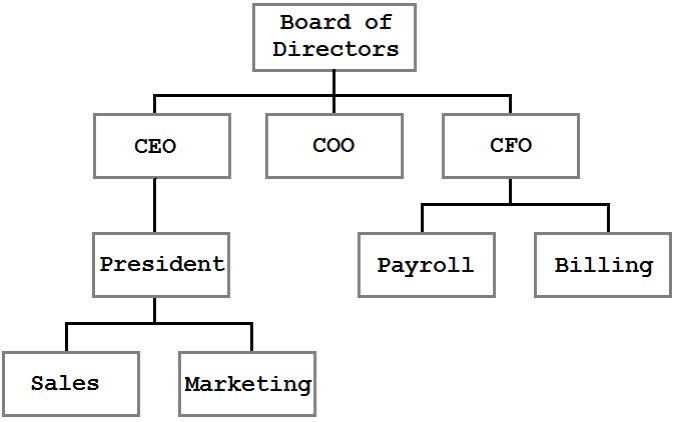
**Дерева**

Дерево - це структура, в якій у кожного вузла може бути нуль або більше подсистемами - «дітей». Наприклад, дерево може виглядати так:



Структура організації

Це дерево показує структуру компанії. Вузли представляють людей або підрозділи, лінії - зв'язки і відносини. Дерево - це найефективніший спосіб представлення та зберігання такої інформації.

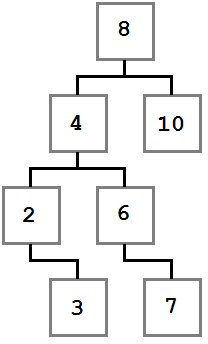
Дерево на зображенні вище дуже просте. Воно відображає лише ставлення спорідненості категорій, але не накладає ніяких обмежень на свою структуру. У генерального директора може бути як один безпосередній підлеглий, так і кілька або жодного. На малюнку відділ продажів знаходиться лівіше відділу маркетингу, але порядок насправді не має значення. Єдине обмеження дерева - кожен вузол може мати не більше одного з батьків. Самий верхній вузол (рада директорів, в нашому випадку) батьків не має. Цей вузол називається «кореневих», або «коренем».

**Двійкове дерево пошуку**

Двійкове дерево пошуку схоже на дерево з прикладу вище, але будується за певними правилами:

* У кожного вузла не більше двох дітей.
* Будь-яке значення менше значення вузла стає лівим дитиною або дитиною лівого дитини.
* Будь-яке значення більше або дорівнює значенню вузла стає правим дитиною або дитиною правого дитини.

Давайте подивимося на дерево, побудоване за цими правилами:



Двійкове дерево пошуку

Зверніть увагу, як зазначені обмеження впливають на структуру дерева. Кожне значення зліва від кореня (8) менше восьми, кожне значення праворуч - більше або дорівнює кореню. Це правило застосовується до будь-якого вузла дерева.

З огляду на це, давайте уявимо, як можна побудувати таке дерево. Оскільки спочатку дерево було порожнім, перше доданий значення - вісімка - стало його коренем.

Ми не знаємо точно, в якому порядку додавалися інші значення, але можемо уявити один з можливих шляхів. Вузли додаються методом Add, який приймає додається значення.

BinaryTree tree = new BinaryTree();

tree.Add(8);

tree.Add(4);

tree.Add(2);

tree.Add(3);

tree.Add(10);

tree.Add(6);

tree.Add(7);

Розглянемо докладніше перші кроки.

В першу чергу додається 8. Це значення стає коренем дерева. Потім ми додаємо 4. Оскільки 4 менше 8, ми кладемо її в лівого дитини, згідно з правилом 2. Оскільки у вузла з вісімкою немає дітей зліва, 4 стає єдиним лівим дитиною.

Після цього ми додаємо 2. 2 менше 8, тому йдемо наліво. Так як зліва вже є значення, порівнюємо його з вставляються. 2 менше 4, а у четвірки немає дітей зліва, тому 2 стає лівим дитиною 4.

Потім ми додаємо трійку. Вона йде лівіше 8 і 4. Але так як 3 більше, ніж 2, вона стає правим дитиною 2, відповідно до третього правилом.

Послідовне порівняння вставляється значення з потенційним батьком триває до тих пір, поки не буде знайдено місце для вставки, і повторюється для кожного вставляється значення до тих пір, поки не буде побудовано все дерево цілком.

#### Клас BinaryTreeNode

Клас BinaryTreeNode представляє один вузол двійкового дерева. Він містить посилання на ліве і праве піддерева (якщо поддерева немає, посилання має значення null), дані вузла і метод IComparable.CompareTo для порівняння вузлів. Він стане в нагоді для визначення, в яке поддерево повинен йти даний вузол. Як бачите, клас BinaryTreeNode дуже простий:

class BinaryTreeNode : IComparable

where TNode : IComparable

{

public BinaryTreeNode(TNode value)

{

Value = value;

}

public BinaryTreeNode Left { get; set; }

public BinaryTreeNode Right { get; set; }

public TNode Value { get; private set; }

///

/// Порівнює поточний вузол з даними.

///

/// Порівняння проводиться по полю Value.

/// Метод повертає 1, якщо значення поточного вузла більше,

/// ніж переданого методу, -1, якщо менше і 0, якщо вони рівні

public int CompareTo(TNode other)

{

return Value.CompareTo(other);

}

}

#### Клас BinaryTree

Клас BinaryTree надає основні методи для маніпуляцій з даними: вставка елемента (Add), видалення (Remove), метод Contains для перевірки, чи є таке значення в дереві, кілька методів для обходу дерева різними способами, метод Count і Clear.

Крім того, в класі є посилання на кореневий вузол дерева і поле з загальною кількістю вузлів.

public class BinaryTree : IEnumerable

where T : IComparable

{

private BinaryTreeNode \_head;

private int \_count;

public void Add(T value)

{

throw new NotImplementedException();

}

public bool Contains(T value)

{

throw new NotImplementedException();

}

public bool Remove(T value)

{

throw new NotImplementedException();

}

public void PreOrderTraversal(Action action)

{

throw new NotImplementedException();

}

public void PostOrderTraversal(Action action)

{

throw new NotImplementedException();

}

public void InOrderTraversal(Action action)

{

throw new NotImplementedException();

}

public IEnumerator GetEnumerator()

{

throw new NotImplementedException();

}

System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()

{

throw new NotImplementedException();

}

public void Clear()

{

throw new NotImplementedException();

}

public int Count

{

get;

}

}

#### Метод Add

* **Поведінка:** Додає елемент в дерево на коректну позицію.
* **Складність:** O (log n) в середньому; O (n) в гіршому випадку.

Додавання вузла не представляє особливої ​​складності. Воно стає ще простіше, якщо вирішувати цю задачу рекурсивно. Є всього два випадки, які треба врахувати:

1. Дерево пусте.

2. Дерево не пусте.

Якщо дерево порожнє, ми просто створюємо новий вузол і додаємо його в дерево. У другому випадку ми порівнюємо передане значення зі значенням в вузлі, починаючи від кореня. Якщо додається значення менше значення даного вузла, повторюємо ту ж процедуру для лівого піддерева. В іншому випадку - для правого.

public void Add(T value)

{

// Випадок 1: Якщо дерево порожнє, просто створюємо кореневої вузол.

if (\_head == null)

{

\_head = new BinaryTreeNode (value);

}

// Випадок 2: Дерево не пусте =>

// шукаємо правильне місце для вставки.

else {

AddTo(\_head, value);

}

\_count++;

}

// Рекурсивна ставка.

private void AddTo (BinaryTreeNode node, T value)

{

// Випадок 1: вставляти значення менше значення вузла

if (value.CompareTo (node.Value) <0)

{

// Якщо немає лівого піддерева, додаємо значення в лівого дитини,

if (node.Left == null)

{

node.Left = new BinaryTreeNode (value);

}

else

{

// в іншому випадку повторюємо для лівого піддерева.

AddTo (node.Left, value);

}

}

// Випадок 2: вставляти значення більше або дорівнює значенню вузла.

else

{

// Якщо немає правого піддерева, додаємо значення в правого дитини,

if (node.Right == null)

{

node.Right = new BinaryTreeNode (value);

}

else

{

// в іншому випадку повторюємо для правого піддерева.

AddTo(node.Right, value);

}

}

}

#### Метод Remove

* **Поведінка:** Видаляє перший вузол з заданим значенням.
* **Складність:** O (log n) в середньому; O (n) в гіршому випадку.

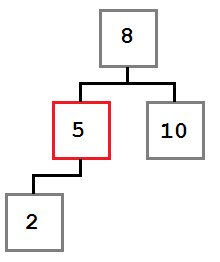
Видалення вузла з дерева - одна з тих операцій, які здаються простими, але насправді таять в собі чимало підводних каменів.

В цілому, алгоритм видалення елемента виглядає так:

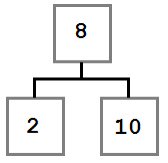
* Знайти вузол, який треба видалити.
* Видалити його.

Перший крок досить простий. Ми розглянемо пошук вузла в методі Contains нижче. Після того, як ми знайшли вузол, який необхідно видалити, у нас можливі три випадки.

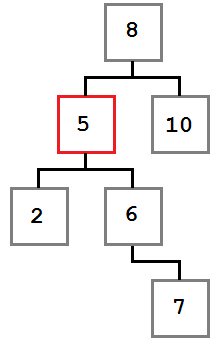
Випадок 1: У видаляється вузла немає правого дитини.



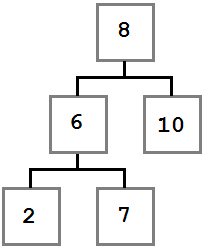
В цьому випадку ми просто переміщаємо лівого дитини (при його наявності) на місце видаленого вузла. В результаті дерево буде виглядати так:



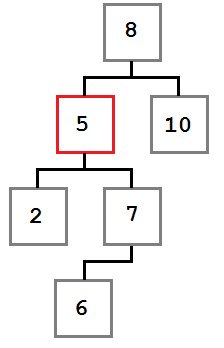
Випадок 2: У видаляється вузла є тільки правий дитина, у якого, в свою чергу немає лівого дитини.



У цьому випадку нам треба перемістити правого дитини видаляється вузла (6) на його місце. Після видалення дерево буде виглядати так:



Випадок 3: У видаляється вузла є перша дитина, у якого є лівий дитина.



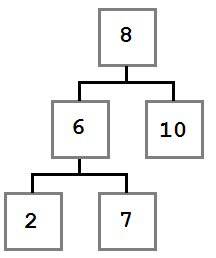
У цьому випадку місце видаляється вузла займає крайній лівий дитина правого дитини видаляється вузла.

Давайте подивимося, чому це так. Ми знаємо про поддереве, що починається з видаляється вузла наступне:

* Всі значення праворуч від нього більше або дорівнюють значенню самого вузла.
* Найменше значення правого піддерева - крайнє ліве.

Ми обіцяли помістити на місце видаленого вузол із значенням, меншим або рівним будь-якого вузла праворуч від нього. Для цього нам необхідно знайти найменше значення в правому піддереві. Тому ми беремо крайній лівий вузол правого піддерева.

Після видалення вузла дерево буде виглядати так:



Тепер, коли ми знаємо, як видаляти вузли, подивимося на код, який реалізує цей алгоритм.

Відзначимо, що метод FindWithParent (див. Метод Contains) повертає знайдений вузол і його батька, оскільки ми повинні замінити лівого або правого дитини батька видаляється вузла.

Ми, звичайно, можемо уникнути цього, якщо будемо зберігати посилання на батька в кожному вузлі, але це збільшить витрату пам'яті і Складність всіх алгоритмів, незважаючи на те, що посилання на батьківський вузол використовується тільки в одному.

public bool Remove(T value)

{

BinaryTreeNode current, parent;

// Знаходимо видаляється вузол.

current = FindWithParent(value, out parent);

if (current == null)

{

return false;

}

\_count--;

// Випадок 1: Якщо немає дітей справа, лівий дитина встає на місце видаленого.

if (current.Right == null)

{

if (parent == null)

{

\_head = current.Left;

}

else

{

int result = parent.CompareTo(current.Value);

if (result > 0)

{

// Якщо значення батька більше поточного,

// лівий дитина поточного вузла стає лівим дитиною батька.

parent.Left = current.Left;

}

else if (result < 0) { // Якщо значення батька менше поточного, // лівий дитина поточного вузла стає правим дитиною батька. parent.Right = current.Left; }}} // Випадок 2: Якщо у правого дитини немає дітей зліва // то він займає місце видаляється вузла.else if (current.Right.Left == null) { current.Right.Left = current.Left; if (parent == null) { \_head = current.Right; } else { int result = parent.CompareTo(current.Value); if (result > 0)

{

// Якщо значення батька більше поточного,

// правий дитина поточного вузла стає лівим дитиною батька.

parent.Left = current.Right;

}

else if (result < 0) { // Якщо значення батька менше поточного, // правий дитина поточного вузла стає правим дитиною батька. parent.Right = current.Right; }}} // Випадок 3: Якщо у правого дитини є діти зліва, крайній лівий дитина // з правого піддерева замінює видаляється вузол. else {// Знайдемо крайній лівий вузол.BinaryTreeNode leftmost = current.Right.Left; BinaryTreeNode leftmostParent = current.Right; while (leftmost.Left != null) { leftmostParent = leftmost; leftmost = leftmost.Left; } // Ліве піддерево батька стає правим піддерево крайнього лівого вузла. leftmostParent.Left = leftmost.Right; // Лівий і правий дитина поточного вузла стає лівим і правим дитиною крайнього лівого.leftmost.Left = current.Left; leftmost.Right = current.Right; if (parent == null) { \_head = leftmost; } else { int result = parent.CompareTo(current.Value); if (result > 0)

{

// Якщо значення батька більше поточного,

// крайній лівий вузол стає лівим дитиною батька.

parent.Left = leftmost;

}

else if (result <0)

{

// Якщо значення батька менше поточного,

// крайній лівий вузол стає правим дитиною батька.

parent.Right = leftmost;

}

}

}

return true;

}

#### Метод Contains

* **Поведінка:** Повертає true якщо значення міститься в дереві. В протилежному випадку повертає false.
* **Складність:** O (log n) в середньому; O (n) в гіршому випадку.

Метод Contains виконується за допомогою методу FindWithParent, який проходить по дереву, виконуючи в кожному вузлі наступні кроки:

1. Якщо поточний вузол null, повернути null.
2. Якщо значення теккущего узна одно шуканого, повернути поточний вузол.
3. Якщо шукане значення менше значення поточного вузла, встановити лівого дитини поточним вузлом і перейти до кроку 1.
4. В іншому випадку, встановити правого дитини поточним вузлом і перейти до кроку 1.

Оскільки Contains повертає логічне значення, воно визначається порівнянням результату виконання FindWithParent з null. Якщо FindWithParent повернув непорожній вузол, Contains повертає true.Метод FindWithParent также используется в методе Remove.

public bool Contains(T value)

{

// Пошук вузла здійснюється іншим методом.

BinaryTreeNode parent;

return FindWithParent (value, out parent)! = null;

}

///

/// Знаходить і повертає перший вузол з заданим значенням. якщо значення

///, не знайдено, повертає null. Також повертає батька знайденого вузла (або null)

/// для використання в методі Remove.

///

private BinaryTreeNode FindWithParent (T value, out BinaryTreeNode parent)

{

// Спробуємо знайти значення в дереві.

BinaryTreeNode current = \_head;

parent = null;

// До тих пір, поки не знайшли ...

while (current != null)

{

int result = current.CompareTo(value);

if (result > 0)

{

// Якщо шукане значення менше, йдемо наліво.

parent = current;

current = current.Left;

}

else if (result <0)

{

// Якщо шукане значення більше, йдемо направо.

parent = current;

current = current.Right;

}

else

{

// Якщо рівні, то зупиняємося

break;

}

}

return current;

}

#### Метод Count

* **Поведінка:** Повертає кількість вузлів дерева або 0, якщо дерево порожнє.
* **Складність:** O(1)

Це поле инкрементируется методом Add і декрементируется методом Remove.

public int Count

{

get

{

return \_count;

}

}

#### Метод Clear

* **Поведінка:** Видаляє всі вузли дерева.
* **Складність:** O(1)

public void Clear()

{

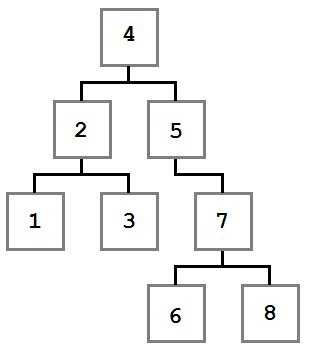
\_head = null;

\_count = 0;

}

### Обхід дерев

Обходи дерева - це сімейство алгоритмів, які дозволяють обробити кожен вузол в певному порядку. Для всіх алгоритмів обходу нижче в якості прикладу візьмемо наступне дерево:



Приклад дерева для обходу

Методи обходу в прикладах будуть приймати параметр Action <T>, який визначає дію, поізводімое над кожним вузлом.

Також, крім опису поведінки і алгоритмічної складності методу буде вказуватися порядок значень, отриманий при обході.

#### Метод Preorder (або префіксний обхід)

* **Поведінка:** Обходить дерево в префіксному порядку, виконуючи вказану дію над кожним вузлом.
* **Складність:** O(n)
* **Порядок обходу:** 4, 2, 1, 3, 5, 7, 6, 8

При префіксному обході алгоритм отримує значення поточного вузла перед тим, як перейти спочатку в ліве піддерево, а потім в праве. Починаючи від кореня, спочатку ми отримаємо значення 4. Потім таким же чином обходяться лівий дитина і його діти, потім правий дитина і все його діти.

Префіксний обхід зазвичай застосовується для копіювання дерева зі збереженням його структури.

public void PreOrderTraversal(Action action)

{

PreOrderTraversal(action, \_head);

}

private void PreOrderTraversal(Action action, BinaryTreeNode node)

{

if (node != null)

{

action(node.Value);

PreOrderTraversal(action, node.Left);

PreOrderTraversal(action, node.Right);

}

}

#### Метод Postorder (або постфіксний обхід)

* **Поведінка:** Обходить дерево в префіксному порядку, виконуючи вказану дію над кожним вузлом.
* **Складність:** O(n)
* **Порядок обходу:** 1, 3, 2, 6, 8, 7, 5, 4

При Постфіксний обході ми відвідуємо ліве піддерево, праве піддерево, а потім, після обходу всіх дітей, переходимо до самого вузла.

Постфіксний обхід часто використовується для повного видалення дерева, так як в деяких мовах програмування необхідно прибирати з пам'яті всі вузли явно, або для видалення поддерева. Оскільки корінь в даному випадку обробляється останнім, ми, таким чином, зменшуємо роботу, необхідну для видалення вузлів.

public void PostOrderTraversal(Action action)

{

PostOrderTraversal(action, \_head);

}

private void PostOrderTraversal(Action action, BinaryTreeNode node)

{

if (node != null)

{

PostOrderTraversal(action, node.Left);

PostOrderTraversal(action, node.Right);

action(node.Value);

}

}

#### Метод Inorder (або інфіксне обхід)

* **Поведінка:** Обходить дерево в інфіксне порядку, виконуючи вказану дію над кожним вузлом.
* **Складність:** O(n)
* **Порядок обходу:** 1, 2, 3, 4, 5, 6, 7, 8

Інфіксне обхід використовується тоді, коли нам треба обійти дерево в порядку, відповідному значенням вузлів. В наведеному вище прикладі в дереві знаходяться числові значення, тому ми обходимо їх від самого маленького до найбільшого. Тобто від лівих піддерев до правих через корінь.

У прикладі нижче показані два способи інфіксне обходу. Перший - рекурсивний. Він виконує вказане дію з кожним вузлом. Другий використовує стек і повертає ітератор для безпосереднього перебору.

Public void InOrderTraversal(Action action)

{

InOrderTraversal(action, \_head);

}

private void InOrderTraversal(Action action, BinaryTreeNode node)

{

if (node != null)

{

InOrderTraversal(action, node.Left);

action(node.Value);

InOrderTraversal(action, node.Right);

}

}

public IEnumerator InOrderTraversal()

{

// Це нерекурсивний алгоритм.

// Він використовує стек для того, щоб уникнути рекурсії.

if (\_head! = null)

{

// Стек для збереження пропущених вузлів.

Stack stack = new Stack ();

BinaryTreeNode current = \_head;

// Коли ми позбавляємося від рекурсії, нам необхідно

// запам'ятовувати, в яку сторони ми повинні рухатися.

bool goLeftNext = true;

// Кладемо в стек корінь.

stack.Push (current);

while (stack.Count> 0)

{

// Якщо ми йдемо наліво ...

if (goLeftNext)

{

// Кладемо все, крім самого лівого вузла на стек.

// Крайній лівий вузол ми повернемо з помощю yield.

while (current.Left! = null) {

stack.Push(current);

current = current.Left;

}

}

// Префіксний порядок: left-> yield-> right.

yield return current.Value;

// Якщо ми можемо піти направо, йдемо.

if (current.Right! = null)

{

current = current.Right;

// Після того, як ми пішли направо один раз,

// ми належним знову піти наліво.

goLeftNext = true;

}

else

{

// Якщо ми не можемо піти направо, ми повинні дістати батьківський вузол

// зі стека, обробити його і йти в його правого дитини.

current = stack.Pop ();

goLeftNext = false;

}

}

}

}

#### Метод GetEnumerator

* **Поведінка:** Повертає ітератор для обходу дерева інфіксне способом.
* **Складність:** Отримання ітератора - O (1). Обхід дерева - O (n).

public IEnumerator GetEnumerator()

{

return InOrderTraversal();

}

System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()

{

return GetEnumerator();

}