|  |  |  |  |
| --- | --- | --- | --- |
| МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ  КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ імені Тараса Шевченка  ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ  **Кафедра програмних систем і технологій**  Дисципліна  **«** **Структури даних, аналіз і алгоритми  комп'ютерної обробки інформації »**  **Лабораторна робота № 2** | | | |
| **Виконав:** | Фесак Андрій Віталійович | **Перевірив**: | Юрчук  Ірина Аркадіївна |
| Група | ІПЗ-12 | Дата перевірки |  |
| Форма навчання | денна | Оцінка |  |
| Спеціальність | 121 |
| 2022 | | | |

**Мета**: аналіз швидкодії алгоритмів роботи з деревами

**Умова задачі:**

Написати програму мовою C# з можливістю вибору різних алгоритмів пошуку. Продемонструвати роботу (ефективність, час виконання) програм на різних дерев (масив, лінійний зв’язаний список), з різними умовами, що забезпечують зменшення часу виконання. Навести аналіз отриманих результатів.

Реалізувати алгоритми:

• Вставка вузла в дерево

• Видалення вузла із дерева

• Перефарбування двох однакових вузлів братів у протилежний колір

• Лівий, правий повороти

• Splay алгоритм

**Аналіз задачі:**    
Нам потрібно проаналізувати роботу різних видів дерев . Для цього ми створимо статичний клас «*Search*», де реалізуємо наші алгоритми. У основній частині коду ми повинні застосувати клас «*Random*» та «*Stopwatch*»:

* *Random –* клас у мові c#, який дає змогу нам генерувати псевдовипадкові числа
* *Stopwatch –* клас у мові c#, який ми використаємо для підрахунку часу виконання алгоритму.

З цим базисом ми можемо перейти до частини підготовки для роботи:

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

**Структура основних вхідних та вихідних даних:**

Для справедливої оцінки роботи всі алгоритми повинні бути в однакових умовах і тому всі вони будуть опрацьовувати один і той самий набір даних. У цьому випадку цей набір значень буде використовуватися задля перевірки швидкодії алгоритмів

for (int i = 0; i < 1000; i++)  
{  
 tree0.Add(i);  
}

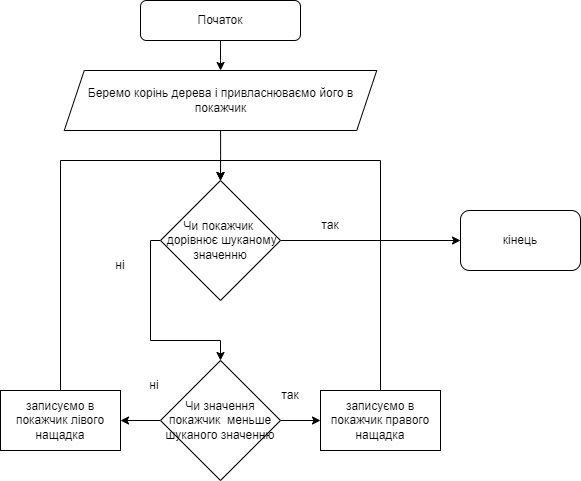
Ми створили 4 дерева, які заповнені однаковими даними:

* Розмір дерева:довільне натуральне н
* Набір вхідних даних: натуральні числа від 1 до 1000

Також ми задали значення для пошуку, яке також є однаковим для всіх алгоритмів та структур: одне з значень в дереві

**1. Бінарне дерево пошуку:**

**Алгоритм:**



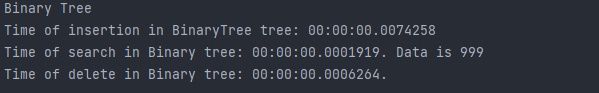
**Текст програми:**

public class BinaryTree  
{  
 public Node? top;  
  
 public BinaryTree()  
 {  
 top = null;  
 }  
  
 public BinaryTree(int value)  
 {  
 top = new Node(value);  
 }  
  
 public void Add(int value)  
 {  
 if (top==null)  
 {  
 top = new Node(value);  
 }  
 else  
 {  
 Node current = top;  
 while (current.Value!=value)  
 {  
 if (value<current.Value )  
 {  
 if (current.Left==null)  
 {  
 current.Left = new Node(current, value);  
 return;  
 }  
 else  
 {  
 current = current.Left;  
 }  
 }else  
 {  
 if (current.Right == null)  
 {  
 current.Right = new Node(current, value);  
 return;  
 }  
 else  
 {  
 current = current.Right;  
 }  
 }  
 }  
  
 throw new Exception("The value Already exist in tree");  
 }  
 }  
  
 public Node Find(int value)  
 {  
 Node current = top;  
 while (current != null && current.Value !=value)  
 {  
 if (current.Value<value)  
 {  
 current = current.Right;  
 }  
 else  
 {  
 current = current.Left;  
 }  
   
 }  
  
 if (current==null)  
 {  
 return null;  
 }  
  
 return current;  
  
 }  
  
 static public Node FindMinInNode(Node node)  
 {  
 Node temp = node;  
 while (temp.Left!=null)  
 {  
 temp = temp.Left;  
 }  
 return temp;  
 }  
 public Node FindMaxInNode(Node node)  
 {  
 Node temp = node;  
 while (temp.Left!=null)  
 {  
 temp = temp.Left;  
 }  
  
 return temp;  
 }  
  
 public void Delete(int value)  
 {  
 Node? current = Find(value);  
 if (current!= null)  
 {  
 if (current == top)  
 {  
 Node temp = FindMinInNode(top.Right);  
 temp.Left = top.Left;  
 top = top.Right;  
 }else if (current.Left !=null && current.Right != null)  
 {  
 Node MinRightNode = BinaryTree.FindMinInNode(current.Right);  
 Node Father = current.Father;  
 Node Right = current.Right;  
 Right.Father = Father;  
 MinRightNode.Left = current.Left;  
 current.Left.Father = MinRightNode;  
 current.Father.Left = current.Right;  
  
 }else if (current.Left !=null || current.Right != null)  
 {  
 if (current.Left==null)  
 {  
 current.Right.Father = current.Father;  
 current.Father.Left = current.Right;  
 }  
 else  
 {  
 current.Left.Father = current.Father;  
 current.Father.Left = current.Left;  
 }  
 }  
 else  
 {  
 if (current.Father.Left ==current)  
 {  
 current.Father.Left = null;  
 }  
 else  
 {  
 current.Father.Right = null;  
 }  
 }  
 }  
 else  
 {  
 throw new Exception("This Node isn`t exist");  
 }  
   
 }  
   
}

**Нода**

public class Node  
{  
 public Node? Father { get; set; }  
 public Node? Left { get; set; }  
 public Node? Right { get; set; }  
 public int Value { get; set; }  
  
 public Node(int value)  
 {  
 Father = null;  
 Left = null;  
 Right = null;  
 Value = value;  
 }  
  
 public Node(Node father,int value)  
 {  
 Father = father;  
 Left = null;  
 Right = null;  
 Value = value;  
 }  
}

**Набір тестів:**

На 1000 елементах: 

**Результат:**

Процес вставки елементів в звичайне бінарне дерево програє в швидкості іншим структурам, але перемагає в швидкості пошуку

**2. Пошук з бар'єром елемента масиву, що дорівнює заданому значенню.:**

**Текст програми:**

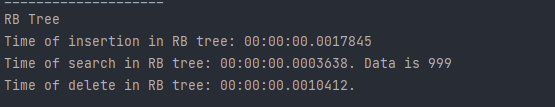
public class RedBlackTree  
 {  
 public TreeNode root;  
 public static TreeNode NIL;  
 int MaxLevel;  
 TreeNode current;  
  
 TreeNode Tree\_Maximum(TreeNode x)  
 {  
 while (x.right != NIL)  
 {  
 x = x.right;  
 }  
 return x;  
 }  
   
 TreeNode Tree\_Minimum(TreeNode x)  
 {  
 while (x.left != NIL)  
 {  
 x = x.left;  
 }  
 return x;  
 }  
   
 public RedBlackTree()   
 {  
 root = null;  
 NIL = new TreeNode(0, TreeColor.black,0);  
 MaxLevel = 0;  
 current = null;  
 }  
  
 public TreeNode Find(int number)  
 {  
 Search\_Node(root, number);  
 return current;  
 }  
  
 void Search\_Node(TreeNode temp, int number)  
 {  
 if (temp != NIL)  
 {  
 if (temp.data == number)  
 current= temp;  
 else  
 {  
 if (number < temp.data)  
   
 Search\_Node(temp.left, number);  
   
 else  
   
 Search\_Node(temp.right, number);  
   
   
 }  
 }  
 else current= null;  
 }  
  
   
 public void Delete(int number)  
 {  
 Search\_Node(root, number);  
 if (current!=null)  
 Delete(current);  
 }  
   
  
   
 void Transplant(TreeNode u, TreeNode v)  
 {  
 if (u.parent == NIL)  
 root = v;  
 else  
 {  
 if (u == u.parent.left)  
 u.parent.left = v;  
 else u.parent.right = v;  
 }  
 v.parent = u.parent;  
 int ColOfLvl = 1;  
 if (v!=NIL)  
 ColOfLvl = v.level - u.level;  
 v.level=u.level;  
   
 DecAllNodes(v.left);  
 DecAllNodes(v.right);  
   
 }  
   
 void Delete(TreeNode z)  
 {  
 bool m=false;  
 TreeNode y = z;  
 TreeNode x = NIL;  
 TreeColor y\_original\_color = y.color;  
 if (z.left == NIL)  
 {  
 x = z.right;  
 Transplant(z, z.right);  
 }  
 else  
 {  
 if (z.right == NIL)  
 {  
 x = z.left;  
 Transplant(z, z.left);  
 }  
 else  
 {  
 y = Tree\_Minimum(z.right);  
 y\_original\_color = y.color;  
 x = y.right;  
 if (y.parent == z) x.parent = y;  
 else  
 {  
 Transplant(y, y.right);  
 Transplant(z, y);  
 y.right.level += 1;  
 y.right = z.right;  
 y.right.parent = y;  
 m = true;  
 }  
 if(m==false)  
 Transplant(z, y);  
 y.left = z.left;  
 y.left.parent = y;  
 y.color = z.color;  
 }  
 if (y\_original\_color == TreeColor.black)  
 deleteFixup(x);  
 }  
 }  
   
 void deleteFixup(TreeNode x)  
 {  
 while (x != root && x.color == TreeColor.black)  
 {  
 if (x == x.parent.left)  
 {  
 TreeNode w = x.parent.right;  
 if (w.color == TreeColor.red)  
 {  
 w.color = TreeColor.black;  
 x.parent.color = TreeColor.red;  
 Left\_Rotate(x.parent);  
 w = x.parent.right;  
 }  
 if (w.left.color == TreeColor.black && w.right.color == TreeColor.black)  
 {  
 w.color = TreeColor.red;  
 x = x.parent;  
 }  
 else  
 {  
 if (w.right.color == TreeColor.black)  
 {  
 w.left.color = TreeColor.black;  
 w.color = TreeColor.red;  
 Right\_Rotate(w);  
 w = x.parent.right;  
 }  
 w.color = x.parent.color;  
 x.parent.color = TreeColor.black;  
 w.right.color = TreeColor.black;  
 Left\_Rotate(x.parent);  
 x = root;  
 }  
 }  
 else  
 {  
 TreeNode w = x.parent.left;  
 if (w.color == TreeColor.red)  
 {  
 w.color = TreeColor.black;  
 x.parent.color = TreeColor.red;  
 Right\_Rotate(x.parent);  
 w = x.parent.left;  
 }  
 if (w.right.color == TreeColor.black && w.left.color == TreeColor.black)  
 {  
 w.color = TreeColor.red;  
 x = x.parent;  
 }  
 else  
 {  
 if (w.left.color == TreeColor.black)  
 {  
 w.right.color = TreeColor.black;  
 w.color = TreeColor.red;  
 Left\_Rotate(w);  
 w = x.parent.left;  
 }  
 w.color = x.parent.color;  
 x.parent.color = TreeColor.black;  
 w.left.color = TreeColor.black;  
 Right\_Rotate(x.parent);  
 x = root;  
 }  
 }  
 }  
 x.color = TreeColor.black;  
 }  
  
 void IncAllNodes(TreeNode p)  
 {  
 if (p != NIL)  
 {  
 IncAllNodes(p.left);  
 p.level += 1;  
 IncAllNodes(p.right);  
 }  
 }  
 void DecAllNodes(TreeNode p)  
 {  
 if (p != NIL)  
 {  
 DecAllNodes(p.left);  
 p.level -= 1;  
 DecAllNodes(p.right);  
 }  
 }  
 void Left\_Rotate(TreeNode x)  
 {  
 TreeNode y = x.right;  
 x.right = y.left;  
 if (y.left != NIL)  
 y.left.parent = x;  
 y.parent = x.parent;  
 if (x.parent == NIL)  
 root = y;  
 else  
 {  
 if (x == x.parent.left)  
 x.parent.left = y;  
 else x.parent.right = y;  
 }  
 y.left = x;  
 x.parent = y;  
  
 x.level += 1;  
 y.level -= 1;  
 DecAllNodes(y.right);  
 IncAllNodes(x.left);  
  
 }  
   
 void Right\_Rotate(TreeNode y)  
 {  
 TreeNode x = y.left;  
 y.left = x.right;  
 if (x.right != NIL)  
 x.right.parent = y;  
 x.parent = y.parent;  
 if (y.parent == NIL)  
 root = x;  
 else  
 {  
 if (y == y.parent.right)  
 y.parent.right = x;  
 else y.parent.left = x;  
 }  
 x.right = y;  
 y.parent = x;  
  
 x.level -= 1;  
 y.level += 1;  
 DecAllNodes(x.left);  
 IncAllNodes(y.right);  
 }  
   
 void Insert\_FixUp(TreeNode z)  
 {  
 TreeNode y = NIL;  
 while (z.parent.color == TreeColor.red)  
 {  
 if (z.parent == z.parent.parent.left)  
 {  
 y = z.parent.parent.right;  
 if (y.color == TreeColor.red)  
 {  
 z.parent.color = TreeColor.black;  
 y.color = TreeColor.black;  
 z.parent.parent.color = TreeColor.red;  
 z = z.parent.parent;  
 }  
 else  
 {  
 if (z == z.parent.right)  
 {  
 z = z.parent;  
 Left\_Rotate(z);  
 }  
 z.parent.color = TreeColor.black;  
 z.parent.parent.color = TreeColor.red;  
 Right\_Rotate(z.parent.parent);  
 }  
 }  
 else  
 {  
 y = z.parent.parent.left;  
 if (y.color == TreeColor.red)  
 {  
 z.parent.color = TreeColor.black;  
 y.color = TreeColor.black;  
 z.parent.parent.color = TreeColor.red;  
 z = z.parent.parent;  
 }  
 else  
 {  
 if (z == z.parent.left)  
 {  
 z = z.parent;  
 Right\_Rotate(z);  
 }  
 z.parent.color = TreeColor.black;  
 z.parent.parent.color = TreeColor.red;  
 Left\_Rotate(z.parent.parent);  
   
  
 }  
 }  
 }  
 root.color = TreeColor.black;  
 }  
  
 private void checkMaxLevel(int newLevel)  
 {  
 MaxLevel = (newLevel > MaxLevel) ? newLevel : MaxLevel;  
 }  
  
 public void insert( int data) {  
 if (root == null) {  
 root = NIL;  
 root.parent = NIL;  
 root.left = NIL;  
 root.right = NIL;  
 root.color = TreeColor.black;  
 }  
 TreeNode z = new TreeNode(data, TreeColor.red,0);  
 TreeNode y = NIL;  
 TreeNode x = root;  
 while (x!=NIL) {  
 y = x;  
 if (z.data < x.data)  
 x = x.left;  
 else  
 {  
 if (z.data > x.data)  
 x = x.right;  
 else return;  
 }  
 }  
 z.parent = y;  
 if (y == NIL)  
 root = z;  
 else {  
 if (z.data<y.data)  
  
 y.left = z;  
 else y.right = z;  
 }  
 z.left = NIL;  
 z.right = NIL;  
 z.color = TreeColor.red;  
 if(z.parent!=NIL)  
 z.level = z.parent.level+1;  
 checkMaxLevel(z.level);  
 Insert\_FixUp(z);  
 }  
   
 }

**Нода**

public enum TreeColor { red, black };  
public class TreeNode  
{  
 public TreeNode left;  
 public TreeNode right;  
 public TreeNode parent;  
 public int data;  
 public TreeColor color;  
 public int level;  
 public bool visited;  
  
 public TreeNode( int data, TreeColor color,int lvl) {  
 this.data = data;  
 this.color = color;  
 left = right = null;  
 level = lvl;  
 visited = false;  
 }  
 public int get\_data(){  
 return data;  
 }  
 public TreeNode getAdjUnvisitedVertex()  
 {  
 if (this.left !=RedBlackTree.NIL)  
 {  
 if (!this.left.visited)  
 {  
 this.left.visited = true;  
 return this.left;  
 }  
 else  
 {  
 if (this.right != RedBlackTree.NIL)  
 {  
 if (!this.right.visited)  
 {  
 this.right.visited = true;  
 return this.right;  
 }  
 }  
 }  
 }  
 else  
 {  
 if (this.right != RedBlackTree.NIL)  
 {  
 if (!this.right.visited)  
 {  
 this.right.visited = true;  
 return this.right;  
 }  
 }  
 }  
 return RedBlackTree.NIL;  
 }  
}

**Набір тестів:**

На 1000 елементах:



**Результат:**

Червоне-чорне дерево показало найгірший результат серед всіх дерев. Це обумовлено вимушеністю зберігати в ноді дані про колір даних та пов’язаним з цим великої кількості непотрібних процесів задля збереження кольорового балансу

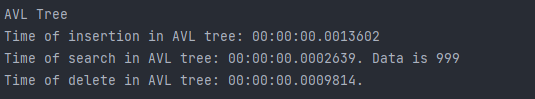
**3. АВЛ Дерево:**

**Текст програми:**

public class AvlTree<T> where T : IComparable<T>  
{  
 public AvlTreeNote<T> Root;   
  
 private bool \_isBalance;   
  
  
 public AvlTreeNote<T> Add(T key) => Root = Add(key, Root);  
   
 private AvlTreeNote<T> Add(T key, AvlTreeNote<T> node)  
 {  
 if (node == null)  
 {  
 node = new AvlTreeNote<T>(key, null, null);  
 }  
 else  
 {  
  
 if (key.CompareTo(node.Key) == 0) return null;  
  
 if (key.CompareTo(node.Key) < 0)  
 {  
  
 node.LChild = Add(key, node.LChild);  
  
 if (node.LChild == null) return node;  
   
 switch (node.Height)  
 {  
 case 1:  
 return LeftBalance(node);  
 case 0:  
 node.Height = \_isBalance ? 0 : 1;  
 break;  
 case -1:  
 node.Height = 0;  
 break;  
 }  
 }  
 else  
 {  
  
 node.RChild = Add(key, node.RChild);  
  
 if (node.RChild == null) return node;  
   
 switch (node.Height)  
 {  
 case 1:  
 node.Height = 0;  
 break;  
 case 0:  
 node.Height = \_isBalance ? 0 : -1;  
 break;  
 case -1:  
 return RightBalance(node);   
 }  
 }  
 }  
   
 \_isBalance = false;  
  
 return node;  
 }  
   
 private AvlTreeNote<T> LeftBalance(AvlTreeNote<T> node)  
 {  
 if (\_isBalance) return node;  
   
 var leftNode = node.LChild;  
  
 switch (leftNode.Height)  
 {  
 case 1:  
  
 node.Height = leftNode.Height = 0;  
  
 node = R\_Rotate(node);  
  
 break;  
  
 case -1:  
  
 node.Height = leftNode.Height = 0;  
  
 node.LChild = L\_Rotate(leftNode);  
  
 node = R\_Rotate(node);  
  
 break;  
 }  
  
 return node;  
 }  
  
 private AvlTreeNote<T> RightBalance(AvlTreeNote<T> node)  
 {  
 if (\_isBalance) return node;  
   
 var rightNode = node.RChild;  
  
 switch (rightNode.Height)  
 {  
 case -1:  
   
 node.Height = rightNode.Height = 0;  
   
 node = L\_Rotate(node);  
   
 break;  
   
 case 1:  
  
 node.Height = rightNode.Height = 0;  
  
 node.RChild = R\_Rotate(rightNode);  
   
 node = L\_Rotate(node);  
   
 break;  
 }  
  
 return node;  
 }  
   
 private AvlTreeNote<T> R\_Rotate(AvlTreeNote<T> node)  
 {  
 var temp = node.LChild;  
   
 node.LChild = temp.RChild;  
  
 temp.RChild = node;  
   
 \_isBalance = true;  
   
 return temp;  
 }  
   
 private AvlTreeNote<T> L\_Rotate(AvlTreeNote<T> node)  
 {  
 var temp = node.RChild;  
   
 node.RChild = temp.LChild;  
   
 temp.LChild = node;  
   
 \_isBalance = true;  
  
 return temp;  
 }  
   
 public AvlTreeNote<T> Find(T key) => Find(key, Root);  
   
 public AvlTreeNote<T> Find(T key,AvlTreeNote<T> node)  
 {  
 if (node == null) return null;  
   
 if (key.CompareTo(node.Key) < 0)  
 {  
 node = Find(key,node.LChild);  
 }  
 else if(key.CompareTo(node.Key)>0)  
 {  
 node = Find(key, node.RChild);  
 }  
   
 return node;  
 }  
  
  
 private AvlTreeNote<T> Move(AvlTreeNote<T> node, AvlTreeNote<T> findNode)  
 {  
 AvlTreeNote<T> moveNode;  
  
 if (findNode != null)  
 {  
 if (findNode.RChild != null)  
 {  
 moveNode = findNode.RChild;  
  
 findNode.RChild = null;  
 }  
 else  
 {  
 findNode.LChild = null;  
  
 moveNode = findNode;  
 }  
   
 if (node.LChild != moveNode) moveNode.LChild = node.LChild;  
  
 if (node.RChild != moveNode) moveNode.RChild = node.RChild;  
 }  
 else  
 {  
 moveNode = null;  
 }  
  
 node.LChild = null;  
  
 node.RChild = null;  
  
 node.Key = default(T);  
  
 node.Height = 0;  
  
 return moveNode;  
 }  
  
  
 public void Remove(T key) => Root = Remove(key, Root);  
  
 private AvlTreeNote<T> Remove(T key, AvlTreeNote<T> node)  
 {  
 if (node == null) return null;  
   
 if (key.CompareTo(node.Key) < 0)  
 {  
 if (node.LChild == null) return node;  
   
 node.LChild = Remove(key, node.LChild);  
   
 switch (node.Height)  
 {  
 case 1:  
 node.Height = 0;  
 break;  
 case 0:  
 node.Height = -1;  
 break;  
 case -1:  
  
 node.Height = 0;  
 return node.LChild == null ? RightBalance(node) : LeftBalance(node);  
 }  
 }  
 else if (key.CompareTo(node.Key) > 0)  
 {  
 if (node.RChild == null) return node;  
   
 node.RChild = Remove(key, node.RChild);  
  
 switch (node.Height)  
 {  
 case 1:  
  
 node.Height = 0;  
 return node.RChild == null ? LeftBalance(node) : RightBalance(node);  
 break;  
 case 0:  
 node.Height = 1;  
 break;  
 case -1:  
 node.Height = 0;  
 break;  
 }  
 }  
 else if (key.CompareTo(node.Key) == 0)  
 {  
 var findNode = Remove(key,node.LChild);  
  
 node = Move(node, findNode);  
 }  
   
 \_isBalance = false;  
  
 return node;  
 }  
}

**Набір тестів:**

На 1000 елементах:



**Результат:**

AVL дерево показав найкращий результат для вставки. Це обумовлено постійним балансом елементів, що робить його вийграшнім на фоні звичайного бінарного дерева та більш простим алгоритмом відносно чб дерева.

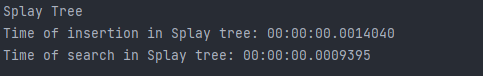
**4. Splay дерево:**

**Текст програми:**

public class SplayTree<T> : IEnumerable<T> where T : IComparable<T>, IComparable  
 {  
 private Node \_root;  
   
 public int Count { get; private set; }  
   
 class Node  
 {  
 public Node Left, Right;  
 public T Key;  
  
 public Node(T pKey)  
 {  
 Key = pKey;  
 Left = null;  
 Right = null;  
 }  
  
 public Node(Node pNode)  
 {  
 Key = pNode.Key;  
 Left = pNode.Left;  
 Right = pNode.Right;  
 }  
  
 public static bool operator true(Node pN)  
 {  
 return pN != null;  
 }  
  
 public static bool operator false(Node pN)  
 {  
 return pN == null;  
 }  
  
 public static bool operator !(Node pN)  
 {  
 return pN == null;  
 }  
 }  
  
  
 public SplayTree()  
 {  
 \_root = null;  
 Count = 0;  
 }  
  
 public SplayTree(T pRoot)  
 {  
 \_root = new Node(pRoot);  
 Count = 1;  
 }  
  
 public SplayTree(T[] pItems)  
 {  
 \_root = new Node(pItems[0]);  
 for (int i = 1; i < pItems.Length; i++)  
 {  
 Add(ref \_root, pItems[i]);  
 }  
 }  
   
 public bool Remove(T pKey)  
 {  
 Node n;  
 Node parent;  
 FindWithParent(\_root, pKey, out n, out parent);  
 if (!n) return false;  
  
 if (!n.Right && !n.Left)  
 {  
 Node next = SubtreeMax(n.Right);  
 n.Key = next.Key;  
   
 FindWithParent(n.Right, next.Key, out n, out parent);  
   
 parent.Right = null;  
 }  
 else   
 {  
 if (parent.Right != null && parent.Right.Key.CompareTo(pKey) == 0)  
 {  
 parent.Right = !n.Right ? n.Right : n.Left;  
 }  
 else  
 {  
 parent.Left = !n.Right ? n.Right : n.Left;  
 }  
 }  
  
 return true;  
 }  
  
   
 public bool Contains(T pKey)  
 {  
 Node currentNode = \_root;  
 Node previousNode = null;   
 if (!\_root) return false;  
 if (pKey.CompareTo(currentNode.Key) == 0) return true;  
  
 Node leftTree = null;  
 Node rightTree = null;  
   
 while (currentNode)  
 {  
   
 if (pKey.CompareTo(currentNode.Key) < 0)  
 {  
 if (!rightTree)  
 {  
 rightTree = new Node(currentNode);  
 rightTree.Left = null;  
 }  
   
 previousNode = currentNode;  
 currentNode = currentNode.Left;  
   
 if (currentNode)  
 {  
 if (pKey.CompareTo(currentNode.Key) > 0)  
 {  
 if (!leftTree)  
 {  
 leftTree = new Node(currentNode);  
 leftTree.Right = null;  
 }  
 else  
 {  
 Add(ref leftTree, currentNode.Key);  
 }  
 }  
 else   
 {  
 Add(ref rightTree, currentNode.Key);  
 }  
 }  
 }  
 else if (pKey.CompareTo(currentNode.Key) > 0)   
 {  
  
 if (!leftTree)  
 {  
 leftTree = new Node(currentNode);  
 leftTree.Right = null;  
 }  
   
 previousNode = currentNode;  
 currentNode = currentNode.Right;  
   
 if (currentNode)  
 {  
 if (pKey.CompareTo(currentNode.Key) > 0)  
 {  
 Add(ref leftTree, currentNode.Key);  
 }  
 else  
 {  
 if (!rightTree)  
 {  
 rightTree = new Node(currentNode);  
 rightTree.Left = null;  
 }  
 else  
 {  
 Add(ref rightTree, currentNode.Key);  
 }  
 }  
 }  
 }  
 else   
 {  
  
 if (currentNode.Left)  
 {  
 Attach(ref leftTree, ref currentNode.Left);  
 }  
  
 if (currentNode.Right)  
 {  
 Attach(ref rightTree, ref currentNode.Right);  
 }  
  
 Node n;  
 Find(pKey.CompareTo(previousNode.Key) < 0 ? leftTree : rightTree, previousNode.Key, out n);  
 if (!n || !n.Left) break;  
   
 if (n.Left.Key.CompareTo(pKey) == 0)  
 {  
 n.Left = null;  
 }  
 else  
 {  
 n.Right = null;  
 }  
  
 break;  
 }  
  
 }  
   
 if (currentNode)  
 {  
 \_root = currentNode;  
 \_root.Left = leftTree;  
 \_root.Right = rightTree;  
  
 return true;  
 }  
 else   
 {  
 leftTree = null;  
 rightTree = null;  
 return false;  
 }  
  
 }  
   
 public void Add(T pKey)  
 {  
 Node currentNode = \_root;  
 Node leftTree = null;  
 Node rightTree = null;  
   
 while (currentNode)  
 {  
 if (pKey.CompareTo(currentNode.Key) < 0)  
 {  
 if (!rightTree)  
 {  
 rightTree = new Node(currentNode);  
 rightTree.Left = null;  
 }  
   
 currentNode = currentNode.Left;  
   
 if (currentNode)  
 {  
 if (pKey.CompareTo(currentNode.Key) > 0)  
 {  
 if (!leftTree)  
 {  
 leftTree = new Node(currentNode);  
 leftTree.Right = null;  
 }  
 else  
 {  
 Add(ref leftTree, currentNode.Key);  
 }  
 }  
 else  
 {  
 Add(ref rightTree, currentNode.Key);  
 }  
 }  
 }  
 else  
 {  
 if (!leftTree)  
 {  
 leftTree = new Node(currentNode);  
 leftTree.Right = null;  
 }  
   
 currentNode = currentNode.Right;  
   
 if (currentNode)  
 {  
 if (pKey.CompareTo(currentNode.Key) > 0)  
 {  
 Add(ref leftTree, currentNode.Key);  
 }  
 else  
 {  
 if (!rightTree)  
 {  
 rightTree = new Node(currentNode);  
 rightTree.Left = null;  
 }  
 else  
 {  
 Add(ref rightTree, currentNode.Key);  
 }  
 }  
 }  
 }  
 }  
   
 \_root = new Node(pKey);  
 \_root.Left = leftTree;  
 \_root.Right = rightTree;  
  
 Count++;  
 }  
  
 public T Minimum()  
 {  
 return SubtreeMin(\_root).Key;  
 }  
   
 public T Maximum()  
 {  
 return SubtreeMax(\_root).Key;  
 }  
  
  
 private void Add(ref Node pNode, T pKey)  
 {  
 if (!pNode)  
 {  
 pNode = new Node(pKey);  
 }  
 else if (pKey.CompareTo(pNode.Key) < 0)  
 {  
 Add(ref pNode.Left, pKey);  
 }  
 else  
 {  
 Add(ref pNode.Right, pKey);  
 }  
 }  
   
 private void Attach(ref Node pNode, ref Node pAttachNode)  
 {  
 if (!pNode)  
 {  
 pNode = pAttachNode;  
 }  
 else if (pAttachNode.Key.CompareTo(pNode.Key) < 0)  
 {  
 Attach(ref pNode.Left, ref pAttachNode);  
 }  
 else  
 {  
 Attach(ref pNode.Right, ref pAttachNode);  
 }  
 }  
  
 private void Find(T pKey, out Node pNode)  
 {  
 Find(\_root, pKey, out pNode);  
 }  
  
 private void Find(Node pSubtree, T pKey, out Node pNode)  
 {  
 Node currentNode = pSubtree;  
 while (currentNode)  
 {  
 if (pKey.CompareTo(currentNode.Key) < 0)  
 {  
 currentNode = currentNode.Left;  
 }  
 else if (pKey.CompareTo(currentNode.Key) > 0)  
 {  
 currentNode = currentNode.Right;  
 }  
 else  
 {  
 pNode = currentNode;  
 return;  
 }  
 }  
  
 pNode = null;  
 }  
  
 private void FindWithParent(Node pRoot, T pKey, out Node pNode, out Node pParent)  
 {  
 Node currentNode = pRoot;  
 Node previousNode = null;  
 while (currentNode)  
 {  
 if (pKey.CompareTo(currentNode.Key) < 0)  
 {  
 previousNode = currentNode;  
 currentNode = currentNode.Left;  
 }  
 else if (pKey.CompareTo(currentNode.Key) > 0)  
 {  
 previousNode = currentNode;  
 currentNode = currentNode.Right;  
 }  
 else  
 {  
 pParent = previousNode;  
 pNode = currentNode;  
 return;  
 }  
 }  
  
 pNode = null;  
 pParent = null;  
 }  
  
 private Node SubtreeMin(Node pNode)  
 {  
 Node retval = pNode;  
 while (pNode.Left)  
 {  
 retval = pNode.Left;  
 }  
  
 return retval;  
 }  
  
 private Node SubtreeMax(Node pNode)  
 {  
 Node retval = pNode;  
 while (pNode.Right)  
 {  
 retval = pNode.Right;  
 }  
  
 return retval;  
 }  
   
 public IEnumerator<T> GetEnumerator()  
 {  
 return new SplayTreeEnumerator(\_root);  
 }  
  
 private IEnumerator GetEnumerator1()  
 {  
 return this.GetEnumerator();  
 }  
 IEnumerator IEnumerable.GetEnumerator()  
 {  
 return GetEnumerator1();  
 }  
   
 private class SplayTreeEnumerator : IEnumerator<T>  
 {  
 private List<Node> \_path;  
 private Node \_root;  
 private Node \_current;  
 private Node \_next;  
  
 public T Current  
 {  
 get { return \_current.Key; }  
 }  
  
 private object Current1  
 {  
 get { return Current; }  
 }  
 object IEnumerator.Current  
 {  
 get { return Current1; }  
 }  
  
 public SplayTreeEnumerator(Node pRoot)  
 {  
 \_path = new List<Node>();  
 \_root = pRoot;  
 \_current = pRoot;  
  
 \_path.Add(\_current);  
 var t = \_current.Left;  
 while (t)  
 {  
 \_path.Add(t);  
 t = t.Left;  
 }  
 \_next = \_path[\_path.Count - 1];  
 }  
  
 public bool MoveNext()  
 {  
 \_current = \_next;  
 \_path.RemoveAt(\_path.Count - 1);  
  
 if (\_path.Count > 0)  
 {  
 \_next = \_path[\_path.Count - 1];  
 }  
 else if (\_next.Right)  
 {  
 \_path.Add(\_next.Right);  
  
 var t = \_next.Right.Left;  
 while (t)  
 {  
 \_path.Add(t);  
 t = t.Left;  
 }  
  
 \_next = \_path[\_path.Count - 1];  
 }  
 else  
 {  
 return false;  
 }  
  
 return true;  
 }  
  
 public void Reset()  
 {  
 \_path = new List<Node>();  
 \_current = \_root;  
  
 \_path.Add(\_current);  
 var t = \_current.Left;  
 while (t)  
 {  
 \_path.Add(t);  
 t = t.Left;  
 }  
 \_next = \_path[\_path.Count - 1];  
 }  
  
 public void Dispose()  
 {  
 \_path = null;  
 }  
 }  
 }

**Набір тестів:**

На 1000 елементах:



**Висновок:**

Бінарні дерева є найкращими структурами для збереження ключів в базі даних. Їхня структура пошуку дозволяє проводити основні операції властиві для блоку чисельних даних: сортування, знаходження мінімальних/максимальних елементів.   
  
Для цього існує декілька модифікацій для цієї структури:

* RB-Tree
* AVL-Tree
* Splay Tree

На основі лабораторної робити ми можемо зробити деякі висновки стосовно цих модифікацій:

* AVL Tree – ця структура гарно себе показує у випадку, якщо ваша база даних має властивість дуже часто змінюватися( до прикладу застосунок мессенджер)
* Splay-Tree – ця структура забезпечує оптимізований пошук часто шуканих данів, підходить до систем виду магазинів
* RB-Tree – сильно навантажена структура за рахунок свого алгоритму тому не бажана для використання
* Просто Бінарне дерево – оптимальний усереджений варіант

З цього ми робимо висновок, що різні структури надбудови над бінарним деревом пошуку потрібно використовувати у різних ситуаціях