Bootstrap assignment

There will be some functions that start with the word "grader" ex: grader_sampples(), grader_30().. etc, you should not change those function definition.

Every Grader function has to return True.

Importing packages

In [19]:
```
import numpy as np # importing numpy for numerical computation
from sklearn.datasets import load_boston # here we are using sklearn's boston dat
from sklearn.metrics import mean_squared_error # importing mean_squared_error met
import random
import pandas as pd
from sklearn.tree import DecisionTreeRegressor
from tqdm import tqdm
```

In [2]:
```
boston = load_boston()
x=boston.data #independent variables
y=boston.target #target variable
```

In [3]:
```
x.shape
```

Out[3]:
```
(506, 13)
```

In [4]:
```
x[:5]
```

Out[4]:
```
array([[6.3200e-03, 1.8000e+01, 2.3100e+00, 0.0000e+00, 5.3800e-01,
        6.5750e+00, 6.5200e+01, 4.0900e+00, 1.0000e+00, 2.9600e+02,
        1.5300e+01, 3.9690e+02, 4.9800e+00],
       [2.7310e-02, 0.0000e+00, 7.0700e+00, 0.0000e+00, 4.6900e-01,
        6.4210e+00, 7.8900e+01, 4.9671e+00, 2.0000e+00, 2.4200e+02,
        1.7800e+01, 3.9690e+02, 9.1400e+00],
       [2.7290e-02, 0.0000e+00, 7.0700e+00, 0.0000e+00, 4.6900e-01,
        7.1850e+00, 6.1100e+01, 4.9671e+00, 2.0000e+00, 2.4200e+02,
        1.7800e+01, 3.9283e+02, 4.0300e+00],
       [3.2370e-02, 0.0000e+00, 2.1800e+00, 0.0000e+00, 4.5800e-01,
        6.9980e+00, 4.5800e+01, 6.0622e+00, 3.0000e+00, 2.2200e+02,
        1.8700e+01, 3.9463e+02, 2.9400e+00],
       [6.9050e-02, 0.0000e+00, 2.1800e+00, 0.0000e+00, 4.5800e-01,
        7.1470e+00, 5.4200e+01, 6.0622e+00, 3.0000e+00, 2.2200e+02,
        1.8700e+01, 3.9690e+02, 5.3300e+00]])
```

In [5]:
```
x1 = pd.DataFrame(data=x)
```

In [6]:
```
x1.head()
```

Out[6]:

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33

In [ ]:

Task 1

Step - 1

- **Creating samples**
  Randomly create 30 samples from the whole boston data points
  - Creating each sample: Consider any random 303(60% of 506) data points from whole data set and then replicate any 203 points from the sampled points

    For better understanding of this procedure lets check this examples, assume we have 10 data points [1,2,3,4,5,6,7,8,9,10], first we take 6 data points randomly , consider we have selected [4, 5, 7, 8, 9, 3] now we will replicate 4 points from [4, 5, 7, 8, 9, 3], consder they are [5, 8, 3,7] so our final sample will be [4, 5, 7, 8, 9, 3, 5, 8, 3,7]
- **Create 30 samples**
  - Note that as a part of the Bagging when you are taking the random samples **make sure each of the sample will have different set of columns**
    Ex: Assume we have 10 columns[1 ,2 ,3 ,4 ,5 ,6 ,7 ,8 ,9 ,10] for the first sample we will select [3, 4, 5, 9, 1, 2] and for the second sample [7, 9, 1, 4, 5, 6, 2] and so on... Make sure each sample will have atleast 3 feautres/columns/attributes

Step - 2

Building High Variance Models on each of the sample and finding train MSE value

- Build a regression trees on each of 30 samples.
- Computed the predicted values of each data point(506 data points) in your corpus.
- Predicted house price of i^{th} data point
  $y^{i}_{pred} = \frac{1}{30} \sum_{k=1}^{30}(\text{predicted value of } x^{i} \text{ with } k^{th} \text{ model})$
- Now calculate the $MSE = \frac{1}{506} \sum_{i=1}^{506}(y^{i} - y^{i}_{pred})^{2}$

Step - 3

- **Calculating the OOB score**

- Predicted house price of i^{th} data point $y^i_{pred} = \frac{1}{k} \sum_{\text{k= model which was buit on samples not included } x^i} (\text{predicted value of } x^i \text{ with } k^{th} \text{ model})$
- Now calculate the $OOBScore = \frac{1}{506} \sum_{i=1}^{506} (y^i - y^i_{pred})^2$.

Task 2

- **Computing CI of OOB Score and Train MSE**
  - Repeat Task 1 for 35 times, and for each iteration store the Train MSE and OOB score
  - After this we will have 35 Train MSE values and 35 OOB scores
  - using these 35 values (assume like a sample) find the confidence intravels of MSE and OOB Score
  - you need to report CI of MSE and CI of OOB Score
  - Note: Refer the Central_Limit_theorem.ipynb to check how to find the confidence intravel

Task 3

- **Given a single query point predict the price of house.**

Consider xq= [0.18,20.0,5.00,0.0,0.421,5.60,72.2,7.95,7.0,30.0,19.1,372.13,18.60] Predict the house price for this point as mentioned in the step 2 of Task 1.

Task - 1

Step - 1

- **Creating samples**

Algorithm

Pesudo Code for generating Sample

```
def generating_samples(input_data, target_data):

    Selecting_rows <--- Getting 303 random row indices from the input_data

    Replcaing_rows <--- Extracting 206 random row indices from the "Selecting_rows"

    Selecting_columns<--- Getting from 3 to 13 random column indices

    sample_data<--- input_data[Selecting_rows[:,None],Selecting_columns]

    target_of_sample_data <--- target_data[Selecting_rows]

    #Replicating Data

    Replicated_sample_data <--- sample_data [Replaceing_rows]

    target_of_Replicated_sample_data<--- target_data[Replaceing_rows]

    # Concatinating data

    final_sample_data <---  perform vertical stack on  sample_data, Replicated_sample_data

    final_target_data<--- perform vertical stack on target_of_sample_data.reshape(-1,1), target_of_Replicated_sample_data.reshape(-1,1)

    return  final_sample_data,  final_target_data,  Selecting_rows,  Selecting_columns
```

- **Write code for generating samples**

In [7]:
```
def generating_samples(input_data, target_data):

    '''In this function, we will write code for generating 30 samples '''
    # you can use random.choice to generate random indices without replacement
    # Please have a look at this link https://docs.scipy.org/doc/numpy-1.16.0/ref
    # Please follow above pseudo code for generating samples


    # return sampled_input_data , sampled_target_data,selected_rows,selected_colu
    #note please return as lists

    selecting_row_list = np.random.choice(len(input_data), 303,replace=False)
    selecting_row   = input_data[selecting_row_list]
    selecting_row_list= np.sort(selecting_row_list)
    replacing_row_list =  np.random.choice(selecting_row_list, 203)
    replacing_row  = input_data[replacing_row_list]
    col_sampling_list  =  np.random.randint(0,input_data.shape[1],size=6)
    col_sampling_list= np.sort(col_sampling_list)



    sample_data  = input_data[selecting_row_list[:,None],col_sampling_list]
    df_sample_data = pd.DataFrame(sample_data,index=selecting_row_list)
    #print(type(sample_data))
    #print(sample_data.shape)
    target_of_sample_data  =target_data[selecting_row_list]

    sample_in_dict = {}
    sample_tar_dict = {}
    for i in selecting_row_list:
        sample_in_dict[i] = input_data[i, col_sampling_list]
        sample_tar_dict[i]= target_data[i]

    rep_sample_rows = []
    rep_target_rows = []
    for i in replacing_row_list:
        rep_sample_rows.append(list(sample_in_dict[i]))
        rep_target_rows.append(sample_tar_dict[i])

    rep_sample_rows = np.array(rep_sample_rows)
    rep_target_rows = np.array(rep_target_rows)
    rep_target_rows = rep_target_rows.reshape(-1,1)

    #Replacting_sampleData
    #print('*'*50)
    #print(replacing_row_list)
    #df_Replicated_sample = df_sample_data.loc[ replacing_row_list , : ]
    #Replicated_sample_data = df_Replicated_sample.to_numpy()
    #Replicated_sample_data = input_data[replacing_row_list[:,None],col_sampling_
    #target_replicated_data  = target_data[replacing_row_list]

    #concatinating data
    #final_sample_data = np.vstack((selecting_row, replacing_row))
    #final_target_data = np.vstack((target_of_sample_data.reshape(-1,1), target_r
    #final_sample_data = np.vstack((sample_data, Replicated_sample_data))
    final_sample_data = np.vstack((sample_data, rep_sample_rows))
```

```
    final_target_data = np.vstack((target_of_sample_data.reshape(-1,1), rep_targe
    #final_target_data = np.vstack((target_of_sample_data.reshape(-1,1), target_r

    return final_sample_data,final_target_data,selecting_row_list,col_sampling_li
```

Grader function - 1

In [8]:
```python
def grader_samples(a,b,c,d):
    length = (len(a)==506  and len(b)==506)
    sampled = (len(a)-len(set([str(i) for i in a]))==203)
    rows_length = (len(c)==303)
    column_length= (len(d)>=3)
    assert(length and sampled and rows_length and column_length)
    return True
a,b,c,d = generating_samples(x, y)
grader_samples(a,b,c,d)
```

Out[8]: True

- **Create 30 samples**

Run this code 30 times, so that you will 30 samples, and store them in a lists as shown below:

```
list_input_data=[]
list_output_data=[]
list_selected_row=[]
list_selected_columns=[]

for i in range(0,30):
   a,b,c,d=generating_sample(input_data,target_data)
  list_input_data.append(a)
  list_output_data.append(b)
  list_selected_row.append(c)
  list_selected_columns.append(d)
```

In [14]:
```
def create_samples(x,y):
    #a,b,c,d = generating_samples(x, y)
    #grader_samples(a,b,c,d)



    list_input_data =[]
    list_output_data =[]
    list_selected_row= []
    list_selected_columns=[]

    for i in range(0,30):
        a,b,c,d = generating_samples(x, y)
        list_input_data.append(a)
        list_output_data.append(b)
        list_selected_row.append(c)
        list_selected_columns.append(d)
    return list_input_data,list_output_data,list_selected_row,list_selected_colum
```

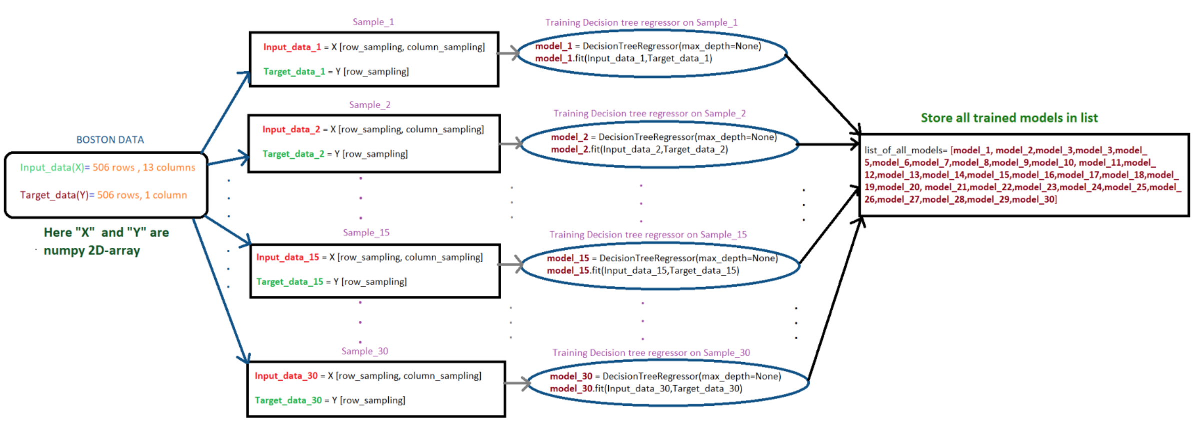
Grader function - 2

In [15]:
```
def grader_30(a):
    assert(len(a)==30 and len(a[0])==506)
    return True
a,b,c,d  = create_samples(x,y)
grader_30(a)
```

Out[15]: True

Step - 2

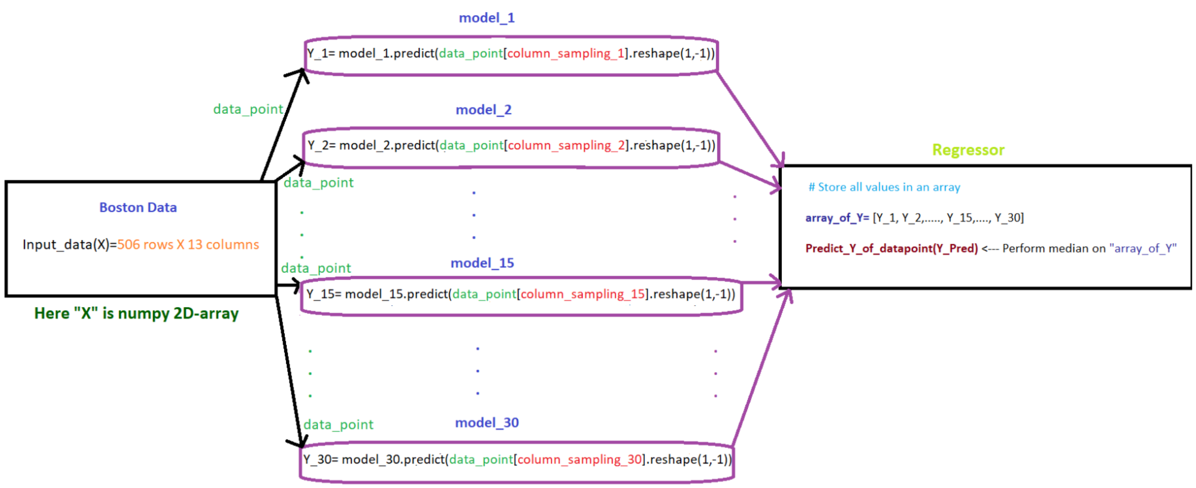
Flowchart for building tree



- **Write code for building regression trees**

In [16]:
```python
def train_model(list_input_data,list_output_data):
    list_all_models  = []
    for i in range(30):
        DT = DecisionTreeRegressor(random_state=123,max_depth=None)
        DT.fit(list_input_data[i],list_output_data[i])
        list_all_models.append(DT)
    return list_all_models
```

Flowchart for calculating MSE



After getting predicted_y for each data point, we can use sklearns mean_squared_error to calculate the MSE between predicted_y and actual_y.

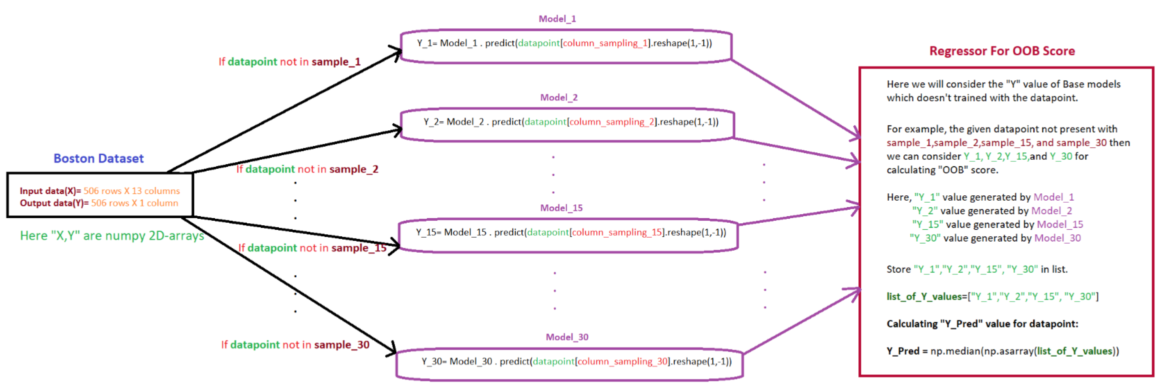
- **Write code for calculating MSE**

In [21]:
```python
def calculate_mse(x,y,list_input_data,list_output_data,list_selected_columns):
    #print(list_selected_columns)

    list_all_modelss  = train_model(list_input_data,list_output_data)

    #Apply ith Data point

    predicted_list = []
    Mean_sq_err = []
    for j, model in enumerate(list_all_modelss):
        #print(j)
        predict_y = model.predict(x[:, list_selected_columns[j]])
        predicted_list.append(predict_y)

    predicted_median = np.median(np.asarray(predicted_list), axis = 0)

    #print('MSE - ', mean_squared_error(y, np.array(predicted_median)))
    Mean_sq_err.append(mean_squared_error(y, np.array(predicted_median)))
    return Mean_sq_err,list_all_modelss

Mean_sq_error_val,list_models_DT   = calculate_mse(x,y,a,b,d)
print('MSE',Mean_sq_error_val)
```

```
MSE [0.03289964866051824]
```

Step - 3

Flowchart for calculating OOB score



Now calculate the $OOBScore = \frac{1}{506}\sum_{i=1}^{506}(y^i - y^i_{pred})^2$.

- **Write code for calculating OOB score**

In [22]:
```
def calculate_oob(x,y,list_input_data,list_output_data,list_selected_columns):
    OOB_pred  = []
    list_all_models = train_model(list_input_data,list_output_data)
    for ind_val in tqdm(range(len(x))):
        pred_Y_value = []
        for i in range(30):
            new_data = x[ind_val,list_selected_columns[i]]
            if list(new_data) not in list_input_data[i].tolist():
                #print(i)
                pred_Y_value.append(list_all_models[i].predict(x[:, list_selecte
    predicted_median_oob = np.median(np.asarray(pred_Y_value), axis = 0)
    OOB_pred.append(predicted_median_oob)

    OOB = []
    for i in range(len(y)):
        OOB_i = (y[i]- OOB_pred[0][i])**2
        OOB.append(OOB_i)
    OOB = float(sum(OOB)/len(y))
    return OOB
OOB_Valss   = calculate_oob(x,y,a,b,d)
print('MSE',OOB_Valss)
```

```
100%|████████████████████████████████████████
██| 506/506 [00:04<00:00, 115.54it/s]

MSE 0.3826648276240667
```

In [ ]:

Task 2

In [23]:
```
# Run this model for 35 times
Main_MSE = []
Main_OOB = []
for run in tqdm(range(35)):
    #in_data,out_data,selc_row,sele_col,Mean_sq_error_val,OOB_squ_error = 0
    in_data,out_data,selc_row,sele_col = create_samples(x,y)
    Mean_sq_error_val,list_models_DT  = calculate_mse(x,y,in_data,out_data,sele_c
    OOB_squ_error =  calculate_oob(x,y,in_data,out_data,sele_col)
    Main_MSE.append(Mean_sq_error_val)
    Main_OOB.append(OOB_squ_error)
```

...

In [59]:
```
from prettytable import PrettyTable
PT_MSE_OOB = PrettyTable()
PT_MSE_OOB = PrettyTable(["MSE","OOB"])
for i in range(30):
    row = []
    row.append(Main_MSE[i])
    row.append(Main_OOB[i])
    PT_MSE_OOB.add_row(row)
#PT_MSE_OOB.add_column(Main_OOB)
#print(Main_MSE)
#print(Main_OOB)
```

In [60]:
```
print(PT_MSE_OOB)
```

```
+-----------------------+---------------------+
|          MSE          |         OOB         |
+-----------------------+---------------------+
| [0.20223152173913034] |  0.4407352900399289 |
| [0.09252099802371547] |  0.5703754940711462 |
|  [0.4805681818181818] |  1.938851593379446  |
| [0.04793478260869568] |  0.4915612648221345 |
| [0.07477669219367586] |   0.43462573561704  |
| [0.09354437558967518] |  4.124439605149585  |
| [0.25939644268774686] |  0.5059288537549408 |
| [0.05691329051383396] | 0.40952569169960457 |
|  [0.5959564229249009] |  0.8900689970355733 |
|  [1.2070899209486163] |  1.7470229468599037 |
| [0.031198342446456386] |  0.8793922924901192 |
| [0.09716482328852552] | 0.49723875225686825 |
| [0.31206838081906013] |  1.0535525691699603 |
|  [0.0353525691699605] |  0.4324264382960036 |
|  [0.067165175758997 9] |  1.4902667984189724 |
| [0.025918972332015813] |  0.3408695652173914 |
|  [0.090706324110672]  |  0.6112697628458501 |
| [0.031472332015810275] |  0.8610869565217392 |
| [0.05297924901185769] | 0.46578667105841015 |
| [0.025423122529644293] | 0.49139086400407483 |
|  [0.4756364253952571] |  0.8644424709046995 |
| [0.32142197895471303] |  0.9458307422046552 |
| [0.08784488910847606] |  0.4499258893280632 |
| [0.030103892182696507] |  0.8314031620553366 |
| [0.18178416166573688] |  0.6634052024529807 |
| [0.028675889328063257] |  0.4247677865612649 |
| [0.20263944906025647] |  0.3870895256916997 |
| [0.12328406346069401] |  4.156658127673149  |
| [0.021244185696608762] |  0.7298403875713659 |
| [0.026209288537549395] |  2.0604347826086955 |
+-----------------------+---------------------+
```

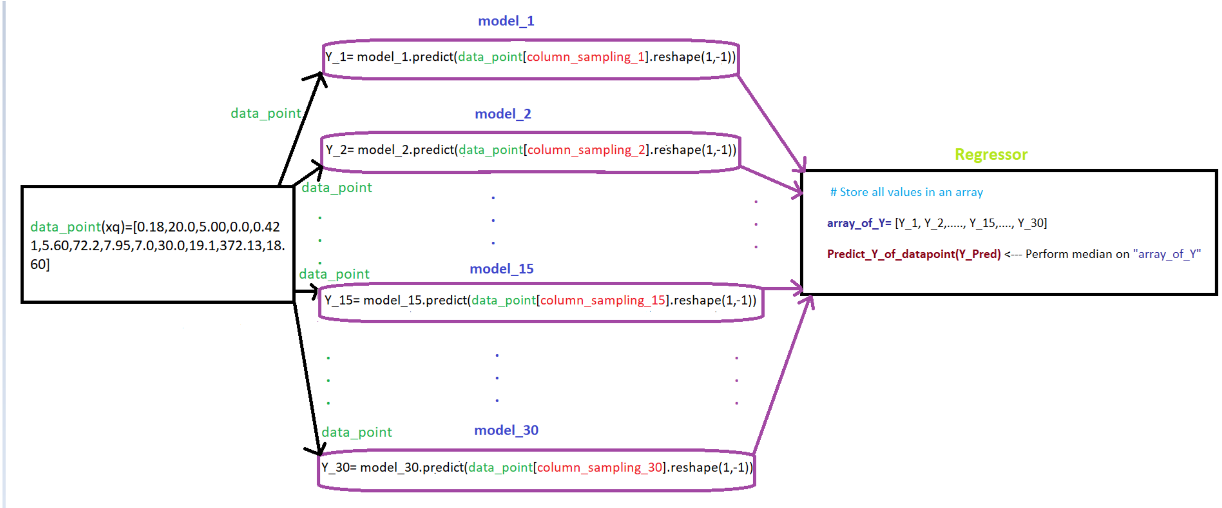
In [24]:
```
from prettytable import PrettyTable
PT = PrettyTable()
PT = PrettyTable(["#samples", "Sample Size", "Sample mean","Left C.I","Right C.I'

sample=np.array(Main_MSE)
sample_mean = sample.mean()
sample_std =  sample.std()
sample_size = len(sample)
# here we are using sample standard deviation instead of population standard devi
left_limit  = np.round(sample_mean - 2*(sample_std/np.sqrt(sample_size)), 3)
right_limit = np.round(sample_mean + 2*(sample_std/np.sqrt(sample_size)), 3)
row = []
#row.append(i+1)
row.append('MSE')
row.append(sample_size)
row.append(sample_mean)
row.append(left_limit)
row.append(right_limit)
PT.add_row(row)
#print(PT)
sample=np.array(Main_OOB)
sample_mean_b = sample.mean()
sample_std_b =  sample.std()
sample_size_b = len(sample)
# here we are using sample standard deviation instead of population standard devi
left_limit_b  = np.round(sample_mean_b - 2*(sample_std_b/np.sqrt(sample_size_b)),
right_limit_b = np.round(sample_mean_b + 2*(sample_std_b/np.sqrt(sample_size_b)),
row_b = []
#row.append(i+1)
row_b.append('OOB')
row_b.append(sample_size_b)
row_b.append(sample_mean_b)
row_b.append(left_limit_b)
row_b.append(right_limit_b)
PT.add_row(row_b)
print(PT)
```

```
+----------+-------------+---------------------+----------+-----------+
| #samples | Sample Size |     Sample mean     | Left C.I | Right C.I |
+----------+-------------+---------------------+----------+-----------+
|   MSE    |      35     | 0.18525431622028068 |  0.109   |   0.262   |
|   OOB    |      35     |  1.0043329280879463 |  0.694   |   1.315   |
+----------+-------------+---------------------+----------+-----------+
```

Task 3

Flowchart for Task 3

Hint: We created 30 models by using 30 samples in TASK-1. Here, we need send query point "xq" to 30 models and perform the regression on the output generated by 30 models.



- **Write code for TASK 3**

In [62]:
```
#Consider xq= [0.18,20.0,5.00,0.0,0.421,5.60,72.2,7.95,7.0,30.0,19.1,372.13,18.60
xq = [0.18,20.0,5.00,0.0,0.421,5.60,72.2,7.95,7.0,30.0,19.1,372.13,18.60]

xq_predicted = []
xq = np.array(xq)
for j, model in enumerate(list_models_DT):
    #print(j)
    predict_y = model.predict(xq[sele_col[j]].reshape(1,-1))
    xq_predicted.append(predict_y)

predicted_median_xq = np.median(np.asarray(xq_predicted), axis = 0)

print('House Price of xq is', predicted_median_xq[0])
```

```
House Price of xq is 19.4
```

Write observations for task 1, task 2, task 3 indetail

Note 1: When the model train with datapoints present in the model, Mean sqaure error almost close to zero, means giving good and almost accuurate results, means low bias

Note 2: when we consider OOB, still model perform descent prediction of our target from that we will conlude that low bias and low variances. so Random forest did pretty much good job.

Note 3: As part of conidences Interval, 95 % C.I, MSE is between 0.1 to 0.2 and OOB is 0.6 to 1.3.

In [ ]:

```

```