DSA

- Arrays are contiguous in memory
  - not a variable ; just a name of a collection
  - not an lvalue that can be written on LHS of =
    - x[n] (x: an array) → x[1] = 2; // valid
      → x = 2; // invalid
    - array address = base addr + elem size * (i - first index)
      - constant access time

- 2D Array
  - in memory, 2D arrays are also contiguous
    - row major $[i_1,j_1 \mid i_1,j_2 \mid i_2,j_1 \cdots]$
    - col. major $[i_1,j_1 \mid i_2,j_1 \mid i_1,j_2 \cdots]$

- row major = base addr + elem size ((i - first index) N ... + (j - first index))
- col. major = base addr + elem size ((i - first index) ... + (j - first index) M)

M : row length
N : col. length

Example : arr [1 ... 10][1 ... 15]
base = 100 size = 1
Find arr [8][6]

Answer
row major → 210

EXCELLENT

- 3D Array
  - (width, row, col.)

→ row major = base addr + elem size (MN(i − first elem) ... + N(j − first elem) + (k − first elem))

Example } arr [1:9, -4:1, 5:10] size = 2

base = 400 → sign is considered

- Find arr[5][-1][8]

Answer

row major → 730

→ len (rows / cols / width) = upper − lower + 1

→ col. major = base addr + elem size (MN(i − first elem) ... + (j − first elem) + M(k − first elem))

Example } arr [1:8, -5:5, -10:5] size = 4

base = 400

- Find arr[3][3][3]

Answer

col major → 2412

Dry Run → slides

1	2
3	4

output

13/02/24

DSA

structure, class } C specific / in C++, functionality is same
- public vs. private scope, polymorphism is not applicable to structs; unlike classes

pointers contain address of a variable

const & pointers

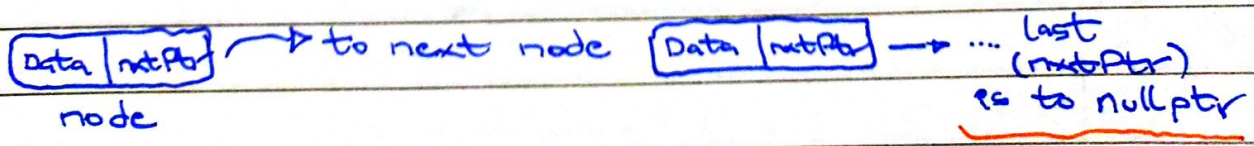
1. pointer to a constant [pointer value cannot be changed but location can]
   - const int * px = &x;
     - *px = 1; => invalid
     - px = &y; => valid
   - example

2. constant pointer [location cant be changed but value can]
   - int * const ptr = &a;
     - *ptr = 32; => valid
     - ptr = &b; => invalid

3. constant pointer to a constant [neither can change]
   - const int * const ptr = &a;

→ int * const r = &n; ≡ int &r = n;

linked list



to grow linked list in size; change nullptr to next (new) node

advantages
- dynamic
- easy insertion and deletion

disadvantages
- efficient search not possible
- O(n) complexity for random access
  - worst case

- Creating a linked list
  - pointers to NULL
  - inserting
  - deleting
  - traversing

  } so on

see implementations from slides (easy stuff :))

→ important : reversing a linked list

DSA

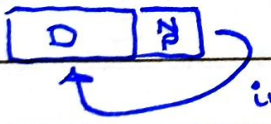
- **Linked List Variants**
  - Doubly linked list
  - Circular list

  } Sorted ; Skip ; Self Organizing) lists

- **Doubly linked list**

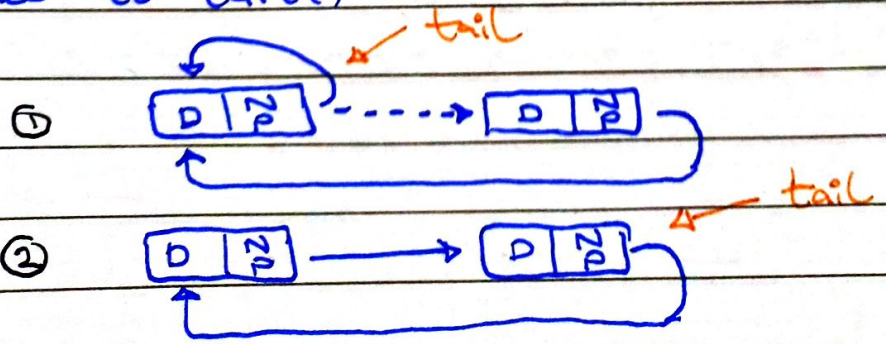
  - add to tail () { Read from slides }
  - delete from tail()

- **Circularly linked list**
  - for compute processes ; current pointer points to the executing process
  - only one permanent pointer ; "tail"



initial case (only one node)

add to tail()



ate

21/02/24

DSA

- Arrays of Linked Lists
  - can manage a collection of linked lists using arrays

- see practical implementaion from slides

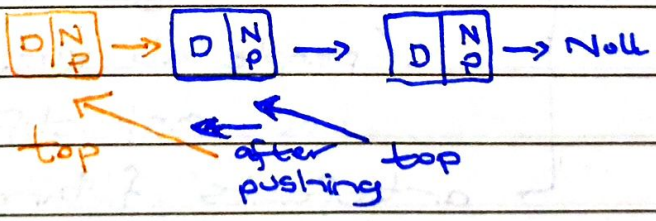
- Stack {LIFO / FILO}

  ignore ; last : top
  first : bottom

  - Top : Pointer to the first (top) element
  - Stack under/over flow >> adding to a full stack results in overflow ; removing from empty stack results in underflow

→ array based
→ linked list based
} easy

push()



→ Stacks for Validating Expression
i.e. (A + B)
[C + (A * B)]

↓

from slides

28/02/24

DSA

- **Operands & Operators**
  - (*) = (/) > (+) = (-) DMAS
  - if same prio. operators are used, predence is towards the left

infix {normal "x+y, a*b" etc.}

prefix = *+xyz → *(x+y)z → (x+y)*z etc.

postfix = xyz*+ etc.

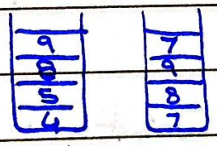
→ Algo.

- (A+B)/C+D → out = AB+C/D+

- **Adding large numbers**

num 1 : 4589
+
num 2 : 7897

carry = 0



stack 1 stack 2

Pop from each stack; add; push unit part to the result stack; continue

if carry is non-zero after both stacks are empty; append carry result to result stack

→ Recursion | Divide & Conquer

07/03/24

DSA

- Queue
  - when front == rear ; queue is empty
  - if rear == size - 1 ; queue is full (!?)
    - not true ; check slides
  - solution (circular queue) :o
    - maintain a counter

Read from slides

- Real World : Data Structures
  - Tree is non-linear {unlike arrays, etc}
  - special node = root
    - other nodes are partioned into disjoint sets
- Trees & Leaves : {leaves do not have any outgoing arcs/ links}
  - depth of a node [links from the root of the tree
  - → root := only outwards
  - → inner := both inwards & outwards
  - → leaf := only inwards
  - Binary Trees {left & right}
    - read from slides

March / 2024

DSA

- Trees {Binary}
  - Traversal
    - Pre order (RLR^i)
    - In order (LRR^i)
    - Post order (LR^{i}R)

  correspond to prefix, infix & postfix when evaluating expressions

  ∴ R^i : right
  R : root
  L : left

- Binary Search Tree
  - value in left child < x < value in right child (x: current value)
  - in order → ascending order

- Huffman Coding {compression}
  - occurrence probabilities » variable length codes
  - lossless
  - prefix of one cannot be another code

$$N_{avg} = \sum_{k}^{L} N_k P(k) \qquad \therefore P(k) = f(k)/\tau$$

(f(k): frequency, τ: total)

~ Deletion of BST ÖÖ

day/date

March / 2024

DSA

- **Big-O Notation**
  - analysis of algorithm's efficiency
    - complexity in terms of input size
    - machine independent
    - instruction dependent
  - → worst / best / avg case

e.g. $O(n)$ implies size increases linearly
$O(n^2)$ → quadratic

rules
- ignore constants
- terms dominate one-another {lower ones are ignored)|dropped}

- **AVL Trees** → Search : $O(\log n)$
  - self-balancing binary search tree (BST)
  - difference b/w left and right subtrees for any node is the "balance factor"

$BF(x) = height(x\text{->}right) - height(x\text{->}left)$

balanced when $BF < 2$

/date

April / 2024

DSA

- Left - Left (LL) case
  - node as well as its child is left-heavy
  - make
    - root → left = left child's right
    - left child's right = root
    - ~ theoretical / not-algorithmic

- Right - Right (RR) case
  - inverse of LL case → same operations but replace left with right and vice versa

- Left - Right (LR) case
  - when node is left-heavy but its left child is right-heavy

- Right - Left (RL) case
  - inverse of LR case

Heap (As almost complete binary tree)
- implemented through arrays
  - Parent(i) → i/2
  - Left(i) → 2i
  - Right(i) → 2i + 1

} indexing

→ max-heap } root of sub-tree holds max
→ min-heap } " " min

ate

- read operations from slides
  ^ _ ^

- when heapifying ($\log_2 n$ = $O(\log n)$ complexity)
  - redundant to call heapify() on leaf nodes
    - → $n/2$ to 1
      - → $O(\frac{n}{2} \log n) \sim O(n \log n)$
        Average is $O(n)$

- heapsort
  - given array A
  - build maxheap (A)
  - exchange A[1] ⇔ A[i]
  - decrease size [detach element]
  - call max-heap again

repeat

priority queue
- each element has a property that defines its priority

day/date 24th April /2024 , 1

DSA

Sorting Algos.

- Bubble Sort → for small arrays
  - 1st loop → (n-1) comparisons
  - 2nd loop → (n-2) ; so on
  - check iterative & recursive implementation from slides

- Insertion Sort
  - efficient for small-arrays
  - rest from slides ~~

- Selection Sort
  - search (min/max) element and put in selected position
  - $O(n^2)$; again, does not work on large datasets

- Merge Sort
  - Divide and Conquer
    - solve multiple subproblems
  - base case » p == r (array of size 1)

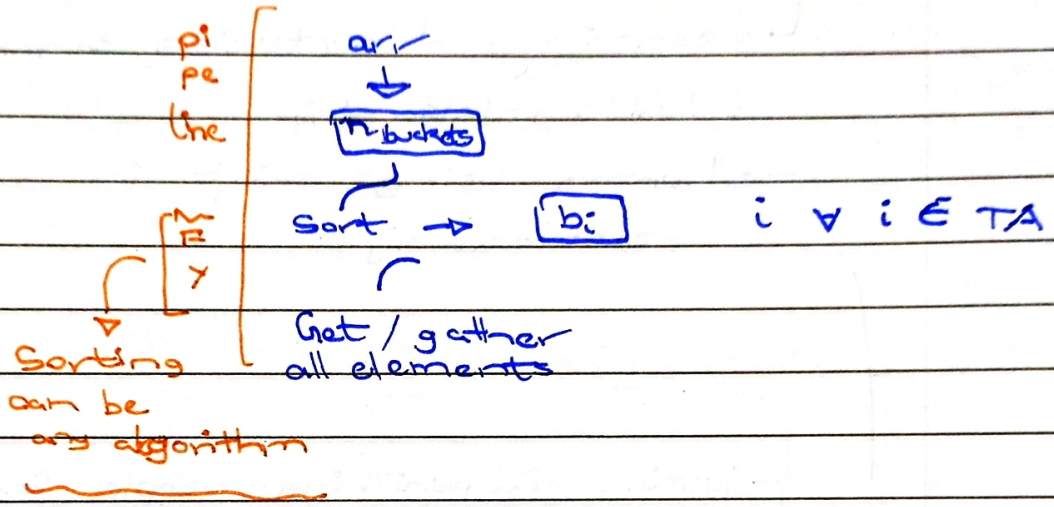
- Bucket Sort
  - create n empty buckets & loop
  - find bucket index and insert into buckets

y/date

April / 2024

DSA

~▸ continued Budget Sort



- Quicksort
  - select pivot element (i.e. rightmost)
  - make larger / smaller pointers for elements ≥ or < pivot
  - swap until pivot is in correct position
  - Kz rest from slides

ay/date

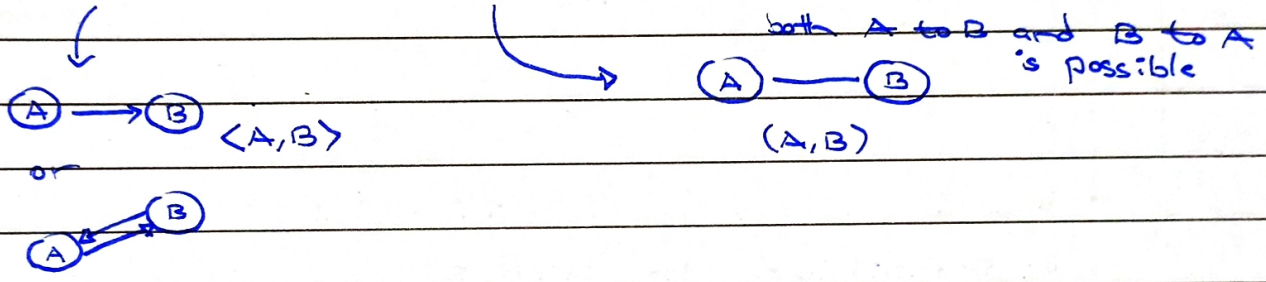
May / 2024

Data Structures & Algos.

Graphs

$G = (V, E)$ where V is set of vertex
E is set of edges b/w the vertices

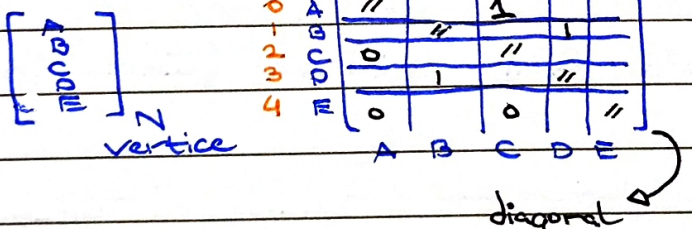
Directed & undirected graphs



↳ also expressed as nodes and paths
→ read slides for classifications

Adjacency Matrix

where 1 denotes existence of edge
↳ for undirected graph, symmetry about digonal exists



Adjacency List

↳ can be implemented using Linked Lists

{See C++ implementations from slides}

EXCELLENT

lay/date

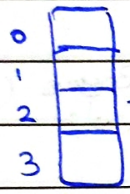
May / 2024

DSA

Graph Traversal

- Depth-first ⟶ Stack
- Breadth-first ⟶ Queue

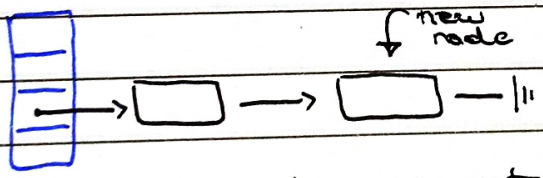
Hash Table {key: value} pairs



2%.4 = 2
14%.4 = 2 as well
} collision

- [Open Hashing] Chaining (Separate)

Same index start chaining



	best	worst	
insert	O(1)	O(n)	at ends of Linked List
find	O(1)	O(n)	
delete	O(1)	O(n)	

Load factor $\lambda = n/c$

- [Closed ~~Hashing~~] ~~Addressing~~ Open
  - Linear Probing [Add: if index is occupied]

EXCELLENT