**Міністерство освіти і науки України**
**Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського"**
**Факультет інформатики та обчислювальної техніки**

**Кафедра інформатики та програмної інженерії**

# Звіт

з лабораторної роботи № 3 з дисципліни
«Проектування алгоритмів»

„ **Проектування структур даних**”

**Виконав(ла)** *ІП-Мочалов Дмитро Юрійович* _____
(шифр, прізвище, ім'я, по батькові)

**Перевірив** *Головченко М.Н.* _____
(прізвище, ім'я, по батькові)

Київ 2022

Зміст

# 1 Мета лабораторної роботи

Мета роботи – вивчити основні підходи проектування та обробки складних структур даних.

# 2 Завдання

Відповідно до варіанту (таблиця 2.1), записати алгоритми пошуку, додавання, видалення і редагування запису в структурі даних за допомогою псевдокоду (чи іншого способу по вибору).

Записати часову складність пошуку в структурі в асимптотичних оцінках.

Виконати програмну реалізацію невеликої СУБД з графічним (не консольним) інтерфейсом користувача (дані БД мають зберігатися на ПЗП), з функціями пошуку (алгоритм пошуку у вузлі структури згідно варіанту таблиця 2.1, за необхідності), додавання, видалення та редагування записів (запис складається із ключа і даних, ключі унікальні і цілочисельні, даних може бути декілька полів для одного ключа, але достатньо одного рядка фіксованої довжини). Для зберігання даних використовувати структуру даних згідно варіанту (таблиця 2.1).

Заповнити базу випадковими значеннями до 10000 і зафіксувати середнє (із 10-15 пошуків) число порівнянь для знаходження запису по ключу.

Зробити висновок з лабораторної роботи.

Таблиця 2.1 – Варіанти алгоритмів

| **№** | **Структура даних** |
|---|---|
| 1 | Файли з щільним індексом з перебудовою індексної області, бінарний пошук |
| 2 | Файли з щільним індексом з областю переповнення, бінарний пошук |
| 3 | Файли з не щільним індексом з перебудовою індексної області, бінарний пошук |
| 4 | Файли з не щільним індексом з областю переповнення, бінарний пошук |
| 5 | АВЛ-дерево |
| 6 | Червоно-чорне дерево |
| 7 | B-дерево t=10, бінарний пошук |

| | |
|---|---|
| 8 | B-дерево t=25, бінарний пошук |
| 9 | B-дерево t=50, бінарний пошук |
| 10 | B-дерево t=100, бінарний пошук |
| 11 | Файли з щільним індексом з перебудовою індексної області, однорідний бінарний пошук |
| 12 | Файли з щільним індексом з областю переповнення, однорідний бінарний пошук |
| 13 | Файли з не щільним індексом з перебудовою індексної області, однорідний бінарний пошук |
| 14 | Файли з не щільним індексом з областю переповнення, однорідний бінарний пошук |
| 15 | АВЛ-дерево |
| 16 | Червоно-чорне дерево |
| 17 | B-дерево t=10, однорідний бінарний пошук |
| 18 | B-дерево t=25, однорідний бінарний пошук |
| 19 | B-дерево t=50, однорідний бінарний пошук |
| 20 | B-дерево t=100, однорідний бінарний пошук |
| 21 | Файли з щільним індексом з перебудовою індексної області, метод Шарра |
| 22 | Файли з щільним індексом з областю переповнення, метод Шарра |
| 23 | Файли з не щільним індексом з перебудовою індексної області, метод Шарра |
| 24 | Файли з не щільним індексом з областю переповнення, метод Шарра |
| 25 | АВЛ-дерево |
| 26 | Червоно-чорне дерево |
| 27 | B-дерево t=10, метод Шарра |
| 28 | B-дерево t=25, метод Шарра |
| 29 | B-дерево t=50, метод Шарра |
| 30 | B-дерево t=100, метод Шарра |

| | |
|---|---|
| 31 | АВЛ-дерево |
| 32 | Червоно-чорне дерево |
| 33 | B-дерево t=250, бінарний пошук |
| 34 | B-дерево t=250, однорідний бінарний пошук |
| 35 | B-дерево t=250, метод Шарра |

# 3 Виконання

## 3.1 Псевдокод алгоритмів

```
B_TREE_SPLIT_CHILD(x, i, y)
 1  z ← ALLOCATE_NODE()
 2  leaf[z] ← leaf[y]
 3  n[z] ← t − 1
 4  for j ← 1 to t − 1
 5        do key_j[z] ← key_{j+t}[y]
 6  if not leaf[y]
 7     then for j ← 1 to t
 8               do c_j[z] ← c_{j+t}[y]
 9  n[y] ← t − 1
10  for j ← n[x] + 1 downto i + 1
11        do c_{j+1}[x] ← c_j[x]
12  c_{i+1}[x] ← z
13  for j ← n[x] downto i
14        do key_{j+1}[x] ← key_j[x]
15  key_i[x] ← key_t[y]
16  n[x] ← n[x] + 1
```

```
B_TREE_INSERT_NONFULL(x, k)
 1  i ← n[x]
 2  if leaf[x]
 3     then while i ≥ 1 и k < key_i[x]
 4               do key_{i+1}[x] ← key_i[x]
 5                  i ← i − 1
 6          key_{i+1}[x] ← k
 7          n[x] ← n[x] + 1
 8          DISK_WRITE(x)
 9     else while i ≥ 1 и k < key_i[x]
10               do i ← i − 1
11          i ← i + 1
12          DISK_READ(c_i[x])
13          if n[c_i[x]] = 2t − 1
14            then B_TREE_SPLIT_CHILD(x, i, c_i[x])
15                 if k > key_i[x]
16                   then i ← i + 1
17          B_TREE_INSERT_NONFULL(c_i[x], k)

B_TREE_INSERT(T, k)
 1  r ← root[T]
 2  if n[r] = 2t − 1
 3     then s ← ALLOCATE_NODE()
 4          root[T] ← s
 5          leaf[s] ← FALSE
 6          n[s] ← 0
 7          c_1[s] ← r
 8          B_TREE_SPLIT_CHILD(s, 1, r)
 9          B_TREE_INSERT_NONFULL(s, k)
10     else B_TREE_INSERT_NONFULL(r, k)
```

```
BtreeSearch(int key){
 Node = SearchNode(TreeRoot,key)
 if (Node exists){
   BinarySearch(Node,key)
 }end if
 else
```

```
 return null
 end else
}

SearchNode(Node NodeForSearch,int key){
 if (key in NodeForSearch.Keys){
   return NodeForSearch
 }end if
 if(NodeForSearch is Leaf){
   return null
 }end if
 nextNode = NodeForSearch.FindChildForKey(key)
 return SearchNode(nextNode,key);
}

BinarySearch(NodeKeys keys, key){
 high = keys.length - 1
 low = 0
 while(low <=high){
   mid = Floor(low + (high-low)/2)
   if(keys[mid] == key){
     return keys[mid]
   }end if
   else if(keys[mid] < key){
     low = mid +1 1
   }end else if
   else if(keys[mid] > key){
     high = mid - 1
   }
 }end while
```

```
 return null
}
DeleteNode(int key){
 Node = SearchNode(TreeRoot, key)
 if(Node.isLeaf()){
   RemoveNodeFromLeaf(Node,key)
 }end if
 else{
   RemoveFromNonLeaf(Node,key)
 }end else
}

RemoveFromLeaf(Node node,int key){
 node.Remove(key)
 RestoreTreeProperty(node)
}

RemoveFromNonLeaf(Node node, int key){
 leftChild = node.FindChildForKey(key)
 if(leftChild.KeysLength > t-1){
   predecessor = GetPredecessor(leftChild)
   predecessorKey = predecessor.GetLastKey()
   node.ReplaceKeyByAnotherKey(key,predecessorKey)
   RemoveNodeFromLeaf(predecessor,predecessprKey)
 }end if
 else{
   rightChild = leftChild.rightSibling
   successor = GetSuccessor(rightChild)
   successorKey = successor.GetFirstKey()
   node.ReplaceKeyByAnotherKey(key,successorKey)
```

```
   RemoveNodeFromLeaf(successor, successorKey)
 }
}

RestoreTreeProperty(Node node){
 if(Node.KeysLength < t -1){
   if(node.isRoot()){
     if(node.isEmpty() && node.ChildrenCount > 0){
       TreeRoot = node.Children[0]
       node.Children.Remove(TreeRoot)
     }end if
     else if(node.isEmpty()){
       TreeRoot = null
     }end else if
   }else(not BorrowLeft(node) and not BorrowRight(node)){
     Merge(node)
     RestoreTreeProperty(node.Parrent)
   }end else
 }end if
}
GetPredecessor(Node node){
 while(!node.isLeaf()){
   node = node.GetLastChild()
 }end while
 return node
}

GetSuccessor(Node node){
 while(!node.IsLeaf()){
   node = node.GetFirstChild()
```

```
 }end while
 return node
}

BorrowLeft(Node node){
 left = node.LeftSibling
 if(left not null and left.KeysLength > t-1){
   siblingKey = left.ExtractLastKey()
   siblingChild = left.ExtractLastChild()
   parentKey = node.parent.keyByChild(left)
   node.parent.reReplaceKeyByAnotherKey(parentKey,siblingKey)
   node.keys.AddToBegin(parentKey)
   node.Children.AddToBegin(siblingChild)
   return  true
 }end if
 return false
}

BorrowLeft(Node node){
 right = node.RightSibling
 if(right not null and right.KeysLength > t-1){
   siblingKey = right.ExtractFirstKey()
   siblingChild = right.ExtractFirstChild()
   parentKey = node.parent.keyByChild(right)
   node.parent.ReplaceKeyByAnotherKey(parentKey,siblingKey)
   node.keys.AddToBegin(parentKey)
   node.Children.AddToBegin(siblingChild)
   return  true
 }end if
 return false
```

```
}

Merge(Node node){
 left = node.LeftSibling
 parent = node.parent
 if(left != null){
   parentKey = parent.KeyByChild(left)
   parent.removeKey(parentKey)
   node.children.AddToBegin(left.Children)
   parent.removeChild(left)
   node.keys.AddToBegin(left.keys + parentKey)
 }end if
 else{
   right = node.rightSibling
   parentKey = parent.KeyByChild(right)
   node.children.AddToEnd(right.children)
   parent.removeChild(right)
   node.keys.AddToEnd(parentKey + right.keys)
 }end else
}
```

### 3.2 Часова складність пошуку

O(t*logn), де n кількість вузлів в дереві,
t параметр дерева

### 3.3 Програмна реалізація

#### 3.3.1 Вихідний код

```
public class Node
{
```

```
public int degree;
public List<NodeValue> NodeValues { get; set; }
public List<Node> Children { get; set; }
public Node? Parent { get; set; }
public Node(int degree)
{
    this.degree = degree;
    NodeValues = new List<NodeValue>(this.degree);
    Children = new List<Node>(this.degree);
    foreach (var child in Children)
    {
        child.Parent = this;
    }
}
public bool IsLeaf
{
    get
    {
        return !this.Children.Any();
    }
}
public bool HasReachedMaxCountOfKeys
{
    get { return this.NodeValues.Count == ((this.degree * 2) - 1);}
}
public int Find(int id, ref int countOfComparsion)
{
    for (int i = 0; i < this.NodeValues.Count; i++)
    {
        countOfComparsion++;
```

```
        if (this.NodeValues[i].NodeValueId == id)
        {
            return i;
        }
    }
    return -1;
}

public Node FindChildForKey(int key)
{
    for (int i = 0; i < NodeValues.Count; i++)
    {
        if (NodeValues[i].NodeValueId > key)
        {
            return Children[i];
        }
    }
    return Children[^1];
}
public NodeValue FindKeyByChild(Node node)
{
    var index = Math.Min(NodeValues.Count - 1, Children.IndexOf(node));
    return NodeValues[index];
}
public NodeValue ExtractLastKey()
{
    var node = NodeValues[^1];
    NodeValues.Remove(node);
    return node;
}
```

```
public NodeValue ExtractFirstKey()
{
    var node = NodeValues[0];
    NodeValues.Remove(node);
    return node;
}
public Node ExtractLastChild()
{
    var node = Children[^1];
    Children.Remove(node);
    return node;
}
public Node ExtractFirstChild()
{
    var node = Children[0];
    Children.Remove(node);
    return node;
}
private int IndexOfValueInChildren
{
    get
    {
        if (Parent == null) return -1;
        else
        {
            return Parent.Children.IndexOf(this);
        }
    }
}
public Node? leftSibling
```

```
        {
            get
            {
                if (this.Parent == null || IndexOfValueInChildren == 0) return null;
                return Parent.Children[IndexOfValueInChildren - 1];
            }
        }
        public Node? rightSibling
        {
            get
            {
                        if (this.Parent == null || IndexOfValueInChildren ==
Parent.Children.Count - 1) return null;
                return Parent.Children[IndexOfValueInChildren + 1];
            }
        }

        public void ReplaceValueByKey(int key, NodeValue value)
        {
            int index = NodeValues.FindIndex(u => u.NodeValueId == key);
            NodeValues[index] = value;
        }
    }

    public class BTree
    {
        private int Degree { get; set; } = 50;
        public Node Root { get; set; } = new (50);

        public BTree(IServiceScopeFactory _serviceScopeFactory)
```

```
        {
            using (var scope = _serviceScopeFactory.CreateScope())
            {
                                                ApplicationDbContext dbContext =
scope.ServiceProvider.GetService<ApplicationDbContext>();
                if (dbContext.NodeValues.Any())
                {
                    foreach (var node in dbContext.NodeValues)
                    {
                        BTreeInsert(node);
                    }
                }
            }
        }

        public NodeValue? BTreeSearch(int key,ref int countOfComparsion)
        {
            var node = SearchNode(Root, key, ref countOfComparsion);
            if (node is null) return null;
            return BinarySearch(node.NodeValues, key, ref countOfComparsion);
        }


        private Node? SearchNode(Node node, int key, ref int countOfComparsion
)
        {
            if (node.Find(key,ref countOfComparsion) != -1) return node;
            if (node.IsLeaf) return null;
            var nextNode = node.FindChildForKey(key);
            return SearchNode(nextNode, key, ref countOfComparsion);
```

```
        }

        private NodeValue? BinarySearch(List<NodeValue> nodeValues, int key,
ref int countOfComparsion)
        {
            int high = nodeValues.Count-1;
            int low = 0;
            while (low <= high)
            {
                var mid = (int)Math.Floor((double)(low + (high - low) / 2));

                if (nodeValues[mid].NodeValueId == key)
                {
                    countOfComparsion++;
                    return nodeValues[mid];
                }
                if (nodeValues[mid].NodeValueId < key)
                {
                    countOfComparsion++;
                    low = mid + 1;
                }
                else if (nodeValues[mid].NodeValueId > key)
                {
                    countOfComparsion++;
                    high = mid - 1;
                }
            }
            return null;
        }
```

```
        public void BTreeInsert(NodeValue node)
        {
            if (this.Root.HasReachedMaxCountOfKeys)
            {
                Node oldRoot = this.Root;
                this.Root = new Node(this.Degree);
                this.Root.Children.Add(oldRoot);
                    this.Root.Children[this.Root.Children.IndexOf(oldRoot)].Parent =
this.Root;
                this.SplitChild(this.Root,0,oldRoot);
                this.InsertNotFull(this.Root,node.NodeValueId,node.Value);
            }
            else
            {
                this.InsertNotFull(this.Root,node.NodeValueId,node.Value);
            }
        }

        private void SplitChild(Node parent,int nodeIdToSplit,Node nodeForSplit)
        {
            Node newNode = new Node(this.Degree);

parent.NodeValues.Insert(nodeIdToSplit,nodeForSplit.NodeValues[this.Degree -
1]);
            parent.Children.Insert(nodeIdToSplit + 1,newNode);
            newNode.Parent = parent;

newNode.NodeValues.AddRange(nodeForSplit.NodeValues.GetRange(this.Degree
,this.Degree -1));
```

```
            nodeForSplit.NodeValues.RemoveRange(this.Degree-1,this.Degree);
            if (!nodeForSplit.IsLeaf)
            {

newNode.Children.AddRange(nodeForSplit.Children.GetRange(this.Degree,this.D
egree));
                foreach (var child in newNode.Children)
                {
                   child.Parent = newNode;
                }
                nodeForSplit.Children.RemoveRange(this.Degree,this.Degree);
            }
        }

        private void InsertNotFull(Node node, int id, string value)
        {
                    int indexForInsert = node.NodeValues.TakeWhile(node =>
id.CompareTo(node.NodeValueId) >= 0).Count();
            if (node.IsLeaf)
            {
                                        node.NodeValues.Insert(indexForInsert,new
NodeValue(){NodeValueId = id,Value = value});
                return;
            }

            Node child = node.Children[indexForInsert];
            child.Parent = node;
            if (child.HasReachedMaxCountOfKeys)
            {
                SplitChild(node,indexForInsert,child);
```

```
                if (id.CompareTo(node.NodeValues[indexForInsert].NodeValueId) >
0) indexForInsert++;
            }
            InsertNotFull(node.Children[indexForInsert],id,value);
        }

        public List<NodeValue> ToList()
        {
            var nodes = new List<NodeValue>();
            ToList(this.Root, nodes);
            return nodes;
        }

        private void ToList(Node node, List<NodeValue> nodes)
        {
            int i = 0;
            for (i = 0; i < node.NodeValues.Count; i++)
            {
                if (!node.IsLeaf)
                {
                    ToList(node.Children[i], nodes);
                }
                nodes.Add(node.NodeValues[i]);
            }

            if (!node.IsLeaf)
            {
                ToList(node.Children[i],nodes);
            }
        }
```

```
        public void DeleteNode(int key)
        {
            int countOfComparsion = 0;
            var node = SearchNode(Root, key,ref countOfComparsion);
            if (node.IsLeaf)
            {
                RemoveNodeFromLeaf(node,key);
            }
            else
            {
                RemoveNodeFromNonLeaf(node,key);
            }

        }

        private void RemoveNodeFromLeaf(Node node, int key)
        {
            int countOfComparsion = 0;
                var nodeValueForRemove = BinarySearch(node.NodeValues, key,ref
countOfComparsion);
            node.NodeValues.Remove(nodeValueForRemove);
            RestorePropertyDelete(node);
        }

        private void RemoveNodeFromNonLeaf(Node node,int key)
        {
            var leftChild = node.FindChildForKey(key);
            if (leftChild.NodeValues.Count > Degree - 1)
            {
```

```
            var predecessor = GetPredecessor(leftChild);
            var predecessorValue = predecessor.NodeValues[^1];
            node.ReplaceValueByKey(key,predecessorValue);
            RemoveNodeFromLeaf(predecessor,predecessorValue.NodeValueId);
        }
        else
        {
            var rightChild = leftChild.rightSibling;
            var successor = GetSuccessor(rightChild);
            var successorKey = successor.NodeValues[0];
            node.ReplaceValueByKey(key,successorKey);
            RemoveNodeFromLeaf(successor,successorKey.NodeValueId);
        }
    }
    private void RestorePropertyDelete(Node node)
    {
        if (node.NodeValues.Count < Degree - 1)
        {
            if (node == Root)
            {
                if (node.NodeValues.Count == 0 && node.Children.Count > 0)
                {
                    Root = node.Children[0];
                    node.Children.Remove(Root);
                }
                else if (node.NodeValues.Count == 0)
                {
                    Root = null;
                }
            }
```

```
        else if (!BorrowLeft(node) && !BorrowRight(node))
        {
            Merge(node);
            RestorePropertyDelete(node.Parent);
        }
    }
}

private Node GetPredecessor(Node node)
{
    while (!node.IsLeaf)
    {
        node = node.Children[^1];
    }
    return node;
}

private Node GetSuccessor(Node node)
{
    while (!node.IsLeaf)
    {
        node = node.Children[0];
    }
    return node;
}

private bool BorrowLeft(Node node)
{
    var left = node.leftSibling;
    var parent = node.Parent;
```

```
    if (left != null && left.NodeValues.Count > Degree - 1)
    {
        var sibKey = left.ExtractLastKey();
        var parentKey = parent.FindKeyByChild(left);
        parent.ReplaceValueByKey(parentKey.NodeValueId,sibKey);
        node.NodeValues.Insert(0,parentKey);
        if (!left.IsLeaf)
        {
            var sibChild = left.ExtractLastChild();
            node.Children.Insert(0, sibChild);
            node.Children[0].Parent = node;
        }

        return true;
    }
    return false;
}
private bool BorrowRight(Node node)
{
    var right = node.rightSibling;
    var parent = node.Parent;
    if (right != null && right.NodeValues.Count > Degree - 1)
    {
        var sibKey = right.ExtractFirstKey();
        var parentKey = parent.FindKeyByChild(right);
        parent.ReplaceValueByKey(parentKey.NodeValueId,sibKey);
        node.NodeValues.Add(parentKey);
        if (!right.IsLeaf)
        {
            var sibChild = right.ExtractFirstChild();
```

```
            node.Children.Add(sibChild);
            node.Children[^1].Parent = node;
        }

        return true;
    }
    return false;
}
private void Merge(Node node)
{
    var left = node.leftSibling;
    var parent = node.Parent;
    if (left != null)
    {
        var parentKey = parent.FindKeyByChild(left);
        parent.NodeValues.Remove(parentKey);
        node.Children.InsertRange(0,left.Children);
        for (int i = 0; i < left.Children.Count; i++)
        {
            node.Children[i].Parent = node;
        }
        parent.Children.Remove(left);
        node.NodeValues.Insert(0,parentKey);
        node.NodeValues.InsertRange(0,left.NodeValues);
    }
    else
    {
        var right = node.rightSibling;
        var parentKey = parent.FindKeyByChild(node);
        parent.NodeValues.Remove(parentKey);
```

```
            node.Children.AddRange(right.Children);
            foreach (var child in node.Children)
            {
               child.Parent = node;
            }

            parent.Children.Remove(right);
            node.NodeValues.Add(parentKey);
            node.NodeValues.AddRange(right.NodeValues);
         }
      }

      public void Edit(NodeValue nodeValue)
      {
         int countOfComparsion = 0;
                  var node = SearchNode(Root, nodeValue.NodeValueId, ref
countOfComparsion);
                     var nodeForEdit = BinarySearch(node.NodeValues,
nodeValue.NodeValueId,ref countOfComparsion);
         if (nodeForEdit != null) nodeForEdit.Value = nodeValue.Value;
      }
   }
```

### 3.3.2 Приклади роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми для додавання і пошуку запису.

| Id | Value | | |
|---|---|---|---|
| 1 | bf7ff3ce-6a28-421b-a6d8-63a572b4d345 | Edit | Remove |
| 2 | 2ac164d1-ca38-432f-8fde-34bd2882882d | Edit | Remove |
| 3 | fa3e976e-2522-4cef-bc4e-9b008a98d0bc | Edit | Remove |
| 4 | bc0ad059-ab31-465f-8e8a-f26ba82079f6 | Edit | Remove |
| 5 | f00768f0-69e2-4ef1-a1b8-19e5ec93cacc | Edit | Remove |
| 6 | b643c9df-5948-445a-b78f-71f1bb468f9a | Edit | Remove |
| 7 | 3af72066-13e8-461f-a7ff-cd9a7238172f | Edit | Remove |
| 8 | 9ea477c6-f12a-415b-b1fd-23c84cfeb151 | Edit | Remove |
| 9 | 58b21d12-7abb-41a4-a6a5-5cb5f4c9aacf | Edit | Remove |
| 10 | 35fe9db4-fde8-4db3-80d4-fd86a5e93bc6 | Edit | Remove |
| 12 | d7169e27-bc97-4f45-a7d4-5b2d4406adca | Edit | Remove |
| 13 | d52de178-c16e-457d-a8ae-b6a674016244 | Edit | Remove |
| 14 | 2799edf2-fb3e-472f-900b-dfa279ce8147 | Edit | Remove |
| 15 | aabbc2f4-3e7b-4965-bc62-159a8a46d226 | Edit | Remove |

Id

11

Value

mynewvalue

Submit

| Id | Value | | |
|---|---|---|---|
| 1 | bf7ff3ce-6a28-421b-a6d8-63a572b4d345 | Edit | Remove |
| 2 | 2ac164d1-ca38-432f-8fde-34bd2882882d | Edit | Remove |
| 3 | fa3e976e-2522-4cef-bc4e-9b008a98d0bc | Edit | Remove |
| 4 | bc0ad059-ab31-465f-8e8a-f26ba82079f6 | Edit | Remove |
| 5 | f00768f0-69e2-4ef1-a1b8-19e5ec93cacc | Edit | Remove |
| 6 | b643c9df-5948-445a-b78f-71f1bb468f9a | Edit | Remove |
| 7 | 3af72066-13e8-461f-a7ff-cd9a7238172f | Edit | Remove |
| 8 | 9ea477c6-f12a-415b-b1fd-23c84cfeb151 | Edit | Remove |
| 9 | 58b21d12-7abb-41a4-a6a5-5cb5f4c9aacf | Edit | Remove |
| 10 | 35fe9db4-fde8-4db3-80d4-fd86a5e93bc6 | Edit | Remove |
| 11 | mynewvalue | Edit | Remove |
| 12 | d7169e27-bc97-4f45-a7d4-5b2d4406adca | Edit | Remove |
| 13 | d52de178-c16e-457d-a8ae-b6a674016244 | Edit | Remove |
| 14 | 2799edf2-fb3e-472f-900b-dfa279ce8147 | Edit | Remove |

Рисунок 3.1 –Додавання запису


Рисунок 3.2 – Пошук запису

## 3.4 Тестування алгоритму

### 3.4.1 Часові характеристики оцінювання

В таблиці 3.1 наведено кількість порівнянь для 15 спроб пошуку запису по ключу.

Таблиця 3.1 – Число порівнянь при спробі пошуку запису по ключу

| Номер спроби пошуку | Число порівнянь |
|---|---|
| 1 | 106 |
| 2 | 206 |
| 3 | 63 |
| 4 | 149 |
| 5 | 84 |
| 6 | 153 |
| 7 | 99 |
| 8 | 100 |
| 9 | 126 |
| 10 | 120 |
| AVG | 120.6 |

# Висновок

В рамках лабораторної роботи я попрацював з такою складною структурою даних як B-Tree та реалізував невеличку БД за допомогою цієї структури.

Програма була реалізована на мові C# з використанням ASP.NET MVC для графічного інтерфейсу. Реалізовані алгоритми пошуку, видалення, додавання і редагування вузла дерева. Часова складність пошуку O(t*logn), де n кількість вузлів в дереві, а t параметр дерева. Також за проведеним долідженням в середньому в алгоритмі пошуку відбувається 121 порівнняння.

## Критерії оцінювання

За умови здачі лабораторної роботи до 13.11.2022 включно максимальний бал дорівнює – 5. Після 13.11.2022 максимальний бал дорівнює – 1.

Критерії оцінювання у відсотках від максимального балу:

- псевдокод алгоритму – 15%;
- аналіз часової складності – 5%;
- програмна реалізація алгоритму – 65%;
- тестування алгоритму – 10%;
- висновок – 5%.

+1 додатковий бал можна отримати за реалізацію графічного зображення структури ключів.