

Complexity $O(height) \rightarrow O(\log n)$

DATE/ PAGE/

Extract Max

~~Replace root with blank (take out info required)~~

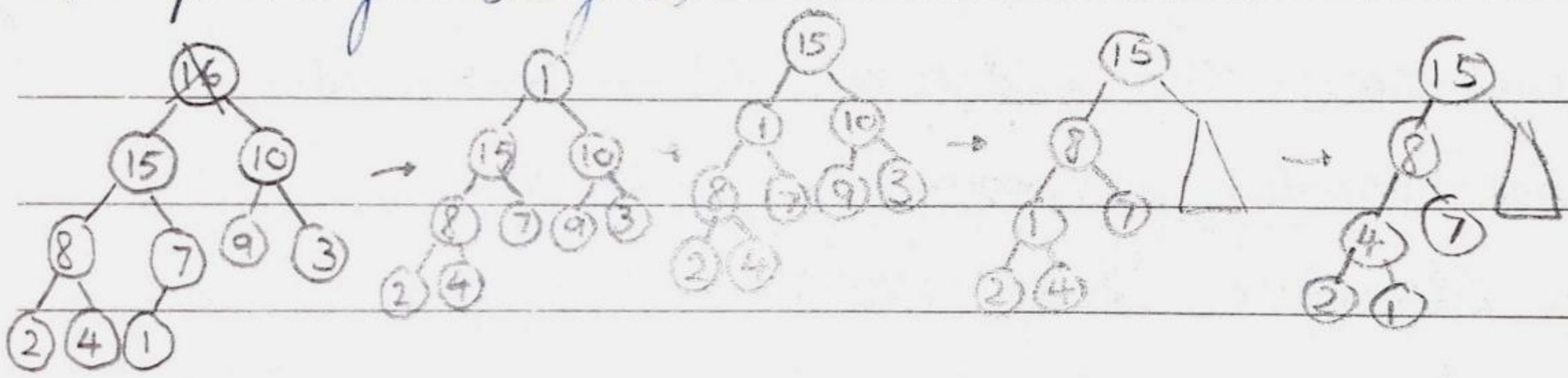
~~swap with to~~

Swap with bottom rightmost item. Heapify.

Heapify

1. Replace root by bottom level, rightmost item
2. let *v (vee) = root, while v has larger child, swap with largest child and keep v as child node.

complexity $O(\log n)$



if n = # nodes, h = height of heap, $2^{h-1} \leq n \leq 2^h - 1$

Implementation of heap

Array of size > # nodes in heap

1. A[0] reserved (for heap size)
2. Highest priority at A[1]
3. Left child of i at $2i$, Right child at $2i+1$

4Parent at $\lfloor \frac{i}{2} \rfloor$

Complexity of Building a Heap

$\sum_{h=2}^{\text{height} \to 2\lfloor \lg n \rfloor + 1}$ (number of trees of height h) $\times (h-1)$ ← the complexity of performing heapify on each tree

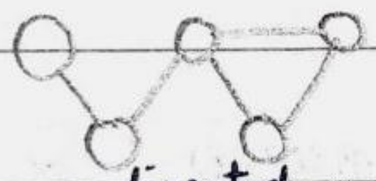
converges to $\frac{n}{2^n}$

$$\leq n \sum_{h=1}^{\infty} \frac{h-1}{2^h} = n \cdot \text{constant}$$

Heap Sort

- convert array to heap - $O(n)$
- repeat extract-max and swap max with item at heap-size position, decrement heap-size - $O(\log n)$
- complexity $O(n \log n)$

GRAPH



Graph $G = (V, E)$ - set of vertices/nodes V and set of edges E

nodes = $|V| = n$ # edges = $|E| = m$

Undirected

edge is set of two vertices $\{u, v\}$

no self loop NOTATION: (u, v)

Directed

edge is ordered pair (u, v) (v, u)

has self loops (u, u)

DATE/ PAGE/

ADJACENT - an edge exist between two nodes

Adjacency Matrix

2D-array

space : $O(n^2)$

edges query : $O(n)$ edge query: $O(1)$

Adjacency Lists

vertices in 1D-array / dictionary

for every vertex, use a list / set for edges

at A[i], stores neighbours of v_i

Space : $O(n+m)$

edges connected to v

edges query, edge query : $O(deg(V))$

Traversal - visit each vertex and perform operations

Path - (distinct) edges from node A to node B

Reachable - v is reachable from u iff a path from u to v exists

Cycle - nonempty sequence (consecutive nodes adjacent, and all edges)

first vertex = last vertex, other vertices distinct

Tree - connected graph with no cycles

Weight graph - w(e) weight of edge of $\forall e \in E$

Connected every two vertices have a path between them

Strongly Connected directed, any 2 vertices u, v there is a

directed path from u to v

BFS AND DFS 03/02/2020

Breadth - First Search BFS(vertex v)

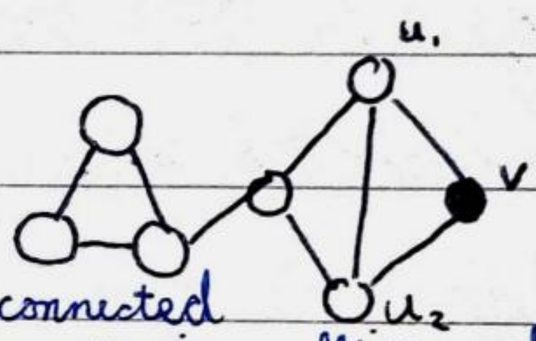
start at v. Visit v and mark as visited

visit every neighbour that is not marked and mark them as visited

Mark v as finished

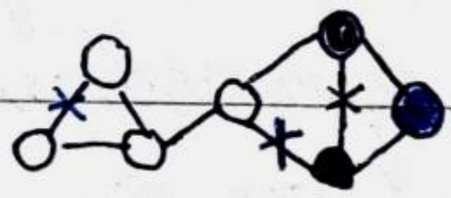
Recurse on each vertex marked as visited in the order they were visited

- shortest path
- whether graph connected → not connected



require calling multiple times

- number of connected components
- produces spanning tree



x = not used

complexity

O(n+m)

adjacency list of nodes examined at most once

Depth - First Search DFS(G, v)

all vertices and edges unmarked

start at v, visit an unvisited vertex and marks vertex visited, edge DFS edge

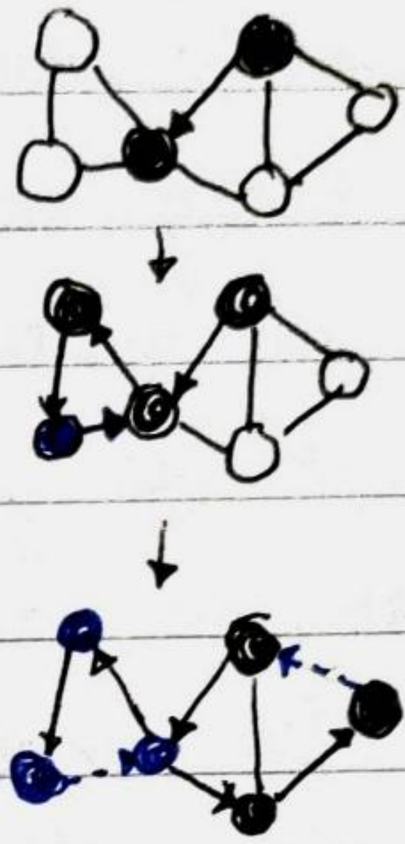
if visited DFS edge → back-edge

current vertex only has visited neighbours → mark finished
back track to first non finished vertex and
start over

- gives a correct order
- finds cycles

Complexity

$O(n+m)$, like BFS



$u \to v$

DFS EDGES (u, v)

tree-edge in DFS tree

v was unvisited ~~((u,~~

back-edge

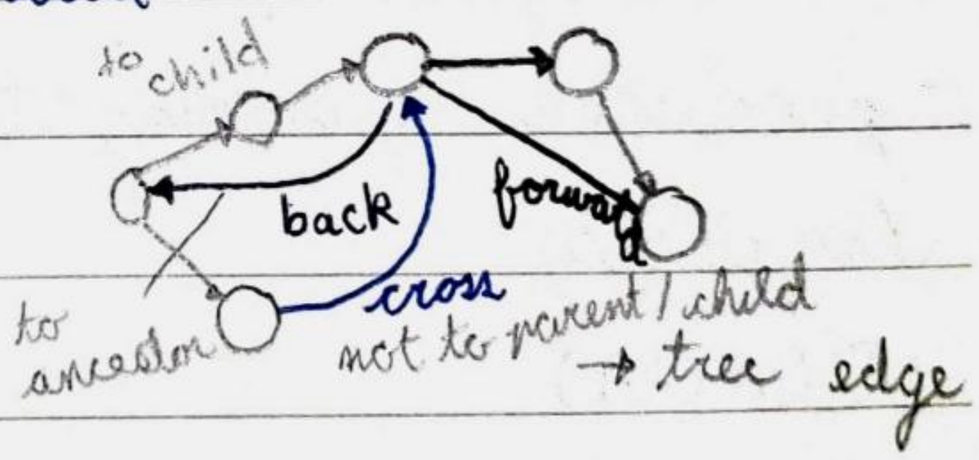
v has ~~already~~ already been visited

(directed) forward-edge

$u \to v$ in DFS tree

(directed) cross-edge

edges that are not in DFS tree



There is a cycle ⇔ there is a ~~b~~ back edge in DFS

DATE/ PAGE/

Minimum Cost Spanning Trees

06/02/2020

Weighted Graphs

set of vertices V

set of edges E

weights : map from edges to numbers $W : E \to \mathbb{R}$

undirected graphs have the same weight, directed graphs (may) have different weights

How are weights stored?

adjacency matrix (∞ = no edge, # number = weight of edge)

adjacency list (stores neighbour vertex as well as weight)

~~Mon~~ Minimum Cost Spanning Tree (MST)

Spanning tree A such that the sum of weights is ~~lower than all other~~ lowest of all possible spanning trees B

$$W(A) = \sum_{e \in A} w(e) \leq w(B)$$

Kruskal's Algorithm

Pick least weight edge that doesn't induce a cycle

Finds an MST

- sort edges with min-priority queue
- use linked lists to represent joined vertices (for now)

Complexity: $O((n+m)\lg n)$ for using linked list

- $n \lg n$: cluster update
- $m \lg m \in O(m \lg n)$: building PQ and removing edge

DATE/ PAGE/

Prims Algorithm

~~Foss~~ Start with a minimum tree / set with one vertex

Add a least weight edge that "grows" tree without creating a ~~cy~~ cycle

Find an MST using similar idea as BST

queue is ~~as~~ min priority queue with priority as smallest edge weight between vertex v and tree T so far (∞ if not ~~connected~~ able to connect yet)

Complexity $O((n+m)\log n)$

$n \log n$ vertex leaves and enters min heap once

$m \log n$ each edge triggers update in priority

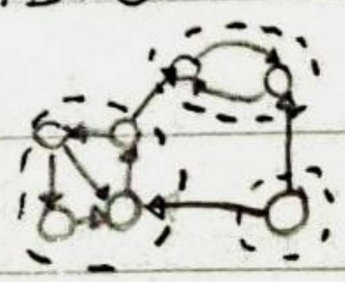
~~DIJKSTRA'S~~ DIJKSTRA'S ALGORITHM 10/02/2020

Finds a path from A to B with least weight

1. have start vertex s added to set of reached vertices S and give distance $d[s] = 0$
2. Use a priority queue to keep track of reachable vertex and total weight from s to those vertices
3. Update queue if ~~as~~ new neighbours get a better priority

STRONGLY CONNECTED COMPONENTS 13/02/2020

maximum subset of vertices reachable from eachother in a directed graph



DATE/ PAGE/

Transpose of G (G^T)

has same vertices as G but direction of edges reversed

$(u, v) \rightarrow (v, u)$ ✡ G^T, G has same strongly connected components

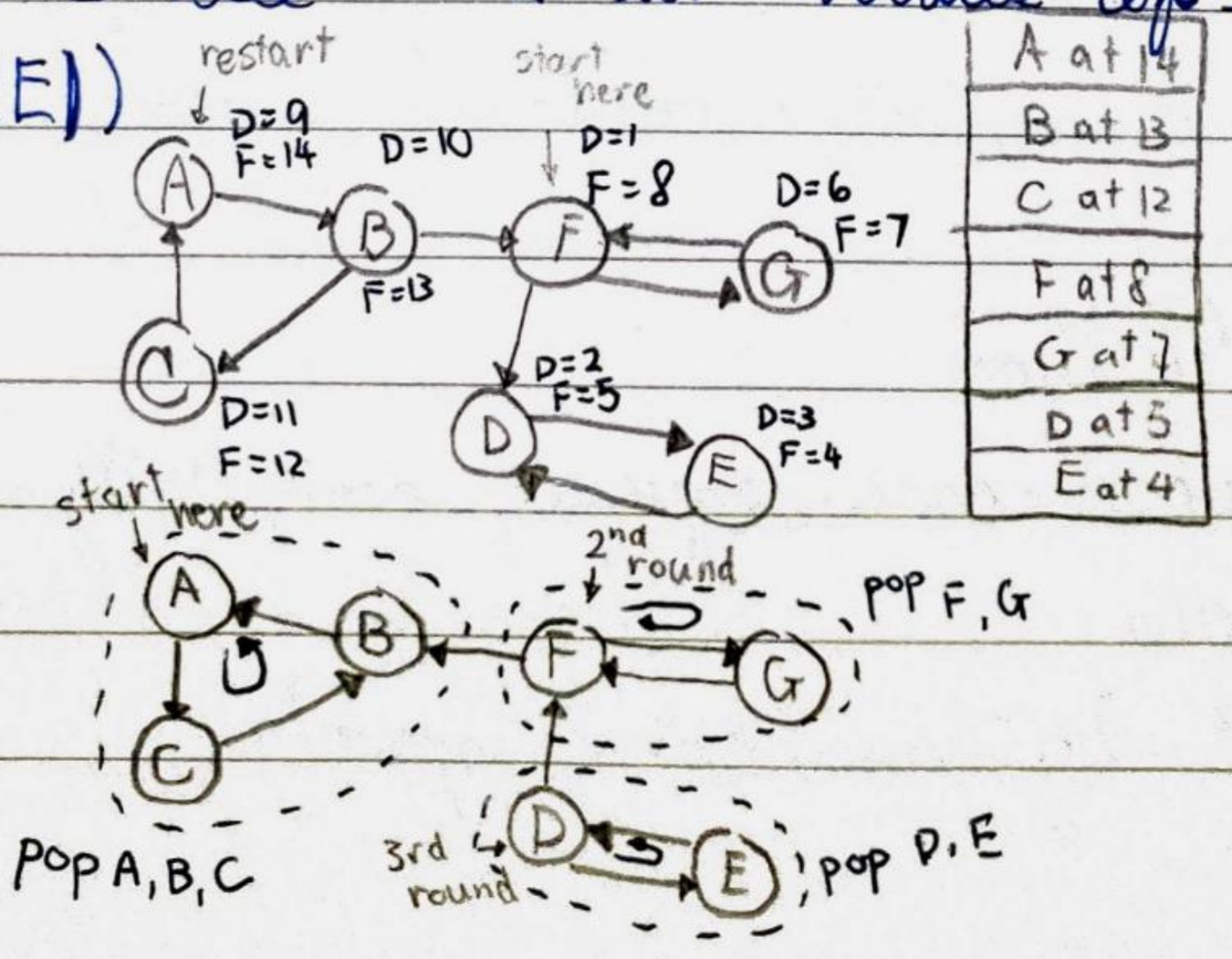
Complement of G (G^c)

has same vertices as G but set of edges is complement of set of edges in G i.e. if $(u,v) \in G$, $(u,v) \notin G^c$ and if $(u,s) \notin G$, $(u,s) \in G^c$

Kosaraju's SCC Algorithm

1. Run DFS on G and push on a stack when a ~~vertice~~ vertex is marked finish
2. Compute G^T (adjacency list)
3. DFS on G^T, starting with vertices on the top of the stack, pop all the ones that were visited and restart with the one at the top of the stack
4. Each DFS tree ~~is~~ ~~a~~ has vertices of one SCC

$O(|V|+|E|)$



The transpose did not affect SCC of G, but when running DFS in the order of stack, the reachable vertices are the ones that are $ strongly connected to it

Has something to do with finish time (Think of it as the last one finished can reach most of the vertices but a lot less in the transpose, and the ones that have earlier finishing time can reach more in the transpose but th not Strongly connected ones ~~are~~ have already been visited.

AMORTIZED ANALYSIS 24/02/2020

analysing the complexity of a sequence of operations (how it averages out over the whole sequence)

Worse case sequence complexity (m operations)

maximum total time over all sequences of m operations

$$\text{amortized complexity} = \frac{\text{worst-case sequence complexity}}{m}$$

Aggregate Method

Calculate worst-case sequence complexity of sequence of operations

Divide that by number of operations

DATE/ PAGE/

Accounting Method

- charge more for some operations and charge nothing for ~~ob~~ others
- leftover amount use as credits to pay for operations that are "free"

* assign charges and distribute credits such that each operation's cost will be payed and total credit is never negative

↳ total amount charged = upper bound of cost on performing a sequence of operations

sum charges over n operations, amortized cost = $\frac{\text{sum}}{n \text{ operations}}$

POTENTIAL METHOD

02/03/2020

$\Phi(h_0) = 0$ h_0 = initial state of data structure

no potential at the beginning

$\Phi(h_t) \geq 0$ h_t = state during the computation

Potential Function

measures how much potential at current state

$$T_A(i) = C_i + \Phi(h_{i+1}) - \Phi(h_i)$$

$T_A(i)$ - amortized time of operation i

C_i - actual cost of operation

$\Phi(h_{i+1}) - \Phi(h_i)$ - change in potential on ith operation

DATE/ PAGE/

potential change for low cost operation should be +ve

potential change for high cost operation should be -ve