Міністерство освіти і науки України

Національний Технічний Університет

«Харківський Політехнічний Інститут»

Кафедра «Стратегічного Керування»

Лабораторна робота № 3

«ЧЕРВОНО-ЧОРНІ ДЕРЕВА»

Перевірила: Мошко Є. О.

Виконав:

Харків, 2018

Лабораторна робота № 3

Тема: ЧЕРВОНО-ЧОРНІ ДЕРЕВА

Завдання

Розробити програму, яка читає з клавіатури числа N, M (1 < N, M < 256); послідовність N ключів (цілих, дійсних чисел або рядків (до 255 символів) в залежності від варіанту завдання); послідовність M ключів. Програма зберігає першу послідовність до червоно-чорного дерева.

Кожного разу, коли до дерева додається новий елемент, потрібно вивести статистику (згідно варіанту завдання).

*1 Мінімальний елемент та його колір;*

Після побудови дерева для кожного елементу x другої послідовності потрібно вивести результати наступних операцій над деревом (згідно варіанту завдання).

*2 Successor(x) та його колір.*

Хід роботи

|  |
| --- |
| #include <iostream>  #include <string>  using namespace std;  enum color {RED, BLACK};  struct Node{  int data;  bool color;  Node \*left, \*right, \*parent;    Node(int data){  this->data = data;  left = right = parent = NULL;  }  };  class RBTree{  private:  Node \*root;  public:  RBTree() : root(NULL){}  void insert(const int &data){  Node\* pt = new Node(data);  root = BSTinsert(root,pt);  fixViolation(root, pt);  }  string min(){  Node \*pt = root;  while (pt->left) pt = pt->left;  return "min: "+nodeInfo(pt)+"\n";  }  void Dump(){  Dump(root, 0);  }  void Dump(Node \*pt, int tabs){  if (!pt) return;  Dump(pt->right, tabs + 1);  for (int i = 0; i < tabs; ++i) cout << "\t\t";  cout << nodeInfo(pt) << endl;  Dump(pt->left, tabs + 1);  }  string successor(int m){  string str;  Node \*pt = root;  while(pt){  if (pt->data<m){  pt=pt->right;  }  else if(pt->data>m){  pt=pt->left;  }  else{  break;  }  }  if (pt){  if ((pt->left)!=NULL){  str = "Left child: " + nodeInfo(pt->left) +"\n";  }  if ((pt->right)!=NULL){  str+= "Right child: " + nodeInfo(pt->right) + "\n";  }  if (str=="") str= "Doesn't have a child\n";  }  else  {  str = "Not found\n";  }  return str;  }  string nodeInfo(Node\* pt){  return ((pt->color)?"B\_":"R\_")+to\_string(pt->data);  }  Node\* BSTinsert(Node \*root, Node \*pt){  if (root==NULL) {  return pt;  }  else if (pt->data<root->data){  root->left=BSTinsert(root->left, pt);  root->left->parent=root;  }  else if (pt->data>root->data){  root->right=BSTinsert(root->right, pt);  root->right->parent=root;  }  return root;  }  void rotateLeft(Node \*& root, Node \*& pt){  Node \*pt\_right = pt->right;    pt->right = pt\_right->left;    if (pt->right != NULL)  pt->right->parent = pt;    pt\_right->parent = pt->parent;    if (pt->parent == NULL)  root = pt\_right;    else if (pt == pt->parent->left)  pt->parent->left = pt\_right;    else  pt->parent->right = pt\_right;    pt\_right->left = pt;  pt->parent = pt\_right;  }  void rotateRight(Node \*&root, Node \*&pt){  Node \*pt\_left = pt->left;    pt->left = pt\_left->right;    if (pt->left != NULL)  pt->left->parent = pt;    pt\_left->parent = pt->parent;    if (pt->parent == NULL)  root = pt\_left;    else if (pt == pt->parent->left)  pt->parent->left = pt\_left;    else  pt->parent->right = pt\_left;    pt\_left->right = pt;  pt->parent = pt\_left;  }  void fixViolation(Node \*& root, Node \*& pt){ // passing a pointer by referece  Node \*parent\_pt = NULL;  Node \*grand\_parent\_pt = NULL;  while ((pt!=root)&&(pt->parent->color==RED)){ //&&(pt->color!=BLACK)  parent\_pt=pt->parent;  grand\_parent\_pt=pt->parent->parent;  //Case A  //Parent of pt is left child of its grandparent    if (parent\_pt==grand\_parent\_pt->left){  Node\* uncle\_pt = grand\_parent\_pt->right;    //Case 1  //The uncle of pt is also red    if ((uncle\_pt!=NULL)&&(uncle\_pt->color==RED)){  grand\_parent\_pt->color = RED;  parent\_pt->color = BLACK;  uncle\_pt->color = BLACK;  pt = grand\_parent\_pt;  }  else{  //Case 2  //pt is right child of its parent  //Left-Rotation required  if (pt==parent\_pt->right){  rotateLeft(root,parent\_pt);  pt=parent\_pt;  parent\_pt = pt->parent;  }  rotateRight(root, grand\_parent\_pt);  swap(parent\_pt->color, grand\_parent\_pt->color);  pt = parent\_pt;  }  }  else  {  Node \*uncle\_pt = grand\_parent\_pt->left;    /\* Case : 1  The uncle of pt is also red  Only Recoloring required \*/  if ((uncle\_pt != NULL) && (uncle\_pt->color == RED))  {  grand\_parent\_pt->color = RED;  parent\_pt->color = BLACK;  uncle\_pt->color = BLACK;  pt = grand\_parent\_pt;  }  else  {  /\* Case : 2  pt is left child of its parent  Right-rotation required \*/  if (pt == parent\_pt->left)  {  rotateRight(root, parent\_pt);  pt = parent\_pt;  parent\_pt = pt->parent;  }    /\* Case : 3  pt is right child of its parent  Left-rotation required \*/  rotateLeft(root, grand\_parent\_pt);  swap(parent\_pt->color, grand\_parent\_pt->color);  pt = parent\_pt;  }  }  }  root->color=BLACK;  }  };  int main(){  RBTree tree;  int N,M;  int n,m;  cout << "Enter amount of keys and amount of lookups: " << endl;  cin >> N >> M;  cout << "Enter " << N << " keys for insertion: " << endl;  for (int i=0; i<N; i++){  cin >> n;  tree.insert(n);  cout << tree.min();  }  tree.Dump();  cout << "Enter " << M << " keys for lookups: " << endl;  for (int i=0;i<M;i++){  cin >> m;  cout << tree.successor(m);  }  system("pause");  } |

Приклад виконання програми:

|  |
| --- |
| Enter amount of keys and amount of lookups:  5 4  Enter 5 keys for insertion:  54  min: B\_54  23  min: R\_23  9  min: R\_9  -54  min: R\_-54  2  min: R\_-54  B\_54  B\_23  R\_9  B\_2  R\_-54  Enter 4 keys for lookups:  23  Left child: B\_2  Right child: B\_54  54  Doesn't have a child  2  Left child: R\_-54  Right child: R\_9  3  Not found  Press any key to continue . . . |

Висновок

Червоно-чорне дерево (red-black tree) -це двійкове дерево пошуку, вершини якого розділені на червоні (red) і чорні (black). Таким чином, кожна вершина зберігає один додатковий біт -її колір. Додавання кольорів дозволило алгоритму (за допомогою подальших операцій змінення дерева – перекручення, перефарбування) не допустити двом піддеревам різнитися більш ніж удвічі. Такий алгоритм побудування дерева (алгоритм само балансування) дозволяє скоротити час пошуку до O(log n).