Университет информационных технологий, механики и оптики
Факультет компьютерных технологий и управления
Кафедра информатики и прикладной математики

ЛАБОРАТОРНАЯ РАБОТА №5
«Бинарный поиск и двоичное дерево поиска»

Выполнил:
студент гр. Р3118
Петкевич К. В.

Принял:
к.т.н старший
преподаватель
Симоненко З. Г.

г. Санкт-Петербург
2016 год

Цель работы

Для выполнения лабораторной работы необходимо сгенерировать тестовые файлы (используя генераторы случайных чисел), содержащие целые числа, в количестве от 2^6 до 2^{20} (можно и больше), при этом количество элементов в следующем файле в два раза больше чем в предыдущем, реализовать алгоритмы используя один из следующих языков программирования: C++, C#, C, Python, для каждого тестового файла из набора выполнить поиск элементов, которые гарантированно имеются во входных данных, построить график зависимости времени поиска одного элемента от количества элементов в файле, выполнить сравнение алгоритмов

Текст генератора исходных данных

```
static public TimeSpan FileCreator(int n, string path)
 {
            Random rnd = new Random((int)DateTime.Now.Ticks);
            Stopwatch timer = new Stopwatch();
            TimeSpan time;
            string s = path + "/TestFile";
            int i = 0, j = 0;
            timer = Stopwatch.StartNew();
            for (i = 0; i < n; i++)
            {
                        string str = @s + i + ".txt";
                        StreamWriter stream = File.AppendText(str);
                        for (j = 0; j < (Math.Pow(2, 6 + i)); j++)
                        {
                                    string line = Convert.ToString(rnd.Next(0, Convert.ToInt32(Math.Pow(2, 6 + i))));
                                    stream.WriteLine(line);
                        }
                        stream.Close();
            }
            Console.WriteLine("\nGenerated!\n");
            timer.Stop();
            time = timer.Elapsed;
            return (time);
 }
```

Коды сортировок

1. Бинарный поиск

```
public static int? BinarySearch(T[] items, T key)
            // array must be sorted
{
            int left = 0;
            int right = items.Length;
            int mid = 0;

            while (!(left >= right))
            {
                        mid = left + (right - left) / 2;

                        if (items[mid].CompareTo(key) == 0)
                                    return mid;

                        if (items[mid].CompareTo(key) > 0)
                                    right = mid;
                        else
                                    left = mid + 1;
            }

            return null;
}
```

2. Бинарное дерево поиска

```
public class BinarySearchTree
{
	public Node root;
	public class Node
	{
		public T Value { get; set; }
		public int Key { get; set; }
		public Node Left;
		public Node Right;

		public Node(int key, T value)
		{
			Key = key;
			Value = value;
			Left = null;
			Right = null;
		}
	}


	public BinarySearchTree (T[] items = null)
	{
		if (items != null)
			FillTree(items);
	}
	void FillTree(T[] items)
	{

		for (int i = 0; i < items.Length; i++)
		{
			Insert(i, items[i]);
		}
	}

	public int? Find (T item)
	{
		Node current = root;

		while (current != null)
		{
			if (current.Value.CompareTo(item) == 0)
				return current.Key;

			if (current.Value.CompareTo(item) > 0)
				current = current.Left;
			else
				current = current.Right;
		}

		return null;
	}

	public int? Find(T[] items, T item)
```

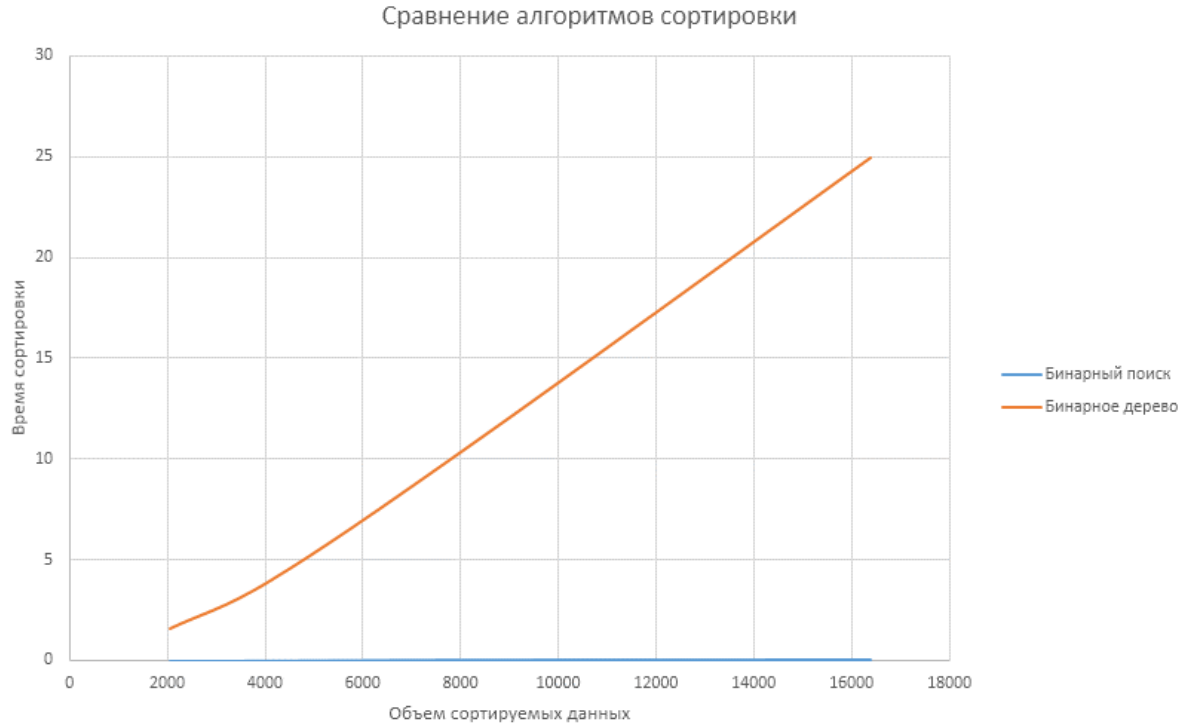
```
{
        FillTree(items);

        return Find(item);
}

public void Insert (int key, T value)
{
        // trivial case
        if (root == null)
        {
                root = new Node(key, value);
                return;
        }

        Node current = root;
        while (current != null)
        {
                if (value.CompareTo(current.Value) <= 0)
                {
                        if (current.Left == null)
                        {
                                current.Left = new Node(key, value);
                                break;
                        }
                        else
                                current = current.Left;
                }
                else
                {
                        if (current.Right == null)
                        {
                                current.Right = new Node(key, value);
                                break;
                        }
                        else
                                current = current.Right;
                }
        }
}

public void Remove (T item)
{
        Remove(item, root);
}
void Remove (T item, Node root)
{
        if (root == null)
                return;

        // find element to remove
        if (root.Value.CompareTo(item) > 0)
                Remove(item, root.Left);
        else if (root.Value.CompareTo(item) < 0)
                Remove(item, root.Right);
```

```
		// current root has left and right child
		else if (root.Left != null && root.Right != null)
		{
			root.Value = min(root.Right).Value;
			Remove(root.Right.Value, root.Right);
		}

		// current root has only left child
		else if (root.Left != null)
			root = root.Left;

		// current root has only right child
		else if (root.Right != null)
			root = root.Right;

		// current root doesn't have children
		else
			root = null;
}

Node min(Node current = null)
{
		if (current == null)
			current = root;

		while (current.Left != null)
			current = current.Left;

		return current;
}

public List<T> preOrder()
		// root, left, right
{
		List<T> items = new List<T>();
		preOrder(root, items);
		return items;
}
void preOrder(Node root, List<T> items)
{
		if (root == null)
			return;

		Node right = root.Right;
		Node left = root.Left;
		items.Add(root.Value);
		preOrder(right, items);
		preOrder(left, items);
}

public List<T> inOrder ()
		// left, root, right
{
		List<T> items = new List<T>();
```

```
                inOrder(root, items);

                return items;
        }
        void inOrder(Node root, List<T> items)
        {
                if (root == null)
                        return;

                Node left = root.Left;
                Node right = root.Right;

                inOrder(left, items);
                items.Add(root.Value);
                inOrder(right, items);
        }

        public List<T> postOrder()
                // left, right, root
        {
                List<T> items = new List<T>();
                inOrder(root, items);

                return items;
        }
        void postOrder(Node root, List<T> items)
        {
                if (root == null)
                        return;

                Node left = root.Left;
                Node right = root.Right;

                postOrder(left, items);
                postOrder(right, items);
                items.Add(root.Value);
        }
}
```

Результаты

	Время сортировки, с	
Кол-во эл-в	Бинарный поиск	Бинарное дерево
2048	0,001116542	1,572676532
4096	0,001172356	3,948386024
8192	0,001529453	10,66314606
16384	0,001694321	24,96921534



Вывод

Алгоритм бинарного поиска оказался самым эффективным по времени. Он прост в реализации, но на вход ему требуется уже отсортированный массив, а это значит, что изначально массив нужно отсортировать в отличии от бинарного дерева.