

Memory Management

SPlay Tree Application

Rakshit Kathawate | CST320 | 14/03/2019

# Splay tree:

## A splay tree is a self-adjusting binary search tree with the additional property that recently accessed elements are quick to access again.

## It performs basic operations such as insertion, look-up and removal in O(log n) amortized time.

## For many sequences of non-random operations, splay trees perform better than other search trees, even when the specific pattern of the sequence is unknown.

## The splay tree was invented by Daniel Sleator and Robert Tarjan in 1985.[1]

## Advantages:

Good performance for a splay tree depends on the fact that it is self-optimizing, in that frequently accessed nodes will move nearer to the root where they can be accessed more quickly. The worst-case height is **O(n),** with the avg being **O (log *n*).**

## DisAdvantages:

The most significant disadvantage of splay trees is that the height of a splay tree can be linear.

## Problem definition:

Design a Splay tree to perform memory management for variables used in the “MAIN” and 3 other functions associated with MAIN function.

## Technology stack:

* NetBeans IDE 8.2
* JDK 1.7

**Function definitions**

Function Splay (Node node, key k):

Given Function splays the tree rooted at Node node for the key K. If the node with the key K exists in the tree, it is splayed(moved) to the root node, else the last node along the search path for key K is splayed to the root.

Function Insert (Node node, key k, Value v):

Given Function splays the Key K in the tree using Splay procedure as described above. If the key is already found in the tree (at root node) then the simply the root value is left unchanged. If the key K is not found then the new root is created with value K, and is assigned as the root of tree with the proper settings of the pointers.

Function get (Key k):

Given function returns the splay the tree for the key, move it to the ROOT and returns the key if the key exists in tree, else it returns the NULL.

Function driverFunction ():

Given Function is the entry point for the code and constitutes the logic for the memory management. Every time we encounter the new variable in the program, we insert the variable along with the scope of variable as the KEY and value of variable as VALUE. If the variable encountered already exists in the tree, GET procedure is executed to search and splay the variable to the ROOT of tree.

**Node Structure**

Node {

Int key; -> Stores the scope of the Variable

Int value; -> Stores the value of the Variable

Node left, right; -> Store the Left and Right child Pointers of the Node

}

**Key Functionality:**

Given project demonstrates the GCC compiler memory management operation and fast access method for recently used variables with the help of Splay tree data structure. Every node in tree represents the variable in the program, with the KEY, VALUE pair. KEY -> Scope of the variable in the program. VALUE -> Value of the variable in the program.

Parser reads one line at instance, it identifies the token and check the scope of it. With (Key, Value) pair formed using the above explanation it searches in the tree if the key already exists, if YES then the key is splayed to root for the further efficient search, else the key is inserted in the tree using the INSERT procedure and it is splayed using SPLAY procedure.

Nested function calls are also demonstrated, in the project to better understand the scope of variables. This project is the Abstract Overview of the complex memory management performed by compiler.

**Code Section:**

package splay\_trees;

import java.io.IOException;

/\*\*

\*

\* **@author champion**

\*/

class SplayBST<Key extends Comparable<Key>, Value> {

public Node root; // root of the BST

// BST helper node data type

public class Node {

public Key key; // key

public Value value; // associated data

public Node left, right; // left and right subtrees

public Node(Key key, Value value) {

this.key = key;

this.value = value;

}

}

public boolean contains(Key key) {

return get(key) != null;

}

// return value associated with the given key

// if no such value, return null

public Value get(Key key) {

root = splay(root, key);

int cmp = key.compareTo(root.key);

if (cmp == 0) return root.value;

else return null;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Splay tree insertion.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public void put(Key key, Value value) {

// splay key to root

if (root == null) {

root = new Node(key, value);

return;

}

// splay for the key,

root = splay(root, key);

int cmp = key.compareTo(root.key);

// Insert new node at root

if (cmp < 0) {

Node n = new Node(key, value);

n.left = root.left;

n.right = root;

root.left = null;

root = n;

}

// Insert new node at root

else if (cmp > 0) {

Node n = new Node(key, value);

n.right = root.right;

n.left = root;

root.right = null;

root = n;

}

// It was a duplicate key. Simply replace the value

else {

root.value = value;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Splay tree deletion.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* This splays the key, then does a slightly modified Hibbard deletion on

\* the root (if it is the node to be deleted; if it is not, the key was

\* not in the tree). The modification is that rather than swapping the

\* root (call it node A) with its successor, it's successor (call it Node B)

\* is moved to the root position by splaying for the deletion key in A's

\* right subtree. Finally, A's right child is made the new root's right

\* child.

\*/

public void remove(Key key) {

if (root == null) return; // empty tree

root = splay(root, key);

int cmp = key.compareTo(root.key);

if (cmp == 0) {

if (root.left == null) {

root = root.right;

}

else {

Node x = root.right;

root = root.left;

splay(root, key);

root.right = x;

}

}

// else: it wasn't in the tree to remove

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Splay tree function.

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// splay key in the tree rooted at Node h. If a node with that key exists,

// it is splayed to the root of the tree. If it does not, the last node

// along the search path for the key is splayed to the root.

private Node splay(Node h, Key key) {

if (h == null) return null;

int cmp1 = key.compareTo(h.key);

if (cmp1 < 0) {

if (h.left == null) {

return h;

}

int cmp2 = key.compareTo(h.left.key);

if (cmp2 < 0) {

h.left.left = splay(h.left.left, key);

h = rotateRight(h);

}

else if (cmp2 > 0) {

h.left.right = splay(h.left.right, key);

if (h.left.right != null)

h.left = rotateLeft(h.left);

}

if (h.left == null) return h;

else return rotateRight(h);

}

else if (cmp1 > 0) {

if (h.right == null) {

return h;

}

int cmp2 = key.compareTo(h.right.key);

if (cmp2 < 0) {

h.right.left = splay(h.right.left, key);

if (h.right.left != null)

h.right = rotateRight(h.right);

}

else if (cmp2 > 0) {

h.right.right = splay(h.right.right, key);

h = rotateLeft(h);

}

if (h.right == null) return h;

else return rotateLeft(h);

}

else return h;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Helper functions.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// height of tree (1-node tree has height 0)

// public int height() { return height(root); }

public int height(Node x) {

if (x == null) return 0;

return 1 + Math.max(height(x.left), height(x.right));

}

public int size() {

return size(root);

}

private int size(Node x) {

if (x == null) return 0;

else return 1 + size(x.left) + size(x.right);

}

// right rotate

private Node rotateRight(Node h) {

Node x = h.left;

h.left = x.right;

x.right = h;

return x;

}

// left rotate

private Node rotateLeft(Node h) {

Node x = h.right;

h.right = x.left;

x.left = h;

return x;

}

// test client

public void inOrder(Node h){

if(h.left!=null){

inOrder(h.left);

}

System.out.println(h.key+" , "+h.value);

if(h.right!=null){

inOrder(h.right);

}

}

public void preOrder(Node h){

System.out.println(h.key+" , "+h.value);

if(h.left!=null){

preOrder(h.left);

}

if(h.right!=null){

preOrder(h.right);

}

}

void printGivenLevel (Node hroot ,int level)

{

if (hroot == null)

return;

if (level == 1) {

System.out.print(" ("+hroot.key+","+hroot.value+") ");

}

else if (level > 1)

{

printGivenLevel(hroot.left, level-1);

printGivenLevel(hroot.right, level-1);

}

}

void printLevelOrder(Node hroot)

{

int h = height(hroot);

//System.out.println("Height of tree: "+h);

int i;

for (i=1; i<=h; i++) {

printGivenLevel(hroot, i);

System.out.println("\n");

}

}

public void statusOfTree(SplayBST<Key,Value>st,String s){

SplayBST.Node r1=st.root;

System.out.println(" \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

System.out.println("Operation: "+s);

System.out.println("----------------- Root of Tree: ------------------");

System.out.println("ROOT -> "+"("+root.key+" ,"+root.value+")");

System.out.println("-----------------------------------------------------------------------");

System.out.println("----------------- Inorder of Tree: ------------------");

inOrder(r1);

System.out.println("-----------------------------------------------------------------------");

System.out.println("----------------- Preorder of Tree: ------------------");

preOrder(r1);

System.out.println("-----------------------------------------------------------------------");

// System.out.println("----------------- Levelorder of Tree: ------------------");

// printLevelOrder(r1);

// System.out.println("\n-----------------------------------");

System.out.println(" \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

}

}

public class Splay\_Trees {

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Driver function

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public static void main(String[] args) throws IOException {

StringBuilder str=new StringBuilder();

str.append("Program Input : \n \n");

str.append("main: \n");

str.append("\t int x; \n");

str.append("\t int y; \n");

str.append("\t fun1(x) \n");

str.append("\t fun2(y) \n");

str.append("\t int c\n");

str.append("\t x=c \n");

str.append("\t z=x+y\n");

str.append("\t print(z)\n");

str.append("\n");

str.append("fun1(x):\n");

str.append("\t y=x+2\n");

str.append("\n");

str.append("fun2(y)\n");

str.append("\t y=2 \n");

str.append("\t fun3(y)\n");

str.append(" \n");

str.append("fun3(y)\n");

str.append("\t y=5 \n");

String programText=str.toString();

System.out.println("\n");

System.out.println("/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/");

System.out.println("\t /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ \t ");

System.out.println(programText);

System.out.println("/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/");

System.out.println("\t /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ \t ");

System.out.println("\n");

/\*

DEMO PROGRAM FOR EXECUTION:

**main:**

**int x;**

**int y;**

**fun1(x);**

**fun2(y);**

**int c;**

**x=c**

**z=x+y;**

**printf("%d",&z);**

**fun1(x):**

**y=x+2;**

**fun2(y):**

**y=2;**

**fun3(y);**

**fun3(y):**

**y=5;**

\*/

SplayBST<String, String> st = new SplayBST<String, String>();

st.put("mainX", "x");

st.statusOfTree(st," Insertion of mainX");

st.put("mainY", "y");

st.statusOfTree(st," Insertion of mainY");

st.get("mainX");

// st.put("mainX", "x");

st.statusOfTree(st, "After Searching for mainX");

st.put("fun1X", "x");

st.statusOfTree(st," Insertion of fun1X");

st.put("fun1Y", "y");

st.statusOfTree(st," Insertion of fun1Y");

st.get("mainY");

// st.put("mainY", "y");

st.statusOfTree(st, "After Searching mainY");

st.put("fun2Y", "y");

st.statusOfTree(st," Insertion of fun2Y");

st.get("fun2Y");

st.statusOfTree(st, "After Searching fun2Y");

st.put("fun3Y","y");

st.statusOfTree(st, "Insertion of fun3Y");

st.put("mainC", "c");

st.statusOfTree(st, "Insetion of mainC");

st.get("mainX");

st.statusOfTree(st, "After Searching for mainX");

st.get("mainX");

// st.put("mainX", "x");

st.statusOfTree(st, "After Searching for mainX");

st.get("mainY");

//st.put("mainY","y");

st.statusOfTree(st, "After Searching for mainY");

st.put("mainZ", "z");

st.statusOfTree(st," Insertion of mainZ");

st.get("mainZ");

// st.put("mainZ", "z");

st.statusOfTree(st,"After Searching for mainZ");

}

}