МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федерально автономное бюджетное образовательное учреждение высшего образования
«Севастопольский государственный университет»
кафедра Информационных систем

Куркчи Ариф Эрнестович

Институт информационных технологий и управления в технических системах
курс 3 группа ИС/б-31-о
09.03.02 Информационные системы и технологии (уровень бакалавриата)

ОТЧЕТ

по лабораторной работе №3

по дисциплине «Методы и Средства Хранения Информации»

на тему «Эффективный доступ к данным во внешней памяти с использованием Б-деревьев»

Отметка о зачете _____ _____
(дата)

Руководитель практикума

ст. преподаватель	_____	Балясный Н.В
(должность)	(подпись)	(инициалы, фамилия)

Севастополь 2016

1. Цель работы

Исследовать возможности применения нелинейных структур, данных – Б-деревьев, для хранения и поиска информации. Приобрести практические навыки использования Б-деревьев для реализации эффективного поиска и доступа к данным. Произвести оценку эффективности использования Б-деревьев для организации хранения данных.

2. Постановка задачи

1. В ходе выполнения настоящей лабораторной работы в начале необходимо ознакомиться с организацией структуры хранения данных типа Б-дерево и программной реализацией классов, реализующих данную структуру.
2. На одном из языков программирования (C++ или Object Pascal) в среде визуального программирования (C++ Builder или Delphi, соответственно), с использованием классов, реализующих Б-деревья (для C++ файлы: CBTree.h, data.h; для ObjectPascal: файл Collection.pas) реализовать Windows приложение, обеспечивающее выполнения следующих функций:

2.1. Открытие файла исходных данных(имя файла и имя ключевого поля определяются вариантом задания – таблица 1) и построение на его основе индексного файла, содержащего Б-дерево порядка n;

2.2. Отображение на визуальной форме содержимого исходного файла (в виде списка - компонент TListView), Б-дерева индексного файла (компонент TtreeView);

2.3. Предоставление интерфейса пользователю для выполнения операций добавления, удаления, изменения и поиска(по ключевому полю) элементов в обоих файлах, и отображения результатов выполнения операций на визуальной форме; предусмотреть возможность ввода количества элементов, ограничивающее обрабатываемое число строк исходного файла(и количества узлов Б-дерва индексного файла);

2.4. Отображение времени выполнения операций добавления, удаления, изменения и поиска данных по заданному пользователем значению ключевого поля;

2.5. Отображение на визуальной форме актуальной информации о исходном файле и индексном файле: количество обрабатываемых записей в исходном файле, количество страниц и глубину Б-дерева;

3. С использованием разработанной программы выполнить исследования времени поиска в обычном файле (последовательный поиск) и в индексном файле (файле, содержащем Б-дерево):

3.1. На основании заданной по варианту таблицы выполнить построение индексного файла, содержащего N1 элементов;

3.2. Выполнить по 5 раз операции добавления, удаления и поиска информации (по случайным значениям ключевого поля), фиксируя в отчете время выполнения операций в простом и индексированном файлах;

3.3. Вычислить среднее время выполнения операций добавления, удаления и поиска информации (по ключевому полю) зафиксированных в п. 3.2.

4. Повторять п. 3.1 – 3.3 для количество элементов N2, N3 и N4, фиксируя получаемые значения времени в таблице

5. На основании данных, полученных при выполнении пп. 3 – 4 построить графики зависимости среднего времени, затрачиваемого на выполнение каждой операции (добавление удаление поиск) от количества элементов N для обычного и индексного файлов.

6. Сформулировать выводы.

Вариант №13

№	Файл данных	Ключевое поле	n	N1	N2	N3	N4
11	Table22.txt	Телефон	4	50	500	1600	6000

3. Текст программы

Main.java

```
1  package ru.justnero.study.dsmnm.lab03;
2
3  import javafx.application.Application;
4  import javafx.fxml.FXMLLoader;
5  import javafx.scene.Parent;
6  import javafx.scene.Scene;
7  import javafx.stage.Stage;
8
9  public class Main extends Application {
10
11     @Override
12     public void start(Stage primaryStage) throws Exception{
13         Parent root = FXMLLoader.load(getClass().getResource("main.fxml"));
14         primaryStage.setTitle("МиСХИ ЛР №3 Куркчи А. Э.");
15         primaryStage.setScene(new Scene(root, 1024, 768));
16         primaryStage.show();
17     }
18 '
19
20     public static void main(String[] args) {
21         launch(args);
22     }
23 }
```

BTreeImpl.java

```
1  package ru.justnero.study.dsmnm.lab03;
2
3  import java.util.Arrays;
4
5  import javafx.scene.control.TreeItem;
6
7  public class BTreeImpl<T extends Comparable<T>> {
8
9      private int minKeySize = 4;
10     private int minChildrenSize = minKeySize + 1;
11     private int maxKeySize = 2 * minKeySize;
12     private int maxChildrenSize = maxKeySize + 1;
13
14     private TreeNode<T> root = null;
15     private int size = 0;
16
17     public BTreeImpl() {
18     }
19
20     public BTreeImpl(int order) {
21         this.minKeySize = order;
22         this.minChildrenSize = minKeySize + 1;
23         this.maxKeySize = 2 * minKeySize;
24         this.maxChildrenSize = maxKeySize + 1;
```

```
25      }
26
27      public boolean add(T value) {
28          if (root == null) {
29              root = new TreeNode<T>(null, maxKeySize, maxChildrenSize);
30              root.addKey(value);
31          } else {
32              TreeNode<T> treeNode = root;
33              while (treeNode != null) {
34                  if (treeNode.numberOfChildren() == 0) {
35                      treeNode.addKey(value);
36                      if (treeNode.numberOfKeys() <= maxKeySize) {
37                          break;
38                      }
39                      split(treeNode);
40                      break;
41                  }
42
43                  T lesser = treeNode.getKey(0);
44                  if (value.compareTo(lesser) <= 0) {
45                      treeNode = treeNode.getChild(0);
46                      continue;
47                  }
48
49                  int numberOfKeys = treeNode.numberOfKeys();
50                  int last = numberOfKeys - 1;
51                  T greater = treeNode.getKey(last);
52                  if (value.compareTo(greater) > 0) {
53                      treeNode = treeNode.getChild(numberOfKeys);
54                      continue;
55                  }
56
57                  for (int i = 1; i < treeNode.numberOfKeys(); i++) {
58                      T prev = treeNode.getKey(i - 1);
59                      T next = treeNode.getKey(i);
60                      if (value.compareTo(prev) > 0 && value.compareTo(next) <= 0) {
61                          treeNode = treeNode.getChild(i);
62                          break;
63                      }
64                  }
65              }
66          }
67
68          size++;
69
70          return true;
71      }
72
73      private void split(TreeNode<T> treeNodeToSplit) {
74          TreeNode<T> treeNode = treeNodeToSplit;
75          int numberOfKeys = treeNode.numberOfKeys();
76          int medianIndex = numberOfKeys / 2;
77          T medianValue = treeNode.getKey(medianIndex);
78
79          TreeNode<T> left = new TreeNode<T>(null, maxKeySize, maxChildrenSize);
80          for (int i = 0; i < medianIndex; i++) {
81              left.addKey(treeNode.getKey(i));
82          }
83          if (treeNode.numberOfChildren() > 0) {
84              for (int j = 0; j <= medianIndex; j++) {
85                  TreeNode<T> c = treeNode.getChild(j);
86                  left.addChild(c);
87              }
88          }
89
90          TreeNode<T> right = new TreeNode<T>(null, maxKeySize, maxChildrenSize);
91          for (int i = medianIndex + 1; i < numberOfKeys; i++) {
92              right.addKey(treeNode.getKey(i));
93          }
94          if (treeNode.numberOfChildren() > 0) {
95              for (int j = medianIndex + 1; j < treeNode.numberOfChildren(); j++) {
96                  TreeNode<T> c = treeNode.getChild(j);
97                  right.addChild(c);
98              }
99          }
100
101         if (treeNode.parent == null) {
102             TreeNode<T> newRoot = new TreeNode<T>(null, maxKeySize, maxChildrenSize);
103             newRoot.addKey(medianValue);
104             treeNode.parent = newRoot;
105             root = newRoot;
106             treeNode = root;
107             treeNode.addChild(left);
```

```
108             treeNode.addChild(right);
109         } else {
110             TreeNode<T> parent = treeNode.parent;
111             parent.addKey(medianValue);
112             parent.removeChild(treeNode);
113             parent.addChild(left);
114             parent.addChild(right);
115 
116             if (parent.numberOfKeys() > maxKeySize) split(parent);
117         }
118     }
119 
120     public T remove(T value) {
121         T removed = null;
122         TreeNode<T> treeNode = this.find(value);
123         removed = remove(value, treeNode);
124         return removed;
125     }
126 
127     private T remove(T value, TreeNode<T> treeNode) {
128         if (treeNode == null) return null;
129 
130         T removed = null;
131         int index = treeNode.indexOf(value);
132         removed = treeNode.removeKey(value);
133         if (treeNode.numberOfChildren() == 0) {
134             if (treeNode.parent != null && treeNode.numberOfKeys() < minKeySize) {
135                 this.combined(treeNode);
136             } else if (treeNode.parent == null && treeNode.numberOfKeys() == 0) {
137                 root = null;
138             }
139         } else {
140             TreeNode<T> lesser = treeNode.getChild(index);
141             TreeNode<T> greatest = this.getGreatestNode(lesser);
142             T replaceValue = this.removeGreatestValue(greatest);
143             treeNode.addKey(replaceValue);
144             if (greatest.parent != null && greatest.numberOfKeys() < minKeySize) {
145                 this.combined(greatest);
146             }
147             if (greatest.numberOfChildren() > maxChildrenSize) {
148                 this.split(greatest);
149             }
150         }
151 
152         size--;
153 
154         return removed;
155     }
156 
157     private T removeGreatestValue(TreeNode<T> treeNode) {
158         T value = null;
159         if (treeNode.numberOfKeys() > 0) {
160             value = treeNode.removeKey(treeNode.numberOfKeys() - 1);
161         }
162         return value;
163     }
164 
165     public void clear() {
166         root = null;
167         size = 0;
168     }
169 
170     public boolean contains(T value) {
171         TreeNode<T> treeNode = find(value);
172         return (treeNode != null);
173     }
174 
175     public TreeNode<T> find(T value) {
176         TreeNode<T> treeNode = root;
177         while (treeNode != null) {
178             T lesser = treeNode.getKey(0);
179             if (value.compareTo(lesser) < 0) {
180                 if (treeNode.numberOfChildren() > 0)
181                     treeNode = treeNode.getChild(0);
182                 else
183                     treeNode = null;
184                 continue;
185             }
186 
187             int numberOfKeys = treeNode.numberOfKeys();
188             int last = numberOfKeys - 1;
189             T greater = treeNode.getKey(last);
190             if (value.compareTo(greater) > 0) {
```

```
191                 if (treeNode.numberOfChildren() > numberOfKeys)
192                     treeNode = treeNode.getChild(numberOfKeys);
193                 else
194                     treeNode = null;
195                 continue;
196             }
197 
198             for (int i = 0; i < numberOfKeys; i++) {
199                 T currentValue = treeNode.getKey(i);
200                 if (currentValue.compareTo(value) == 0) {
201                     return treeNode;
202                 }
203 
204                 int next = i + 1;
205                 if (next <= last) {
206                     T nextValue = treeNode.getKey(next);
207                     if (currentValue.compareTo(value) < 0 && nextValue.compareTo(value) > 0) {
208                         if (next < treeNode.numberOfChildren()) {
209                             treeNode = treeNode.getChild(next);
210                             break;
211                         }
212                         return null;
213                     }
214                 }
215             }
216         }
217         return null;
218     }
219 
220     private TreeNode<T> getGreatestNode(TreeNode<T> treeNodeToGet) {
221         TreeNode<T> treeNode = treeNodeToGet;
222         while (treeNode.numberOfChildren() > 0) {
223             treeNode = treeNode.getChild(treeNode.numberOfChildren() - 1);
224         }
225         return treeNode;
226     }
227 
228     private boolean combined(TreeNode<T> treeNode) {
229         TreeNode<T> parent = treeNode.parent;
230         int index = parent.indexOf(treeNode);
231         int indexOfLeftNeighbor = index - 1;
232         int indexOfRightNeighbor = index + 1;
233 
234         TreeNode<T> rightNeighbor = null;
235         int rightNeighborSize = -minChildrenSize;
236         if (indexOfRightNeighbor < parent.numberOfChildren()) {
237             rightNeighbor = parent.getChild(indexOfRightNeighbor);
238             rightNeighborSize = rightNeighbor.numberOfKeys();
239         }
240 
241         if (rightNeighbor != null && rightNeighborSize > minKeySize) {
242             T removeValue = rightNeighbor.getKey(0);
243             int prev = getIndexOfPreviousValue(parent, removeValue);
244             T parentValue = parent.removeKey(prev);
245             T neighborValue = rightNeighbor.removeKey(0);
246             treeNode.addKey(parentValue);
247             parent.addKey(neighborValue);
248             if (rightNeighbor.numberOfChildren() > 0) {
249                 treeNode.addChild(rightNeighbor.removeChild(0));
250             }
251         } else {
252             TreeNode<T> leftNeighbor = null;
253             int leftNeighborSize = -minChildrenSize;
254             if (indexOfLeftNeighbor >= 0) {
255                 leftNeighbor = parent.getChild(indexOfLeftNeighbor);
256                 leftNeighborSize = leftNeighbor.numberOfKeys();
257             }
258 
259             if (leftNeighbor != null && leftNeighborSize > minKeySize) {
260                 T removeValue = leftNeighbor.getKey(leftNeighbor.numberOfKeys() - 1);
261                 int prev = getIndexOfNextValue(parent, removeValue);
262                 T parentValue = parent.removeKey(prev);
263                 T neighborValue = leftNeighbor.removeKey(leftNeighbor.numberOfKeys() - 1);
264                 treeNode.addKey(parentValue);
265                 parent.addKey(neighborValue);
266                 if (leftNeighbor.numberOfChildren() > 0) {
267                     treeNode.addChild(leftNeighbor.removeChild(leftNeighbor.numberOfChildren() -
    1));
268                 }
269             } else if (rightNeighbor != null && parent.numberOfKeys() > 0) {
270                 T removeValue = rightNeighbor.getKey(0);
271                 int prev = getIndexOfPreviousValue(parent, removeValue);
272                 T parentValue = parent.removeKey(prev);
```

```
273                 parent.removeChild(rightNeighbor);
274                 treeNode.addKey(parentValue);
275                 for (int i = 0; i < rightNeighbor.keysSize; i++) {
276                     T v = rightNeighbor.getKey(i);
277                     treeNode.addKey(v);
278                 }
279                 for (int i = 0; i < rightNeighbor.childrenSize; i++) {
280                     TreeNode<T> c = rightNeighbor.getChild(i);
281                     treeNode.addChild(c);
282                 }
283 
284                 if (parent.parent != null && parent.numberOfKeys() < minKeySize) {
285                     this.combined(parent);
286                 } else if (parent.numberOfKeys() == 0) {
287                     treeNode.parent = null;
288                     root = treeNode;
289                 }
290             } else if (leftNeighbor != null && parent.numberOfKeys() > 0) {
291                 T removeValue = leftNeighbor.getKey(leftNeighbor.numberOfKeys() - 1);
292                 int prev = getIndexOfNextValue(parent, removeValue);
293                 T parentValue = parent.removeKey(prev);
294                 parent.removeChild(leftNeighbor);
295                 treeNode.addKey(parentValue);
296                 for (int i = 0; i < leftNeighbor.keysSize; i++) {
297                     T v = leftNeighbor.getKey(i);
298                     treeNode.addKey(v);
299                 }
300                 for (int i = 0; i < leftNeighbor.childrenSize; i++) {
301                     TreeNode<T> c = leftNeighbor.getChild(i);
302                     treeNode.addChild(c);
303                 }
304 
305                 if (parent.parent != null && parent.numberOfKeys() < minKeySize) {
306                     this.combined(parent);
307                 } else if (parent.numberOfKeys() == 0) {
308                     treeNode.parent = null;
309                     root = treeNode;
310                 }
311             }
312         }
313 
314         return true;
315     }
316 
317     private int getIndexOfPreviousValue(TreeNode<T> treeNode, T value) {
318         for (int i = 1; i < treeNode.numberOfKeys(); i++) {
319             T t = treeNode.getKey(i);
320             if (t.compareTo(value) >= 0)
321                 return i - 1;
322         }
323         return treeNode.numberOfKeys() - 1;
324     }
325 
326     private int getIndexOfNextValue(TreeNode<T> treeNode, T value) {
327         for (int i = 0; i < treeNode.numberOfKeys(); i++) {
328             T t = treeNode.getKey(i);
329             if (t.compareTo(value) >= 0)
330                 return i;
331         }
332         return treeNode.numberOfKeys() - 1;
333     }
334 
335 
336     public int size() {
337         return size;
338     }
339 
340 
341     public boolean validate() {
342         if (root == null) return true;
343         return validateNode(root);
344     }
345 
346     private boolean validateNode(TreeNode<T> treeNode) {
347         int keySize = treeNode.numberOfKeys();
348         if (keySize > 1) {
349             for (int i = 1; i < keySize; i++) {
350                 T p = treeNode.getKey(i - 1);
351                 T n = treeNode.getKey(i);
352                 if (p.compareTo(n) > 0)
353                     return false;
354             }
355         }
```

```
356         int childrenSize = treeNode.numberOfChildren();
357         if (treeNode.parent == null) {
358             if (keySize > maxKeySize) {
359                 return false;
360             } else if (childrenSize == 0) {
361                 return true;
362             } else if (childrenSize < 2) {
363                 return false;
364             } else if (childrenSize > maxChildrenSize) {
365                 return false;
366             }
367         } else {
368             if (keySize < minKeySize) {
369                 return false;
370             } else if (keySize > maxKeySize) {
371                 return false;
372             } else if (childrenSize == 0) {
373                 return true;
374             } else if (keySize != (childrenSize - 1)) {
375                 return false;
376             } else if (childrenSize < minChildrenSize) {
377                 return false;
378             } else if (childrenSize > maxChildrenSize) {
379                 return false;
380             }
381         }
382 
383         TreeNode<T> first = treeNode.getChild(0);
384         if (first.getKey(first.numberOfKeys() - 1).compareTo(treeNode.getKey(0)) > 0)
385             return false;
386 
387         TreeNode<T> last = treeNode.getChild(treeNode.numberOfChildren() - 1);
388         if (last.getKey(0).compareTo(treeNode.getKey(treeNode.numberOfKeys() - 1)) < 0)
389             return false;
390 
391         for (int i = 1; i < treeNode.numberOfKeys(); i++) {
392             T p = treeNode.getKey(i - 1);
393             T n = treeNode.getKey(i);
394             TreeNode<T> c = treeNode.getChild(i);
395             if (p.compareTo(c.getKey(0)) > 0)
396                 return false;
397             if (n.compareTo(c.getKey(c.numberOfKeys() - 1)) < 0)
398                 return false;
399         }
400 
401         for (int i = 0; i < treeNode.childrenSize; i++) {
402             TreeNode<T> c = treeNode.getChild(i);
403             boolean valid = this.validateNode(c);
404             if (!valid)
405                 return false;
406         }
407 
408         return true;
409     }
410 
411     public TreeItem<String> convert() {
412         if (root != null) {
413             TreeItem<String> tmp = convert(root);
414             return tmp;
415         }
416         return null;
417     }
418 
419     private TreeItem<String> convert(TreeNode<T> cur) {
420         if (cur == null) {
421             return null;
422         }
423         TreeItem<String> treeItem = new TreeItem<>(Arrays.toString(cur.keys));
424 
425         for (TreeNode<T> item : cur.children) {
426             treeItem.getChildren().add(convert(item));
427         }
428         return treeItem;
429     }
430 }
```

TData.java

```
1  package ru.justnero.study.dsmnm.lab03;
2  
3  import java.io.PrintStream;
4  import java.util.Scanner;
5  
6  public class TData implements Comparable<TData> {
```

```
 7
 8      private String name;
 9      private long phone;
10      private String home;
11      private float account;
12
13      public TData(long phone) {
14          this.phone = phone;
15      }
16
17      public TData(String name, long phone, String home, float account) {
18          this.name = name;
19          this.phone = phone;
20          this.home = home;
21          this.account = account;
22      }
23
24      public static TData read(Scanner inp) {
25          String name = inp.next();
26          name = name.replaceAll(",", ", ")
27                  .replaceAll("_", " ")
28                  .replaceAll("\"\"", "\"");
29          if (name.startsWith("\"") && name.endsWith("\"")) {
30              name = name.substring(1, name.length() - 1);
31          }
32          long phone = inp.nextLong();
33          String home = inp.next();
34          float account = inp.nextFloat();
35          return new TData(name, phone, home, account);
36      }
37
38      public void write(PrintStream out) {
39          out.print(name);
40          out.print(" ");
41          out.print(phone);
42          out.print(" ");
43          out.print(home);
44          out.print(" ");
45          out.print(account);
46          out.println();
47      }
48
49      public String getName() {
50          return name;
51      }
52
53      public void setName(String name) {
54          this.name = name;
55      }
56
57      public long getPhone() {
58          return phone;
59      }
60
61      public void setPhone(long phone) {
62          this.phone = phone;
63      }
64
65      public String getHome() {
66          return home;
67      }
68
69      public void setHome(String home) {
70          this.home = home;
71      }
72
73      public float getAccount() {
74          return account;
75      }
76
77      public void setAccount(float account) {
78          this.account = account;
79      }
80
81      @Override
82      public String toString() {
83          return String.valueOf(phone);
84      }
85
86      @Override
87      public int compareTo(TData to) {
88          return phone == to.phone ? 0 : (phone > to.phone ? 1 : -1);
89      }
```

```
90
91      @Override
92      public boolean equals(Object o) {
93          if (this == o) return true;
94          if (o == null || getClass() != o.getClass()) return false;
95
96          TData tData = (TData) o;
97
98          return phone == tData.phone;
99
100     }
101
102     @Override
103     public int hashCode() {
104         return (int) (phone ^ (phone >>> 32));
105     }
106 }
```

TreeNode.java

```
1   package ru.justnero.study.dsmnm.lab03;
2
3   import java.util.Arrays;
4   import java.util.Comparator;
5
6   class TreeNode<Data extends Comparable<Data>> {
7
8       TreeNode<Data> parent = null;
9       Data[] keys = null;
10      int keysSize = 0;
11      TreeNode<Data>[] children = null;
12      int childrenSize = 0;
13      private Comparator<TreeNode<Data>> comparator = (a, b) ->
    a.getKey(0).compareTo(b.getKey(0));
14
15      TreeNode(TreeNode<Data> parent, int maxKeySize, int maxChildrenSize) {
16          this.parent = parent;
17          this.keys = (Data[]) new Comparable[maxKeySize + 1];
18          this.keysSize = 0;
19          this.children = new TreeNode[maxChildrenSize + 1];
20          this.childrenSize = 0;
21      }
22
23      Data getKey(int index) {
24          return keys[index];
25      }
26
27      int indexOf(Data value) {
28          for (int i = 0; i < keysSize; i++) {
29              if (keys[i].equals(value)) return i;
30          }
31          return -1;
32      }
33
34      void addKey(Data value) {
35          keys[keysSize++] = value;
36          Arrays.sort(keys, 0, keysSize);
37      }
38
39      Data removeKey(Data value) {
40          Data removed = null;
41          boolean found = false;
42          if (keysSize == 0) return null;
43          for (int i = 0; i < keysSize; i++) {
44              if (keys[i].equals(value)) {
45                  found = true;
46                  removed = keys[i];
47              } else if (found) {
48                  keys[i - 1] = keys[i];
49              }
50          }
51          if (found) {
52              keysSize--;
53              keys[keysSize] = null;
54          }
55          return removed;
56      }
57
58      Data removeKey(int index) {
59          if (index >= keysSize)
60              return null;
61          Data value = keys[index];
62          System.arraycopy(keys, index + 1, keys, index + 1 - 1, keysSize - (index + 1));
63          keysSize--;
```

```
64          keys[keysSize] = null;
65          return value;
66      }
67
68      int numberOfKeys() {
69          return keysSize;
70      }
71
72      TreeNode<Data> getChild(int index) {
73          if (index >= childrenSize)
74              return null;
75          return children[index];
76      }
77
78      int indexOf(TreeNode<Data> child) {
79          for (int i = 0; i < childrenSize; i++) {
80              if (children[i].equals(child))
81                  return i;
82          }
83          return -1;
84      }
85
86      boolean addChild(TreeNode<Data> child) {
87          child.parent = this;
88          children[childrenSize++] = child;
89          Arrays.sort(children, 0, childrenSize, comparator);
90          return true;
91      }
92
93      boolean removeChild(TreeNode<Data> child) {
94          boolean found = false;
95          if (childrenSize == 0)
96              return false;
97          for (int i = 0; i < childrenSize; i++) {
98              if (children[i].equals(child)) {
99                  found = true;
100             } else if (found) {
101                 children[i - 1] = children[i];
102             }
103         }
104         if (found) {
105             childrenSize--;
106             children[childrenSize] = null;
107         }
108         return found;
109     }
110
111     TreeNode<Data> removeChild(int index) {
112         if (index >= childrenSize)
113             return null;
114         TreeNode<Data> value = children[index];
115         children[index] = null;
116         System.arraycopy(children, index + 1, children, index + 1 - 1, childrenSize - (index +
    1));
117         childrenSize--;
118         children[childrenSize] = null;
119         return value;
120     }
121
122     int numberOfChildren() {
123         return childrenSize;
124     }
125 }
```

Controller.java

```
1   package ru.justnero.study.dsmnm.lab03;
2
3   import java.io.IOException;
4   import java.nio.file.Paths;
5   import java.util.Optional;
6   import java.util.Scanner;
7
8   import javafx.collections.FXCollections;
9   import javafx.collections.ObservableList;
10  import javafx.fxml.FXML;
11  import javafx.fxml.FXMLLoader;
12  import javafx.scene.Scene;
13  import javafx.scene.control.TableColumn;
14  import javafx.scene.control.TableView;
15  import javafx.scene.control.TextInputDialog;
16  import javafx.scene.control.TreeView;
```

```
17  import javafx.scene.control.cell.PropertyValueFactory;
18  import javafx.stage.Stage;
19
20  public class Controller {
21
22      @FXML
23      private TreeView<String> treeView;
24      @FXML
25      private TableView<TimeLog> tableView;
26
27      private ObservableList<TimeLog> timeLogs = FXCollections.observableArrayList();
28      private ITree tree = new BTree();
29
30      private StatsController statsController;
31      private Stage testingStage;
32
33      private FormController formController;
34      private Stage formStage;
35
36      @SuppressWarnings("unused")
37      @FXML
38      public void initialize() {
39          initTable();
40
41          reloadTree();
42
43          try {
44              loadStages();
45          } catch (Exception ex) {
46              ex.printStackTrace();
47          }
48      }
49
50      @FXML
51      public void addAction() {
52          showForm("add");
53      }
54
55      @FXML
56      public void findAction() {
57          showForm("find");
58      }
59
60      @FXML
61      public void loadAction() {
62          TextInputDialog dialog = new TextInputDialog("10000");
63          dialog.setTitle("Загрузка данных");
64          dialog.setHeaderText("Загрузка из файла input.txt");
65          dialog.setContentText("Сколько объектов загрузить:");
66
67          Optional<String> result = dialog.showAndWait();
68          result.ifPresent(str -> {
69              try {
70                  @SuppressWarnings("unused")
71                  int cnt = Integer.valueOf(str);
72                  load(cnt);
73              } catch (NumberFormatException ignored) {
74              }
75          });
76      }
77
78      private void load(int count) {
79          int i = 0;
80          try (Scanner inp = new Scanner(Paths.get("input.txt"))) {
81              inp.nextLine();
82              long tmp, time = 0;
83              for (i = 0; i < count; i++) {
84                  TData data = TData.read(inp);
85                  tmp = System.nanoTime();
86                  tree.add(data);
87                  time += System.nanoTime() - tmp;
88              }
89              reloadTree();
90              timeLogs.add(new TimeLog("Загрузка", time));
91          } catch (Exception e) {
92              e.printStackTrace();
93              System.out.println(i);
94          }
95      }
96
97      @FXML
98      public void testAction() {
99          if(!testingStage.isShowing()) {
```

```
100             testingStage.show();
101         }
102         statsController.load("input.txt");
103     }
104 
105     public void treeAdd(TData data) {
106         long time = System.nanoTime();
107         tree.add(data);
108         time = System.nanoTime() - time;
109         timeLogs.add(new TimeLog("Добавление", time));
110         reloadTree();
111 
112         hideForm();
113     }
114 
115     public TreeNode<TData> treeFind(TData data) {
116         TreeNode<TData> ret;
117         long time = System.nanoTime();
118         ret = tree.find(data);
119         time = System.nanoTime() - time;
120         timeLogs.add(new TimeLog("Поиск", time));
121 
122         return ret;
123     }
124 
125     public void treeDel(TData data) {
126         long time = System.nanoTime();
127         tree.remove(data);
128         time = System.nanoTime() - time;
129         timeLogs.add(new TimeLog("Удаление", time));
130         reloadTree();
131 
132         hideForm();
133     }
134 
135     private void showForm(String action) {
136         formController.setAction(action);
137         if(!formStage.isShowing()) {
138             formStage.show();
139         }
140     }
141 
142     private void hideForm() {
143         formStage.close();
144     }
145 
146     private void loadStages() throws IOException {
147         FXMLLoader fxmlLoader = new FXMLLoader();
148         fxmlLoader.load(getClass().getResource("stats.fxml").openStream());
149         statsController = fxmlLoader.getController();
150 
151         testingStage = new Stage();
152         testingStage.setTitle("Тестирование");
153         testingStage.setScene(new Scene(fxmlLoader.getRoot(), 800, 600));
154 
155         fxmlLoader = new FXMLLoader();
156         fxmlLoader.load(getClass().getResource("form.fxml").openStream());
157         formController = fxmlLoader.getController();
158         formController.setController(this);
159 
160         formStage = new Stage();
161         formStage.setTitle("Форма ввода");
162         formStage.setScene(new Scene(fxmlLoader.getRoot(), 400, 200));
163     }
164 
165     private void initTable() {
166         TableColumn<TimeLog, String> sizeCol = new TableColumn<>("Операция");
167         sizeCol.prefWidthProperty().bind(tableView.widthProperty().divide(2));
168         sizeCol.setCellValueFactory(new PropertyValueFactory<>("name"));
169         tableView.getColumns().add(sizeCol);
170 
171         TableColumn<TimeLog, Long> timeCol = new TableColumn<>("Время");
172         timeCol.prefWidthProperty().bind(tableView.widthProperty().divide(2).subtract(1));
173         timeCol.setCellValueFactory(new PropertyValueFactory<>("time"));
174         tableView.getColumns().add(timeCol);
175 
176         tableView.setItems(timeLogs);
177     }
178 
179     private void reloadTree() {
180         treeView.setRoot(tree.convert());
181     }
182 
```

```
}
```

FormController.java

```
package ru.justnero.study.dsmnm.lab03;

import javafx.fxml.FXML;
import javafx.scene.control.Button;
import javafx.scene.control.TextField;

public class FormController {

    @FXML
    private TextField nameF;
    @FXML
    private TextField phoneF;
    @FXML
    private TextField homeF;
    @FXML
    private TextField accountF;
    @FXML
    private Button sendB;

    private TData data;
    private String action;

    private Controller controller;

    public void setController(Controller controller) {
        this.controller = controller;
    }

    @SuppressWarnings("unused")
    @FXML
    public void initialize() {
        clearFields();
    }

    @FXML
    public void sendAction() {
        if (action.equalsIgnoreCase("add")) {
            if (!validateForm()) {
                return;
            }
            TData target = new TData(
                    nameF.getText(),
                    Long.valueOf(phoneF.getText()),
                    homeF.getText(),
                    Float.valueOf(accountF.getText())
            );
            controller.treeAdd(target);
        } else if (action.equalsIgnoreCase("find")) {
            TData target = new TData(
                    "",
                    Long.valueOf(phoneF.getText()),
                    "",
                    0F
            );
            TreeNode<TData> node = controller.treeFind(target);
            if (node != null) {
                setAction("del");
                setData(node.getKey(node.indexOf(target)));
            }
        } else if (action.equalsIgnoreCase("del")) {
            controller.treeDel(data);
        }
    }

    public void setData(TData data) {
        this.data = data;
        nameF.setText(data.getName());
        phoneF.setText(String.valueOf(data.getPhone()));
        homeF.setText(data.getHome());
        accountF.setText(String.valueOf(data.getAccount()));
    }

    public void setFieldsMask(int mask) {
        int i = -1;
        nameF.setDisable((mask & (1 << ++i)) == 0);
        phoneF.setDisable((mask & (1 << ++i)) == 0);
        homeF.setDisable((mask & (1 << ++i)) == 0);
        accountF.setDisable((mask & (1 << ++i)) == 0);
    }

```

```
81      public void setAction(String action) {
82          this.data = null;
83          this.action = action;
84          int mask = 0;
85          if (action.equalsIgnoreCase("add")) {
86              sendB.setText("Добавить");
87              sendB.setDefaultButton(true);
88              mask = 1 + 2 + 4 + 8;
89              clearFields();
90          } else if (action.equalsIgnoreCase("find")) {
91              sendB.setText("Найти");
92              sendB.setDefaultButton(true);
93              mask = 2;
94              clearFields();
95          } else if (action.equalsIgnoreCase("del")) {
96              sendB.setText("Удалить");
97              sendB.setDefaultButton(false);
98              mask = 0;
99          }
100         setFieldsMask(mask);
101     }
102
103     public void clearFields() {
104         nameF.setText("");
105         phoneF.setText("");
106         homeF.setText("");
107         accountF.setText("");
108     }
109
110     private boolean validateForm() {
111         if (nameF.getText().isEmpty() ||
112                 phoneF.getText().isEmpty() ||
113                 homeF.getText().isEmpty() ||
114                 accountF.getText().isEmpty()) {
115             return false;
116         }
117         try {
118             @SuppressWarnings("unused")
119             long l = Long.valueOf(phoneF.getText());
120             @SuppressWarnings("unused")
121             float f = Float.valueOf(accountF.getText());
122         } catch (NumberFormatException ex) {
123             return false;
124         }
125
126         return true;
127     }
128
129 }
```

StatsController.java

```
1   package ru.justnero.study.dsmnm.lab03;
2
3   import java.nio.file.Paths;
4   import java.util.ArrayList;
5   import java.util.List;
6   import java.util.Random;
7   import java.util.Scanner;
8
9   import javafx.collections.FXCollections;
10  import javafx.collections.ObservableList;
11  import javafx.fxml.FXML;
12  import javafx.scene.chart.LineChart;
13  import javafx.scene.chart.XYChart;
14  import javafx.scene.control.TableColumn;
15  import javafx.scene.control.TableView;
16  import javafx.scene.control.cell.PropertyValueFactory;
17
18  public class StatsController {
19
20      private final ObservableList<TestLog> dataList = FXCollections.observableArrayList();
21      @FXML
22      private TableView<TestLog> table;
23      @FXML
24      private LineChart<String, Long> chart;
25      private List<TData> list;
26
27      @SuppressWarnings("unused")
28      @FXML
29      public void initialize() {
30          initTable(table, dataList);
```

```
31      }
32
33      void load(String fileName) {
34          list = read(fileName, 10000);
35          fillData();
36      }
37
38      private List<TData> read(String fileName, int maxCount) {
39          List<TData> list = new ArrayList<>(maxCount);
40          try (Scanner inp = new Scanner(Paths.get(fileName))) {
41              inp.nextLine();
42              for (int i = 0; i < maxCount; i++) {
43                  list.add(TData.read(inp));
44              }
45          } catch (Exception e) {
46              e.printStackTrace();
47          }
48          return list;
49      }
50
51      private void fillData() {
52          int tests[] = new int[]{50, 500, 1600, 6000};
53
54          ITree tree = new BTree();
55
56          XYChart.Series<String, Long> addSeries = new XYChart.Series<>();
57          addSeries.setName("BTree Добавление");
58          XYChart.Series<String, Long> findSeries = new XYChart.Series<>();
59          findSeries.setName("BTree Поиск");
60          XYChart.Series<String, Long> delSeries = new XYChart.Series<>();
61          delSeries.setName("BTree Удаление");
62          dataList.clear();
63
64          for (int test : tests) {
65              fillTest(dataList, test, tree, addSeries, findSeries, delSeries);
66          }
67          chart.getData().clear();
68          chart.getData().add(addSeries);
69          chart.getData().add(findSeries);
70          chart.getData().add(delSeries);
71      }
72
73      private void fillTest(ObservableList<TestLog> list, int size, ITree tree,
74                            XYChart.Series<String, Long> addSeries,
75                            XYChart.Series<String, Long> findSeries,
76                            XYChart.Series<String, Long> delSeries) {
77          tree.clear();
78          String category = String.valueOf(size);
79          int ids[] = generateRandomIds(5, size);
80          long times[] = new long[5];
81          TData data;
82          long time;
83          long average = 0;
84          for (int i = 0; i < size; i++) {
85              data = this.list.get(i);
86              time = System.nanoTime();
87              tree.add(data);
88              time = System.nanoTime() - time;
89              for (int j = 0; j < 5; j++) {
90                  if (ids[j] == i) {
91                      times[j] = time;
92                      average += time;
93                  }
94              }
95          }
96          average /= 5;
97          list.add(new TestLog(size, "Добавление", times[0], times[1], times[2], times[3],
    times[4], average));
98          addSeries.getData().add(new XYChart.Data<>(category, average));
99
100         average = 0;
101         for (int i = 0; i < 5; i++) {
102             data = this.list.get(ids[i]);
103             time = System.nanoTime();
104             tree.find(data);
105             time = System.nanoTime() - time;
106             times[i] = time;
107             average += time;
108         }
109         average /= 5;
110         list.add(new TestLog(size, "Поиск", times[0], times[1], times[2], times[3], times[4],
    average));
111         findSeries.getData().add(new XYChart.Data<>(category, average));
```

```
112
113         average = 0;
114         for (int i = 0; i < 5; i++) {
115             data = this.list.get(ids[i]);
116             time = System.nanoTime();
117             tree.remove(data);
118             time = System.nanoTime() - time;
119             times[i] = time;
120             average += time;
121         }
122         average /= 5;
123         list.add(new TestLog(size, "Удаление", times[0], times[1], times[2], times[3], times[4],
    average));
124         delSeries.getData().add(new XYChart.Data<>(category, average));
125     }
126
127     private int[] generateRandomIds(int count, int max) {
128         int result[] = new int[count];
129
130         Random rnd = new Random();
131         for (int i = 0; i < count; i++) {
132             result[i] = rnd.nextInt(max);
133         }
134
135         return result;
136     }
137
138     private void initTable(TableView<TestLog> table, ObservableList<TestLog> list) {
139         TableColumn<TestLog, Integer> sizeCol = new TableColumn<>("Размер");
140         sizeCol.prefWidthProperty().bind(table.widthProperty().divide(8));
141         sizeCol.setCellValueFactory(new PropertyValueFactory<>("size"));
142         table.getColumns().add(sizeCol);
143
144         TableColumn<TestLog, String> opCol = new TableColumn<>("Операция");
145         opCol.prefWidthProperty().bind(table.widthProperty().divide(8));
146         opCol.setCellValueFactory(new PropertyValueFactory<>("operation"));
147         table.getColumns().add(opCol);
148
149         TableColumn<TestLog, Long> timeCol;
150         for (int i = 1; i <= 5; i++) {
151             String is = String.valueOf(i);
152             timeCol = new TableColumn<>(is);
153             timeCol.prefWidthProperty().bind(table.widthProperty().divide(8));
154             timeCol.setCellValueFactory(new PropertyValueFactory<>("time" + is));
155             timeCol.setSortable(false);
156             table.getColumns().add(timeCol);
157         }
158
159         timeCol = new TableColumn<>("Среднее");
160         timeCol.prefWidthProperty().bind(table.widthProperty().divide(8).subtract(2));
161         timeCol.setCellValueFactory(new PropertyValueFactory<>("timeA"));
162         timeCol.setSortable(false);
163         table.getColumns().add(timeCol);
164
165         table.setItems(list);
166     }
167
168 }
```

4. Результаты

В таблице 4.1 представлены результаты проделанной работы.

Таблица 4.1 – Результаты

Размер	Операция	1	2	3	4	5	Среднее
50	Добавление	1838	3307	3409	6028	2153	3347
50	Поиск	8129	2820	1684	11403	2078	5222
50	Удаление	21300	10355	21300	4979	4804	12547
500	Добавление	2035	14365	1185	1883	1650	4223
500	Поиск	3210	2808	2148	1905	1945	2403
500	Удаление	17439	5616	5271	11407	5594	9065
1600	Добавление	1179	1581	13806	1608	455	3725
1600	Поиск	4221	2520	2264	2035	2020	2612
1600	Удаление	7404	5909	5688	6495	5362	6171
6000	Добавление	2337	804	667	551	750	1021
6000	Поиск	5019	2957	1929	2288	2529	2944
6000	Удаление	8199	5461	12561	12492	4803	8703

На рисунке 1 представлен график скорости выполнения операций добавления, удаления, поиска на количество элементов дерева.

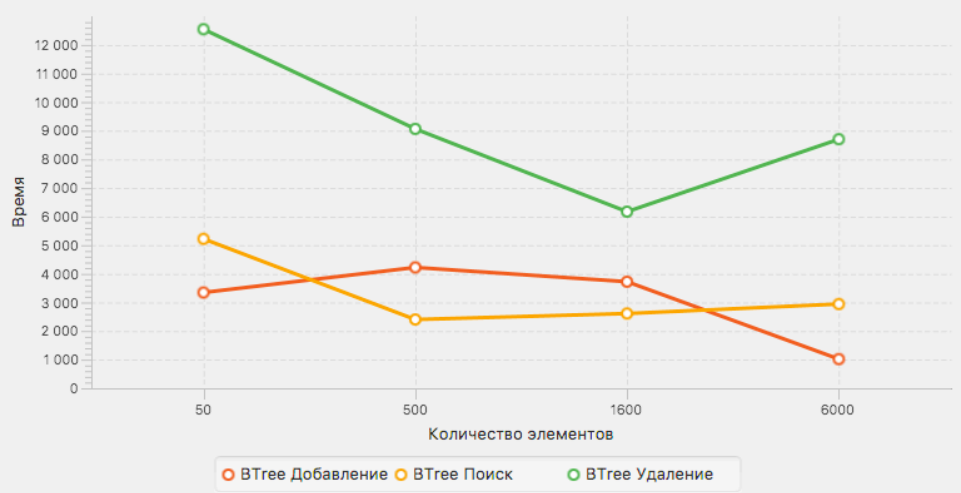

Рисунок 1 – График

Исходя из графика можно сделать выводы, что с ростом количества элементов находящимся в B-дереве, происходит увеличение скорости выполнения операций добавления и удаления, но снижение скорости поиска. Это обусловлено тем, что при добавление или удалении, происходит меньшее количество перераспределения элементов, находящихся в узле.

Вывод

В ходе выполнения лабораторной работы были исследованы возможности применения нелинейных структур, данных – Б-деревьев, для хранения и поиска информации. Приобретены практические навыки использования Б-деревьев для реализации эффективного поиска и доступа к данным. Произведена оценка эффективности использования Б-деревьев для организации хранения данных.