Всестороннее изучение структур данных с использованием C# 2.0

Часть III: Бинарные деревья и бинарные деревья поиска (BST)

Эта статья, третья из цикла шести статей о структурах данных в.NET Framework, рассматривает структуры данных не включенные в .NET Framework Base Class Library — бинарные деревья. В то время как массивы организуют данные линейно, бинарные деревья можно рассматривать как структуры для хранения данных в двух измерениях. Специальный вид бинарного дерева называемый бинарным деревом поиска, или BST, позволяет иметь более оптимальное время поиска, чем в неотсортированных массивах.

1. Введение

В первой части этой серии статей мы рассмотрели, что такое структуры данных, как измеряется их производительность и как эти соглашения по производительности влияют на выбор той или иной структуры данных для использования в определенном алгоритме. В дополнение к обзору базовых структур данных и их анализу мы также рассмотрели наиболее часто используемую структуру данных – массив – и его родственника Список (List). В Части II мы рассмотрели двоюродных братьев Списка (List) – Стек (Stack) и Очередь (Queue), которые хранят данные так же как и Список, но ограничены способом доступа к этим данным. В Части II мы также рассмотрели классы Хеш таблиц (Hashtable) и Словарей (Dictionary), которые по существу представляют собой массив, индексируемый произвольным объектом, в противоположность индексированию целым значением.

Список (List), Стек (Stack), Очередь (Queue), Хеш-таблица (Hashtable) и Словарь (Dictionary) используют массив посредством которого они хранят свои данные. Это означает, что под покровом эти структуры данных ограничены возможностями массива. Вспомните из первой части, что массив хранится линейно в оперативной памяти, требуя явного изменения размера при достижении его емкости, и предоставляет линейное время поиска.

В этой третьей статье серии мы изучим новую структуру данных – бинарное дерево. Как мы увидим бинарные деревья хранят данные не в линейном порядке. После обсуждения свойств бинарных деревьев мы рассмотрим более специфический вид бинарного дерева – бинарное дерево поиска или BST. BST навязывает определенные правила по организации элементов дерева. Эти правила обеспечивают BST сублинейным временем поиска.

2. Организация данных в дереве

Если вы когда-либо видели генеалогическое дерево или цепочку подчинения в корпорации, то вы видели данные организованные в виде *дерева*. Дерево состоит из набора узлов, где каждый узел имеет некоторые ассоциированные с ним данные и множество *потомков* (*children*). Потомки узла это те узлы, которые находятся

непосредственно сразу под данным узлом. Родитель (*parent*) узла это узел, который находится непосредственно над данным узлом. Корень (*root*) дерева это единственный узел, который не имеет родителя.

Рисунок 1 показывает пример цепочки подчинения в фиктивной компании.

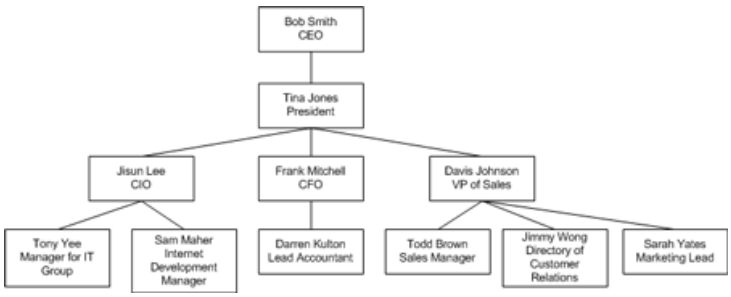

Рисунок 1. Цепочки подчинения в фиктивной компании в представленные виде дерева

В этом примере корнем дерева является Боб Смит (Bob Smith), CEO. Этот узел является корнем потому что он не имеет родителя. Узел Боб Смит имеет одного потомка – Тину Джонс (Tina Jones), президента, родителем которой является Боб Смит. Узел Тины Джонс (Tina Jones) имеет трех потомков — Джисун Ли, Франк Мтчел и Дэвис Джонсон. Родителем каждого из этих узлов является узел Тины Джонс. Узел Джисун Ли имеет два потомка — Тони Йи и Сэм Махер; узел Франка Митчела имеет одного потомка — Дарен Култон; и, наконец, узел Дэвис Джонсон имеет трех потомков — Тод Браун, Джимми Вонг и Сара Йетс.

Все три дерева проявляют следующие свойства:

- Существует в точности один корень.
- Все узлы за исключением корня имеют в точности одного родителя.
- В дереве не существует циклов (*cycles*). То есть, начиная с любого заданного узла не существует никакого пути, который приведет к исходному узлу. Первые два свойства— существование одного корня и того, что все узлы имеют одного родителя — гарантируют отсутствие циклов.

Деревья очень удобны для организации иерархических данных. Как мы будем далее обсуждать в статье время для поиска элемента в дереве может быть существенно уменьшено путем грамотной организации иерархии. Перед тем как мы перейдем к обсуждению этого вопроса нам сперва необходимо рассмотреть специальный вид дерева – бинарное дерево (*binary tree*).

3. Понятие бинарных, или двоичных, деревьев

Бинарное дерево это особый вид дерева, такого, в котором все узлы имеют, по крайней мере, два потомка. Для данного узла в бинарном дереве первый потомок называется *левым* потомком, а второй потомок – *правым* потомком. Рисунок 2 показывает два примера бинарных деревьев.

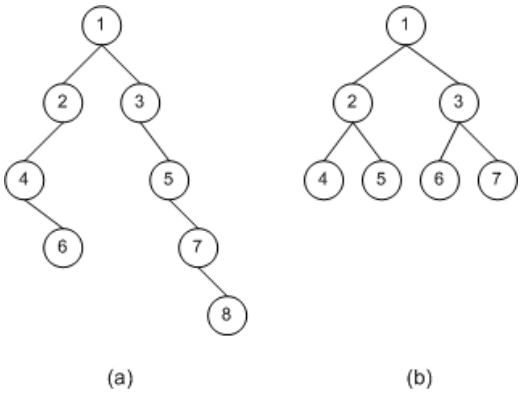

Рисунок 2. Иллюстрация двух бинарных деревьев

Бинарное дерево (a) имеет 8 узлов, где узел 1 является корнем дерева. Левым потомком узла 1 является узел 2; правым потомком узла 1 является узел 3. Отметьте, что узел не обязан иметь и левого и правого потомков. В бинарном дереве (a), узел 4, например, имеет только одного правого потомка. Более того, узел вообще может не иметь потомков. В бинарном дереве (b), узлы 4, 5, 6, и 7 не имеют потомков.

Узлы, которые не имеют потомков, называются листьями (*leaf nodes*). Узлы, которые имеют одного или двух потомков называются внутренними узлами (*internal nodes*). Используя эти новые определения можно сказать, что в бинарном дереве (a) узлы 6 и 8 являются листьями; внутренние узлы это узлы с номерами 1, 2, 3, 4, 5, и 7.

К сожалению .NET Framework не содержит класса бинарных деревьев, поэтому давайте потратим немного времени на создание нашего собственного класса бинарного дерева.

3.1 Первый шаг: создание базового класса узла

Первый шаг в проектировании нашего класса бинарных деревьев состоит в создании класса, который представляет узлы бинарного дерева. Вместо того, чтобы создавать класс специфичный узлам бинарного дерева, давайте создадим базовый класс `Node`, который можно расширить до требуемой функциональности узла бинарного дерева с использованием наследования. Базовый класс `Node` представляет узел обобщенного дерева, такого, чьи узлы могут содержать произвольное число потомков. Чтобы смоделировать это мы построим не только класс `Node`, но также и класс `NodeList`. Класс `Node` содержит некоторые данные и экземпляр класса `NodeList`, который представляет потомков узла. Класс `Node` предоставляет замечательную возможность использования мощности параметризованных типов (Generics), которые дадут нам возможность позволить разработчикам, использующим класс, принимать решение во время разработки о том какой тип данных хранить в узле.

Далее следует код класса `Node`.

Comment [w1]: Побуждение, стоящее за созданием параметризованного класса **Node** состоит в том, что далее в этой статье, а также в последующих статьях, мы будем создавать другие классы, которые состоят из множества узлов. Вместо того, чтобы каждый класс создавал свой собственный специфичный класс узла, каждый класс будет заимствовать функциональность базового класса `Node` и затем расширять базовый класс до требуемой функциональности.

```
public class Node<T>
{
        // Приватные переменные класса
        private T data;
        private NodeList<T> neighbors = null;
```

```
        public Node() {}
        public Node(T data) : this(data, null) {}
        public Node(T data, NodeList<T> neighbors)
        {
            this.data = data;
            this.neighbors = neighbors;
        }

        public T Value
        {
            get
            {
                return data;
            }
            set
            {
                data = value;
            }
        }

        protected NodeList<T> Neighbors
        {
            get
            {
                return neighbors;
            }
            set
            {
                neighbors = value;
            }
        }
    }
}
```

Заметьте, что класс `Node` имеет две приватных переменных класса:

- `data`, типа `T`. Эта переменная класса содержит данные, хранимые в узле, и типа, определенного разработчиком, использующим этот класс.
- `neighbors`, типа `NodeList<T>`. Эта переменная класса представляет потомков узла.

Оставшаяся часть реализации класса содержит конструкторы и свойства, которые предоставляют доступ к двум переменным класса.

Класс `NodeList` содержит строго типизированную коллекцию экземпляров `Node<T>`. Как демонстрирует код приведенный ниже класс `NodeList` является производным от класса `Collection<T>` в пространстве имен `System.Collections.Generics`. Класс `Collection<T>` обеспечивает базовую функциональность для строго типизированной коллекции с помощью таких методов как `Add(T)`, `Remove(T)`, и `Clear()`, а также свойства `Count` и индексатора. В дополнение к методам и свойствам, унаследованным от `Collection<T>`, `NodeList` предоставляет конструктор, который создает заданное количество узлов в коллекции, и метод, с помощью которого осуществляется поиск определенного элемента в коллекции.

```
public class NodeList<T> : Collection<Node<T>>
{
    public NodeList() : base() { }

    public NodeList(int initialSize)
    {
```

```
        // Добавить заданное количество элементов в коллекцию
        for (int i = 0; i < initialSize; i++)
            base.Items.Add(default(Node<T>));
    }

    public Node<T> FindByValue(T value)
    {
        // искать значение в списке
        foreach (Node<T> node in Items)
            if (node.Value.Equals(value))
                return node;

        // если мы дошли до этого места, то нет совпадающего узла
        return null;
    }
}
```

3.2 Расширение базового класса узла Node

В то время как класс `Node` является достаточным для обобщенного дерева, бинарное дерево имеет более строгие ограничения. Как это уже обсуждалось ранее узел бинарного дерева имеет по крайней мере двух потомков часто называемых левым и правым. Чтобы предоставить класс узла специфичный для бинарного дерева мы можем расширить базовый класс Node путем создания класса **BinaryTreeNode**, который предоставляет два свойства — `Left` и `Right` — оперирующих над свойством базового класса `Neighbors`.

```
public class BinaryTreeNode<T> : Node<T>
{
    public BinaryTreeNode() : base() {}
    public BinaryTreeNode(T data) : base(data, null) {}
    public BinaryTreeNode(T data, BinaryTreeNode<T> left, BinaryTreeNode<T>
right)
    {
        base.Value = data;
        NodeList<T> children = new NodeList<T>(2);
        children[0] = left;
        children[1] = right;

        base.Neighbors = children;
    }

    public BinaryTreeNode<T> Left
    {
        get
        {
            if (base.Neighbors == null)
                return null;
            else
                return (BinaryTreeNode<T>) base.Neighbors[0];
        }
        set
        {
            if (base.Neighbors == null)
                base.Neighbors = new NodeList<T>(2);

            base.Neighbors[0] = value;
        }
    }

    public BinaryTreeNode<T> Right
    {
```

```
        get
        {
            if (base.Neighbors == null)
                return null;
            else
                return (BinaryTreeNode<T>) base.Neighbors[1];
        }
        set
        {
            if (base.Neighbors == null)
                base.Neighbors = new NodeList<T>(2);

            base.Neighbors[1] = value;
        }
    }
}
```

Львиная доля работы этого расширенного класса находится в свойствах `Right` и `Left`. Внутри этих свойств нам требуется удостовериться, что `NodeList` базового класса `Neighbors` был создан. Если он не был создан, то метод доступа get возвращает `null`; в методе доступа set нам необходимо создать новый `NodeList` имеющий в точности два элемента. Как вы можете видеть в коде свойство `Left` обращается к первому элементу в коллекции `Neighbors` (`Neighbors[0]`), а свойство `Right` обращается ко второму элементу (`Neighbors[1]`).

3.3 Создание класса BinaryTree

Имея реализованным класс `BinaryTreeNode`, реализация класса `BinaryTree` представляет собой пустяк. Класс `BinaryTree` содержит единственную приватную переменную класса — `root`. `root` имеет тип `BinaryTreeNode` и представляет собой корень бинарного дерева. Эта приватная переменная класса предоставляется для использования в качестве свойства общего использования (public property). (Класс `BinaryTree` также использует параметризованные типы (Generics); тип заданный для класса `BinaryTree` является типом, используемым для корня `BinaryTreeNode`.)

Класс `BinaryTree` имеет единственный общедоступный метод, `Clear()`, который стирает содержимое дерева. `Clear()` работает просто устанавлиявая значение переменной `root` в `null`. Кроме свойства root и метода `Clear()`, класс **BinaryTree** не содержит других свойств или методов. Построение содержимого бинарного дерева целиком лежит на ответственности разработчика, который использует эту структуру данных.

Ниже приведен код класса `BinaryTree`.

```
public class BinaryTree<T>
{
   private BinaryTreeNode<T> root;

   public BinaryTree()
   {
      root = null;
   }

   public virtual void Clear()
   {
      root = null;
   }
```

```
   public BinaryTreeNode<T> Root
   {
      get
      {
         return root;
      }
      set
      {
         root = value;
      }
   }
}
```

Следующий блок кода иллюстрирует как использовать класс `BinaryTree` для создания бинарного дерева с такой же самой структурой как бинарное дерево (а) показанное на Рисунке 2.

```
BinaryTree<int> btree = new BinaryTree<int>();
btree.Root = new BinaryTreeNode<int>(1);
btree.Root.Left = new BinaryTreeNode<int>(2);
btree.Root.Right = new BinaryTreeNode<int>(3);

btree.Root.Left.Left = new BinaryTreeNode<int>(4);
btree.Root.Right.Right = new BinaryTreeNode<int>(5);

btree.Root.Left.Left.Right = new BinaryTreeNode<int>(6);
btree.Root.Right.Right.Right = new BinaryTreeNode<int>(7);

btree.Root.Right.Right.Right.Right = new BinaryTreeNode<int>(8);
```

Заметьте, что мы начинаем с создания экземпляра класса `BinaryTree`, а затем создаем его корень. Затем мы должны вручную добавить новые экземпляры класса `BinaryTreeNode` для соответствующих правых и левых потомков. Например, чтобы добавить узел 4, который является левым потомком левого потомка корня мы используем:
`btree.Root.Left.Left = new BinaryTreeNode<int>(4);`

Вспомните из первой части статей этой серии, что элементы массива хранятся в непрерывном блоке памяти. Организуя элементы таким образом массив предлагает константное время доступа к элементам. То есть время доступа к элементу массива не изменяется при увеличении числа элементов.

Бинарные деревья, однако, не хранятся в непрерывной памяти, как это проиллюстрировано на Рисунке 3. Вместо этого экземпляр класса `BinaryTree` ссылается на корневой экземпляр класса `BinaryTreeNode`. Корневой экземпляр класса `BinaryTreeNode` имеет ссылки на экземпляры своих правого и левого потомков класса `BinaryTreeNode`; эти экземпляры потомков ссылаются на экземпляры их потомков и так далее. Различные экземпляры `BinaryTreeNode` которые формируют бинарное дерево могут быть разбросаны по управляемой куче CLR. Они не обязательно должны и будут находится в непрерывном участке памяти, как это происходит с элементами массива.

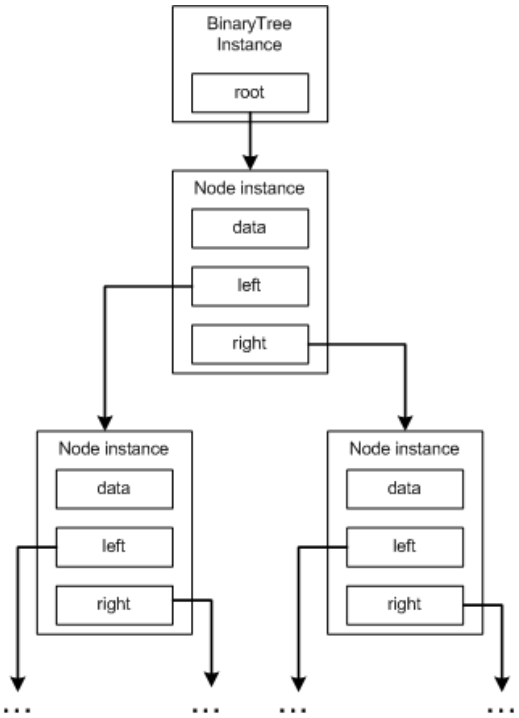

Рисунок 3. Хранение бинарного дерева в памяти

Binary Tree Instance – экземпляр бинарного дерева
Root – корень
Node instance – экземпляр узла
Data – данные
Left – левый
Right – правый

Представим, что мы хотим получить доступ к определенному узлу бинарного дерева. Чтобы осуществить это нам необходимо осуществить поиск определенного узла в множестве узлов бинарного дерева. Здесь нет непосредственного доступа к данному узлу, как это имеет быть в массиве. Поиск в бинарном дереве может занимать линейное время так как потенциально необходимо проверить все узлы. То есть с ростом числа узлов в бинарном дереве растет и время поиска заданного узла.

Поэтому, если время доступа к узлу бинарного дерева линейно, и время поиска в бинарном дереве также линейно, то каким же образом бинарное дерево лучше чем массив, чье время поиска также линейно, а время доступа является константой? Да, обобщенное бинарное дерево не предлагает нам никакого преимущества над линейным массивом. Однако, разумно организуя элементы в бинарном дереве, мы можем существенно улучшить время поиска (и тем самым время доступа к узлу).

4. Улучшение времени поиска в бинарных деревьях поиска (BST)

Бинарное дерево поиска (*binary search tree*) это особый вид бинарного дерева созданного для улучшения эффективности поиска содержимого бинарного дерева. Бинарные деревья поиска предлагают следующее свойство: для любого узла *n*, значение каждого последующего узла в левом поддереве (*subtree*) узла *n* является меньше, чем значение узла *n*, и значение каждого последующего узла в правом поддереве больше, чем значение узла *n*.

Поддерево, имеющее своим корнем узел *n*, представляет собой дерево с воображаемым корнем в виде узла *n*. То есть, узлы поддерева являются потомками узла *n* и корнем поддерева является сам узел *n*. Рисунок 4 иллюстрирует концепцию поддеревьев и свойства бинарного дерева поиска.

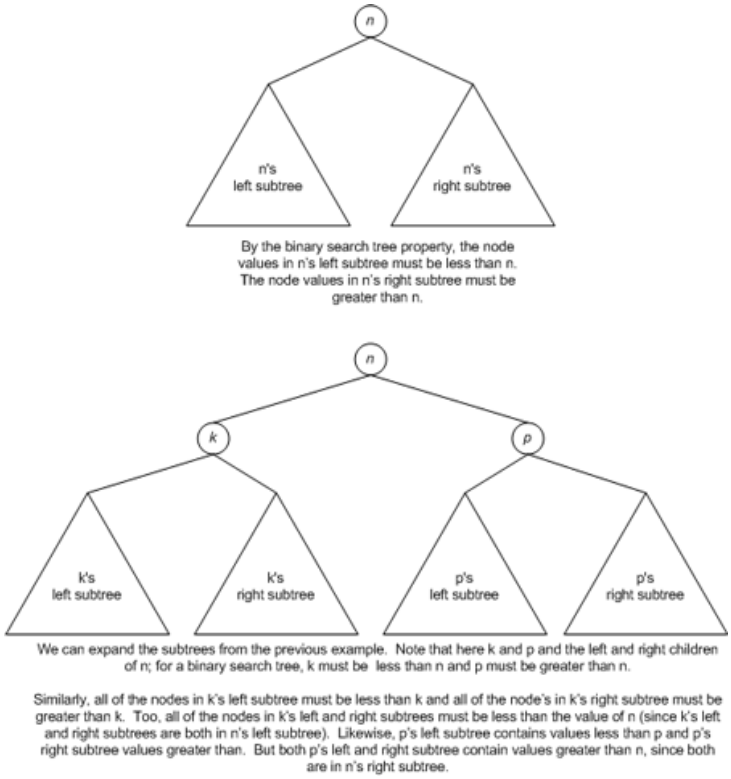

Рисунок 4. Поддеревья и свойство бинарного дерева поиска

N's left subtree – левое поддерево узла n
N's right subtree – правое поддерево узла n
Согласно свойству бинарного дерева поиска значения в левом поддереве узла n должны быть меньше, чем n. Значения узлов в правом поддереве узла n должны быть больше, чем n.
Левое поддерево узла k
Правое поддерево узла k
Левое поддерево узла p
Правое поддерево узла p
Мы можем расширить, или раскрыть, поддеревья из предыдущего примера. Отметьте, что здесь k и p – левый и правый потомки узла n соответственно; для бинарного дерева поиска k должно быть меньше, чем n и p должно быть больше, чем n.
Аналогично все узлы в левом поддереве k должны быть меньше, чем k и все узлы в правом поддереве k должны быть больше, чем k. А также, все узлы в правом и левом поддеревьях k должны быть меньше, чем n (так как и левое и правое поддеревья k расположены в левом поддереве n). Аналогично, левое поддерево p содержит значения, которые меньше,

чем p и правое поддерево p содержит значения, которые больше, чем p. Но левое и правое поддеревья p содержат величины, которые больше, чем n, так как оба поддерева находятся в правом поддереве n/

Рисунок 5 показывает два примера бинарных деревьев. Дерево справа, бинарное дерево (b), является BST деревом потому что оно проявляет свойство бинарного дерева поиска. Бинарное дерево (a), однако, не является BST потому что не все узлы дерева проявляют свойство бинарного дерева поиска. Точнее, правый потомок 8 узла 10, меньше чем 10, однако он находится в правом поддереве узла 10. Аналогично, правый потомок узла 8 узел 4 меньше, чем 8, однако он находится в правом поддереве узла 8. Это свойство нарушается также и в других местах. Например, правое поддерево узла 9 содержит величины, которые меньше 9, а именно 8 и 4.

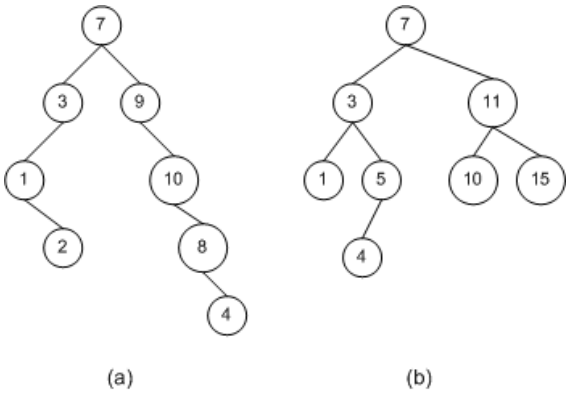

Рисунок 5. Сравнение не-BST бинарного дерева (a) и BST бинарного дерева (b)

Отметьте, что для того, чтобы выдержать свойство бинарного дерева поиска, данные, хранимые в узлах BST, должны иметь возможность сравнения друг с другом. В частности, имея два узла, BST должно иметь возможность определять является ли один узел меньше, больше или равен другому.

Теперь представьте, что вы хотите произвести поиск определенного узла в BST. Например, для BST на Рисунке 5 (бинарное дерево (b)), представьте, что мы хотим произвести поиск узла 10. BST, также как и обычное бинарное дерево, имеет непосредственный доступ только к одному узлу – корню. Можете ли вы придумать оптимальный способ поиска в дереве, чтобы определить существует ли узел 10? Существует лучший способ поиска, чем проверка каждого узла дерева.

Чтобы увидеть, что 10 существует в дереве мы должны начать с его корня. Мы видим, что значение корня (7) меньше значения узла, который мы ищем. Поэтому, если 10 существует в BST, то оно должно находиться в правом поддереве корня. Поэтому, мы продолжаем наш поиск в узле 11. Здесь мы замечаем, что 10 меньше 11, поэтому если 10 существует в BST оно должно находиться в левом поддереве 11. Перемещаясь к левому потомку узла 11, мы находим 10, и тем самым мы нашли искомый узел.

Что произойдет если мы ищем узел, который не существует в дереве? Предположим, что мы хоти найти узел 9. Мы начинаем повторяя те же шаги, что и описанные ранее. По достижению узла 10, мы видим, что узел 10 больше чем 9, поэтому 9, если оно существует

то оно должно находиться в левом поддереве узла 10. Однако мы видим, что 10 не имеет левого потомка, поэтому 9 не должно существовать в дереве.

Более формально наш алгоритм поиска работает следующим образом. У нас имеется узел *n*, который мы желаем найти (или определить, что он существует), и мы имеем ссылку на корень BST. Этот алгоритм выполняет некоторое количество сравнений до тех пор пока не будет встречена пустая ссылка, либо не будет найден искомый узел. На каждом шаге мы имеем дело с двумя узлами: узлом в дереве, назовем его *c*, который мы в данный момент времени сравниваем с *n*, узлом, который мы ищем. Изначально *c* это корень BST. Мы применяем следующие шаги:

1. Если *c* это пустая (null) ссылка, то завершить алгоритм . *n* не находится в BST.
2. Сравнить значения *c* и *n*.
3. Если значения равны, то мы нашли *n*.
4. Если значение *n* меньше, чем значение *c*, то *n*, если существует, должно находиться в левом поддереве *c*. Поэтому, вернуться к шагу 1, сделав *c* левым потомком текущего узла *c*.
5. Если значение *n* больше, чем значение *c*, то *n*, если существует, должно находиться в правом поддереве *c*. Поэтому, вернуться к шагу 1, сделав *c* правым потомком текущего узла *c*.

Мы применили эти шаги ранее при поиске узла 10. Мы начали с корневого узла и отметили, что 10 больше, чем 7, поэтому мы повторили наше сравнение с правым потомком корня, 11. Здесь мы отметили, что 10 меньше, чем 11, поэтому мы повторили наше сравнение с левым потомком узла 11. В этом месте мы нашли узел 10. При поиске узла 9, который не существует, мы прекратили бы сравнения на левом потомку узла 10, который является пустой (null) ссылкой. Тем самым мы заключили, что 9 не существует в BST.

4.1 Анализ алгоритма поиска BST

Для поиска узла в BST, на каждой стадии мы, в идеале, уменьшаем число узлов, которые нам нужно проверить, вдвое. Например, рассмотрим BST на Рисунке 6, которое содержит 15 узлов. Когда мы начинаем наш алгоритм поиска с корня дерева наше первое сравнение приведет нас либо к левому, либо к правому потомку корня. В любом случае, как только этот шаг сделан, то число узлов, которые нам нужно рассмотреть только что было разделено пополам, с 15 до 7. Аналогично, на следующем шаге число узлов снова уменьшится вдвое, от 7 до 3, и так далее.

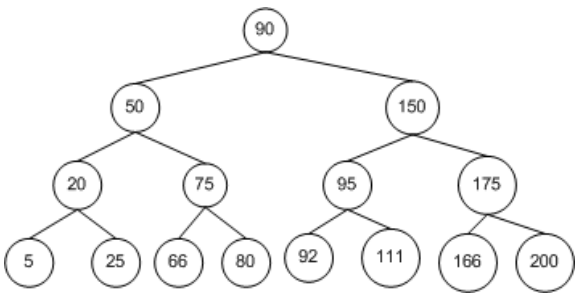

Рисунок 6. BST с 15 узлами

Важная концепция, которую нужно здесь понимать, состоит в том, что на каждом шаге алгоритма число узлов которые надлежит просмотреть уменьшается вдвое. Сравним это с поиском в массиве. Выполняя поиск в массиве мы должны просмотреть ВСЕ элементы, один за другим. То есть, производя поиск в массиве с n элементами, после того как мы проверим первый элемент нам все еще понадобиться проверить $n - 1$ элементов. Для BST с n узлами, однако, после проверки корня мы урезаем проблему до поиска в BST с $n/2$ узлами.

Поиск в бинарном дереве при анализе аналогичен поиску в отсортированном массиве. Например, представим, что вы хотите найти Джона Кинга в телефонном справочнике. Вы можете начать открыв телефонный справочник посередине. Здесь вы вероятно найдете людей чьи фамилии начинаются с буквы М. Так как К идет раньше М по алфавиту, то вы перелистаете половину между началом телефонного справочника и тем местом где вы его открыли, и возможно вы откроете на странице с фамилиями на И. Так как К следует после И, то вы откроете половину между И и М. На этот раз вы увидите К, где вы быстро найдете Джона Кинга.

Это аналогично поиску в BST. В идеально организованном (сбалансированном) BST средняя точка это корень. Далее мы проходим вниз по дереву перемещаясь к левому и правому потомкам. Эти приближения делят пространство поиска пополам на каждом шаге. Такие алгоритмы, которые проявляют это свойство имеют асимптотическое время выполнения $\log_2 n$, обычно сокращаемое до $\lg n$. Вспомните из наших математических рассуждений из части 1 этой серии статей, что $\log_2 n = y$ означает, что $2^y = n$. То есть, с ростом n, $\log_2 n$ растет очень медленно. Фактор роста $\log_2 n$ в сравнении с линейным ростом показан на графике на Рисунке 7. Из-за более медленного фактора роста $\log_2 n$ сравнительно с линейным временем алгоритмы, которые имеют асимптотическое время выполнения $\log_2 n$ называются сублинейными.

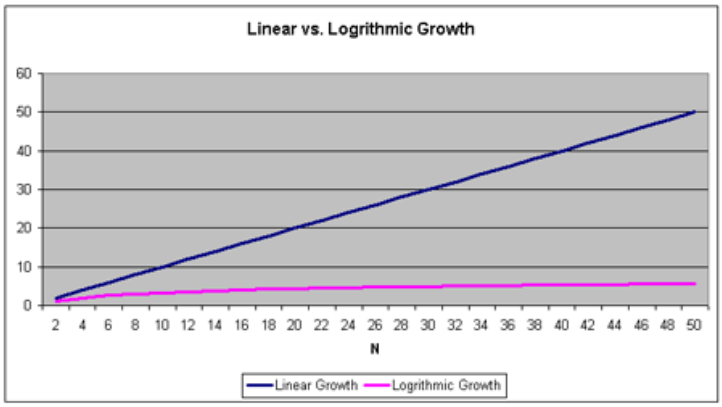

Рисунок 7. Сравнение функции линейного роста с ростом порядка $\log_2 n$

Linear vs. Logarithmic Growth – сравнение линейного и логарифмического роста
Linear Growth – линейный рост
Logarithmic Growth – логарифмический рост

В то время как логарифмическая кривая может показаться плоской (прямой), на самом деле она возрастает хотя и довольно медленно. Чтобы оценить разницу между линейным и сублинейным ростом рассмотрим поиск в массиве с 1,000 элементами в сравнении с поиском в BST с 1,000 элементов. Для массива мы должны просмотреть до 1,000

элементов. Для BST, в идеальном случае нам понадобиться просмотреть не более 10 узлов! (Отметьте, что $\log_{10}$ 1024 равен 10.)

На протяжении всего анализа алгоритма поиска в BST, я несколько раз повторял слово «в идеале». Это потому что время поиска для BST зависит от его топологии (*topology*) или того, как узлы расположены один относительно другого. Для бинарного дерева подобного представленному на Рисунке 6, каждая стадия сравнения в алгоритме поиска убирает из дальнейшего рассмотрения половину узлов. Однако, давайте рассмотрим BST показанное на Рисунке 8, чья топология аналогична тому как расположены элементы в массиве.

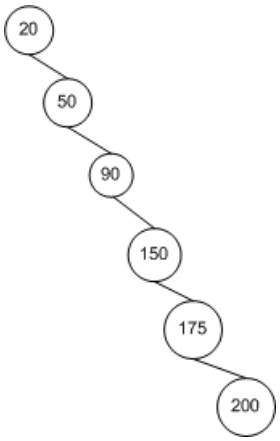

Рисунок 8. Пример BST, которое имеет линейное время поиска

Поиск в BST на Рисунке 8 потребует линейного времени так как после каждого сравнения проблемное пространство (problem space) уменьшится всего лишь на один узел, а не на половину от существующего количества узлов, как это происходит с BST на Рисунке 6.

Поэтому время необходимое для поиска в BST зависит от его топологии. В лучшем случае это время имеет порядок $\log_2 n$, однако в худшем случае поиск требует линейного времени. Как мы увидим в следующем разделе топология BST зависит от порядка в котором в дерево добавляются узлы. Поэтому порядок, в котором добавляются узлы в дерево влияет на время выполнения алгоритма поиска в BST.

4.2 Добавление узла в BST

Мы видели каким образом производить поиск в BST, чтобы определить существование определенного узла в дереве, но мы еще должны обратить внимание на то, как добавлять новый узел. При добавлении нового узла мы не можем осуществить это произвольным образом; более того, мы должны добавить новый узел таким образом, чтобы не нарушалось свойство бинарного дерева поиска.

При вставке узла мы всегда добавляем его как лист дерева. Единственная проблема состоит в нахождении узла BST, который станет родителем этого нового узла. Так же как и в случае алгоритма поиска, мы будем производить сравнения узла c и узла n, который необходимо добавить в дерево. Мы также будем отслеживать родителя узла c. В начале, c это корень BST, а его родитель это пустая ссылка. Нахождение нового родительского узла осуществляется с помощью следующего алгоритма:

1. Если *c* это пустая ссылка, то родителем будет родитель *n*. Если значение узла *n* меньше чем значение узла родителя, то *n* будет новым левым потомком данного родителя; в противном случае *n* будет новым правым потомком родителя.
2. Сравнить значения узлов *c* и *n*.
3. Если значение узла *c* равно значению узла *n*, то пользователь пытается добавить дубликат узла. В этом случае либо просто отбросьте новый узел, либо возбудите исключение. (Отметьте, что значение узла в BST должно быть уникальным.)
4. Если значение узла *n* меньше, чем значение узла *c*, то *n* должно очутиться в левом поддереве узла *c*. Пусть родителем станет *c*, а *c* примет значение левого потомка *c*, и затем вернуться к шагу 1.
5. Если значение узла *n* больше, чем значение узла *c*, то *n* должно очутиться в правом поддереве узла *c*. Пусть родителем станет *c*, а *c* примет значение правого потомка *c*, и затем вернуться к шагу 1.

Этот алгоритм завершается когда будет найден подходящий лист, который присоединяет новый узел к BST путем присоединения нового узла в качестве соответствующего потомка родителя (*parent*). Существует один особый случай о котором вы должны побеспокоится в алгоритме вставки: если BST не содержит корня, то родителем будет пустая ссылка, и поэтому шаг добавления нового узла в качестве потомка родителя будет опущен; более того, в этом случае корень BST должен указывать на новый узел.

Рисунок 9 графически изображает процесс вставки в BST.

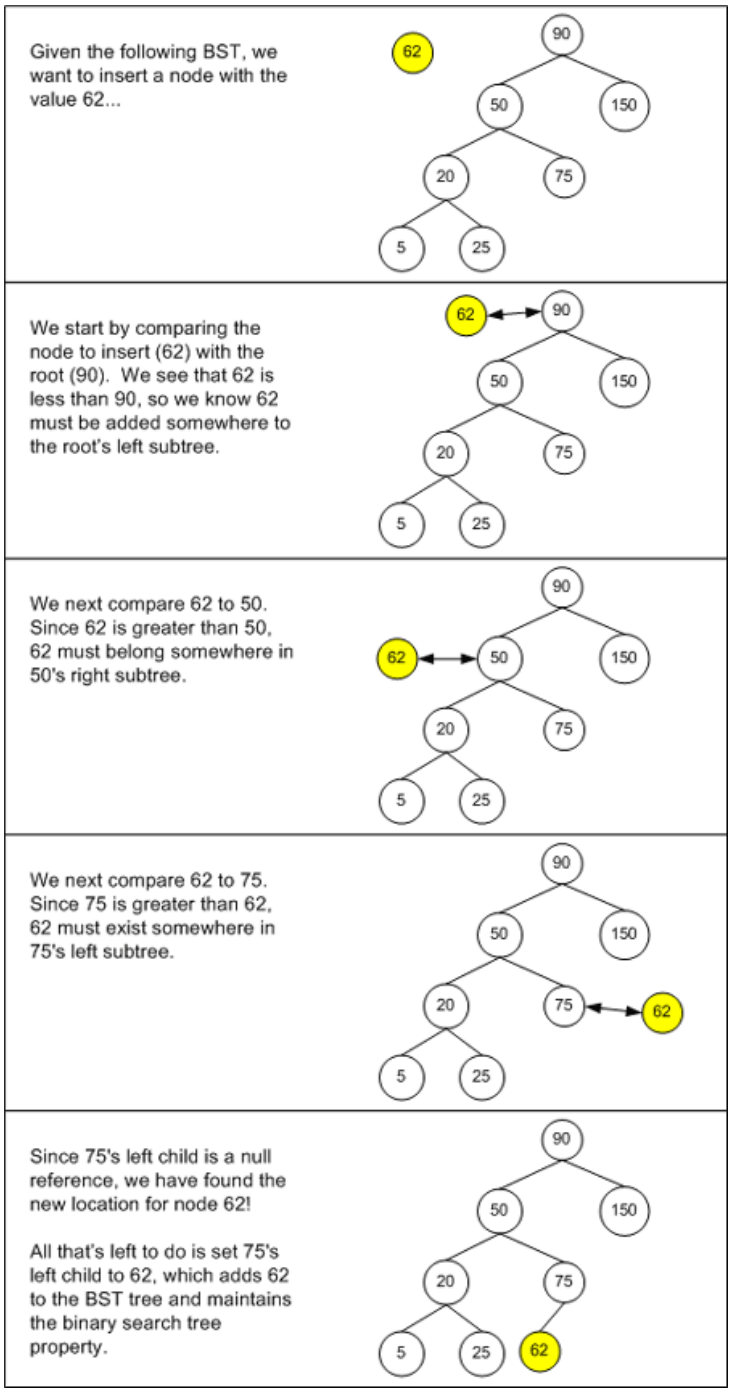

Рисунок 9. Вставка узла в BST

Пусть дано следующее BST и мы хотим вставить в него узел со значением 62 ...

Мы начинаем со сравнения вставляемого узла (62) со значением корня (90). Мы видим, что 62 меньше, чем 90, поэтому мы знаем, что 62 должно быть добавлено в левое поддерево корня.

Далее мы сравниваем 62 с 50. Так как 62 больше, чем 50, то 62 должно находиться где-то в правом поддереве 50.

Затем мы сравниваем 62 с 75. Так как 75 больше, чем 62, то 62 должно находиться где-то в левом поддереве 75.

Далее мы сравниваем 62 с 75. Так как 75 больше, чем 62, то 62 должно существовать где-то в левом поддереве узла 75.

Так как левый потомок узла 75 является пустой ссылкой, то мы нашли новое место для узла 62!
Все, что осталось сделать это назначить левого потомка узла 75 узлу 62, что добавляет узел 62 в дерево и сохраняет свойство бинарного дерева поиска.

И алгоритм поиска в BST и алгоритм вставки узла оба имеют одинаковое время выполнения — $\log_2 n$ в лучшем случае, и линейное в худшем случае. Время выполнения алгоритма вставки узла имитирует время выполнения поиска потому что по существу использует ту же самую тактику, что и алгоритм поиска, при нахождении места, куда будет вставляться новый узел.

4.2.1 Порядок добавления узлов определяет топологию BST

Так как новые узлы вставляются в BST как листья, то порядок вставки непосредственно влияет на топологию самого BST. Например, представим, что мы добавляем следующие узлы в BST: 1, 2, 3, 4, 5, и 6. Когда добавляется 1, то он добавляется как корень. Далее 2 добавляется как левый потомок узла 1. 3 вставляется как правый потомок узла 2, 4 как правый потомок узла 3 и так далее. Результирующее BST имеет структуру показанную на Рисунке 8.

Если же значения 1, 2, 3, 4, 5, и 6 вставляются более разумным способом, то BST будет иметь большую ширину и выглядеть так как это показано на Рисунке 6. Идеальный порядок вставки может иметь вид: 4, 2, 5, 1, 3, 6. Это приведет к тому, что 4 будет корнем, 2 – левым потомком узла 4, 5 правым потомком узла 4, 1 и 3 левым и правым потомками соответственно узла 2, а 6 правым потомком узла 5.

Так как топология BST может существенно повлиять на время выполнения операций поиска, вставки и (как мы это увидим в следующем разделе) удаления, вставка данных в возрастающем или убывающем порядке может нанести сокрушительный удар по эффективности BST. Мы обсудим эту тему более детально в конце статьи в разделе «Бинарные деревья поиска в реальном мире».

4.3 Удаление узлов из BST

Удаление узлов из BST несколько более сложная операция, чем вставка узла, потому что удаление узла, который имеет потомков требует, чтобы какой-либо другой узел был выбран для замещения дырки, созданной удаленным узлом. Если узел для замещения этой дырки не будет выбран достаточно тщательно, то может быть нарушено свойство бинарного дерева поиска. Например, рассмотрим BST на Рисунке 6. Если удаляется узел 150, то какой-то узел должен быть перемещен в дырку созданную в результате удаления узла 150. Если мы произвольно выберем узел для перемещения, скажем 92, то свойство BST будет нарушено, так как новое левое поддерево узла 92 будет содержать узлы 95 и 111, оба из которых больше чем 92 и поэтому нарушают свойство бинарного дерева поиска.

Первый шаг в алгоритме удаления узла состоит сперва в нахождении узла подлежащего удалению. Это можно сделать используя алгоритм поиска рассмотренный ранее и поэтому имеющий время выполнения $\log_2 n$. Далее, необходимо выбрать узел BST который заменит удаляемый узел. Существует три случая, которые необходимо рассмотреть при выборе замещающего узла, каждый из которых проиллюстрирован на Рисунке 10.

- **Случай 1:** Если узел подлежащий удалению не имеет правого потомка, то левый потомок узла может быть использован в качестве замещающего. Свойство бинарного дерева поиска выполняется потому что мы знаем, левое поддерево удаляемого узла само выполняет свойство бинарного дерева поиска, и что значения в узлах левого поддерева либо все меньше чем, либо все больше чем значение в родительском узле удаляемого узла, в зависимости от того является ли удаляемый узел левым или правым потомком. Поэтому замещение удаленного узла его левым поддеревом будет выполнять свойство бинарного дерева поиска.
- **Случай 2:** Если правый потомок удаляемого узла не имеет левого потомка, то правый потомок удаляемого узла может заменить сам удаляемый узел. Свойство бинарного дерева поиска выполняется потому что правый потомок удаляемого узла больше, чем все узлы в левом поддереве удаляемого узла и либо больше чем, либо меньше чем значение родителя удаляемого узла, в зависимости от того был ли удаляемый узел правым или левым потомком родителя. Поэтому замещение удаленного узла его правым потомком будет выполнять свойство бинарного дерева поиска.
- **Случай 3:** И наконец, если правый потомок удаляемого узла имеет левого потомка, то удаляемый узел необходимо заменить самым левым потомком правого потомка удаляемого узла. То есть мы заменяем удаляемый узел наименьшим значением правого поддерева удаляемого узла.

  Эта замена сохраняет свойство бинарного дерева поиска, потому что она выбирает наименьший узел из правого поддерева удаляемого узла, который гарантированно больше, чем все узлы в правом поддереве удаляемого узла. Также, так как это наименьший узел из правого поддерева удаляемого узла, то помещение его на место удаляемого узла приведет к тому, что все узлы правого поддерева будут иметь большее значение.

Comment [w2]: Помните, что для любого BST, наименьшее значение в дереве является его самым левым узлом, а наибольшее значение его самым правым узлом

Рисунок 10 иллюстрирует выбор замещающего узла для каждого из трех случаев.

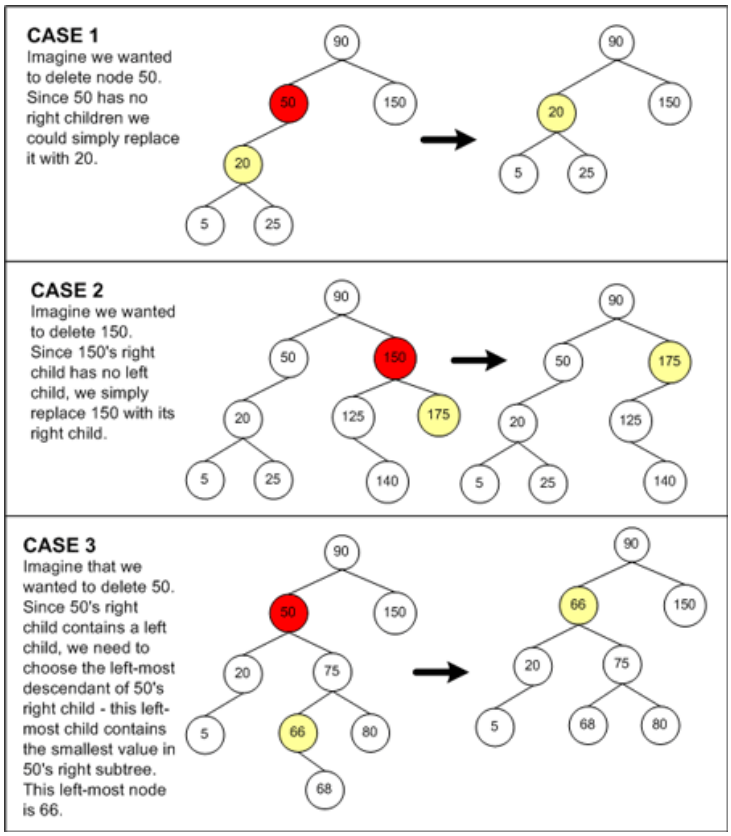

Рисунок 10. Случаи подлежащие рассмотрению при удалении узла

СЛУЧАЙ 1. Предположим, что мы хотим удалить узел 50. Так как узел 50 не имеет правого потомка мы можем просто заменить его узлом 20.
СЛУЧАЙ 2. Предположим, что мы хотим удалить узел 150. Так как правый потомок узла 150 не имеет левого потомка, то мы просто замещаем узел 150 его правым потомком.
СЛУЧАЙ 3. Предположим, что мы хотим удалить узел 50. Так как правый потомок узла 50 имеет левого потомка, то нам необходимо выбрать самого левого потомка правого потомка узла 50 – этот самый левый потомок содержит наименьшее значение в правом поддереве узла 50. Этот самый левый узел 66.

Так же как и с алгоритмами поиска и вставки асимптотическое время выполнения операции удаления зависит от топологии BST. В идеальном случае, время выполнения имеет порядок $\log_2 n$. Однако, в худшем случае, он требует линейного времени.

4.4 Обход узлов в BST

Массив, у которого элементы расположены непрерывно, позволяет проходить по его элементам непосредственно: начните с первого элемента массива и затем переходите к последующему элементу. Для BST существует три различных вида обхода элементов, которые широко применяются:

- Обход в прямом порядке (preorder traversal)
- Обход в симметричном (внутреннем) порядке (inorder traversal)

- Обход в обратном порядке (postorder traversal)

По существу, все три обхода работают приблизительно одинаково. Они начинают с корня и посещают текущий узел и его потомков. Разница между этими тремя методами обхода состоит в порядке в котором происходит посещение самого узла и его потомков. Чтобы помочь разъяснить это рассмотрим BST на Рисунке 11. (Отметьте, что BST на Рисунке 11 это то же самое BST, что и на Рисунке 6 и приведено здесь только для удобства.)

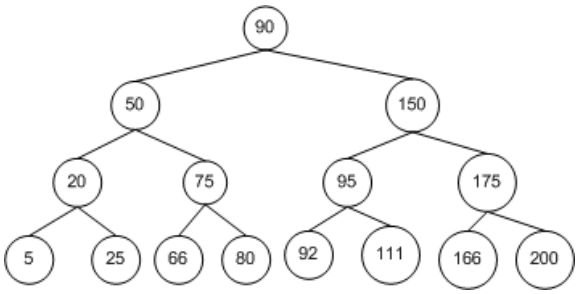

Рисунок 11.Пример бинарного дерева поиска

4.4.1 Обход в прямом порядке

Обход в прямом порядке начинается с текущего узла — назовем его *c* — затем посещается его левый потомок и затем его правый потомок. Начиная с корня BST в качестве *c*, этот алгоритм можно записать следующим образом:

1. Посетить *c*. Это может означать вывод на печать значения узла, добавления узла в Список (List), или что-нибудь другое. Это зависит от того, что вы хотите выполнить при обходе BST.
2. Повторить шаг 1 используя левого потомка *c*.
3. Повторить шаг 1 используя правого потомка *c*.

Представьте, что на шаге 1 алгоритма мы выводим на печать значение *c*. В этом случае возникает вопрос, как будет выглядеть вывод на печать BST на Рисунке с использованием обхода в прямом порядке? Начиная с шага 1 мы напечатали бы значение корня. Шаг 2 заставил бы нас повторить шаг 1 с левым потомком корня, то есть мы напечатали бы число 50. Шаг 2 заставил бы нас повторить шаг 1 с левым потомком левого потомка корня, и мы напечатали бы 20. Это повторялось бы до самого низа дерева по его левой стороне. Когда бы мы достигли 5, мы сперва бы напечатали это значение (шаг 1). Так как этот узел не имеет левых потомков, то мы вернулись бы к узлу 20, и выполнили бы шаг 3, который повторяет шаг 1 с правым потомком узла 20, то есть с узлом 25. Так как узел 25 не имеет потомков, мы бы вернулись к узлу 20, но мы уже выполнили все три шага для узла 20, поэтому мы вернулись бы к узлу 50, и затем применили бы шаг 3 для узла 50, который повторяет шаг 1 для правого потомка узла 50. Этот процесс продолжался бы до тех пор пока каждый узел BST не был бы посещен. В результате обхода в прямом порядке BST на Рисунке 11 мы бы получили следующий вывод: 90, 50, 20, 5, 25, 75, 66, 80, 150, 95, 92, 111, 175, 166, 200.

Следующий код представляет метод для перебора элементов BST с обходом в прямом порядке. Заметьте, что этот метод получает в качестве входного значения экземпляр класса `BinaryTreeNode`. Этот входной узел представляет собой узел *c* из шагов алгоритма

обхода. Помните, что обход в прямом порядке дерева бинарного поиска должен начинаться с вызова этого метода и передачи в него корня BST.

```
void PreorderTraversal(Node current)
{
   if (current != null)
   {
          // Вывести значение текущего узла
      Console.WriteLine(current.Value);

          // Рекурсивно печатать левого и правого потомков
      PreorderTraversal(current.Left);
      PreorderTraversal(current.Right);
   }
}
```

4.4.2 Обход в симметричном (внутреннем) порядке

Обход в симметричном порядке начинается с посещения левого потомка текущего узла, затем самого текущего узла, а затем правого потомка. Начиная с корня BST обозначаемого как *c*, этот алгоритм может быть записан следующим образом:

1. Повторить шаг 1 используя левого потомка узла c.
2. Посетить c. Это может означать печать значения узла, добавление узла в ArrayList, или что-нибудь еще. Это зависит от того, что вы желаете осуществить обходя BST.
3. Повторить шаг 1 используя правого потомка узла c.

Код метода `InorderTraversal()` аналогичен коду `PreorderTraversal()` за исключением того, что добавление данных в узел `current` в `StringBuilder` производится после другого вызова `InorderTraversal()`, передавая ему левого потомка узла `current`.

```
void InorderTraversal(Node current)
{
   if (current != null)
   {
          // Посетить левого потомка...
      InorderTraversal(current.Left);

          // Вывести значение текущего узла
      Console.WriteLine(current.Value);

          // Посетить правого потомка...
      InorderTraversal(current.Right);
   }
}
```

Применение обхода в симметричном (внутреннем) порядке к BST на Рисунке 11, приведет к следующему выводу: 5, 20, 25, 50, 66, 75, 80, 90, 92, 95, 111, 150, 166, 175, 200. Заметьте, что сформированный результат сформирован в возрастающем порядке.

4.4.3 Обход в обратном порядке

И наконец, обход в обратном порядке начинается с посещения левого потомка текущего узла, затем посещается правый потомок, и в конце сам узел. Начиная с корня BST обозначаемого как *c*, этот алгоритм может быть записан в следующем виде:

1. Повторить шаг 1 используя левого потомка узла c.

2. Повторить шаг 1 используя правого потомка узла c.
3. Посетить узел c. Это может означать печать значения узла, добавление узла в ArrayList, или что-нибудь еще. Это зависит от того, что вы желаете осуществить обходя BST.

Вывод для обхода в обратном порядке BST представленного на Рисунке 11 будет: 5, 25, 20, 66, 80, 75, 50, 92, 111, 95, 166, 200, 175, 150, 90.

Помните, что все три варианта обхода показывают асимптотическое линейное время выполнения. Это потому что каждый вариант обхода требует посещения каждого узла в BST в точности один раз. Итак, если число узлов в BST удвоится, то количество работы требуемой для обхода дерева также удвоится.

4.4.4 Стоимость рекурсии

Рекурсивные функции очень часть подходят для визуализации алгоритма, так как они зачастую красноречиво описывают алгоритм в виде нескольких строк кода. Однако, на практике при проходе по элементам структуры данных рекурсивные функции обычно суб-оптимальны. Поэтому класс `BinarySearchTree` использует нерекурсивное решение для итерации по элементам.

4.5 Реализация класса BST

.NET Framework Base Class Library не предлагает для использования класс бинарного дерева поиска, поэтому давайте реализуем его самостоятельно. Класс `BinarySearchTree` может использовать класс `BinaryTreeNode`, который мы изучили раньше. В следующих подразделах мы мы рассмотрим основные методы класса.

4.5.1 Поиск узла

Причина по которой изучение структуры BST является достаточно важным состоит в том, что она предлагает сублинейное время поиска. Поэтому имеет смысл сперва изучить метод `Contains()` бинарного дерева поиска. Метод `Contains()` получает единственное значение в качестве входного параметра и возвращает логическое значение говорящее о том, существует ли требуемое значение в BST.

`Contains()` начинает с корня и итеративно опускается вниз по дереву до тех пор пока либо не будет достигнута пустая (`null`) ссылка, и в данном случае будет возвращено значение `false`, либо не будет найден искомый узел, и в данном случае будет возвращено значение `true`. В цикле `while`, метод `Contains()` сравнивает содержимое внутренней переменной `Value` текущего экземпляра класса `BinaryTreeNode` с искомыми данными, и продвигается вниз по дереву соответственно к левому или правому поддереву. Сравнение производится с помощью внутренней переменной класса, `comparer`, имеющей тип `IComparer<T>` (где `T` это тип задаваемый с использованием синтаксиса параметризуемых типов для BST). По умолчанию, `comparer` имеет значение стандартного класса `Comparer` для типа T, а конструктор класса `BST` имеет возможность задать ваш собственный экземпляр класса `Comparer`.

```
public bool Contains(T data)
{
    // искать узел в дереве, который содержит требуемые данные
    BinaryTreeNode<T> current = root;
    int result;
```

```
    while (current != null)
    {
        result = comparer.Compare(current.Value, data);
        if (result == 0)
            // мы нашли данные
            return true;
        else if (result > 0)
            // current.Value > data, искать в левом поддереве
            current = current.Left;
        else if (result < 0)
            // current.Value < data, искать в правом поддереве
            current = current.Right;
    }

    return false;       // данные не найдены
}
```

4.5.2 Добавление узла в BST

В отличие от ранее созданного нами класса `BinaryTree`, класс `BinarySearchTree` не предоставляет непосредственного доступа к корню дерева. Вместо этого узлы добавляются в BST с помощью метода `Add()`. `Add()` получает в качестве входного значения элемент, который нужно добавить в BST, который затем далее опускается вниз по дереву, в поисках своего нового родителя. (Вспомните, что любые новые узлы, добавляемые в BST будут добавлены как листья дерева.) Как только будет найден родитель узла, то узел станет его левым либо правым потомком, в зависимости от того меньше или больше его значение значения родительского узла.

```
public virtual void Add(T data)
{
    // создать новый экземпляр Node
    BinaryTreeNode<T> n = new BinaryTreeNode<T>(data);
    int result;

    // теперь можно вставить n в дерево
    BinaryTreeNode<T> current = root, parent = null;
    while (current != null)
    {
        result = comparer.Compare(current.Value, data);
        if (result == 0)
            // равны - попытка вставить дубликат - ничего не делать
            return;
        else if (result > 0)
        {
            // current.Value > data, добавить n в левое поддерево
            parent = current;
            current = current.Left;
        }
        else if (result < 0)
        {
            // current.Value < data, добавить n в правое поддерево
            parent = current;
            current = current.Right;
        }
    }

    // Мы готовы добавить узел!
    count++;
    if (parent == null)
        // дерево было пустым, сделать n корнем
        root = n;
```

```
    else
    {
        result = comparer.Compare(parent.Value, data);
        if (result > 0)
            // parent.Value > data, поэтому n должен быть добавлен в левое
поддерево
            parent.Left = n;
        else
            // parent.Value < data, поэтому n должен быть добавлен в правое
поддерево
            parent.Right = n;
    }
}
```

Как вы можете увидеть при изучении кода, если пользователь пытается добавить дубликат, то метод `Add()` ничего не делает. То есть BST не может содержать узлов с одинаковыми значениями. Если вы хотите, то вместо простого выхода из процедуры, вы можете внести изменения в этот код таким образом, чтобы попытка добавления дубликата приводила к возникновению исключения.

4.5.3 Удаление узла из BST

Вспомните, что удаление узла из BST самая мудреная операция BST. Мудреность операции возникает из-за того факта, что удаление узла из BST требует, чтобы замещающий узел был выбран соответствующим образом для того, чтобы занять место удаленного узла. Необходимо очень внимательно выбрать этот замещающий узел таким образом, чтобы не нарушить свойство бинарного дерева поиска.

Ранее, в разделе «Удаление узлов из BST», мы обсуждали три различных сценария для принятия решения какой узел выбрать для замещения удаляемого узла. Эти сценарии проиллюстрированы на Рисунке 10. Ниже вы можете увидеть как эти сценарии реализованы в методе `Remove()`.

```
public bool Remove(T data)
{
    // сперва удостоверимся, что в дереве существуют узлы
    if (root == null)
        return false;       // нет элементов для удаления

    // Теперь попытаемся найти данные в дереве
    BinaryTreeNode<T> current = root, parent = null;
    int result = comparer.Compare(current.Value, data);
    while (result != 0)
    {
        if (result > 0)
        {
            // current.Value > data, если данные существуют, то в левом
поддереве
            parent = current;
            current = current.Left;
        }
        else if (result < 0)
        {
            // current.Value < data, если данные существуют, то в правом
поддереве
            parent = current;
            current = current.Right;
        }
```

```
        // Если current == null, то мы не нашли элемент для удаления
        if (current == null)
            return false;
        else
            result = comparer.Compare(current.Value, data);
    }

    // В этом месте мы нашли узел для удаления
    count--;

    // Теперь нам нужно «перепрошить» дерево
    // СЛУЧАЙ 1: Если текущий узел не имеет правого потомка, то левый
    // потомок текущего узла становится узлом на который ссылается
    // родитель
    if (current.Right == null)
    {
        if (parent == null)
            root = current.Left;
        else
        {
            result = comparer.Compare(parent.Value, current.Value);
            if (result > 0)
                // parent.Value > current.Value, поэтому сделать левого
потомка текущего узла левым потомком родителя
                parent.Left = current.Left;
            else if (result < 0)
                // parent.Value < current.Value, поэтому сделать левого
потомка текущего узла правым потомком родителя
                parent.Right = current.Left;
        }
    }
    // СЛУЧАЙ 2: Если правый потомок текущего узла не имеет левого потомка,
    // то правый потомок текущего узла замещает текущий узел в дереве
    else if (current.Right.Left == null)
    {
        current.Right.Left = current.Left;

        if (parent == null)
            root = current.Right;
        else
        {
            result = comparer.Compare(parent.Value, current.Value);
            if (result > 0)
                // parent.Value > current.Value, поэтому сделать правого
потомка текущего узла левым потомком родителя
                parent.Left = current.Right;
            else if (result < 0)
                // parent.Value < current.Value, поэтому сделать правого
потомка текущего узла правым потомком родителя
                parent.Right = current.Right;
        }
    }
    // СЛУЧАЙ 3: Если правый потомок текущего узла имеет левого потомка,
    // то заменить текущий узел самым левым наследником правого потомка
    // текущего узла
    else
    {
        // Сперва мы должны найти самого левого потомка правого узла
        BinaryTreeNode<T> leftmost = current.Right.Left, lmParent =
current.Right;
        while (leftmost.Left != null)
        {
            lmParent = leftmost;
            leftmost = leftmost.Left;
```

```
        }

        // левое поддерево родителя становится правым поддеревом самого
левого
        lmParent.Left = leftmost.Right;

        leftmost.Left = current.Left;
        leftmost.Right = current.Right;

        if (parent == null)
            root = leftmost;
        else
        {
            result = comparer.Compare(parent.Value, current.Value);
            if (result > 0)
                // parent.Value > current.Value, поэтому сделать самый левый
узел левым потомком родителя
                parent.Left = leftmost;
            else if (result < 0)
                // parent.Value < current.Value, поэтому сделать самый левый
узел правым потомком родителя
                parent.Right = leftmost;
        }
    }

    return true;
}
```

Метод `Remove()` возвращает логическое значение, означающее, был ли узел успешно удален из бинарного дерева поиска. «Ложь» возвращается в том случае, если элемент подлежащий удалению не был найден в дереве.

4.5.4 Оставшиеся свойства и методы BST

Осталось несколько свойств и методов BST, которые не рассмотрены в этой статье. Оставшиеся методы и свойства это:

- `Clear()`: удаляет все узлы из BST.
- `CopyTo(Array, index[, TraversalMethods])`: копирует содержимое BST в переданный массив. По умолчанию, используется обход в симметричном (внутреннем) порядке, хотя можно задать специфический метод обхода. Перечисление `TraversalMethods` имеет три варианта `Preorder`, `Inorder`, и `Postorder`.
- `GetEnumerator([TraversalMethods])`: обеспечивает итерацию BST используя обход в симметричном порядке по умолчанию. Возможно указание других методов обхода.
- `Count`: общедоступное свойство только для чтения, которое возвращает число узлов в BST.
- `Preorder`, `Inorder`, and `Postorder`: эти три свойства возвращают экземпляры `IEnumerable<T>` которые могут быть использованы для перечисления элементов BST in a specified traversal order.

Класс `BST` реализует интерфейсы `ICollection`, `ICollection<T>`, `IEnumerable`, и `IEnumerable<T>` interfaces.

5 Бинарные деревья поиска в реальном мире

В то время как бинарные деревья поиска показывают сублинейное время выполнения операций вставки, поиска и удаления, это время выполнения зависти от топологии дерева. Топология же зависит от порядка в котором данные добавляются в BST. Добавляемые данные, которые являются упорядоченными или почти упорядоченными приведут к топологии BST, которая выглядит как длинное и тонкое дерево, а не широкое и короткое. Во многих сценариях реального мира данные существуют в упорядоченном или почти упорядоченном виде.

Основная проблема с BST состоит в том, что они очень легко могут стать несбалансированными (*unbalanced*). Сбалансированное (*balaced*) бинарное дерево это такое дерево, которое обеспечивает хорошее отношение ширины дерева к его глубине. Мы исследуем в следующей статье нашей серии специальный класс BST, которые являются самобалансирующимися. То есть, когда добавляются новые узлы или удаляются существующие узлы, то такие BST автоматически корректируют свою топологию, чтобы обеспечить оптимальный баланс. В случае идеального баланса время поиска, вставки и удаления даже в худшем случае составляет $\log_2 n$.

В следующей статье мы также рассмотрим пару производных от самобалансирующихся BST, включая красно-черные деревья, а затем остановим наше внимание на структуре известной как SkipList, которая предоставляет преимущества самобалансирующихся бинарных деревьев, но без необходимости изменения топологии. Изменение топологии требует работы с большим объемом сложного кода, который является трудным для понимания.

Ну, теперь пока, счастливого программирования!