Problem 1

First, we need to find the frequent items in the set of transactions

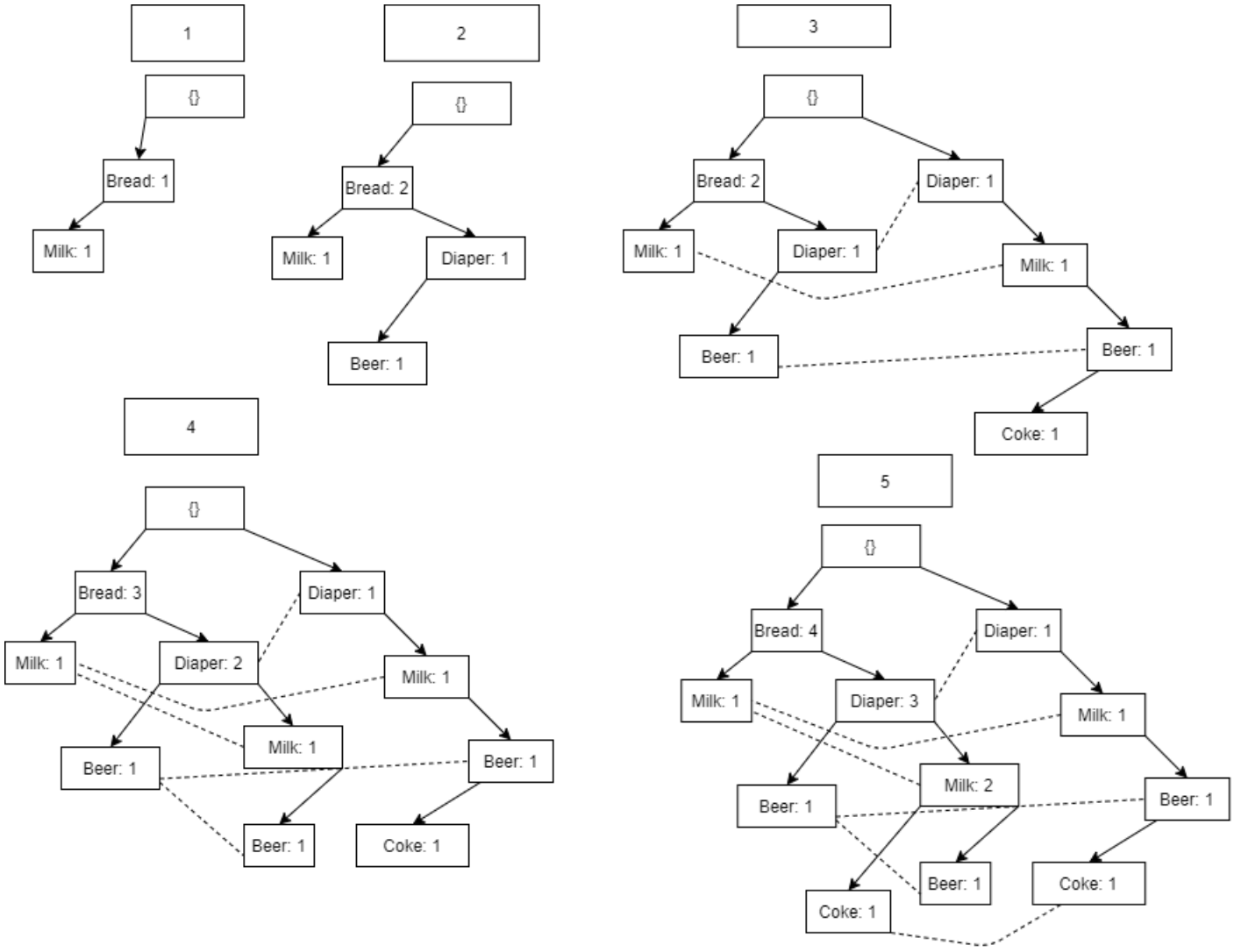
We can make a table of the frequencies of each item

Item	Count
Bread	4
Milk	4
Beer	3
Diaper	4
Eggs	1
Coke	2

Next, we find each item with a count above 2, then we make it so each transaction only contains frequent items sorted by frequency

TID	Items
1	Bread, Milk
2	Bread, Diaper, Beer
3	Diaper, Milk, Beer, Coke
4	Bread, Diaper, Milk, Beer
5	Bread, Diaper, Milk, Coke

Steps to create FP Tree



In [1]:
```
transactions = {'1': set(('bread', 'milk')),
                '2': set(('beer', 'bread', 'diaper', 'eggs')),
                '3': set(('beer', 'coke', 'diaper', 'milk')),
                '4': set(('beer', 'bread', 'diaper', 'milk')),
                '5': set(('bread', 'coke', 'diaper', 'milk'))}
minsup = 2
```

In [2]:
```
def getFrequencies(transactions): #build dictionary of items and their frequencies
    frequencies = {}
    for key in transactions:
        for item in transactions[key]:
            if item in frequencies:
                frequencies[item] += 1
            else:
                frequencies[item] = 1
    return frequencies

frequencies = getFrequencies(transactions)
frequencies
```

Out[2]:
```
{'bread': 4, 'milk': 4, 'diaper': 4, 'eggs': 1, 'beer': 3, 'coke': 2}
```

In [3]:
```
def getFrequentItems(frequencies, minsup): #only show items with count above minimum support
    items = {}
    for key in frequencies:
        if frequencies[key] >= minsup:
            items[key] = frequencies[key]
    return items

frequentItems = getFrequentItems(frequencies, minsup)
frequentItems
```

Out[3]:
```
{'bread': 4, 'milk': 4, 'diaper': 4, 'beer': 3, 'coke': 2}
```

In [4]:
```
def frequentInTransactions(transactions, minsup):
    #return dictionary of transactions and sorted frequent items
    frequentTransactions = {}
    frequencies = getFrequencies(transactions)
    frequentItems = getFrequentItems(frequencies, minsup)
    for key in transactions:
        frequentTransactions[key] = []
        for item in transactions[key]:
            if item in frequentItems.keys():
                frequentTransactions[key].append((item, frequentItems[item]))

    for key in frequentTransactions:
        frequentTransactions[key].sort(key=lambda x: x[1], reverse=True)
        frequentTransactions[key] = list(map(lambda pair: pair[0], frequentTra
nsactions[key]))
    return frequentTransactions

frequent = frequentInTransactions(transactions, 2)
frequent
```

Out[4]:
```
{'1': ['bread', 'milk'],
 '2': ['diaper', 'bread', 'beer'],
 '3': ['diaper', 'milk', 'beer', 'coke'],
 '4': ['diaper', 'milk', 'bread', 'beer'],
 '5': ['diaper', 'bread', 'milk', 'coke']}
```

Construct class to represent FPTree and internal nodes in each tree

In [5]:
```
class node:
    def __init__(self, value = None):
        self._value = (value, 1)
        self._children = {} #dictionary to keep track of children in node

    def insert(self, value):
        #insert value into node, and return the new node created in order to k
eep track of pointers to node
        if value == self._value[0]:
            self._value = (self._value[0], self._value[1] + 1)
            return self
        elif str(value) in self._children:
            self._children[str(value)].insert(value)
        else:
            newNode = node(value)
            self._children[str(value)] = newNode
            return newNode
    '''
    def printTree(self):
        s = str(self._value[0])
        for elem in self._children:
            s += " " + self._children[elem].printTree() + "\n"
        s += '\n\t'
        return s
    '''

class FPTree:
    def __init__(self, transactions, minsup):
        self._root = node()
        self._transactions = transactions
        self._minsup = minsup
        self._pointers = {}
        self._constructTree()

    def _constructTree(self):
        frequent = frequentInTransactions(self._transactions, self._minsup)
        for key in frequent:
            self.insert(frequent[key])

    def insert(self, values):
        curr = self._root
        for value in values:
            newNode = None
            if str(value) in curr._children:
                curr = curr._children[str(value)]
                newNode = curr.insert(value)
            else:
                newNode = curr.insert(value)
                curr = curr._children[str(value)]
            if not str(value) in self._pointers:
                self._pointers[str(value)] = set()
            self._pointers[str(value)].add(newNode)
    '''
    def printTree(self):
        return self._root.printTree()
    '''
```

In [6]:
```
fp = FPTree(transactions, 2)
```

In [7]:
```
fp._pointers
```

Out[7]:
```
{'bread': {<__main__.node at 0x1c4e88fc4e0>,
  <__main__.node at 0x1c4e88fc6a0>,
  <__main__.node at 0x1c4e88fccc0>},
 'milk': {<__main__.node at 0x1c4e88fc2e8>,
  <__main__.node at 0x1c4e88fc8d0>,
  <__main__.node at 0x1c4e890f748>},
 'diaper': {<__main__.node at 0x1c4e88fc400>},
 'beer': {<__main__.node at 0x1c4e88fc470>,
  <__main__.node at 0x1c4e88fc978>,
  <__main__.node at 0x1c4e890f198>},
 'coke': {<__main__.node at 0x1c4e88fcf98>, <__main__.node at 0x1c4e890f438
>}}
```

In [8]:
```
for key in fp._pointers:
    for elem in fp._pointers[key]:
        print(elem._value)
```

```
('bread', 2)
('bread', 1)
('bread', 1)
('milk', 1)
('milk', 2)
('milk', 1)
('diaper', 4)
('beer', 1)
('beer', 1)
('beer', 1)
('coke', 1)
('coke', 1)
```

Problem 2

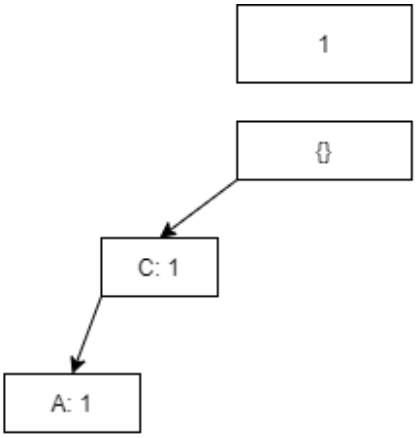
Table of the count of each item

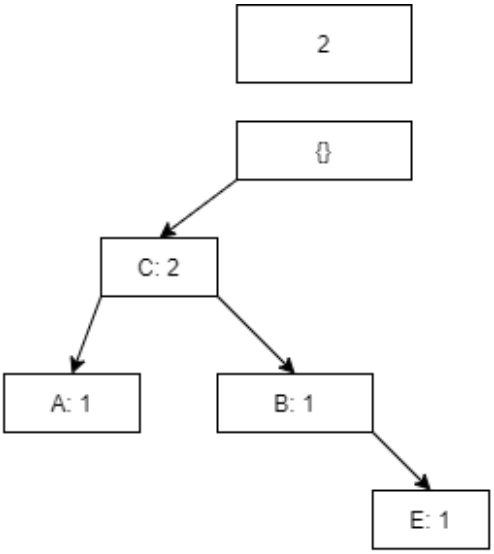
Item	Count
A	2
C	3
D	1
B	3
E	3

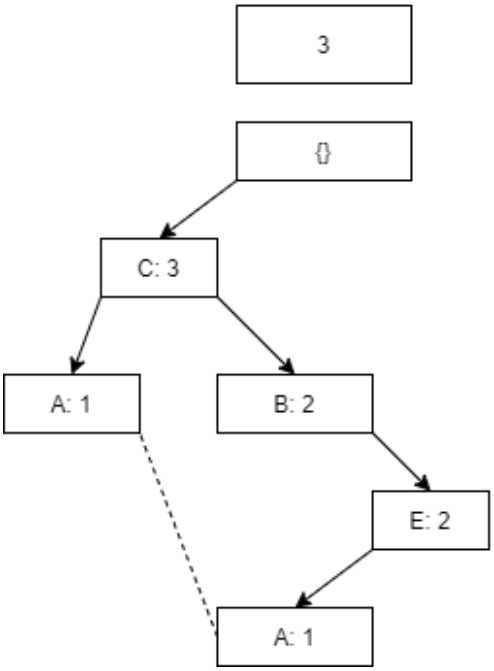
Table of transactions with sorted frequent items only

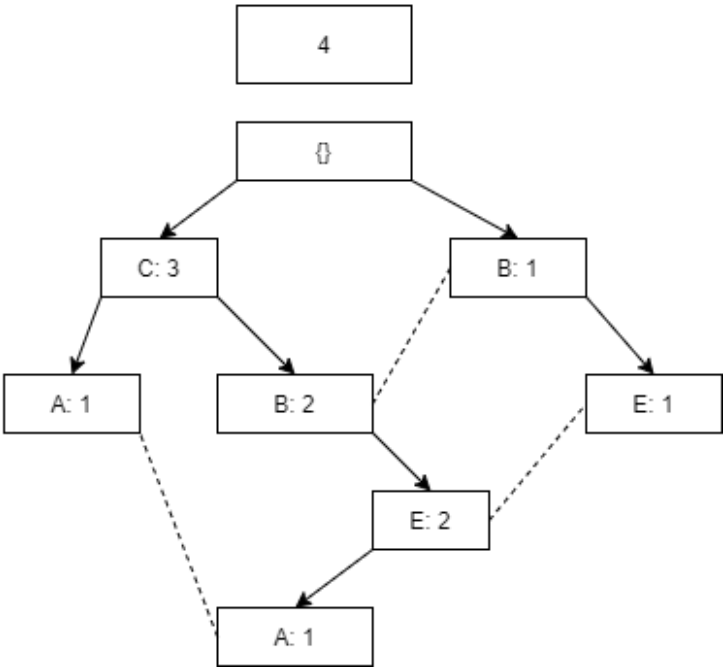
TID	Items
10	C, A
20	C, B, E
30	C, B, E, A
40	B, E

Final FP Tree created step by step









In [9]:
```
transactions2 = {'10': set(('A', 'C', 'D')),
                 '20': set(('B', 'C', 'E')),
                 '30': set(('A', 'B', 'C', 'E')),
                 '40': set(('B', 'E'))}
fp2 = FPTree(transactions2, 2)
```

In [10]:
```
for key in fp2._pointers:
    for elem in fp2._pointers[key]:
        print(elem._value)
```

```
('C', 3)
('A', 1)
('A', 1)
('B', 2)
('B', 1)
('E', 2)
('E', 1)
```

Notice here that the algorithm I implemented created a different tree than the one in the above diagram, though both have the same number of nodes and should be another optimal representation of the FPTree

In [11]:
```
fp2._pointers
```

Out[11]:
```
{'C': {<__main__.node at 0x1c4e8931080>},
 'A': {<__main__.node at 0x1c4e89310b8>, <__main__.node at 0x1c4e8931160>},
 'B': {<__main__.node at 0x1c4e89310f0>, <__main__.node at 0x1c4e89311d0>},
 'E': {<__main__.node at 0x1c4e8931048>, <__main__.node at 0x1c4e8931198>}}
```