CS F211

Data Structures and Algorithms

Practice Problems

Hashmaps, AVL Trees, and Red-Black Trees

Problem 1

Use hashmaps and prefix sums to write a program that, given an array of integers and a target sum, finds the number of subarrays of the array such that its elements sum up to the target.

```cpp
#include <iostream>
#include <vector>
#include <list>
#include <string>

using namespace std;

class HashMap {
private:
    static const int TABLE_SIZE = 10;
    vector< list< pair< int, int > > > table;

    int hashFunction(const int &key) {
        hash<int> hashFunc;
        return hashFunc(key) % TABLE_SIZE;
    }

public:
    HashMap() {
        table.resize(TABLE_SIZE);
    }

    void set(const int &key, int value) {
        int index = hashFunction(key);
        for (auto &p : table[index]) {
            if (p.first == key) {
                p.second = value;
                return;
            }
        }
        table[index].emplace_back(key, value);
    }

    int get(const int &key) {
        int index = hashFunction(key);
        for (auto &p : table[index]) {
            if (p.first == key)
                return p.second;
```

```
        }
        return 0;
    }
};

int main() {
    /* ??? */
}
```

```
>> 8 7
>> 3 4 7 2 -3 1 4 2
4
```

Next task: Modify the code to output one such subarray with the least length.

Problem 2

Write code that behaves as an interpreter for a simple programming language. In this simple language, each line is an assignment consisting of a variable on the LHS and a postfix expression on the RHS which contains both integers and previously defined variables. Use stacks for evaluating expressions, and a hashmap (which we call symbol table) for storing the values assigned to variables.

```
#include <iostream>
#include <vector>
#include <list>
#include <string>
#include <stack>

using namespace std;

class HashMap {
private:
    static const int TABLE_SIZE = 10;
    vector< list< pair< string, int > > > table;

    int hashFunction(const string &key) {
        hash<string> hashFunc;
        return hashFunc(key) % TABLE_SIZE;
    }

public:
    HashMap() {
        table.resize(TABLE_SIZE);
    }

    void set(const string &key, int value) {
        int index = hashFunction(key);
        for (auto &p : table[index]) {
            if (p.first == key) {
                p.second = value;
                return;
            }
        }
        table[index].emplace_back(key, value);
    }

```

```
    int get(const string &key) {
        int index = hashFunction(key);
        for (auto &p : table[index]) {
            if (p.first == key)
                return p.second;
        }
        return 0;
    }
};

int evaluate(string expr, HashMap* m) {
    /* ??? */
}

void assign(string token, string expr, HashMap* m) {
    int val = evaluate(expr, m);
    m->set(token, val);
    cout << token << " = " << val << '\n';
}

int main() {
    HashMap m;;
    assign("a", "5 3 +", &m);
    assign("b", "a 5 * 8 -", &m);
    assign("result", "b a /", &m);
}
```

```
a = 8
b = 32
result = 4
```

Next task: Use the BT expression evaluating code you have written in a previous problem sheet to allow the program to instead evaluate regular arithmetic expressions.

Problem 3

You are given a list of n integers and a target sum. Write a program to find if there exists a subset of the integers such that they sum up to the target. The naive way is to explore all 2^n possibilities, but a more efficient way involves first dividing the array into two halves, finding the two sets of $2^{\frac{n}{2}}$ possible sums for either half, and then checking if there exists a way to add a sum from the first half and a sum from the second half to obtain the target sum.

The recursive function that generates all possible sums for a half stores the sums in a set implemented using an AVL tree. This automatically handles duplication and ensures that all possible sums can be retrieved in the end in sorted order, which is important for the merge step. Use two pointers on the two sorted arrays of possible sums to efficiently check if the target sum is obtainable from two sums belonging to the two different halves.

```
#include <iostream>
#include <vector>

using namespace std;

struct Node {
    int key, height;
    Node *left, *right;

    Node(int k) : key(k), height(1), left(nullptr), right(
        nullptr) {}
};

class AVLTree {
private:
    Node* root;

    int getHeight(Node* node) {
        return node ? node->height : 0;
    }

    int getBalance(Node* node) {
        return node ? getHeight(node->left) - getHeight(node
            ->right) : 0;
    }

    Node* rotateRight(Node* y) {
```

```
26          Node* x = y->left;
27          Node* T2 = x->right;
28          x->right = y;
29          y->left = T2;
30          y->height = max(getHeight(y->left), getHeight(y->
                right)) + 1;
31          x->height = max(getHeight(x->left), getHeight(x->
                right)) + 1;
32          return x;
33      }
34
35      Node* rotateLeft(Node* x) {
36          Node* y = x->right;
37          Node* T2 = y->left;
38          y->left = x;
39          x->right = T2;
40          x->height = max(getHeight(x->left), getHeight(x->
                right)) + 1;
41          y->height = max(getHeight(y->left), getHeight(y->
                right)) + 1;
42          return y;
43      }
44
45      Node* insert(Node* node, int key) {
46          if (!node) return new Node(key);
47
48          if (key < node->key) node->left = insert(node->left,
                key);
49          else if (key > node->key) node->right = insert(node->
                right, key);
50          else return node;
51
52          node->height = max(getHeight(node->left), getHeight(
                node->right)) + 1;
53          int balance = getBalance(node);
54
55          if (balance > 1 && key < node->left->key) return
                rotateRight(node);
56          if (balance < -1 && key > node->right->key) return
                rotateLeft(node);
57          if (balance > 1 && key > node->left->key) {
58              node->left = rotateLeft(node->left);
59              return rotateRight(node);
60          }
61          if (balance < -1 && key < node->right->key) {
```

```
62              node->right = rotateRight(node->right);
63              return rotateLeft(node);
64          }
65
66          return node;
67      }
68
69      void inorder(Node* node, vector<int>& v) {
70          if (!node) return;
71          inorder(node->left, v);
72          v.push_back(node->key);
73          inorder(node->right, v);
74      }
75
76  public:
77      AVLTree() : root(nullptr) {}
78
79      void insert(int key) {
80          root = insert(root, key);
81      }
82
83      void getElems(vector<int>& v) {
84          inorder(root, v);
85      }
86  };
87
88  void genSumsRec(int idx, int sum, vector<int>& v, AVLTree* s)
        {
89      /* ??? */
90  }
91
92  void genSums(vector<int>& v, AVLTree* s) {
93      /* ??? */
94  }
95
96  int main() {
97      int n, sum;
98      cin >> n >> sum;
99      vector<int> v1, v2;
100     for (int i = 0; i < n; i++) {
101         int a;
102         cin >> a;
103         if ((i % 2) == 0) v1.push_back(a);
104         else v2.push_back(a);
105     }
```

```
    AVLTree* s1 = new AVLTree(), *s2 = new AVLTree();
    genSums(v1, s1);
    genSums(v2, s2);
    vector<int> b1, b2;
    s1->getElems(b1), s2->getElems(b2);
    /* ??? */
    return 0;
}
```

```
>> 7 16
>> 1 3 4 2 5 6 8
Yes
```

Next task: As is clear in the given code, there is a method that converts the elements of the AVLT into a vector, and this vector is then later used in the merge step. We could simplify this by instead implementing direct methods in the AVLT to retrieve next and previous elements starting from a particular state. Implement these methods. Of course, the methods only work if each node has a pointer to its parent, so add a parent attribute and modify the other methods to handle changes to parent pointers.

Problem 4

Modify the AVLT code given in the previous problem to let the tree find the median of its elements in $O(\log n)$. Formally, your code must handle two types of queries, which are described below.

1. `1 x` – add integer x to the set.
2. `2` – return the median of all the integers in the tree.

To achieve this, as a hint, consider adding a size attribute to node that keeps track of size of the subtree rooted at that node.

```
>> 5
>> 1 6
>> 1 2
>> 2
2
>> 1 9
>> 1 4
>> 1 5
>> 2
5
```

Next task: Modify the code to answer general kth-smallest integer queries.

Problem 5

Red-black trees are special binary search trees which are capable of balancing themselves, like AVL trees. In this problem, you have to examine the efficiency of RBTs using a metric called *balance*, which we define as the sum of differences of the heights of left and right subtrees of all nodes. Add a recursive method to the RBT that calculates this metric.

```
#include <iostream>
#include <string>
#include <cstdlib>
#include <ctime>

using namespace std;

enum Color { RED, BLACK };

struct Node {
    int data;
    Color color;
    Node *left, *right, *parent;
    int height, balance;

    Node(int data) {
        this->data = data;
        left = right = parent = nullptr;
        color = RED;
    }
};

class RedBlackTree {
private:
    Node* root;

    void rotateLeft(Node* &root, Node* &pt) {
        Node* rightChild = pt->right;
        pt->right = rightChild->left;

        if (pt->right)
            pt->right->parent = pt;

        rightChild->parent = pt->parent;

```

```
36          if (!pt->parent)
37              root = rightChild;
38          else if (pt == pt->parent->left)
39              pt->parent->left = rightChild;
40          else
41              pt->parent->right = rightChild;
42
43          rightChild->left = pt;
44          pt->parent = rightChild;
45      }
46
47      void rotateRight(Node* &root, Node* &pt) {
48          Node* leftChild = pt->left;
49          pt->left = leftChild->right;
50
51          if (pt->left)
52              pt->left->parent = pt;
53
54          leftChild->parent = pt->parent;
55
56          if (!pt->parent)
57              root = leftChild;
58          else if (pt == pt->parent->left)
59              pt->parent->left = leftChild;
60          else
61              pt->parent->right = leftChild;
62
63          leftChild->right = pt;
64          pt->parent = leftChild;
65      }
66
67      void fixViolation(Node* &root, Node* &pt) {
68          Node* parent = nullptr;
69          Node* grandparent = nullptr;
70
71          while ((pt != root) && (pt->color != BLACK) && (pt->
                parent->color == RED)) {
72              parent = pt->parent;
73              grandparent = parent->parent;
74
75              if (parent == grandparent->left) {
76                  Node* uncle = grandparent->right;
77
78                  if (uncle && uncle->color == RED) {
79                      grandparent->color = RED;
```

```
80                      parent->color = BLACK;
81                      uncle->color = BLACK;
82                      pt = grandparent;
83                  } else {
84                      if (pt == parent->right) {
85                          rotateLeft(root, parent);
86                          pt = parent;
87                          parent = pt->parent;
88                      }
89
90                      rotateRight(root, grandparent);
91                      swap(parent->color, grandparent->color);
92                      pt = parent;
93                  }
94              } else {
95                  Node* uncle = grandparent->left;
96
97                  if (uncle && uncle->color == RED) {
98                      grandparent->color = RED;
99                      parent->color = BLACK;
100                     uncle->color = BLACK;
101                     pt = grandparent;
102                 } else {
103                     if (pt == parent->left) {
104                         rotateRight(root, parent);
105                         pt = parent;
106                         parent = pt->parent;
107                     }
108
109                     rotateLeft(root, grandparent);
110                     swap(parent->color, grandparent->color);
111                     pt = parent;
112                 }
113             }
114         }
115
116         root->color = BLACK;
117     }
118
119 public:
120     RedBlackTree() {
121         root = nullptr;
122     }
123
124     void insert(const int &data) {
```

```
125         Node* newNode = new Node(data);
126 
127         Node* parent = nullptr;
128         Node* current = root;
129 
130         while (current) {
131             parent = current;
132             if (newNode->data < current->data)
133                 current = current->left;
134             else
135                 current = current->right;
136         }
137 
138         newNode->parent = parent;
139 
140         if (!parent)
141             root = newNode;
142         else if (newNode->data < parent->data)
143             parent->left = newNode;
144         else
145             parent->right = newNode;
146 
147         fixViolation(root, newNode);
148     }
149 
150     void inorderRec(Node* root) {
151         /* ??? */
152     }
153 
154     void inorder() {
155         inorderRec(root);
156         cout << endl;
157     }
158 
159     void preorderRec(Node* root) {
160         /* ??? */
161     }
162 
163     void preorder() {
164         preorderRec(root);
165         cout << endl;
166     }
167 
168     void getBalanceRec(Node* root) {
169         /* ??? */
```

```
    }

    int getBalance() {
        /* ??? */
    }
};

class Random {
private:
    int a, c, m, x;

public:
    Random(int _m) {
        a = 1664525;
        c = 1013904223;
        m = _m;
        x = static_cast<int>(time(0));
    }

    int generate() {
        x = (int)(((long long)a * (long long)x + c) % m);
        return x + 1;
    }
};

int main() {
    RedBlackTree rbt;
    rbt.insert(10);
    rbt.insert(3);
    rbt.insert(7);
    rbt.insert(2);
    rbt.insert(8);
    rbt.insert(1);
    rbt.inorder();
    rbt.preorder();
    cout << rbt.getBalance() << '\n';
    return 0;
}
```

```
1 2 3 7 8 10
7 2 1 3 10 8
1
```

Next task: Use the random generation class given in the code to generate a

large number of random values (like 1000 or 10000) to insert into the RBT. Insert these values into the tree and calculate the balance at regular intervals of insertion. Examine how the balance of the tree changes with size. Try the same experiment on other kinds of binary tree.