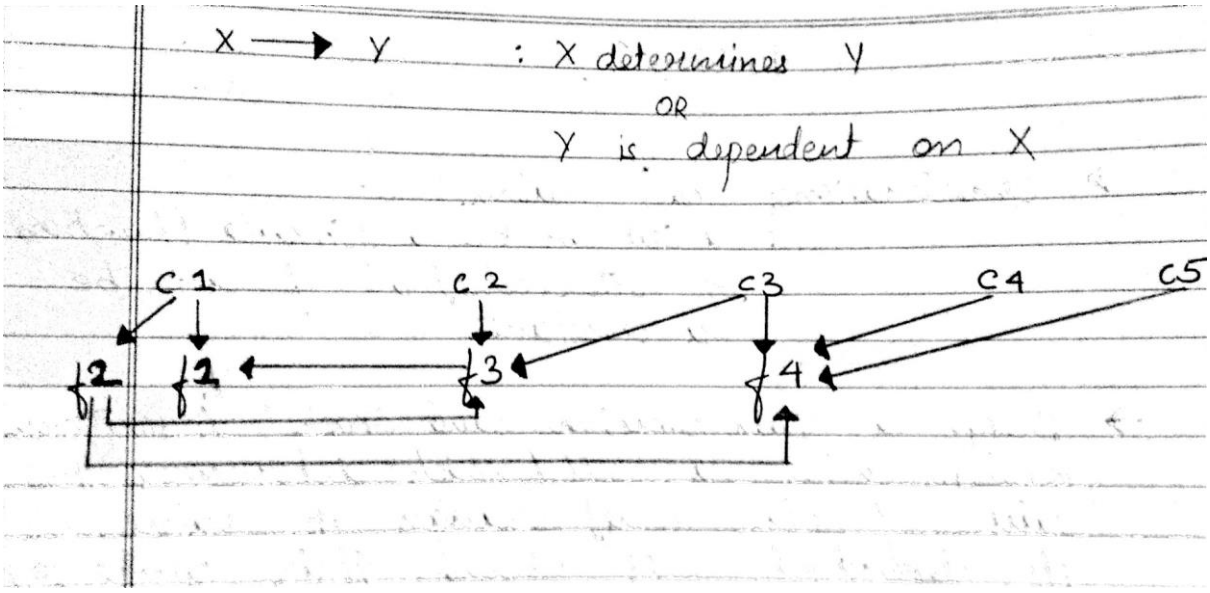
DEPENDENCY TREE

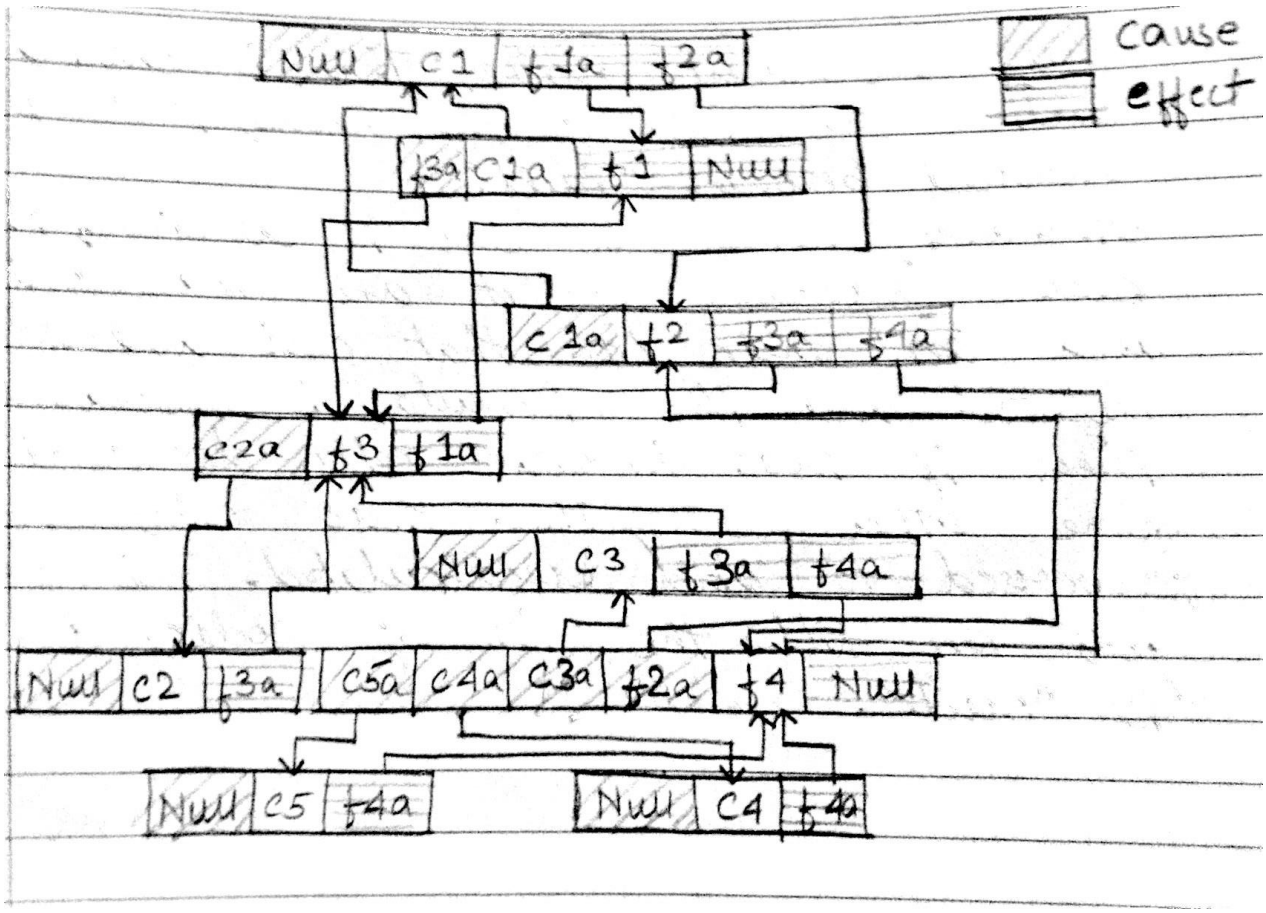
- **Considerations:**
  - n x n matrix
  - A cell can be a formula (function of other cells and/or other constants), or it can be a constant.
- Design, implement and analyse a space and time efficient algorithm and necessary data structure for the cells of M such that when a particular cell i_0j_0 is updated there is least amount of computation required to update the cells depending on i_0j_0 and thus producing an efficient matrix update mechanism.
- **Notations:**
  - C_i: constant valued cell, where i is an integer
  - f_i: formula cell which may depend on a constant cell, another formula cell or both, where i is an integer
  - *a (suffix 'a'): address of the cell(contents)
- Now consider an example of functions (formula cells) and constants.
  f1(c1, f3) f2(c1) f3(c2, f2, c3) f4(c3, f2, c4, c5)



For the given problem, each cell can be stored as a node in the following manner:

Cause (cell address)	Cell Contents	Effects (Cell Address)

> At the centre of the node, the cell contents (constant or formula) is contained.
> The left side contains addresses of all cells that determine the cell contained ("null" in case of constants)
> The right side contains the addresses of all cells that are dependent on or determined by the cell contents ("null" if the cell does not determine any other cell)



➔ Each Node contains the content of the cell along with the addresses (if any) of the cells that this particular cell determines or is dependent on.
Thus, when the content of a constant cell changes, that change is detected and the RHS of the node is traversed. This RHS contains the addresses of all cells that are dependent on this constant. So, these addresses are traversed to and they are recalculated (using the LHS addresses the contents are fetched and used for the recalculation). Similarly, these functions' nodes right side is checked, and if it is not "null", then the process is repeated for these dependent nodes. This is carried on until the RHS of the node(s) that is (are) reached becomes "null".
Similarly, when a cell is deleted, the cells dependent on it are travelled to and on their LHS that cell address is deleted and the formula is recalculated. Then if these formula cells determine some other cell, then these cells are traversed to and recalculated. This process goes on till the RHS of node(s) reached is "null".

➔ **Changing an element**

```
Cell temp;
System.out.println("Enter location of cell that you want to edit(Row and Column consecutively): ");
row = read.nextInt();
col = read.nextInt();
System.out.println("Enter New Value");
content = read.nextDouble();
cell[row][col].setContent(content);
for (int i = 0; i < cell[row][col].getEffects().size(); i++) { //time complexity = c1n
    //getting the formula cells that are affected by the cell that we have edited
    temp = cell[row][col].getEffects().get(i);  //time complexity = c2
    temp.setContent(eVal.evaluate(temp.getContentFormula())); //recalculating the formula in the cell//time complexity = c3
    int j = 0;
    //changning values of the cells that the formula cell affects (if any)
    if (temp.getEffects().size() != 0) { //time complexity = c4
        while (temp.getEffects().size() < j) { //time complexity = c5n
            Cell tempEffects = temp.getEffects().get(j); //time complexity = c6
            tempEffects.setContent(eVal.evaluate(tempEffects.getContentFormula())); //time complexity = c7
            j++;
        }
    }
}

//Thus, the time complexity here is O(n^2)
```

➔ **Deleting an element**

```
System.out.println("Enter location of cell that you want to delete(Row and Column consecutively): ");
row = read.nextInt();
col = read.nextInt();
cell[row][col].getEffects().clear(); //time complexity = c1
cell[row][col] = null; //time complexity = c1
break;
//Thus the time complexity here is a constant, O(1)
```

- **Aashima Yuthika, 1401071**