Generalized Linked Lists

→ A GLL is a finite sequence of $n \geq 0$ elements, $a_0, a_1, a_2 \cdots a_{n-1}$, where a_i is either an atom or a list.

→ The elements a_i that are not atoms are said to be sublists. $(0 \leq i \leq n-1)$

→ A list A is written as $A = (a_0, a_1, \cdots a_{n-1})$, & length of list is n.

→ A list name is a capital letter & an atom is represented by a lowercase letter.

→ Here, a_0 is the head of list A & the rest $(a_1, \cdots, a_{n-1})$ is the tail of list A.

E.g.

→ A = () → null, ~~length 2 in~~ len = 0

→ B = (a, (b, c)) → 1st element = a
2nd —"— = (b, c)

len = 2

→ C = (B, B, ()) → 1st element = list B
2nd —"— = —"—
3rd —"— = null

len = 3

→ D = (a, D) → recursive list
D corr. to ∞ list D = (a, (a, (a, ---)))

→ head(B) = 'a' tail(B) = (b, c)

head(tail(C)) = B tail(tail(C)) = C

→ Lists may be shared by other lists
→ —"— —"— recursive.

→ If a list atom is not a sublist, it is atomic
→ GLL can be represented by seq. or linked representation.

Representation :

Flag	Data	Down pointer	Next ptr

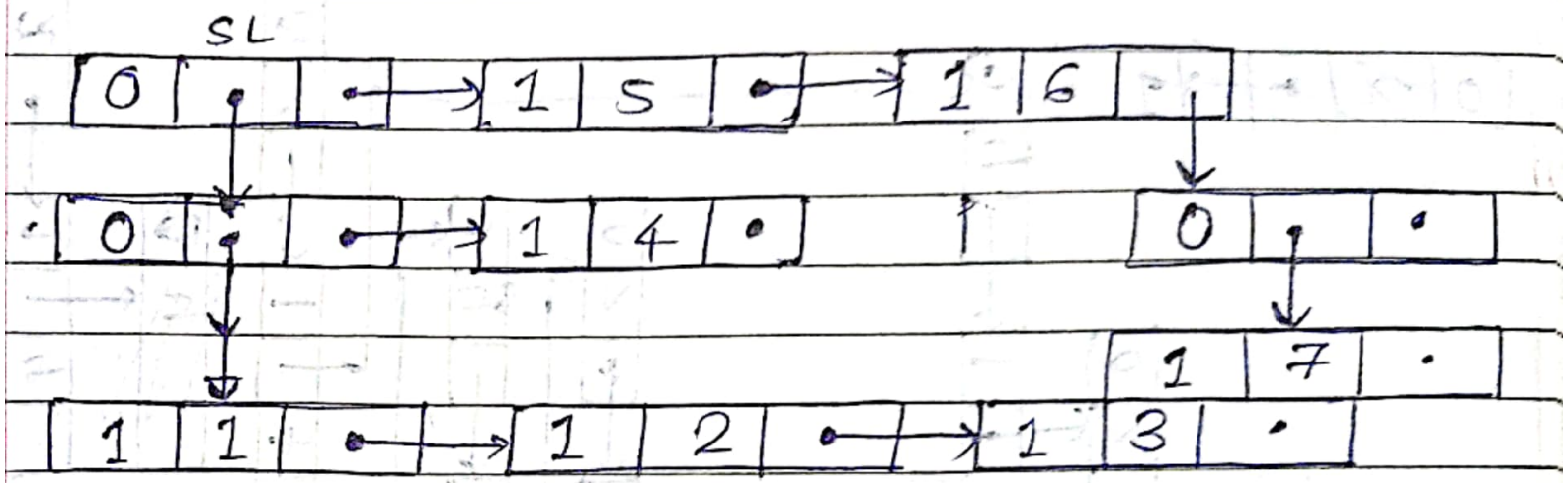
→ Flag = 0 → Down ptr exists → Sublist
→ Flag = 1 → Next ptr exists → atom

→ Data means atom
→ Down ptr is addr. of node which is down of the curr. node.

→ Next ptr is the add. of node which is next

E.g. Rep. without shared sublists.

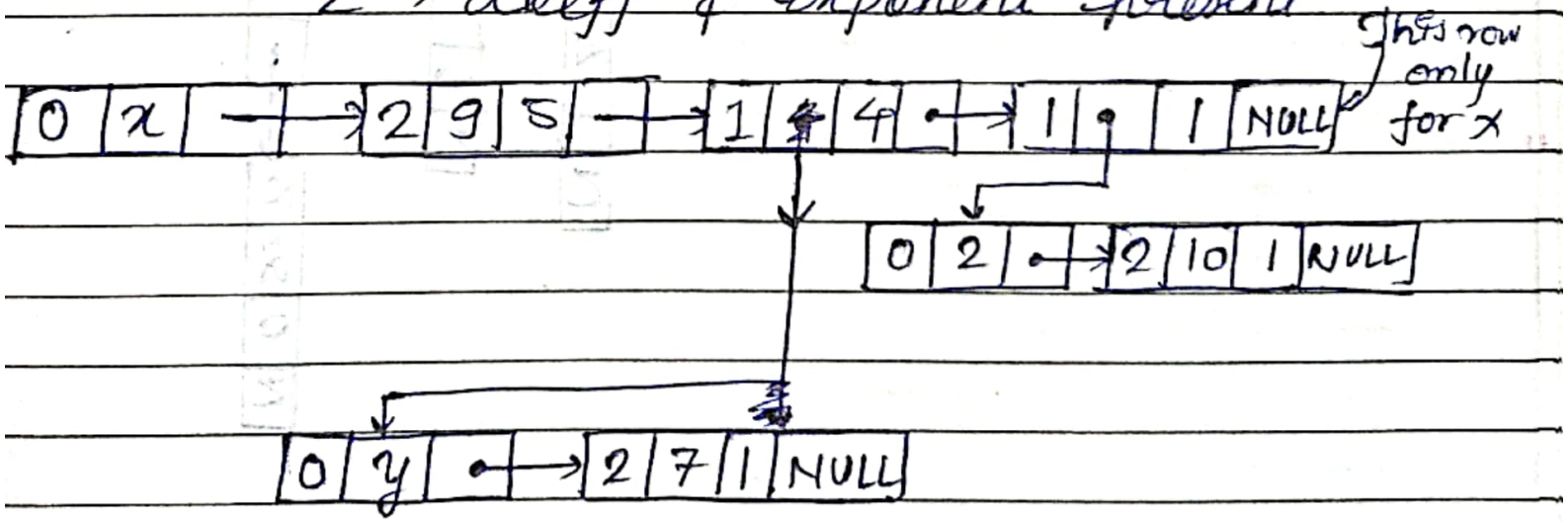
L=(((1,2,3), 4), 5, 6, (7))



For multi-variable

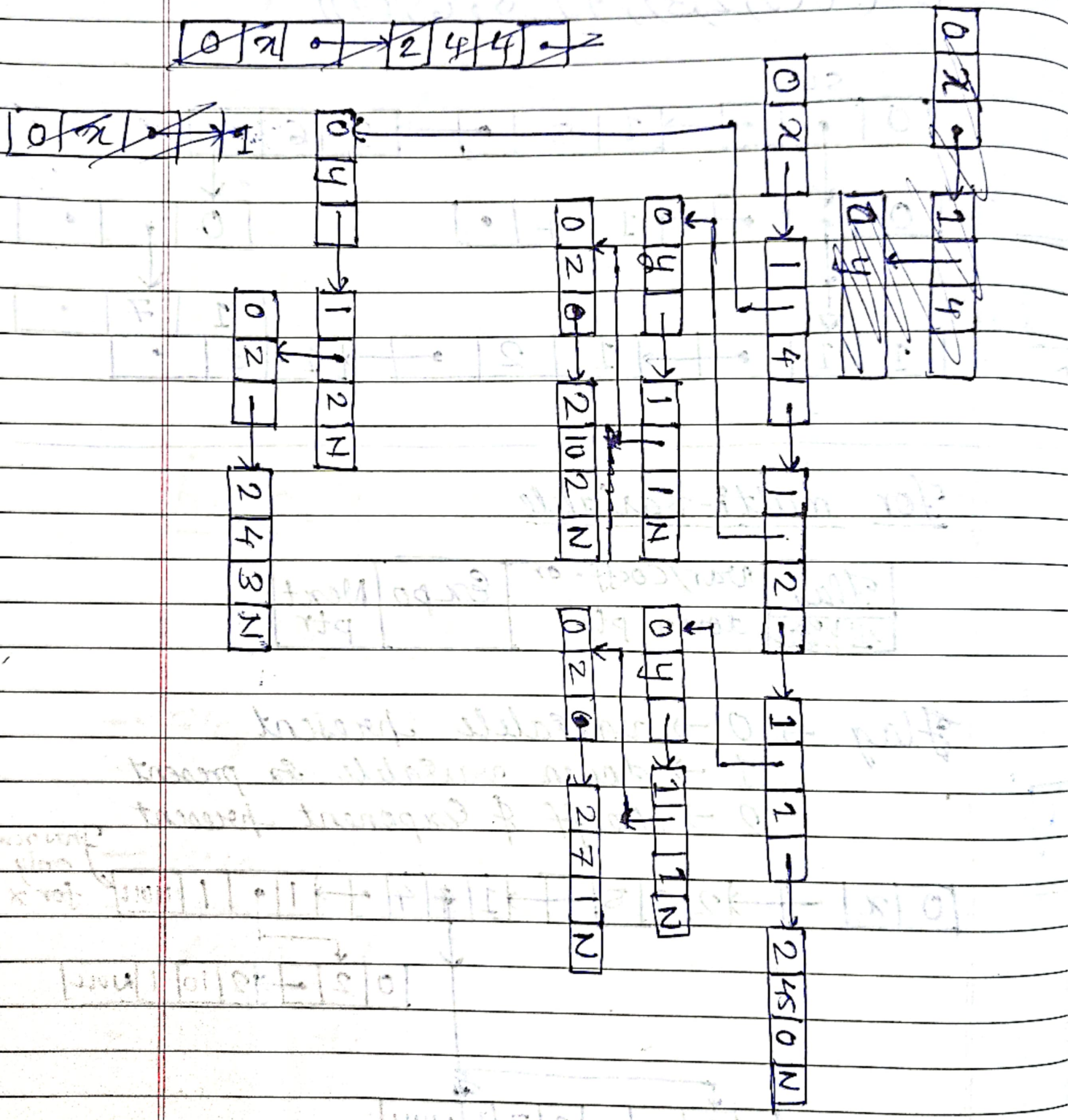
Flag	Var/Coeff. or down ptr	Expo	Next ptr

Flag → 0 → variable present
1 → down variable is present
2 → Coeff & Exponent present



$9x^5 + 7x^4y + 10xz$

Q2 $4x^4y^2z^3 + 10x^2yz^2 + 7xyz + 45$



Polynomial Add^{n}

```
struct node * insert(struct node* head, float co,
                                        int ex)
{
  struct node* temp;
  struct node* newP = malloc(sizeof(struct
                                    node));
  newP → coeff = co;
  newP → expo = ex;
  newP → link = NULL;
If(head == NULL || ex > head → expo)
  {
  newP → link = head;
  head = newP;
  }
else{
   temp = head;
  while(temp → link != NULL &&
        temp → link → expo >= ex)
    {
      temp = temp → link;
      }

**
newP → link = temp → link;
 temp → link = newP;

    }
return head;
  }
```

```
void polyAdd(struct node* head1,
                 "        " * head2)

  struct node *ptr1 = head1;
  struct node * ptr2 = head2;
  - " ——— " - * head3 = NULL;

while (ptr1 != NULL && ptr2 != NULL)
   {

      if(ptr1 → expo == ptr2 → expo)
        {
       head3 = insert(head3, ptr1 → coeff +
                 ptr2 → coeff, ptr1 → expo);

       ptr1 = ptr1 → link;
       ptr2 = ptr2 → link;

        }

      else if(ptr1 → expo > ptr2 → expo)
        {
       head3 = insert(head3, ptr1 → coeff,
                   ptr1 → expo);
        ptr1 = ptr1 → link;

        }
```

```
else if(ptr2->expo > ptr1->expo)
{
head3 = insert(head3, ptr2->coeff, ptr2->expo);
ptr2 = ptr2->link;
}
}
while (ptr1 != NULL)
{
head3 = insert(head3, ptr1->coeff, ptr1->expo);
ptr1 = ptr1->link;
}
while (ptr2 != NULL)
{
head3 = insert(head3, ptr2->coeff, ptr2->expo);
ptr2 = ptr2->link;
}
struct node *p1 = head3;
pf("Added Polynomial:\n");
while (p1 != NULL){
printf("%d ", p1->da
pf("(%dx^%d)+", p1->coeff, p1->expo);
p1 = p1->link;
}
if(p1 != NULL)
pf(" + ");
else
pf("\n");
}
}
```