**Arman Vardanyan**

DSA - Assignment 1 – Search in dynamic sets

This documentation is about manipulations in dynamic sets. There are 6 parts

- Splay Tree implementation

- AVL Tree implementation

- Their comparison

- Separate chaining Hash Table

- Robin Hood Hash Table

- Their comparison

So let’s start from what files my project contains for first part

There are.

* Node.js wich is the object, its represents each element in the tree
* *public class* Node {  
   *private* Node parent;  
   *private* Node leftHeir;  
   *private* Node rightHeir;  
   *private int* heightvalue = 0;  
   *private int* value;

….

* MainTree.java file which contains main logic of Binary Tree.

Here are some function as for example.

*public class* MainTree {  
 *private* Node root;  
  
 *public* MainTree(*int* value) {  
 root = *new* Node(value);  
 } *public* Node getRoot() {  
 *return* root;  
 }  
  
 *public* Node search(*int* key) {,,,}

*public void* insert(*int* key) {,,,}

public void rotate(Node node) {,,,}

*public* Node restructure(Node a) {,,,}

*private void* printNode(Node node) {,,,}

etc…  
than I have AVLTree.java wich contains AVL Tree implementation

*public class* AVLTree *extends* MainTree{  
  
 *public* AVLTree(*int* value) {  
 *super*(value);  
 }  
  
 *// Returns the height of a given node in the AVLTree  
 public int* getHeight(Node node) {  
 *return* node == *null* ? 0 : node.getHeightvalue();  
 }  
  
 *// Recomputes the height of a given node in the AVLTree  
 public void* updateHeight(Node node) {  
 node.setHeightvalue(1 + Math.max(getHeight(node.getLeftHeir()), getHeight(node.getRightHeir())));  
 }  
  
 *// Checks if a given node in the AVLTree is balanced  
 public boolean* isNodeBalanced(Node node) {  
 *return* Math.abs(getHeight(node.getLeftHeir()) - getHeight(node.getRightHeir())) <= 1;  
 }  
  
 *// Returns the child node with the greater height of a given node in the AVLTree  
 public* Node getTallerChild(Node node) {  
 Node left = node.getLeftHeir();  
 Node right = node.getRightHeir();  
  
 *return* getHeight(left) > getHeight(right) ? left : right;  
 }

*// Rebalances a given node in the AVLTree by performing rotations as necessary  
 public void* rebalanceNode(Node node) {  
 *while* (node != *null*) {  
 updateHeight(node);  
 *if* (!isNodeBalanced(node)) {  
 node = restructure(getTallerChild(getTallerChild(node)));  
 updateHeight(node.getLeftHeir());  
 updateHeight(node.getRightHeir());  
 }  
 node = node.getParent();  
 }  
 }  
  
 *// Rebalances the AVLTree after an insertion  
 public void* rebalanceInsertion(Node node) {  
 rebalanceNode(node);  
 }  
  
 *// Rebalances the AVLTree after a deletion  
 public void* rebalanceDeletion(Node node) {  
 *if* (node != getRoot()) {  
 rebalanceNode(node.getParent());  
 }  
 }  
}

Also AVLTester.java

*public class* AVLTreeTester {  
 *public void* run() {  
 AVLTree tree = *new* AVLTree(10);  
 tree.insert(5);  
 tree.insert(3);  
 tree.insert(2);  
 tree.insert(1);  
 tree.insert(4);  
 tree.insert(11);  
 tree.insert(25);  
 tree.insert(15);  
 tree.inorder(tree.getRoot());  
 System.out.println();  
 tree.remove(25);  
 tree.inorder(tree.getRoot());  
 }  
}

and SplayTree.java

*public class* SplayTree *extends* MainTree {  
  
 *// Creates a new SplayTree instance with a root node containing the given value.  
 public* SplayTree(*int* value) {  
 *super*(value);  
 }  
  
 *// Splays the given node to the root of the tree.  
 private void* splay(Node node) {  
 *while* (node != getRoot()) {  
 Node parent = node.getParent();  
 Node grandparent = parent.getParent();  
  
 *if* (grandparent == *null*) {  
 rotate(node);  
 } *else if* (isZigZig(node, parent, grandparent)) {  
 rotate(parent);  
 rotate(node);  
 } *else* {  
 rotate(node);  
 rotate(node);  
 }  
 }  
 }  
  
 *//  
 // Determines if the given node is in a Zig-Zig case.  
 private boolean* isZigZig(Node node, Node parent, Node grandparent) {  
 *return* (parent == grandparent.getLeftHeir() && node == parent.getLeftHeir())  
 || (parent == grandparent.getRightHeir() && node == parent.getRightHeir());  
 }  
  
  
 *// Rebalances the tree after inserting a new node.  
 public void* rebalanceInsert(Node insertedNode) {  
 splay(insertedNode);  
 }  
  
 *// Rebalances the tree after accessing a node.  
 public void* rebalanceAccess(Node accessedNode) {  
 *if* (accessedNode != *null*) {  
 splay(accessedNode);  
 }  
 }  
  
 *// Rebalances the tree after deleting a node.  
 public void* rebalanceDelete(Node deletedNode) {  
 *if* (deletedNode != getRoot()) {  
 splay(deletedNode.getParent());  
 }  
 }  
}

and SplayTreeTester.java

*public class* SplayTreeTester {  
 *public void* run() {  
 SplayTree splayTree = *new* SplayTree(10);  
 splayTree.insert(7);  
 splayTree.insert(1);  
 splayTree.insert(2);  
 splayTree.insert(11);  
 splayTree.insert(6);  
 splayTree.insert(3);  
 splayTree.insert(25);  
 splayTree.insert(15);  
 splayTree.inorder(splayTree.getRoot());  
 System.out.println();  
 splayTree.remove(11);  
 splayTree.inorder(splayTree.getRoot());  
 System.out.println();  
 splayTree.search(6);  
 splayTree.inorder(splayTree.getRoot());  
 System.out.println();  
 splayTree.search(3);  
 splayTree.inorder(splayTree.getRoot());  
 }  
}

tester files are not similar but they are working very similar and can better provide their difference.

So, what is the difference between Splay Tree and AVL Tree. At first it was very difficult to understand what the difference was, except for the method, since with chaotic tests the results were + - the same. speed was sometimes more influenced by open programs in the computer than by the program itself, since the compiler always tries to do everything very quickly.

After in-depth study of the issue, I found out which method is better in which cases and began to test them with more problematic cases. They are both BVS-s with good performance, but shape of AVL Tree is always constrained, so that means that that the height of the tree never exceeds O(log n). This shape is better for deletions and insertions and not change during searches. In compare Splay trees can maintain efficient by reshaping the tree depending on problem. Splay trees are more memory efficient than AVL trees because they don't need to store balance information at the nodes. However, AVL trees are more useful in multi-threaded environments with a lot of searches because searching in AVL tree can be performed in parallel, while searches in Splay trees cannot. The difference is that in Splay trees, after each operation, we try to keep the tree almost perfectly balanced so that subsequent operations take less time. Also they are very similar that we are getting about Log(n) time in both Binary Trees.

So lets go to next part, to Hash Tables. I implemented to types of Hash Table algorythms, Separate chaining and Robin hood. So these are file

SeparateChainingHashTable.java

*import* java.util.Objects;  
  
*public class* SeparateChainingHashTable {  
  
 *// Array of HashNodes that represents the separate chains in the hash table  
 private* HashNode[] chains;  
 *// Capacity of the hash table  
 private* Integer capacity;  
 *private* Integer size; *// number of key-value pairs in hash table or number of hash nodes in a HashTable  
  
 // Default constructor with default capacity of 10  
 public* SeparateChainingHashTable() {  
 *this*(10); *// default capacity* }  
  
 *// Constructor that initializes the hash table with the given capacity  
 public* SeparateChainingHashTable(Integer capacity) {  
 *this*.capacity = capacity;  
 *this*.chains = *new* HashNode[capacity];  
 *this*.size = 0;  
 }  
  
 *// Inner class representing each node in the separate chains  
 private static class* HashNode {  
 *private* Integer key;  
 *private* String value;  
 *private* HashNode next; *// reference to next HashNode  
  
 public* HashNode(Integer key, String value) {  
 *this*.key = key;  
 *this*.value = value;  
 }  
 }  
  
 *// Returns the number of key-value pairs in the hash table  
 public* Integer size() {  
 *return* size;  
 }  
  
 *// Returns whether the hash table is empty or not  
 public boolean* isEmpty() {  
 *return* size == 0;  
 }  
  
 *// Adds a key-value pair to the hash table  
 public void* put(Integer key, String value) {  
 Objects.requireNonNull(key, "Key must not be null");  
 Objects.requireNonNull(value, "Value must not be null");  
  
 *int* bucketIndex = getBucketIndex(key);  
 HashNode head = chains[bucketIndex];  
 *while* (head != *null*) {  
 *if* (head.key.equals(key)) {  
 head.value = value;  
 *return*;  
 }  
 head = head.next;  
 }  
 size++;  
 HashNode newNode = *new* HashNode(key, value);  
 newNode.next = chains[bucketIndex];  
 chains[bucketIndex] = newNode;  
 }  
  
 *// Gets the index of the bucket in which the key should be stored  
 private* Integer getBucketIndex(Integer key) {  
 *return* key % capacity;  
 }  
  
 *// Gets the value associated with the given key  
 public* String get(Integer key) {  
 Objects.requireNonNull(key, "Key must not be null");  
  
 *int* bucketIndex = getBucketIndex(key);  
 *for* (HashNode currentNode = chains[bucketIndex]; currentNode != *null*; currentNode = currentNode.next) {  
 *if* (currentNode.key.equals(key)) {  
 *return* currentNode.value;  
 }  
 }  
  
 *return null*;  
 }  
  
 *// Removes the key-value pair associated with the given key  
  
  
 public* String remove(*final* Integer key) {  
 *if* (key == *null*) {  
 *throw new* IllegalArgumentException("Key must not be null!");  
 }  
  
 *final int* bucketIndex = getBucketIndex(key);  
 *final* HashNode head = chains[bucketIndex];  
 *if* (head == *null*) {  
 *return null*;  
 }  
 *if* (head.key.equals(key)) {  
 chains[bucketIndex] = head.next;  
 size--;  
 *return* head.value;  
 }  
 HashNode previous = head;  
 HashNode current = head.next;  
 *while* (current != *null*) {  
 *if* (current.key.equals(key)) {  
 previous.next = current.next;  
 size--;  
 *return* current.value;  
 }  
 previous = current;  
 current = current.next;  
 }  
 *return null*;  
 }  
}

HashTableSeparateTester.java

*public class* HashTableSeparateTester {  
 *public void* run () {  
 SeparateChainingHashTable table = *new* SeparateChainingHashTable(10);  
  
 table.put(1, "apple");  
 table.put(2, "banana");  
 table.put(3, "cherry");  
 table.put(4, "date");  
 System.out.println(table.size());  
 System.out.println(table.get(1)); *// apple* System.out.println(table.get(2)); *// banana* System.out.println(table.get(3)); *// cherry* System.out.println(table.get(4)); *// date* table.put(5, "apple");  
 System.out.println(table.remove(2));  
 System.out.println(table.get(2));  
 }  
}

RobinHoodHashTable.java

*import* java.util.Arrays;  
  
*public class* RobinHoodHashTable {  
 *private final int* size;  
 *private final* String[] keys;  
 *private final int*[] values;  
 *private final int*[] distances;  
  
 *public* RobinHoodHashTable(*int* size) {  
 *this*.size = size;  
 keys = *new* String[size];  
 values = *new int*[size];  
 distances = *new int*[size];  
 Arrays.fill(distances, -1);  
 }  
  
 *private int* hash(String key) {  
 *return* Math.abs(key.hashCode()) % size;  
 }  
  
 *public void* put(String key, *int* value) {  
 *int* index = hash(key);  
 *int* currentDistance = 0;  
  
 *while* (keys[index] != *null*) {  
 *if* (keys[index].equals(key)) {  
 values[index] = value;  
 *return*;  
 }  
  
 *int* distance = getAbsoluteDistance(index, hash(keys[index]));  
  
 *if* (distance < distances[index]) {  
 swap(index, distance);  
 currentDistance = distances[index];  
 }  
  
 index = (index + 1) % size;  
 currentDistance++;  
 }  
  
 keys[index] = key;  
 values[index] = value;  
 distances[index] = currentDistance;  
 }  
  
 *public* String remove(*int* value) {  
 *int* index = -1;  
  
 *// Find the index of the key-value pair with the given value  
 for* (*int* i = 0; i < size; i++) {  
 *if* (values[i] == value) {  
 index = i;  
 *break*;  
 }  
 }  
  
 *if* (index == -1) {  
 *// Key-value pair with the given value not found  
 return null*;  
 }  
  
 *// Found the key-value pair with the given value, remove it* String removedKey = keys[index];  
 keys[index] = *null*;  
 values[index] = 0;  
 distances[index] = -1;  
  
 *// Shift subsequent elements backward to fill the gap  
 int* nextIndex = (index + 1) % size;  
 *while* (keys[nextIndex] != *null* && distances[nextIndex] > 0) {  
 swap(nextIndex - 1, 1);  
 nextIndex = (nextIndex + 1) % size;  
 }  
  
 *// Return the name of the removed key-value pair  
 return* removedKey;  
 }  
  
 *// Function to calculate the absolute distance between two indices  
 private int* getAbsoluteDistance(*int* a, *int* b) {  
 *int* distance = Math.abs(b - a);  
 *return* (distance <= size / 2) ? distance : size - distance;  
 }  
  
 *// Function to swap the key-value pairs at two indices  
 private void* swap(*int* index, *int* distance) {  
 String tempKey = keys[index];  
 *int* tempValue = values[index];  
 *int* tempDistance = distances[index];  
  
 keys[index] = keys[(index + distance) % size];  
 values[index] = values[(index + distance) % size];  
 distances[index] = distances[(index + distance) % size];  
  
 keys[(index + distance) % size] = tempKey;  
 values[(index + distance) % size] = tempValue;  
 distances[(index + distance) % size] = tempDistance;  
 }  
  
 *public* String getKey(*int* index) {  
 *return* keys[index];  
 }  
  
  
 *public int* getValue(*int* index) {  
 *return* values[index];  
 }  
  
 *// Function to get the key corresponding to a given value  
 public* String get(*int* num) {  
 *for* (*int* i = 0; i < size; i++) {  
 *if* (values[i] == num) {  
 *return* keys[i];  
 }  
 }  
  
 *return null*;  
 }  
}

RobinHoodHashTableTester.java

*public void* run () {  
 RobinHoodHashTable table = *new* RobinHoodHashTable(10);  
  
 table.put("apple", 1);  
 table.put("banana", 2);  
 table.put("cherry", 3);  
 table.put("date", 4);  
 table.put("phone", 7);  
 table.put("bottle", 9);  
 table.put("knife", 10);  
 table.put("table", 11);  
 table.put("chair", 12);  
  
 System.out.println(table.get(1)); *// apple* System.out.println(table.get(2)); *// banana* System.out.println(table.get(3)); *// cherry* System.out.println(table.get(4)); *// date* table.put("apple", 5);  
 System.out.println();  
 System.out.println(table.remove(2)); *// banana* System.out.println(table.get(2)); *// null* System.out.println(table.get(5)); *// apple* System.out.println(table.get(8)); *// null* }  
}

So, here we have two Hash Tables and as it was in previous examples, here I also got asem problem, they were very similar in intelijIDEA. So I tried to understand what is difference between them except implementation. The Separate Chaining’s concept includes a method where each index key is built with a linked list. The Robin Hoods concept says that keys and values are stored in a contiguous array. One of main privilege of separate chaining is that it allows for efficient handling od collisions, as they can be resolved with simple linked list traversal. But the idea behind Robin Hoods strategy is to minimize the difference in spacing berween elements and their ideal indexes, which improves searching performance. To insert an element into Robin Hood hash table, the key is first hashed to find the corredsponding index in the array.

In general Separate Chaining is more memory efficient and works well when the number of collisions is relatively low. But the Robin Hood hashing is more computationally expensive, but can handle high collision rates more efficiently.