Addis Ababa Science and Technology University

College of Electrical and Mechanical Engineering

Department of Electrical and Computer Engineering

Computer Engineering Stream

Course name: Data structure and Algorithm

Group members

	Name	ID
1.	Yohannes Girmaw	ETS 1030/09
2.	Yohannes Abera	ETS 1025/09
3.	Mulualem Fekadu	ETS 0730/09
4.	Sara Reta	ETS 0858/09
5.	Zenaw Asfaw	ETS 1078/09
6.	Chara Biratu	ETS 0282/09
7.		

Submitted Date: 13/1/2020 GC

Submitted To: Dr. Solomon

RED BLACK TREE PROJECT

Introduction

A red–black tree is a kind of self-balancing binary search tree in computer science. Each node of the binary tree has an extra bit, and that bit is often interpreted as the color (red or black) of the node. These color bits are used to ensure the tree remains approximately balanced during insertions and deletions.

Balance is preserved by painting each node of the tree with one of two colors in a way that satisfies certain properties, which collectively constrain how unbalanced the tree can become in the worst case. When the tree is modified, the new tree is subsequently rearranged and repainted to restore the coloring properties. The properties are designed in such a way that this rearranging and recoloring can be performed efficiently.

The balancing of the tree is not perfect, but it is good enough to allow it to guarantee searching in O (log n) time, where n is the total number of elements in the tree. The insertion and deletion operations, along with the tree rearrangement and recoloring, are also performed in O (log n) time.

Algorithm	**Average**	**Worst case**
Space	O (n)	O (n)
Search	O (log n)	O (log n)
Insert	O (log n)	O (log n)
Delete	O (log n)	O (log n)

In addition to the requirements imposed on a binary search tree the following must be satisfied by a red–black tree:

1. Each node is either red or black.
2. The root is black. This rule is sometimes omitted. Since the root can always be changed from red to black, but not necessarily vice versa, this rule has little effect on analysis.
3. All leaves (NIL) are black.
4. If a node is red, then both its children are black.
5. Every path from a given node to any of its descendant NIL nodes contains the same number of black nodes.

Operation

Read-only operations on a red–black tree require no modification from those used for binary search trees, because every red–black tree is a special case of a simple binary search tree. However, the immediate result of an insertion or removal may violate the properties of a red–black tree. Restoring the red–black properties requires a small number (O(log n) or amortized O(1)) of color changes (which are very quick in practice) and no more than three tree rotations (two for insertion). Although insert and delete operations are complicated, their times remain O (log n).

```
enum COLOR { RED, BLACK };

class Node {
public:
      int val;
      COLOR color;
      Node *left, *right, *parent;

      Node(int val) : val(val) {
        parent = left = right = NULL;

        // Node is created during insertion
        // Node is red at insertion
        color = RED;
      }

      // returns pointer to uncle
      Node *uncle() {
        // If no parent or grandparent, then no uncle
        if (parent == NULL or parent->parent == NULL)
          return NULL;

        if (parent->isOnLeft())
          // uncle on right
          return parent->parent->right;
        else
          // uncle on left
          return parent->parent->left;
      }

      // check if node is left child of parent
      bool isOnLeft() { return this == parent->left; }

      // returns pointer to sibling
      Node *sibling() {
        // sibling null if no parent
        if (parent == NULL)
          return NULL;
        if (isOnLeft())
          return parent->right;

        return parent->left;
      }

      // moves node down and moves given node in its place
      void moveDown(Node *nParent) {
        if (parent != NULL) {
          if (isOnLeft()) {
            parent->left = nParent;
          } else {
            parent->right = nParent;
          }
        }
        nParent->parent = parent;
        parent = nParent;
      }

      bool hasRedChild() {
        return (left != NULL and left->color == RED) or
              (right != NULL and right->color == RED);
      }
};
```

```
class BuildRedBlack {
  Node *root;

  // left rotates the given node
  void leftRotate(Node *x) {
    // new parent will be node's right child
        Node *nParent = x->right;

        // update root if current node is root
        if (x == root)
          root = nParent;

        x->moveDown(nParent);

        // connect x with new parent's left element
        x->right = nParent->left;
        // connect new parent's left element with node
        // if it is not null
        if (nParent->left != NULL)
          nParent->left->parent = x;

        // connect new parent with x
        nParent->left = x;
  }

  void rightRotate(Node *x) {
    // new parent will be node's left child
        Node *nParent = x->left;

        // update root if current node is root
        if (x == root)
          root = nParent;

        x->moveDown(nParent);

        // connect x with new parent's right element
        x->left = nParent->right;
        // connect new parent's right element with node
        // if it is not null
        if (nParent->right != NULL)
          nParent->right->parent = x;

        // connect new parent with x
        nParent->right = x;
  }

  void swapColors(Node *x1, Node *x2) {
        COLOR temp;
        temp = x1->color;
        x1->color = x2->color;
        x2->color = temp;
  }

  void swapValues(Node *u, Node *v) {
        int temp;
        temp = u->val;
        u->val = v->val;
        v->val = temp;
  }
```

Insertion

Insertion begins by adding the node in a very similar manner as a standard binary search tree insertion and by coloring it red. The big difference is that in the binary search tree a new node is added as a leaf, whereas leaves contain no information in the red–black tree, so instead the new node replaces an existing leaf and then has two black leaves of its own added.

```
void insert(int n) {
    Node *newNode = new Node(n);
    if (root == NULL) {
      // when root is null
      // simply insert value at root
      newNode->color = BLACK;
      root = newNode;
    } else {
      Node *temp = contain(n);

      if (temp->val == n) {
        // return if value already exists
        return;
      }
```

What happens next depends on the color of other nearby nodes. There are several cases of red–black tree insertion to handle:

Case 1: Current node is the root node, i.e., first node of red–black tree

Case 2: Current node's parent is black

Case 3: Parent is red (so it can't be the root of the tree) and current node's uncle is red

Case 4: Parent is red and Uncle is black

Case 1: The current node is at the root of the tree. In this case, it is repainted black to satisfy the root is black. Since this adds one black node to every path at once, all paths from any given node to its leaf nodes contain the same number of black nodes is not violated.

```
Node *getRoot() { return root; }
Node *newNode = new Node(n);

    if (root == NULL) {

      // when root is null

      // simply insert value at root

      newNode->color = BLACK;

      root = newNode;
```

Case 2: The current node's parent is black, so both children of every red node are black is not invalidated. In this case, the tree is still valid. all paths from any given node to its leaf nodes contain the same number of black nodes is not threatened, because the current node has two black leaf children, but because current node is red, the paths through each of its children have the same number of black nodes as the path through the leaf it replaced, which was black, and so this property remains satisfied.

In case two, the condition is valid. So we left it as it is.

- In the following cases below it can be assumed that current node has a grandparent node, because its parent is red, and if it were the root, it would be black. Thus, current node also has an uncle node, although it may be a leaf in case 4.

- In the remaining cases, it is shown in the diagram that the parent node is the left child of its parent even though it is possible for parent node to be on either side. The code samples already cover both possibilities.

Case 3: If both the parent and the uncle are red, then both of them can be repainted black and the grandparent becomes red to maintain all paths from any given node to its leaf nodes contain the same number of black nodes. Since any path through the parent or uncle must pass through the grandparent, the number of black nodes on these paths has not changed. However, the grandparent may now violate the root is black if it is the root or both children of every red node are black if it has a red parent. To fix this, the tree's red-black repair procedure is rerun on grandparent.

```
  // fix red red at given node
  void fixRedRed(Node *x) {
        // if x is root color it black and return
        if (x == root) {
          x->color = BLACK;
          return;
    }

    // initialize parent, grandparent, uncle
  Node *parent = x->parent, *grandparent = parent->parent,
         *uncle = x->uncle();
    if (parent->color != BLACK) {
      if (uncle != NULL && uncle->color == RED) {
        // uncle red, perform recoloring and recurse
        parent->color = BLACK;
        uncle->color = BLACK;
        grandparent->color = RED;
        fixRedRed(grandparent);
      } else {
        // Else perform LR, LL, RL, RR
        if (parent->isOnLeft()) {
          if (x->isOnLeft()) {
            // for left right
            swapColors(parent, grandparent);
          } else {
            leftRotate(parent);
            swapColors(x, grandparent);
          }
          // for left left and left right
          rightRotate(grandparent);
        } else {
          if (x->isOnLeft()) {
            // for right left
            rightRotate(parent);
            swapColors(x, grandparent);
          } else {
            swapColors(parent, grandparent);
          }

          // for right right and right left
          leftRotate(grandparent);
        }
      }
    }
  }
```

Case 4: The parent is red but the uncle is black. The ultimate goal will be to rotate the current node into the grandparent position, but this will not work if the current node is on the "inside" of the subtree under grandparent (i.e., if current node is the left child of the right child of the grandparent or the right child of the left child of the grandparent). In this case, a left rotation on parent that switches the roles of the current node and its parent can be performed. The rotation causes some paths (those in the sub-tree labelled "1") to pass through the current node where they did not before. It also causes some paths (those in the sub-tree labelled "3") not to pass through the node parent where they did before. However, both of these nodes are red, so all paths from any given node to its leaf nodes contain the same number of black nodes is not violated by the rotation. After this step has been completed, both children of every red node are black is still violated.

```
void fixDoubleBlack(Node *x) {
        if (x == root)
          // Reached root
          return;

        Node *sibling = x->sibling(), *parent = x->parent;
        if (sibling == NULL) {
          // No sibiling, double black pushed up
          fixDoubleBlack(parent);
        } else {
          if (sibling->color == RED) {
            // Sibling red
            parent->color = RED;
            sibling->color = BLACK;
            if (sibling->isOnLeft()) {
              // left case
              rightRotate(parent);
            } else {
              // right case
              leftRotate(parent);
            }
            fixDoubleBlack(x);
          } else {
            // Sibling black
            if (sibling->hasRedChild()) {
              // at least 1 red children
              if (sibling->left != NULL and sibling->left->color ==
RED) {
                if (sibling->isOnLeft()) {
                  // left left
                  sibling->left->color = sibling->color;
                  sibling->color = parent->color;
                  rightRotate(parent);
                } else {
                  // right left
                  sibling->left->color = parent->color;
                  rightRotate(sibling);
                  leftRotate(parent);
                }
              } else {
                if (sibling->isOnLeft()) {
                  // left right
                  sibling->right->color = parent->color;
                  leftRotate(sibling);
                  rightRotate(parent);
                } else {
                  // right right
                  sibling->right->color = sibling->color;
                  sibling->color = parent->color;
                  leftRotate(parent);
                }
              }
              parent->color = BLACK;
            } else {
              // 2 black children
              sibling->color = RED;
              if (parent->color == BLACK)
                fixDoubleBlack(parent);
              else
                parent->color = BLACK;
        }
      }
    }
```

Deletion:

There are several cases of red–black tree deletion to handle:

Case 1: Current node is new root

Case 2: Son is red

Case 3: Parent, son, and son's children are black

Case 4: Son and son's children are black, but parent is red

Case 5: Son is black, son's left child is red, son's right child is black, and current node is the left child of its parent.

Case 6: Son is black, son's right child is red, and current node is the left child of its parent P

```
// utility function that deletes the node with given value
  void deleteByVal(int n) {
    if (root == NULL)
      // Tree is empty
      return;

    Node *v = contain(n), *u;

    if (v->val != n) {
      cout << "No node found to delete with value:" << n << endl;
      return;
    }

    deleteNode(v);
  }
```

Case 1: current node is the new root. In this case, we are done. We removed one black node from every path, and the new root is black, so the properties are preserved.

❖ In cases 2, 5, and 6, we assume current node is the left child of its parent. If it is the right child, left and right should be reversed throughout these three cases. Again, the code take both cases into account.

Case 2: Son is red (Assume current node is the left child). In this case we reverse the colors of parent and son, and then rotate left at parent, turning son into current node's grandparent. Note that parent has to be black as it had a red child. The resulting subtree has a path short one black node so we are not done. Now current node has a black sibling and a red parent, so we can proceed to step 4, 5, or 6. (Its new sibling is black because it was once the child of the red son.) In later cases, we will relabel current node's new sibling as son.

Case 3: Parent, son, and son's children are black. In this case, we simply repaint son red. The result is that all paths passing through son, which precisely those paths are not passing through current node, have one less black node. Because deleting current node's original parent made all paths passing through current node have one less black node, this evens things up. However, all paths through parent now have one fewer black node than paths that do not pass through parent, so all paths from any given node to its leaf nodes contain the same number of black nodes is still violated. To correct this, we perform the rebalancing procedure on parent, starting at case 1.

Case 4: Son and son's children are black, but parent is red. In this case, we simply exchange the colors of son and parent. This does not affect the number of black nodes on paths going through son, but it does add one to the number of black nodes on paths going through current node, making up for the deleted black node on those paths.

Case 5: Son is black, son's left child is red, son's right child is black, and current node is the left child of its parent. In this case we rotate right at son, so that son's left child becomes son's parent and current node's new sibling. We then exchange the colors of son and its new parent. All paths still have the same number of black nodes, but now current node has a black sibling whose right child is red, so we fall into case 6. Neither current node nor its parent are affected by this transformation. (Again, for case 6, we relabel current's new sibling as son.)

Those cases are all implemented in one function called deleteNode().The code is given below:

```
  // deletes the given node
  void deleteNode(Node *v) {
        Node *u = BSTreplace(v);

        // True when u and v are both black
        bool uvBlack = ((u == NULL or u->color == BLACK) and (v->color
== BLACK));
        Node *parent = v->parent;
```

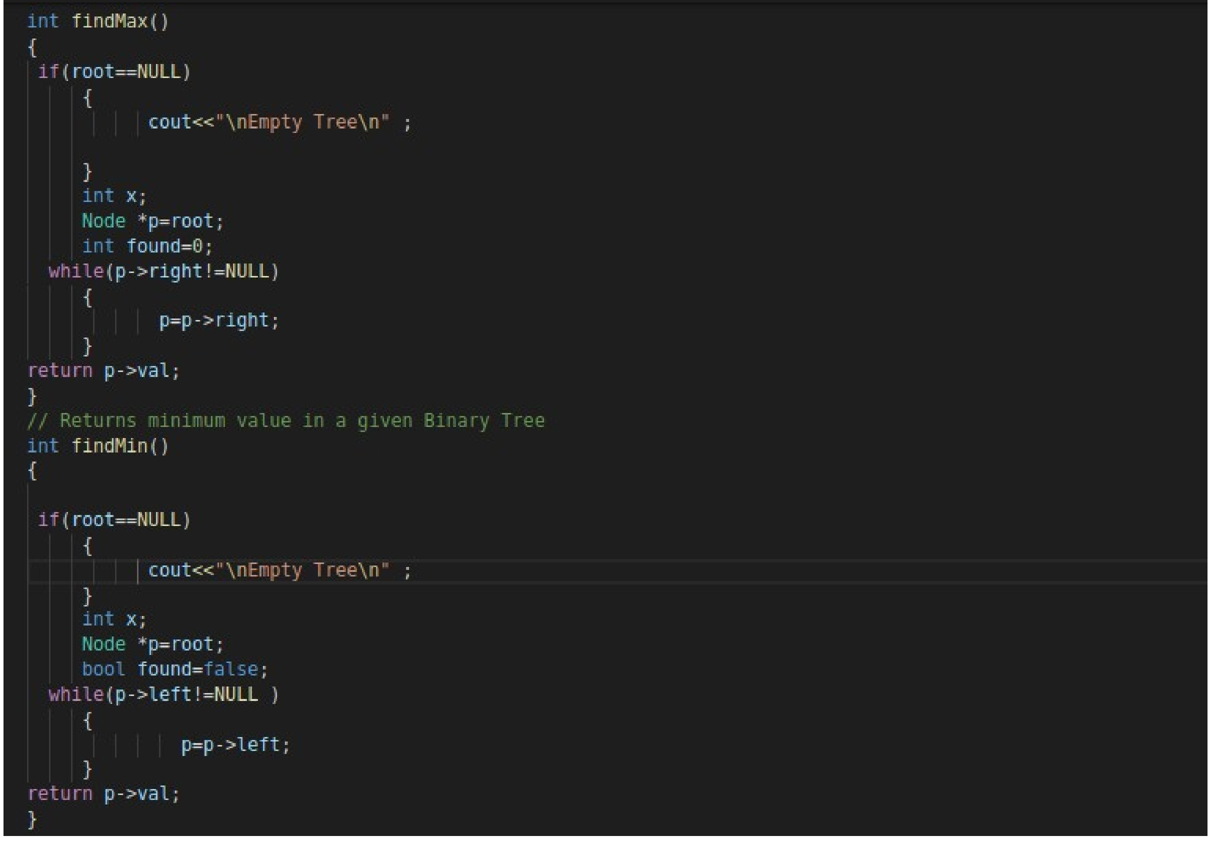
```cpp
    if (u == NULL) {
      // u is NULL therefore v is leaf
      if (v == root) {
        // v is root, making root null
        root = NULL;
      } else {
        if (uvBlack) {
          // u and v both black
          // v is leaf, fix double black at v
          fixDoubleBlack(v);
        } else {
          // u or v is red
          if (v->sibling() != NULL)
            // sibling is not null, make it red"
            v->sibling()->color = RED;
        }

        // delete v from the tree
        if (v->isOnLeft()) {
          parent->left = NULL;
        } else {
          parent->right = NULL;
        }
      }
      delete v;
      return;
    }

    if (v->left == NULL or v->right == NULL) {
      // v has 1 child
      if (v == root) {
        // v is root, assign the value of u to v, and delete u
        v->val = u->val;
        v->left = v->right = NULL;
        delete u;
      } else {
        // Detach v from tree and move u up
        if (v->isOnLeft()) {
          parent->left = u;
        } else {
          parent->right = u;
        }
        delete v;
        u->parent = parent;
        if (uvBlack) {
          // u and v both black, fix double black at u
          fixDoubleBlack(u);
```

```
            } else {
              // u or v red, color u black
              u->color = BLACK;
            }
          }
          return;
        }

        // v has 2 children, swap values with successor and recurse
        swapValues(u, v);
        deleteNode(u);
  }
```

Find min and find max

- To perform a findMin, start at the root and go left as long as there is a left child.
- To perform a findMax routine is the same, except that branching is to the right child
- The stopping point is the smallest element.

```
// Returns minimum value in a given Binary Tree
```



```
int findMax()
{
 if(root==NULL)
     {
         cout<<"\nEmpty Tree\n" ;

     }
     int x;
     Node *p=root;
     int found=0;
  while(p->right!=NULL)
     {
          p=p->right;
     }
return p->val;
}
// Returns minimum value in a given Binary Tree
int findMin()
{

 if(root==NULL)
     {
         cout<<"\nEmpty Tree\n" ;
     }
     int x;
     Node *p=root;
     bool found=false;
  while(p->left!=NULL )
     {
            p=p->left;
     }
return p->val;
}
```

Inorder

The general strategy of an inorder traversal is to process the left subtree first, then perform processing at the current node, and finally process the right subtree.

```
void inorder(Node *x) {

    if (x == NULL)

      return;

    inorder(x->left);

    cout << x->val << " ";

    inorder(x->right);

  }
```

```
void printInOrder() {
    cout << "Inorder: " << endl;
    if (root == NULL)
      cout << "Tree is empty" << endl;
    else
      inorder(root);
    cout << endl;
  }
```

Level order

Given a binary tree, print its node level by level. This means all nodes present at level 1 should be printed first followed by nodes of level 2 and so on.

```
  // prints level order for given node
  void levelOrder(Node *x) {
    if (x == NULL)
      // return if node is null
      return;

    // queue for level order
    queue<Node *> q;
    Node *curr;

    // push x
    q.push(x);

    while (!q.empty()) {
      // while q is not empty
      // dequeue
      curr = q.front();
      q.pop();
      // print node value
      cout << curr->val << " ";

      // push children to queue
      if (curr->left != NULL)
        q.push(curr->left);
      if (curr->right != NULL)
        q.push(curr->right);
    }
  }
```

```
  void printLevelOrder() {
    cout << "Level order: " << endl;
    if (root == NULL)
      cout << "Tree is empty" << endl;
    else
      levelOrder(root);
    cout << endl;
  }
```

Contains or Searching

- This operation requires returning
  - True if there is a node in tree *T* that has item *X*, or
  - False if there is no such node.
- If *T* is empty, then we can just return false.
- Otherwise, if the item stored at *T* is *X*, we can return true.
- Otherwise, we make a recursive call on a subtree of *T*, either left or right accordingly.
- If the X value is less than node go to left otherwise go to right

The Code which is used to Contain for our red black tree is:

```
Node *contain(int n) {
    Node *temp = root;
    while (temp != NULL) {
      if (n < temp->val) {
        if (temp->left == NULL)
          break;
        else
          temp = temp->left;
      } else if (n == temp->val) {
       // cout<<"the number we searched for is:"<<n;
        break;
      } else {
        if (temp->right == NULL)
```

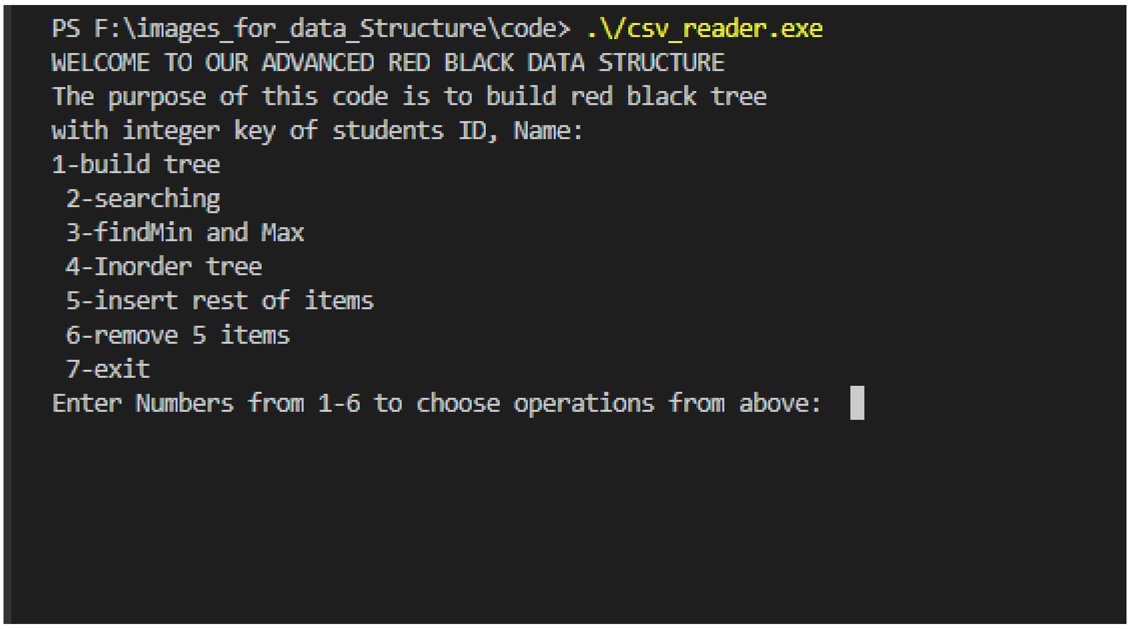
```
          break;
        else
          temp = temp->right;
      }
    }

    return temp;
  }
```

The Code which is used to Search for a value from the given excel file is:

```
void search()
{
     if(root==NULL)
     {
           cout<<"\nEmpty Tree\n" ;
           return  ;
     }
     int x;
     cout<<"\n Enter val of the node to be searched: ";
     cin>>x;
     Node *p=root;
     int found=0;
     while(p!=NULL&& found==0)
     {
            if(p->val==x)
                found=1;
            if(found==0)
            {
                 if(p->val<x)
                      p=p->right;
                 else
                      p=p->left;
            }
     }
     if(found==0)
          cout<<"\nElement Not Found.";
     else
     {
                cout<<"\n\t FOUND NODE: ";
                cout<<"\n val: "<<p->val;
               // cout<<"\n Colour: ";

                if(p->parent!=NULL)
                       cout<<"\n Parent: "<<p->parent->val;
                else
                       cout<<"\n There is no parent of the node.  ";
                if(p->right!=NULL)
                       cout<<"\n Right Child: "<<p->right->val;
                else
                       cout<<"\n There is no right child of the node.
";
                if(p->left!=NULL)
                       cout<<"\n Left Child: "<<p->left->val;
                else
                       cout<<"\n There is no left child of the node.
";
                cout<<endl;

     }
}
```

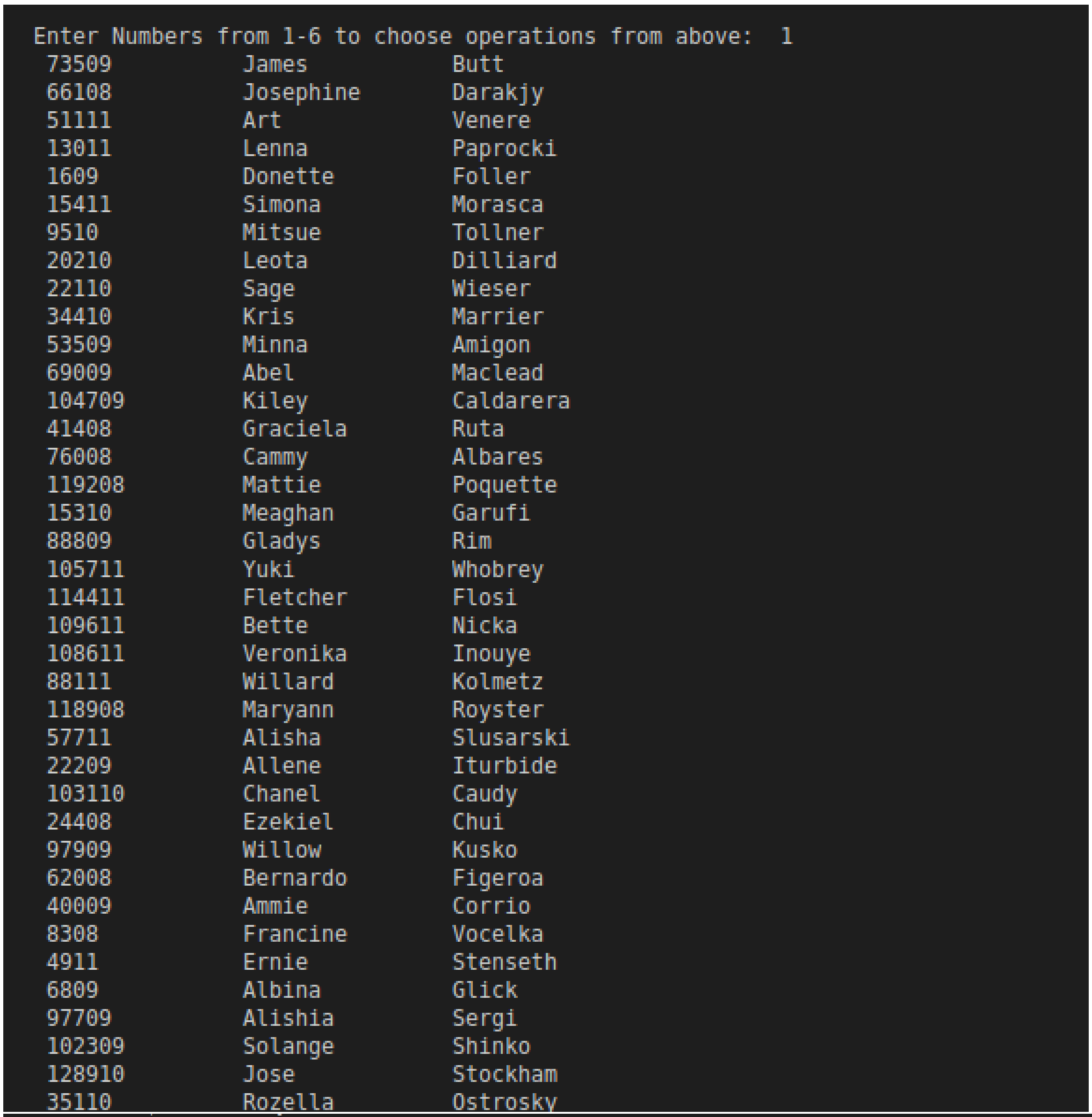
The outputs:

The output is printed below from operation one(1) to operation seven(7)

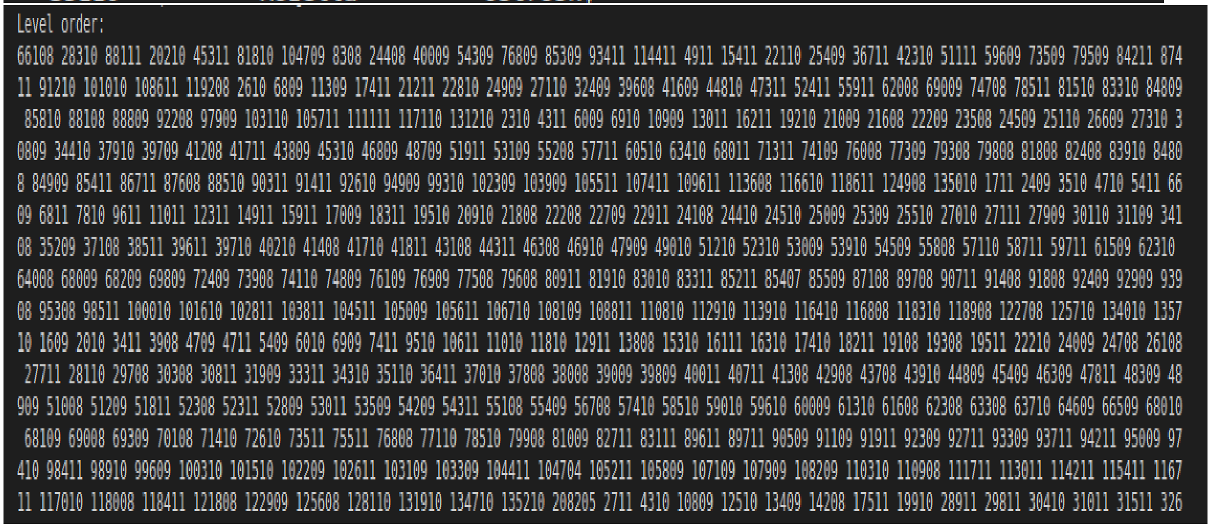


```
PS F:\images_for_data_Structure\code> .\/csv_reader.exe
WELCOME TO OUR ADVANCED RED BLACK DATA STRUCTURE
The purpose of this code is to build red black tree
with integer key of students ID, Name:
1-build tree
 2-searching
 3-findMin and Max
 4-Inorder tree
 5-insert rest of items
 6-remove 5 items
 7-exit
Enter Numbers from 1-6 to choose operations from above:
```

a) **BuildRedblack – with the first 450 items**

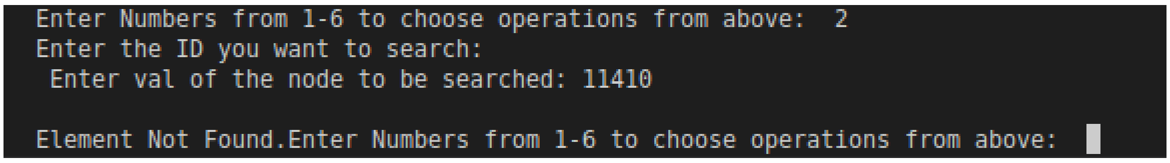


```
 Enter Numbers from 1-6 to choose operations from above:  1
  73509          James            Butt
  66108          Josephine        Darakjy
  51111          Art              Venere
  13011          Lenna            Paprocki
  1609           Donette          Foller
  15411          Simona           Morasca
  9510           Mitsue           Tollner
  20210          Leota            Dilliard
  22110          Sage             Wieser
  34410          Kris             Marrier
  53509          Minna            Amigon
  69009          Abel             Maclead
  104709         Kiley            Caldarera
  41408          Graciela         Ruta
  76008          Cammy            Albares
  119208         Mattie           Poquette
  15310          Meaghan          Garufi
  88809          Gladys           Rim
  105711         Yuki             Whobrey
  114411         Fletcher         Flosi
  109611         Bette            Nicka
  108611         Veronika         Inouye
  88111          Willard          Kolmetz
  118908         Maryann          Royster
  57711          Alisha           Slusarski
  22209          Allene           Iturbide
  103110         Chanel           Caudy
  24408          Ezekiel          Chui
  97909          Willow           Kusko
  62008          Bernardo         Figeroa
  40009          Ammie            Corrio
  8308           Francine         Vocelka
  4911           Ernie            Stenseth
  6809           Albina           Glick
  97709          Alishia          Sergi
  102309         Solange          Shinko
  128910         Jose             Stockham
  35110          Rozella          Ostrosky
```

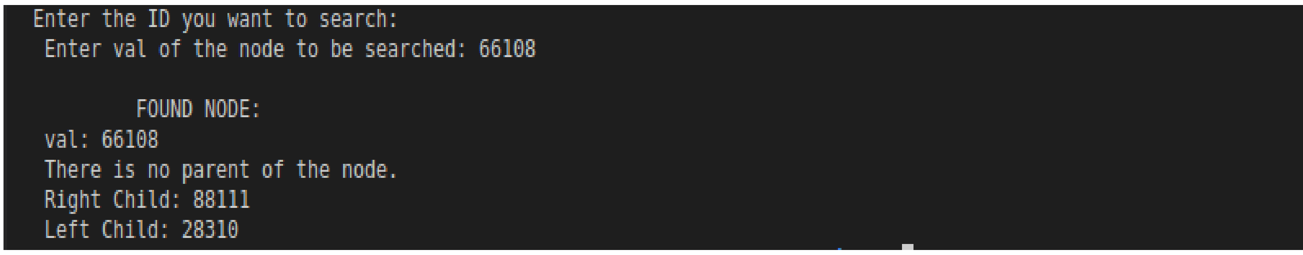


```
Level order:
66108 28310 88111 20210 45311 81810 104709 8308 24408 40009 54309 76809 85309 93411 114411 4911 15411 22110 25409 36711 42310 51111 59609 73509 79509 84211 874
11 91210 101010 108611 119208 2610 6809 11309 17411 21211 22810 24909 27110 32409 39608 41609 44810 47311 52411 55911 62008 69009 74708 78511 81510 83310 84809
 85810 88108 88809 92208 97909 103110 105711 111111 117110 131210 2310 4311 6009 6910 10909 13011 16211 19210 21009 21608 22209 23508 24509 25110 26609 27310 3
0809 34410 37910 39709 41208 41711 43809 45310 46809 48709 51911 53109 55208 57711 60510 63410 68011 71311 74109 76008 77309 79308 79808 81808 82408 83910 8480
8 84909 85411 86711 87608 88510 90311 91411 92610 94909 99310 102309 103909 105511 107411 109611 113608 116610 118611 124908 135010 1711 2409 3510 4710 5411 66
09 6811 7810 9611 11011 12311 14911 15911 17009 18311 19510 20910 21808 22208 22709 22911 24108 24410 24510 25009 25309 25510 27010 27111 27909 30110 31109 341
08 35209 37108 38511 39611 39710 40210 41408 41710 41811 43108 44311 46308 46910 47909 49010 51210 52310 53009 53910 54509 55808 57110 58711 59711 61509 62310
64008 68009 68209 69809 72409 73908 74110 74809 76109 76909 77508 79608 80911 81910 83010 83311 85211 85407 85509 87108 89708 90711 91408 91808 92409 92909 939
08 95308 98511 100010 101610 102811 103811 104511 105009 105611 106710 108109 108811 110810 112910 113910 116410 116808 118310 118908 122708 125710 134010 1357
10 1609 2010 3411 3908 4709 4711 5409 6010 6909 7411 9510 10611 11010 11810 12911 13808 15310 16111 16310 17410 18211 19108 19308 19511 22210 24009 24708 26108
 27711 28110 29708 30308 30811 31909 33311 34310 35110 36411 37010 37808 38008 39009 39809 40011 40711 41308 42908 43708 43910 44809 45409 46309 47811 48309 48
909 51008 51209 51811 52308 52311 52809 53011 53509 54209 54311 55108 55409 56708 57410 58510 59010 59610 60009 61310 61608 62308 63308 63710 64609 66509 68010
 68109 69008 69309 70108 71410 72610 73511 75511 76808 77110 78510 79908 81009 82711 83111 89611 89711 90509 91109 91911 92309 92711 93309 93711 94211 95009 97
410 98411 98910 99609 100310 101510 102209 102611 103109 103309 104411 104704 105211 105809 107109 107909 108209 110310 110908 111711 113011 114211 115411 1167
11 117010 118008 118411 121808 122909 125608 128110 131910 134710 135210 208205 2711 4310 10809 12510 13409 14208 17511 19910 28911 29811 30410 31011 31511 326
```

b) **Contains or Searching – check if an item is in the tree or not**

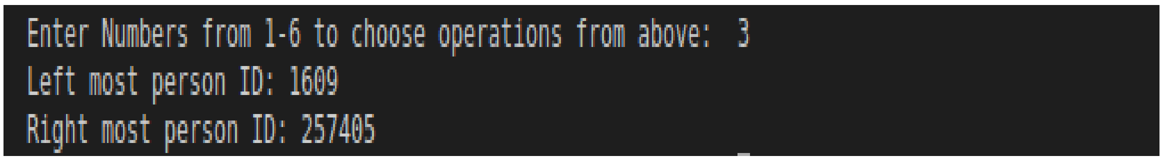


```
Enter Numbers from 1-6 to choose operations from above:  2
Enter the ID you want to search:
 Enter val of the node to be searched: 11410

Element Not Found.Enter Numbers from 1-6 to choose operations from above:
```



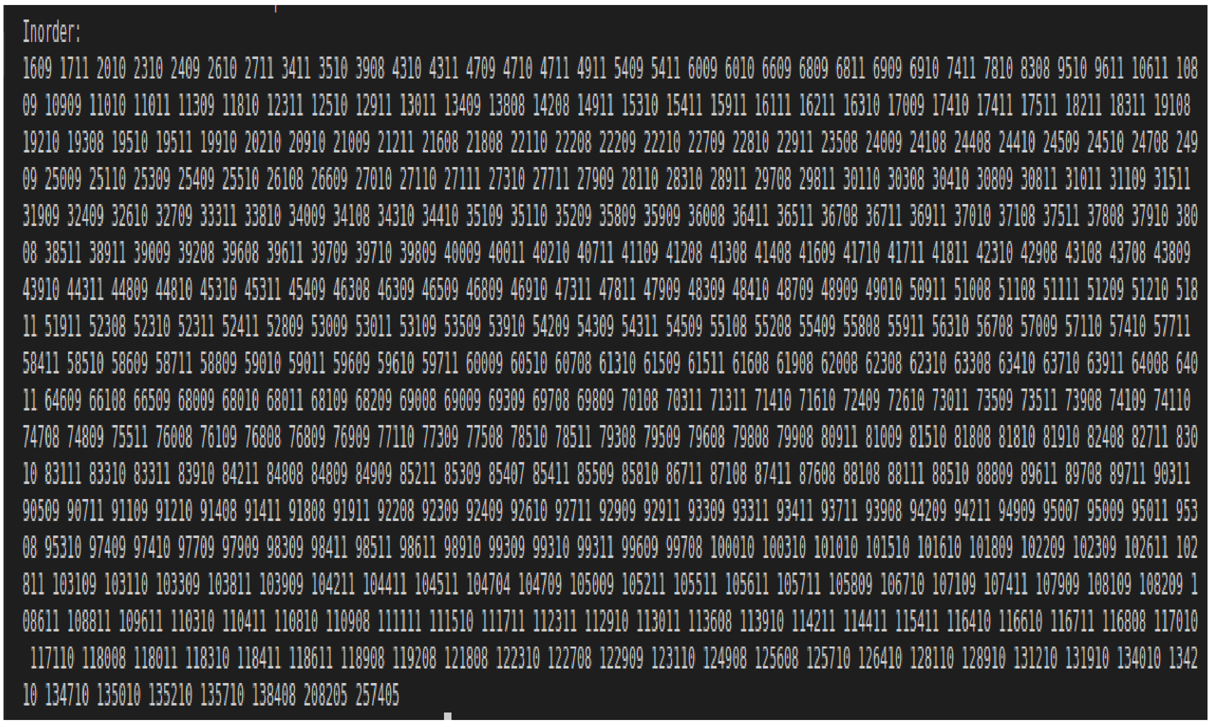
```
Enter the ID you want to search:
 Enter val of the node to be searched: 66108

        FOUND NODE:
 val: 66108
 There is no parent of the node.
 Right Child: 88111
 Left Child: 28310
```

c) **Find Min and Find Max: You can check it from the In order traversal**



```
Enter Numbers from 1-6 to choose operations from above:  3
Left most person ID: 1609
Right most person ID: 257405
```

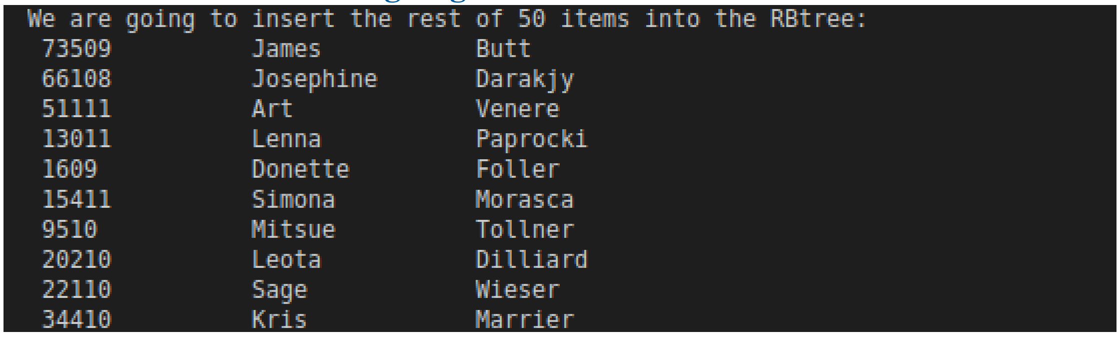
d) **Inorder traversal of the red black tree**



```
Inorder:
1609 1711 2010 2310 2409 2610 2711 3411 3510 3908 4310 4311 4709 4710 4711 4911 5409 5411 6009 6010 6609 6809 6811 6909 6910 7411 7810 8308 9510 9611 10611 108
09 10909 11010 11011 11309 11810 12311 12510 12911 13011 13409 13808 14208 14911 15310 15411 15911 16111 16211 16310 17009 17410 17411 17511 18211 18311 19108
19210 19308 19510 19511 19910 20210 20910 21009 21211 21608 21808 22110 22208 22209 22210 22709 22810 22911 23508 24009 24108 24408 24410 24509 24510 24708 249
09 25009 25110 25309 25409 25510 26108 26609 27010 27110 27111 27310 27711 27909 28110 28310 28911 29708 29811 30110 30308 30410 30809 30811 31011 31109 31511
31909 32409 32610 32709 33311 33810 34009 34108 34310 34410 35109 35110 35209 35809 35909 36008 36411 36511 36708 36711 36911 37010 37108 37511 37808 37910 380
08 38511 38911 39009 39208 39608 39611 39709 39710 39809 40009 40011 40210 40711 41109 41208 41308 41408 41609 41710 41711 41811 42310 42908 43108 43708 43809
43910 44311 44809 44810 45310 45311 45409 46308 46309 46509 46809 46910 47311 47811 47909 48309 48410 48709 48909 49010 50911 51008 51108 51111 51209 51210 518
11 51911 52308 52310 52311 52411 52809 53009 53011 53109 53509 53910 54209 54309 54311 54509 55108 55208 55409 55808 55911 56310 56708 57009 57110 57410 57711
58411 58510 58609 58711 58809 59010 59011 59609 59610 59711 60009 60510 60708 61310 61509 61511 61608 61908 62008 62308 62310 63308 63410 63710 63911 64008 640
11 64609 66108 66509 68009 68010 68011 68109 68209 69008 69009 69309 69708 69809 70108 70311 71311 71410 71610 72409 72610 73011 73509 73511 73908 74109 74110
74708 74809 75511 76008 76109 76808 76809 76909 77110 77309 77508 78510 78511 79308 79509 79608 79808 79908 80911 81009 81510 81808 81810 81910 82408 82711 830
10 83111 83310 83311 83910 84211 84808 84809 84909 85211 85309 85407 85411 85509 85810 86711 87108 87411 87608 88108 88111 88510 88809 89611 89708 89711 90311
90509 90711 91109 91210 91408 91411 91808 91911 92208 92309 92409 92610 92711 92909 92911 93309 93311 93411 93711 93908 94209 94211 94909 95007 95009 95011 953
08 95310 97409 97410 97709 97909 98309 98411 98511 98611 98910 99309 99310 99311 99609 99708 100010 100310 101010 101510 101610 101809 102209 102309 102611 102
811 103109 103110 103309 103811 103909 104211 104411 104511 104704 104709 105009 105211 105511 105611 105711 105809 106710 107109 107411 107909 108109 108209 1
08611 108811 109611 110310 110411 110810 110908 111111 111510 111711 112311 112910 113011 113608 113910 114211 114411 115411 116410 116610 116711 116808 117010
 117110 118008 118011 118310 118411 118611 118908 119208 121808 122310 122708 122909 123110 124908 125608 125710 126410 128110 128910 131210 131910 134010 1342
10 134710 135010 135210 135710 138408 208205 257405
```

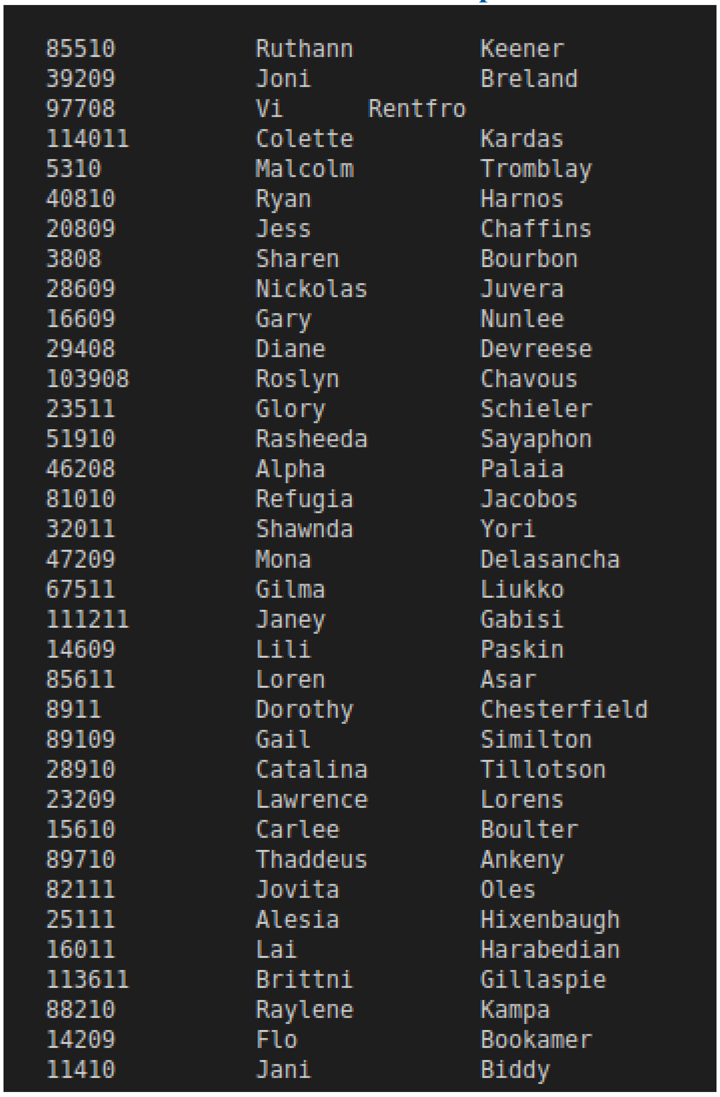
e) **Insert the rest 50 items**

those are values at the beigning of the excel



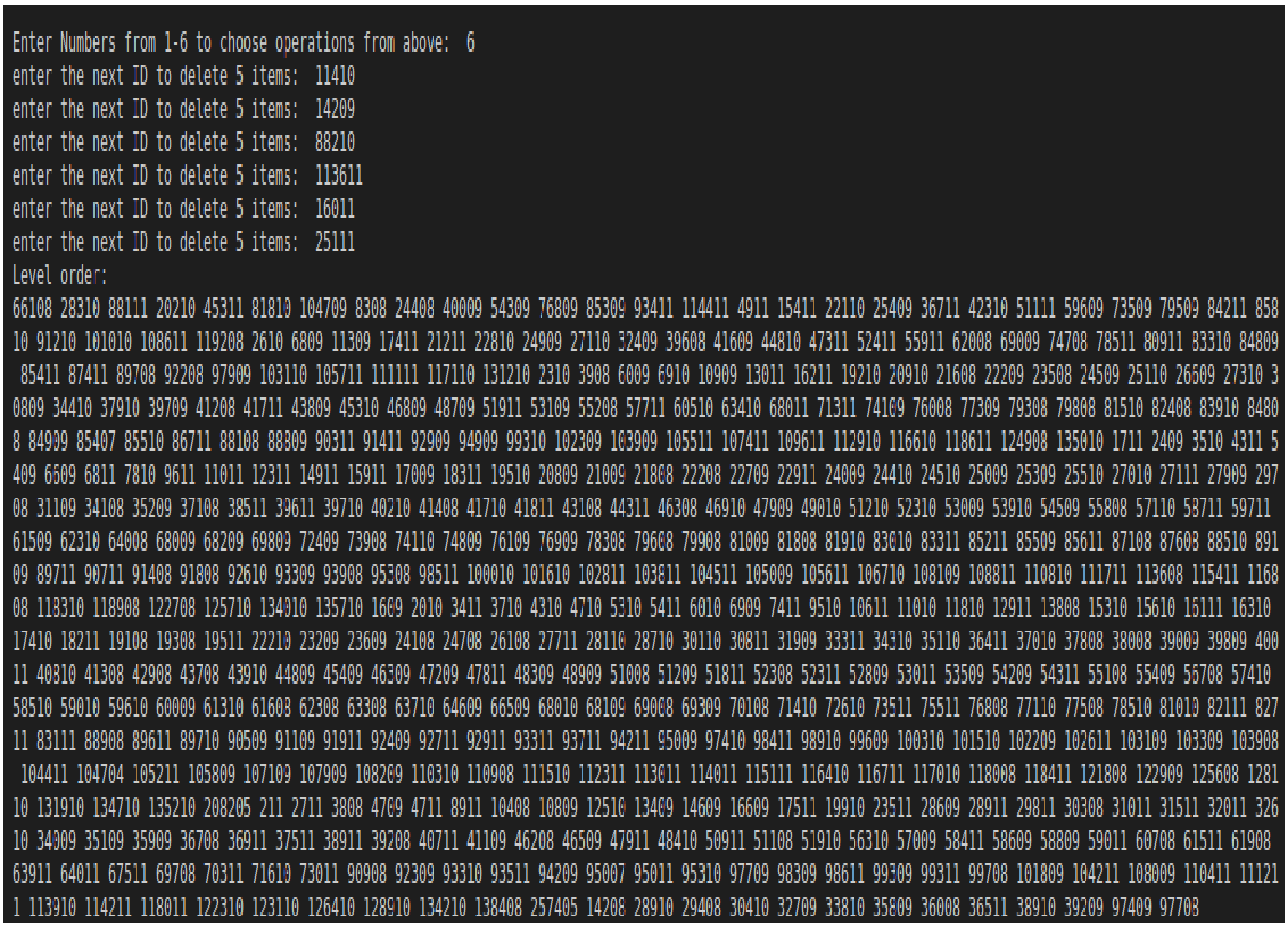
```
 We are going to insert the rest of 50 items into the RBtree:
  73509          James          Butt
  66108          Josephine      Darakjy
  51111          Art            Venere
  13011          Lenna          Paprocki
  1609           Donette        Foller
  15411          Simona         Morasca
  9510           Mitsue         Tollner
  20210          Leota          Dilliard
  22110          Sage           Wieser
  34410          Kris           Marrier
```

those are values from the bottom up to ID 85510



```
  85510          Ruthann        Keener
  39209          Joni           Breland
  97708          Vi      Rentfro
  114011         Colette        Kardas
  5310           Malcolm        Tromblay
  40810          Ryan           Harnos
  20809          Jess           Chaffins
  3808           Sharen         Bourbon
  28609          Nickolas       Juvera
  16609          Gary           Nunlee
  29408          Diane          Devreese
  103908         Roslyn         Chavous
  23511          Glory          Schieler
  51910          Rasheeda       Sayaphon
  46208          Alpha          Palaia
  81010          Refugia        Jacobos
  32011          Shawnda        Yori
  47209          Mona           Delasancha
  67511          Gilma          Liukko
  111211         Janey          Gabisi
  14609          Lili           Paskin
  85611          Loren          Asar
  8911           Dorothy        Chesterfield
  89109          Gail           Similton
  28910          Catalina       Tillotson
  23209          Lawrence       Lorens
  15610          Carlee         Boulter
  89710          Thaddeus       Ankeny
  82111          Jovita         Oles
  25111          Alesia         Hixenbaugh
  16011          Lai            Harabedian
  113611         Brittni        Gillaspie
  88210          Raylene        Kampa
  14209          Flo            Bookamer
  11410          Jani           Biddy
```

f) Remove 5 different items from the red black tree



```
Enter Numbers from 1-6 to choose operations from above:  6
enter the next ID to delete 5 items:  11410
enter the next ID to delete 5 items:  14209
enter the next ID to delete 5 items:  88210
enter the next ID to delete 5 items:  113611
enter the next ID to delete 5 items:  16011
enter the next ID to delete 5 items:  25111
Level order:
66108 28310 88111 20210 45311 81810 104709 8308 24408 40009 54309 76809 85309 93411 114411 4911 15411 22110 25409 36711 42310 51111 59609 73509 79509 84211 858
10 91210 101010 108611 119208 2610 6809 11309 17411 21211 22810 24909 27110 32409 39608 41609 44810 47311 52411 55911 62008 69009 74708 78511 80911 83310 84809
 85411 87411 89708 92208 97909 103110 105711 111111 117110 131210 2310 3908 6009 6910 10909 13011 16211 19210 20910 21608 22209 23508 24509 25110 26609 27310 3
0809 34410 37910 39709 41208 41711 43809 45310 46809 48709 51911 53109 55208 57711 60510 63410 68011 71311 74109 76008 77309 79308 79808 81510 82408 83910 8480
8 84909 85407 85510 86711 88108 88809 90311 91411 92909 94909 99310 102309 103909 105511 107411 109611 112910 116610 118611 124908 135010 1711 2409 3510 4311 5
409 6609 6811 7810 9611 11011 12311 14911 15911 17009 18311 19510 20809 21009 21808 22208 22709 22911 24009 24410 24510 25009 25309 25510 27010 27111 27909 297
08 31109 34108 35209 37108 38511 39611 39710 40210 41408 41710 41811 43108 44311 46308 46910 47909 49010 51210 52310 53009 53910 54509 55808 57110 58711 59711
61509 62310 64008 68009 68209 69809 72409 73908 74110 74809 76109 76909 78308 79608 79908 81009 81808 81910 83010 83311 85211 85509 85611 87108 87608 88510 891
09 89711 90711 91408 91808 92610 93309 93908 95308 98511 100010 101610 102811 103811 104511 105009 105611 106710 108109 108811 110810 111711 113608 115411 1168
08 118310 118908 122708 125710 134010 135710 1609 2010 3411 3710 4310 4710 5310 5411 6010 6909 7411 9510 10611 11010 11810 12911 13808 15310 15610 16111 16310
17410 18211 19108 19308 19511 22210 23209 23609 24108 24708 26108 27711 28110 28710 30110 30811 31909 33311 34310 35110 36411 37010 37808 38008 39009 39809 400
11 40810 41308 42908 43708 43910 44809 45409 46309 47209 47811 48309 48909 51008 51209 51811 52308 52311 52809 53011 53509 54209 54311 55108 55409 56708 57410
58510 59010 59610 60009 61310 61608 62308 63308 63710 64609 66509 68010 68109 69008 69309 70108 71410 72610 73511 75511 76808 77110 77508 78510 81010 82111 827
11 83111 88908 89611 89710 90509 91109 91911 92409 92711 92911 93311 93711 94211 95009 97410 98411 98910 99609 100310 101510 102209 102611 103109 103309 103908
 104411 104704 105211 105809 107109 107909 108209 110310 110908 111510 112311 113011 114011 115111 116410 116711 117010 118008 118411 121808 122909 125608 1281
10 131910 134710 135210 208205 211 2711 3808 4709 4711 8911 10408 10809 12510 13409 14609 16609 17511 19910 23511 28609 28911 29811 30308 31011 31511 32011 326
10 34009 35109 35909 36708 36911 37511 38911 39208 40711 41109 46208 46509 47911 48410 50911 51108 51910 56310 57009 58411 58609 58809 59011 60708 61511 61908
63911 64011 67511 69708 70311 71610 73011 90908 92309 93310 93511 94209 95007 95011 95310 97709 98309 98611 99309 99311 99708 101809 104211 108009 110411 11121
1 113910 114211 118011 122310 123110 126410 128910 134210 138408 257405 14208 28910 29408 30410 32709 33810 35809 36008 36511 38910 39209 97409 97708
```

Conclusion:

Red black tree data structure is better case for storing large files.