261p Project 2

Zhecheng Li, 35688296

1. Data structures

In this lab, I implemented 4 different data structures, which are standard (unoptimized) binary tree, AVL tree, Treap and Splay tree.

The time complexity of insertion, search, deletion of each data structure is O(log n).

For standard binary tree, there is no rotation in it. For AVL tree, insertion needs at most 2 rotations and deletion may need as many as O(log n) rotations. For Treap, expected number of rotation is at most 2. For Splay tree, number of rotations depends on which node to delete. If it is close to the leaf, cost is higher and if it is close to the root, cost is lower.

1. Implementation

I implemented all these data structures in Python.

1. Description of data structures
   1. Standard binary tree
      1. Insert (k)

If no node in tree:

Make it root

Else:

Node = root

If k < node.key

node = node .leftChild

elif k > node.key

node = node.rightChild

else

break

make node child of its parent

* + 1. Search (k)

node = root

if k < node.key

node = node.leftChild

elif k > node.key

node = node.rightChild

else

break

return node

* + 1. Delete (k)

node = Self.Search(k)

if node is a leaf node:

delete node

elif node has only leftChild

node.leftChild.parent = node.parent

elif node has only rightChild

node.rightChild.parent = node.parent

else

find maxNode which has max key in node’s left subtree

replace node with maxNode

* 1. AVL tree
     1. Insert (k)

Start from root, node = root

If node is None

Node = new AVLNode(k)

elif k < node.key

Node.leftChild = self.insert(leftChild, k)

If height(node.leftChild) == height(node.rightChild) + 2

If k < node.leftChild.key

Node = self.ll(node)

Else

Node = self.lr(node)

Elif k > node.key

Node.rightChild = self.insert(node.rightChild, k)

If height(node.rightChild) == height(node.leftChild) + 2

If k > node.rightChild.key

Ndoe = self.rr(node)

Else

Node = self.rl(node)

Node.height = max(height(node.leftChild), height(node.rightChild)) + 1

Return node

* + 1. Search (k)

If root is None

Make AVLNode(k) root

Else  
 node = root

While node is not None

If k < node.key

Node = node.leftChild

Elif k > node.key

Node = node.rightChild

Else

Break

Return node

* + 1. Delete (k)

Start from root, tree = root, node = self.search(k)

If node.key < tree.key

Tree.leftChild = self.delete(tree.leftChild, node)

If height(tree.rightChild) == height(tree.leftChild) + 2

R = tree.rightChild

If height(r.leftChild) > height(r.rightChild)

Tree = self.rl(tree)

Else

Tree = self.rr(tree)

Elif node.key > tree.key

Tree.rightChild = self.delete(tree.rightChild, node)

If height(tree.leftChild) == height(tree.rightChild) + 2

L = tree.leftChild

If height(l.rightChild) > height(l.leftChild)

Tree = self.lr(tree)  
 else

Tree = self.ll(tree)

Else

If tree.leftChild is not None and tree.rightChild is not None

If height(tree.leftChild) > height(tree.rightChild)

Find MaxNode, node with max key in tree’s left subtree

Tree.key = maxNode.key

Tree.leftChild = self.delete(tree.leftChild, maxNode)

Else

Find minNode, node with min key in tree’s right subtree

Tree.key = minNode.key

Tree.rightChild = self.delete(tree.rightChild, minNode)

Else

Tree = tree.leftChild is tree.leftChild is not None else tree.rightChild

Return tree

* + 1. ll (k2)

k1 = k2.leftChild

k2.leftChild = k1.rightChild

k1.rightChild = k2

k2.height = max(height(k2.leftChild), height(k2.rightChild)) + 1

k1.height = max(height(k1.leftChild), height(k1.rightChild)) + 1

return k1

* + 1. rr (k1)

k2 = k1.leftChild

k1.rightChild = k2.leftChild

k2.leftChild = k1

k1.height = max(height(k1.leftChild), height(k1.rightChild)) + 1

k2.height = max(height(k2.leftChild), height(k2.rightChild)) + 1

return k2

* + 1. lr (k3)

k3.leftChild = rr(k3.leftChild)

return self.ll(k3)

* + 1. rl (k1)

k1.rightChild = self.ll(k1.rightChild)

return self.rr(k1)

* 1. Treap
     1. Insert (k)

Node = insert k with standard binary tree insertion

Node.priority = random()

While node is not root and node.priority > node.parent.priority

If node is a left child

Rotate\_right(node.parent)

Else

Rotate\_left(node.parent)

* + 1. Search (k)

Node = Binary tree search(k)

* + 1. Delete (node)

While node is not a leaf

If node.leftChild is None

Rotate\_left(node)

Elif node.rightChild is None or node.leftChild.priority > node.rightChild.priority

Rotate\_right(node)

Else

Rotate\_left(node)

Delete node

* 1. Splay tree
     1. Insert (k)

Node = Binary tree search(k)

Splay(node)

* + 1. Search (k)

Node = Binary tree search(k)

If node is not None

Splay(node)

Else

Splay(last node on search path)

* + 1. Delete

Node = binary tree search(k)

If node is not None

Delete node

Splay(node.parent)

Else

Splay(last node on search path)

* + 1. Splay (node)

If node has no grandparent

Zig(node)

Elif node is LL or RR grandchild

Zig-zig(node)

Elif node is LR or RL grandchild

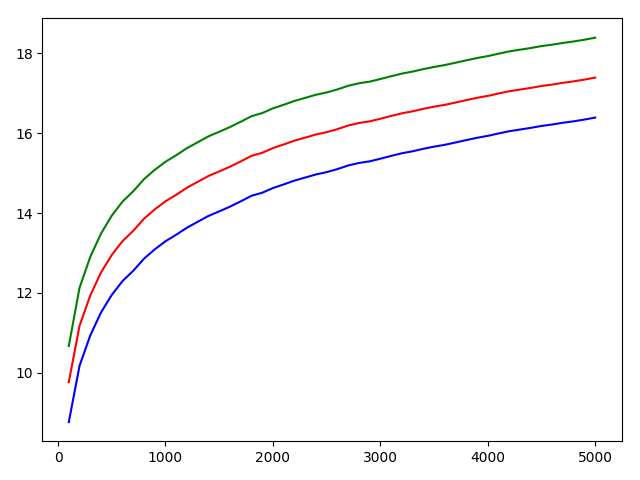
Zig-zag(node)

1. Performance
   1. Testing

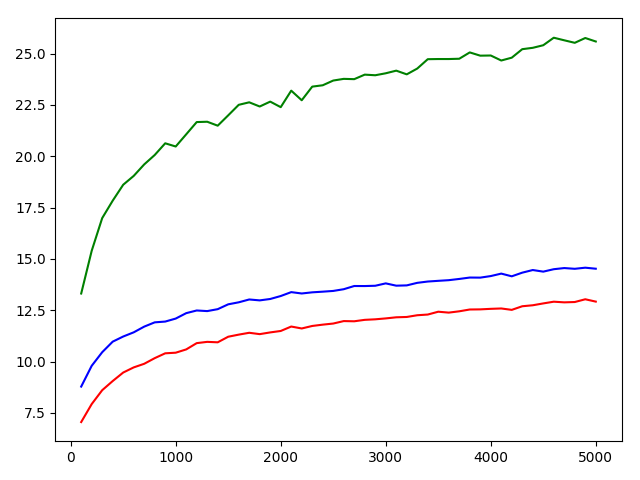
In this lab, each data structure’s performance is tested on insertion, search, deletion operations. Data set sizes from 100, 200 to 5000. Elements in data set are chosen randomly. The testing program runs around 50 times to get the average result. The results are numbers of operations instead of time.

Plots below show 3 curves with different colors. Blue one means one insertion operation, red one means one search operation and green one means one deletion operation. X axis is the number of nodes in the tree when searching, inserting or deleting. Y axis is the cost to do the operation.

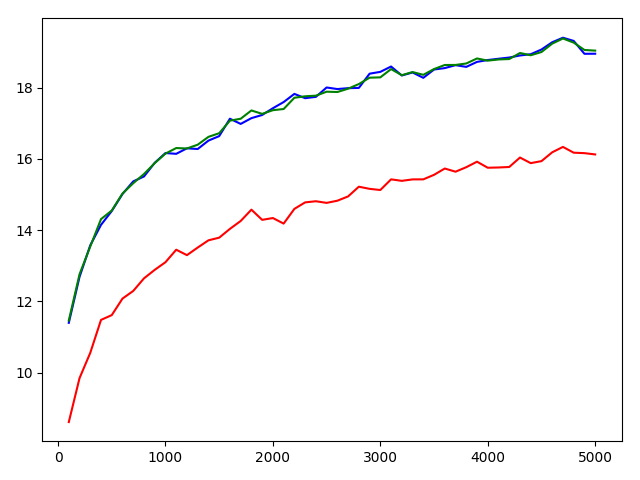
* 1. Complexity
     1. Standard binary tree.



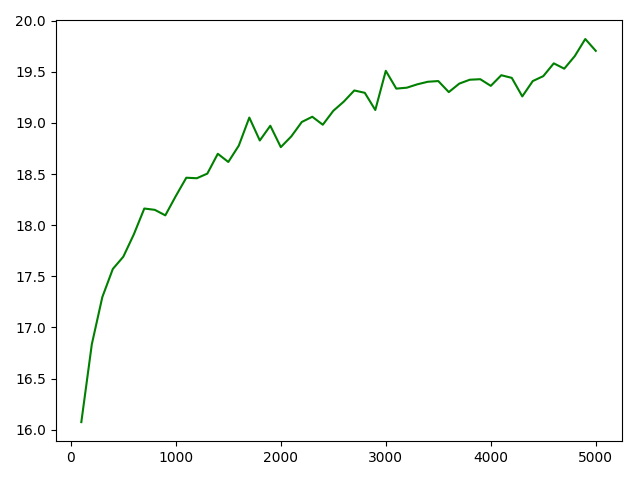
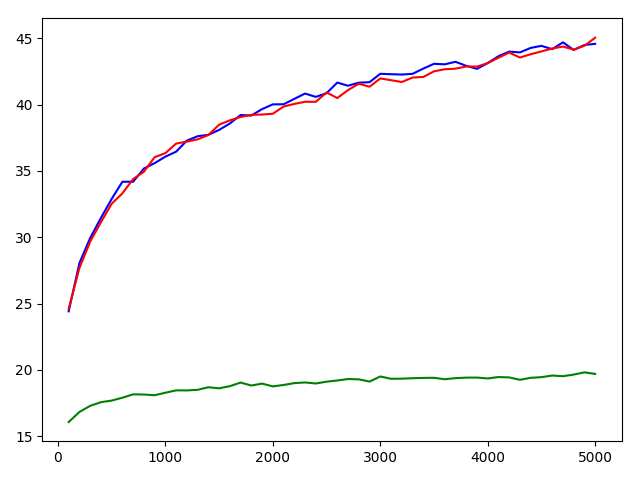
* + 1. AVL tree



* + 1. Treap



* + 1. Splay tree



In case that deletion operation curve is not obvious in left plot, I put the right plot here.

1. Conclusion

We can see from the plots in part 4 that the performance of each operation is O(log n). For standard binary tree, costs of three kinds of operations are similar. Deletion cost is slightly higher because it needs to find the maximum node in left subtree to replace the deleted one.

For AVL tree, search cost is low because it doesn’t need any rotation. Insertion cost is slightly higher because it needs at most 2 rotations. Deletion cost is much higher because it may need as many as O(log n) rotations.

For Treap, cost of search is lowest because it is just like the binary tree search. Costs of other two operations are similar because their expected number of rotations is at most 2.

For Splay tree, search and insertion costs are similar and they are higher than deletion cost. I think it is due to the order of sequence I use. It affects whether the node to delete is close to leaf or root. If it is close to the root, the cost is lower.