In [557]:

```
import numpy as np
import pandas as pd
from sklearn.tree import DecisionTreeClassifier
from sklearn.tree import export_graphviz
from sklearn.grid_search import GridSearchCV
from sklearn.grid_search import RandomizedSearchCV
from sklearn import tree
import pydotplus
from sklearn.cross_validation import  cross_val_score
from IPython.display import Image
from sklearn.externals.six import StringIO
from sklearn.utils import shuffle
from sklearn.metrics import precision_recall_fscore_support
from sklearn.metrics import mean_squared_error
import statistics as st
```

In [558]:

```
data = pd.read_csv("wpbc.data.csv")
```

Question 1

1a)

In [559]:

```
X = data.loc[:,:]
y = data.loc[:,'Outcome']
```

In [560]:

```
X = X.drop(X.columns[[1, 2]], 1)
```

In [561]:

```
dt_old = DecisionTreeClassifier(criterion = "entropy", random_state = 100, min_samples_
leaf=25)
dt_old.fit(X, y)
scores = cross_val_score(dt_old, X, y, cv=4)
print("Mean Accuracy using four fold: {:.3f} (std: {:.3f})".format(scores.mean(),scores
.std()),)
```

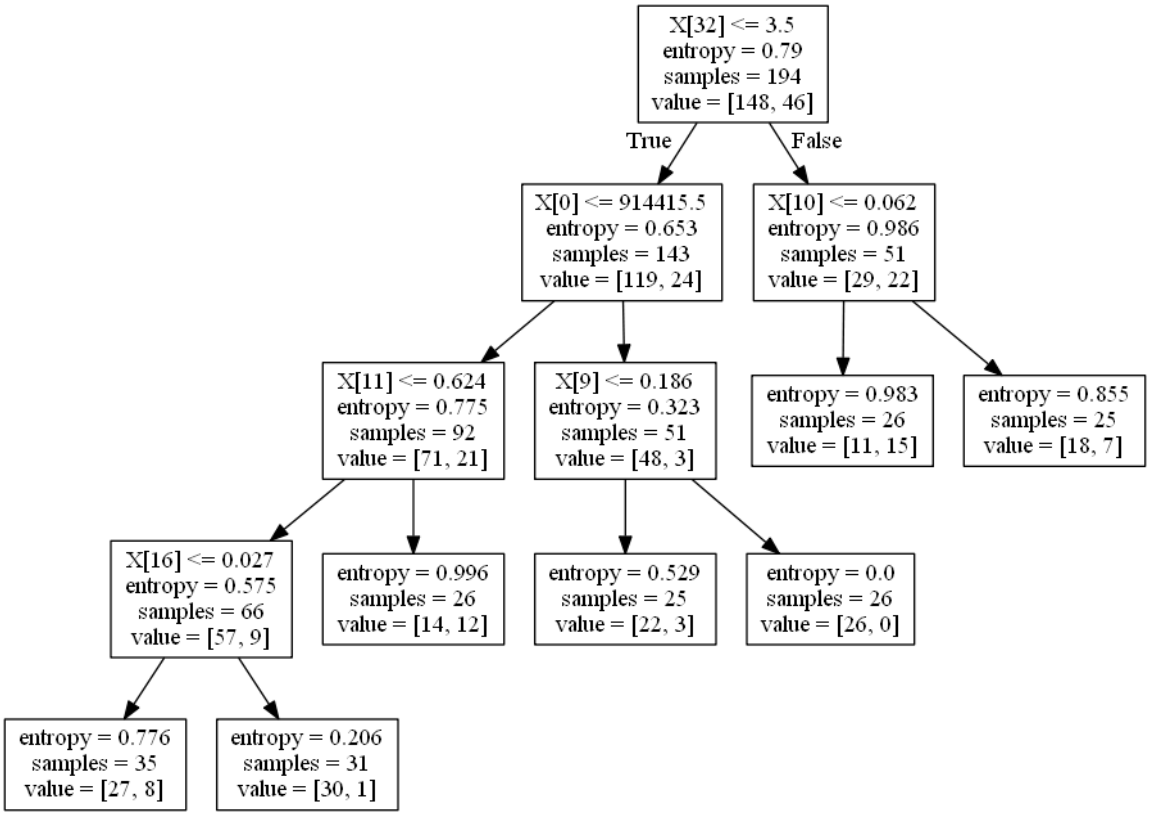
```
Mean Accuracy using four fold: 0.763 (std: 0.008)
```

For the decision tree generated above, parameters used are : criterion = "entropy", random_state = 100 and min_samples_leaf=25

In [562]:

```
dot_data = StringIO()
tree.export_graphviz(dt_old,out_file=dot_data)
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
Image(graph.create_png())
```

Out[562]:



In [563]:

```
pred = dt_old.predict(X)
result = precision_recall_fscore_support(y, pred, pos_label='N', average='binary')
print("For Non-Recurrence class  Precision " +str(result[0])
                         +"\n\t\t\t Recall "  +str(result[1]) + "\t F1 Score " + str(r
esult[2]))
result = precision_recall_fscore_support(y, pred, pos_label='R', average='binary')
print("For Recurrence class  Precision " +str(result[0])
                         +"\n\t\t\t Recall "  +str(result[1]) + "\t F1 Score " + str(r
esult[2]))
```

```
For Non-Recurrence class  Precision 0.8154761904761905
                         Recall 0.9256756756756757       F1 Score 0.867088
6075949368
For Recurrence class  Precision 0.5769230769230769
                         Recall 0.32608695652173914      F1 Score 0.416666
66666666663
```

1b) As Seen from above output the precision and recall value is high for Non-reccurrence class than Recurrence class. This is because in the given dataset there are more number of Non-recurrence class records than the Recurrence class records. Hence, the decision tree is more biased towards predicting Non-recurrence class than Recurrence class

Question 2

Split dataset into train and test

In [564]:

```
data = shuffle(data)

X = data.loc[:,:]
y = data.loc[:,'Time ']
X = X.drop(X.columns[[1]], 1)
X_train,X_test,y_train,y_test = X[:129] , X[129:], y[:129] , y[129:]
list(X_test.keys())
X.columns[[1, 2]]
data = data.drop(data.columns[[1]], 1)
```

2a) Assuming the average number of disease-free years for the entire training dataset as the initial predicted value for each record of the dataset.

Finding the mean square error (MSE) of making this prediction for all the training and the test set records (separately for training and test sets).

In [565]:

```
y_pred = y_train.mean()
y_predict = [y_pred] * len(y_train)
train_MSE = mean_squared_error(y_train, y_predict)
y_predict = [y_pred] * len(y_test)
test_MSE  = mean_squared_error(y_test, y_predict)
print("TRAIN MSE = "+str(train_MSE)+"\tTEST MSE = "+str(test_MSE))
leaf_population = []
pred = []
groupse = []
```

```
TRAIN MSE = 1211.5191394747912  TEST MSE = 1137.2880161602714
```

2b)

Below are the Functions to generate leaf node , Split dataset, calculate MSE, Generate tree recursively

In [566]:

```
def to_terminal(group):
    outcomes =  group.loc[:,'Time '].mean()
    leaf_population.append(len(group))
    pred.append(outcomes)
    groupse.append(group)
    return outcomes
```

In [567]:

```
def df_split(df, feature, value):
    left = df[df[feature]<=value]
    right = df[df[feature]>value]
    return left, right
```

In [568]:

```
def get_MSE(dataset):
    y_pred = dataset.loc[:,'Time '].mean()
    y_predict = [y_pred] * len(dataset.loc[:,'Time '])
    train_MSE = mean_squared_error(dataset.loc[:,'Time '], y_predict)
    return train_MSE
```

In [569]:

```
def get_split(dataset):
    b_index, b_value, b_score, b_groups = 999, 999, 999, None
    feature = best_feature_to_split(dataset.drop(dataset.columns[[1]], 1), dataset.loc
[:,'Time '])
    b_value = best_value_to_split(dataset, feature)
    b_groups = df_split(dataset, feature, b_value)
    return {'index':feature, 'value':b_value, 'groups':b_groups}
```

Taking median value of this attribute as the split-point

In [570]:

```
def best_value_to_split(df, feature):
    col_med = st.median(df.loc[:,feature])
    return col_med
```

Finding the attribute most closely correlated to the number of disease-free years in the training data

In [571]:

```
def best_feature_to_split(X_tr, y_train):
    corr = X_tr.apply(lambda x: x.corr(y_train))
    max_col = corr.idxmax()
    return max_col
```

Function to generate splits recursively and form the leaf node if MSE value is below 1000

In [572]:

```
def split(node):
    left, right = node['groups']
    del(node['groups'])
    # check for a no split
    if len(left) == 0 or len(right) == 0 :
        return
    # process left child
    left_MSE = get_MSE(left)
    right_MSE = get_MSE(right)
    if left_MSE <= 1100:
        node['left'] = to_terminal(left)
    else:
        node['left'] = get_split(left)
        split(node['left'])
    # process right child
    if right_MSE <= 1100:
        node['right'] = to_terminal(right)

    else:
        node['right'] = get_split(right)
        split(node['right'])
```

In [573]:

```
def build_tree(train):
    root = get_split(train)
    split(root)
    return root
```

2c) Below is the resulting tree and details about each leaf node including its population, predicted value, and MSE values for the training and the test datasets.

MSE value threshold is selected so that it should be less than the MSE value computed above with mean of all data as predicted value, and also the depth of the tree is minimum

In [574]:

```
import pprint
tree = build_tree(X_train)
print("Below is the generated regression tree\n")
pprint.pprint(tree)
```

```
Below is the generated regression tree

{'index': 'fractal dimension',
 'left': 36.95384615384615,
 'right': {'index': 'fractal dimension',
           'left': {'index': 'M_fractal dimension',
                    'left': 30.8125,
                    'right': 64.1875,
                    'value': 0.062305},
           'right': {'index': 'fractal dimension',
                     'left': {'index': 'M_symmetry ',
                              'left': {'index': 'SE_symmetry ',
                                       'left': 29.75,
                                       'right': {'index': 'SE_compactness
',
                                                 'left': 40.5,
                                                 'right': {'index': 'SE_te
xture ',
                                                           'left': 53.0,
                                                           'right': 125.0,
                                                           'value': 0.7955
5},
                                                 'value': 0.030865},
                                       'value': 0.014655000000000001},
                              'right': {'index': 'M_symmetry ',
                                        'left': 32.5,
                                        'right': {'index': 'Lymph node sta
tus',
                                                  'left': 26.0,
                                                  'right': 117.5,
                                                  'value': 0.5},
                                        'value': 0.2237},
                              'value': 0.20529999999999998},
                     'right': 75.9375,
                     'value': 0.11465},
           'value': 0.1028},
 'value': 0.08621000000000001}
```

In [578]:

```
print("Printing leaf Node population and predcited value in INORDER manner\n\n")
for i in range(len(pred)):
    print("For leaf node                        "+str(i))
    print("Predicted value of the leaf node    "+str(pred[i]))
    print("Population in the leaf node         "+str(leaf_population[i]))
    print("MSE for this leaf                    "+str(get_MSE(groupse[i]))+"\n")
```

```
Printing leaf Node population and predcited value in INORDER manner


For leaf node                       0
Predicted value of the leaf node    36.95384615384615
Population in the leaf node         65
MSE for this leaf                   923.9824852071006

For leaf node                       1
Predicted value of the leaf node    30.8125
Population in the leaf node         16
MSE for this leaf                   940.77734375

For leaf node                       2
Predicted value of the leaf node    64.1875
Population in the leaf node         16
MSE for this leaf                   914.27734375

For leaf node                       3
Predicted value of the leaf node    29.75
Population in the leaf node         4
MSE for this leaf                   764.1875

For leaf node                       4
Predicted value of the leaf node    40.5
Population in the leaf node         2
MSE for this leaf                   462.25

For leaf node                       5
Predicted value of the leaf node    53.0
Population in the leaf node         1
MSE for this leaf                   0.0

For leaf node                       6
Predicted value of the leaf node    125.0
Population in the leaf node         1
MSE for this leaf                   0.0

For leaf node                       7
Predicted value of the leaf node    32.5
Population in the leaf node         4
MSE for this leaf                   787.25

For leaf node                       8
Predicted value of the leaf node    26.0
Population in the leaf node         2
MSE for this leaf                   289.0

For leaf node                       9
Predicted value of the leaf node    117.5
Population in the leaf node         2
MSE for this leaf                   2.25

For leaf node                       10
Predicted value of the leaf node    75.9375
Population in the leaf node         16
MSE for this leaf                   689.43359375
```

In [576]:

```
def predict(node, row):
    col = node['index']
    if row[col] < node['value']:
        if isinstance(node['left'], dict):
            return predict(node['left'], row)
        else:
            return node['left']
    else:
        if isinstance(node['right'], dict):
            return predict(node['right'], row)
        else:
            return node['right']
```

Calculating MSE for TEST and TRAIN data using above generated regression tree

In [388]:

```
predictions_train = list()
test = X_train
for i in range(len(X_train)):
        prediction_train = predict(tree, X_train.iloc[i])
        predictions_train.append(prediction_train)


predictions_test = list()
test = X_test
for i in range(len(X_test)):
        prediction_test = predict(tree, X_test.iloc[i])
        predictions_test.append(prediction_test)

train_MSE  = mean_squared_error(y_train, predictions_train)
print("\tTrain MSE = "+str(train_MSE))

test_MSE  = mean_squared_error(y_test, predictions_test)
print("\tTEST MSE = "+str(test_MSE))
```

```
        Train MSE = 916.3741147481767
        TEST MSE = 1457.7749361060537
```

1d)

As seen from the result of the regression tree, train MSE is very low as compared to the test MSE, this means there is very high variance. Resulted tree is very dense at some nodes whereas very thin at some nodes.

The leaf path with split attribute "SE_compactness" is the best as the MSE value for this leaf is very low whereas leaf path with split attribute "fractal dimension" which is the left most leaf is worst path which has MSE 923.98, highest of all.

Prediction depends on fractal dimension, SE_texture, M_symmetry, SE_symmetry, SE_compactness

The main difference between decision tree and regression tree is, decision tree gives discrete valued output, whereas regression tree gives continous valued output.

Question3

3a) The regression tree below is grown such that MSE at leaf should be minimum and also, the number of leaf nodes should be relatively smaller. hence the tree growth is stopped for MSE value less than 0.708.

details of each leaf node including its population, predicted value and the MSE values for the training and the test datasets are printed in he output

In [604]:

```
data = pd.read_csv("winequality-white.csv")
list(data.keys())
```

Out[604]:

```
['fixed acidity',
 'volatile acidity',
 'citric acid',
 'residual sugar',
 'chlorides',
 'free sulfur dioxide',
 'total sulfur dioxide',
 'density',
 'pH',
 'sulphates',
 'alcohol',
 'quality']
```

In [605]:

```
data = shuffle(data)

X = data.loc[:,:]
y = data.loc[:,'quality']
X_train,X_test,y_train,y_test = X[:3266] , X[3266:], y[:3266] , y[3266:]
```

In [606]:

```
y_pred = y_train.mean()
y_predict = [y_pred] * len(y_train)
train_MSE = mean_squared_error(y_train, y_predict)
y_predict = [y_pred] * len(y_test)
test_MSE  = mean_squared_error(y_test, y_predict)
print("TRAIN MSE = "+str(train_MSE)+"\tTEST MSE = "+str(test_MSE))
leaf_population_wine = []
pred_wine = []
groups = []
```

```
TRAIN MSE = 0.7684309081411443  TEST MSE = 0.8159380507100019
```

In [607]:

```
def toW_terminal(group):
    outcomes =  group.loc[:,'quality'].mean()
    leaf_population_wine.append(len(group))
    pred_wine.append(outcomes)
    groups.append(group)
    return outcomes
```

In [608]:

```
def getW_MSE(dataset):
    y_pred = dataset.loc[:,'quality'].mean()
    y_predict = [y_pred] * len(dataset.loc[:,'quality'])
    train_MSE = mean_squared_error(dataset.loc[:,'quality'], y_predict)
    return train_MSE
```

In [609]:

```
def getW_split(dataset):
    b_index, b_value, b_score, b_groups = 999, 999, 999, None
    feature = best_feature_to_split(dataset.drop(dataset.columns[[11]], 1), dataset.loc
[:,'quality'])
    b_value = best_value_to_split(dataset, feature)
    b_groups = df_split(dataset, feature, b_value)
    return {'index':feature, 'value':b_value, 'groups':b_groups}
```

In [610]:

```
def splitW(node):
    left, right = node['groups']
    del(node['groups'])
    # check for a no split
    if len(left) == 0 or len(right) == 0 :
        return
    # process left child
    left_MSE = getW_MSE(left)
    right_MSE = getW_MSE(right)
    if left_MSE <= 0.708:
        node['left'] = toW_terminal(left)

    else:
        node['left'] = getW_split(left)
        splitW(node['left'])
    # process right child
    if right_MSE <= 0.708:
        node['right'] = toW_terminal(right)
    else:
        node['right'] = getW_split(right)
        splitW(node['right'])
```

In [611]:

```
def buildW_tree(train):
    root = getW_split(train)
    splitW(root)
    return root
```

In [612]:

```
tree = buildW_tree(X_train)
print("Below is the generated regression tree\n")
pprint.pprint(tree)
```

Below is the generated regression tree


```
{'index': 'alcohol',
 'left': 5.548238012709416,
 'right': {'index': 'alcohol',
           'left': {'index': 'alcohol',
                    'left': 5.867684478371501,
                    'right': {'index': 'free sulfur dioxide',
                              'left': {'index': 'free sulfur dioxide',
                                       'left': {'index': 'free sulfur diox
ide',
                                                'left': {'index': 'free su
lfur '
                                                                  'dioxid
e',
                                                         'left': {'index':
'free '

'sulfur '

'dioxide',
                                                                  'left':
{'index': 'residual '

          'sugar',

'left': 4.5,

'right': {'index': 'citric '

                    'acid',

           'left': {'index': 'alcohol',

                    'left': {'index': 'pH',

                             'left': 5.0,

                             'right': 7.0,

                             'value': 3.17},

                    'right': {'index': 'chlorides',

                              'left': 4.0,

                              'right': 8.0,

                              'value': 0.040499999999999994},

                    'value': 11.2},

           'right': 6.0,

           'value': 0.38},

'value': 1.4},
                                                                  'right':
5.545454545454546,
                                                                  'value':
10.0},
```

```
                                                                  'right': {'inde
x': 'alcohol',
                                                                            'left':
{'index': 'alcohol',

  'left': {'index': 'pH',

           'left': {'index': 'citric '

                                'acid',

                    'left': 5.0,

                    'right': 6.666666666666667,

                    'value': 0.37},

           'right': {'index': 'pH',

                     'left': 5.333333333333333,

                     'right': 7.333333333333333,

                     'value': 3.285},

           'value': 3.2},

  'right': 6.5,

  'value': 11.1},
                                                                            'righ
t': 6.222222222222222,
                                                                            'valu
e': 11.2},
                                                                   'value': 12.0},
                                                          'right': 6.0,
                                                          'value': 17.0},
                                                 'right': 6.139784946236559,
                                                 'value': 22.0},
                                        'right': 6.267379679144385,
                                        'value': 32.0},
                    'value': 10.9},
           'right': 6.4769433465085635,
           'value': 11.4},
 'value': 10.4}
```

In [613]:

```
print("Printing leaf Node population and predcited value in INORDER manner\n\n")
for i in range(len(pred_wine)):
    print("For leaf node                        "+str(i))
    print("Predicted value of the leaf node    "+str(pred_wine[i]))
    print("Population in the leaf node         "+str(leaf_population_wine[i]))
    print("MSE for this leaf                    "+str(getW_MSE(groups[i]))+"\n")
```

```
Printing leaf Node population and predcited value in INORDER manner


For leaf node                       0
Predicted value of the leaf node    5.548238012709416
Population in the leaf node         1731
MSE for this leaf                   0.5596314996757733

For leaf node                       1
Predicted value of the leaf node    5.867684478371501
Population in the leaf node         393
MSE for this leaf                   0.628803035306153

For leaf node                       2
Predicted value of the leaf node    4.5
Population in the leaf node         10
MSE for this leaf                   0.25

For leaf node                       3
Predicted value of the leaf node    5.0
Population in the leaf node         2
MSE for this leaf                   0.0

For leaf node                       4
Predicted value of the leaf node    7.0
Population in the leaf node         1
MSE for this leaf                   0.0

For leaf node                       5
Predicted value of the leaf node    4.0
Population in the leaf node         1
MSE for this leaf                   0.0

For leaf node                       6
Predicted value of the leaf node    8.0
Population in the leaf node         1
MSE for this leaf                   0.0

For leaf node                       7
Predicted value of the leaf node    6.0
Population in the leaf node         4
MSE for this leaf                   0.0

For leaf node                       8
Predicted value of the leaf node    5.545454545454546
Population in the leaf node         11
MSE for this leaf                   0.6115702479338844

For leaf node                       9
Predicted value of the leaf node    5.0
Population in the leaf node         4
MSE for this leaf                   0.0

For leaf node                       10
Predicted value of the leaf node    6.666666666666667
Population in the leaf node         3
MSE for this leaf                   0.22222222222222224

For leaf node                       11
Predicted value of the leaf node    5.333333333333333
Population in the leaf node         3
```

```
MSE for this leaf                   0.22222222222222224

For leaf node                       12
Predicted value of the leaf node    7.333333333333333
Population in the leaf node         3
MSE for this leaf                   0.22222222222222224

For leaf node                       13
Predicted value of the leaf node    6.5
Population in the leaf node         6
MSE for this leaf                   0.5833333333333334

For leaf node                       14
Predicted value of the leaf node    6.222222222222222
Population in the leaf node         9
MSE for this leaf                   0.3950617283950617

For leaf node                       15
Predicted value of the leaf node    6.0
Population in the leaf node         45
MSE for this leaf                   0.5777777777777777

For leaf node                       16
Predicted value of the leaf node    6.139784946236559
Population in the leaf node         93
MSE for this leaf                   0.7008902763325235

For leaf node                       17
Predicted value of the leaf node    6.267379679144385
Population in the leaf node         187
MSE for this leaf                   0.6664760216191485

For leaf node                       18
Predicted value of the leaf node    6.4769433465085635
Population in the leaf node         759
MSE for this leaf                   0.6816159533121212
```

In [616]:

```
def predictW(node, row):
    col = node['index']
    if row[col] < node['value']:
        if isinstance(node['left'], dict):
            return predictW(node['left'], row)
        else:
            return node['left']
    else:
        if isinstance(node['right'], dict):
            return predictW(node['right'], row)
        else:
            return node['right']
```

In [617]:

```
predictions_train = list()
test = X_train
for i in range(len(X_train)):
        prediction_train = predictW(tree, X_train.iloc[i])
        predictions_train.append(prediction_train)


predictions_test = list()
test = X_test
for i in range(len(X_test)):
        prediction_test = predictW(tree, X_test.iloc[i])
        predictions_test.append(prediction_test)

train_MSE  = mean_squared_error(y_train, predictions_train)
print("For White wine dataset\n")
print("\tTrain MSE = "+str(train_MSE))

test_MSE  = mean_squared_error(y_test, predictions_test)
print("\tTEST MSE = "+str(test_MSE))
```

```
For White wine dataset

        Train MSE = 0.6318652731332102
        TEST MSE = 0.6734914560748957
```

3b)

Best path for this tree is path reching the leaf with predicted value 5.0 and has split attribute "citric acid" as it is having 0.0 MSE value, Whereas worst path for this tree is path reaching leaf with predicted value 6.13 and has split attribute "free sulfur dioxide" as it is having 0.7008 MSE value.

Question 4

In [618]:

```
dataR = pd.read_csv("winequality-red.csv")
y_test = dataR.loc[:,'quality']
list(dataR.keys())
```

Out[618]:

```
['fixed acidity',
 'volatile acidity',
 'citric acid',
 'residual sugar',
 'chlorides',
 'free sulfur dioxide',
 'total sulfur dioxide',
 'density',
 'pH',
 'sulphates',
 'alcohol',
 'quality']
```

In [620]:

```
predictions = list()
test = dataR
for i in range(len(dataR)):
        prediction_test = predict(tree, dataR.iloc[i])
        predictions.append(prediction_test)

MSE  = mean_squared_error(y_test, predictions)
print("For Red wine dataset")
print("MSE = "+str(MSE))
```

```
For Red wine dataset
MSE = 0.6524361467105195
```

From the above values it can be concluded that tree model works better on white wines because tree is trained using the values of white wine instances.

As the calculated MSE value for red wine dataset is same w.r.t white wine dataset. Model designed with white wine datset can be used to predict the ouput for red wine dataset

Q5

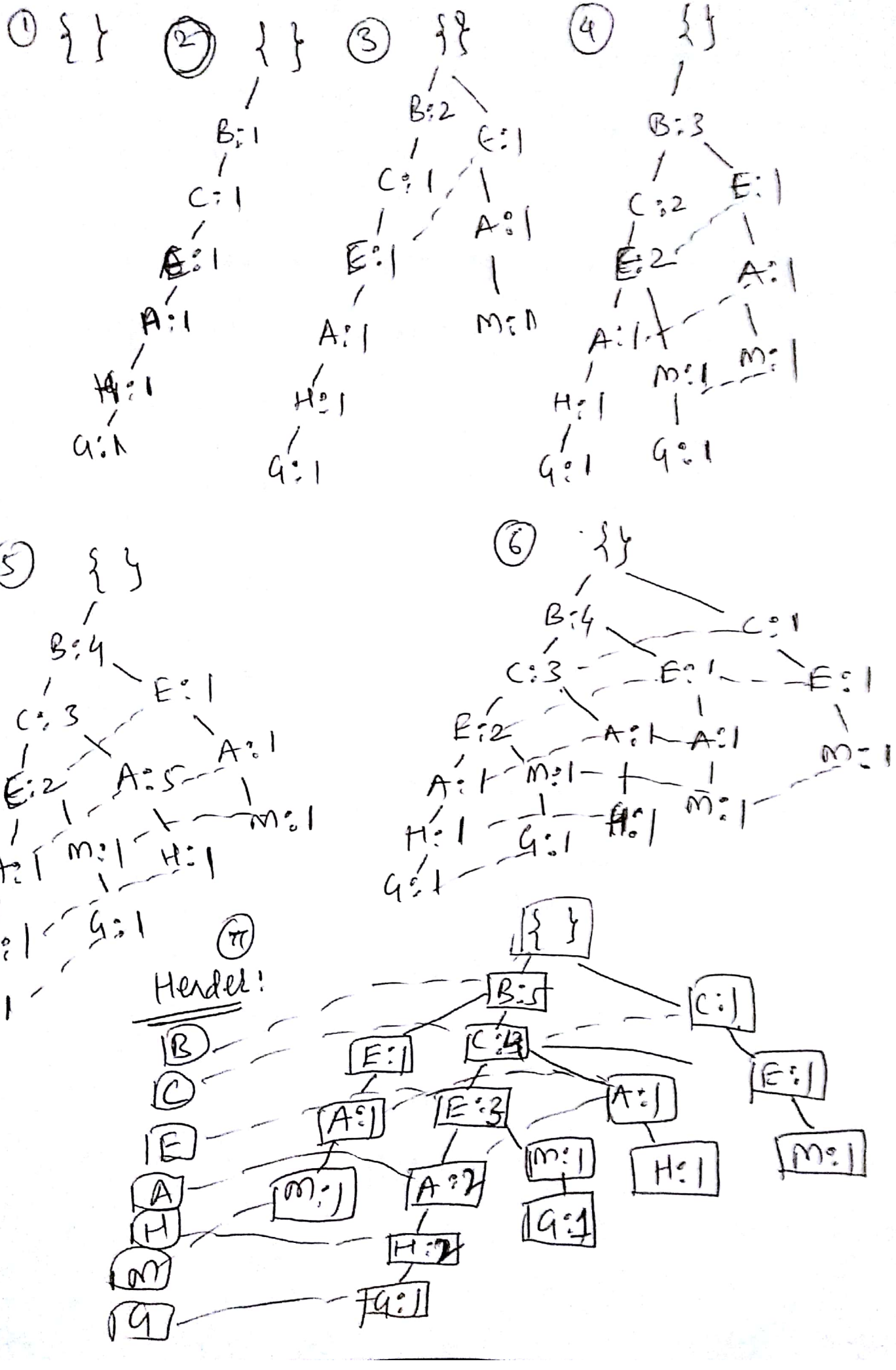
T1: A B C E G H
T2: A B E F M
T3: B C D E G M
T4: A B C H
T5: C D E F M
T6: A B C E H
T7: B C E G H M

(a) FP- Growth tree for these transactions.

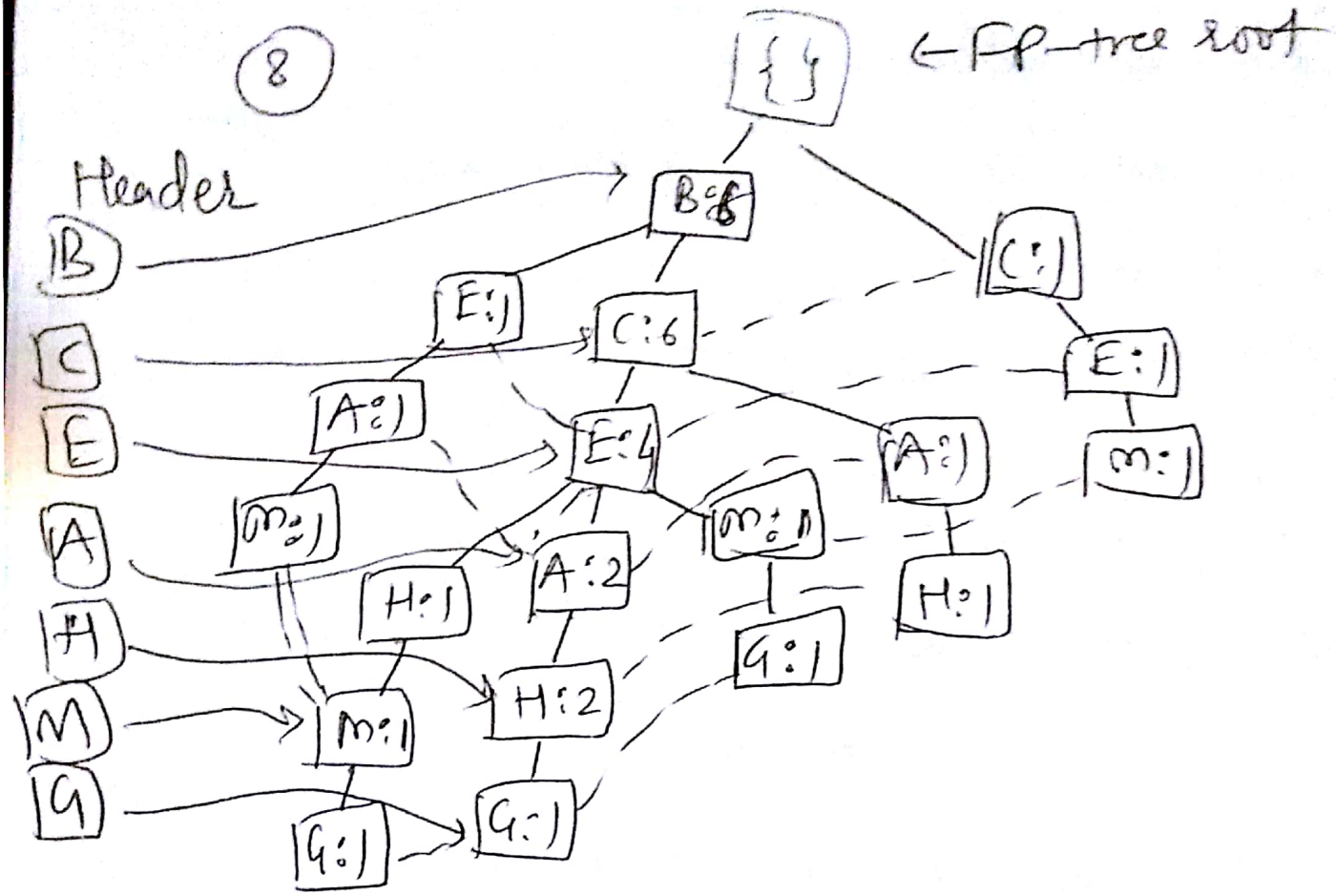
Frequency	Items with min support > 3
A - 4	B - 6
B - 6	C - 6
C - 6	E - 6
D - 2	A - 4
E - 6	H - 4
F - 2	M - 4
G - 3	G - 3
H - 4	
M - 4	

Transformed Transaction:

T1: B C E A H G
T2: B E A B
T3: B C E M G
T4: B C A H
T5: C E M
T6: B C E A H
T7: B C E H M G



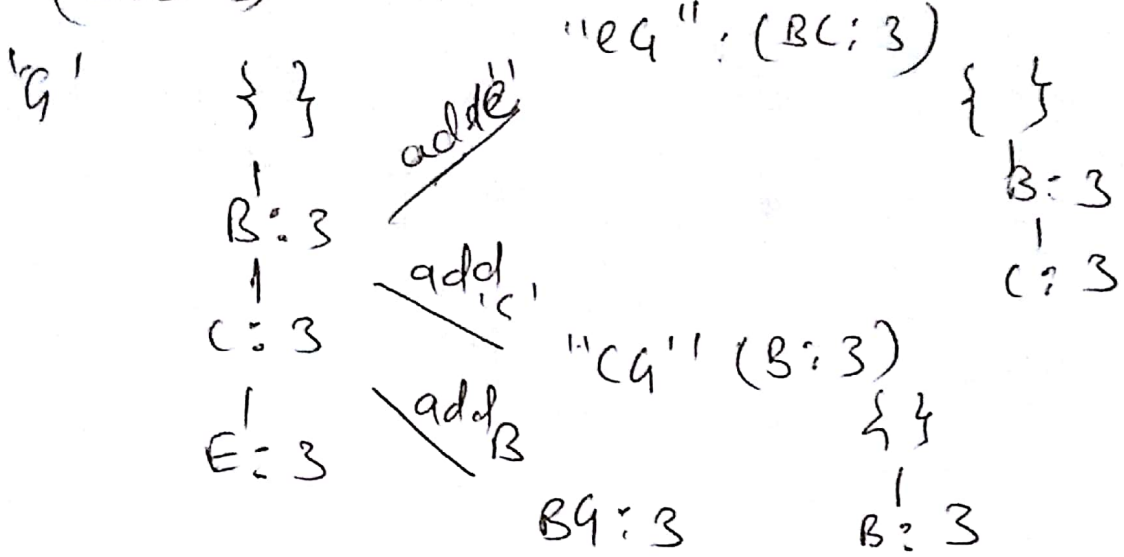
(8)



5b) Frequent Patterns

Let's take conditional pattern Base:

(BCE:3)



The first four frequent itemsets.

G, EG, CG, BG.

Steps :-

① Scan the database for 1st time to generate frequent itemlist. Scan the database for 2nd time and order frequent itemset in each transactn.

② Then Grow the FP-Tree by adding each tran at a time to the tree

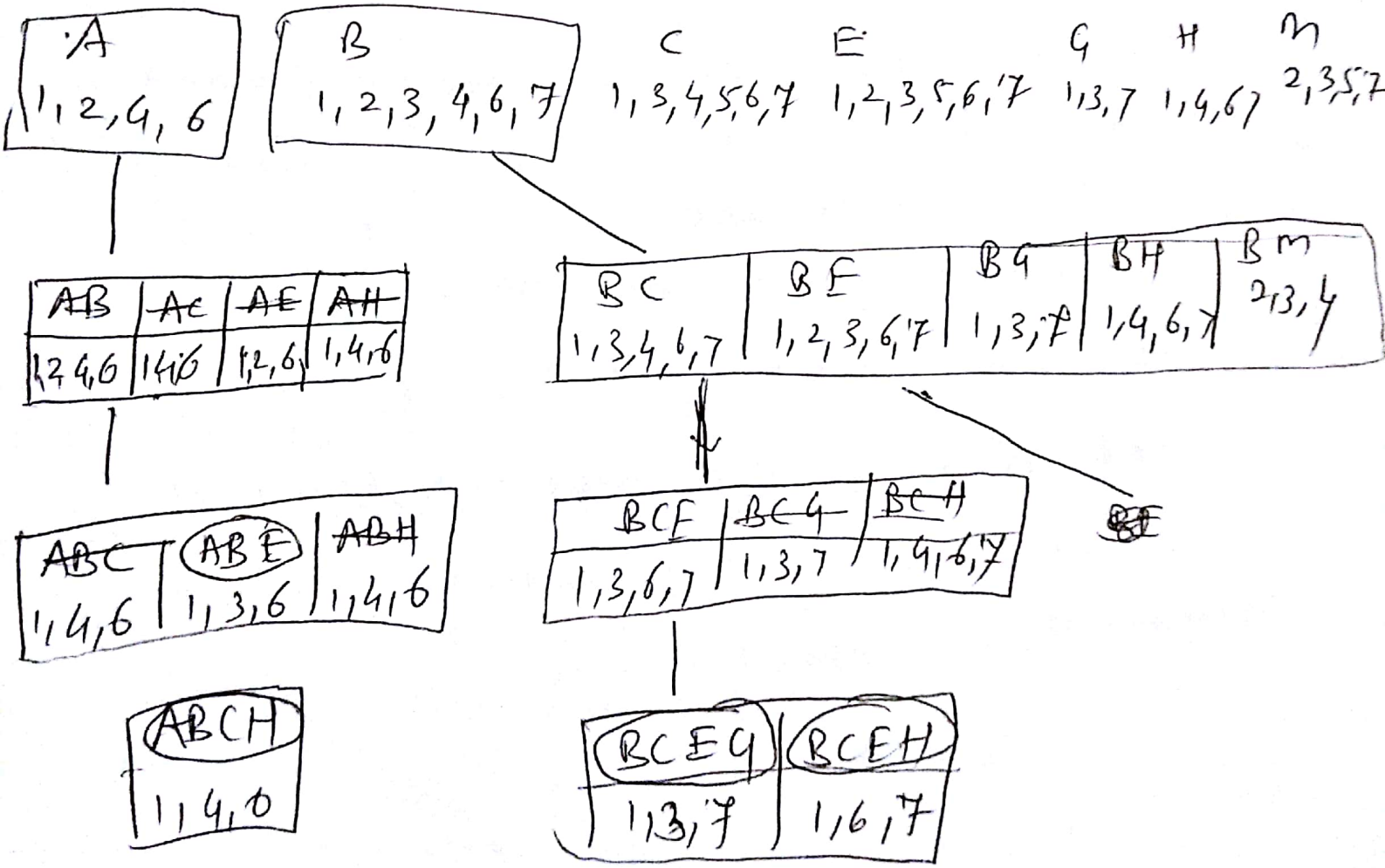
③ To find the frequent itemsets, start from bottom of frequent item header table in the FP tree.

④ Go over the FP-tree by following the links of each frequent item.

(5) For each pattern base accumulate the count for each item in the base and construct the conditional FP-tree for the frequent itemset of the pattern base.

(6) Finally, Recursively mine the conditional FP-tree until all the frequent itemsets are found.

5 c] GenMax Algorithm with this dataset and finding first three maximal frequent Itemsets.



Maximal frequent itemsets are:

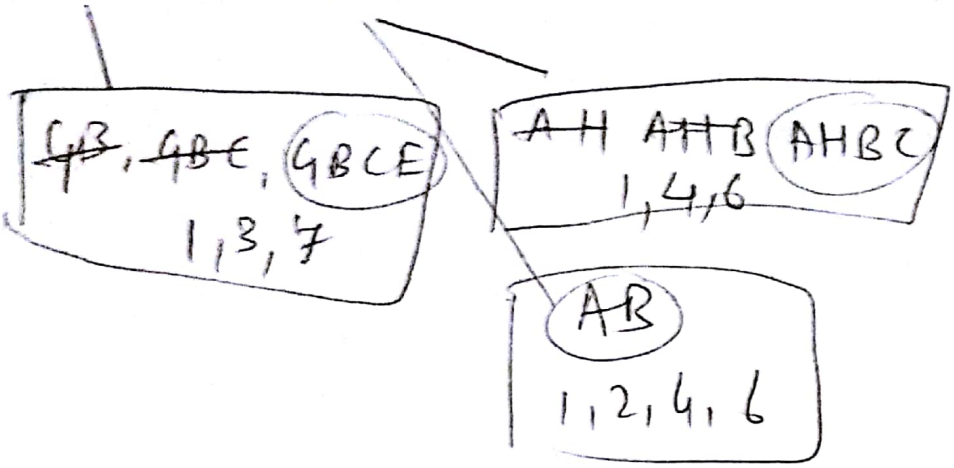
ABCH, ABE, BCEG.

Steps Description:

1. Transform the transaction items collection to item transaction collection.
2. Remove which are less than minimum support
3. Then prune by element wise.
4. Prune A gives. AB, AE, AE, AH.
   Again prune AB, giving ABC, ABE, ABH.
   Further pruning leaves ABC as ABCH - {1, 4} as maximal itemset with minm support
5. After traceback, ABE can't be pruned
   Hence, ABE is second maximal itemset.
6. As, ABH and AE are subset of maximal itemset, they shouldn't be pruned.
7. Similarly, Pruning B gives BC, BE, BG, BH, BM and Pruning BC gives BCE, BCG, BCH.
   Finally, Pruning BCE will give third maximal Itemset BCEG.

5 d) CHARM Algorithm with this dataset.

G	A	H	M	E	C	B
1,3,7	1,2,4,6	1,4,6,7	2,3,5,7	1,2,3,5,6,7	1,3,4,6,7	1,2,3,4,6



The closed Itemsets are: AB, AHBC, GBCE

(1) Initial pass are shown above after support based sorting of items. Sorted order is G, A, H, M, B, C, E.

(2) Then process extension from G. prune the itemset until we get all closed itemsets, Containing G has found.

(3) Continue charming algorithm for rest of branches.