No. Date.

Data Structures and Algorithm - 2024 (November)

01) a) What is the key principle by which a stack operates and what are the two main operations associated with it.

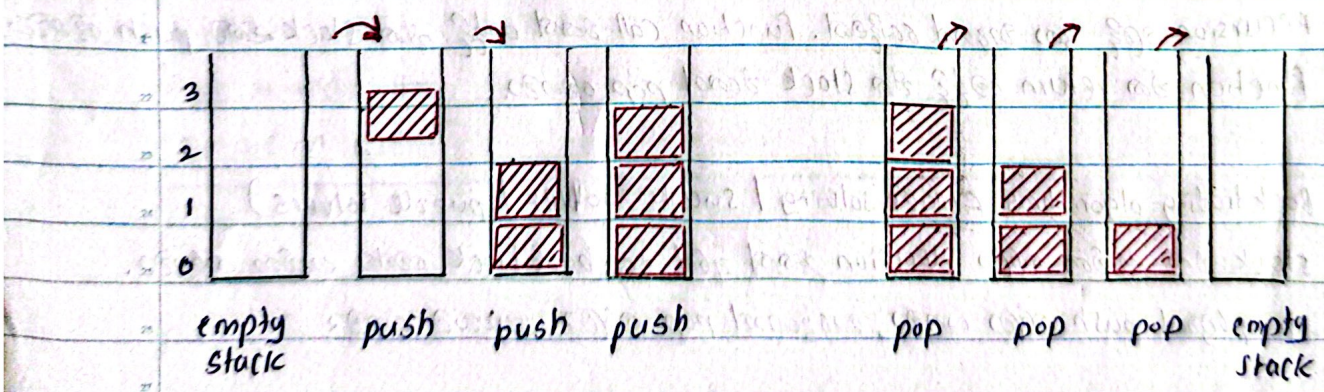
principle : A stack works on the LIFO (Last In, FirstOut) Principle.

Main operations : push - Adds an element to the top of the stack
pop - Removes top element from the stack

extra points -

stack කියන්නේ data structure එකක්, data store කරන්න හාවිත කරන ක්‍රමයක්. stack වල විශේෂත්වය තමයි මේකේ LIFO කියන නියමයට අනුව. stack එකේ හාවිත වන operations.

push - නව දත්ත stack එකට ඉහළින් එකතු කරනවා - push(s)
pop - stack එකේ ඉහළම දත්තය ඉවත් කරනවා - pop()
peek/Top - stack එකේ ඉහළම අගය බලනවා - peek()
isEmpty - stack එක හිස්ද කියලා check කරනවා - True or False
isFull - stack එක පිරිලද කියලා check කරනවා - True or False



b) How does the stack data structure support real ~~data~~ world applications such as browser history or undo operations in text editors.

- Browser History – when navigating web pages, each visited page is pushed onto a stack. The "Back" button pops the most recent page, allowing users to return to the previous one.
- Undo operations – In text editors, each action is pushed onto a stack. when undo is triggered, the last action is popped and reversed

Extra Points: real world applications in using stack

1) undo mechanism in Text editors (Microsoft word / Notepad / google Docs)
අපි type කරන හැම step එකක්ම stack එකේ save වෙනවා. undo (Ctrl+Z) click කරනකොට අවසානයේ කරපු step එක pop වෙලා ඉවත් වෙලා යනවා.

2) Browse History Navigation (Google chrome / firefox)
අපි යන හැම webpage එකක්ම stack එකේ save වෙනවා. back button එක select කරාම ඉහළම web page එක pop වෙලා ආපහු යනවා.

3) function call stack (Recursion & Programming execution) – Java, python, C++
Recursion වලදී ඉතා වැදගත් දෙයක්. Function call එකක් වෙද්දී ඒක stack එකට push වෙනවා. Function එක return වෙද්දී ඒක stack එකෙන් pop වෙනවා.

4) Backtracking algorithms (Maze solving / sudoku solver / puzzle solvers)
stack එක භාවිත කරලා decision එකක් අතේ එය backtrack වෙන්න භාවිත කරනවා.
නව පාරවල් push කරලා යනවා, වැරදුනොත් pop කරලා පසුපසට යනවා

5) Expression Evaluation & syntax Parsing (Compiler design / calculator)
postfix / infix / prefix expressions evaluate කරන්න stack එක භාවිත වෙනවා. විශේෂයෙන් operator precedence handle කරන්න stack එක භාවිත වෙනවා.

6) Reversing Data (String reverse කිරීම)
String එකේ characters stack එකට push කරලා පසුව pop කරන විදියට print කරද්දී reverse වෙනවා.

7) Balancing Symbols (code editor එකේ {}, [], () check කිරීම)
Symbols වල balance check කිරීම. opening symbol එකක් push කරලා closing එකක් එක්ක pair එකද බලලා check කරලා pop කරනවා)

c) Can queues perform the task mentioned in b) more effectively.

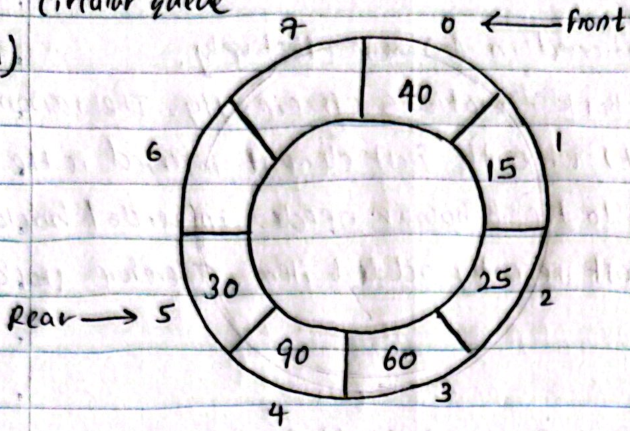
No, queues cannot perform undo or back operations efficiently. The reason is queues work on FIFO (First In First Out), where the first element inserted is the first to be removed. This is opposite to the behavior needed in undo / back operations which require access to the most recently added item. Therefore stacks are better suited for such tasks.

extra points: Queue කියන්නේ FIFO (First in first out) කියන්නේ දන්නවා, ඒනිසා Queue එකේ ඇතුළත් කළ දත්තය මැකුවම ඉවත් වෙලා යනවා. undo or Back operations වල අවශ්‍ය වෙන්නේ කළ දත්තය මැකුවම ඉවත් නෙවේ. එනිසා LIFO කියන්නේ අවශ්‍යයි.

Feature	stack	Queue
Operation Principle	Last In, First Out	First In, First Out
Main operations	push() - add to top	enqueue() - add to rear
	pop() - remove from top	dequeue() - remove from front
Data access direction	Top	Rear
Deletion location	Top	Front
Access to last element	Fast & Easy (pop)	Slow / impossible without extra work
Structure type	Linear	linear / circular
Access to middle item	Not allowed directly	Not allowed directly
Recursion use	widely used	Not used

Circular queue

d)



සාමාන්‍ය queue එකේ ඉඩ space wastefully වෙනවා. ඒ වළක්වන්න circular queue එකක් භාවිත වෙනවා. මෙහිදී queue එකේ අගට ගියාම ආපහු මුලට වන ලෙසට wrap around ක්‍රියාත්මක වෙනවා.

Formula:

$$Rear = (Rear + 1) \% size$$

data එකට තියෙන ස්ථානය rear එකේ ලෙස තෝරාගන්නේ මේ formula එක use කරලා.

(i) Insert (50)

Rear = (5+1) % 8 = 6

Insert 50 at index 6

Count = 7

(ii) Insert (70)

Rear = (6+1) % 8 = 7

Insert 70 at index 7

Count = 8 (Queue is full now)

(iii) Insert (100)

Cannot insert → Queue is full

No Changes

(iv) Remove ()

Remove at index front = 0 (value = 40)

Front (0+1) % 8 = 1

Count = 7

(v) Remove ()

Remove at index front = 1 (value = 15)

Front (1+1) % 8 = 2

Count = 6

(vi) Insert (35)

Rear = (7+1) % 8 = 0

Insert 35 at index 0

Count = 7

Initial State

Index : 0 1 2 3 4 5 6 7

Queue : [40, 15, 25, 60, 90, 30, —, —]

Front = 0, Rear = 5, Count = 6

[40, 15, 25, 60, 90, 30, 50, —]

[40, 15, 25, 60, 90, 30, 50, 70]

[40, 15, 25, 60, 90, 30, 50, 70]

[—, 15, 25, 60, 90, 30, 50, 70]

[—, —, 25, 60, 90, 30, 50, 70]

[35, —, 25, 60, 90, 30, 50, 70]

e) A circular queue is considered full when (rear+1) % size == front"

Agree with this statement. in a circular queue, elements wrap around the array. The condition (rear+1) % size == front indicates that the next position of rear would overlap with the current front, leaving no space for a new element. This is standard way to detect a full circular queue without confusing it with the empty state

Extra points :

circular queue එකේ කියන්නේ normal queue එකක් වගේමයි. එකම වින්ශාවය Queue එකේ අග end වෙද්දී start වලට යනවා. ඒ නිසා arrays වලින් හැසිරෙනවා

(rear +1) % size == front මේ condition එක circular queue එක පිරිලද කියලා හඳුනාගන්න භාවිත කරනවා.

rear කියන්නේ දැන් queue එකේ අවසාන element එක තියෙන තැන. (rear+1)% size කියන්නේ next position එක. circular queue එකේ rear එක wrap around වෙනවා. එනෙව wrap වුනොත් rear+1 % size → 0 වෙනවා. Queue එකේ නව element එකක් දාපුවට තියෙන ස්ථානය (rear+1)% size. නමුත් ඒ ඔබාගෙ front එක වුනොත්

(rear +1) % size == front

No. Date.

(a)
```
public class stackx {
  private char [] wordstack;
  private int maxSize;
  private int top;

public stackx (int size) {
  maxSize = size;
  wordstack = new char[maxsize];
  top = -1;
 }

public void push (char c) {
 if (!isfull()){
  wordstack [++top] = c;
 }
}

public char pop(){
if (!isEmpty()) {
  return wordstack [top--];
 }
  return '\0'; //return null char if empty
}

public char peek(){
 return wordstack[top];
}

public is Empty(){
  return top == -1;
}

public boolean isfull(){
   return top == maxsize-1;
}
```

```
public String reverseString (String input) {
 StringBuilder output = new StringBuilder();
 int length = input.length();

 StackX s = new stackx (length);

 for (int i = 0; i<length; i++) {
  char ch = input.charAt(i);
 if (ch != ' ') {
  s.push (ch);
 } else {
  while (!s.isEmpty()) {
   output.append (s.pop());
 }
  output.append (' ');
 }
}

while (!s.isEmpty()) {
  output.append (s.pop());
}
 return output.toString();
}
}
```

extra points :

```java
public class Stackx {
    private char[] wordStack;   // character එකක් වශයෙන් වචන ආපිළිම ගත stack එක
    private int maxSize;        // stack එකට දැමිය හැකි උපරිම ගණන
    private int top;            // stack එකේ උඩම ඇති element එකේ index එක

    public Stackx(int size) {   // size එකට අනුව අලුත් stack එකක්
        maxSize = size;         // initialize කරනවා. top = -1 කියන්නේ මේ
        wordStack = size;       // stack එක හිස් කියලා.
        wordStack = new char[maxSize];
        top = -1;
    }
```

→ push() කියන්නේ stack එකට අලුත් character එකක් දමන function එකක්.
++top කියන එකෙන් top එක එකකින් වැඩිවෙලා එම තැනට character එක දමනවා.
isFull() එකෙන් බලන්නේ stack එක පිරිලද නැද්ද කියලා.

→ pop() කියන්නේ stack එකෙන් උඩම character එක අයින් කරන එකට.
top -- කියන්නේ ඒ character එකක් remove කරන එකට. හිස් නම් return කරනවා

→ isEmpty() & isFull() method වලින් stack එක වෙලාවෙන් හිස්ද, පිරිලද බලනවා.

```java
public String reverseString(String input) {
    StringBuilder output = new StringBuilder();
    int length = input.length();
    Stackx s = new Stackx(length);
```

→ main method එක තමයි වැදගත්ම function එක. මෙහි output කියන එකෙන් final result එක store වෙනවා. length කියන එකෙන් represent වෙන්නේ input එකේ දිග. අලුත් object name එක තමයි s කියන්නේ.

→ Loop එකේදී සිද්ධ වෙන්නේ space එකක් ආවම ඒ වන ප්‍රමාණය කියලා තියෙන stack එකේ ඒ word එක reverse කරලා output එකට දානවා පස්සේ ඒ space එකත් output එකට එකතු කරනවා.

```java
    while(!s.isEmpty()) {
        output.append(s.pop());
    }
```

input එක ඉවර වෙද්දී stack එකේ තියෙන්නේ අවසාන වචනය ඒක reverse කරලා output එකට දෙනවා. Final result එක String එකක් විදියට දෙනවා (return output.toString();

9) public class ReverseWordsApp {
public static void main (String [] args) {
String original = "Hello how are you";

StackX stackobj = new StackX (original.length());
String reversed = stackobj.reverseString (original);

System.out.println ("original:" +original);
System.out.println ("reversed words:" + reversed);
}
}

No. Date.

Question 2

a) differentiate between singly linked list and doubly linked lists.

Feature	Singly linked list	Doubly linked list
Structure	each node contains data and a pointer to the next node	each nodes contain data, a pointer to the next node and a pointer to the previous node
Traversal Direction	only forward traversal is possible	Both forward and backward traversal is possible
Memory usage	uses less memory (1 pointer per node)	uses more memory (2 pointers per node)
Insertion / Deletion	slower (must traverse from head)	faster (can move both directions)

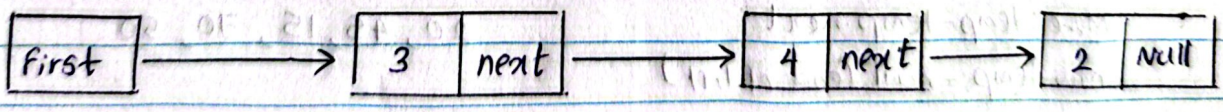
Applications:

Singly linked lists - used in implementing stacks and queues (call stack in recursion)

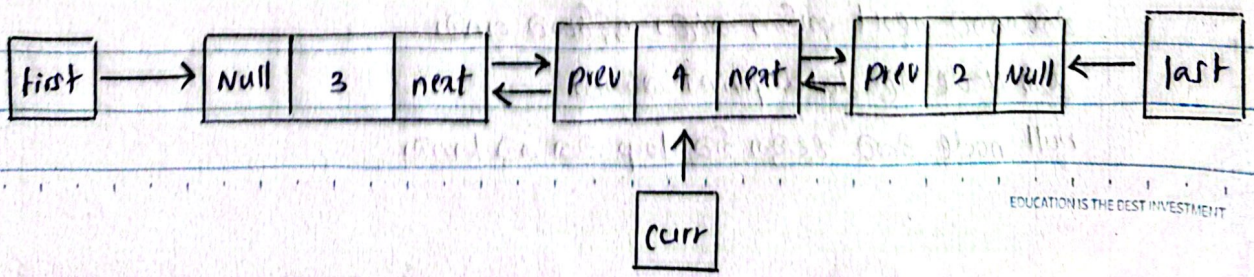
Doubly linked list - used in web bowsers to navigate backward and forward through history

Extra points:

Singly linked list

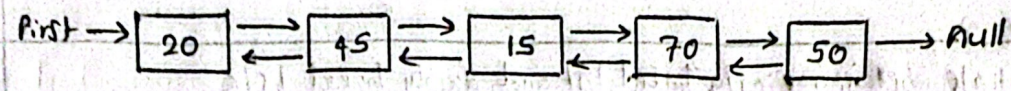


Doubly-linked list



EDUCATION IS THE BEST INVESTMENT

b)



(i) Link temp = first;

while (temp != null)

{

system.out.print(temp.data + ", ");

temp = temp.next;

}

first කියන්නේ 20 data එක තියෙන node එක ⇌ කියන එකෙන් පෙන්නන්නේ backword සහ forword pointers.

link temp = first; temp කියන temporary variable එකෙට first node එක assign කරනවා. ඒ කියන්නේ temp.data = 20, temp.next = node (45)

while (temp != null) කියන්නේ temp කියන node එක null නොවන දුරට මේකෙන් කියවෙන්නේ

//iteration by iteration

Iteration 1:

temp.data = 20

print: 20,

Move: temp = temp.next

now temp points to node with 45

Iteration 2

temp.data = 45

print: 45,

Move: temp = temp.next

now temp points to node with 15

Iteration 3:

temp.data = 15

print: 15,

Move: temp = temp.next

now temp point to node with 70

Iteration 4:

temp.data = 70

print: 70,

Move: temp = temp.next

now temp point to node with 50

Iteration 5:

temp.data = 50

print: 50,

Move: temp = temp.next

now temp = null (end of list)

දැන් temp == null නිසා while loop එක නතරවෙනවා.

final output එක:

20, 45, 15, 70, 50

– extra points –

loop එක first node එකෙන් පටන්ගන්නවා

එකින්එක next හාවිත කරලා ඉදිරියට යනවා

එක් Node එකේ data print කරනවා

null node එකට එළඹුන විට loop එක නවතිනවා

(ii)
```
Link temp = first;
 while (temp.next != null)
     temp = temp.next;
while (temp != null)
{
System.out.print(temp.data + ", ");
 temp = temp.prev;
}
```

first = node(20) නම් node එකෙම තියෙන්නේ data value එක / next pointer එක / prev pointer එක.

link temp = first; temp කියන variable එක first node එකට assign වෙනවා. අපේ list එකේ first.data = 20

while (temp.next != null) temp = temp.next;

මේ loop එකෙන් temp එක list එකේ අග node එකට යනවා.

Traversal steps:

- temp = node(20) → temp.next != null → go to 45
- temp = node(45) → go to 15
- temp = node(15) → go to 70
- temp = node(70) → go to 50
- temp = node(50) → temp.next == null → stop here

දැන් temp point වෙන්නේ list එකේ අවසාන node එකට (50)

while (temp != null) මේ loop එකෙන් prev pointer පාවිච්චි කරමින් පසුපස යනවා

Iteration	temp.data	printed	Node (temp = temp.prev) → Next
1	50	50,	70
2	70	70,	15
3	15	15,	45
4	45	45,	20
5	20	20,	null → loop ends

final output - 50, 70, 15, 45, 20

c)(i) Insert value 30 after the existing node value 70

```
Link temp = first;                    temp කියන variable එකෙන් list එක traverse
while (temp != null && temp.data != 70)   කරලා 70 තියෙන node එක හොයනවා.
  temp = temp.next;
if (temp != null) {
  Link newNode = new Link(30);   // අලුත් node එක හදනවා (data = 30)
  newNode.next = temp.next;      (30 → 50)
  newNode.prev = temp;           (30 ← 70)
if (temp.next != null)
  temp.next.prev = newNode;      70 → 30 → 50   } මෙතෙක links update වෙනවා
  temp.next = newNode;           50 ← 30 ← 70   }
}
```

20 ⇄ 45 ⇄ 15 ⇄ 70 ⇄ 30 ⇄ 50

(ii) Remove the first link from the linked list

```
if (first != null) {
  first = first.next;     // 20 node එක skip වෙනවා
  if (first != null)
    first.prev = null;    // 45 node එකේ prev null කරලා "new head" වෙනවා
}
```

45 ⇄ 15 ⇄ 70 ⇄ 30 ⇄ 50

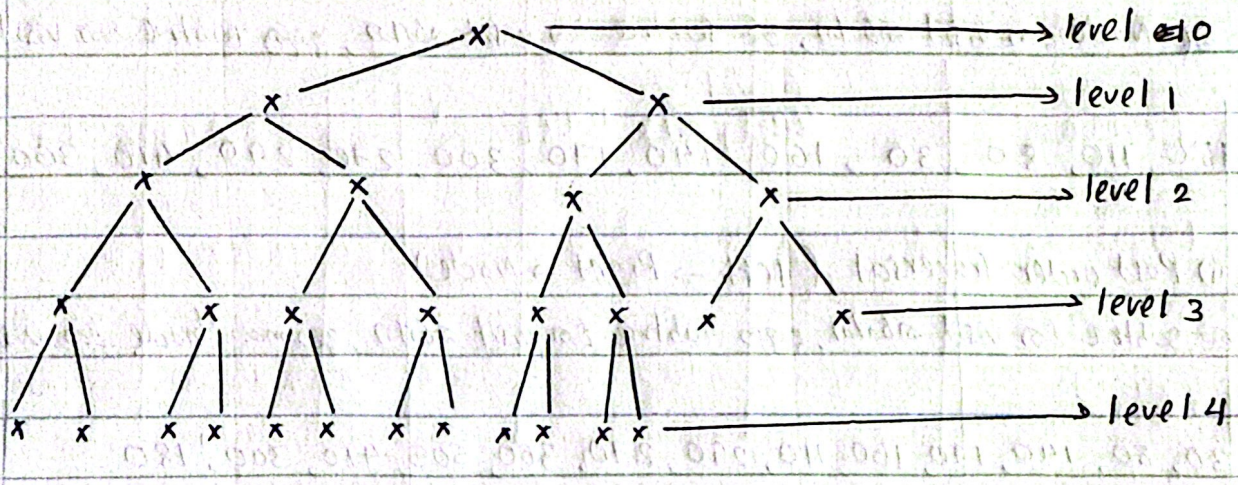
d) Assume a complete binary tree consists of 27 nodes. Calculate the height of the tree

level 0 has up to $2^0 = 1$ node
level 1 has up to $2^1 = 2$ nodes
level 2 has up to $2^2 = 4$ nodes
level 3 has up to $2^3 = 8$ nodes
level 4 has up to $2^4 = 16$ nodes

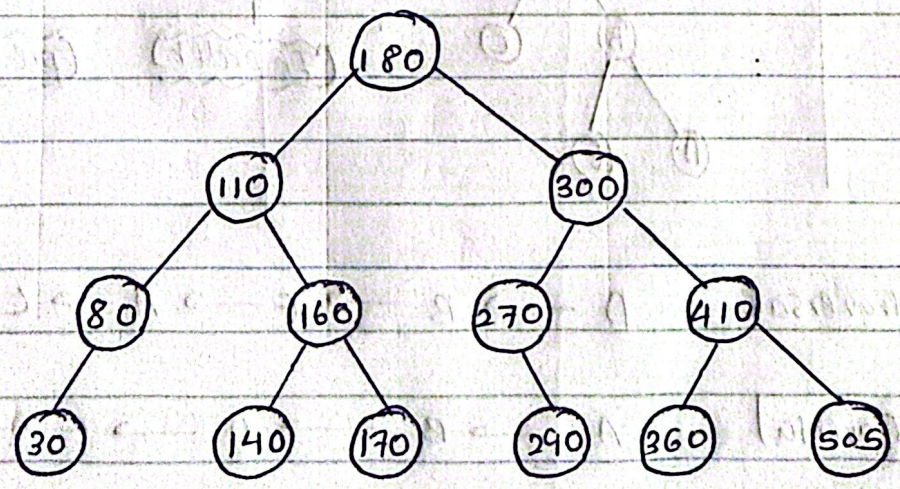
$h = \lceil \log_2(n) \rceil$

$\log_2(27) \simeq 4.75 = 4$

Height = 4



e) Consider the values given below. Arrange them into a binary search tree
(180 / 300 / 270 / 110 / 80 / 160 / 410 / 140 / 290 / 360 / 170 / 30 / 505)



(A) (i) in order Traversal (left → Root → Right)

පළමුව left subtree එකට යනවා, ඊළඟට root node එකට යනවා, ඊළඟට right subtre යනවා.
(Left → Root → Right)

30, 80, 110, 140, 160, 170, 180, 270, 290, 300, 360, 410, 505

(ii) Pre order Trarersal (Node → left → Right)

මුල Node එක visit කරනවා, වම් subtree එක visit කරනවා, දකුණු subtree එක visit කරනවා.

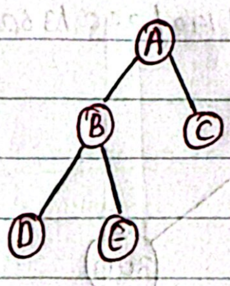
180, 110, 80, 30, 160, 140, 170, 300, 270, 290, 410, 360, 505

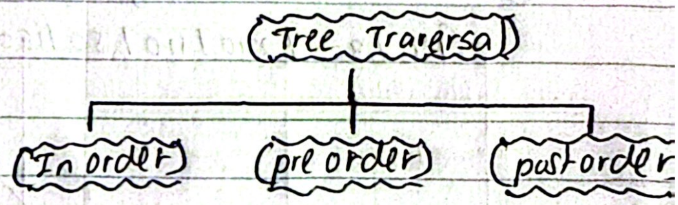
(iii) Post order Traversal (left → Right → Node)

වම් subtree එක visit කරනවා, දකුණු subtree එක visit කරනවා, අවසානයේ මුල node එක visit කරනවා.

30, 80, 140, 170, 160, 110, 290, 270, 360, 505, 410, 300, 180

extra point-



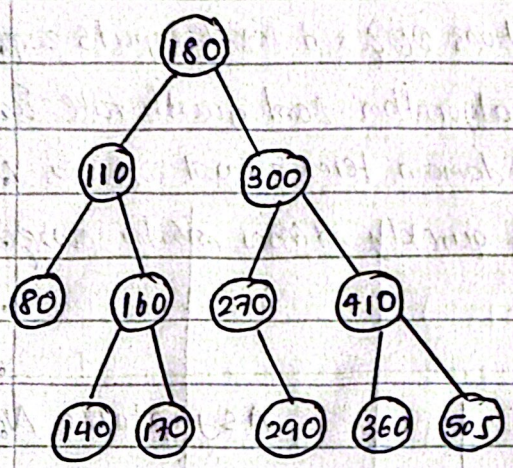


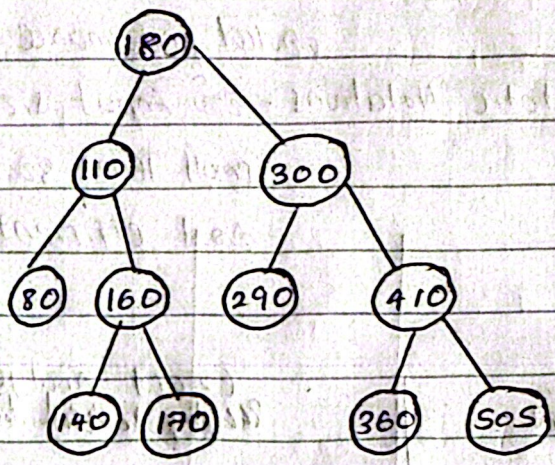
In order Traversal - D → B → E → A → C

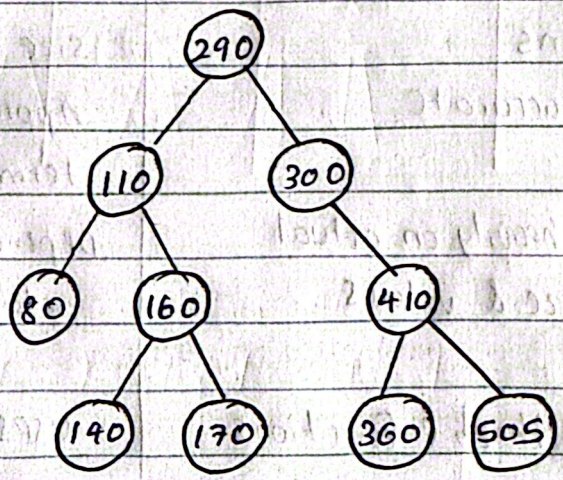
pre order Traversal - A → B → D → E → C

post order Trarersal - D → E → B → C → A

9) (i) Delete (30), (ii) Delete (270), (iii) Delete (180)







Question (03)

a) Compare and Contrast Exact anlysis and asymptotic Notations

Exact analysis - Algorithm එකක් තුළ තිබෙන operation ගණන ගණනය කරයි. මෙමගින් නිවැරදි performance simulations වලදී හෝ small inputs සඳහා

Asymptotic Notations - මෙහි input size වලදී algorithm එකේ growth rate විශ්ලේෂණය කරනවා small terms ඉවත් කරලා domain term විතරක් බලනවා. Algorithm එකක් efficient ද නැද්ද quickly තීරණය කරගන්න හොඳයි.

Feature	Exact Analysis	Asymptotic Notations
Definition	Measures the exact number of operations	Describes the growth rate as input size increases.
Accuracy	Highly accurate	Approximate - focuses on dominant term only
Dependence on Input	Depends heavily on actual input size & values	Depends only on input size, not actual values.
Notation used	uses arithmetic functions	uses Big-O, Big-Ω, Big-Θ notations
Ease of use	More complex to compute	easier and more general for performance comparisons

No. Date.

b) (i)

```
i=0
for (j=0 to 4)
    while (i<=2)
        i=i+1
    print j
```

මේ code segment එක step by step explain කරන්න ඒකේ time complexity එක O(1) කියන එක input size n එකට අදාල වෙන්නෙ නෑ කියන එක ලියන්න.

i=0 එකෙන් කියන්නේ i එකේ value එක 0

for (j=0 to 4) j එක 0 ඉඳන් 4 වෙනකම් ඒනවා. ඒ කියන්නේ 5 වතාවක් loop එක යනවා. j = 0, 1, 2, 3, 4

→ Iteration 1: j=0

i=0

while (i<=2) → true

i=1

again check i<=2 → true → i=2

again check i<=2 → true → i=3

check again i<=2 → false → exit loop

So while loop එක 3 වතාවක් run වෙලා i=3 වෙනවා. ඒතුවම print j (#output: 0)

→ Iteration 2: j=1

NOW i=3

check while (i<=2) → false skip the loop - (Print j) #output: 1

→ Iteration 3: j=2

i=3 still

while (i<=2) → false → skip (print J) #output: 2

→ Iteration 4: j=3

i=3

while (i<=2) → false → skip (print j) #output: 3

→ Iteration 5: j=4

i=3

while (i<=2) → false → skip (print j) #output: 4

$T(n) = O(1)$

j value	0	1	2	3	4
i Before	0	3	3	3	3
while loop runs ↓ 3 times	✓	x	x	x	x
i After	3	3	3	3	3
output	0	1	2	3	4

(ii) j = 10
while (j >= 0)
print j
j = j - 2

මේ code segment එකට අදාළව step by step loop execution, number of steps & Time complexity බලමු.

Initial value j = 10 (j 0 ට වැඩි හෝ සමාන නම් run වෙනවා)

inside loop j එක 2 කින් අඩු වෙනවා (j = j - 2)

loop iterations:

Iteration	j Before	Condition	Action	j After	Output
1	10	10 ≥ 0 ✓	print(10)	8	10
2	8	8 ≥ 0 ✓	print(8)	6	8
3	6	6 ≥ 0 ✓	print(6)	4	6
4	4	4 ≥ 0 ✓	print(4)	2	4
5	2	2 ≥ 0 ✓	print(2)	0	2
6	0	0 ≥ 0 ✓	print(0)	-2	0
7	-2	-2 ≥ 0 ✗	exit loop		

Total iterations = 6 Loop එක 6 වතාවක් විතරක් run වෙනවා, නමුත් මේක කවදාවත් input n එක මත දුවන්නේ නෑ. මේ loop එකේ execution count එක constant එකක්. කිසිම input variable එකකට අයත් නැහැ. j එක 10 කියලා fixed value එකක්. j = j - 2 කියන එකෙන් value එක 2 කින් අඩු වෙනවා. ඒ නිසා while loop එක always run වෙන වාර count එකේ count එක fixed = 6

So T(n) = O(1) No matter what happens outside. code block එක fixed count එකක් විතරක් දුවනවා.

Fixed starting value (j = 10)
Fixed step down (-2)
Fixed end condition (j >= 0)
So fixed steps → ✓ O(1)

c) Any recursive problem can have only one base (initial) condition

Recursive function එකක් කියන්නේ එකම function එක තමන්ට කතා කරන function එකකට. Base case(s) කියන්නේ මේ recursion එක නවත්වන්නේ. (infinite recursion වළක්වන්න) Recursive problem එකකදී එක base case එකක් පමණක් නොවේ · බොහෝවිට base case කිහිපයක් තියෙන්න පුළුවන්. ඒක completely problem එකේ structure එක මත රඳේ.

d) QUICKSORT (A, p, r)

```
1 if p < r
2   q = PARTITION(A, p, r)
3   QUICKSORT(A, p, q-1)
4   QUICKSORT(A, q+1, r)
```

Quicksort එකේ Best case එකට Recurrence Equation එක

partition එකකින් array එක 02කට බෙදෙනවා. එනිසා දෙපස recursive call 02ක් දානවා

Best case එක කියන්නේ array එක හරියටම සමාන 02කට බෙදුනොත් partitioning step එකට වැය වෙන්නේ O(n).

Recursive call වලට n/2 ක තරම් array එකකට යන්නේ

$$\therefore \quad T(n) = 2T\left(\frac{n}{2}\right) + cn$$

Identify the cost of each step

Step	Description	Time Cost
Step 1	Condition check	O(1)
Step 2	PARTITIONing (rearranging array)	O(n) (linear time)
Step 3	left subarray recursive call	unknown → write as T
Step 4	Right subarray recursive call	unknown → write as T

Best case assumption → portion එකෙන් array එක හරි 02කට බෙදේ.

left array size → n/2

Right array size → n/2

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

$$\boxed{T(n) = 2T(n/2) + Cn}$$

e) $T(n) = 2T(\frac{n}{2}) + cn$ මේක master theorem වගේ ලියන්න

$T(n) = aT(\frac{n}{b}) + f(n)$ $a=2$ | $b=2$ | $f(n)=cn$

$\log_b a = \log_2 2 = 1$

$f(n) = O(n)$

$n^{\log_b a} = n$

$T(n) = \Theta(n \log n)$

master theorem කියන්නේ divide and conquer algorithms වල recurrence relations වලට පාවිච්චි කරන්න rule set එකක්.

$$T(n) = aT(\frac{n}{b}) + f(n)$$

a = Recursive call වල ගණන

b = Input size එක කීවෙනි divide කරෙන්නේද කියන එක

$f(n)$ = Divide / combine stages වලට වෙන්න ගන්න කාලය

(f)

Property	Quick Sort	Merge Sort
Best case Time	$\Theta(n \log n)$	$\Theta(n \log n)$
worst case Time	$\Theta(n^2)$	$\Theta(n \log n)$
Average Case	$\Theta(n \log n)$	$\Theta(n \log n)$
Memory (Space)	in place ($O(\log n)$ stack space)	$\Theta(n)$ extra space
Stability	✗ Not stable	Stable
Speed (practical)	Fastest in practice	Slower due to copy.

(9)

	1	2	3	4
A	7	1	5	3

```
New Insertion-SORT(A)
for j=2 to A.length
    i=1
    while A[j] > A[i]
        i = i+1
    key = A[j]
    for k=0 to j-i-1
        A[j-k] = A[j-k-1]
    A[i] = key
```

j=2 to 4

i=1

while $A[2]^{1} > A[1]^{7}$

X

key = A[2] = 1

for k=0 to 2-1-1 (0)

A[2-0] = A[2-0-1]

$A[2] \leftarrow A[1]^{7}$

A[1] = 1

A	1	2	3	4
	1	7	5	3

j=3 to 4

i=1

while $A[3]^{5} > A[1]^{1}$

i = 1+1 = 2

key = A[3]

for k=0 to 3-2-1

A[3-0] = A[3-0-1]

$A[3] \leftarrow A[2]^{7}$

A[2] = 5

A	1	2	3	4
	1	5	7	3

j=4 to 4

i=1

while $A[4]^{3} > A[1]^{1}$

i = 1+1 = 2

key = A[4]

for k=0 to 4-2-1

A[4-1] = A[4-1-1]

$A[3] \leftarrow A[2]^{5}$

A[2] = 3

A	1	2	3	4
	1	3	5	7

Question 4

a) (i) All heaps are full binary trees. - False

හැම heap එකම full binary tree එකක් වෙන්න ඕන නැහැ. Heap එකක් කියන්නේ complete binary tree එකක්. full binary tree → හැම node එකකම 0 නැත්නම් 2 children තියෙන්න ඕන. Complete binary tree → tree එක level by level, left to right පුරවමින් පිරී යන්න ඕන.

(ii) Heapify() is a recursive algorithm - TRUE

Heapify() function එක බොහෝවිට recursive ලෙස ලියනවා. එය කෙලෙසද node i එකේ සිට (downward) children check කරමින් heap property එක නඩත්තු කරනවා. Recursive approach එකේ swap වීමක් වුණොත් recursive call එකක් දාලා child node එක heapify() කරන්නේ.

(iii) Time complexity of Build_heap() is O(n). True

build_heap() කියන්නේ array එකක් heap එකක් බවට හැරවීම. surface level එකේ බලද්දී නම් O(n log n) වගේ පේන්න පුළුවන්. නමුත් actual complexity එක O(n) කියලා prove කරන්න mathematically පුළුවන්. ඒ බොහෝ heapify calls logarithmic cost එකක් දාවීමම total cost එක summation එකක් වීම.

(iv) Time complexities of heapsort() and mergesort() are equal True

දෙකම average සහ worst case වලදී O(n log n) වේලාවක් ගන්නවා. හැබැයි practically efficiency එක වෙනස්.

- Merge Sort - stable / predictable
- Heap Sort - in-place / but not stable

(v) priority queue is an application of heaps True

priority Que එකක තියෙන දේ highest priority item එකට access කිරීම Min heap / max heap use කරලා ඉතා efficiently implement කරනවා. insert, extract max/min වගේ operations heap structure එකක් හරහා O(log n) වේලාවට පුළුවන්.

Scanned with CamScanner

b) Naive String matching Algorithm ~~එකෙක්~~ එකෙදී pattern එක (length = m) ට text එක (length = n) තුළ සෑම්කොම possible position වලදී check කරලා match එකක් තියෙනවද බලනවා. Text එකේ position s = 0 to n-m වෙනකම් shift වෙනවා. හැම shift එකකම pattern එක text එකට match වෙවද් කියලා character by character check කරනවා.

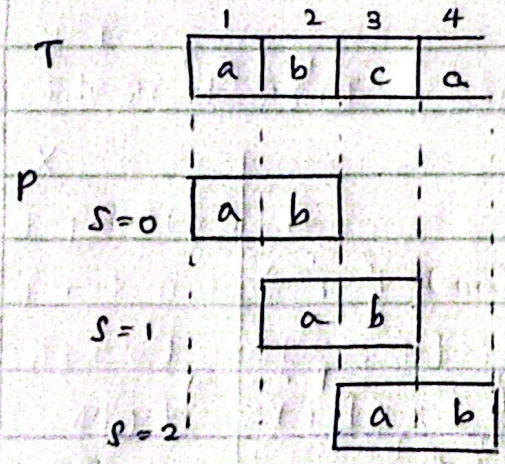
n = Text එකේ character count

m = pattern එකේ character count

Naive algorithm එකේ shift count = (n-m+1)

pattern එක text එකේ last match attempt position එක වෙනකල් n-m ඒ කියන position s = 0 to s = n-m total of (n-m+1) shifts

Number of Shifts = n-m+1



n-m+1

4-2+1

3 (0 → 2)

c)

(c) Text (T) = "abcdefg"
pattern (P) = "add"

T.length - P.length + 1
n - m + 1 = 7 - 3 + 1 = 5

pattern = "add"
length of text (T) = 7
length of pattern = 3

KMP algorithm එකෙන් optimized shifting එකක් කරනවා, නමුත් මෙහි pattern එක කිසිම තැනක match නොවේ. එබැවින් full shift count = 5 ලෙස ගණන ලැබෙයි.

shift	compare substring	match
0	abc	x
1	bcd	x
2	cde	x
3	def	x
4	efg	x

(d) Naive Algorithm එක worst case අවස්ථාවේදී pattern එක text එකේ එක් එක් shift කරමින් check කරනවා. performance:

Comparisons per shift: up to m
Number of shifts: n - m + 1
Total comparisons (worst case)

$$O(m(n-m+1)) \simeq O(mn)$$

When worst case happens:

Text = "aaaaa"...
Pattern = "aaaab"

මේ වගේ situation වල pattern එක match නොවී කිට්ටුවට close match කරනෙ තියා comparisons වැඩි වෙනවා.

(e)

```
MAX-HEAPIFY (A, i)
  l = LEFT (i)
  r = RIGHT (i);
  if l ≤ A.heap_size and A[l] > A[i]
     largest = l;
  else largest = i;
  if r ≤ A.heap_size and A[r] > A[largest]
     largest = r;
  if largest ≠ i
     exchange A[i] with A[largest]
       MAX_HEAPIFY (A, largest)
```

(i) if $l \leq A.heap\_size$ and $A[l] > A[i]$

l කියන left child index එක valid ද,
$A[l] > A[i]$ නම් left child එක parent එකට වඩා විශාලයි නම් කියලා තීරණය කරලා largest = l කියලා assign කරනවා. මෙය max-heap property එක satisfy කරන්න වගේ කෙරෙනවා.

To check whether the left child exists and is larger than the current node so we can mark it as largest and possibly swap to maintain max-heap order

(ii) if $r \leq A.heap\_size$ and $A[r] > A[largest]$

r කියන right child index valid ද,
$A[r] > A[largest]$ නම් → right child එක is even larger than both parent and left, ඒ නිසා largest = r කියලා update කරනවා.
මෙයත් max-heap property එක තහවුරු කිරීමට සහායවෙනවා.

To check whether right child exists and is greater than the current largest so it can be marked as largest to maintain max-heap structure