Структири даних та алгоритми

Кляцко Семен Михайлович ТК-31

Завдання 1

Бінарне дерево пошуку. Для заданого масиву ключів (більше 15 значень, задати випадково – цілі числа з множини [0, 100]) побудувати бінарне дерево пошуку, реалізувати всі варіанти обходів (прямий, обернений, симетричний). Вивести побудоване дерево і результати обходів.

Вивід

```
Enter array length: 20
Preorder print:
426,94,113,189,253,273,319,368,539,574,587,652,653,751,771,804,812,889,974,987,
Inorder print:
94,113,189,253,273,319,368,426,539,574,587,652,653,751,771,804,812,889,974,987,
Postorder print:
94,113,189,253,273,319,368,539,574,587,652,653,751,771,804,812,889,974,987,426,
```

Програмний код

```
struct node{
  int value;
  node *left;
  node *right;
};

class btree{
public:
  btree();
  ~btree();
```

```cpp
  void insert(int key);
  node *search(int key);
  void destroy_tree();
  void inorder_print();
  void postorder_print();
  void preorder_print();

private:
  void destroy_tree(node *leaf);
  void insert(int key, node *leaf);
  node *search(int key, node *leaf);
  void inorder_print(node *leaf);
  void postorder_print(node *leaf);
  void preorder_print(node *leaf);

  node *root;
};


btree::btree(){
  root = NULL;
}

btree::~btree(){
  destroy_tree();
}

void btree::destroy_tree(node *leaf){
  if(leaf != NULL){
     destroy_tree(leaf->left);
     destroy_tree(leaf->right);
     delete leaf;
  }
}

void btree::insert(int key, node *leaf){

  if(key < leaf->value){
     if(leaf->left != NULL){
        insert(key, leaf->left);
     }else{
        leaf->left = new node;
        leaf->left->value = key;
        leaf->left->left = NULL;
```

```
            leaf->left->right = NULL;
        }
    }else if(key >= leaf->value){
        if(leaf->right != NULL){
            insert(key, leaf->right);
        }else{
            leaf->right = new node;
            leaf->right->value = key;
            leaf->right->right = NULL;
            leaf->right->left = NULL;
        }
    }

}

void btree::insert(int key){
    if(root != NULL){
        insert(key, root);
    }else{
        root = new node;
        root->value = key;
        root->left = NULL;
        root->right = NULL;
    }
}

node *btree::search(int key, node *leaf){
    if(leaf != NULL){
        if(key == leaf->value){
            return leaf;
        }
        if(key < leaf->value){
            return search(key, leaf->left);
        }else{
            return search(key, leaf->right);
        }
    }else{
        return NULL;
    }
}

node *btree::search(int key){
    return search(key, root);
}
```

```cpp
void btree::destroy_tree(){
  destroy_tree(root);
}

void btree::inorder_print(){
  inorder_print(root);
  cout << "\n";
}

void btree::inorder_print(node *leaf){
  if(leaf != NULL){
     inorder_print(leaf->left);
     cout << leaf->value << ",";
     inorder_print(leaf->right);
  }
}

void btree::postorder_print(){
  postorder_print(root);
  cout << "\n";
}

void btree::postorder_print(node *leaf){
  if(leaf != NULL){
     inorder_print(leaf->left);
     inorder_print(leaf->right);
     cout << leaf->value << ",";
  }
}

void btree::preorder_print(){
  preorder_print(root);
  cout << "\n";
}

void btree::preorder_print(node *leaf){
  if(leaf != NULL){
     cout << leaf->value << ",";
     inorder_print(leaf->left);
     inorder_print(leaf->right);
  }
}
```

Завдання 2

Червоно-чорне дерево. Для заданого масиву ключів (більше 15 значень, задати випадково – цілі числа з множини [0, 100]) побудувати червоно-чорне дерево, реалізувати операції додавання елемента, видалення елемента. Вивести побудовані дерева.

Вивід

```
Enter array length: 20
R----446(BLACK)
  L----268(BLACK)
  |  L----199(RED)
  |  |  L----25(BLACK)
  |  |  |  L----11(RED)
  |  |  R----207(BLACK)
  |  R----320(RED)
  |     L----312(BLACK)
  |     |  L----290(RED)
  |     R----377(BLACK)
  R----852(BLACK)
     L----675(RED)
     |  L----669(BLACK)
     |  |  L----563(RED)
     |  R----770(BLACK)
     |     R----828(RED)
     R----930(RED)
        L----920(BLACK)
        R----933(BLACK)
           R----944(RED)

After deleting
R----446(BLACK)
  L----268(BLACK)
  |  L----199(RED)
  |  |  L----25(BLACK)
  |  |  |  L----11(RED)
  |  |  R----207(BLACK)
  |  R----320(RED)
  |     L----312(BLACK)
  |     |  L----290(RED)
```

```
|     R----377(BLACK)
R----920(BLACK)
   L----675(RED)
   |  L----669(BLACK)
   |  |  L----563(RED)
   |  R----770(BLACK)
   |     R----828(RED)
   R----933(RED)
      L----930(BLACK)
      R----944(BLACK)
```

Process finished with exit code 0

Програмний код

```cpp
#include <iostream>
#include <random>

using namespace std;

struct Node {
 int data;
 Node *parent;
 Node *left;
 Node *right;
 int color;
};

typedef Node *NodePtr;

class RedBlackTree {
  private:
 NodePtr root;
 NodePtr TNULL;

 void initializeNULLNode(NodePtr node, NodePtr parent) {
    node->data = 0;
    node->parent = parent;
    node->left = nullptr;
    node->right = nullptr;
    node->color = 0;
```

```
}

// Preorder
void preOrderHelper(NodePtr node) {
  if (node != TNULL) {
    cout << node->data << " ";
    preOrderHelper(node->left);
    preOrderHelper(node->right);
  }
}

// Inorder
void inOrderHelper(NodePtr node) {
  if (node != TNULL) {
    inOrderHelper(node->left);
    cout << node->data << " ";
    inOrderHelper(node->right);
  }
}

// Post order
void postOrderHelper(NodePtr node) {
  if (node != TNULL) {
    postOrderHelper(node->left);
    postOrderHelper(node->right);
    cout << node->data << " ";
  }
}

NodePtr searchTreeHelper(NodePtr node, int key) {
  if (node == TNULL || key == node->data) {
    return node;
  }

  if (key < node->data) {
    return searchTreeHelper(node->left, key);
  }
  return searchTreeHelper(node->right, key);
}

// For balancing the tree after deletion
void deleteFix(NodePtr x) {
  NodePtr s;
  while (x != root && x->color == 0) {
    if (x == x->parent->left) {
```

```
      s = x->parent->right;
      if (s->color == 1) {
        s->color = 0;
        x->parent->color = 1;
        leftRotate(x->parent);
        s = x->parent->right;
      }

      if (s->left->color == 0 && s->right->color == 0) {
        s->color = 1;
        x = x->parent;
      } else {
        if (s->right->color == 0) {
          s->left->color = 0;
          s->color = 1;
          rightRotate(s);
          s = x->parent->right;
        }

        s->color = x->parent->color;
        x->parent->color = 0;
        s->right->color = 0;
        leftRotate(x->parent);
        x = root;
      }
    } else {
      s = x->parent->left;
      if (s->color == 1) {
        s->color = 0;
        x->parent->color = 1;
        rightRotate(x->parent);
        s = x->parent->left;
      }

      if (s->right->color == 0 && s->right->color == 0) {
        s->color = 1;
        x = x->parent;
      } else {
        if (s->left->color == 0) {
          s->right->color = 0;
          s->color = 1;
          leftRotate(s);
          s = x->parent->left;
        }
```

```
                s->color = x->parent->color;
                x->parent->color = 0;
                s->left->color = 0;
                rightRotate(x->parent);
                x = root;
            }
        }
    }
    x->color = 0;
}

void rbTransplant(NodePtr u, NodePtr v) {
    if (u->parent == nullptr) {
        root = v;
    } else if (u == u->parent->left) {
        u->parent->left = v;
    } else {
        u->parent->right = v;
    }
    v->parent = u->parent;
}

void deleteNodeHelper(NodePtr node, int key) {
    NodePtr z = TNULL;
    NodePtr x, y;
    while (node != TNULL) {
        if (node->data == key) {
            z = node;
        }

        if (node->data <= key) {
            node = node->right;
        } else {
            node = node->left;
        }
    }

    if (z == TNULL) {
        cout << "Key not found in the tree" << endl;
        return;
    }

    y = z;
    int y_original_color = y->color;
    if (z->left == TNULL) {
```

```
      x = z->right;
      rbTransplant(z, z->right);
    } else if (z->right == TNULL) {
      x = z->left;
      rbTransplant(z, z->left);
    } else {
      y = minimum(z->right);
      y_original_color = y->color;
      x = y->right;
      if (y->parent == z) {
        x->parent = y;
      } else {
        rbTransplant(y, y->right);
        y->right = z->right;
        y->right->parent = y;
      }

      rbTransplant(z, y);
      y->left = z->left;
      y->left->parent = y;
      y->color = z->color;
    }
    delete z;
    if (y_original_color == 0) {
      deleteFix(x);
    }
  }

  // For balancing the tree after insertion
  void insertFix(NodePtr k) {
    NodePtr u;
    while (k->parent->color == 1) {
      if (k->parent == k->parent->parent->right) {
        u = k->parent->parent->left;
        if (u->color == 1) {
          u->color = 0;
          k->parent->color = 0;
          k->parent->parent->color = 1;
          k = k->parent->parent;
        } else {
          if (k == k->parent->left) {
            k = k->parent;
            rightRotate(k);
          }
          k->parent->color = 0;
```

```
        k->parent->parent->color = 1;
        leftRotate(k->parent->parent);
      }
    } else {
      u = k->parent->parent->right;

      if (u->color == 1) {
        u->color = 0;
        k->parent->color = 0;
        k->parent->parent->color = 1;
        k = k->parent->parent;
      } else {
        if (k == k->parent->right) {
          k = k->parent;
          leftRotate(k);
        }
        k->parent->color = 0;
        k->parent->parent->color = 1;
        rightRotate(k->parent->parent);
      }
    }
    if (k == root) {
      break;
    }
  }
  root->color = 0;
}

void printHelper(NodePtr root, string indent, bool last) {
  if (root != TNULL) {
    cout << indent;
    if (last) {
      cout << "R----";
      indent += "   ";
    } else {
      cout << "L----";
      indent += "|  ";
    }

    string sColor = root->color ? "RED" : "BLACK";
    cout << root->data << "(" << sColor << ")" << endl;
    printHelper(root->left, indent, false);
    printHelper(root->right, indent, true);
  }
}
```

```
  public:
  RedBlackTree() {
    TNULL = new Node;
    TNULL->color = 0;
    TNULL->left = nullptr;
    TNULL->right = nullptr;
    root = TNULL;
  }

  void preorder() {
    preOrderHelper(root);
  }

  void inorder() {
    inOrderHelper(root);
  }

  void postorder() {
    postOrderHelper(root);
  }

  NodePtr searchTree(int k) {
    return searchTreeHelper(root, k);
  }

  NodePtr minimum(NodePtr node) {
    while (node->left != TNULL) {
      node = node->left;
    }
    return node;
  }

  NodePtr maximum(NodePtr node) {
    while (node->right != TNULL) {
      node = node->right;
    }
    return node;
  }

  NodePtr successor(NodePtr x) {
    if (x->right != TNULL) {
      return minimum(x->right);
    }
```

```
    NodePtr y = x->parent;
    while (y != TNULL && x == y->right) {
      x = y;
      y = y->parent;
    }
    return y;
  }

  NodePtr predecessor(NodePtr x) {
    if (x->left != TNULL) {
      return maximum(x->left);
    }

    NodePtr y = x->parent;
    while (y != TNULL && x == y->left) {
      x = y;
      y = y->parent;
    }

    return y;
  }

  void leftRotate(NodePtr x) {
    NodePtr y = x->right;
    x->right = y->left;
    if (y->left != TNULL) {
      y->left->parent = x;
    }
    y->parent = x->parent;
    if (x->parent == nullptr) {
      this->root = y;
    } else if (x == x->parent->left) {
      x->parent->left = y;
    } else {
      x->parent->right = y;
    }
    y->left = x;
    x->parent = y;
  }

  void rightRotate(NodePtr x) {
    NodePtr y = x->left;
    x->left = y->right;
    if (y->right != TNULL) {
      y->right->parent = x;
```

```
    }
    y->parent = x->parent;
    if (x->parent == nullptr) {
      this->root = y;
    } else if (x == x->parent->right) {
      x->parent->right = y;
    } else {
      x->parent->left = y;
    }
    y->right = x;
    x->parent = y;
  }

  // Inserting a node
  void insert(int key) {
    NodePtr node = new Node;
    node->parent = nullptr;
    node->data = key;
    node->left = TNULL;
    node->right = TNULL;
    node->color = 1;

    NodePtr y = nullptr;
    NodePtr x = this->root;

    while (x != TNULL) {
      y = x;
      if (node->data < x->data) {
        x = x->left;
      } else {
        x = x->right;
      }
    }

    node->parent = y;
    if (y == nullptr) {
      root = node;
    } else if (node->data < y->data) {
      y->left = node;
    } else {
      y->right = node;
    }

    if (node->parent == nullptr) {
      node->color = 0;
```

```
      return;
    }

    if (node->parent->parent == nullptr) {
      return;
    }

    insertFix(node);
  }

  NodePtr getRoot() {
    return this->root;
  }

  void deleteNode(int data) {
    deleteNodeHelper(this->root, data);
  }

  void printTree() {
    if (root) {
      printHelper(this->root, "", true);
    }
  }
};
```

Задача 3.

Бінарна куча. Для заданого масиву ключів (більше 15 значень, задати випадково – цілі числа з множини [0, 100]) побудувати бінарну кучу, реалізувати операції додавання елемента, видалення мінімального елемента. Вивести побудовані дерева.

Вивід

Enter array length: 20
301 447 742 491 305 310 570 95 708 67 478 559 705 586 595 646 921 411 742 626
Min = 67
Extract Min 67
New Min = 95

Програмний код

```
// Prototype of a utility function to swap two integers
void swap(int *x, int *y);

// A class for Min Heap
class MinHeap
{
 int *harr; // pointer to array of elements in heap
 int capacity; // maximum possible size of min heap
 int heap_size; // Current number of elements in min heap
public:
 // Constructor
 MinHeap(int capacity);

 // to heapify a subtree with the root at given index
 void MinHeapify(int );

 int parent(int i) { return (i-1)/2; }

 // to get index of left child of node at index i
 int left(int i) { return (2*i + 1); }

 // to get index of right child of node at index i
 int right(int i) { return (2*i + 2); }

 // to extract the root which is the minimum element
 int extractMin();
```

```
// Decreases key value of key at index i to new_val
void decreaseKey(int i, int new_val);

// Returns the minimum key (key at root) from min heap
int getMin() { return harr[0]; }

// Deletes a key stored at index i
void deleteKey(int i);

// Inserts a new key 'k'
void insertKey(int k);
};

// Constructor: Builds a heap from a given array a[] of given size
MinHeap::MinHeap(int cap)
{
 heap_size = 0;
 capacity = cap;
 harr = new int[cap];
}

// Inserts a new key 'k'
void MinHeap::insertKey(int k)
{
 if (heap_size == capacity)
 {
   cout << "\nOverflow: Could not insertKey\n";
   return;
 }

 // First insert the new key at the end
 heap_size++;
 int i = heap_size - 1;
 harr[i] = k;

 // Fix the min heap property if it is violated
 while (i != 0 && harr[parent(i)] > harr[i])
 {
   swap(&harr[i], &harr[parent(i)]);
   i = parent(i);
 }
}

// Decreases value of key at index 'i' to new_val. It is assumed that
```

```cpp
// new_val is smaller than harr[i].
void MinHeap::decreaseKey(int i, int new_val)
{
 harr[i] = new_val;
 while (i != 0 && harr[parent(i)] > harr[i])
 {
    swap(&harr[i], &harr[parent(i)]);
    i = parent(i);
 }
}

// Method to remove minimum element (or root) from min heap
int MinHeap::extractMin()
{
 if (heap_size <= 0)
    return INT_MAX;
 if (heap_size == 1)
 {
    heap_size--;
    return harr[0];
 }

 // Store the minimum value, and remove it from heap
 int root = harr[0];
 harr[0] = harr[heap_size-1];
 heap_size--;
 MinHeapify(0);

 return root;
}


// This function deletes key at index i. It first reduced value to
minus
// infinite, then calls extractMin()
void MinHeap::deleteKey(int i)
{
 decreaseKey(i, INT_MIN);
 extractMin();
}

// A recursive method to heapify a subtree with the root at given
index
// This method assumes that the subtrees are already heapified
void MinHeap::MinHeapify(int i)
```

```
{
 int l = left(i);
 int r = right(i);
 int smallest = i;
 if (l < heap_size && harr[l] < harr[i])
    smallest = l;
 if (r < heap_size && harr[r] < harr[smallest])
    smallest = r;
 if (smallest != i)
 {
    swap(&harr[i], &harr[smallest]);
    MinHeapify(smallest);
 }
}

// A utility function to swap two elements
void swap(int *x, int *y)
{
 int temp = *x;
 *x = *y;
 *y = temp;
}
```