Association Analysis:
FP-growth for Frequent Itemsets

葉建華

jhyeh@mail.au.edu.tw

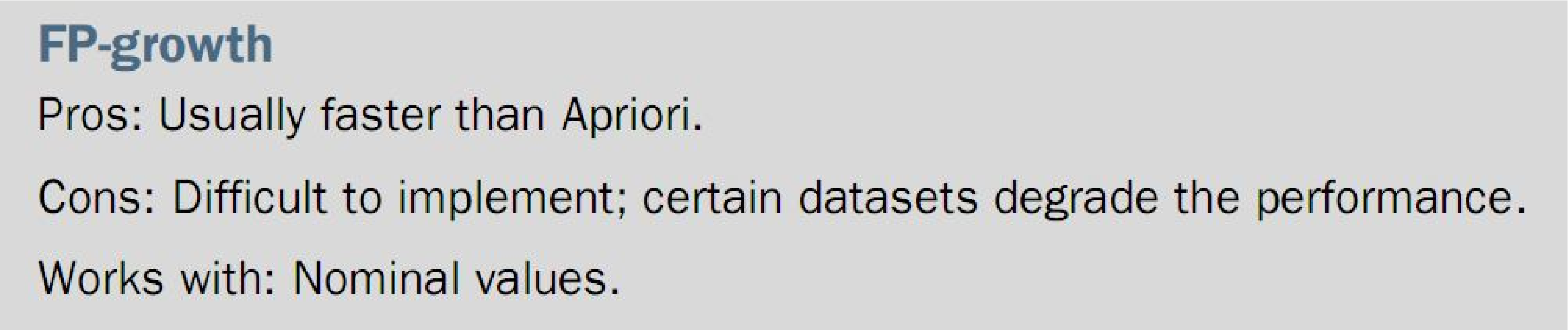
http://jhyeh.csie.au.edu.tw/

Scenario

- Two of the most common ways of looking at things in the dataset
  - Frequent itemsets
  - Association rules based frequent itemsets
- FP-growth algorithm
  - Faster than Apriori algorithm
  - Apriori scans the dataset for every potential frequent item
  - FP-growth requires only two scans of the database

FP-growth Algorithm

- Two steps
  - Build the FP-tree
  - Mine frequent itemsets from the FP-tree



FP-growth

Pros: Usually faster than Apriori.

Cons: Difficult to implement; certain datasets degrade the performance.

Works with: Nominal values.

FP(Frequent Pattern)-tree

- A compact data structure where the FP-growth algorithm stores data
- A tree branch connects similar items
- The linked items can be thought of as a linked list
- An item can appear multiple times
- Used to store the frequency of occurrence for sets of items

FP-tree Example

- Minimum support: 3

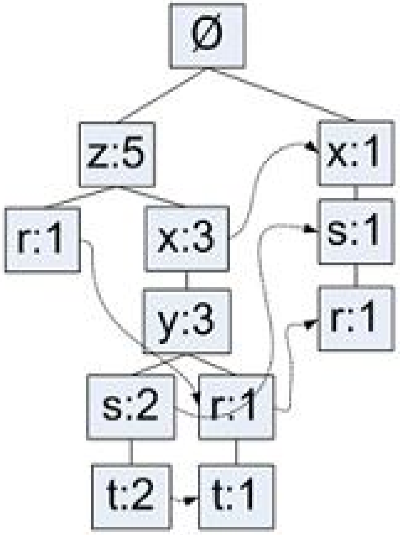
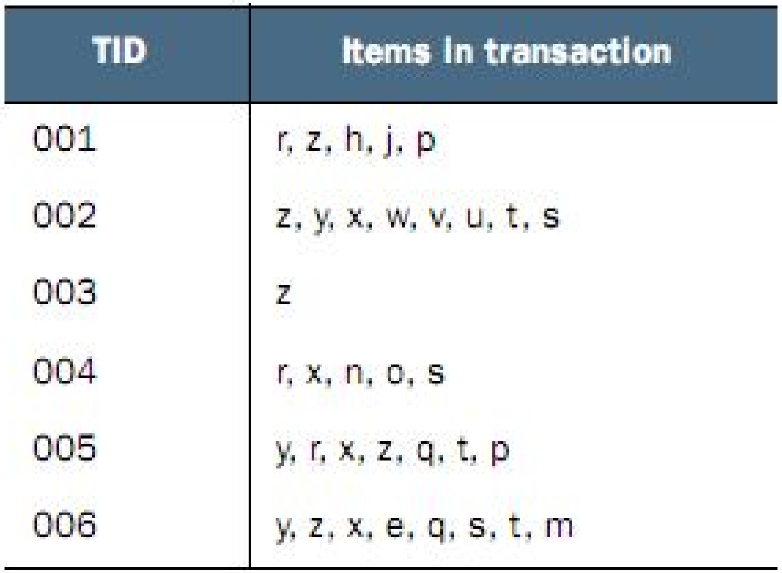


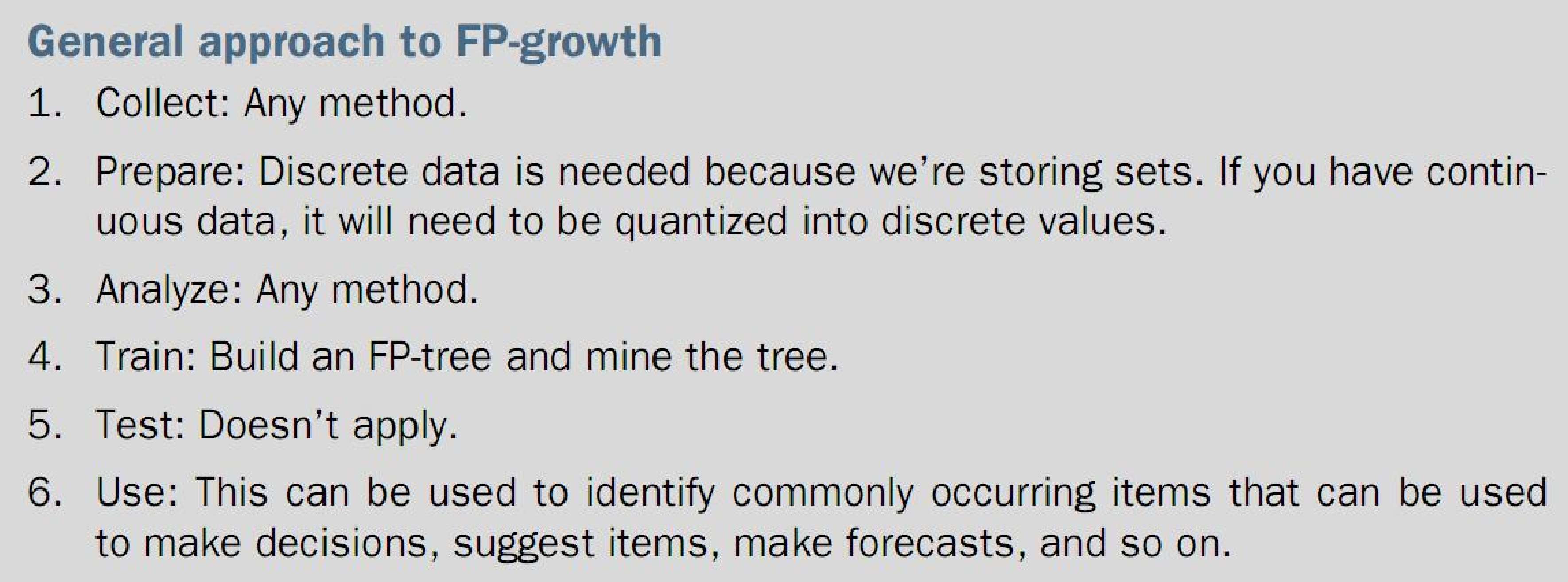
Figure 12.1 An example FP-tree. The FP-tree looks like a generic tree with links connecting similar items.



TID	Items in transaction
001	r, z, h, j, p
002	z, y, x, w, v, u, t, s
003	z
004	r, x, n, o, s
005	y, r, x, z, q, t, p
006	y, z, x, e, q, s, t, m

Table 12.1 Sample transaction dataset, used to generate the FP-tree in figure 12.1

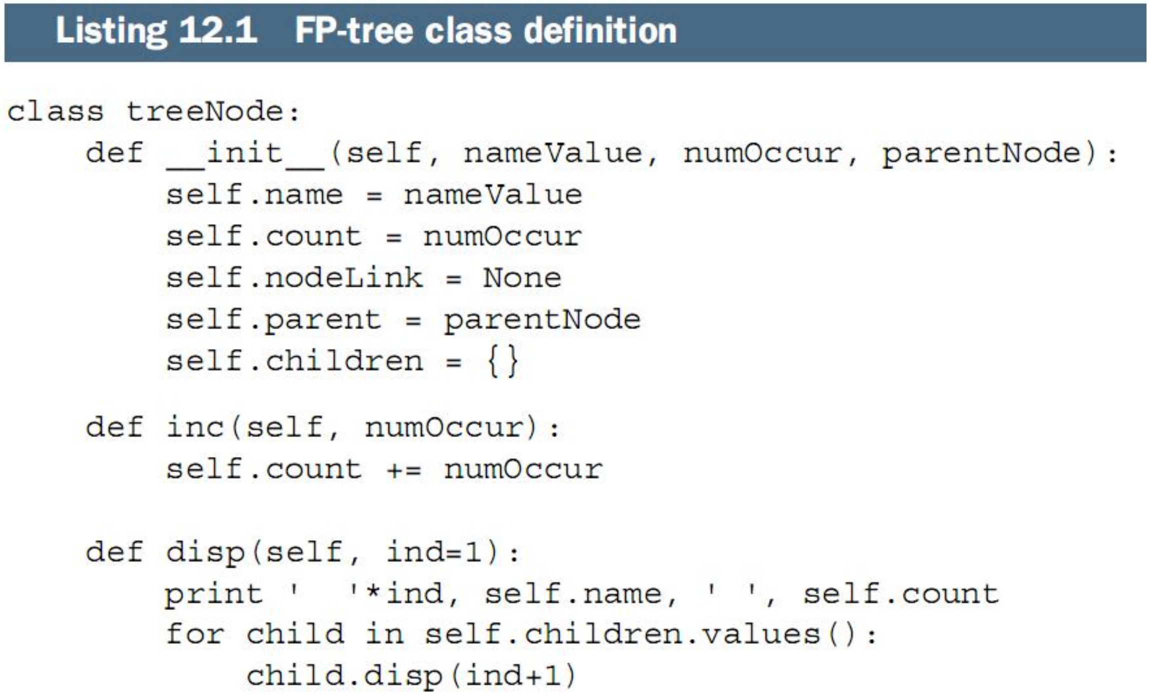
General Approach



General approach to FP-growth

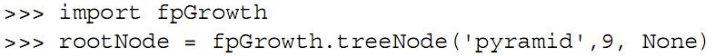
1. Collect: Any method.
2. Prepare: Discrete data is needed because we're storing sets. If you have continuous data, it will need to be quantized into discrete values.
3. Analyze: Any method.
4. Train: Build an FP-tree and mine the tree.
5. Test: Doesn't apply.
6. Use: This can be used to identify commonly occurring items that can be used to make decisions, suggest items, make forecasts, and so on.

FP-tree Node Structure

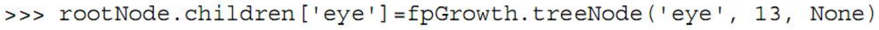


Listing 12.1 FP-tree class definition

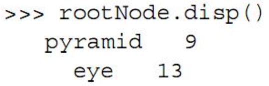
```
class treeNode:
    def __init__(self, nameValue, numOccur, parentNode):
        self.name = nameValue
        self.count = numOccur
        self.nodeLink = None
        self.parent = parentNode
        self.children = {}

    def inc(self, numOccur):
        self.count += numOccur

    def disp(self, ind=1):
        print '  '*ind, self.name, ' ', self.count
        for child in self.children.values():
            child.disp(ind+1)
```



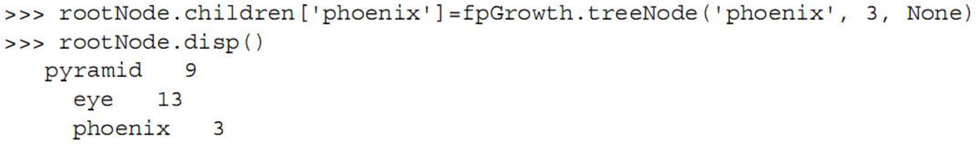
```
>>> import fpGrowth
>>> rootNode = fpGrowth.treeNode('pyramid',9, None)
```



```
>>> rootNode.children['eye']=fpGrowth.treeNode('eye', 13, None)
```



```
>>> rootNode.disp()
   pyramid   9
     eye   13
```



```
>>> rootNode.children['phoenix']=fpGrowth.treeNode('phoenix', 3, None)
>>> rootNode.disp()
   pyramid   9
     eye   13
     phoenix   3
```

FP-tree Construction

- FP-tree with header table
  - Header table: point to find similar items

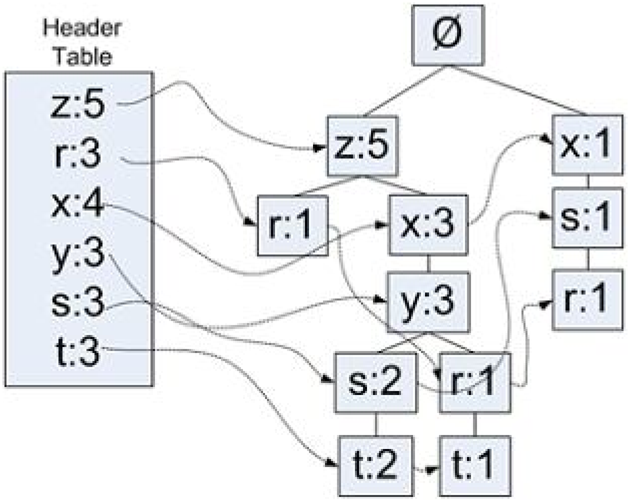
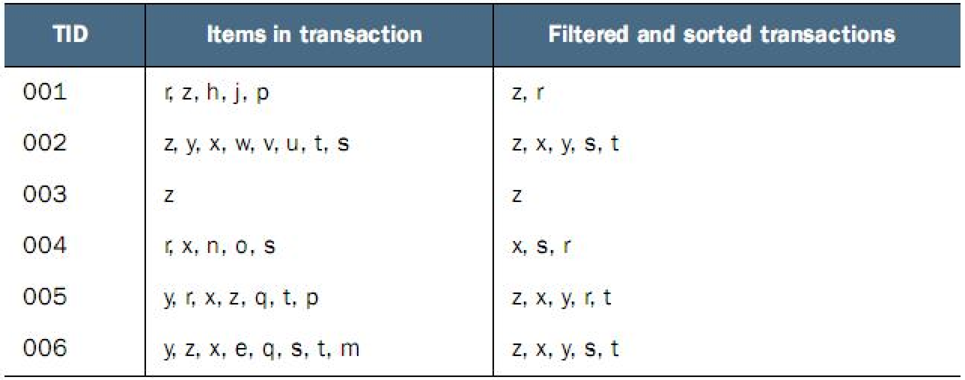


Figure 12.2 FP-tree with header table. The header table serves as a starting point to find similar items.

First Pass: Count Frequency

- Count occurrence for each item
- Eliminate items under minimum support
- Build FP-tree

Table 12.2 Transaction dataset with infrequent items removed and items reordered



TID	Items in transaction	Filtered and sorted transactions
001	r, z, h, j, p	z, r
002	z, y, x, w, v, u, t, s	z, x, y, s, t
003	z	z
004	r, x, n, o, s	x, s, r
005	y, r, x, z, q, t, p	z, x, y, r, t
006	y, z, x, e, q, s, t, m	z, x, y, s, t

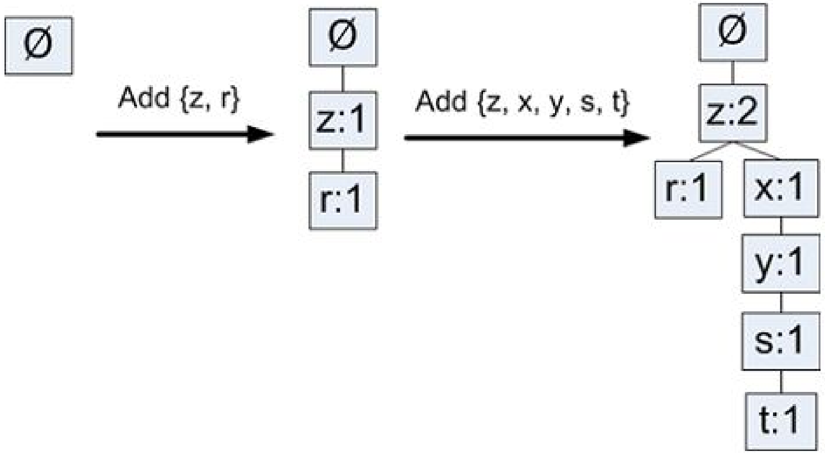
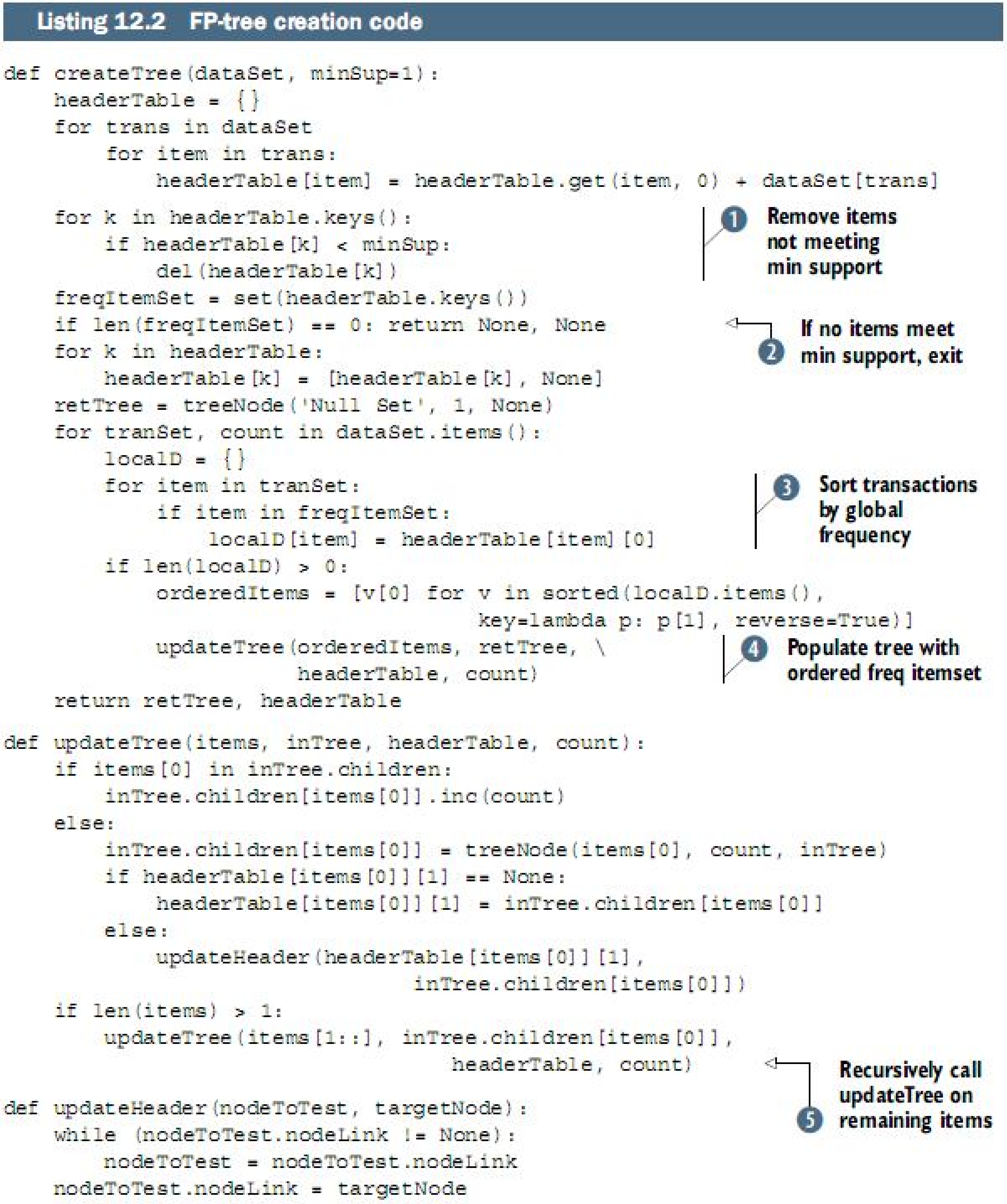


Figure 12.3 An illustration of the FP-tree creation process, showing the first two steps in creating the FP-tree using the data in table 12.2

Construction Code



Listing 12.2 FP-tree creation code

```
def createTree(dataSet, minSup=1):
    headerTable = {}
    for trans in dataSet
        for item in trans:
            headerTable[item] = headerTable.get(item, 0) + dataSet[trans]
    for k in headerTable.keys():
        if headerTable[k] < minSup:
            del(headerTable[k])
    freqItemSet = set(headerTable.keys())
    if len(freqItemSet) == 0: return None, None
    for k in headerTable:
        headerTable[k] = [headerTable[k], None]
    retTree = treeNode('Null Set', 1, None)
    for tranSet, count in dataSet.items():
        localD = {}
        for item in tranSet:
            if item in freqItemSet:
                localD[item] = headerTable[item][0]
        if len(localD) > 0:
            orderedItems = [v[0] for v in sorted(localD.items(),
                                      key=lambda p: p[1], reverse=True)]
            updateTree(orderedItems, retTree, \
                           headerTable, count)
    return retTree, headerTable

def updateTree(items, inTree, headerTable, count):
    if items[0] in inTree.children:
        inTree.children[items[0]].inc(count)
    else:
        inTree.children[items[0]] = treeNode(items[0], count, inTree)
        if headerTable[items[0]][1] == None:
            headerTable[items[0]][1] = inTree.children[items[0]]
        else:
            updateHeader(headerTable[items[0]][1],
                                  inTree.children[items[0]])
    if len(items) > 1:
        updateTree(items[1::], inTree.children[items[0]],
                                  headerTable, count)

def updateHeader(nodeToTest, targetNode):
    while (nodeToTest.nodeLink != None):
        nodeToTest = nodeToTest.nodeLink
    nodeToTest.nodeLink = targetNode
```

1 Remove items not meeting min support

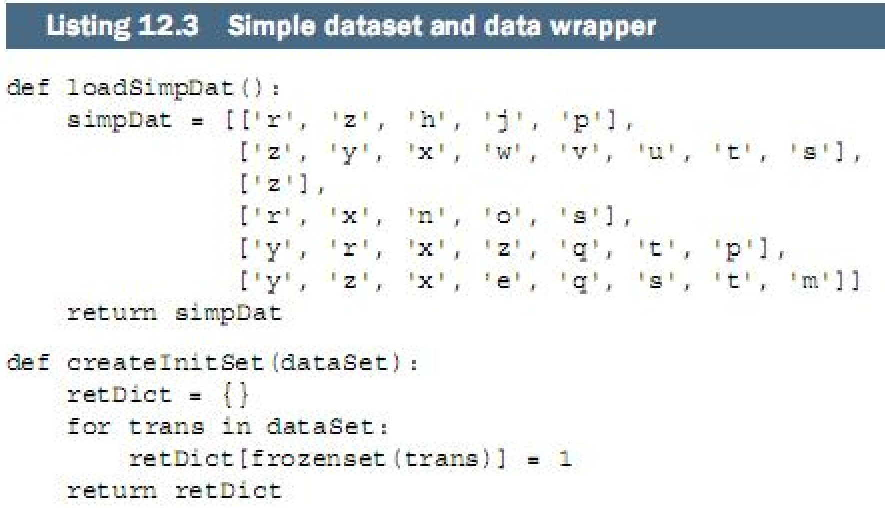
2 If no items meet min support, exit

3 Sort transactions by global frequency

4 Populate tree with ordered freq itemset

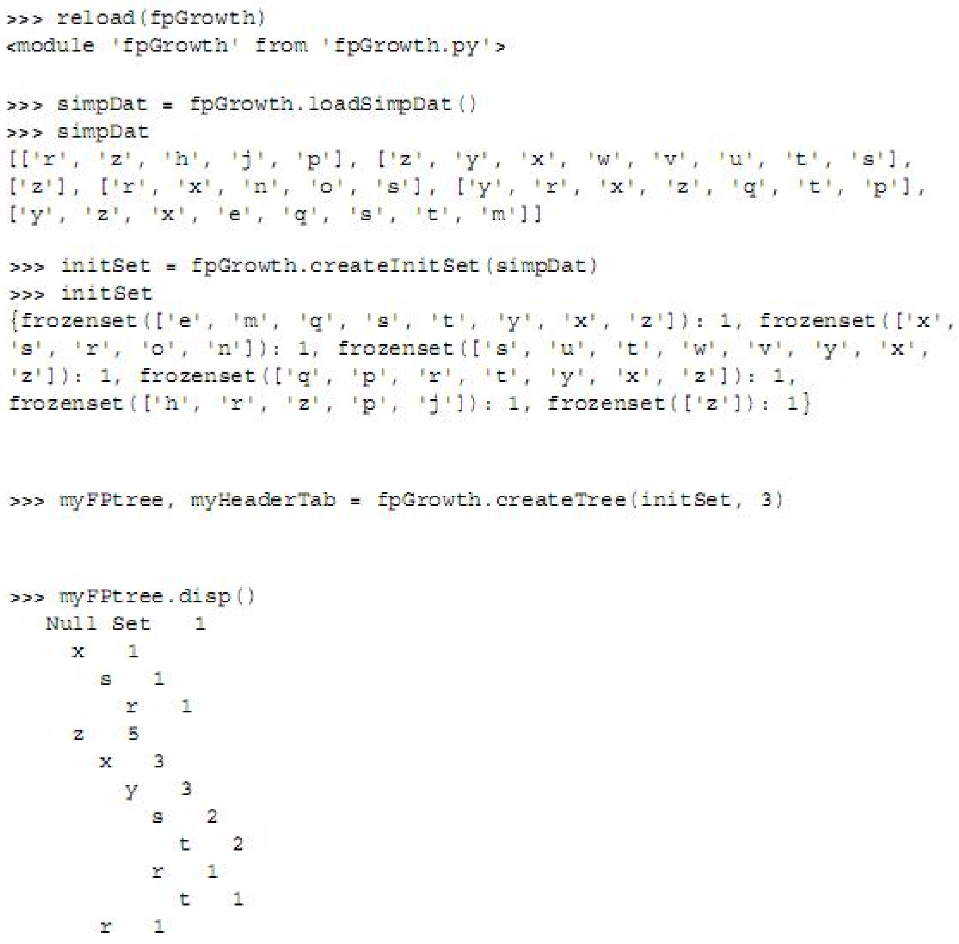
5 Recursively call updateTree on remaining items

Simple Data



Listing 12.3 Simple dataset and data wrapper

```
def loadSimpDat():
    simpDat = [['r', 'z', 'h', 'j', 'p'],
               ['z', 'y', 'x', 'w', 'v', 'u', 't', 's'],
               ['z'],
               ['r', 'x', 'n', 'o', 's'],
               ['y', 'r', 'x', 'z', 'q', 't', 'p'],
               ['y', 'z', 'x', 'e', 'q', 's', 't', 'm']]
    return simpDat

def createInitSet(dataSet):
    retDict = {}
    for trans in dataSet:
        retDict[frozenset(trans)] = 1
    return retDict
```



```
>>> reload(fpGrowth)
<module 'fpGrowth' from 'fpGrowth.py'>

>>> simpDat = fpGrowth.loadSimpDat()
>>> simpDat
[['r', 'z', 'h', 'j', 'p'], ['z', 'y', 'x', 'w', 'v', 'u', 't', 's'],
['z'], ['r', 'x', 'n', 'o', 's'], ['y', 'r', 'x', 'z', 'q', 't', 'p'],
['y', 'z', 'x', 'e', 'q', 's', 't', 'm']]

>>> initSet = fpGrowth.createInitSet(simpDat)
>>> initSet
{frozenset(['e', 'm', 'q', 's', 't', 'y', 'x', 'z']): 1, frozenset(['x',
's', 'r', 'o', 'n']): 1, frozenset(['s', 'u', 't', 'w', 'v', 'y', 'x',
'z']): 1, frozenset(['q', 'p', 'r', 't', 'y', 'x', 'z']): 1,
frozenset(['h', 'r', 'z', 'p', 'j']): 1, frozenset(['z']): 1}
```

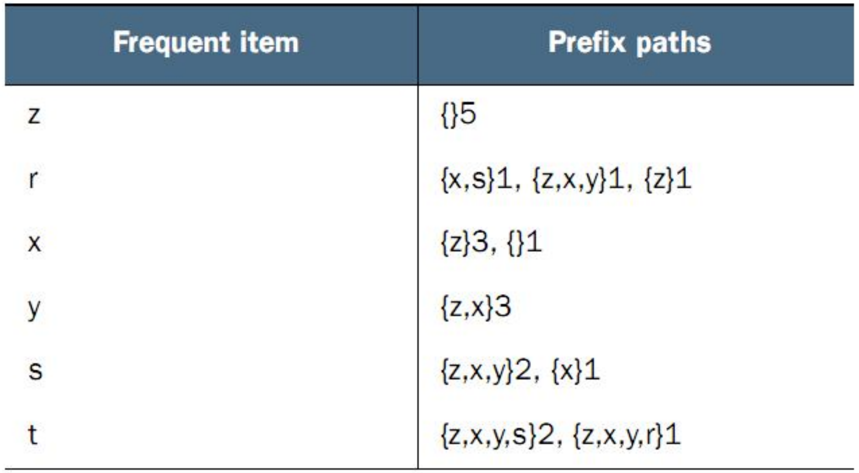
```
>>> myFPtree, myHeaderTab = fpGrowth.createTree(initSet, 3)
```

```
>>> myFPtree.disp()
   Null Set    1
     x   1
       s   1
         r   1
     z   5
       x   3
         y   3
           s   2
             t   2
           r   1
             t   1
       r   1
```

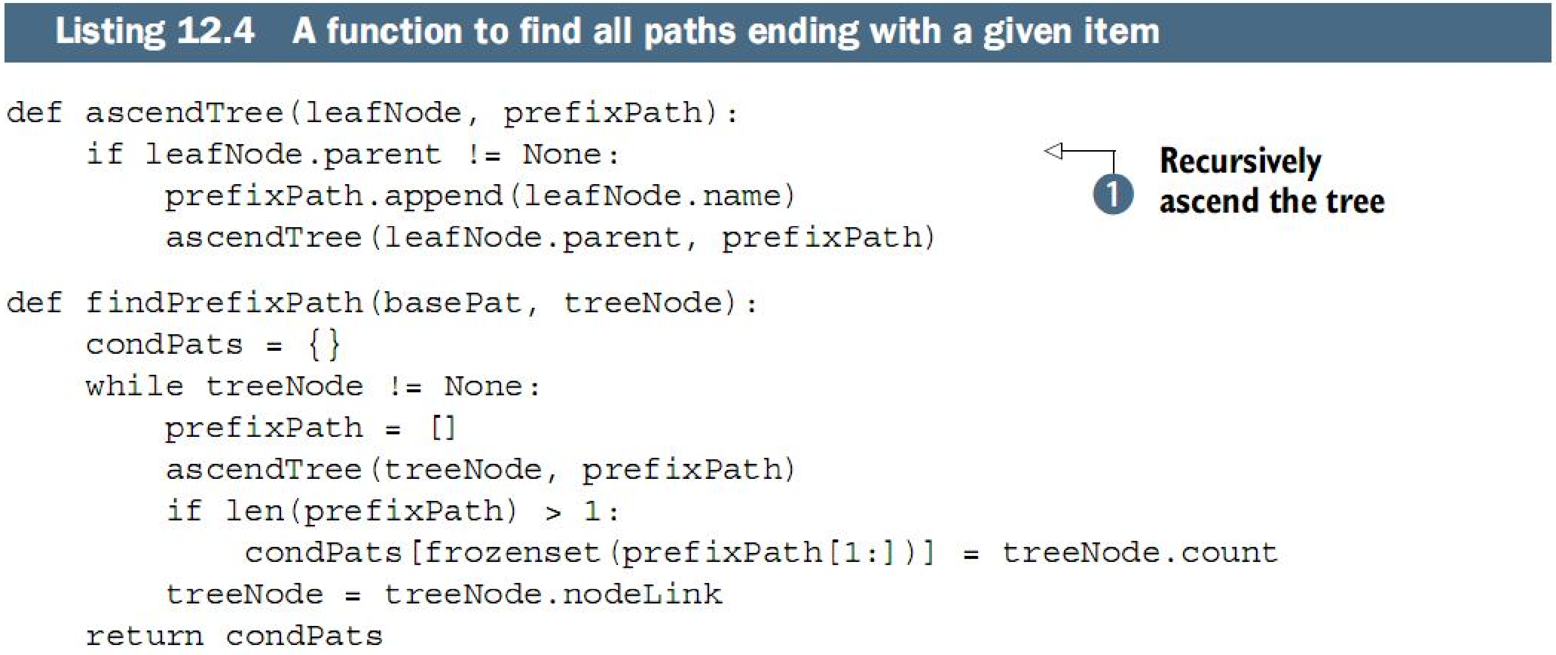
Mining Frequent Items

- Extract conditional pattern bases from the FP-tree
- From the conditional pattern base, construct a conditional FP-tree
- Recursively repeat steps 1 and 2 on until the tree contains a single item

Extract Conditional Pattern Bases



Frequent item	Prefix paths
z	{}5
r	{x,s}1, {z,x,y}1, {z}1
x	{z}3, {}1
y	{z,x}3
s	{z,x,y}2, {x}1
t	{z,x,y,s}2, {z,x,y,r}1



Listing 12.4 A function to find all paths ending with a given item

```
def ascendTree(leafNode, prefixPath):
    if leafNode.parent != None:
        prefixPath.append(leafNode.name)
        ascendTree(leafNode.parent, prefixPath)

def findPrefixPath(basePat, treeNode):
    condPats = {}
    while treeNode != None:
        prefixPath = []
        ascendTree(treeNode, prefixPath)
        if len(prefixPath) > 1:
            condPats[frozenset(prefixPath[1:])] = treeNode.count
        treeNode = treeNode.nodeLink
    return condPats
```

1 Recursively ascend the tree

Creating Conditional FP-tree

Conditional FP-tree for item: t

Conditional pattern bases: {y, x, s, z}:2 , {y, x, r, z}: 1
Min support = 3
Remove: s & r

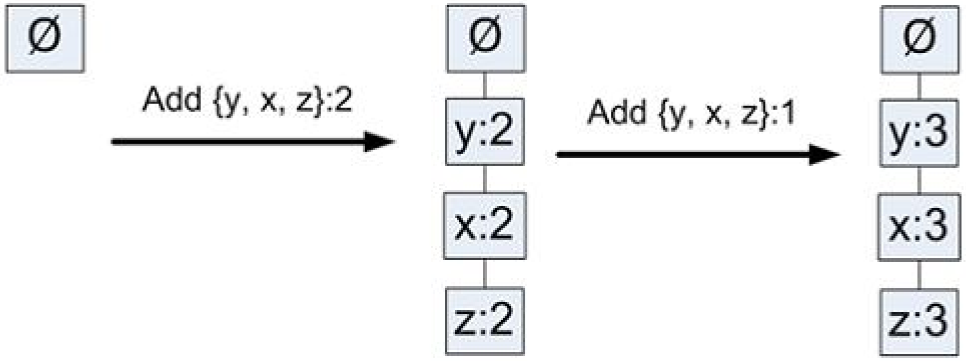
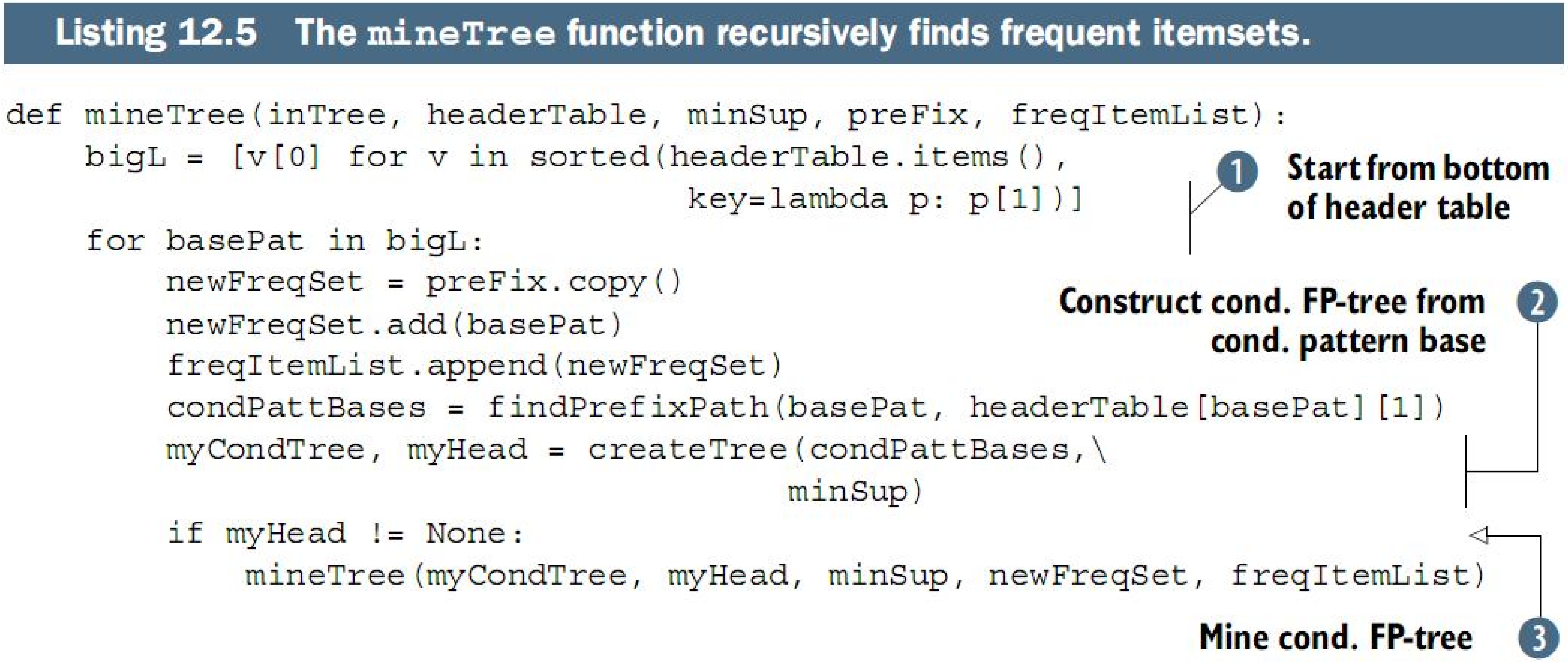


Figure 12.4 The creation of the conditional FP-tree for item t. Initially the tree starts out as only the null set as the root. Next, the set {y,x,z} is added from the original set {y,x,s,z}; the character s didn't make it because it didn't meet the minimum support. Similarly, {y,x,z} is added from the original set {y,x,r,z}.

Finding Frequent Itemsets



Listing 12.5 The mineTree function recursively finds frequent itemsets.

```
def mineTree(inTree, headerTable, minSup, preFix, freqItemList):
    bigL = [v[0] for v in sorted(headerTable.items(),
                                  key=lambda p: p[1])]
    for basePat in bigL:
        newFreqSet = preFix.copy()
        newFreqSet.add(basePat)
        freqItemList.append(newFreqSet)
        condPattBases = findPrefixPath(basePat, headerTable[basePat][1])
        myCondTree, myHead = createTree(condPattBases,\
                                       minSup)
        if myHead != None:
            mineTree(myCondTree, myHead, minSup, newFreqSet, freqItemList)
```

1 Start from bottom of header table

2 Construct cond. FP-tree from cond. pattern base

3 Mine cond. FP-tree

Examples

- Finding co-occurring words in a Twitter feed
- Mining a clickstream from a news site
- Help yourself

Summary

- FP-growth: an efficient way of finding frequent patterns
  - Works with the Apriori principle, but faster
  - Scan dataset only twice
- The dataset is stored in a structure called an FP-tree