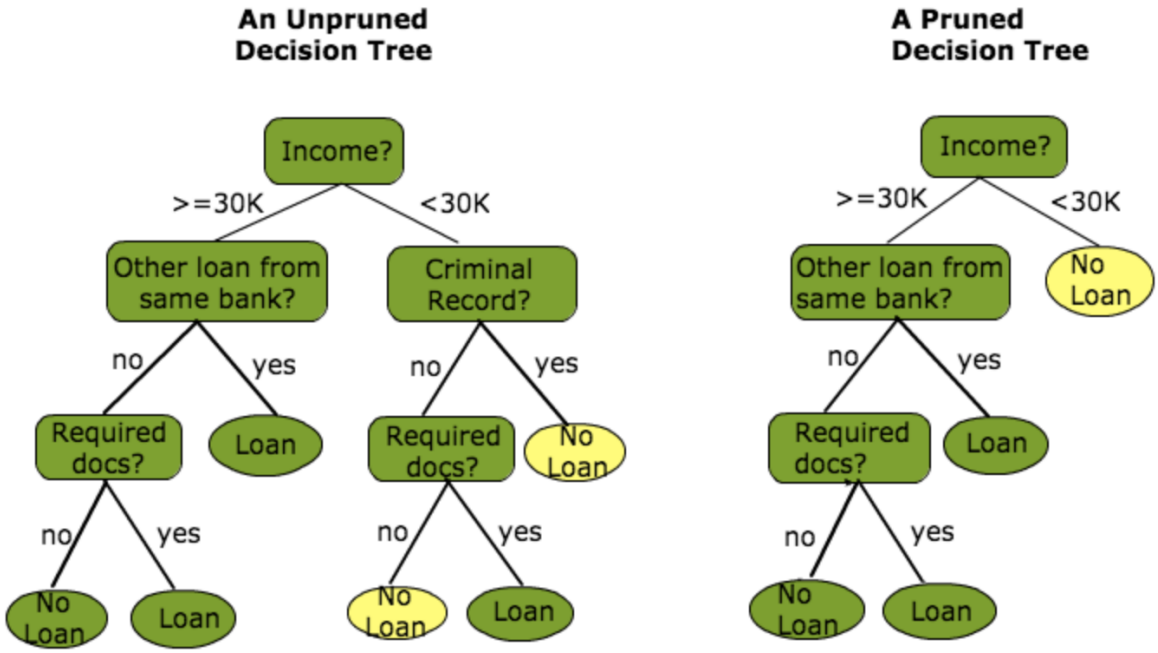
Pruning

- Machine learning is a problem of trade-offs. The classic issue is over-fitting versus under-fitting. Over-fitting happens when a model fits on training data so well and it fails to generalize well.ie, it also learns noises on top of the signal. Under-fitting is an opposite event: the model is too simple to find the patterns in the data.
- Pruning is a technique used to deal with overfitting, that reduces the size of DTs by removing sections of the Tree that provide little predictive or classification power.
- The goal of this procedure is to reduce complexity and gain better accuracy by reducing the effects of overfitting and removing sections of the DT that may be based on noisy or erroneous data. There are two different strategies to perform pruning on DTs:

```
  1. Pre-prune: When you stop growing DT branches when information b
ecomes unreliable.
  2. Post-prune: When you take a fully grown DT and then remove leaf
nodes only if it results in a better model performance. This way, yo
u stop removing nodes when no further improvements can be made.
```



Example On A Dataset

In [1]:
```
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_breast_cancer
from sklearn.tree import DecisionTreeClassifier
```

In [2]:
```
X, y = load_breast_cancer(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

clf = DecisionTreeClassifier(random_state=0)
clf.fit(X_train,y_train)
```

Out[2]:
```
DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None, criterion='gini',
                       max_depth=None, max_features=None, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, presort='deprecated',
                       random_state=0, splitter='best')
```

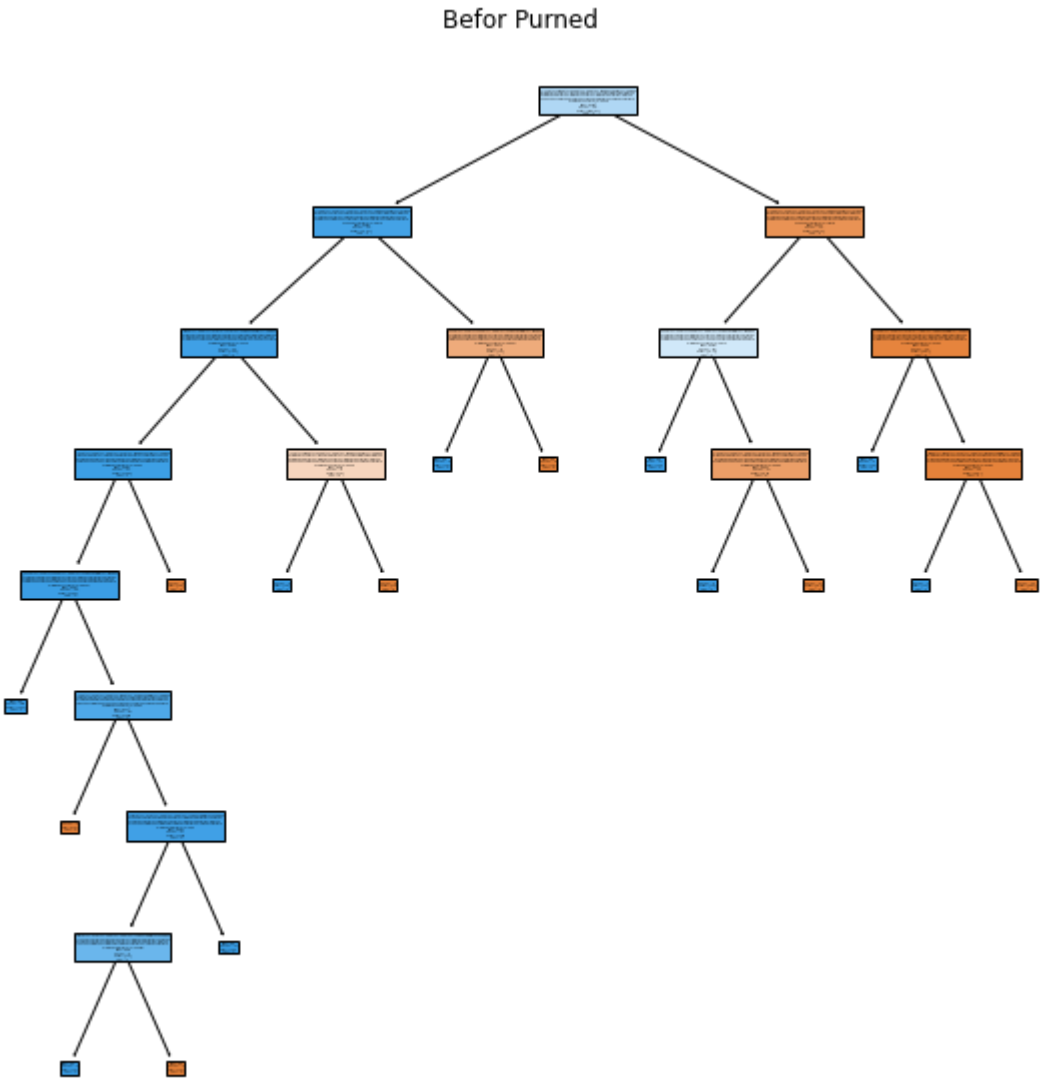
In [3]:
```
pred=clf.predict(X_test)
from sklearn.metrics import accuracy_score
print("Training Accuracy :", clf.score(X_train, y_train))
print("Testing Accuracy :", accuracy_score(y_test,pred))
```

```
Training Accuracy : 1.0
Testing Accuracy : 0.8811188811188811
```

We can see that in our train data we have 100% accuracy. But in test data model is not well generalizing. We have just 88% accuracy. Over model is clearly overfitting. We will avoid overfitting through pruning. We will do cost complexity prunning

In [4]:
```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
```

In [5]:
```
from sklearn import tree
plt.figure(figsize=(10,10))
features = X
classes = ['0','1']
tree.plot_tree(clf,feature_names=features,class_names=classes,filled=True)
plt.title('Befor Pruning')
plt.show()
```



So here We use Two techniques Post and Pre pruning

Methood 1: Pre Pruning techniques

- Pre pruning is nothing but stoping the growth of decision tree on an early stage. For that we can limit the growth of trees by setting constrains. We can limit parameters like max_depth , min_samples etc.
- An effective way to do is that we can grid search those parameters and choose the optimum values that gives better performace on test data.
- it is our manually hyperparameter technique
- As of now we will control these parameters

```
        - max_depth: maximum depth of decision tree
        - min_sample_split: The minimum number of samples required to
    split an internal node:
        - min_samples_leaf: The minimum number of samples required to
    be at a leaf node.
```

In [6]:
```
from sklearn.model_selection import RandomizedSearchCV, GridSearchCV
params = {'max_depth': [2,4,6,8,10,12],
          'min_samples_split': [2,3,4],
          'min_samples_leaf': [1,2]}

clf = tree.DecisionTreeClassifier()
gcv = GridSearchCV(estimator=clf,param_grid=params)
gcv.fit(X_train,y_train)
```

Out[6]:
```
GridSearchCV(cv=None, error_score=nan,
             estimator=DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None,
                                              criterion='gini', max_depth=None,
                                              max_features=None,
                                              max_leaf_nodes=None,
                                              min_impurity_decrease=0.0,
                                              min_impurity_split=None,
                                              min_samples_leaf=1,
                                              min_samples_split=2,
                                              min_weight_fraction_leaf=0.0,
                                              presort='deprecated',
                                              random_state=None,
                                              splitter='best'),
             iid='deprecated', n_jobs=None,
             param_grid={'max_depth': [2, 4, 6, 8, 10, 12],
                         'min_samples_leaf': [1, 2],
                         'min_samples_split': [2, 3, 4]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring=None, verbose=0)
```

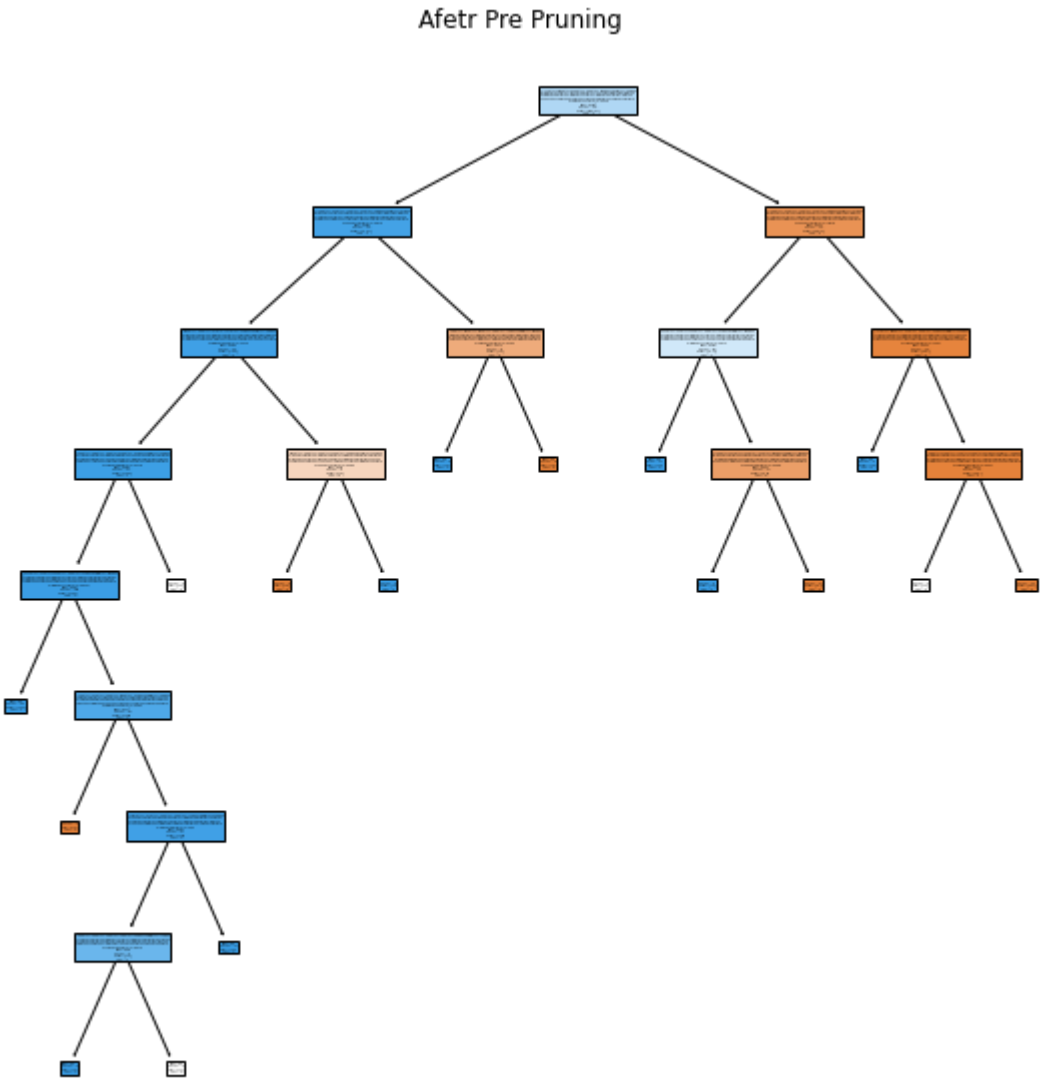
In [7]:
```
model = gcv.best_estimator_
model.fit(X_train,y_train)
y_train_pred = model.predict(X_train)
y_test_pred = model.predict(X_test)

print(f'Train score {accuracy_score(y_train_pred,y_train)}')
print(f'Test score {accuracy_score(y_test_pred,y_test)}')
```

```
Train score 0.9929577464788732
Test score 0.9230769230769231
```

We can see that tree is pruned and there is less improvement in test accuracy.But still there is still scope of improvement. So now we Go towards Post Pruning.

In [8]:
```
from sklearn import tree
plt.figure(figsize=(10,10))
features = X
classes = ['0','1']
tree.plot_tree(model,feature_names=features,class_names=classes,filled=True)
plt.title('Afetr Pre Pruning')
plt.show()
```



Method 2: Post Pruning Example

- There are several post pruning techniques. Cost complexity pruning is one of the important among them.

Cost Complexity Pruning:

```
    - Decision trees can easily overfit. One way to avoid it is to l
imit the growth of trees by setting constrains. We can limit paramet
ers like max_depth , min_samples etc. But a most effective way is to
use post pruning methods like cost complexity pruning. This helps to
improve test accuracy and get a better model.

    - Cost complexity pruning is all about finding the right paramet
er for alpha.We will get the alpha values for this tree and will che
ck the accuracy with the pruned trees.
```

In [9]:
```
path = clf.cost_complexity_pruning_path(X_train, y_train)
ccp_alphas, impurities = path.ccp_alphas, path.impurities
```

- cost_complexity_pruning_path function gives you two values one is alphas and another is impurity
- If you use alpha value you will be know how depth your tree is.
- ccp_alphas is going to find out the weak point with respect to the leaf node and it return a list of values

In [10]:
```
ccp_alphas
```

Out[10]:
```
array([0.        , 0.00226647, 0.00464743, 0.0046598 , 0.0056338 ,
       0.00704225, 0.00784194, 0.00911402, 0.01144366, 0.018988  ,
       0.02314163, 0.03422475, 0.32729844])
```

In [11]:
```
clfs = []
for ccp_alpha in ccp_alphas:
    clf = DecisionTreeClassifier(random_state=0, ccp_alpha=ccp_alpha)
    clf.fit(X_train, y_train)
    clfs.append(clf)
print("Number of nodes in the last tree is: {} with ccp_alpha: {}".format(
     clfs[-1].tree_.node_count, ccp_alphas[-1]))
```

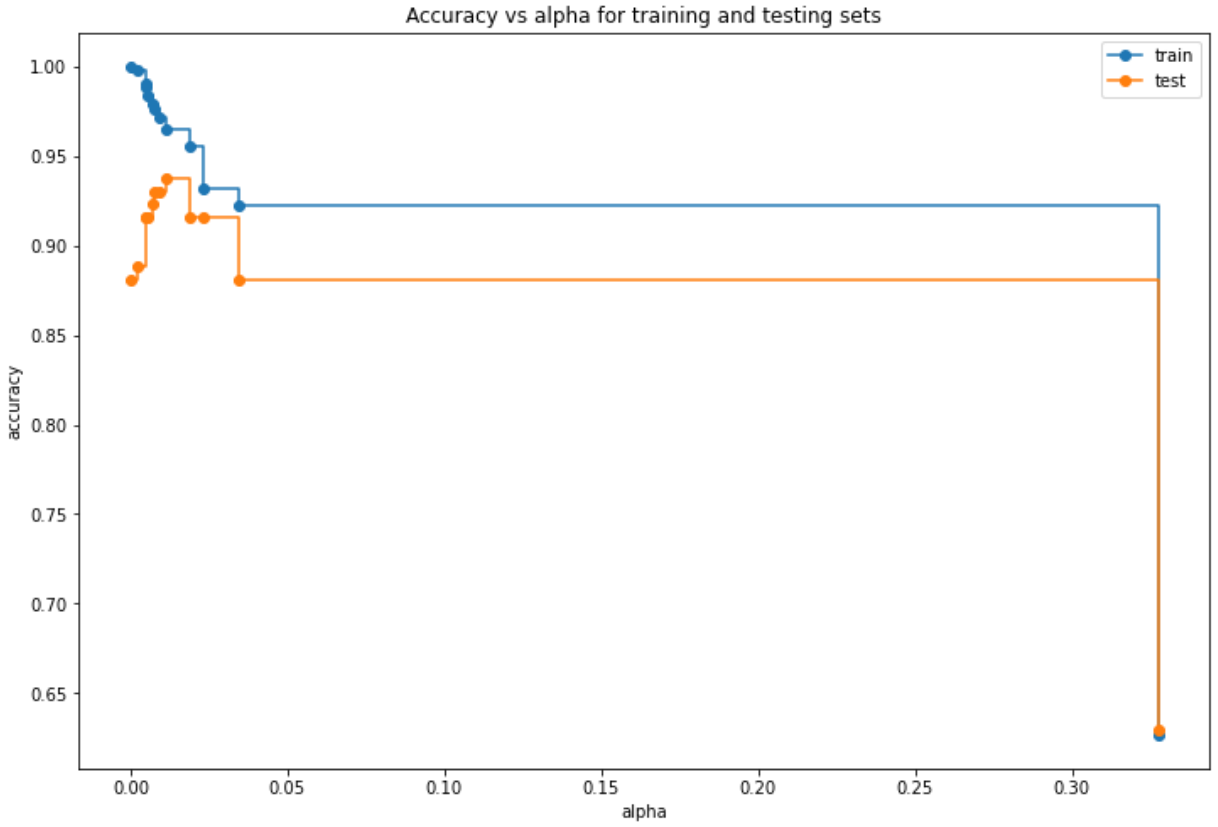
```
Number of nodes in the last tree is: 1 with ccp_alpha: 0.3272984419327777
```

For the remainder of this example, we remove the last element in clfs and ccp_alphas, because it is the trivial tree with only one node. Here we show that the number of nodes and tree depth decreases as alpha increases.

Accuracy vs alpha for training and testing sets

In [12]:
```
train_scores = [clf.score(X_train, y_train) for clf in clfs]
test_scores = [clf.score(X_test, y_test) for clf in clfs]


fig, ax = plt.subplots(figsize=(12,8))
ax.set_xlabel("alpha")
ax.set_ylabel("accuracy")
ax.set_title("Accuracy vs alpha for training and testing sets")
ax.plot(ccp_alphas, train_scores, marker='o', label="train",
        drawstyle="steps-post")
ax.plot(ccp_alphas, test_scores, marker='o', label="test",
        drawstyle="steps-post")
ax.legend()
plt.show()
```



- When ccp_alpha is set to zero and keeping the other default parameters of :class:DecisionTreeClassifier, the tree overfits, leading to a 100% training accuracy and 88% testing accuracy.
- As alpha increases, more of the tree is pruned, thus creating a decision tree that generalizes better.
- In this example, setting ccp_alpha=0.03 maximizes the testing accuracy. after that the accuracy is going to reduce.

In [13]:
```
clf = DecisionTreeClassifier(random_state=0, ccp_alpha=0.03)
clf.fit(X_train,y_train)
```

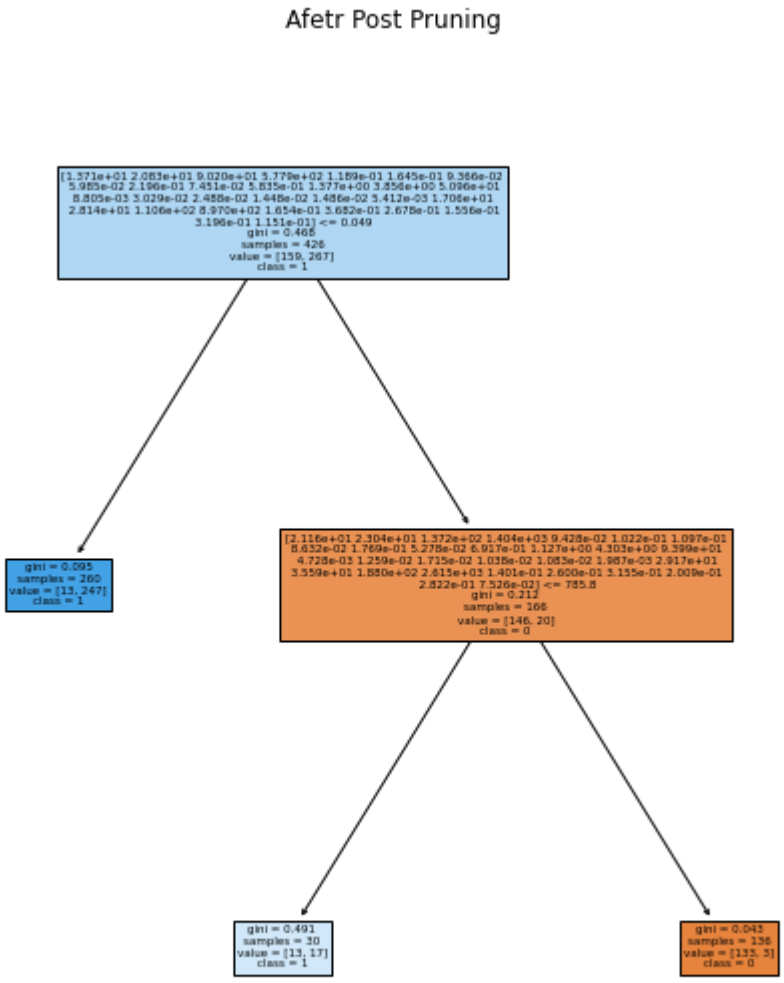
Out[13]:
```
DecisionTreeClassifier(ccp_alpha=0.03, class_weight=None, criterion='gini',
                       max_depth=None, max_features=None, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, presort='deprecated',
                       random_state=0, splitter='best')
```

In [14]:
```
pred=clf.predict(X_test)
from sklearn.metrics import accuracy_score
print("Training Accuracy :", clf.score(X_train, y_train))
print("Testing Accuracy :", accuracy_score(y_test,pred))
```

```
Training Accuracy : 0.931924882629108
Testing Accuracy : 0.916083916083916
```

- Now see the problem of overfit is gone and our model now perfect work.

In [15]:
```
from sklearn import tree
plt.figure(figsize=(10,10))
features = X
classes = ['0','1']
tree.plot_tree(clf,feature_names=features,class_names=classes,filled=True)
plt.title('Afetr Post Pruning')
plt.show()
```



- Now you see that the size of decision tree significantly got reduced.
- postpruning is much efficient than prepruning. This all about Purning.