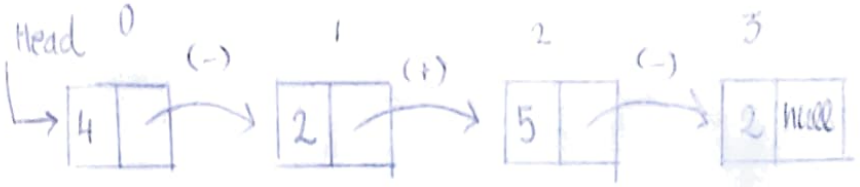
Linked List

1) 

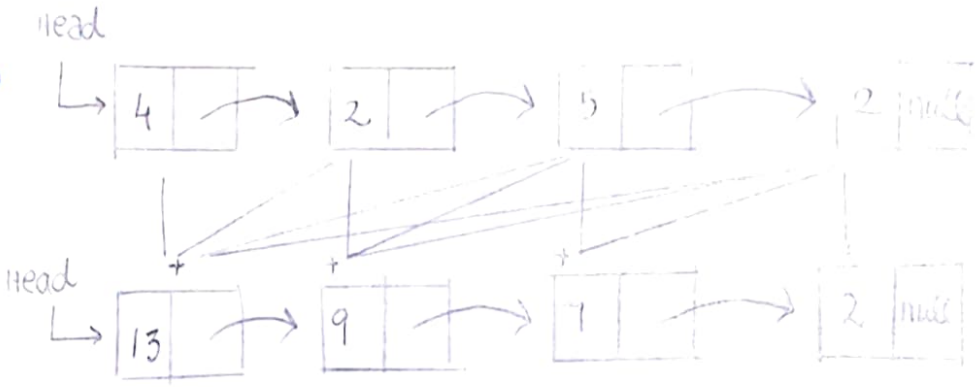
algorithm rt_sum(head)

1) If (head == null)
   return 0

2) else {
   ~~while~~
   temp = head
   count = 0
   sum = 0
   while (temp != null) {
      if (count % 2 == 0) {
         sum += temp → data
      }
      else {
         sum -= temp → data
      }
      count++;
      temp = temp → next
   }
3) return sum }

4) end.

2)



Basic logic : for each node sum up the value of current node and the nodes that come after the current node

Algorithm add (head)

1) current_node = head

2) temp = head

~~3) for (; head != null~~

this loop is to iterate through the linked list

3) for (; current_node != NULL ; current_node = curr_node →next) {

to iterate through the next set of nodes that are after the current node

~~for (temp = head →ne~~

for (temp = curr_node →next ; temp != null ; temp = temp →next) {

current_node →data += temp → data

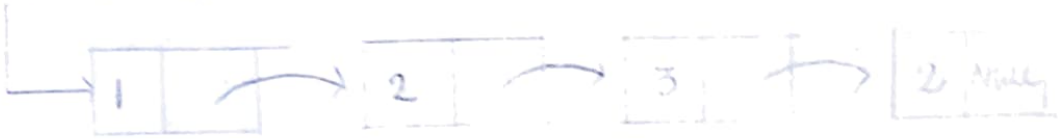
}

}

4) or return head

5) end.

3) head x = 2



RESULT:

Algorithm delete_var (head, x)

1) IF (head == NULL) // empty list
   return 0

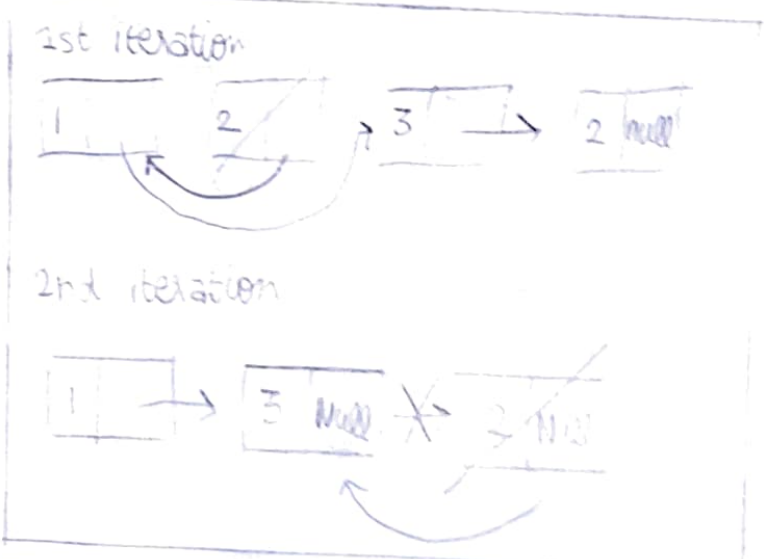
2) while (head → data == x) // delete in front
   head = head → next

3) temp = head

4) while (temp → next → next != NULL) // iterate till last before node
   if (temp → next → data == x) {
      temp → next = temp → next → next
   }
   else {
      temp = temp → next
   }
}

5) Return head

6) end

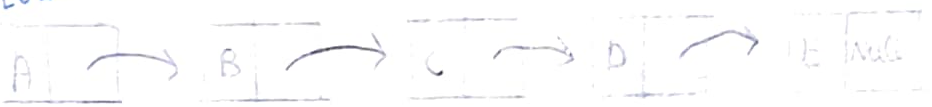


4)

Eg1:

ANS:

Eg2:



ANS:

TRACING the algorithm: for eg 2

prev = null

first = head

(1)st iteration

∵ first = head (A) & first → next = B

while loop is true

first = A

second = B

prev = NULL

start = head → next

first → next = second → next

⇒ A → C → D → E

first

second → next = first

B → A → C → D → E

second first

done for next iteration purpose ⇐ prev = first , first = prev → next

B → A → C → D → E

second prev first

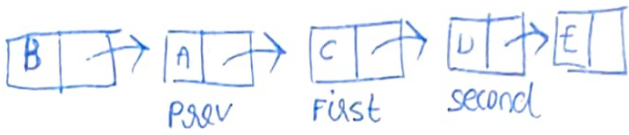
(2)nd iteration

1) second = first → next

first = [C|]

⇒ second = [D|]

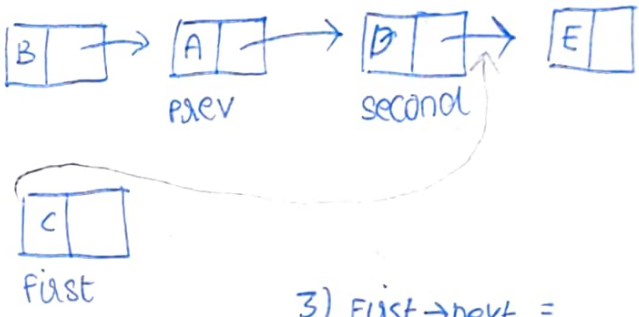
second = [B|]



Prev = [A|]

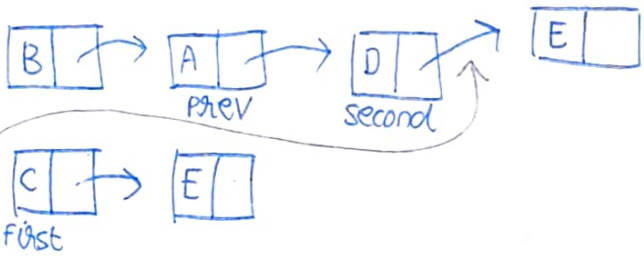
∵ (Prev → next != null)

2) Prev → next = second

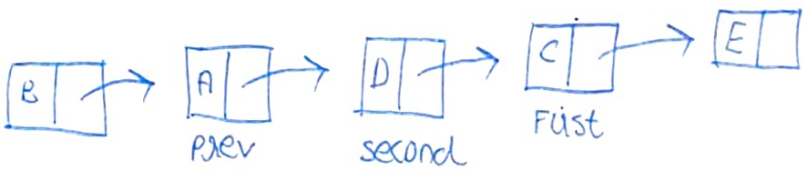


3) First → next = second → next

~~second → next = first~~



4) second → next = First



5) For next iteration Prev = First

First = Prev → next

(3)rd iteration is false because First → next is null

So return start

Algorithm swap (head)

1) if (head == NULL || head→next == NULL) {

return head; }

2) start = head →next

3) prev = null

4) first = head

5) while (first != null && first →next != null) {

second = first →next;

if (prev != null) {

prev →next = second

}

first → next = second →next

second →next = first

prev = first

first = prev →next

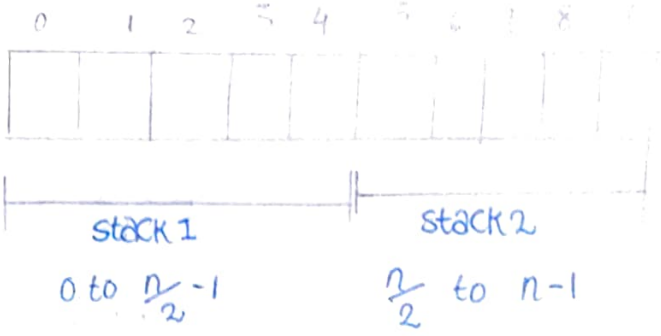
}

6) return start

7) end.

Stacks

1) Implementing 2 stacks in a single array



top = 0 (starts from) top = n/2

end = n/2 - 1 end = n-1

Algorithm 2stacks()

1) initialize an array of size n

2) top1 = -1

3) top2 = n/2 - 1

4) to Push into first stack do the following operations

(i) check if top1 != n/2 - 1
top1++;
Push (element)

(ii) else Print stack 1 overflow

5) to POP from first stack

(i) if (top1 != -1)
POP()
top1--;

(ii) else Print stack 1 underflow

6) to Push into second stack

(i) if (top2 != n) top2++; Push (element);

(ii) else Print stack 2 overflow

7) to POP from stack 2

(i) if (top2 != n/2)
POP()
top2--;

(ii) else Print stack2 underflow

8) end

5) Convert infix to postfix in stacks

Input: A+B*C+D Output: ABC*+D+

LOGIC:

Precedence of operators

1) ( )	associativity
2) ^	R to L
3) / *	L to R
4) + -	L to R

A+B*C+D

1) A+BC*+D

2) ABC*++D

3) ABC*+D+

RULES FOR CONVESION using stack

→ Write operand directly as output

→ If operator, perform push operation based on the following conditions

(i) If the precedance of the incoming operator is higher than the top operator in the stack then push it into stack

(ii) else POP and keep checking again

(iii) If there is a same precedence like + and - check if LtoR or RtoL. If L to R pop and check else push

INPUT A+B * C+D

Input	stack	postfix
A	-	A
+	+	A
B	+	AB
*	* > + * +	AB
C	* +	ABC
+	; + < * POP +	ABC* +
+	+ = + + L to R so POP	ABC*+
D	+	ABC*+D
empty	pop	ABC*+D+

Algorithm intopost (expression)

1) Iterate through every character and perform the following operations

2) If the character is an operand display as output

3) else

Check the Priority of the current operator and the operator in the top of the stack

4) If the incoming priority is greater, then push operator into stack and goto step 6

5) else pop from stack and display as output and repeat step 3

6) Repeat from step 2 for every character

7) If the expression is fully completed and the stack is not empty, pop out all the operators from the stack

8) end.

3) well formedness of brackets

{()} ✓ {(}) ×

Algorithm brackets (expression)

1) Iterate through every character of the expression

2) If the character is an opening bracket, then push into stack

3) Else if the character is a closing bracket check if the top of the stack is an equivalent opening bracket

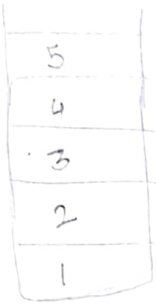
4) If yes then POP the top of stack and continue from step 2

5) If not, then return that the brackets are not balanced and exit

6) If the expression is fully completed and stack is emptied return that the brackets are balanced.

7) end.

4) Range count problem



var = 2

Output → 4

because 4 numbers in the stack are greater than or equal to 2

Algorithm range_count (stack, var)

1) count = 0

2) while (isEmpty (stack == False))

if (stack[top] ≥ var) {

count ++;

}

top --;

3) return count;

6) end