DATA STRUCTURES & ALGORITHMS

- Labs (10 marks) (best 8 of ~11)
- Assignments (10 marks) (5 assignments)
- Mid lab + End lab : (10 marks + 15 marks) → ↑ increased
- Quiz 2 : (10 marks)
- Mid exam : (15 marks)
- End exam : (20 marks)
- Slip tests (Some marks from above will be alloted)

~ Lectures by Prof. Lini Thomas & Prof. Kshitij Gajjar, compiled by Aaryan Shah

__/__/____

*] Sorting:

① Insertion Sort:

→ 6 10 24 36 12

Pseudo Code: key = arr[i+1]

for (i=0, i<n-1, i++) → 2nd element

for (j = i-1, j ≥ 0, j--)

if (key < arr[j])

temp = arr[j]

arr[j] = arr[j+1]

arr[j+1] = temp

Improved Pseudo-code:

for (i=0, i<n-1, i++)

temp = arr[i+1]

for (j = i-1, j ≥ 0, j--)

if (temp < arr[j])

arr[j+1] = arr[j]

else

arr[j+1] = temp

⇒ Best Case: O(n)

Worst Case: O(n²)

HW: Write Code

o] Binary Search:

- Search for an element in a sorted array.

Step 1: Check middle element. (if < then no. to right)

(if > then no. to left)

Step 2: Check middle element of desired sub-array & recurse.

HW: Write code

⇒ Time Complexity: No. halved each time ⇒ O(log₂n)

⇒ * Q] Can binary search to find the right place to insert the next item help us better sort an array of integers?

→ NO. As the second loop would still be needed to shift the no.s in the array to make space.

⇒ * Q] In a permutation $a_1, a_2, \ldots, a_n$, an inversion pair (a_i, a_j) s.t. $i < j$ & $a_i > a_j$. What is the relation b/w insertion sort & inversions?

→ Total no. of steps in the insertion sort = No. of inversions in initial array. (as essentially swapping out inversions)

② Bubble Sort: H.W. Write code

- like.

⇒ Best case: O(n) with optimizatn

Worst case: O(n²)

NOTE: 1] Minimise writes: Selection Sort
2] Minimise swaps: Selection Sort
3] Almost sorted: Insertion Sort

__/__/____

③ Selection Sort: H.W. Write code

- Find largest no. put at last.
- Find largest till n-1 put at n-1, recurse.

Pseudo code:
```
for j → 1 to n-1
    smallest → j
    for i → j+1 to n
        if A[i] < A[smallest]
            smallest → i
    Exchange A[j] ↔ A[smallest]
```

HW Write code

Q] Let A be an array containing integer values. The distance is defined as $\min_i$ no. of swaps s.t. array sorted in non-decreasing order. The distance of [2,5,3,1,4,2,6] is?

→ least swaps ⇒ insertion ⇒ 3

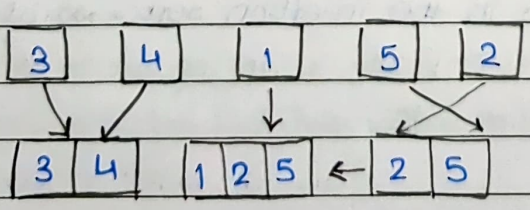
NOTE: Insertion sort uses the least no. of comparisons to sort an array.

*] Merge Sort: $O(n \log_2 n)$ ← Best Case & Worst Case

- The key to Merge Sort is merging two sorted lists into one, s.t. if you have 2 lists $X (x_1 \leq x_2 \leq \ldots \leq x_m)$ & $Y (y_1 \leq y_2 \leq \ldots \leq y_n)$ the resulting list is $Z (z_1 \leq z_2 \leq \ldots \leq z_{m+n})$.
- Minimum time required to sort any array with no constraints = $O(n \log n)$

→ Break the array into single units (by halving each time)

→ Then Merge



→ Rearrange pointers accordingly.

__/__/____

Ex: 56 5 7 2 99 17 3 9 85

5,56 | 2,7 | 17,99 | 3 | 9,85

2,5,7,56 | 17,99 | 3,9,85

2,5,7,56 | 3,9,17,85,99

2,3,5,7,9,17,56,85,99

} Whatever split merge back sorted

Code: mergeSort(arr, int left, int right){

if (left ≥ right){

return;}

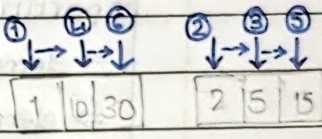
int mid = (left + right)/2;

mergeSort(arr, left, mid);

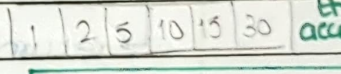
mergeSort(arr, mid+1, right);

merge(arr, left, mid, right);}

(⇒) merge(arr, beg, mid, end){ → essentially



⇒ Compare pointers & move them accordingly



int start1 = beg, start2 = mid+1, pos = 0;

while (start1 ≤ mid && start2 ≤ end){

if (arr[start1] ≤ arr[start2]){

arr_temp[pos++] = arr[start1]; start1++;}

else{arr_temp[pos++] = arr[start2]; start2++;}}

if (start1 > mid){

while (start2 ≤ end){

arr_temp[pos++] = arr[start2++];}}

if (start2 > end){while (start1 ≤ mid){

arr_temp[pos++] = arr[start1++];}}

copy arr_temp to arr(beg, end)}

NOTE: Instead of using $\log_2 n$ arrays, make an extra array of the same size & keep swapping using those two in each iteration.

"DOUBLE-STORAGE"

Time complexity:

- Both Worst & Best case $O(n \log_2 n)$

⇒ $T(n) = T(n/2) + T(n/2) + Cn$ → as ptrs iterating through total n

$= 2T(n/2) + Cn$

$2(2T(n/4) + Cn/2) + Cn = 4T(n/4) + 2Cn$

$4(2T(n/8) + Cn/4) + 2Cn = 8T(n/8) + 3Cn$

⋮

→ $2^{\log_2 n} T(n/2^{\log_2 n}) + \log_2 n \cdot Cn = nT(1) + Cn\log n$

→ $Kn + Cn\log n$ ∴ $T(n) = \max[Kn, Cn\log_2 n]$ ∴ $T(n) = O(n \log_2 n)$

$\exists\ c_*$ s.t. $T(n) \le c_* n \log_2 n \Rightarrow$ for $n \ge n_0, n_0 \in \mathbb{N}$

HW Q] Why 2-way Merge Sort? Is it more efficient?

→ Generally 2-way Merge Sort is more efficient than merge sort. For large arrays the efficiency is less than 5%. [redacted] No. of steps reduces.

*] Quicksort:

- Given an array of n elements:

→ if array contains only one element, return

→ else:

- pick one element to use as pivot
- partition elements into two sub-arrays
  - ⇒ elements ≤ pivot
  - ⇒ elements > pivot
- qsort the two sub-arrays
- return

Ex: 40 (pivot) 20 10 80 60 50 7 30 100

0 1 2 3 4 5 6 7 8

→ start pointer from index 1 & 2nd end pointer from last index

1] keep moving ptr start fwd until no. > 40 & stop start from end backwards & when no < 40 ⇒ swap

2] Re[peat] until start ≥ end.

3] Now, put pivot @ correct index by swapping with element @ index - 1 from the 1st no. > 40.

1st iter: 7 20 10 30 40 50 60 80 100

2nd iter: 7 20 10 30 40 50 60 80 100

3rd iter: 7 10 20 30 40 50 60 80 100

4] As seen above pivot ends up in correct positions. Then qsort the sub-arrays made by the pivot

Time Complexity: (Assuming that pivot element [redacted] always ends up in the middle)

→ $T(n) = 2T(\frac{n}{2}) + Cn \Rightarrow \boxed{T(n) = O(n \log_2 n)}$

__/__/____

Time Complex: (Assuming pivot ends up @ beginning or end)

→ $T(n) = T(n-1) + C_0 n$

$= T(n-2) + C_1(n-1) + C_0 n$

$= T(1) + \sum_{i=0}^{n-2} C_i (n-i)$

$\leq n^2 \quad \because \left(\sum = \frac{n(n+1)}{2}\right) \quad \therefore \boxed{T(n) = O(n^2)}$ ⇒ n steps of almost n elements

◦] Improved Pivot Selection:

- Pick median value of three elements from the array:
  data[0], data[n/2], data[n-1]
- use this median value as pivot.

*] Stacks:

- abstract data structure
- Push (add element to top of stack), pop (remove from top)

◦] Implementation of operations:

1] Push: int myTop = -1

→ Push(int x):

if array is not FULL (myTop < capacity-1)

myTop++

store x in array[myTop]

else output out of space

2] Pop:

→ Pop(int myTop)

if array not EMPTY (myTop ≠ -1)

remove x from array[myTop]

myTop --

else output array empty

0] What if static array initially allocated for stack is too small?

→ The most efficient solution is to use linked lists as it ensures best TC along with no memory wastage.

P.T.O.

__/__/____

0] Applications:

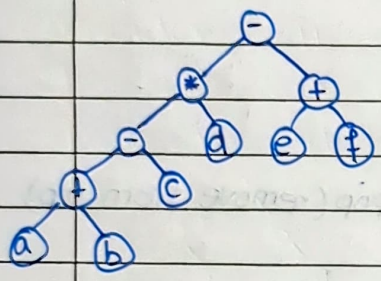
1] Check Balanced Paranthesis:

→ ([]){([[]]{})}

- Push open brackets into stack.
- If closed bracket encountered, if top of stack ≠ corresponding closed bracket → error ∴ Not balanced
- If top of stack = corresponding closed bracket ⇒ pop.
  Base case
- If stack empty ⇒ balanced ; Recurse

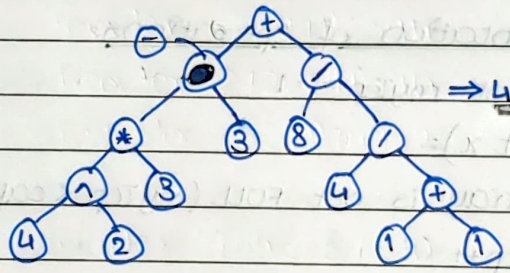
2] Converting Infix to Postfix:

1] (a+b-c)*d - (e+f)



⇒ ab+c-d*ef+-

2] 4^2*3-3+8/4/(1+1)



⇒ 42^3*3-84/11+/+

3] a+b/c*(d+e)-f

(a+[(b/c)*(d+e)]-f)

⇒ a+[(bc/)*(de+)]-f) = a+[bc/de+*]-f = abc/de+*+f-

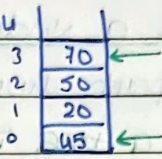
→ Scanning from left to right.

- If token = operand, append to end of output list.
- If token = (push to stack
- If token =) pop until corresponding (removed, append each operator to end of output list
- If token = *, /, + or - push to stack. However first remove any operators already on stack that have higher or equal precedence (BODMAS) and append them to end.
- Once done scanning check if any operators in stack, append them to the end.

__/__/____

NOTE: In case of repeated exponential (a^b^c) pop only higher not equal precedence, as otherwise: abc^^ (correct) instead we get ab^c^ (wrong)

*] Queues:



- Enqueue ⇒ Enter queue
- Dequeue ⇒ Leave queue

⇒ Everytime enqueue Rear ↑, dq ⇒ Front ↑ i.e: dq ⇒ 45 leaves.

0] Linear Queue:

1] Enqueue: if (front == -1) { front = 0; }

- if (rear == n-1) { printf("Queue full"); }
  else { rear++;
  scanf("%d", &arr[rear]); }

2] Dequeue:

- if (front == -1 || front == n) { printf("Queue empty"); }
  print dequeued element before front++;
  else { front++; }

0] Circular Queue:

1] Enqueue: if (front == -1) { front = 0; }

- if ((front == 0 && rear == n-1) || (front == rear+1)) { printf("full"); }
  else { rear = (rear+1)%n;
  scanf("%d", &arr[rear]); }

2] Dequeue:

- if (front == -1 && rear == -1) { printf("empty"); }
  else { printf("%d", arr[front]);
  front = (front+1)%n; }

*] Implementing stack using 2 Queues:

1] Push = Enqueue
2] Pop = Dequeue & enqueue into other queue until last element

*] Implementing Queues using 2 Stacks:

1] Enqueue = Push
2] Dequeue = Pop & Push into other stack until last element

__/__/____

* Binary Search Tree:

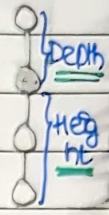
- For large input, linear traversal bad bad bad

◦ Depth of a Node:

- No. of edges in unique path from root to node

◦ Height of a Node:

- Longest path from the Node to its own leaf node



⟹ Depth of a tree is equal to height of the tree (Depth defined differently for entire tree)

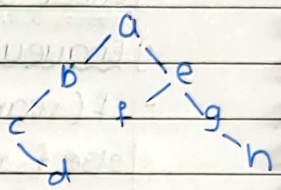
⊛ BUT: follow Lini's defn: Depth of tree = 0

◦ Traversals:

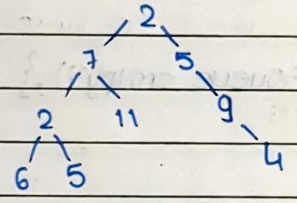
1] Inorder: Left Root Right: cdafegh

2] Preorder: Root Left Right: abcdefgh

3] Postorder: Left Right Root: dcbfhgea



Ex:

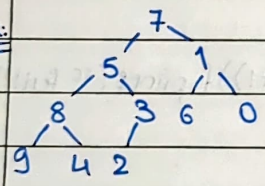


Inorder: 6257(11)2594

Preorder: 27265(11)594

Postorder: 652(11)74952

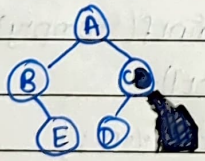
Ex:



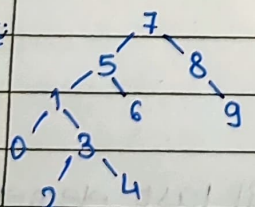
Inorder: 9845237610

Ex: Preorder = ABECD, Inorder = BEADC, Level order?

Put in BST ⟹



Ex:



Inorder = 0123456789

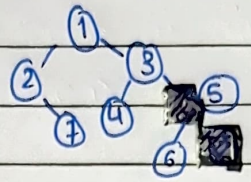
Pivot

Smaller Larger

NOTE: Inorder Traversal of BST gives you ascending sort

__/__/____

Ex: Preorder: 1,2,7,3,4,5,6 & 2,7,1,4,3,6,5 = Inorder



1] Full-Binary Tree:

- Also called proper binary tree/2-tree in which every node other than the leaves has two children.

Ex: Pre = 1,2,3,4,5,6,7 , Post = 2,4,6,7,5,3,1 . Construct full Binary Tree.