SPLAY TREE VS RED BLACK TREE

***Splay tree**: In splay trees accessing a node moves it to the top, no other balance conditions are guaranteed. Whereas the balance of a BST is dependent upon the only the order in which nodes were inserted into the tree, splay tree balance is dependent upon the order in which they were accessed.The worst case scenario for balance is when nodes are accessed in ascending or descending order. Random access is generally OK on average,although this doesn't necessarily yield the multi access advantage. It is most efficient, nearly constant time, to access a recently accessed node. An example use case is a cache (files may be accessed multiple times).*

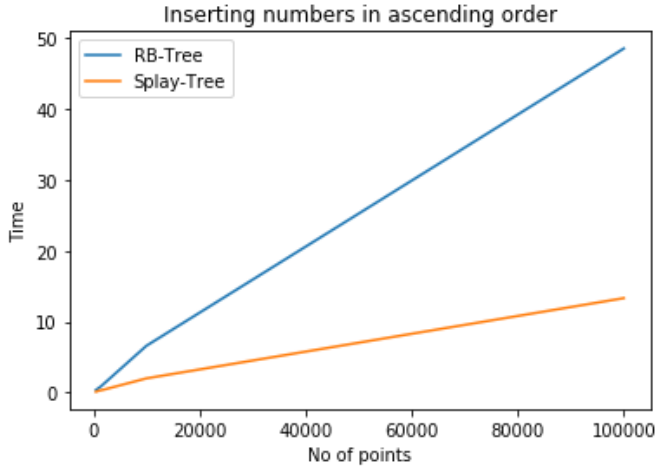
Red Black Tree: Red-Black trees are a popular tree structure for real time applications. They are used in the Linux Kernel, as well as the C++ STL (map).Red-black trees are height balanced. Since find operations are bound by height for worst case analysis,and since find operations are inherent in many other operations (such as insertand delete), maintaining height balance for a tree increases efficiency. The findalgorithm loses efficiency when compared to AVL trees which maintain both heightand weight balance. However, maintaining balance in a red-black tree is moreefficient than an AVL tree, making it a better choice for some real-time applications.

We will compare this two trees based on different operations:
1.Insertion.
2.Searching.
3.Deletion.

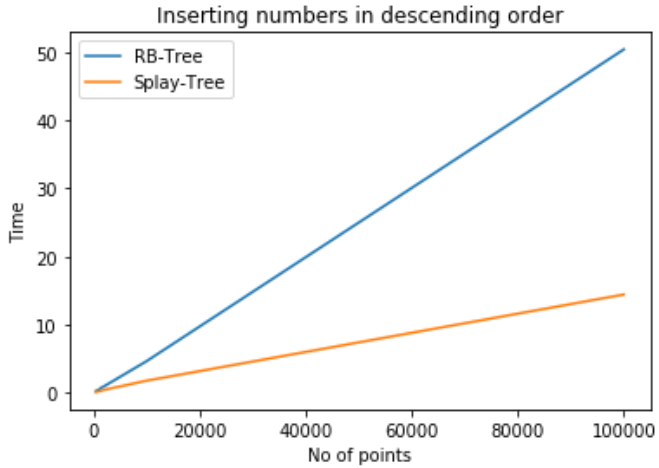
Insertion

Consider three different cases while inserting elements into the tree
1.Elements inserted in ascending order.
2.Elements inserted in descending order.
3.Elements inserted in random order.

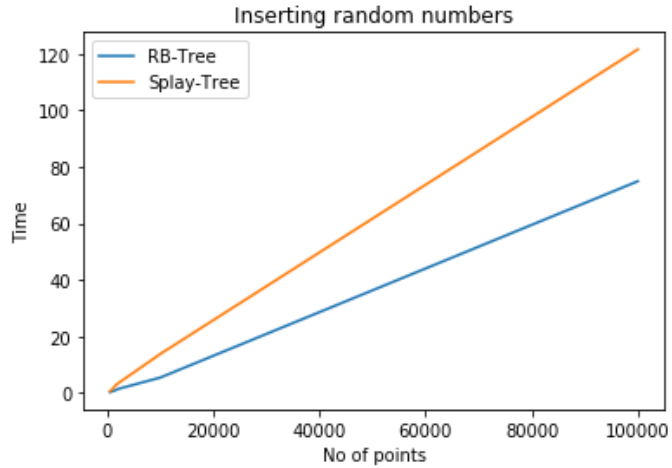
In [10]:
```
import matplotlib.pyplot as plt
#inserting elements in rb tree in ascending order
y=[0.38,0.659,0.93,1.27,6.604,48.468]
x=[500,1000,1500,2000,10000,100000]
#inserting elements into sply tree in ascending order
y1=[0.14,0.278,0.362,0.44,2.022,13.32]
x1=[500,1000,1500,2000,10000,100000]
plt.ylabel("Time")
plt.xlabel("No of points")
plt.title("Inserting numbers in ascending order")
plt.plot(x,y,label="RB-Tree")
plt.plot(x1,y1,label="Splay-Tree")
plt.legend(loc="upper left")
plt.show()
```



In [3]:
```
#inserting elements in rb tree in descending order
y=[0.26,0.43,0.75,0.94,4.618,50.39]
x=[500,1000,1500,2000,10000,100000]
#inserting elements into splay tree in descending order
y1=[0.095,0.217,0.298,0.422,1.77,14.42]
x1=[500,1000,1500,2000,10000,100000]
plt.ylabel("Time")
plt.xlabel("No of points")
plt.title("Inserting numbers in descending order")
plt.plot(x,y,label="RB-Tree")
plt.plot(x1,y1,label="Splay-Tree")
plt.legend(loc="upper left")
plt.show()
```



In [4]:
```
#inserting random elements in rb tree
y=[0.303,0.681,0.936,1.415,5.466,74.908]
x=[500,1000,1500,2000,10000,100000]
#inserting random elements into splay tree
y1=[0.634,1.458,2.587,3.378,13.822,121.55]
x1=[500,1000,1500,2000,10000,100000]
plt.ylabel("Time")
plt.xlabel("No of points")
plt.title("Inserting random numbers")
plt.plot(x,y,label="RB-Tree")
plt.plot(x1,y1,label="Splay-Tree")
plt.legend(loc="upper left")
plt.show()
```



Observation for inserting elements

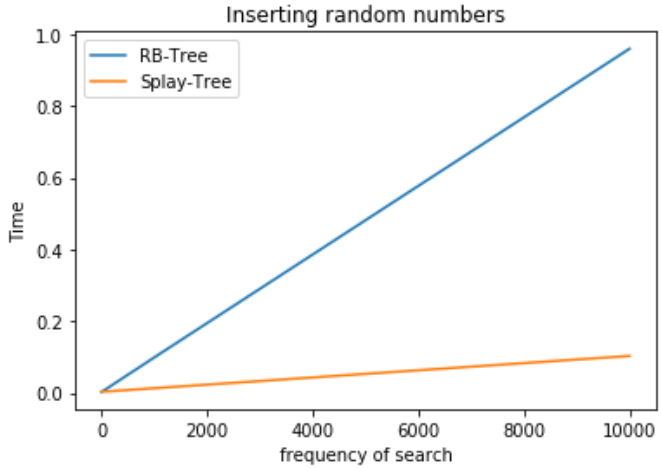
Case 1&2: Skew trees will be formed in splay tree because of performing splaying operation after each insertion of element and bringing that to the root,whereas in red-black tree there will be multiple rotations inorder to maintain the property of red-black trees so inserting elements either in ascending order or in descending order will take more time in red-black tree when compared to splay tree

Case 3: While inserting elements in random order splay tree will take more time when compared to red-black tree because splay will bring the newly inserted element to the rooot after each insertion but red-black tree maintains its property while inserting elements into the tree by performing appropraite

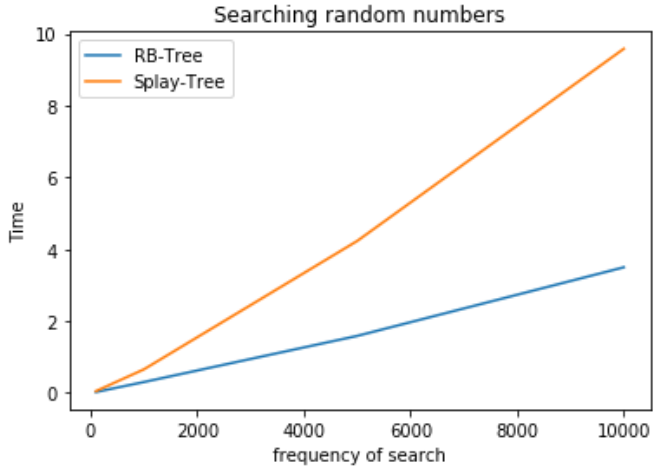
Search

To find the performance of search operation in both trees we have inserted 100000 random elements in bothe trees

In [5]:
```
#inserting random values between 1-100000 and searching for element 100 repeatdely for different freqencies
x=[1,100,1000,10000]
y=[0.003,0.012,0.099,0.96]
x1=[1,100,1000,10000]
y1=[0.004,0.004,0.013,0.103]
plt.ylabel("Time")
plt.xlabel("frequency of search")
plt.title("Inserting random numbers")
plt.plot(x,y,label="RB-Tree")
plt.plot(x1,y1,label="Splay-Tree")
plt.legend(loc="upper left")
plt.show()
```



In [8]:
```
#searching random elements for differnt freqencies
x=[100,1000,5000,10000]
y=[0.025,0.303,1.587,3.497]
x1=[100,1000,5000,10000]
y1=[0.05,0.656,4.23,9.58]
plt.ylabel("Time")
plt.xlabel("frequency of search")
plt.title("Searching random numbers")
plt.plot(x,y,label="RB-Tree")
plt.plot(x1,y1,label="Splay-Tree")
plt.legend(loc="upper left")
plt.show()
```

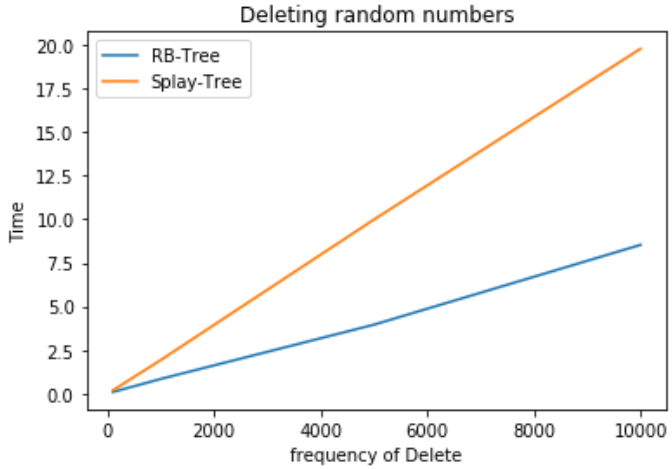


Observation for searching elements

Case 1: We are performing search operation of constant element for multiple times say(1,100,1000,10000) due to splaying property of splay tree there will be huge time gap between this two trees,in splay tree after first search operation it will brimg that constant searched element into the root and for next search operations it will take constant time to access that element whereas in red-black trees it takes the same amount of time for every of search operation of that constant element

Case 2:we are searching random elements multiple times say(1,100,1000,10000), in this case red-black trees performs better than splay trees because after every search opeation the splay structure will be changed whereas the structure of red-black tree remains constant

In [9]:
```
#deleteing random elements with given freqencies
x=[100,1000,5000,10000]
y=[0.085,0.835,3.95,8.53]
x1=[100,1000,5000,10000]
y1=[0.179,1.922,9.985,19.789]
plt.ylabel("Time")
plt.xlabel("frequency of Delete")
plt.title("Deleting random numbers")
plt.plot(x,y,label="RB-Tree")
plt.plot(x1,y1,label="Splay-Tree")
plt.legend(loc="upper left")
plt.show()
```



Observation for deleting elements

Here we are deleting a random number for multiple times say(100,1000,5000,10000).Splay tree will take more time when compared to red-black tree because while deleting element we will first search that element and bring that element to the root and delete that element and make inorder succesor as the root whereas in red-black tree we will search that element and delete it and we will perform approprate rotations to make the properties of the red black tree

Conclusion

1.If our application has more number of insert operation than search operation prefer **red-black tree.**
2.If our application has repeated number of search operation for the same element prefer **splay tree**(cache implementation).