DATE: __/__/____

Data Structures

Assignment - 01

Submitted To:

Dr. Inayat ur Rehman

Submitted By:

Muhammad Rizwan Shafiq

SP24-BCS-069

BCS-3B

DATE : ___/___/_____

Singly Linked List

TASK-01

Reusable Code Snippets:

① List is Empty:

```
If(first==NULL){
    cout<<"List is Empty "<<endl;
    return;
}
```

② Only one node in list:

```
If(first->next==NULL){
    cout<<"Only one node in list: "<<first->data<<endl;
    return;
}
```

Algorithm:

Step 1: Check if list is empty.
Step 2: Check if only one node.
Step 3: Initialize temp to last node: Nodetype *temp= last;
Step 4: Loop until temp decrements to NULL before start:

```
while(temp!=NULL){
    cout<<temp->data<<" "; // print current node's data
```

4a: Find Previous node of temp. // nested while

```
Singly Nodetype *p = first;
Nodetype *prev = NULL;
```

DATE __/__/____

```
while (p->next != temp && p->next != NULL){
        prev = p;
        p = p->next;
    }
```

4b: Move temp to previous node.

```
    temp = prev;
}

cout << endl;
}
```

Diagram:

Let's assume:- |10| → |20| → |30| → |40| → NULL

Step ②: temp → |40| → NULL

print = 40

Step ② find previous of 40

prev p (loop will continue till p reaches temp)

|10| → |20| → |30| → |40| → NULL

temp

→ prev found = 30

→ we decrement temp to 30 by temp = prev.

→ condition is checked for outer loop:

→ in the begning of while loop it prints temp which is now 30.

output: |40| → |30|

Step ② is repeated untill temp becomes NULL in case:

p

prev → NULL |10| → |20| → |30| → |40| ← temp

} this won't enter the nested while loop & temp decremented to NULL which stops the outer loop as well

DATE: __/__/____

Final Output: |40| → |30| → |20| → |10| → NULL

NOTE: With this algorithm, the list is only printed in reverse and there is no physical change in nodes location of list.

TASK 02

Algorithm:

Step 1: Check if list is Empty.
Step 2: Check if only one node in list.
Step 3: Initialize pointers to keep track.

```
Nodetype *prev = NULL;
         *current = first;
         *next = NULL;
```

Step 4: Traverse the list untill it has been reversed completely & current becomes NULL.

```
while (current != NULL) {
    next = current->next;        // Store next node
    current->next = prev;        // Reverse list
    prev = current;              // Move prev
    current = next;              // Move current
}
```

Step 5: Update heads of list:

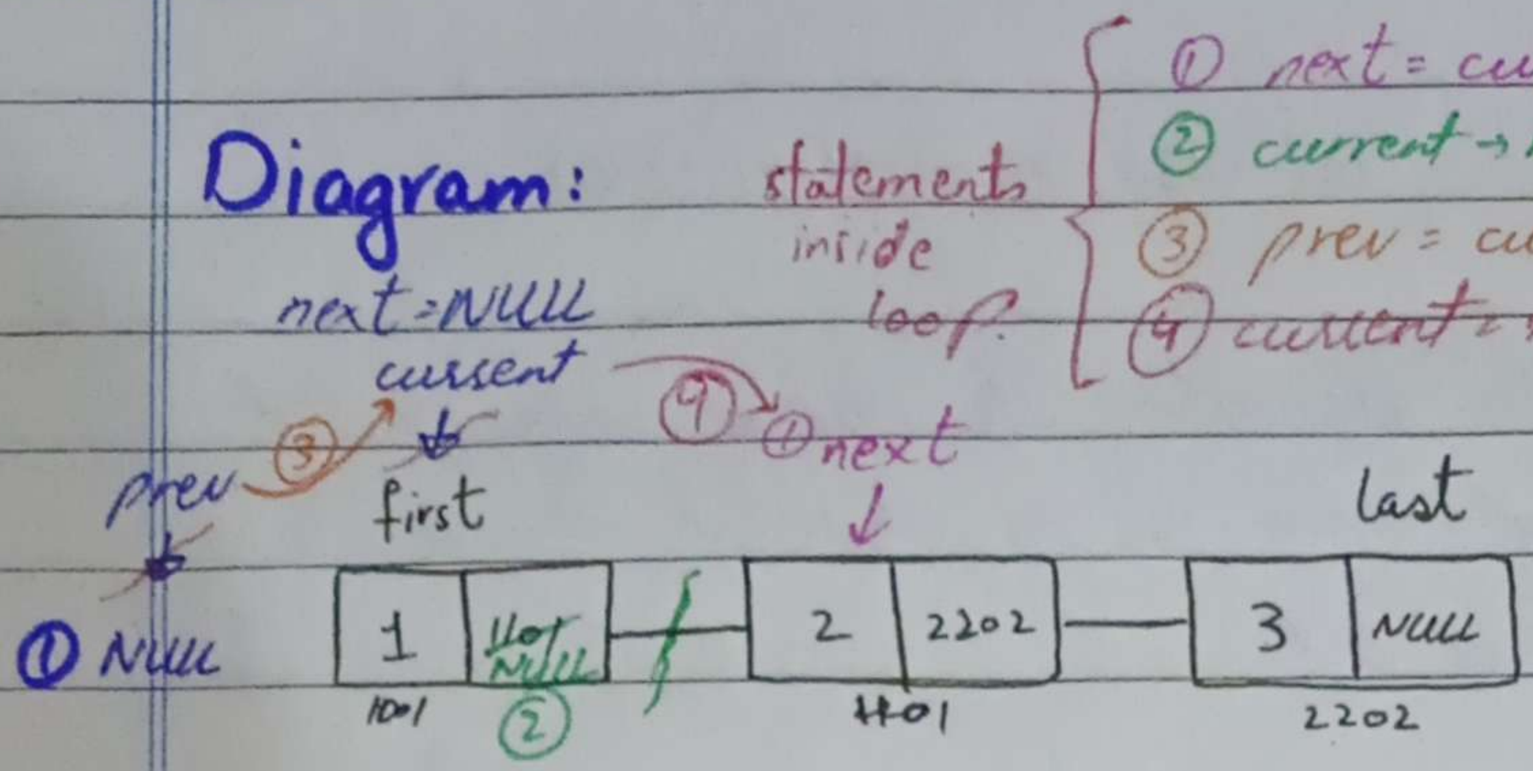
```
last = first;
first = prev
```

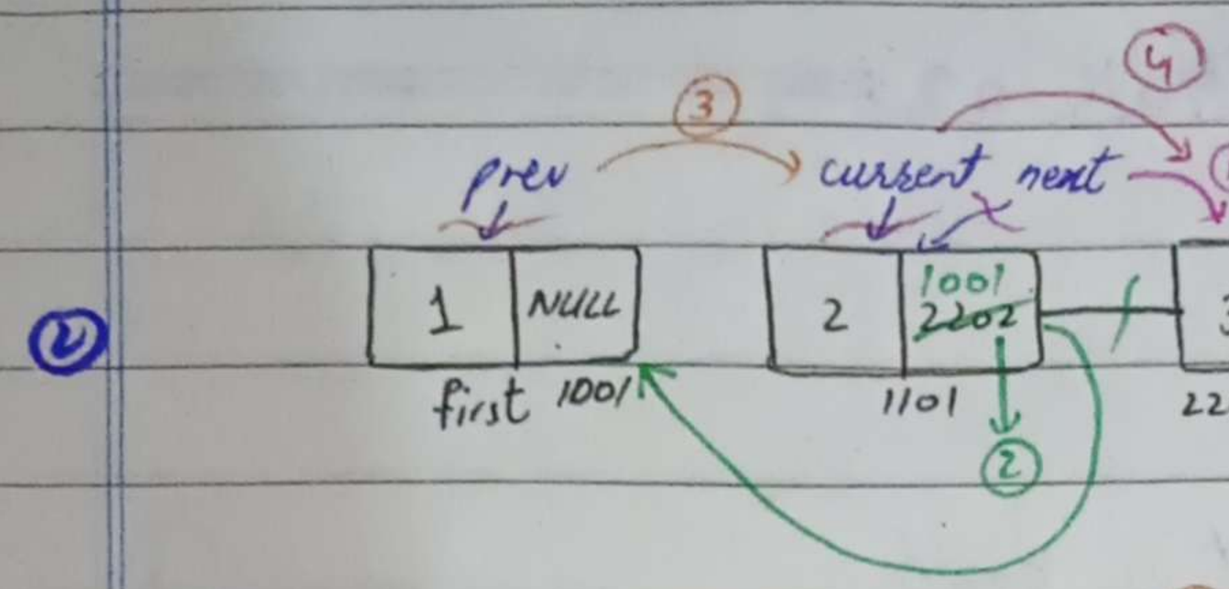
DATE: __/__/____

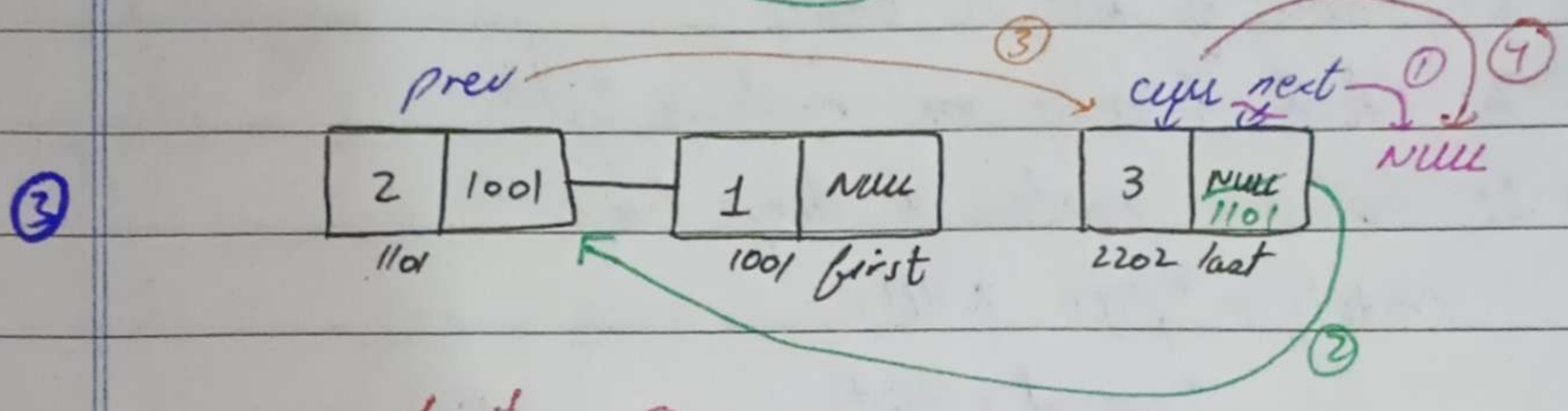
Diagram:

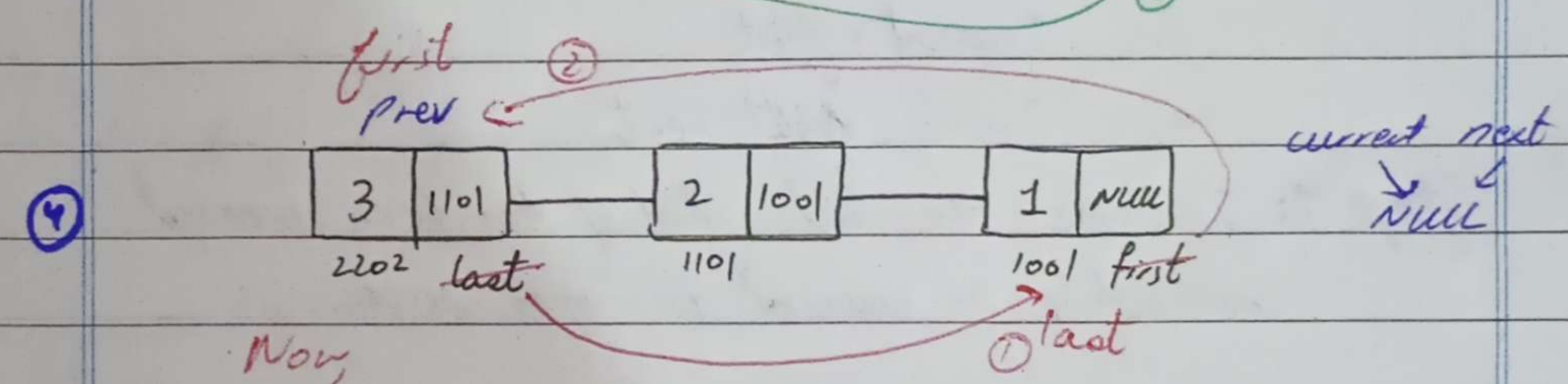
statements inside loop:

① next = current → next;
② current → next = prev;
③ prev = current;
④ current = next;





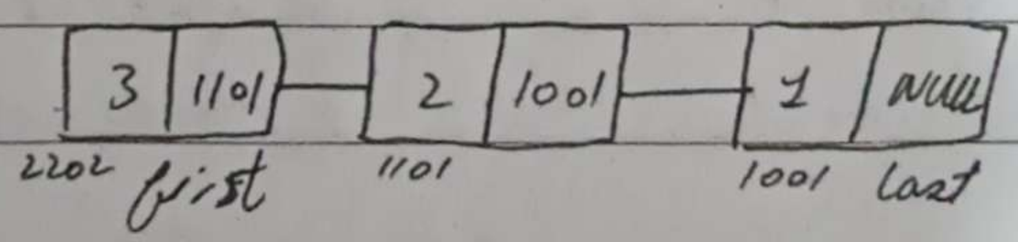




Now,

① last = first;
② first = prev;

Output:



DATE: ___/___/_____

TASK 03

Algorithm:

Step 1: Check if list is empty.

Step 2: Check if only one node in list.

Step 3: Set a current pointer from first.

Step 4: loop while curr is not NULL.

- Set 'prev' to 'current' & 'next' to 'current → next'.
- loop while next is not NULL.
  - if next → data is equal to current → data.
    - Duplicate found:
      - Skip duplicate node: prev → next = next → next;
      - update last if node to be deleted is last.
      - Delete next;
    - Move next = prev → next;
  - ELSE (no duplicate):
    - Move prev = next & next = next → next;
  - Move current = current → next;

End when all duplicates removed.

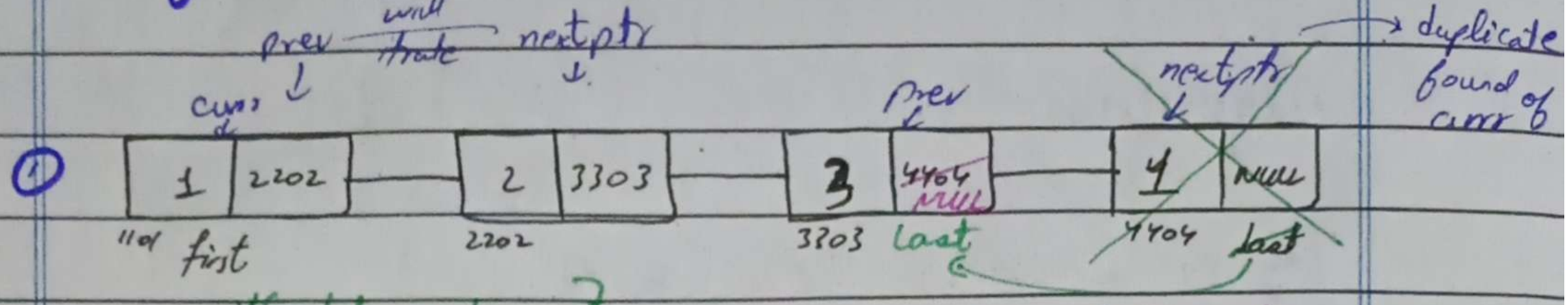
Code step 4:

```
Nodetype *curr = first;
while(curr!=NULL && curr→next!=NULL){ //for each node
    Nodetype *prev = curr, *nextptr = curr→next;
    while(nextptr!=NULL){ // this checks duplicate
        if(nextptr→data == curr→data){ // duplicate found
            prev→next = nextptr→next;
            delete nextptr;
            nextptr = prev→next;
        }else{ prev = nextptr;
            nextptr = nextptr→next;
        }
    }
    curr = curr→next; // Move to next node
}
```

```
if(nextptr→next==NULL)
    last = prev;
} //update last
  if node to be
  deleted is last
```

DATE ___/___/_____

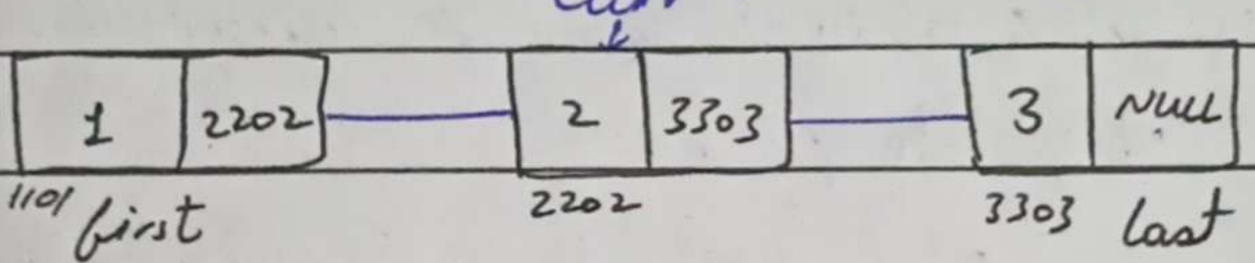
Diagram: for curr = first node; prev = curr; nextptr = curr->next




```
if(nextptr->next==NULL){
    last = prev
}
```

→ ① prev->next = nextptr->next;
② deleted nextptr
③ next → becomes NULL.

In outer loop → curr = curr->next.



No duplicate found in this case as
(nextptr->data != curr->data).

TASK - 04

⇨ Floyd's Cycle/Tortoise & Hare Algorithm

Algorithm:

Step 1: Check if list is Empty

Step 2: check if only one node.

Step 3: Initialize pointers.

- slow = first (Moves one step)
- fast = first (Moves two steps

Step 4: Traverse until fast & fast->next not null.

- Move slow 1 step & fast 2 steps.
- if (slow == fast) → loop exist. & return true.
- If no loop found. return false

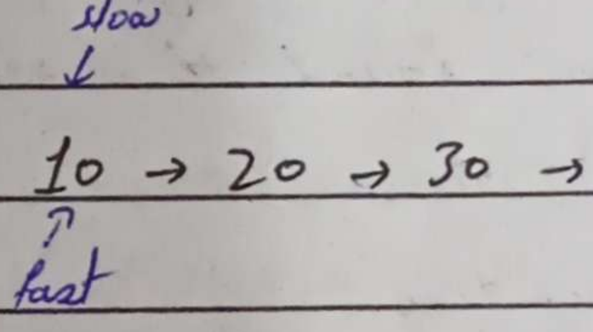
DATE: __/__/____

Code:

```
Nodetype * slow = first, *fast = first;
while (fast != NULL && fast->next != NULL) {
    slow = slow -> next;        // one step
    fast = fast->next->next;    // two steps
    if (slow = fast) {
        cout<<"loop Exist"<<endl;
        return true;
    }
}
cout << "No loop found" << endl;
return false;
```

Diagram:

Case 1:

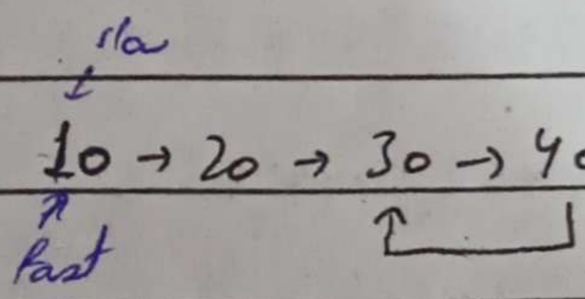


First Itration: slow = 20 fast = 30

2nd Itration slow = 30 fast = NULL (list ended)

Output: false. ("No loop").

Case 2:



Itration 1: slow = 20 fast = 30

Itration 2: slow = 30 fast = 30 (slow = fast).

~~Itration 3:~~

Output: True ("loop Exist")

DATE : ___ /___ /_____

TASK -05

Algorithm:

Step 1: Check if list is empty.

Step 2: Check if only one node in list.

Step 3: If Both keys are same, then return.

```
If (key1 == key2) return;
```

Step 4: **Initialise pointers.**

```
Nodetype *prevX = NULL; *prevY = NULL, *x = first, *y = last.
```

Step 5: Find x & y.

```
while (x && x->data != key) { prevX = x; x = x->next; }
while (y && y->data != key) { prevY = y; y = y->next; }
```

Step 6: If any one not found then return.

```
If (!x || !y) { return; }
```

Step 7: Adjust prev pointers.

```
If (prevX != NULL) prevX -> next = y;
else first = y;
If (prevY != NULL) prevY -> next = x;
else first = x;
```

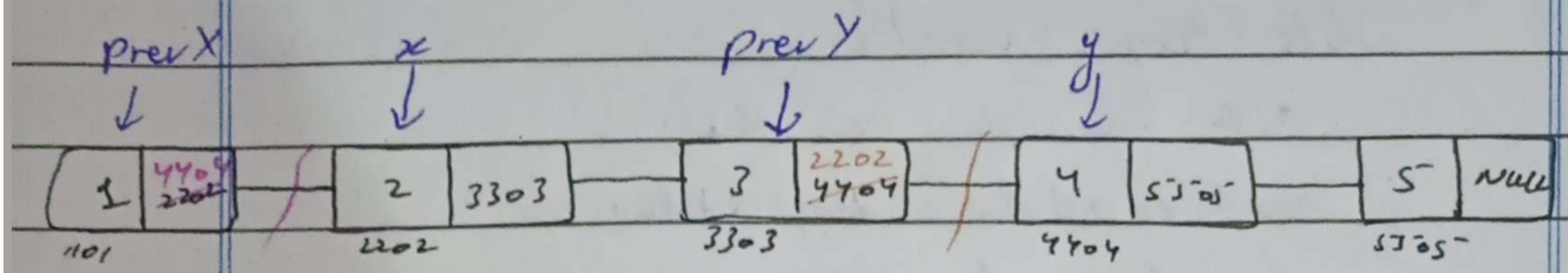
Step 8: Swap the next pointers

```
Nodetype *temp = x->next;
x->next = y->next;
y->next = temp;
```

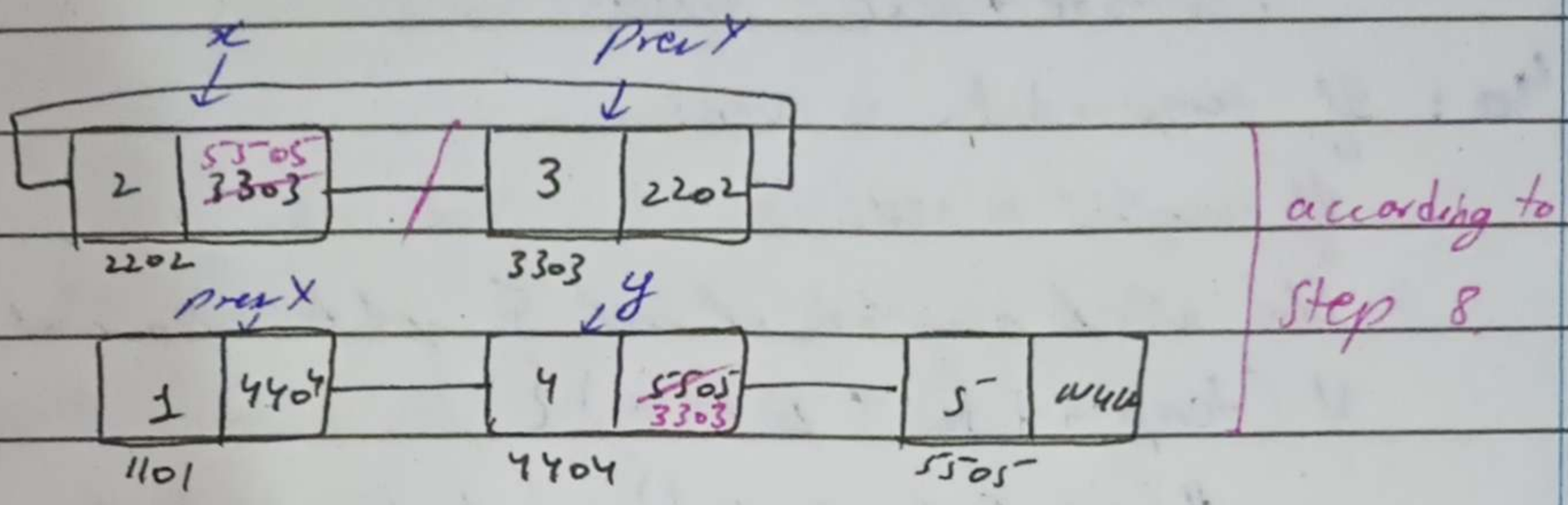
DATE: __/__/____

Diagram: let's assume keys are 2 and 4. then lets begin from step 7 to visualize it.

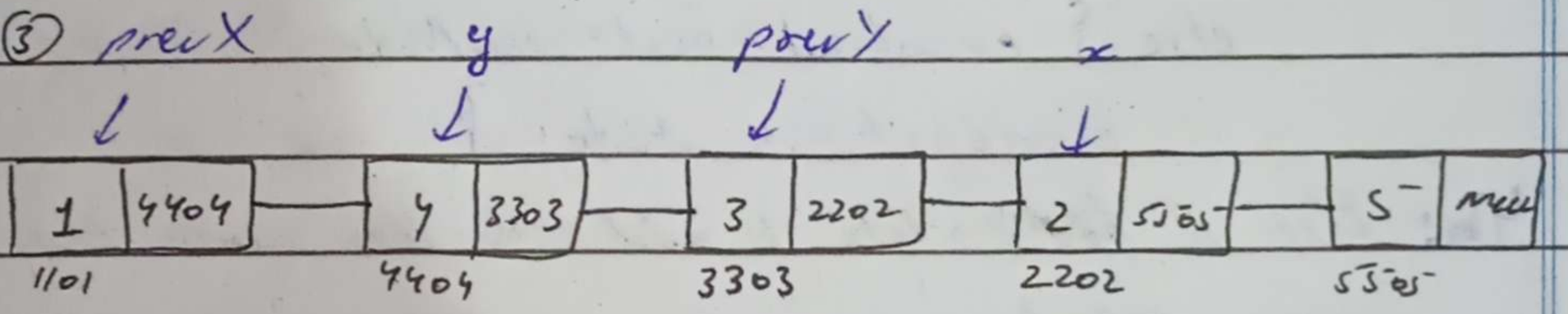
(1)



(2)



(3)



TASK - 06

Algorithm:

let's say that we have the pointers declared globally.

Node *first = NULL, *last = NULL;

*evenfirst = NULL, even last = NULL;

*oddfirst = NULL; oddLast = NULL;

DATE : __/__/____

Step 1: Check if list is empty.

Step 2: Check if only one node

Step 3: Initialize a temp pointer: `Node *temp = first;`

Step 4: Traverse and split:

```
while (temp != NULL) {
    Node *newNode = new Node;
    newNode → data = temp → data;
    newNode → next = NULL;
```

4a: If temp → data is even:

If evenfirst is NULL,

Else attach newNode at end & update evenlast.

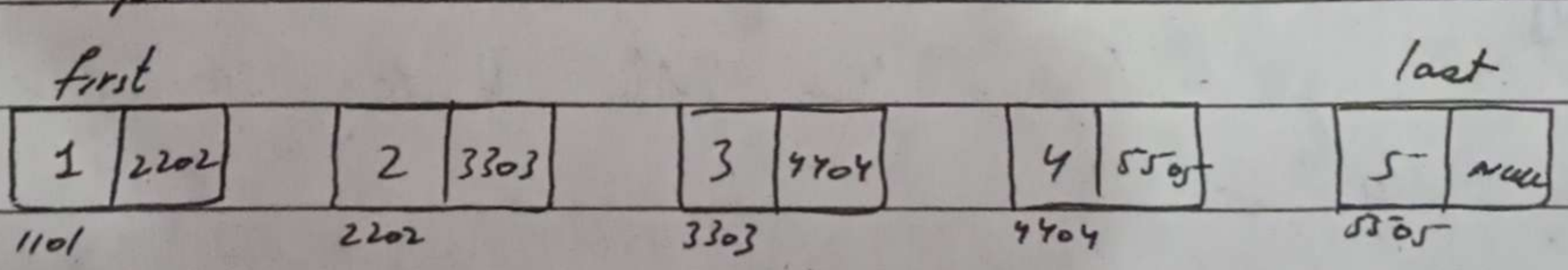
```
If (temp → data % 2 == 0) {
    If (evenFirst = NULL) { evenfirst = evenlast = newNode; }
    else { evenLast → next = newNode;
           evenLast = newNode; }
```

4b: Else if temp → data is odd, do the same thing using oddFirst.

Step 5: Move temp to temp → next;

Diagram:

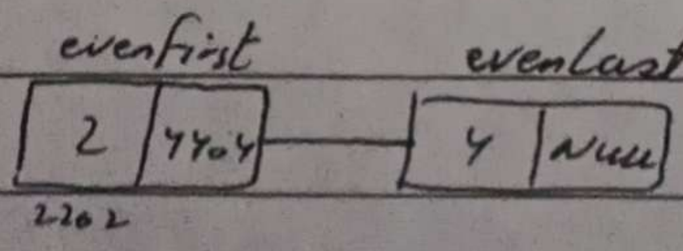
Example list.

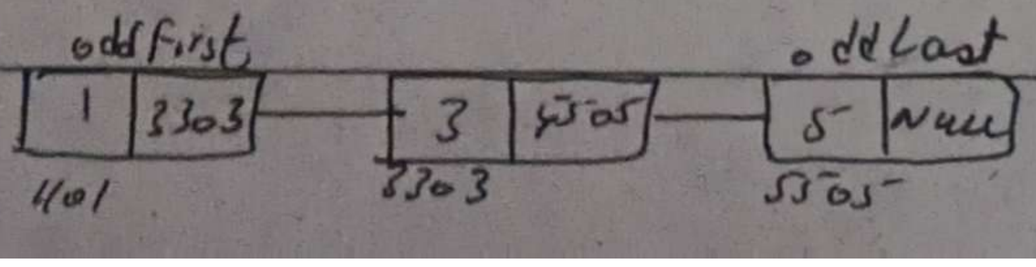


Step 1 ⇒ temp = 1 (odd). → added to odd list.

Step 2 ⇒ temp = 2 (even) → added to even list.

Final Outputs:





DATE : ___/___/_____

TASK - 07

Step 1: Check if list is Empty

Step 2: check if only one node.

Step 3: Find middle of list.

```
Nodetype *slow, first, *fast = first; *firstHalfEnd = NULL;
while (fast != NULL && fast->next != NULL) {
    firstHalfEnd = slow;
    slow = slow->next;
    fast = fast->next->next;
}
```

Step 4: Break the list into halves

```
firstHalfEnd->next = NULL;
```

Step 5: Reverse first half.

```
Nodetype *prev = NULL, *curr = first, *next = NULL;
while (curr != NULL) {
    next = curr->next;
    curr->next = prev;
    prev = curr;
    curr = next;
}
firstHalfEnd = first      // last node of reversed first half
*first = prev;            // new head of first half
```

Step 6: Reverse 2nd half

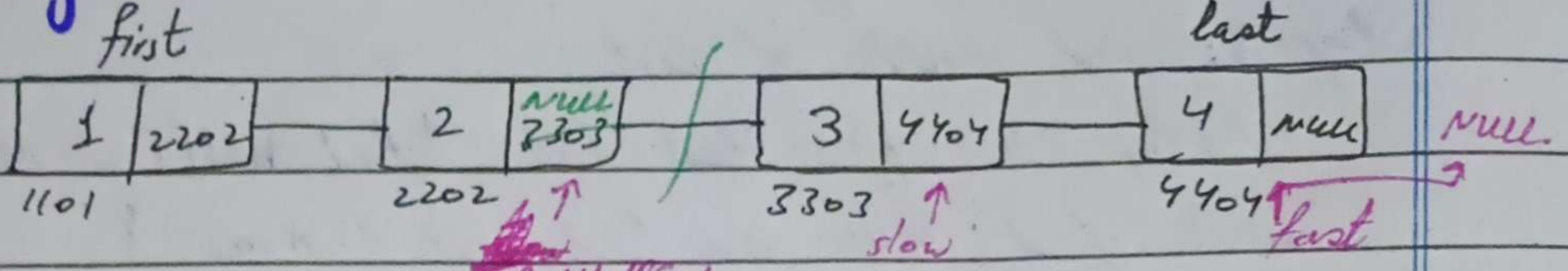
```
prev = NULL, curr = slow, next = NULL.
while (curr != NULL) {
    next = curr->next;
    curr->next = prev;
    prev = curr;
    curr = next;
} last = slow (// updated last head)
```

DATE: __/__/____

Step 7: Connect first half to 2nd half.

firstHalfEnd → next = prev;

Diagram: Lets assume.

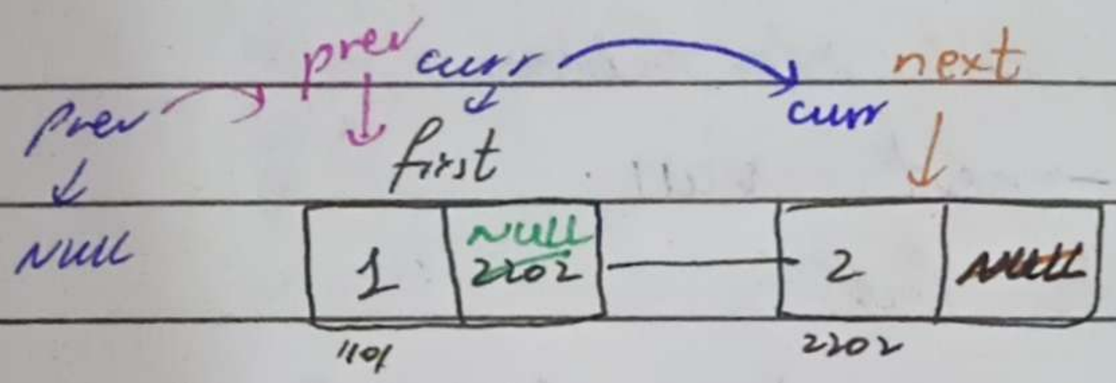


According to step 3:→ loop will end at:→

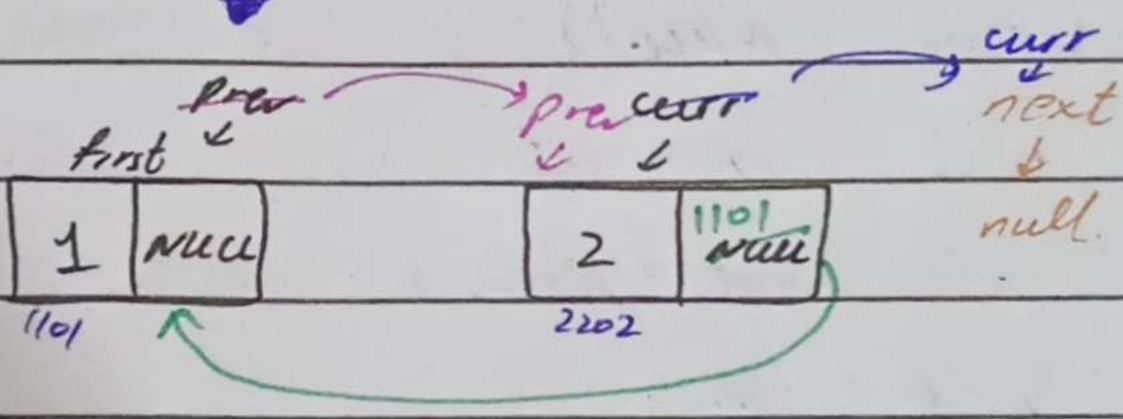
Now by step 4 → shown with green

Now by step ⑤ →

Iteration 1:- ① next = curr→next ② curr→next = prev ③ prev = curr ④ curr = next

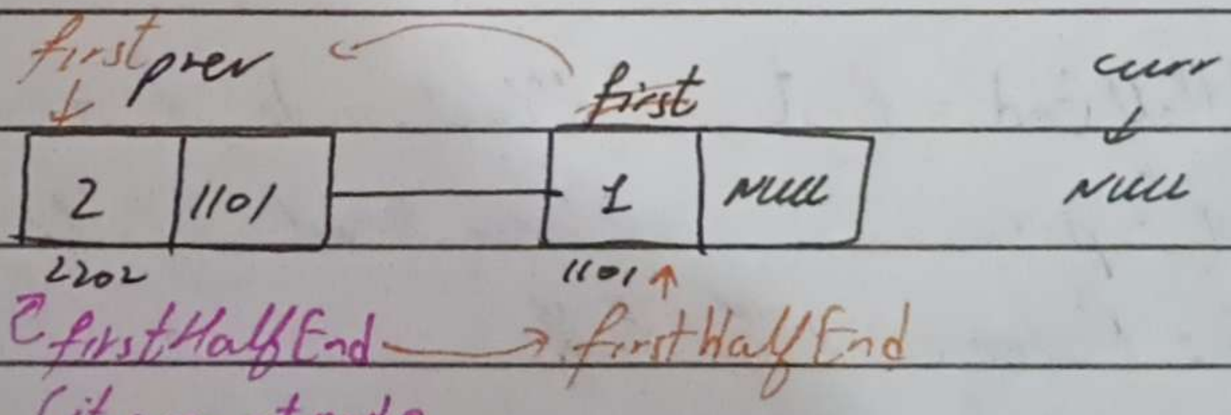


Iteration 2:



output.

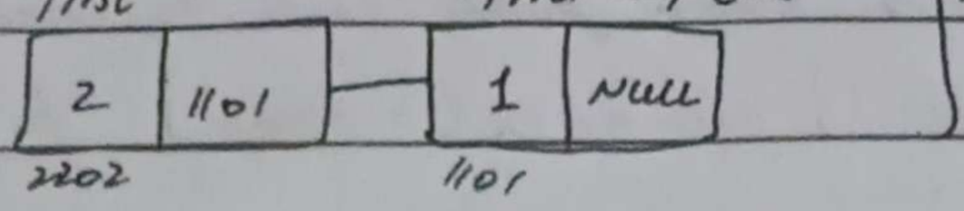
as curr = NULL.



According to step end. { firstHalfEnd = first ; first = prev } ⑤

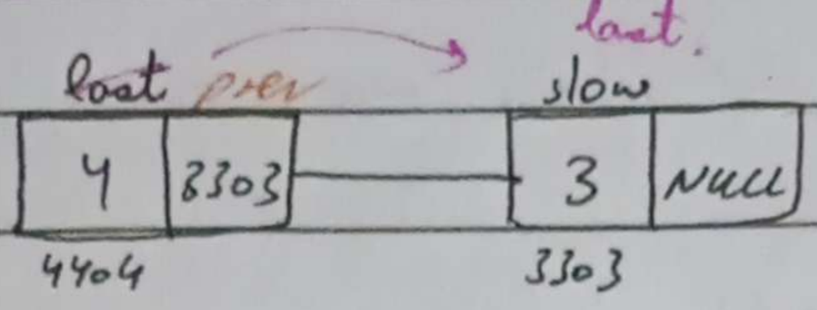
DATE: __/__/____

output:-



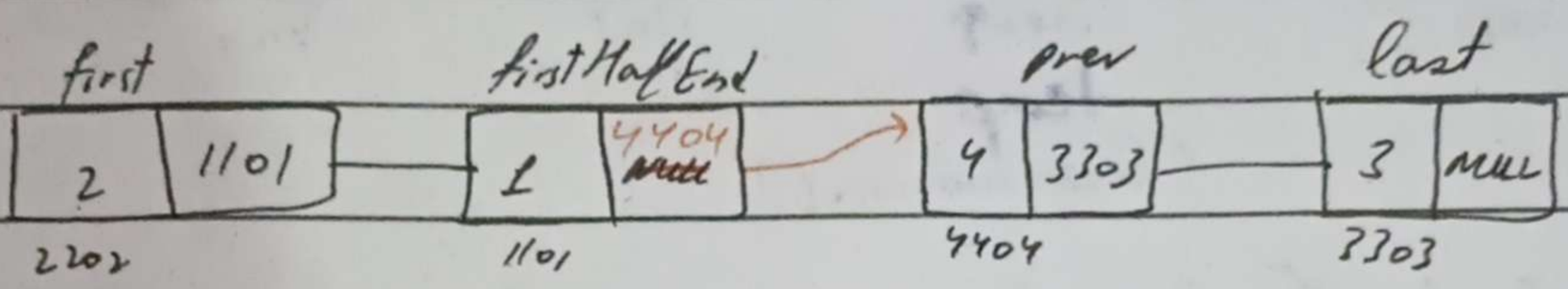
first half of list.

Same process will carry for 2nd half of list and we'll get:-



last = slow

Now by step 7, we join the lists. firstHalfEnd → next = prev



TASK - 08

"Pairwise Swap of Nodes"

ALGORITHM:

Step 1: Check if list is Empty

Step 2: Check if only one node in list.

Step 3: Initialize two pointers:

prev = NULL → to keep track of prev node of current pair

curr = first → to start from head.

Step 4: Update first to first → next as it will become new head after swap.

first = first → next;

Step 5: Loop until current & curr → next aren't NULL.

- Set temp = curr → next;

swap the pair: if (curr → next == last) { last = curr }

curr → next ⇒ temp → next (skip temp to connect to next pair)

DATE : __/__/____

temp->next = curr. (temp points back to curr, completing swap)

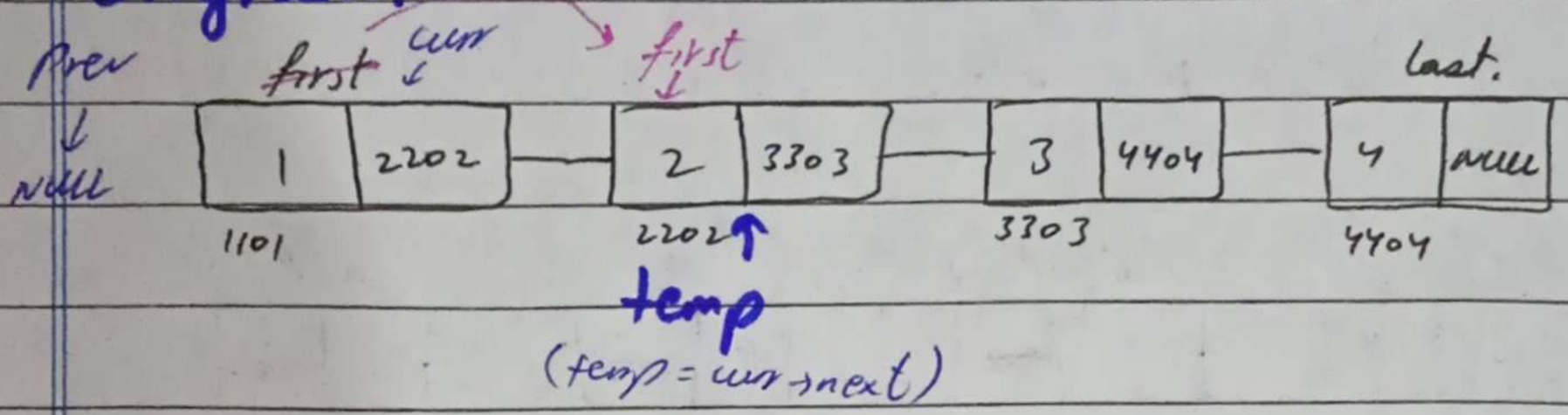
→ if (prev != NULL){ prev->next = temp } to connect with previous swapped pair.

prev = curr;

curr = curr->next. ~~if(curr->next = last){ last = curr }~~

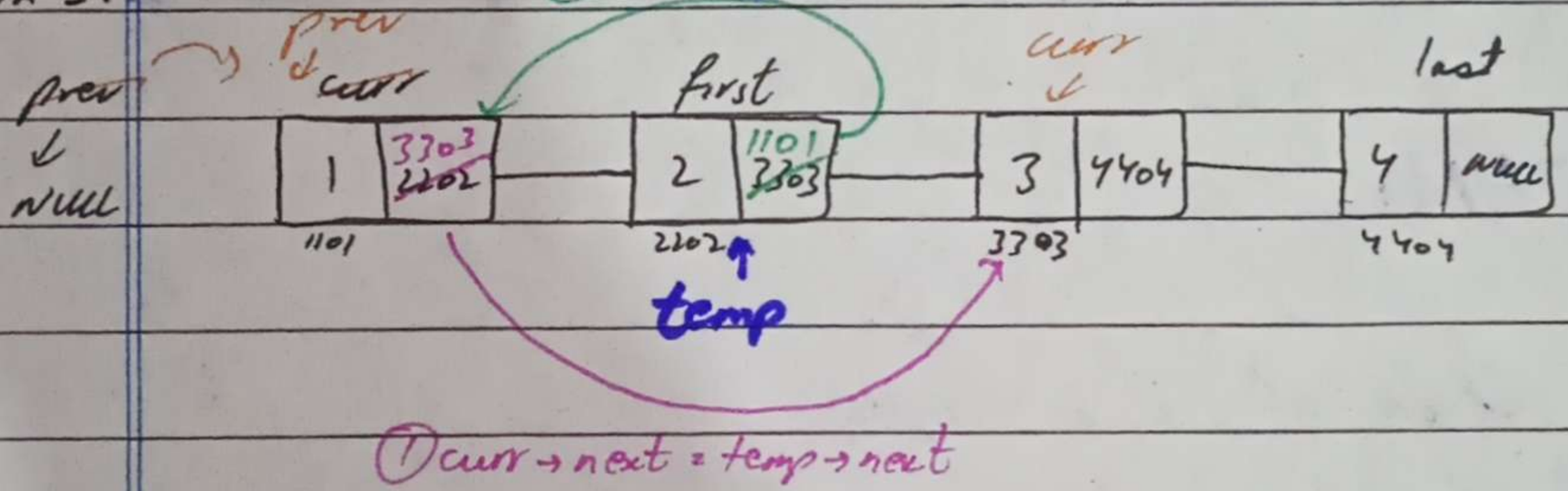
After loop ends, All nodes are swapped.

Diagram: first = first->next



Now Inside the loop:

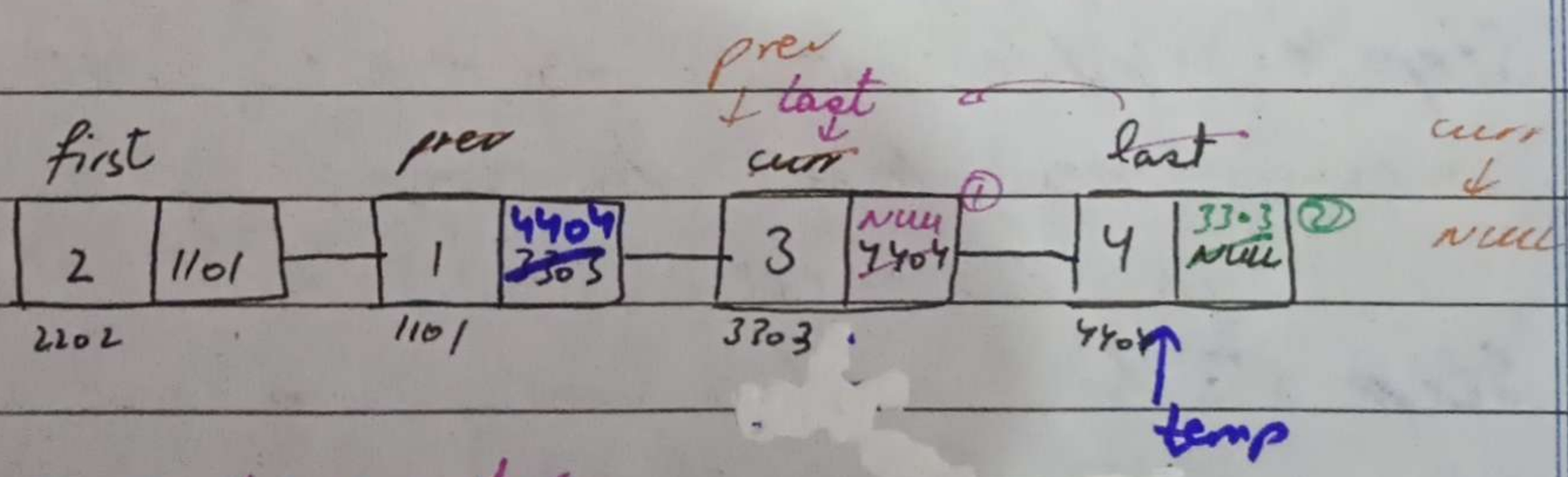
Iteration 1:



③ if condition → not execute

③ prev = curr; curr = curr->next;

Iteration ②



Before swap if is executed.

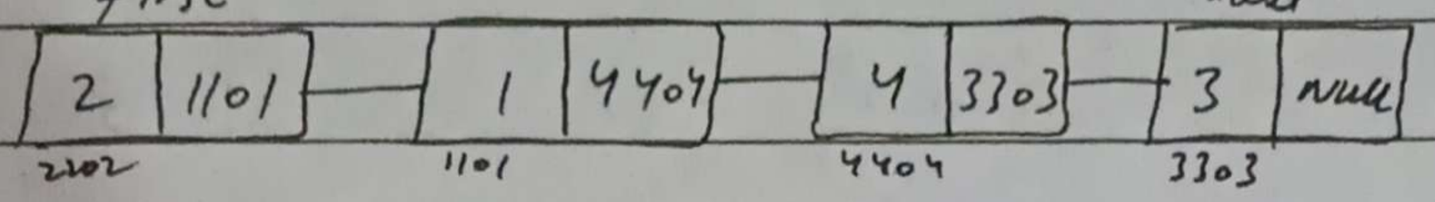
① if (curr->next == last) last = curr

② if (prev != NULL) prev->next = temp

→ After this curr becomes NULL so loop ends.

DATE : __/__/____

final output.



Circular List

TASK -1

"Josephus Problem"

Step 1: Check if list is empty.

Step 2: Check if only one node. if(first->next=first) return;

Step 3: Initialize pointers

*prev = last, curr = last -> next;

Step 4: loop untill one node left.

while (last != last -> next){

4a: count = 1;

4b: if(count == k){ // eliminate k-th person

prev -> next = curr -> next

if(curr == last) last = prev // update last if last node removed.

delete curr;

curr = prev -> next;

count = 1; // Reset count

} else {

prev = curr;

curr = curr -> next;

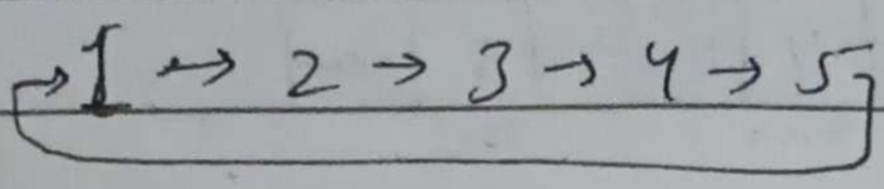
count ++;

}

}

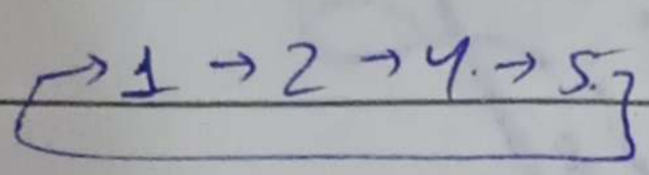
DATE: __/__/____

Diagram: (پہلا پیپر میں تھک گیا تھا سر جی!)

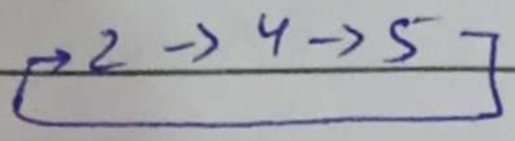


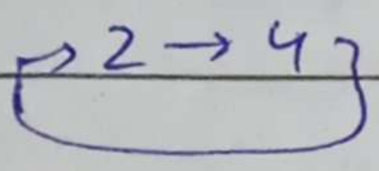
lets assume k = 3

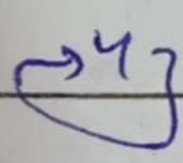
→ count starts from 1 and when reaches 3



now count = 1 again & curr = 4.



~~3~~ 



only one left so it wins.

Doubly List

TASK - 01

ALGORITHM:

Step 1: Check if list empty. if(first == NULL) return;
Step 2: check if only one node. if (first == last) return;
Step 3: count total nodes.

```
Nodetype * countptr = first.    int count = 0;
while (countptr != NULL) {
        count++;
        countptr = countptr → next;
}
```

DATE: __/__/____

Step 4: If list has only 2 nodes then just swap the two nodes and return.

```
if (count == 2) {
        Nodetype *temp = first;
        first = last;    last = temp;
        first -> prev = NULL;    first -> next = last;
        last -> prev = first;    last -> next = NULL;
        return}
```

Step 5: Loop to swap alternate nodes, by looping from i=2 to count/2 with a step of 2.

- find i-th node from start (left)
- find i-th node from end (right).
- If (left = right) -> break;
- if alternate node handle case
- else do as is normal.

```
Nodetype *left, *right;
for(int i=2, i<count/2, i+=2) {
        left = first;    right = last;
        for (int i, j<i, j++) {
                left = left -> next;
                right = right -> prev;
        }
        if (left == right) break;
        if (left -> next == right) {    // Adjacent node case
                left -> prev -> next = right;
                left -> next -> prev = left -> prev;
                right -> prev -> next = left -> next;
                right -> next -> prev = left;
        }
```

DATE : __/__/____

left → prev → next = right;

left → next → prev = right;

right → prev → next = left;

right → next → prev = right;

}

Nodetype * leftPrev = left → prev ;

Nodetype * tempNext = left → next;

x right → prev =

Nodetype * leftPrev = left → prev; *rightNext = right → next;

// right → prev → next = right;

// left → next → prev = le

right → prev = leftPrev

left → next = rightNext;

right → next = left;

leftPrev → prev = right;

} else {

left → prev → next = right;

left → next → prev = right;

right → prev → next = left;

right → next → prev = left;

Nodetype * tempPrev = left → prev ; *tempNext = left → next;

left → prev = right → prev;

left → next = right → next;

right → prev = tempPrev;

right → next = tempNext;

}

}

DATE: __/__/____

Diagram:-

Example List:-

1 ↔ 2 ↔ 3 ↔ 4 ↔ 5 ↔ 6 ↔ 7 ↔ 8 ↔ 9

Step 1: First swap (2nd from start & end):-

- Swap 2 and 8.

1 ↔ 8 ↔ 3 ↔ 4 ↔ 5 ↔ 6 ↔ 7 ↔ 2 ↔ 9

Step 2: 2nd swap (4th from start & end):-

- swap 4 and 6

1 ↔ 8 ↔ 3 ↔ 6 ↔ 5 ↔ 4 ↔ 7 ↔ 2 ↔ 9

Step 3: reached center, stop swapping.

Final List:-

1 ↔ 8 ↔ 3 ↔ 6 ↔ 5 ↔ 4 ↔ 7 ↔ 2 ↔ 9

TASK-02

Structure of System:-

- The system has Stores.
- Each store has multiple Sections.
- Each section contains multiple Items.
- All are maintained using Doubly Linked Lists (DLL).

A. Adding a New Store:-

- create a new node of Store.
- If store list is empty, set storeFirst and storeLast to new store.

DATE __/__/____

- Else, add a new store at end of list and adjust prev and next links

CODE FOR THIS:-

```
void addStore(string storeName) {
    Store * newStore = new Store;
    newStore -> stormName = storeName;
    if (!storeFirst) {
        storeFirst = storeLast = newStore;
    } else { storeLast -> next = newStore;
        newStore -> prev = storeLast;
        storeLast = newStore;
    }
}
```

B. Adding New Section to Store: · Search for store
- create new node of ~~Store~~Section
- then add it to Store.

CODE:-

```
void addSectionToStore(string storeName, string sectionName) {
    Store * storeptr = storeFirst;
    while (storeptr && storeptr -> storeName != storeName) {
        storeptr = storeptr -> next;
    }
    if (!storeptr) return;
    // same code as used in above method.
    but only for Section Struct.
}
```

DATE: __/__/____

C. Adding new item to section:

```
void addItem (string storeName, string secName, string itemName){
    // search for store. (same code as used in above method).
    // search for section using sectionptr and same code
      as for store search.
    // create a new Item and the add it. as:
    Item *newItem = new Item;
    newItem->itemName = itemName;
    if (!sectionptr->itemfirst) {      //adding first item
      sectionptr->itemfirst = sectionptr->itemLast = newItem.
    }else {
      sectionptr->itemLast->next = newItem;
      newItem->prev = sectionptr->itemLast;
      sectionptr->itemLast = newItem;    //increment last.
    }
```

D. Remove item from Section:

```
void removeItem(string storeName, string secName, string itemName){
    // search for store
    // search for section      } if any not found then
    // search for item.          return

    if (itemptr->prev){      // if it is not first.
      itemptr->prev->next = itemptr->next;
    }else {
      sectionptr->itemfirst = itemptr->next;
    }   // if it's first of item list then increment itemfirst.
```

DATE: __/__/____

```
// handle itemLast with same logic.
if (itemptr->next)
    itemptr->next->prev = itemptr->prev;
else
    sectionptr->itemLast = itemptr->prev;

    delete itemptr;
}
```

E. Display Items of Section:

```
void displayItems(string storeName, string sectionName){
    // search for store
    // search for section
    // Print all available items:-
    cout << "Items in section: " << sectionName << endl;
    Item *itemptr = sectionptr->itemFirst;
    while (itemptr){
        cout << "- " << itemptr->itemName << endl;
        itemptr = itemptr->next;
    }
}
```

used in (F)

F. Display Items of Store:

```
void displayItemsOfStore(string StoreName){
    // search for store.
    Section *sectionptr = storeptr->sectionFirst;
    while (sectionFirst)
        // code used in E
        sectionptr = sectionptr->next;
    }
}
```

DATE: __/__/____

Structures of Nodes:-

// ~~Singly~~ Store

```
Struct Item {
    string itemName;
    Item * prev = NULL;
    Item * next = NULL;
}

struct Section {
    string secName;
    Item* itemFirst = NULL, *itemLast = NULL;
    Section * prev = NULL, *next = NULL;
}

struct Store {
    string storeName;
    Section *sectionFirst = NULL; * sectionLast = NULL;
    Store * prev = NULL; * next = NULL;
}

Store * storeFirst = NULL, *storeLast = NULL;
```

Diagram:-