AN ALTERNATIVE METHOD FOR ASSOCIATION RULES

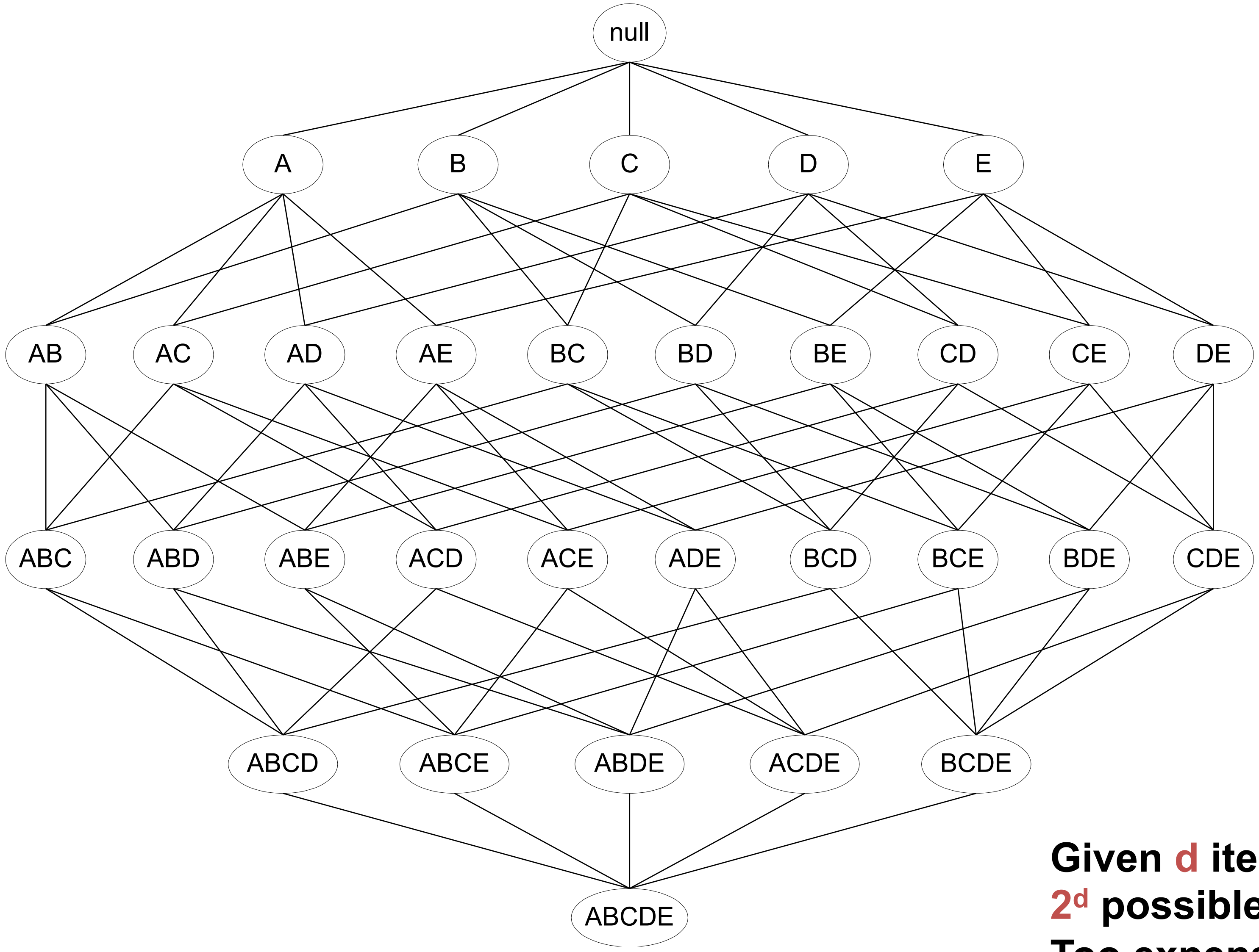
RECAP

Mining Frequent Itemsets

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

- **Itemset**
  - A collection of one or more items
    - Example: {Milk, Bread, Diaper}
  - k-itemset
    - An itemset that contains k items
- **Support (σ)**
  - **Count**: Frequency of occurrence of an itemset
  - E.g. σ({Milk, Bread,Diaper}) = 2
  - **Fraction**: Fraction of transactions that contain an itemset
  - E.g. s({Milk, Bread, Diaper}) = 40%
- **Frequent Itemset**
  - An itemset whose support is greater than or equal to a minsup threshold, minsup
- **Problem Definition**
  - Input: A set of transactions T, over a set of items I, minsup value
  - Output: All itemsets with items in I having minsup

The itemset lattice



Given d items, there are 2^d possible itemsets
Too expensive to test all!

The Apriori Principle

- Apriori principle (Main observation):
  - If an itemset is frequent, then all of its subsets must also be frequent
  - If an itemset is not frequent, then all of its supersets cannot be frequent

$$\forall X, Y : (X \subseteq Y) \Rightarrow s(X) \geq s(Y)$$

  - The support of an itemset never exceeds the support of its subsets
  - This is known as the anti-monotone property of support

Illustration of the Apriori principle

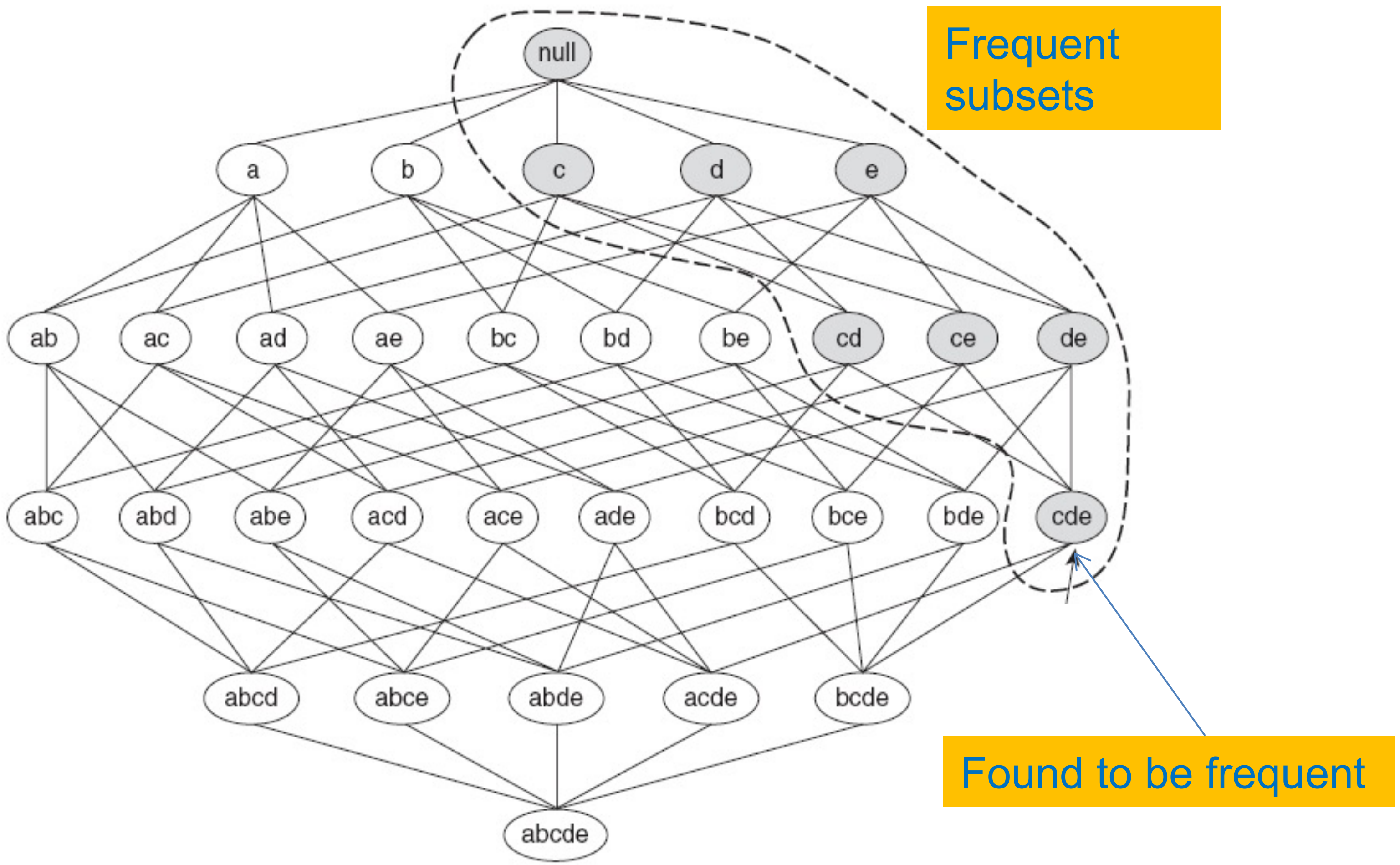
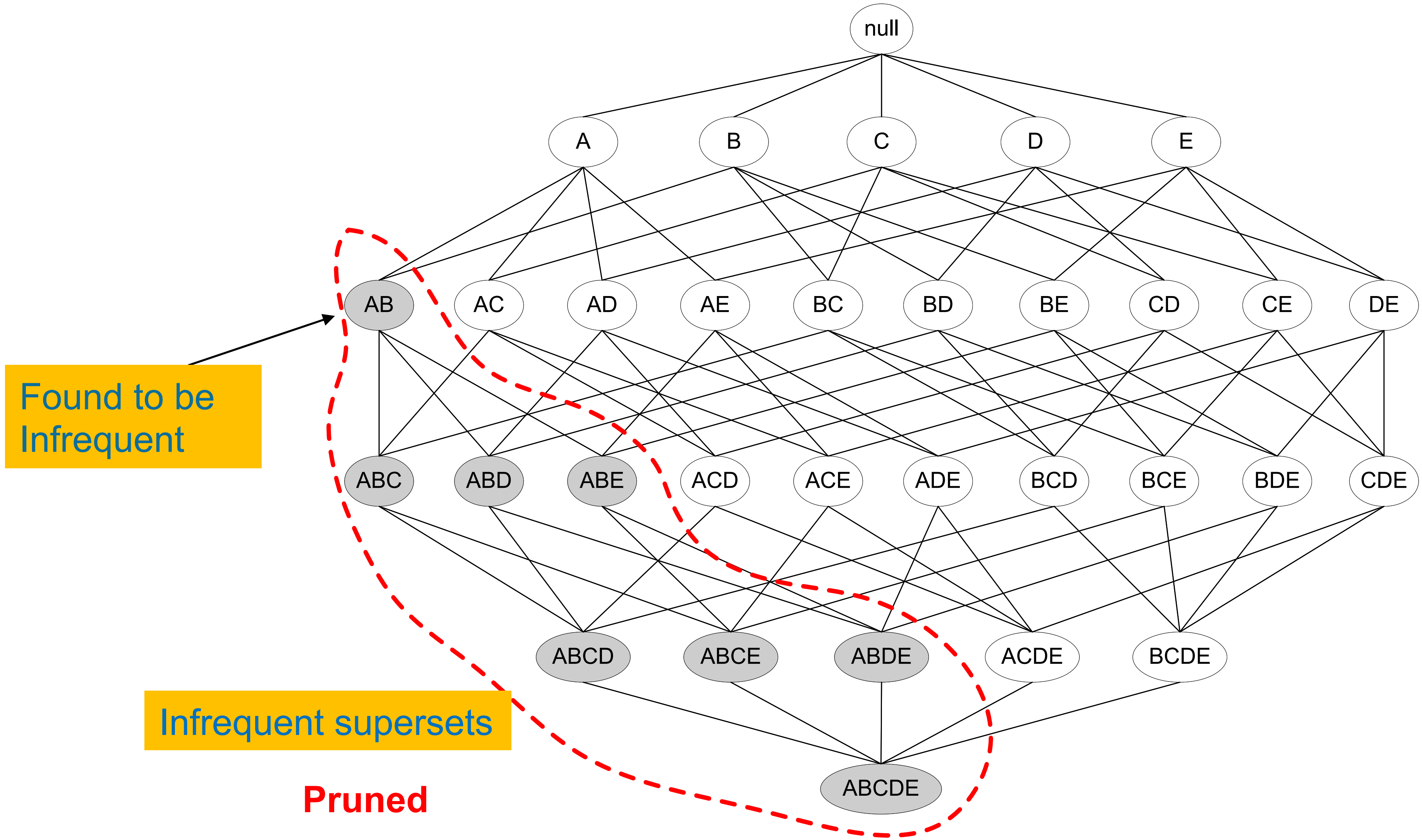


Figure 6.3. An illustration of the *Apriori* principle. If $\{c, d, e\}$ is frequent, then all subsets of this itemset are frequent.

Illustration of the Apriori principle



The Apriori algorithm

Level-wise approach

C_k = candidate itemsets of size k
L_k = frequent itemsets of size k

1. $k = 1$, C_1 = all items
2. While C_k not empty

Frequent itemset generation

3. Scan the database to find which itemsets in C_k are frequent and put them into L_k

Candidate generation

4. Use L_k to generate a collection of candidate itemsets C_{k+1} of size $k+1$
5. $k = k+1$

R. Agrawal, R. Srikant: "Fast Algorithms for Mining Association Rules", *Proc. of the 20th Int'l Conference on Very Large Databases*, 1994.

Candidate Generation

- **Basic principle (Apriori):**
  - An itemset of size k+1 is candidate to be frequent only if all of its subsets of size k are known to be frequent
- **Main idea:**
  - Construct a candidate of size k+1 by combining two frequent itemsets of size k
  - Prune the generated k+1-itemsets that do not have all k-subsets to be frequent

Factors affecting the complexity

- **Choice of minimum support threshold**
  - lowering min support results in more frequent itemsets this may increase number of candidates and max length of frequent itemsets
- **Dimensionality (number of items of the dataset)**
  - more space is needed to store support count of each item
  - if number of frequent items also increases, both computation and I/O costs may also increase
- **Size of database**
  - since Apriori makes multiple passes, run time of algorithm may increase with number of transactions
- **Average transaction length**
  - transaction length increases with denser data sets
  - this may increase max length of frequent itemsets and traversals of hash tree (number of subsets in a transaction increases with its length)

THE FP-TREE AND THE FP-GROWTH ALGORITHM

Overview

- The FP-tree contains a compressed representation of the transaction database.
  - A trie (prefix-tree) data structure is used
  - Each transaction is a path in the tree – paths can overlap.

- Once the FP-tree is constructed the recursive, divide-and-conquer FP-Growth algorithm is used to enumerate all frequent itemsets.

FP-tree Construction

TID	Items
1	{A,B}
2	{B,C,D}
3	{A,C,D,E}
4	{A,D,E}
5	{A,B,C}
6	{A,B,C,D}
7	{B,C}
8	{A,B,C}
9	{A,B,D}
10	{B,C,E}

- The FP-tree is a trie (prefix tree)
- Since transactions are sets of items, we need to transform them into ordered sequences so that we can have prefixes
  - Otherwise, there is no common prefix between sets {A,B} and {B,C,A}
- We need to impose an order to the items
  - Initially, assume a lexicographic order.

FP-tree Construction

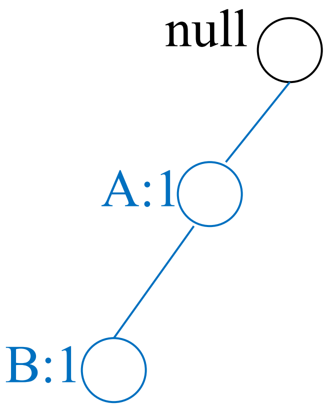
- Initially the tree is empty

TID	Items
1	{A,B}
2	{B,C,D}
3	{A,C,D,E}
4	{A,D,E}
5	{A,B,C}
6	{A,B,C,D}
7	{B,C}
8	{A,B,C}
9	{A,B,D}
10	{B,C,E}

FP-tree Construction

- Reading transaction TID = 1

TID	Items
1	{A,B}
2	{B,C,D}
3	{A,C,D,E}
4	{A,D,E}
5	{A,B,C}
6	{A,B,C,D}
7	{B,C}
8	{A,B,C}
9	{A,B,D}
10	{B,C,E}



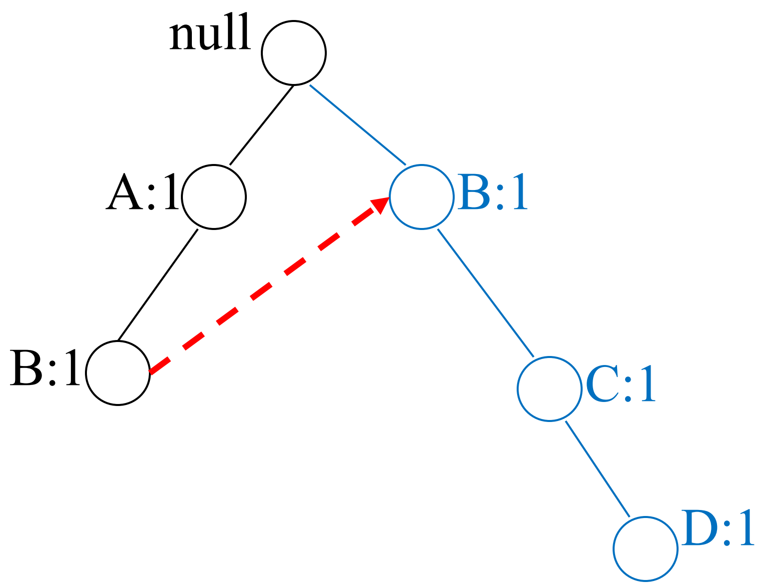
Node label = item:support

- Each node in the tree has a label consisting of the item and the support (number of transactions that reach that node, i.e. follow that path)

FP-tree Construction

- Reading transaction TID = 2

TID	Items
1	{A,B}
2	{B,C,D}
3	{A,C,D,E}
4	{A,D,E}
5	{A,B,C}
6	{A,B,C,D}
7	{B,C}
8	{A,B,C}
9	{A,B,D}
10	{B,C,E}



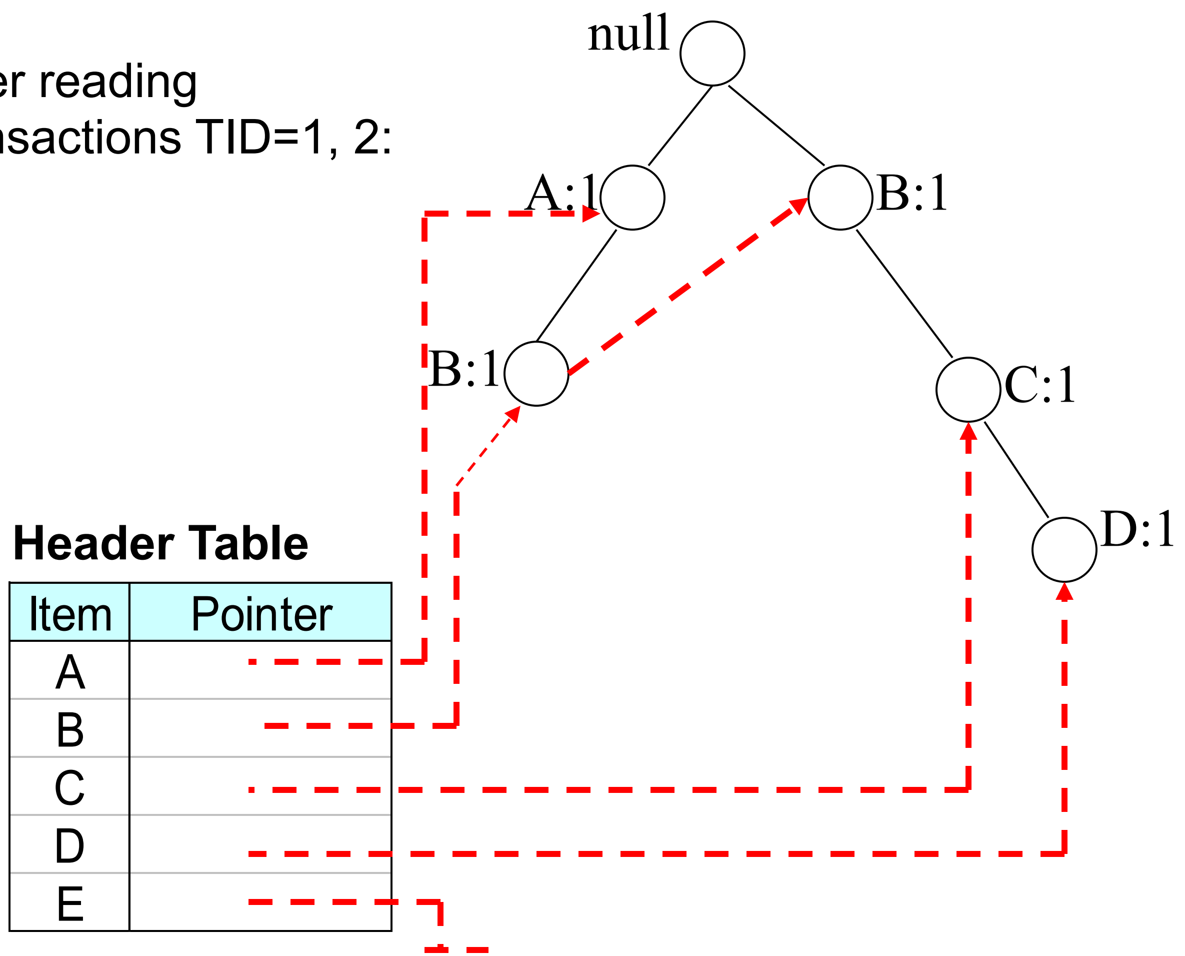
Each transaction is a path in the tree

- We add pointers between nodes that refer to the same item

FP-tree Construction

TID	Items
1	{A,B}
2	{B,C,D}
3	{A,C,D,E}
4	{A,D,E}
5	{A,B,C}
6	{A,B,C,D}
7	{B,C}
8	{A,B,C}
9	{A,B,D}
10	{B,C,E}

After reading transactions TID=1, 2:

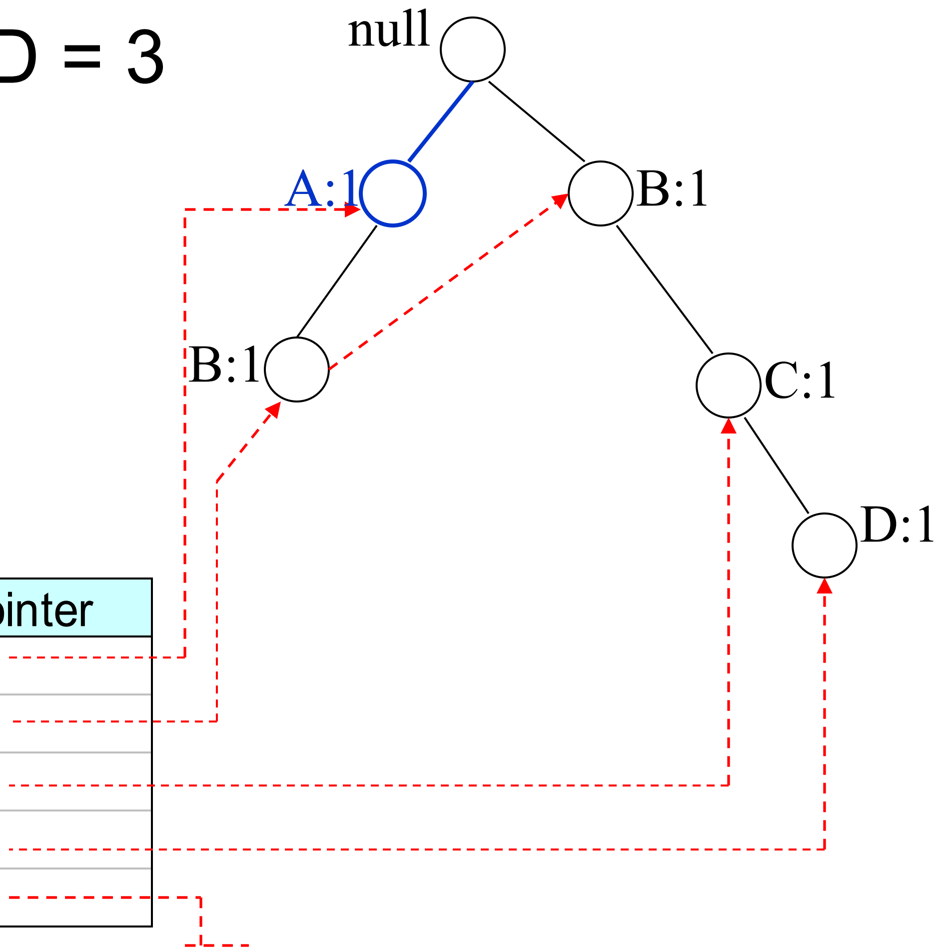


The Header Table and the pointers assist in computing the itemset support

FP-tree Construction

- Reading transaction TID = 3

TID	Items
1	{A,B}
2	{B,C,D}
3	{A,C,D,E}
4	{A,D,E}
5	{A,B,C}
6	{A,B,C,D}
7	{B,C}
8	{A,B,C}
9	{A,B,D}
10	{B,C,E}



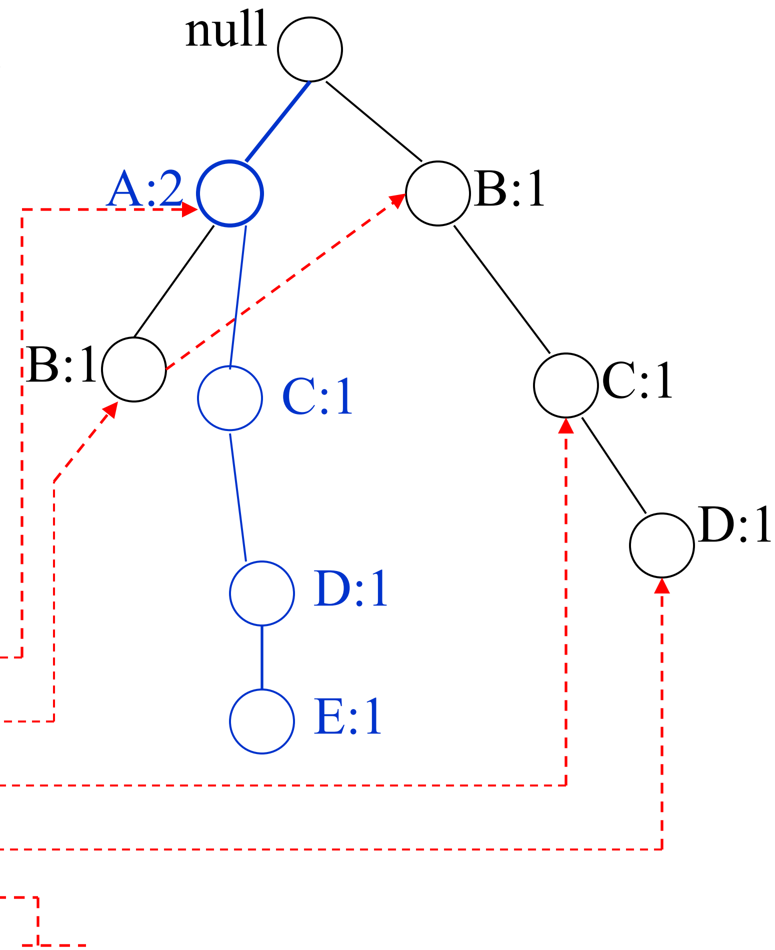
Item	Pointer
A	
B	
C	
D	
E	

FP-tree Construction

- Reading transaction TID = 3

TID	Items
1	{A,B}
2	{B,C,D}
3	{A,C,D,E}
4	{A,D,E}
5	{A,B,C}
6	{A,B,C,D}
7	{B,C}
8	{A,B,C}
9	{A,B,D}
10	{B,C,E}

Item	Pointer
A	
B	
C	
D	
E	

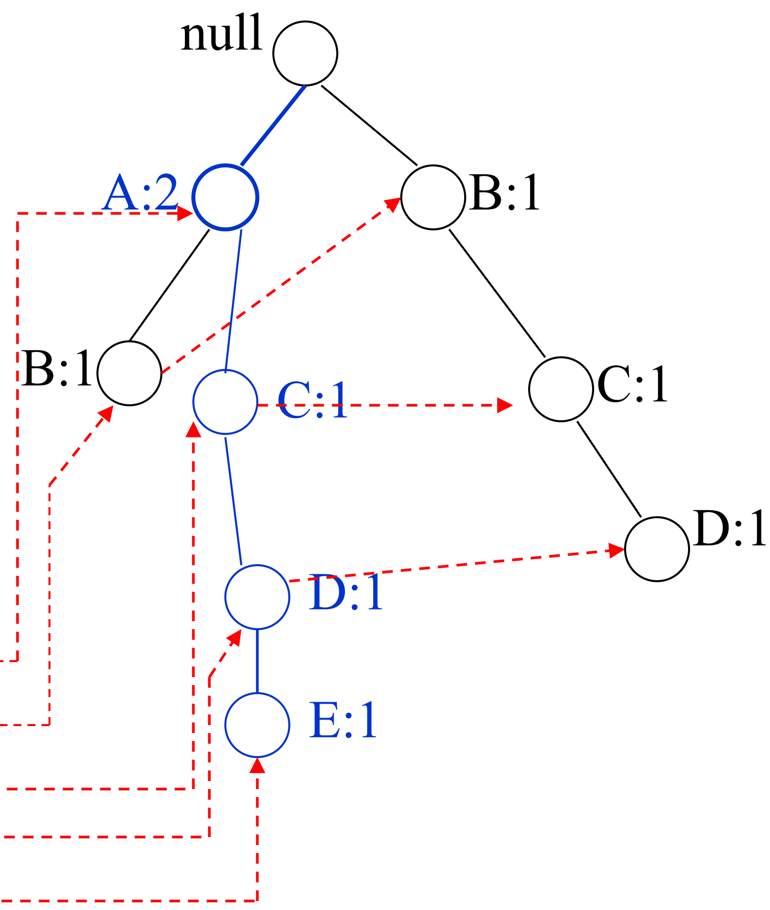


FP-tree Construction

- Reading transaction TID = 3

TID	Items
1	{A,B}
2	{B,C,D}
3	{A,C,D,E}
4	{A,D,E}
5	{A,B,C}
6	{A,B,C,D}
7	{B,C}
8	{A,B,C}
9	{A,B,D}
10	{B,C,E}

Item	Pointer
A	
B	
C	
D	
E	



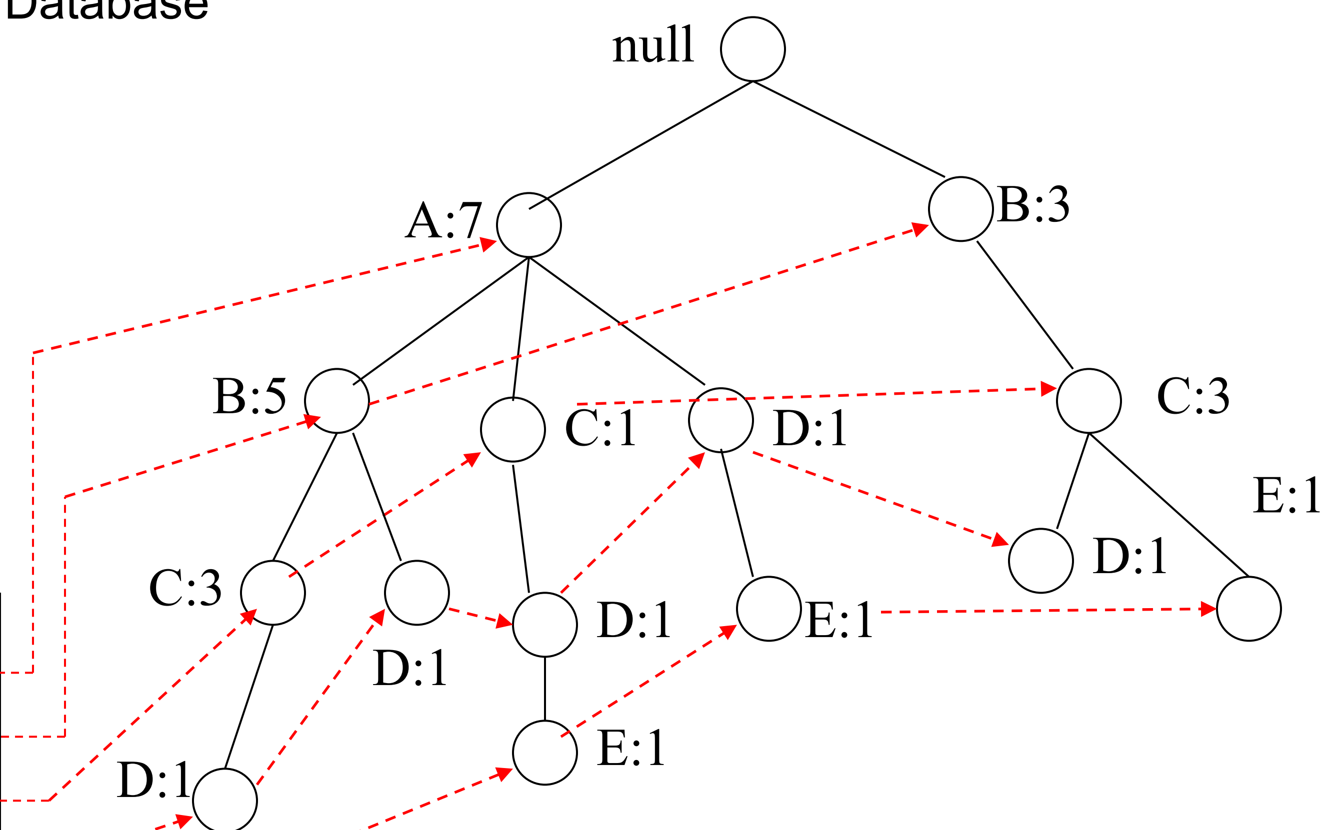
Each transaction is a path in the tree

FP-Tree Construction

TID	Items
1	{A,B}
2	{B,C,D}
3	{A,C,D,E}
4	{A,D,E}
5	{A,B,C}
6	{A,B,C,D}
7	{B,C}
8	{A,B,C}
9	{A,B,D}
10	{B,C,E}

Transaction Database

Each transaction is a path in the tree



Header table

Item	Pointer
A	
B	
C	
D	
E	

Pointers are used to assist frequent itemset generation

FP-tree size

- Every transaction is a path in the FP-tree
- **The size of the tree depends on the compressibility of the data**

  - **Extreme case**: All transactions are the same, the FP-tree is a single branch

  - **Extreme case**: All transactions are different the size of the tree is the same as that of the database (bigger actually since we need additional pointers)

Item ordering

- **The size of the tree also depends on the ordering of the items.**
- **Heuristic**: order the items in according to their frequency from larger to smaller.
  - We would need to do an extra pass over the dataset to count frequencies

- Example:

TID	Items
1	{A,B}
2	{B,C,D}
3	{A,C,D,E}
4	{A,D,E}
5	{A,B,C}
6	{A,B,C,D}
7	{B,C}
8	{A,B,C}
9	{A,B,D}
10	{B,C,E}

$\sigma(A)=7$, $\sigma(B)=8$, $\sigma(C)=7$, $\sigma(D)=5$, $\sigma(E)=3$

Ordering : B,A,C,D,E

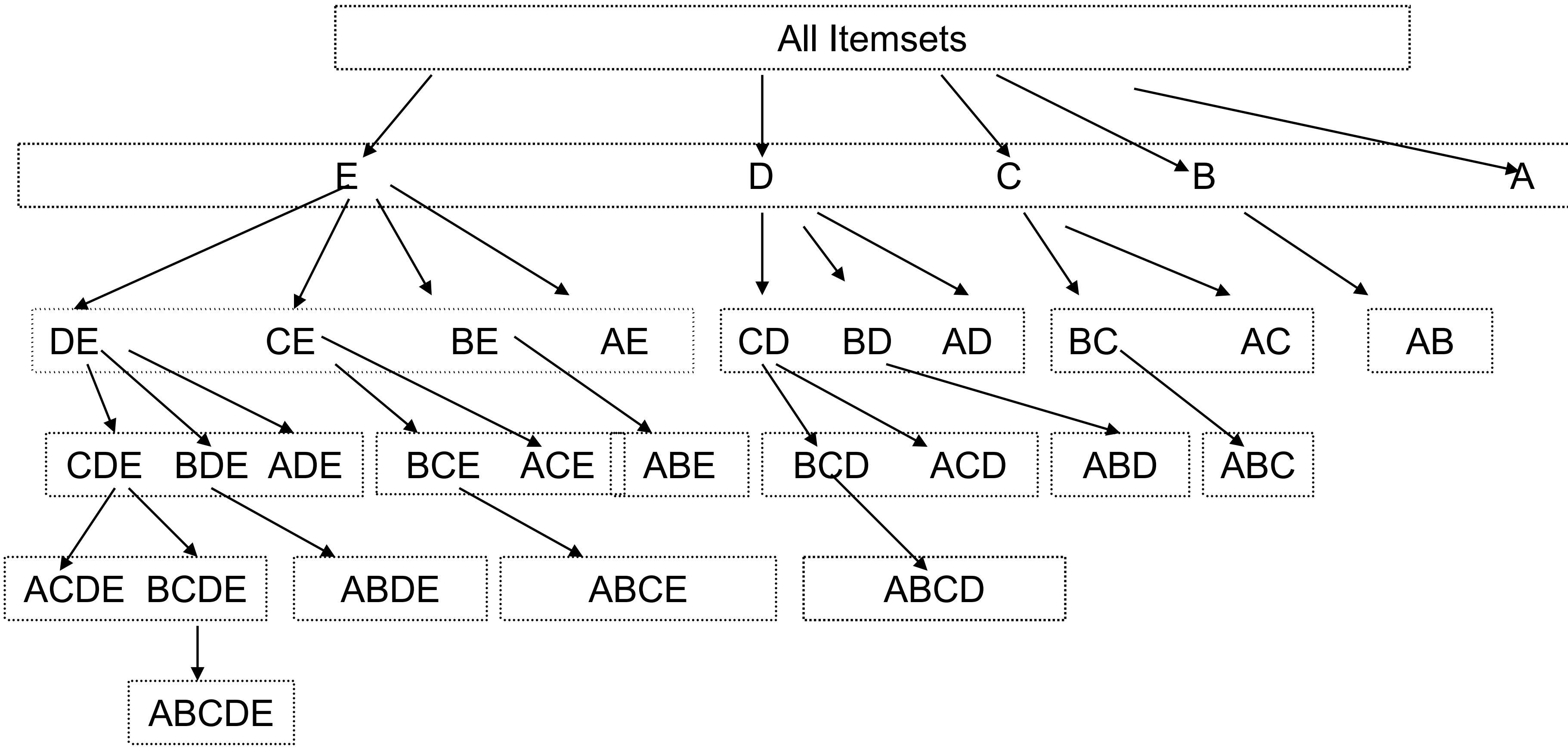
→

TID	Items
1	{B,A}
2	{B,C,D}
3	{A,C,D,E}
4	{A,D,E}
5	{B,A,C}
6	{B,A,C,D}
7	{B,C}
8	{B,A,C}
9	{B,A,D}
10	{B,C,E}

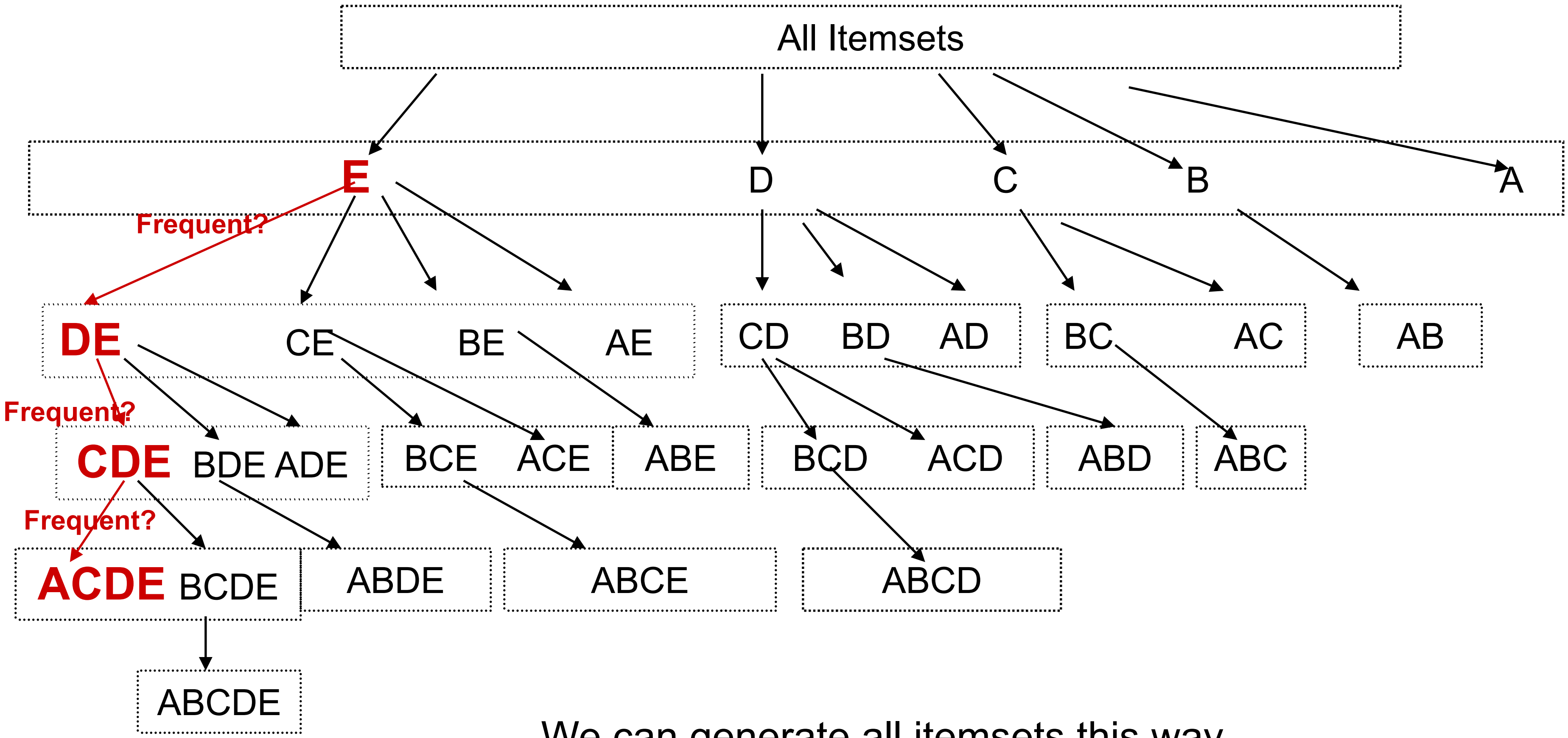
Finding Frequent Itemsets

- **Input**: The FP-tree
- **Output**: All Frequent Itemsets and their support
- **Method**: Divide and Conquer:
  - Consider all itemsets that end in: E, D, C, B, A
    - For each possible ending item, consider the itemsets with last items one of items preceding it in the ordering
    - E.g, for E, consider all itemsets with last item D, C, B, A. In this way we get all the itemsets ending at DE, CE, BE, AE
    - Proceed recursively this way.
    - Do this for all items.

Frequent itemsets



Frequent Itemsets



We can generate all itemsets this way
We expect the FP-tree to contain a lot less

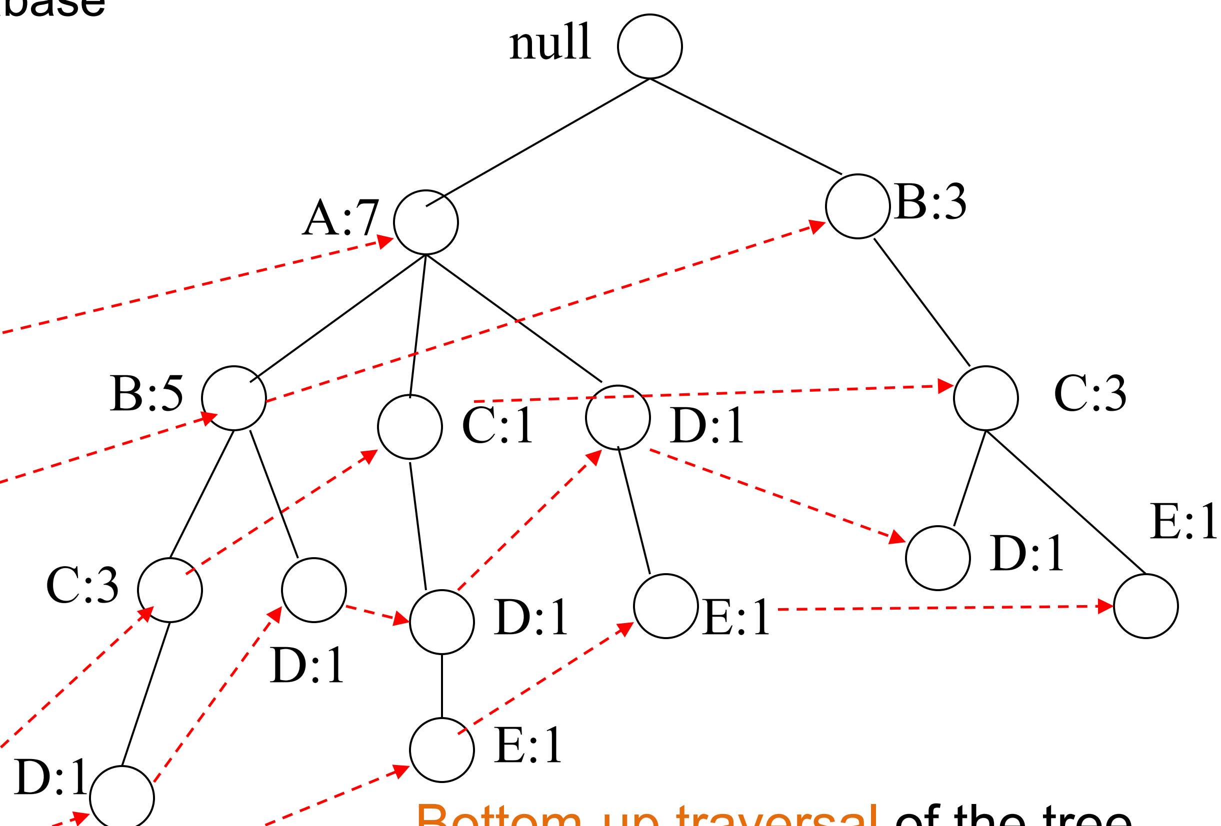
Using the FP-tree to find frequent itemsets

TID	Items
1	{A,B}
2	{B,C,D}
3	{A,C,D,E}
4	{A,D,E}
5	{A,B,C}
6	{A,B,C,D}
7	{B,C}
8	{A,B,C}
9	{A,B,D}
10	{B,C,E}

Transaction Database

Header table

Item	Pointer
A	
B	
C	
D	
E	

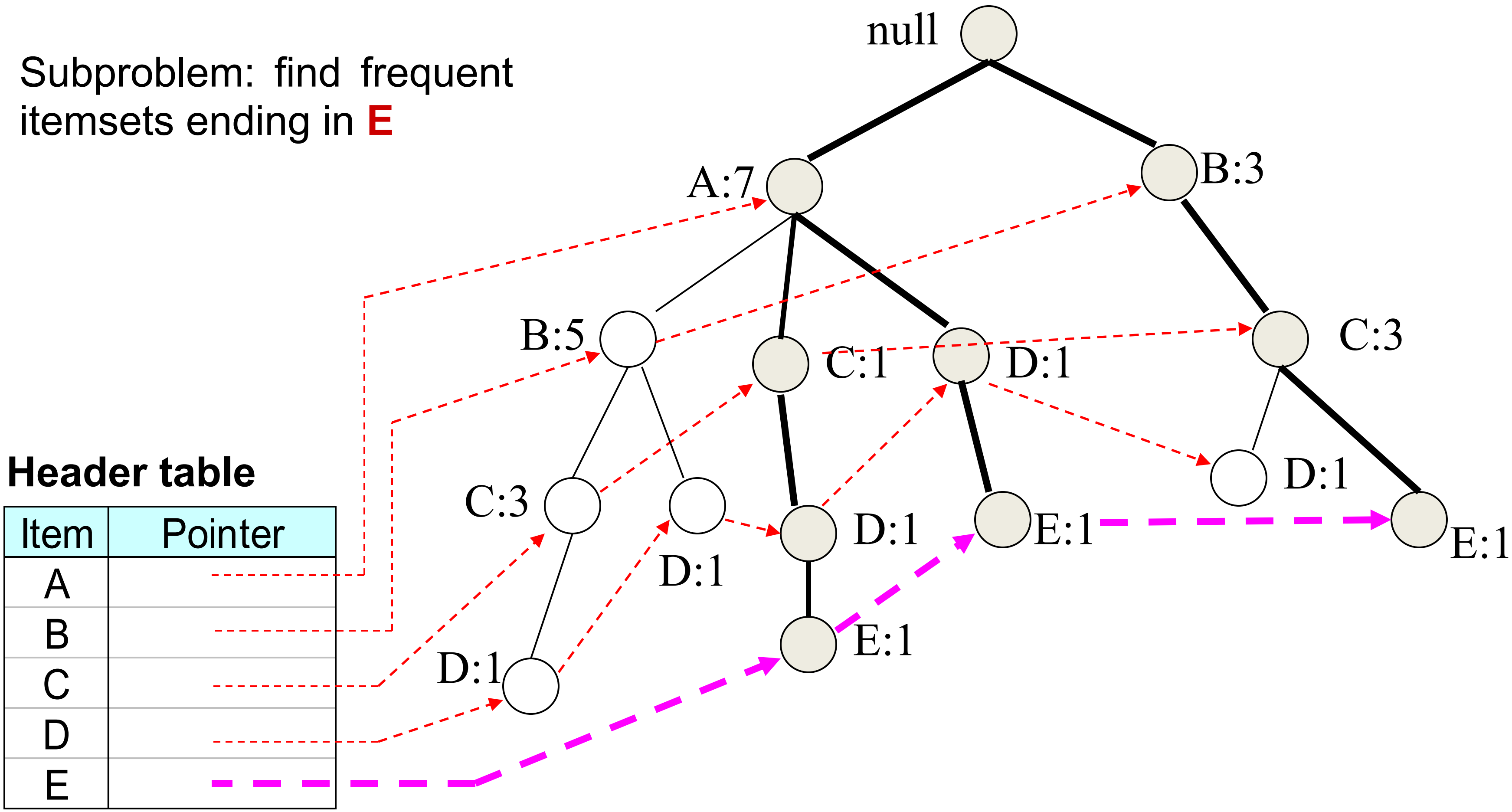


Bottom-up traversal of the tree.

First, itemsets ending in E, then D, etc, each time a suffix-based class

Finding Frequent Itemsets

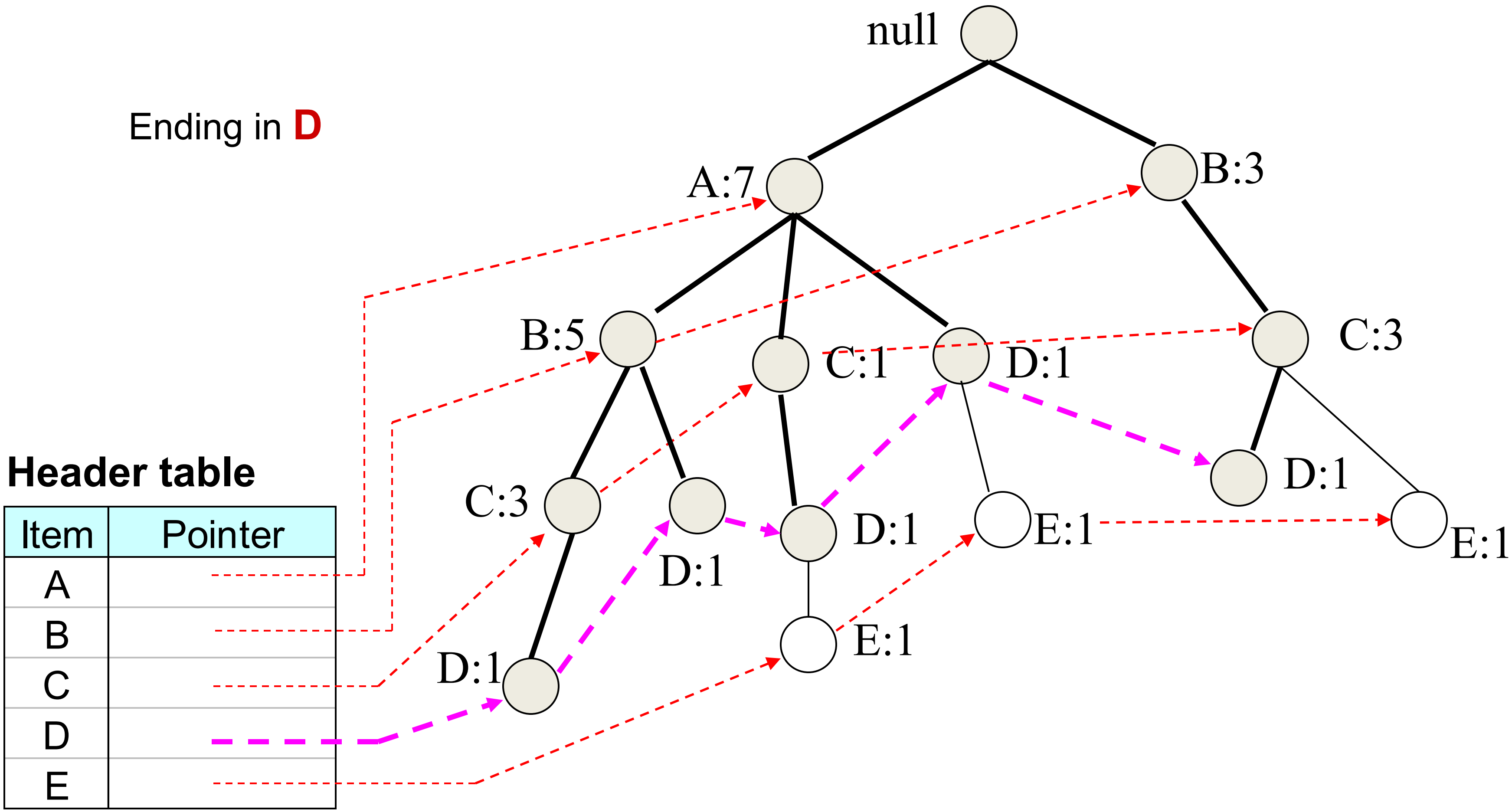
Subproblem: find frequent itemsets ending in **E**



- We will then see how to compute the support for the possible itemsets

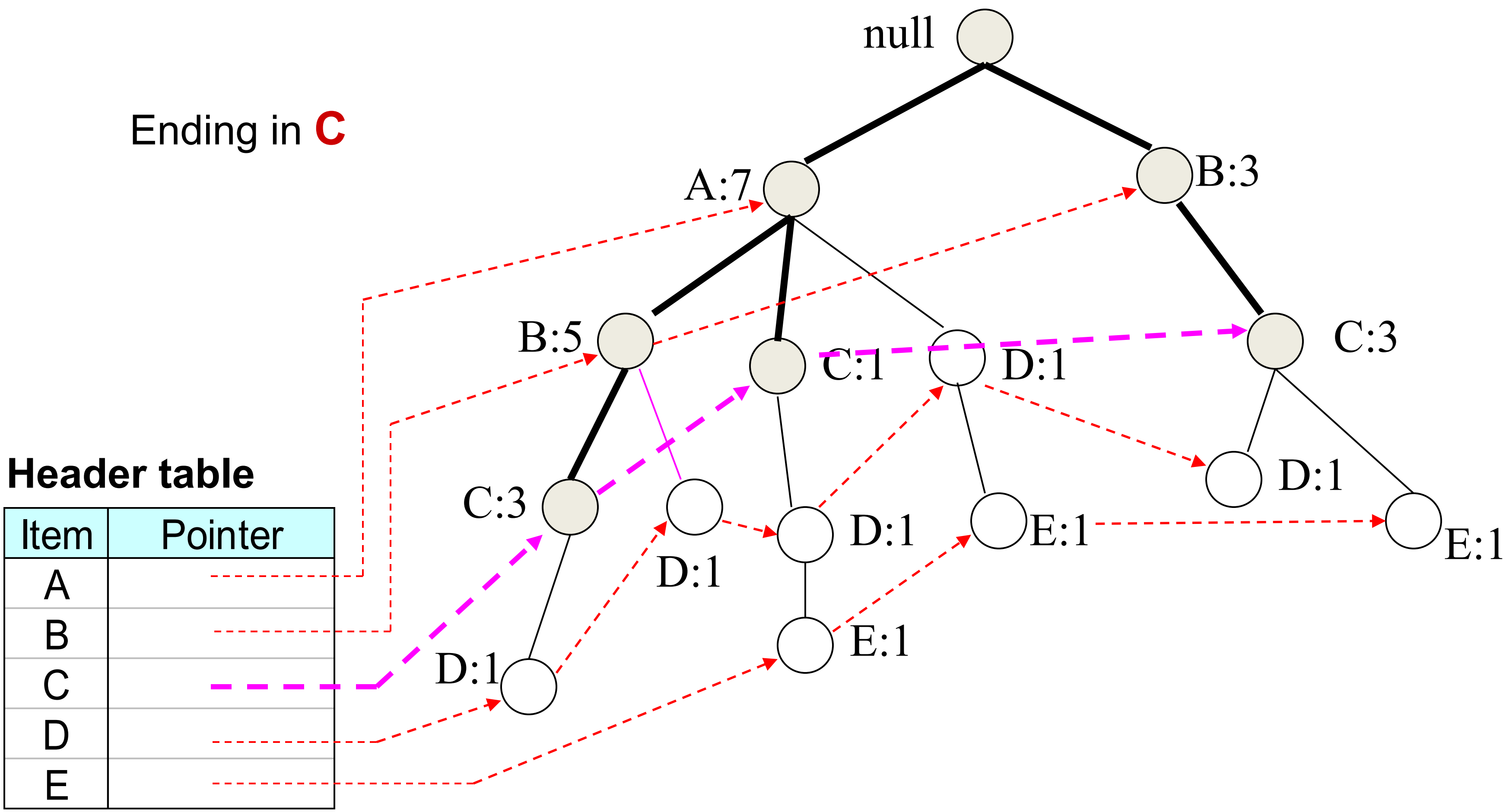
Finding Frequent Itemsets

Ending in **D**



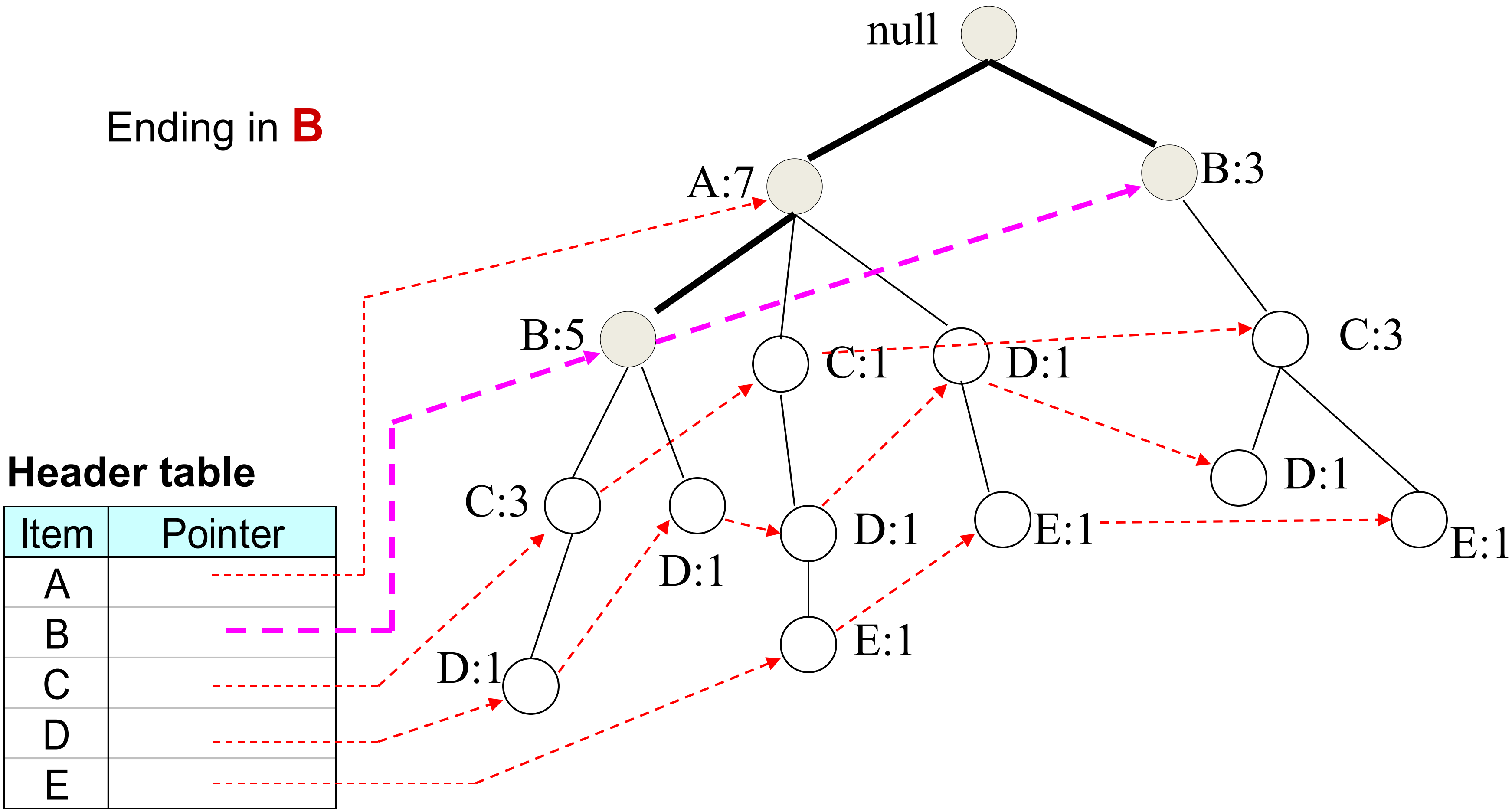
Finding Frequent Itemsets

Ending in **C**



Finding Frequent Itemsets

Ending in **B**

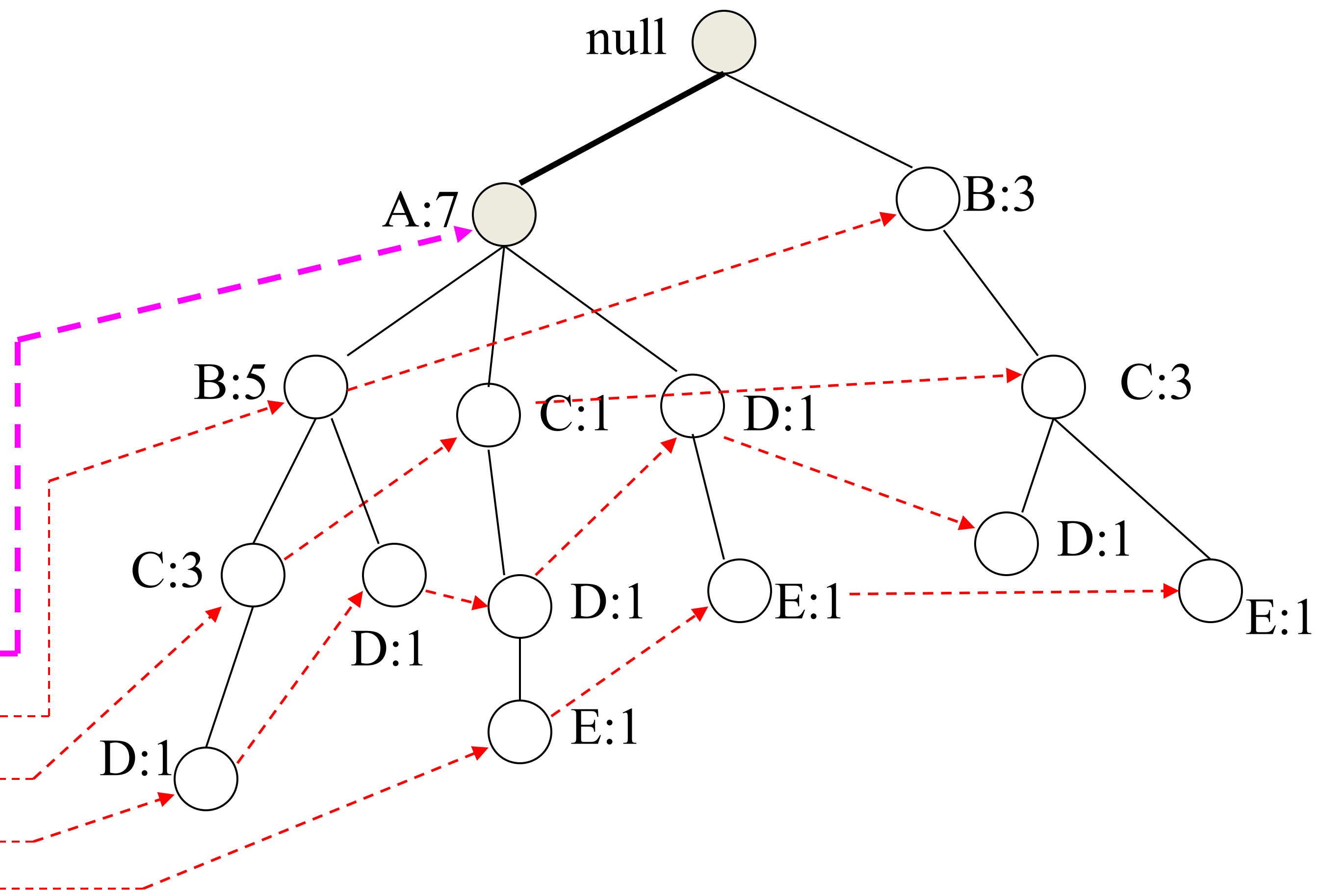


Finding Frequent Itemsets

Ending in **A**

Header table

Item	Pointer
A	
B	
C	
D	
E	



Algorithm

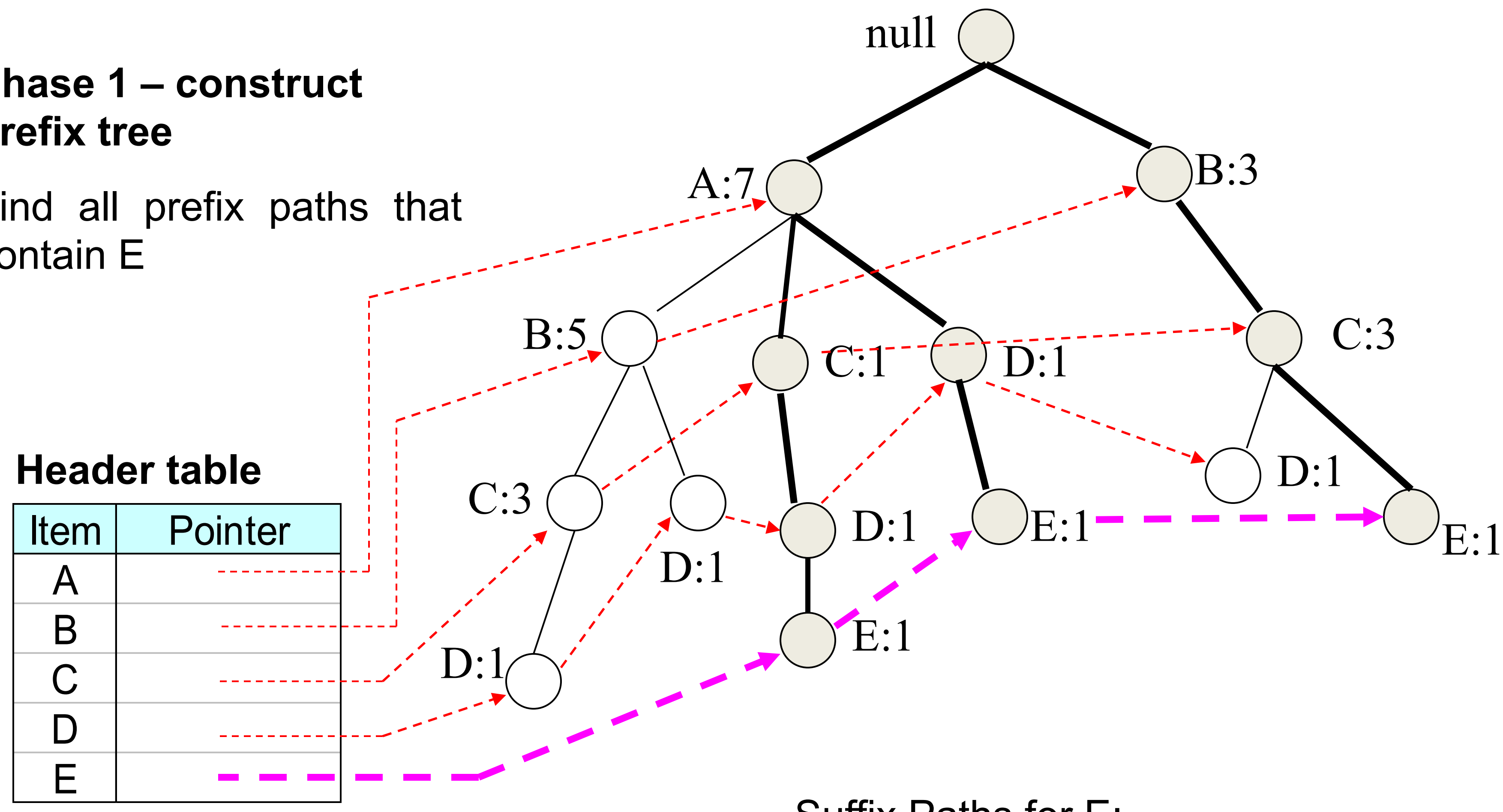
- For each suffix X
- Phase 1
  - Construct the **prefix tree** for X as shown before, and compute the **support** using the header table and the pointers

- Phase 2
  - If X is frequent, construct the conditional FP-tree for X in the following steps
    1. Recompute support
    2. Prune infrequent items
    3. Prune leaves and recurse

Example

Phase 1 – construct prefix tree

Find all prefix paths that contain E



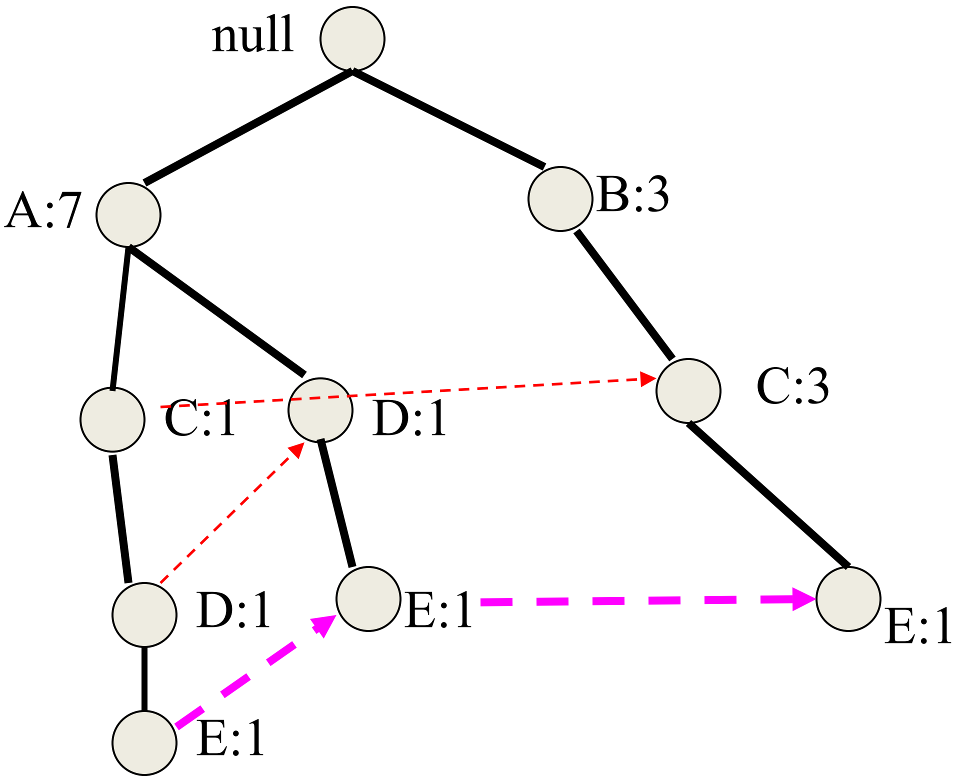
Suffix Paths for E:

{A,C,D,E}, {A,D,E}, {B,C,E}

Example

Phase 1 – construct prefix tree

Find all prefix paths that contain E



Prefix Paths for E:

{A,C,D,E}, {A,D,E}, {B,C,E}

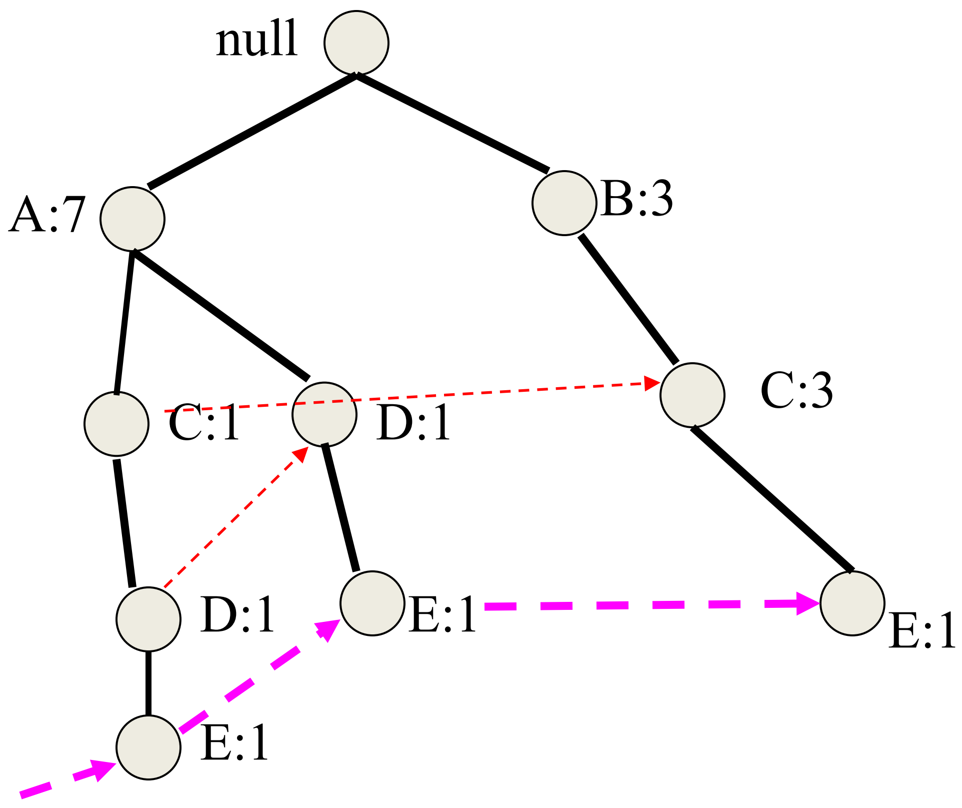
Example

Compute Support for E
(minsup = 2)

How?

Follow pointers while summing up counts: 1+1+1 = 3 > 2

E is frequent



{E} is frequent so we can now consider **suffixes DE, CE, BE, AE**

Example

E is frequent so we proceed with Phase 2

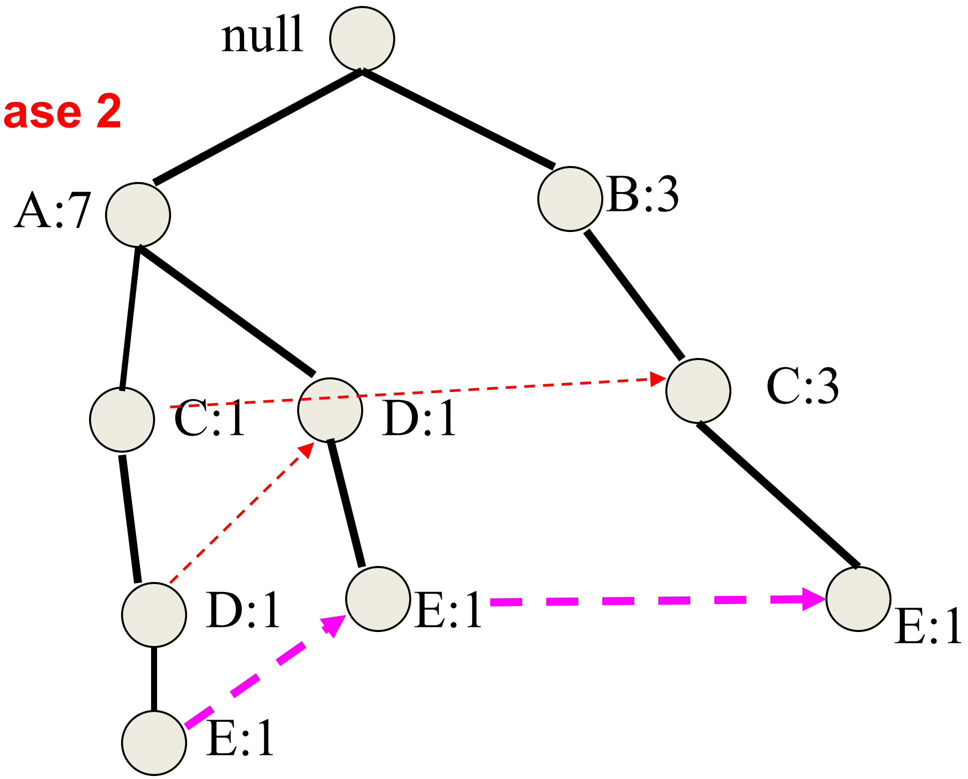
Phase 2

Convert the prefix tree of E into a conditional FP-tree

Two changes

(1) Recompute support

(2) Prune infrequent



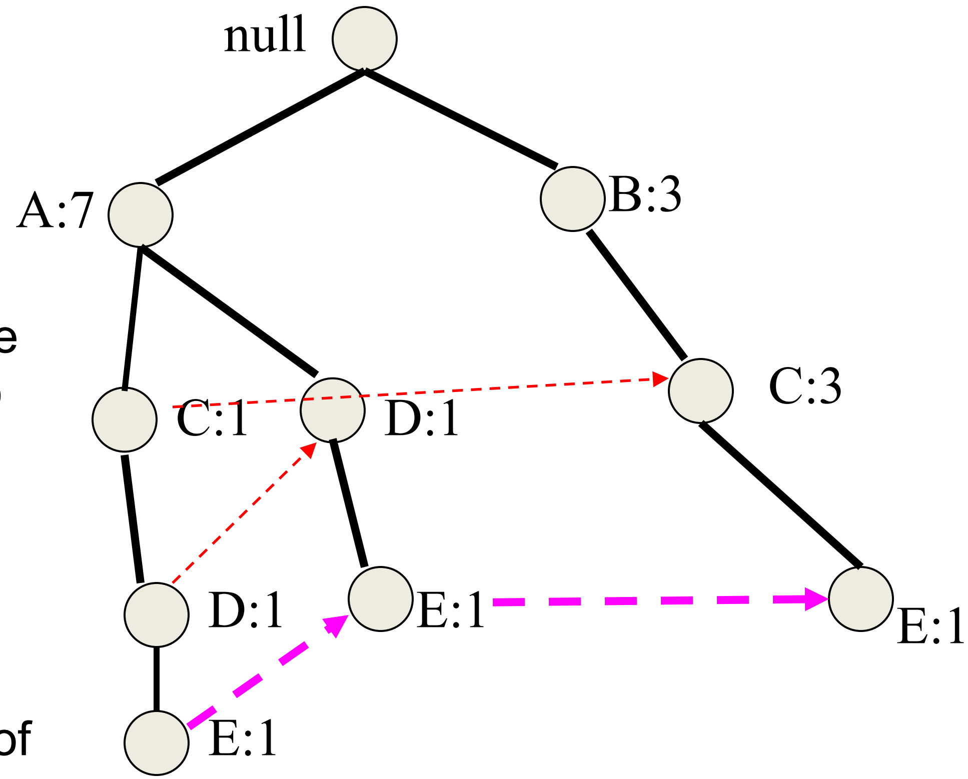
Example

Recompute Support

The support counts for some of the nodes include transactions that do not end in E

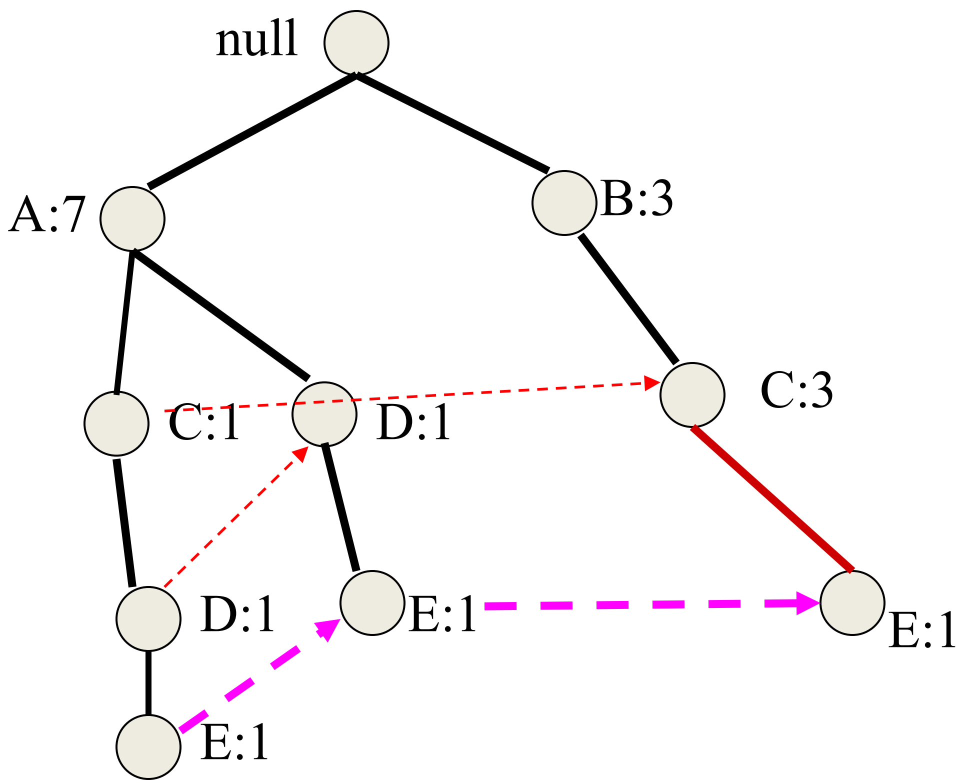
For example in null->B->C->E we count {B, C}

Property to satisfy: The support of any node is equal to the sum of the support of leaves with label E in its subtree

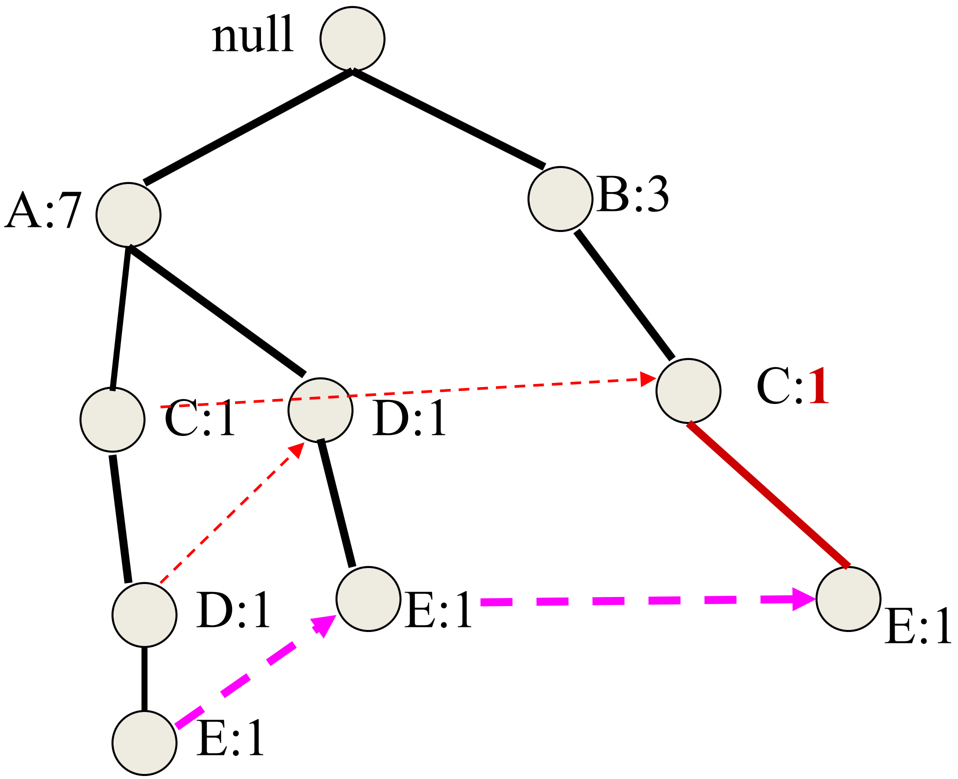


Example

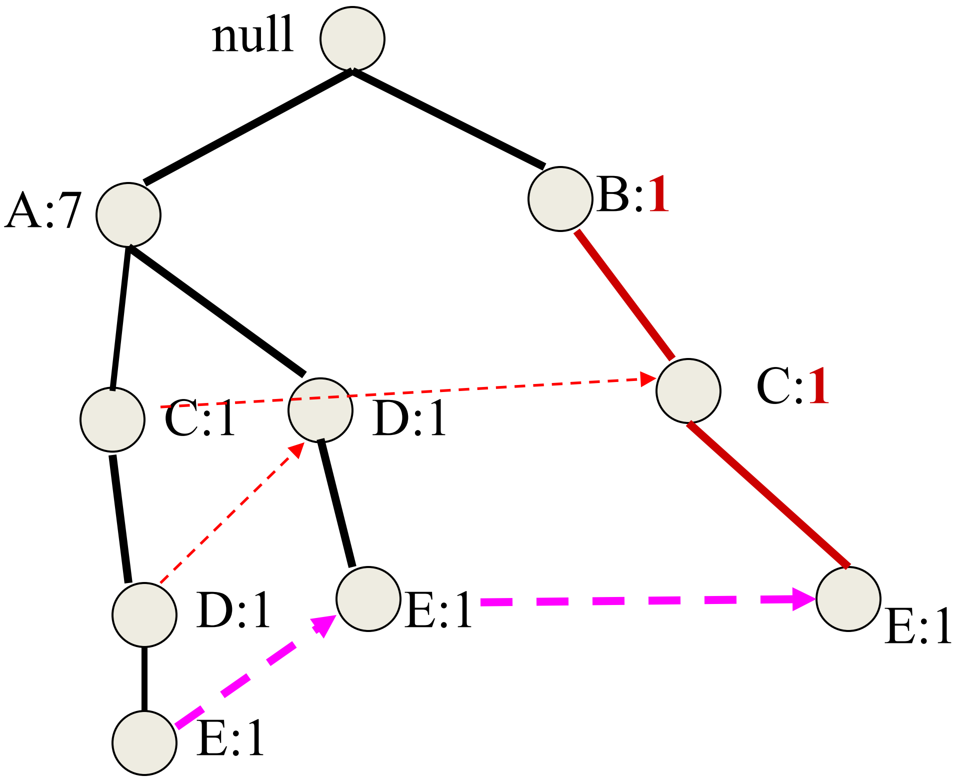
The support of any node is equal to the sum of the support of leaves with label E in its subtree



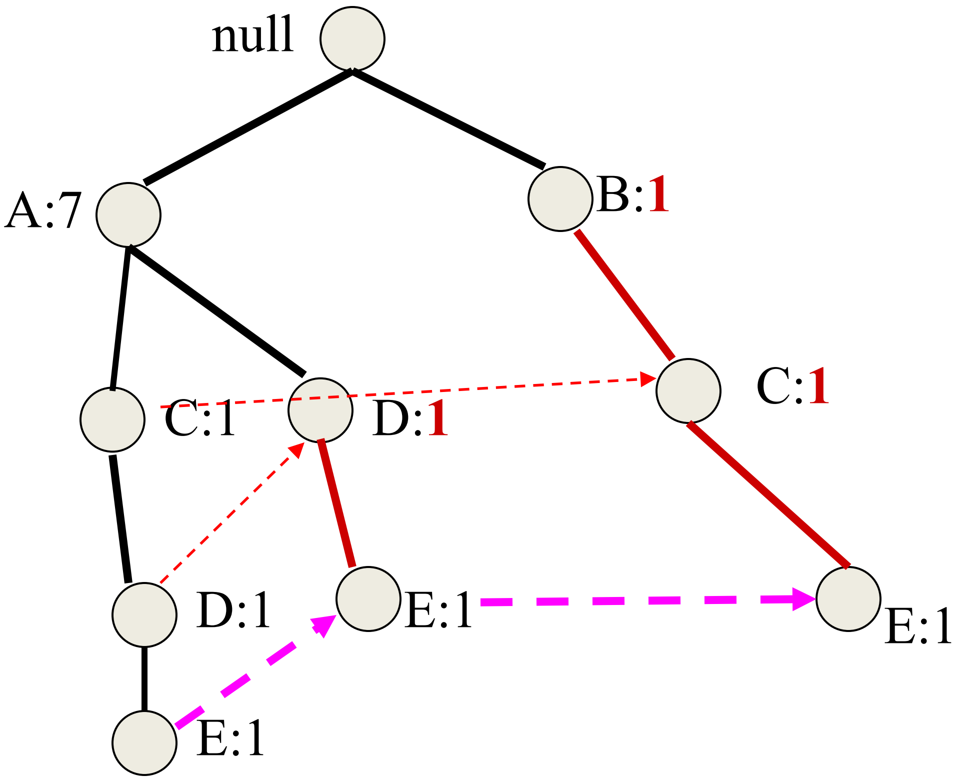
Example



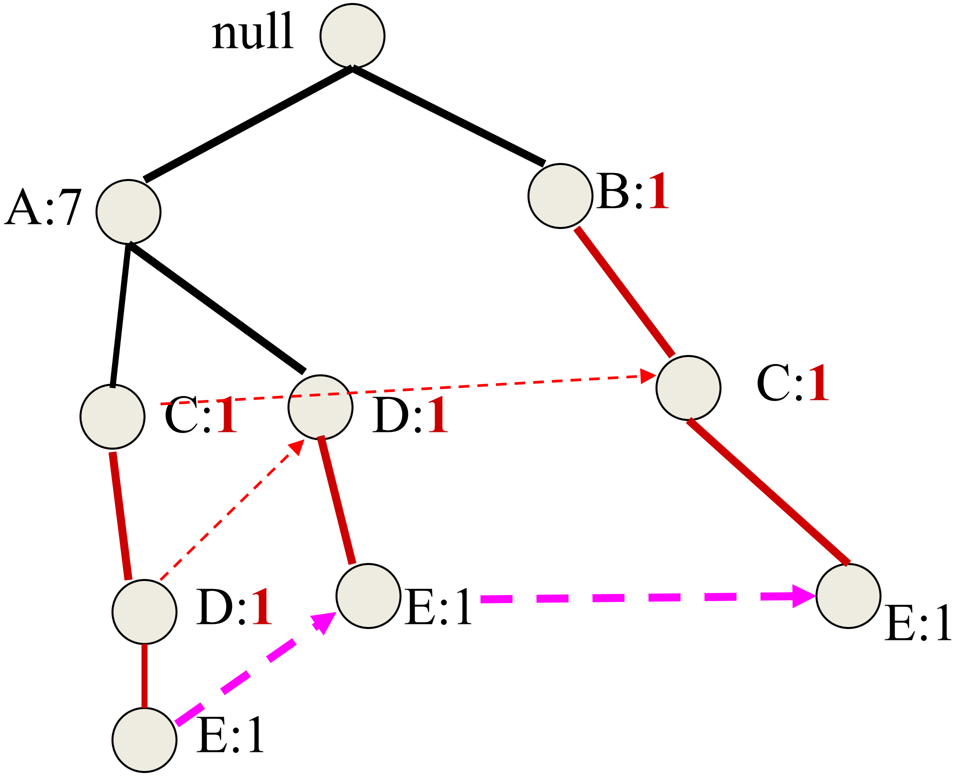
Example



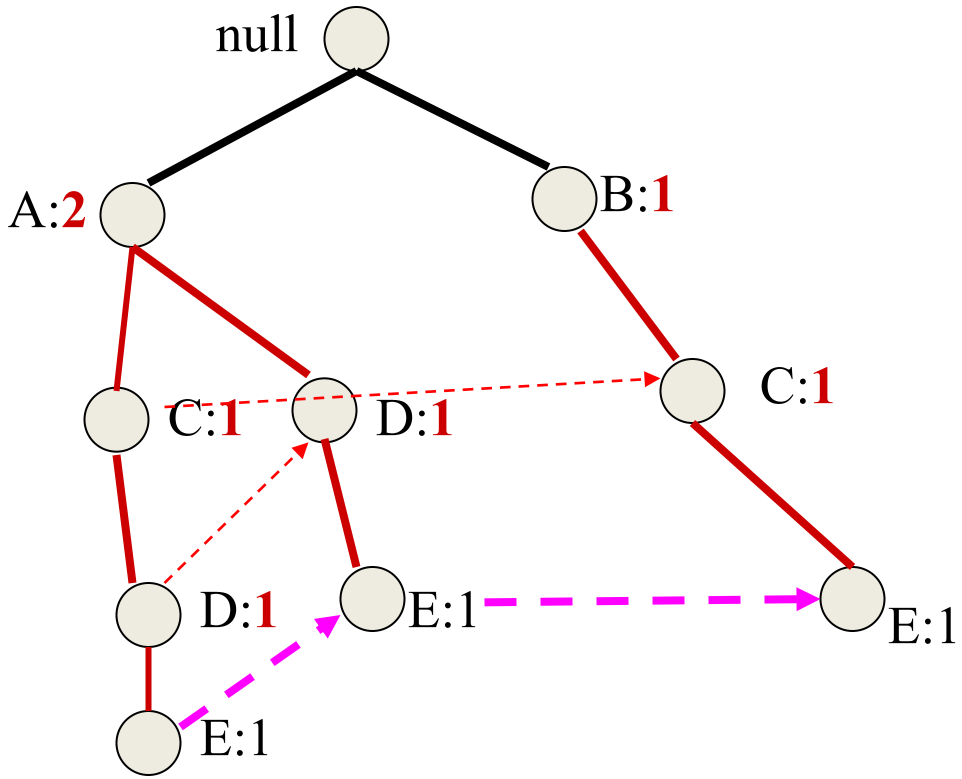
Example



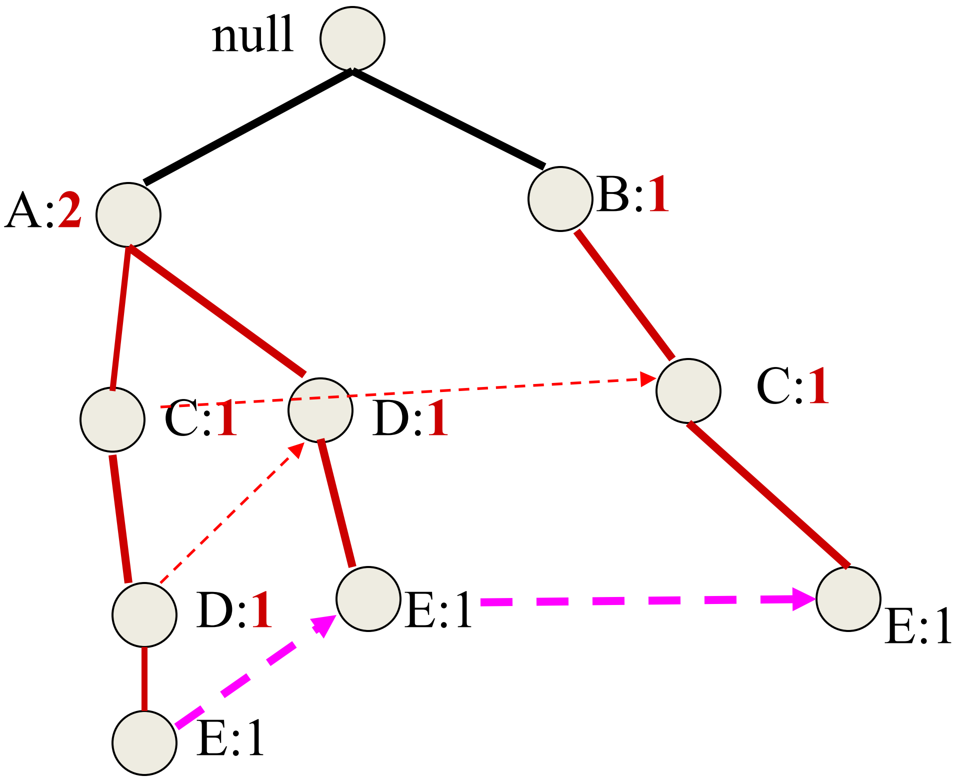
Example



Example



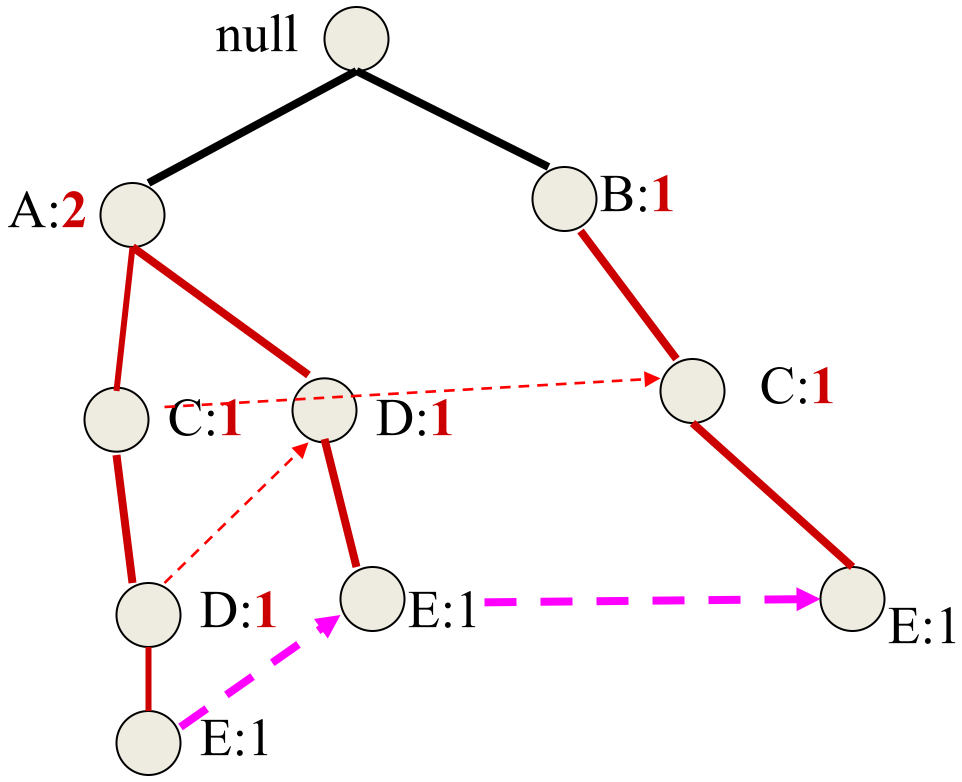
Example



Example

Truncate

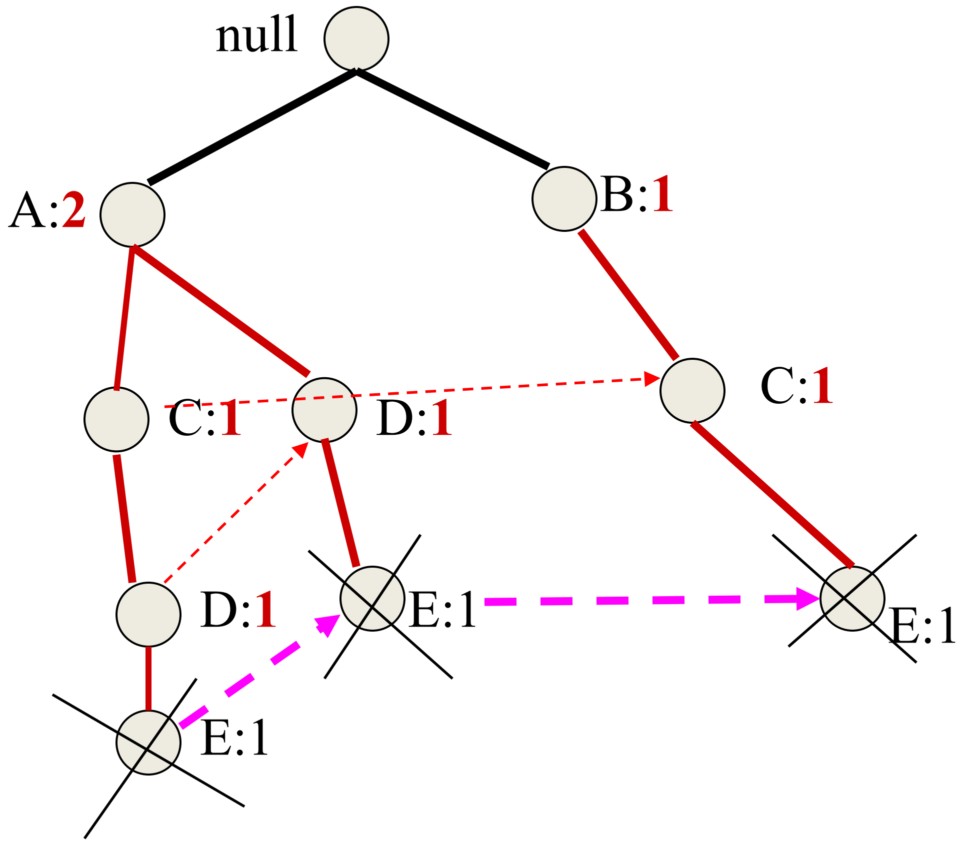
Delete the nodes of E



Example

Truncate

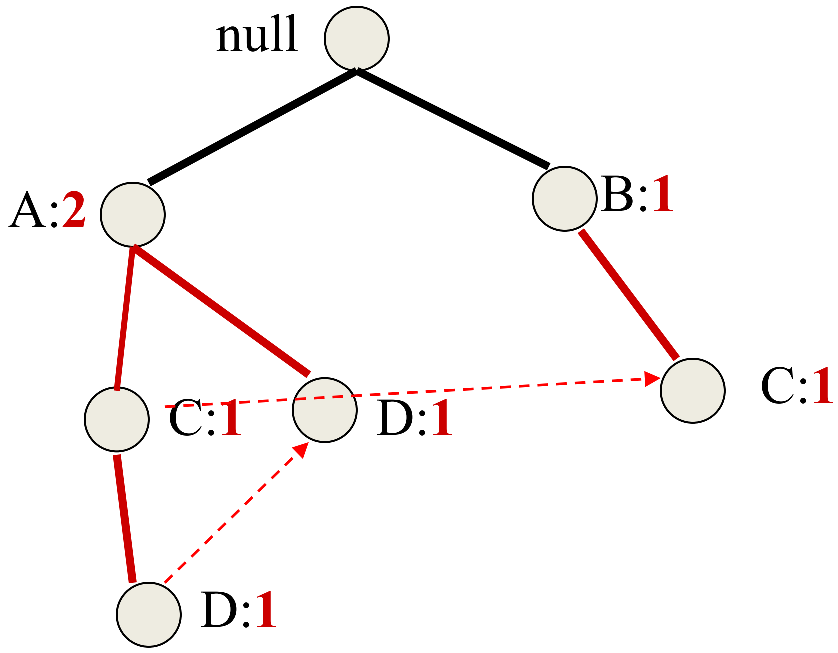
Delete the nodes of E



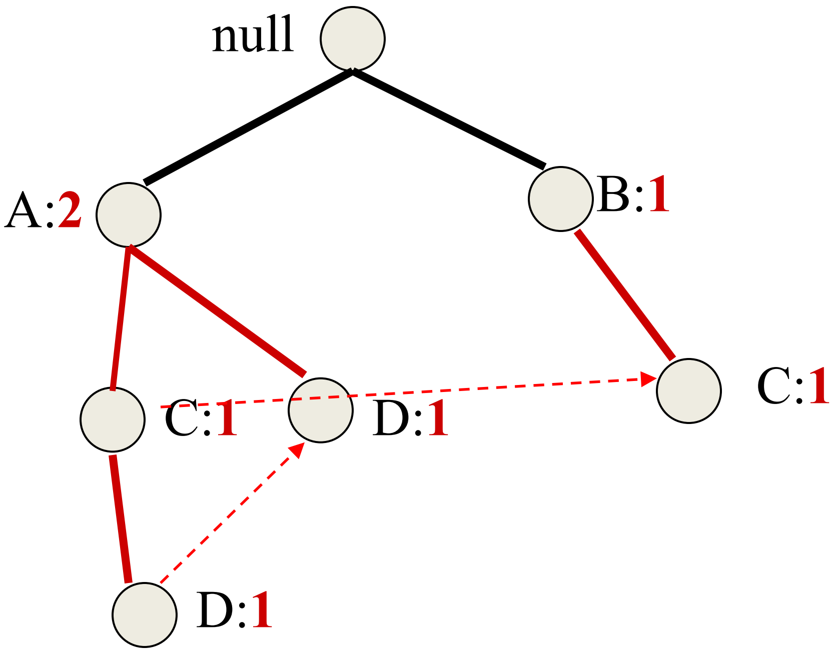
Example

Truncate

Delete the nodes of E



Example



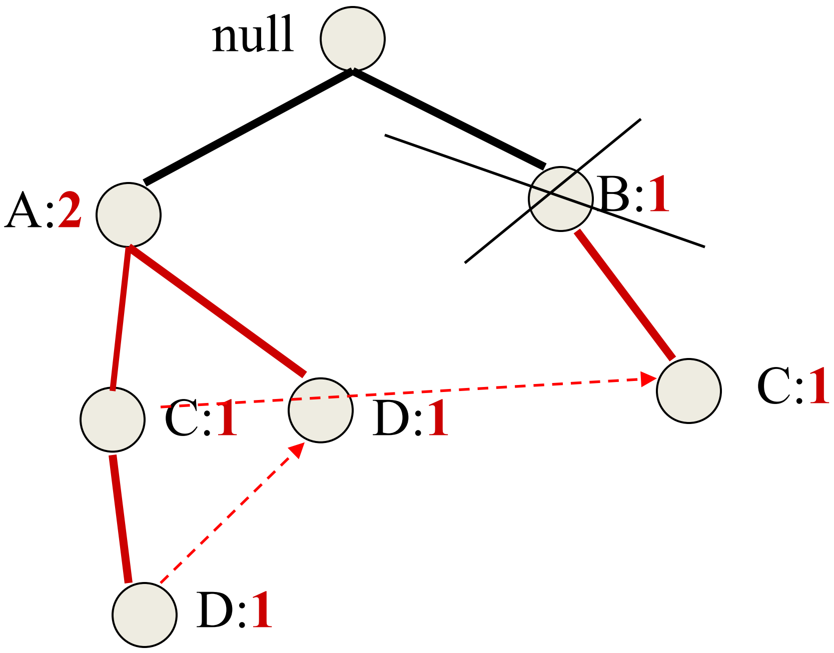
Prune infrequent

In the conditional FP-tree some nodes may have support less than minsup

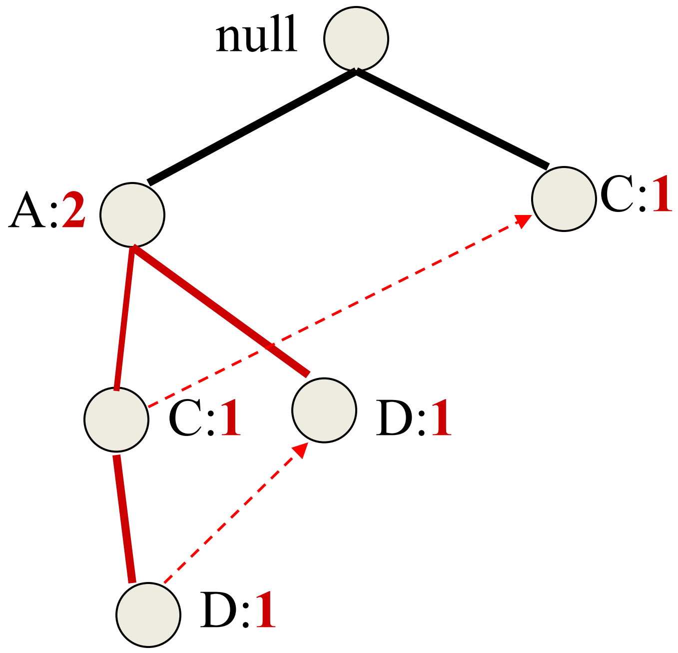
e.g., B needs to be pruned

This means that B appears with E less than minsup times

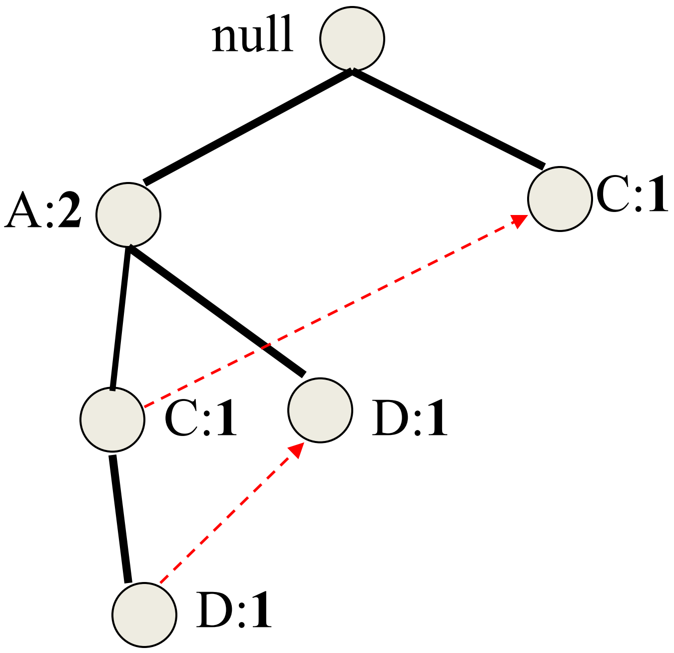
Example



Example



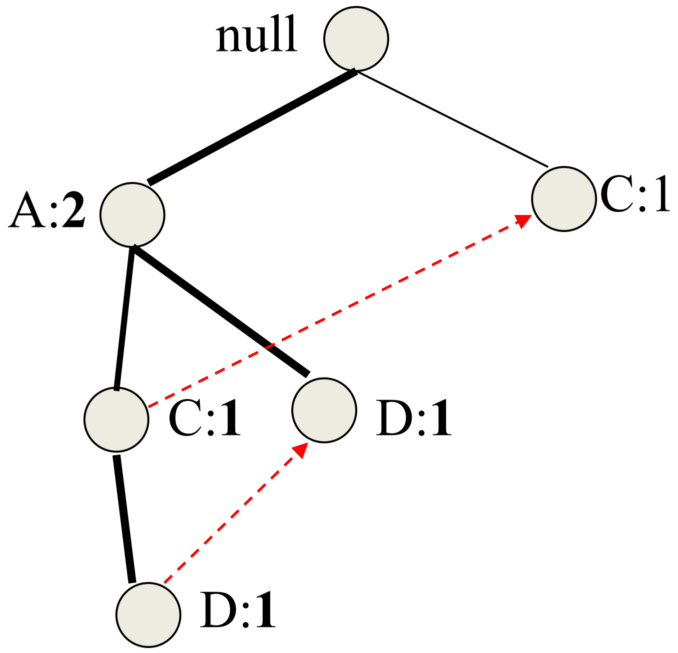
Example



The conditional FP-tree for E

Repeat the algorithm for {D, E}, {C, E}, {A, E}

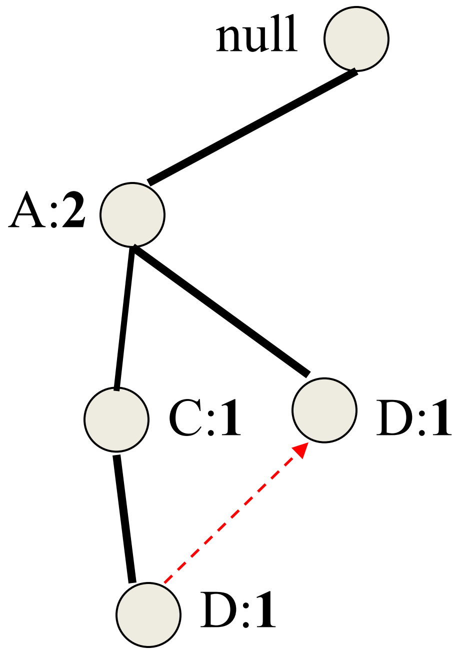
Example



Phase 1

Find all prefix paths that contain D (DE) in the conditional FP-tree

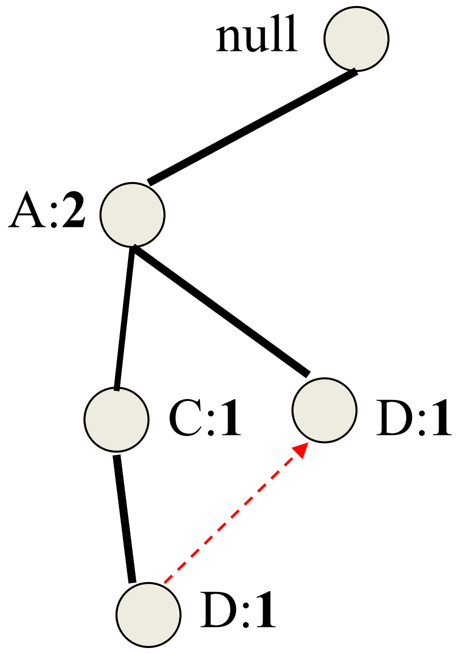
Example



Phase 1

Find all prefix paths that contain D (DE) in the conditional FP-tree

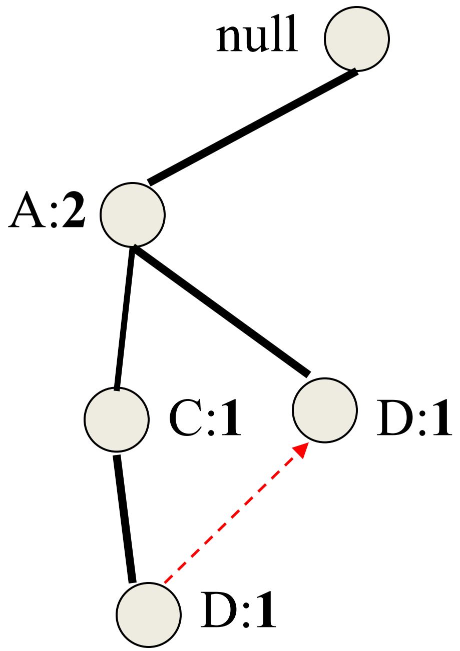
Example



Compute the support of {D,E} by following the pointers in the tree
1+1 = 2 ≥ 2 = minsup

{D,E} is frequent

Example

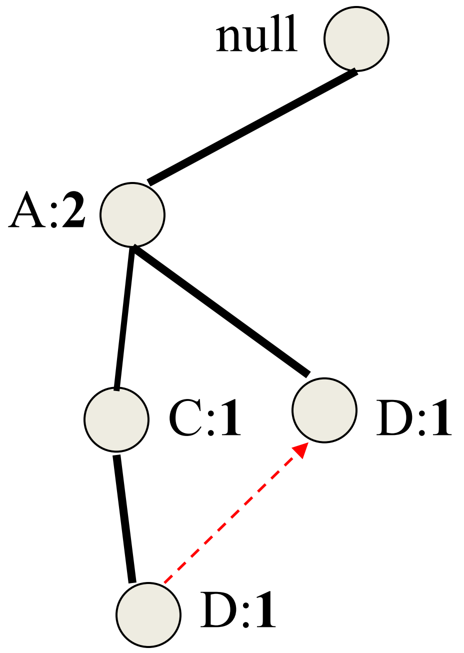


Phase 2

Construct the conditional FP-tree
1. Recompute Support
2. Prune nodes

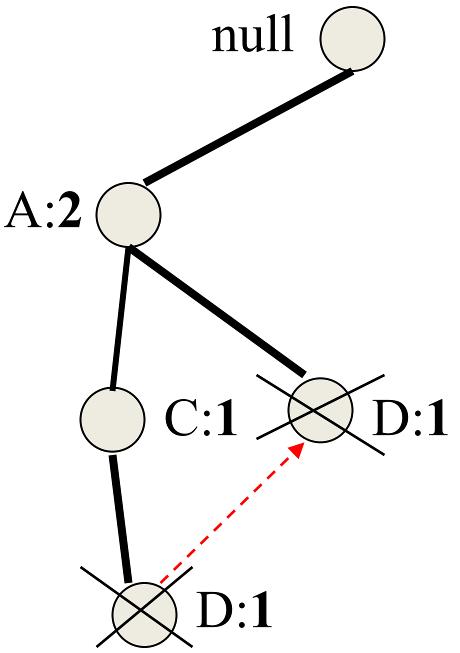
Example

Recompute support

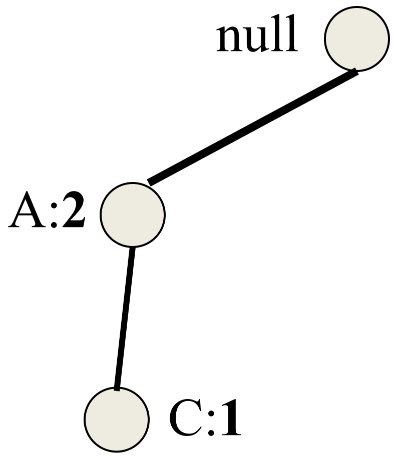


Example

Prune nodes



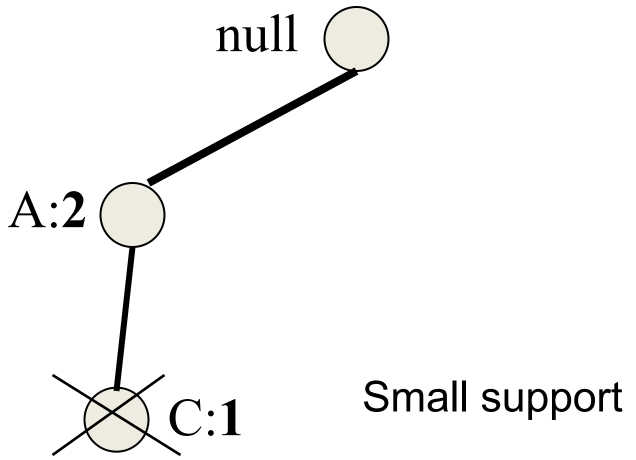
Example



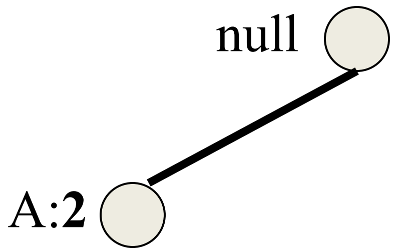
Prune nodes

Example

Prune nodes



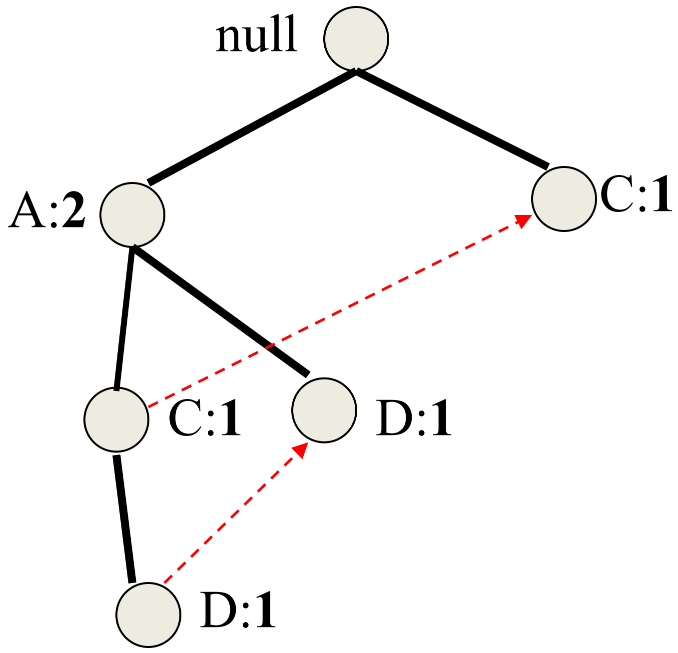
Example



Final condition FP-tree for {D,E}

The support of A is ≥ minsup so {A,D,E} is frequent
Since the tree has a single node we return to the next subproblem

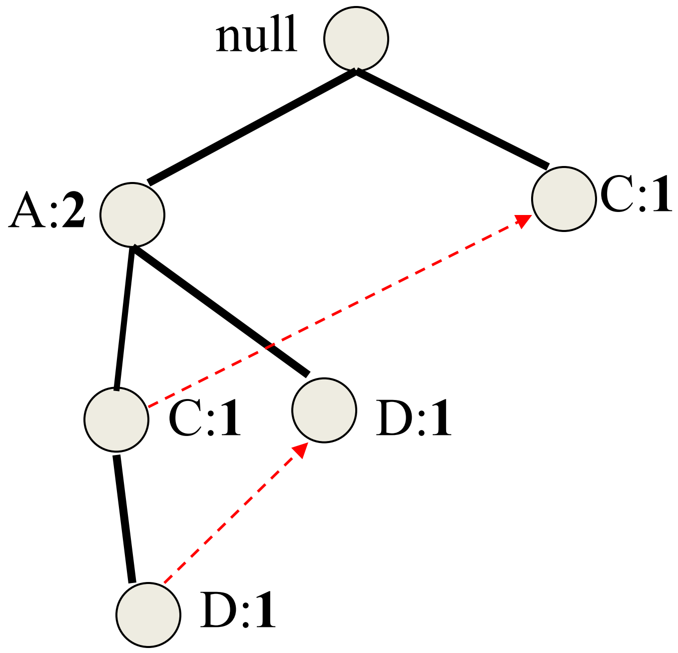
Example



The conditional FP-tree for E

We repeat the algorithm for ~~{D,E}~~, {C,E}, {A,E}

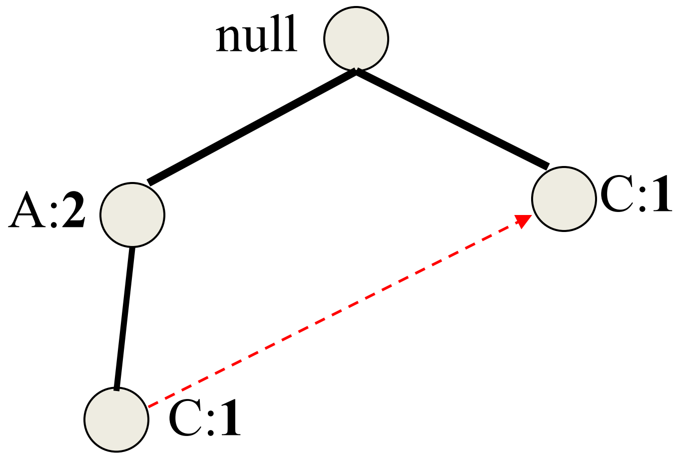
Example



Phase 1

Find all prefix paths that contain C (CE) in the conditional FP-tree

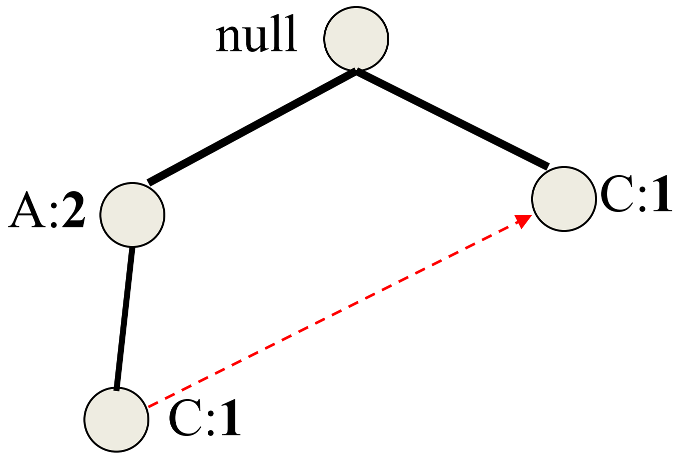
Example



Phase 1

Find all prefix paths that contain C (CE) in the conditional FP-tree

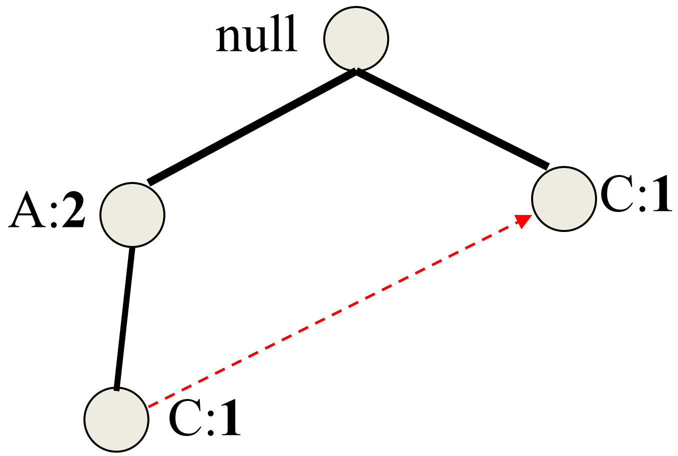
Example



Compute the support of {C,E} by following the pointers in the tree
1+1 = 2 ≥ 2 = minsup

{C,E} is frequent

Example

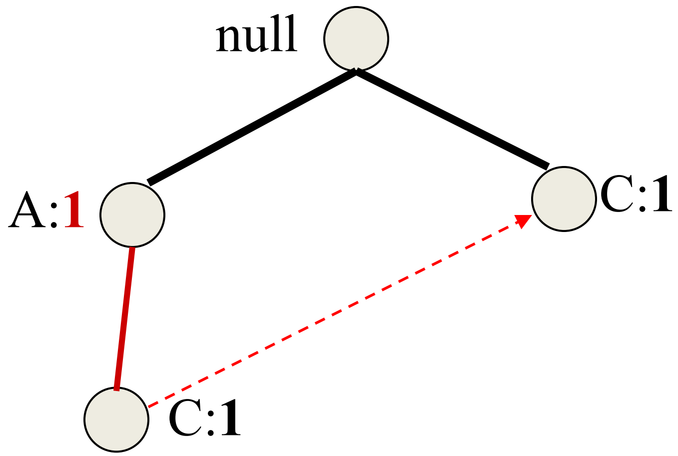


Phase 2

Construct the conditional FP-tree

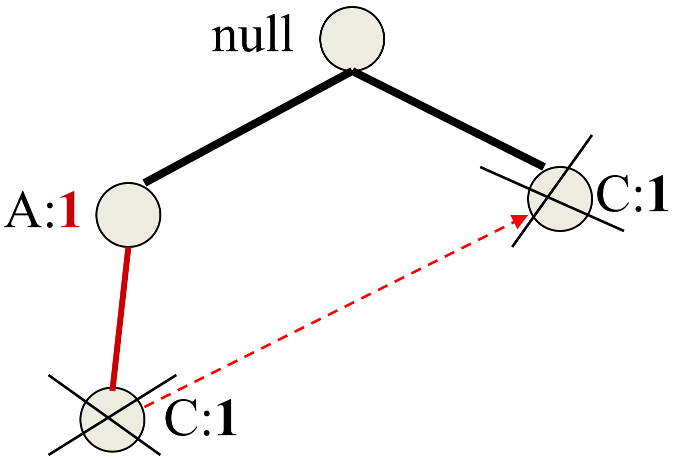
1. Recompute Support
2. Prune nodes

Example



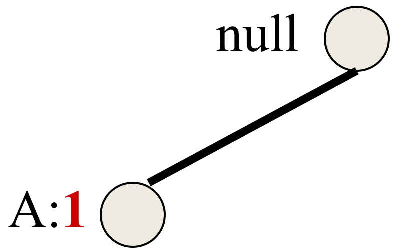
Recompute support

Example



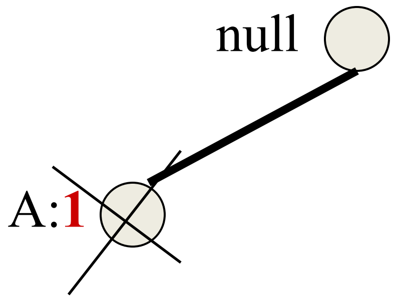
Prune nodes

Example



Prune nodes

Example



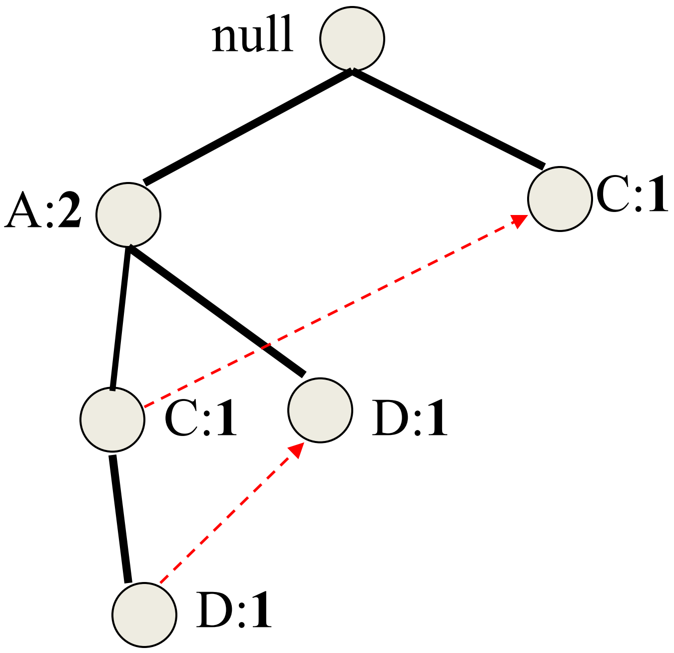
Prune nodes

Example

Prune nodes

Return to the previous subproblem

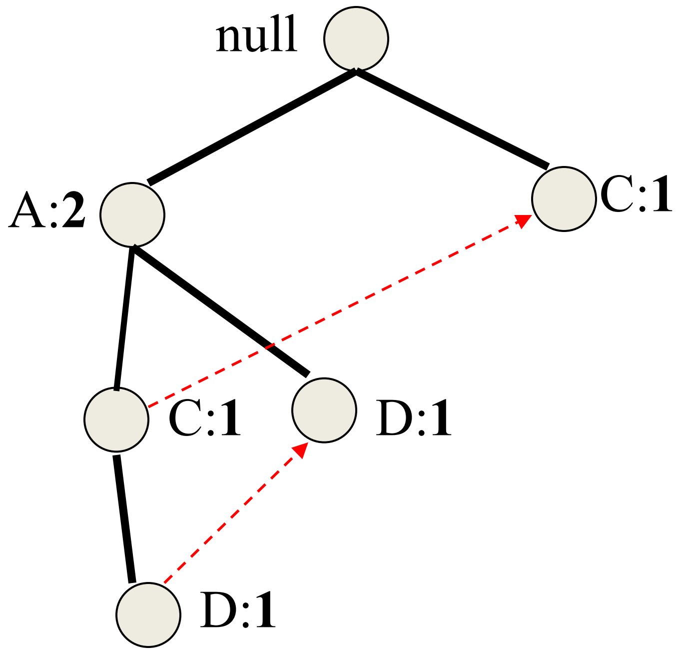
Example



The conditional FP-tree for E

We repeat the algorithm for ~~{D,E}, {C,E},~~ {A,E}

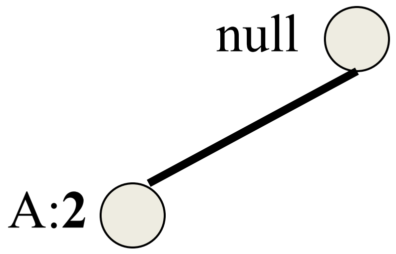
Example



Phase 1

Find all prefix paths that contain A (AE) in the conditional FP-tree

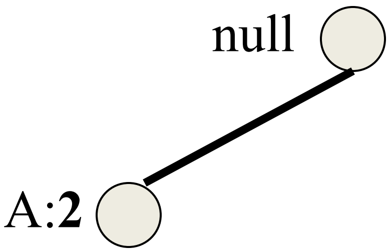
Example



Phase 1

Find all prefix paths that contain A (AE) in the conditional FP-tree

Example



Compute the support of {A,E} by following the pointers in the tree
2 ≥ minsup

{A,E} is frequent

There is no conditional FP-tree for {A,E}

Example

- So for E we have the following frequent itemsets

{E}, {D,E}, {C,E}, {A,E} {ADE}

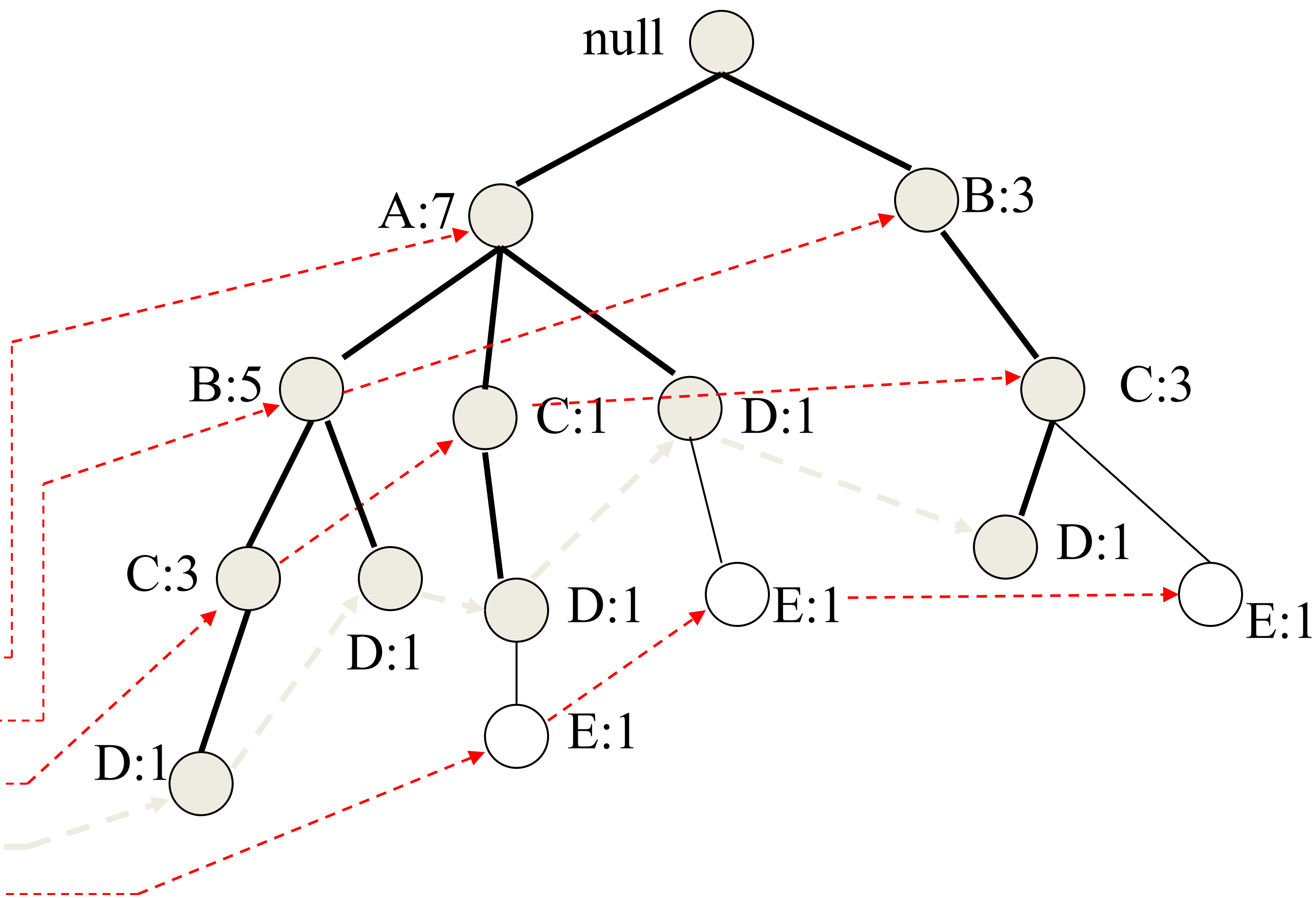
- We proceed with D

Example

Ending in **D**

Header table

Item	Pointer
A	
B	
C	
D	
E	



Example

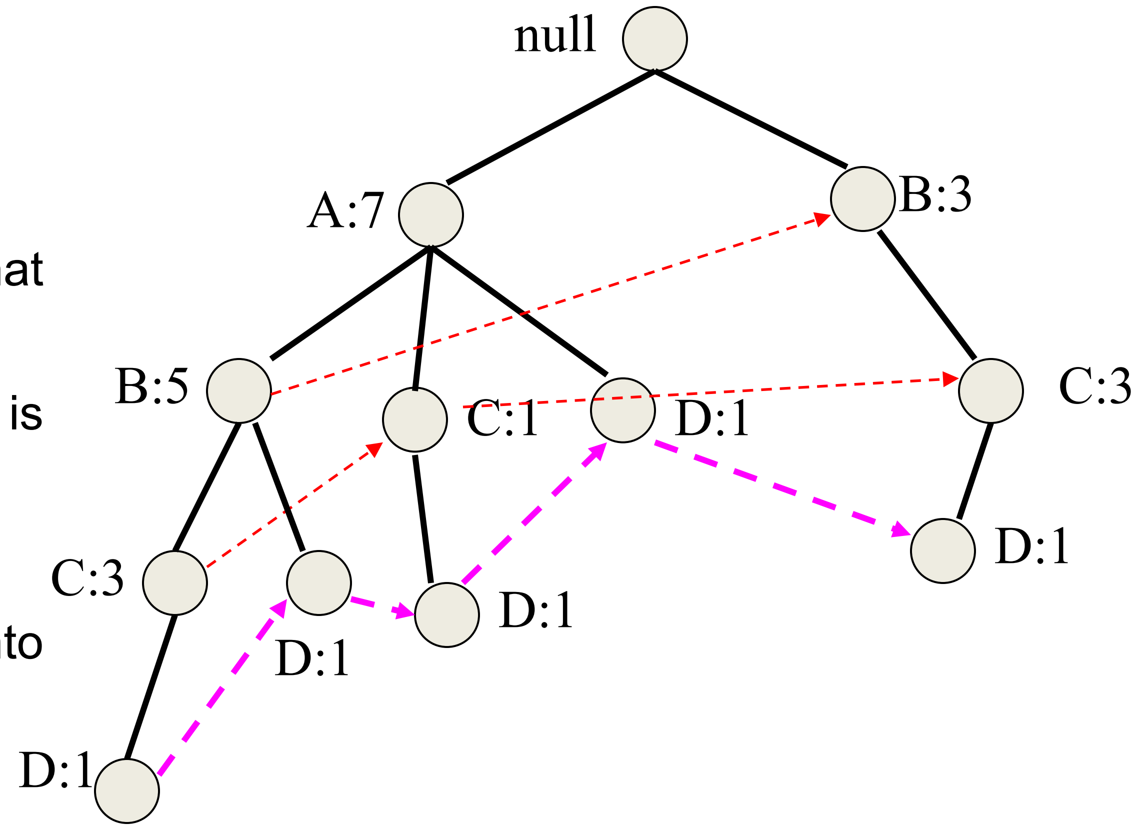
Phase 1 – construct prefix tree

Find all prefix paths that contain D

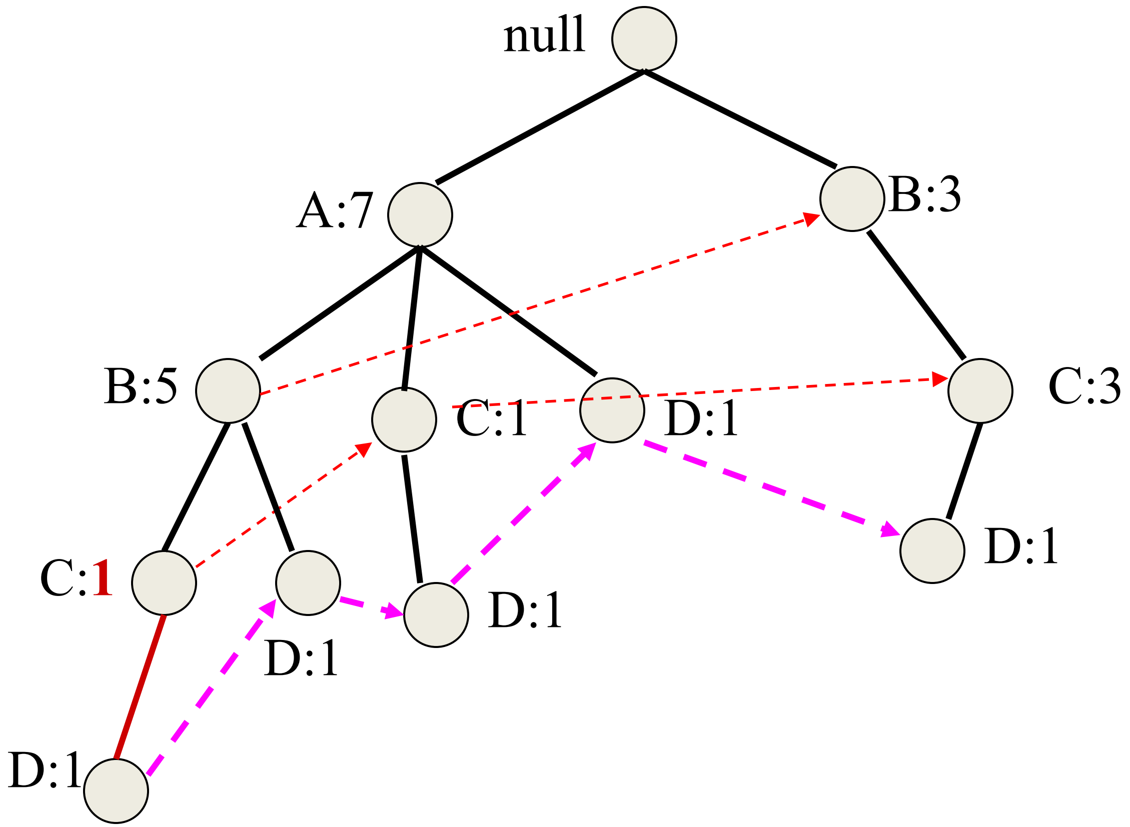
Support 5 > minsup, D is frequent

Phase 2

Convert prefix tree into conditional FP-tree

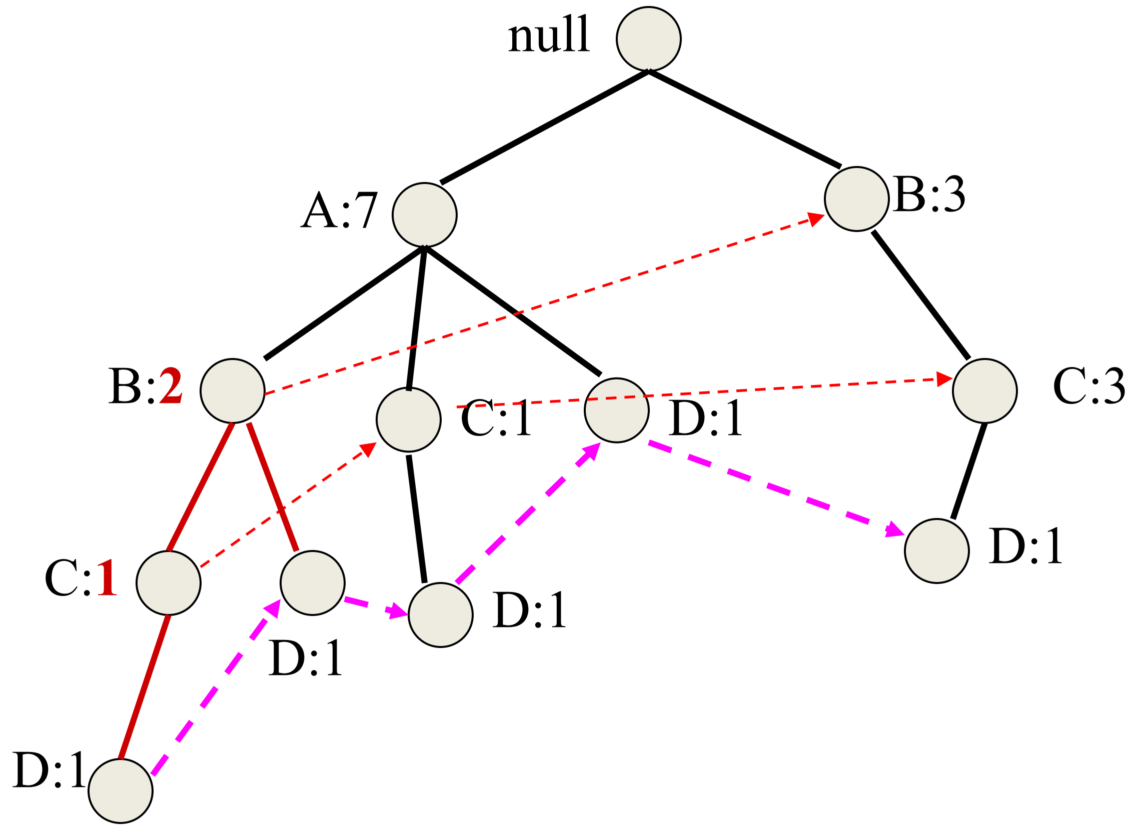


Example



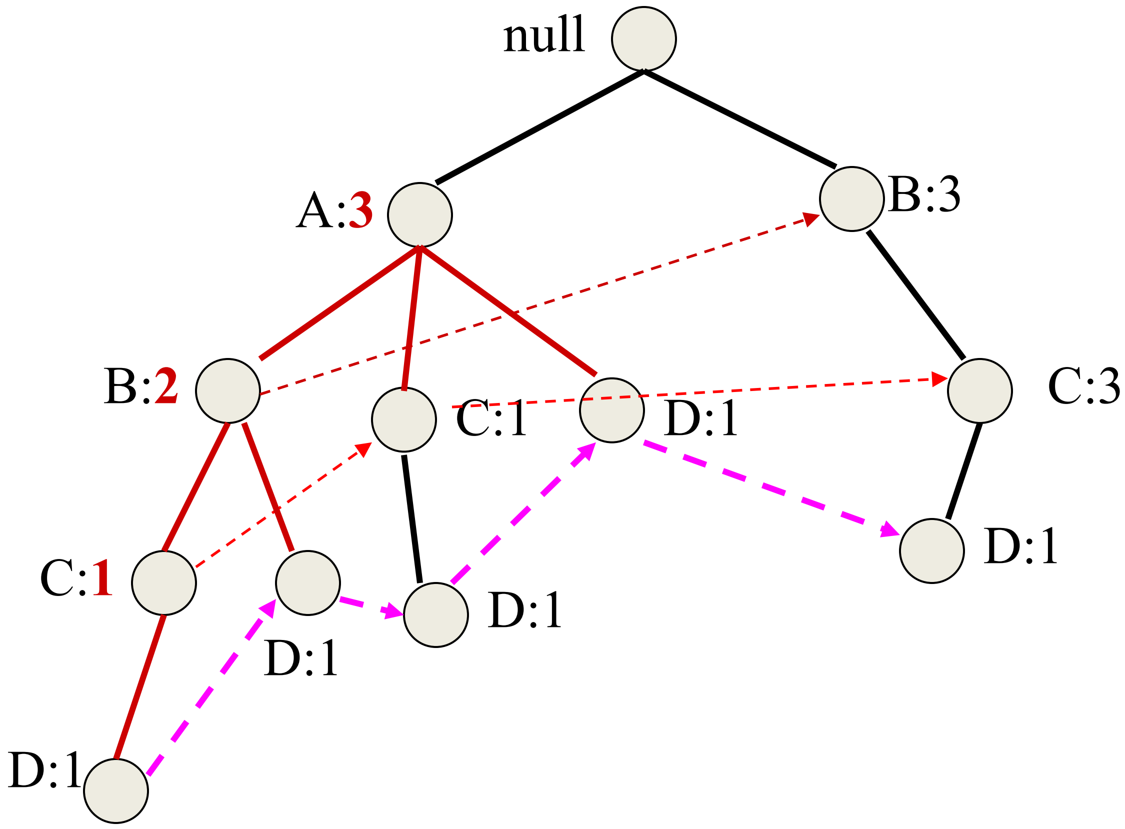
Recompute support

Example



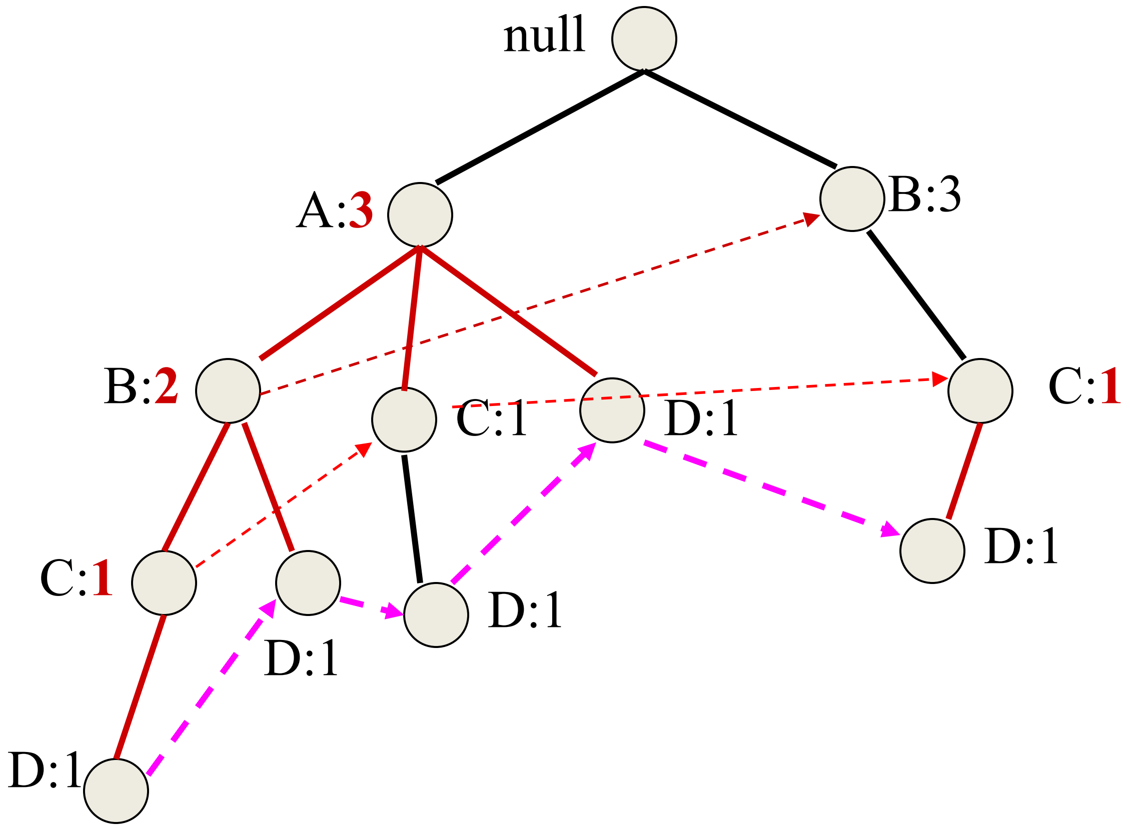
Recompute support

Example



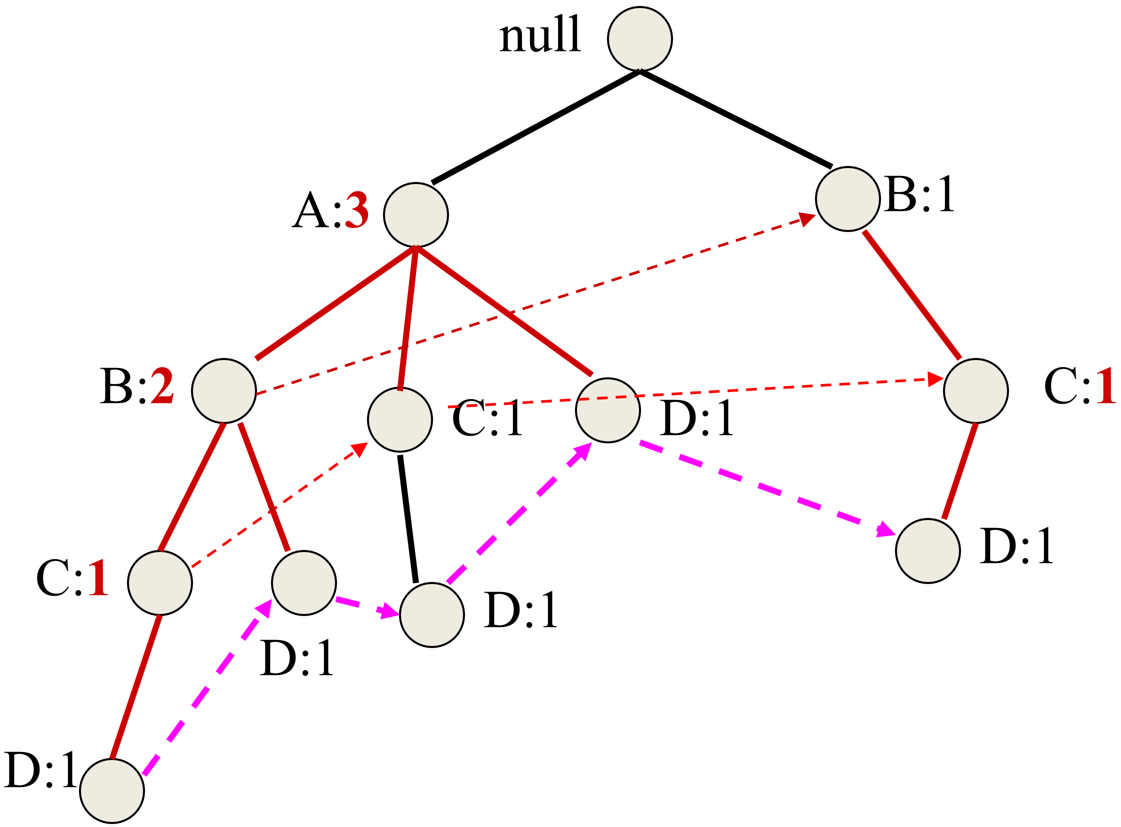
Recompute support

Example



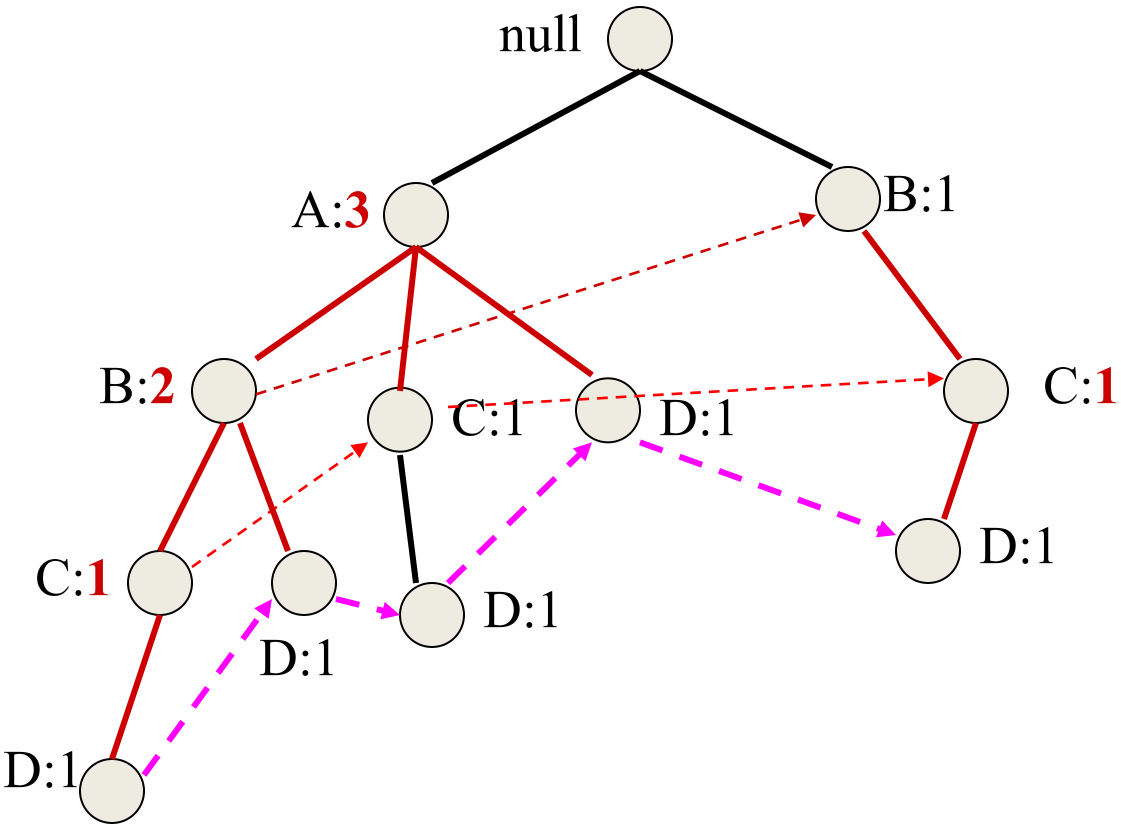
Recompute support

Example



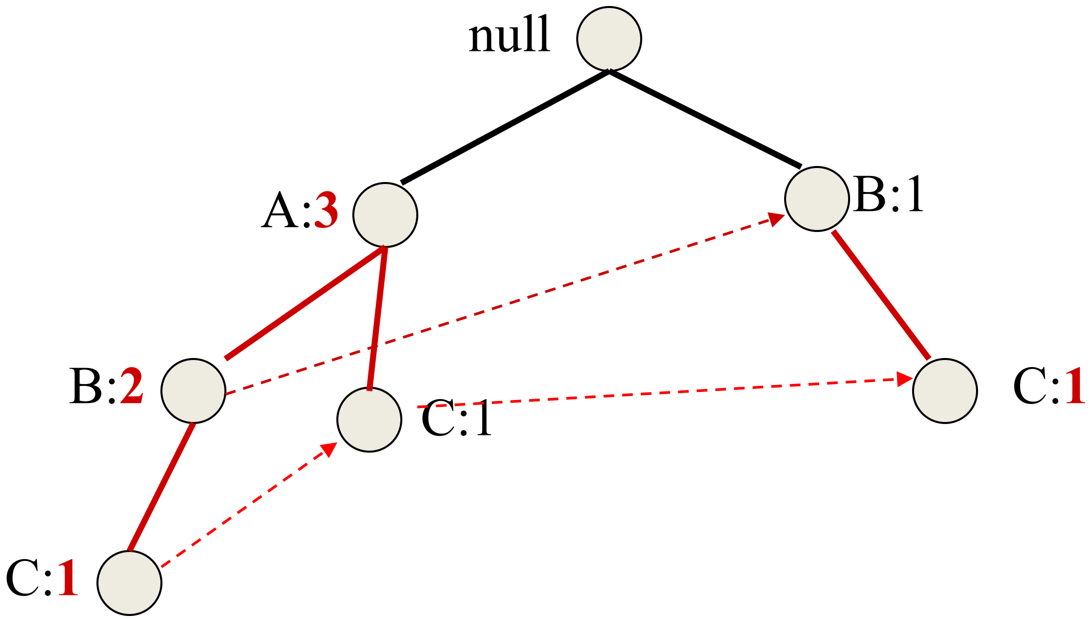
Recompute support

Example



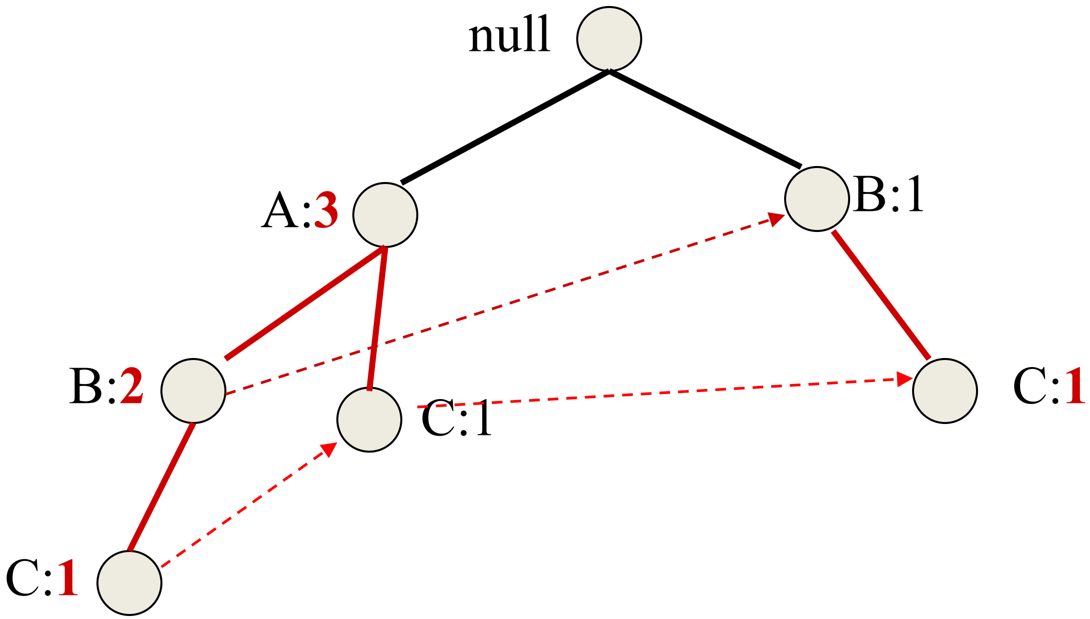
Prune nodes

Example



Prune nodes

Example



Construct conditional FP-trees for {C,D}, {B,D}, {A,D}

And so on….

Observations

- At each recursive step we solve a subproblem
  - Construct the prefix tree
  - Compute the new support
  - Prune nodes
- Subproblems are disjoint so we never consider the same itemset twice

- **Support computation is efficient** – happens together with the computation of the frequent itemsets.

Observations

- The efficiency of the algorithm depends on the compaction factor of the dataset

- If the tree is bushy then the algorithm does not work well, it increases a lot of number of subproblems that need to be solved.