**Структури от данни**

Може би се чудите защо отделяме толкова голямо внимание на струк­турите данни и защо ги разглеждаме в такива големи детайли? Причината е, че сме си поставили за задача да ви направим мислещи софтуерни инженери. Без да познавате добре основните структури от данни в прог­рамирането и основните компютърни алгоритми, вие не можете да бъдете добри програмисти и рискувате да си останете обикновени "занаятчии". Който владее добре структурите от данни и алгоритми и успее да си развие мисленето в посока правилното им използване, има големи шан­сове да стане добър софтуерен инженер – който анализира проблемите в дълбочина и предлага ефективни решения.

## Сравнение на основните структури от данни

След като се запознахме с понятието сложност на алгоритъм, вече сме готови да направим съпоставка на основните структури от данни, които разгледахме до момента, и да оценим с каква сложност всяка от тях извършва основните операции като добавяне, търсене, изтриване и други. Така ще можем лесно да съобразяваме според операциите, които са ни необходими, коя структура от данни ще е най-подходяща. В таблицата по-долу са дадени сложностите на основните операции при основните струк­тури данни, които разгледахме в предходните глави:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **структура** | **доба­вяне** | **търсене** | **изтри­ване** | **достъп по индекс** |
| масив (**Т[]**) | O(N) | O(N) | O(N) | О(1) |
| свързан списък (**LinkedList<Т>**) | О(1) | O(N) | O(N) | O(N) |
| динамичен масив (**List<T>**) | О(1) | O(N) | O(N) | O(1) |
| стек (**Stack<Т>**) | О(1) | - | О(1) | - |
| опашка (**Queue<Т>**) | О(1) | - | О(1) | - |

##

## Кога да използваме дадена структура?

Нека разгледаме всяка от посочените в таблицата структури от данни поотделно и обясним в какви ситуации е подходящо да се ползва такава структура и как се получават сложностите, дадени в таблицата.

### Масив (T[])

Масивите са наредени съвкупности от фиксиран брой елементи от даден тип (например числа), до които достъпът става по индекс. Масивите пред­ставляват област от паметта с определен, предварително зададен размер. Добавянето на нов елемент в масив е много бавна операция, защото реално трябва да се задели нов масив с размерност по-голяма с 1 от теку­щата и да се прехвърлят старите елементи в новия масив. Търсенето в масив изисква сравнение на всеки елемент с търсената стойност. В средния случай са необходими N/2 сравнения. Изтриването от масив е много бавна операция, защото е свързана със заделяне на масив с размер с 1 по-малък от текущия и преместване на всички елементи без изтрития в новия масив. Достъпът по индекс става директно и затова е много бърза операция.

Масивите трябва да се ползват само когато трябва да обработим фиксиран брой елементи, до които е необходим достъп по индекс. Например, ако сортираме числа, можем да запишем числата в масив и да приложим някой от добре известните алгоритми за сортиране. Когато по време на работа е необходимо да променяме броя елементи, с които работим, масивът не е подходяща структура от данни.

|  |  |
| --- | --- |
| **clip_image001[33]** | **Използвайте масиви, когато трябва да обработите фикси­ран брой елементи, до които ви е необходим достъп по индекс.** |

###

### Свързан / двусвързан списък (LinkedList<Т>)

Свързаният списък и неговият вариант двусвързан списък съхраняват наредена съвкупност от елементи. Добавянето е бърза операция, но е малко по-бавна от добавяне в **List<T>**, защото всяко добавяне заделя памет. Заделянето на памет работи със скорост, която трудно може да бъде предвидена. Търсенето в свързан списък е бавна операция, защото е свързано с обхождане на всички негови елементи. Достъпът до елемент по индекс е бавна операция, защото в свързания списък няма индексиране и се налага обхождане на списъка, започвайки от началния елемент и придвижвайки се напред елемент по елемент. Изтриването на елемент по индекс е бавна операция, защото достигането до елемента с посочения индекс е бавна операция. Изтриването по стойност на елемент също е бавно, защото включва в себе си търсене.

Свързаният списък може бързо (с константна сложност) да добавя и изтрива елементи от двата си края, поради което е удобен за имплемен­тация на стекове, опашки и други подобни структури.

Свързан списък в практиката се използва много рядко, защото дина­мично-разширяемият масив(**List<Т>**) изпълнява почти всички операции, които могат да бъдат изпълнени с **LinkedList**, но за повечето от тях работи по-бързо и по-удобно.

Ползвайте **List<T>**, когато ви трябва свързан списък – той работи не по-бавно, а ви дава по-голяма бързина и удобство. Ползвайте **LinkedList**, ако има нужда от добавяне и изтриване на елементи в двата края на структурата.

|  |  |
| --- | --- |
| **clip_image001[34]** | **Използвайте свързан списък (LinkedList<T>), когато трябва да добавяте и изтри­вате елементи от двата края на списъка. В про­тивен случай ползвайтеList<T>.** |

###

### Динамичен масив (List<T>)

Динамичният масив (**List<T>**) е една от най-използваните в практиката структура от данни. Той няма фиксиран размер, както масивите, и позволява директен достъп по индекс, за разлика от свързания списък (**LinkedList<T>**). Динамичният масив е известен още с наименованията "списък реализиран с масив" и "динамично-разширяем масив".

**List<T>** вътрешно съхранява елементите си в масив, който има размер по-голям от броя съхранени елементи. При добавяне на елемент обикновено във вътрешния масив има свободно място и затова тази операция отнема константно време. Понякога масивът се препълва и се налага да се разшири. Това отнема линейно време, но се случва много рядко. В крайна сметка при голям брой добавяния усреднената сложност на добавянето на елемент към **List<T>** е константна – O(1). Тази усреднена сложност се нарича **амортизирана сложност**. Амортизирана линейна сложност озна­чава, че ако добавим последователно 10 000 елемента, ще извършим сумарно брой стъпки от порядъка на 10 000 и болшинството от тях ще се изпълнят за константно време, а останалите (една много малка част) ще се изпълнят за линейно време.

Търсенето в **List<T>** е бавна операция, защото трябва да се обходят всички елементи. Изтриването по индекс или по стойност се изпълнява за линейно време. Изтриването е бавна операция, защото е свързана с преместване на всички елементи, които са след изтрития с една позиция наляво. Достъпът по индекс в **List<T>** става непосредствено, за кон­стантно време, тъй като елементите се съхраняват вътрешно в масив.

На практика **List<T>** комбинира добрите страни на масивите и на списъците, заради което е предпочитана структура данни в много ситуа­ции. Например, ако трябва да обработим текстов файл и да извлечем от него всички думи, отговарящи на даден регулярен израз, най-удобната структура, в която можем да ги натрупваме, е **List<T>**, тъй като ни трябва списък, чиято дължина не е предварително известна и който да нараства динамично.

Динамичният масив (**List<T>**) е подходящ, когато трябва често да доба­вяме елементи и искаме да запазваме реда им на добавяне и да ги достъпваме често по индекс. Ако често търсим или изтриваме елемент,**List<T>** не е подходяща структура.

|  |  |
| --- | --- |
| **clip_image001[35]** | **Ползвайте List<T>, когато трябва бързо да добавяте елементи и да ги достъпвате по индекс.** |

###

### Стек (Stack)

Стекът е структура от данни, в която са дефинирани 3 операции: добавя­не на елемент на върха на стека, изтриване на елемент от върха на стека и извличане на елемент от върха на стека без премахването му. Всички тези операции се изпълняват бързо, с константна сложност. Операциите търсене и достъп по индекс не се поддържат.

Стекът е структура с поведение LIFO (last in, first out) – последен влязъл, пръв излязъл. Използва се, когато трябва да моделираме такова поведе­ние, например, ако трябва да пазим пътя до текущата позиция при рекур­сивно търсене.

|  |  |
| --- | --- |
| **clip_image001[36]** | **Ползвайте стек, когато е необходимо да реализирате поведението "последен влязъл, пръв излязъл" (LIFO).** |

###

### Опашка (Queue)

Опашката е структура от данни, в която са дефинирани две операции: добавяне на елемент и извличане на елемента, който е наред. Тези две операции се изпълняват бързо, с константна сложност, тъй като опашката обикновено се имплементира чрез свързан списък. Припомняме, че свър­заният списък може да добавя и изтрива бързо елементи в двата си края.

Поведението на структурата опашка е FIFO (first in, first out) – пръв влязъл, пръв излязъл.Операциите търсене и достъп по индекс не се поддържат. Опашката по естествен начин моделира списък от чакащи хора, задачи или други обекти, които трябва да бъдат обработени последователно, в реда на постъпването им.

Като пример за използване на опашка можем да посочим реализацията на алгоритъма "търсене в ширина", при който се започва от даден начален елемент и неговите съседи се добавят в опашка, след което се обработват по реда им на постъпване, а по време на обработката им техните съседи се добавят към опашката. Това се повтаря докато не се достигне до даден елемент, който търсим.

|  |  |
| --- | --- |
| **clip_image001[37]** | **Ползвайте опашка, когато е необходимо да реализирате поведе­нието "пръв влязъл, пръв излязъл" (FIFO).** |

###

## Дървета

В програмирането дърветата са изключително често използвана структура от данни, защото те моделират по естествен начин всякакви йерархии от обекти, които постоянно ни заобикалят в реалния свят. Нека дадем един пример, преди да изложим терминологията, свързана с дърветата.

### Пример – йерархия на участниците в един софтуерен проект

Да вземем за пример един екип, отговорен за изработването на даден софтуерен проект. Участниците в него са взаимно свързани с връзката ръководител-подчинен. Ще разгледаме една конкретна ситуация, в която имаме екип от 9 души:



Каква информация можем да извлечем от така изобразената йерархия? Прекият шеф на програмистите е съответно "Ръководител програмисти". "Ръководител проект" е също е техен началник, но непряк, т.е. те отново са му подчинени. "Ръководител програмисти" е подчинен само на "Ръково­дител проект". От друга страна, ако погледнем "Програмист 1", той няма нито един подчинен. "Ръководител проект" стои най-високо в йерархията и няма шеф.

По аналогичен начин можем да опишем и ситуацията с останалите участ­ници в проекта. Виждаме как една на пръв поглед малка фигура ни носи много информация.

### Терминология свързана с дърветата

За по-доброто разбиране на тази точка силно препоръчваме на читателя да се опита на всяка стъпка да прави аналогия между тяхното абстрактно значение и това, което използваме в ежедневието.



Нека да опростим начина, по който изобразихме нашата йерархия. Можем да приемем, че тя се състои от точки, свързани с отсечки. За удобство, точките ще номерираме с произволни числа, така че после лесно да можем да говорим за някоя конкретна.

Всяка една точка, ще наричаме **връх**, а всяка една отсечка – **ребро**. Върховете "19", "21" и "14" стоят под върха "7" и са директно свързани с него. Тях ще наричаме **преки наследници (деца)** на "7", а "7" – техен **родител (баща)**. Аналогично "1", "12" и "31" са деца на "19" и "19" е техен родител. Съвсем естествено ще казваме, че "21" е **брат**на "19", тъй като са деца на "7" (обратното също е вярно – "19" е брат на "21"). От гледна точка на "1", "12", "31", "23" и "6", "7" е предшестващ ги в йерархията (в случая е родител на техните родители). Затова "7" ще наречем техен **непряк предшественик (дядо, прародител)**, а тях – негови **непреки наследници**.

**Корен** е върхът, който няма предшественици. В нашия случай той е "7".

**Листа**са всички върхове, които нямат наследници. В примера – "1", "12", "31", "21", "23" и "6" са листа.

**Вътрешни върхове**са всички върхове, различни от корена и листата (т.е. всички върхове, които имат както родител, така и поне един наследник). Такива са "19" и "14".

**Път**ще наричаме последователност от свързани чрез ребра върхове, в която няма повтарящи се върхове. Например последователността "1", "19", "7" и "21" е път. "1", "19" и "23" не е път, защото "19" и "23" не са свързани помежду си с ребро.

**Дължина на път**е броят на ребрата, свързващи последователността от върхове в пътя. Практически този брой е равен на броят на върховете в пътя минус единица. Дължината на примера ни за път ("1", "19", "7" и "21") е три.

**Дълбочина на връх**ще наричаме дължината на пътя от корена до дадения връх. На примера ни "7" като корен е с дълбочина нула, "19" е с дълбочина едно, а "23" – с дълбочина две.

И така, ето и дефиницията за това какво е дърво:

**Дърво (tree)** –[рекурсивна](http://www.introprogramming.info/intro-csharp-book/read-online/glava10-rekursia) структура от данни, която се състои от върхове, които са свързани помежду си с ребра. За дърветата са в сила твърденията:

-     Всеки връх може да има 0 или повече преки наследници (деца).

-     Всеки връх има най-много един баща. Съществува точно един спе­циален връх, който няма предшественици – коренът (ако дървото не е празно).

-     Всички върхове са достижими от корена, т.е съществува път от корена до всички тях.

Можем да дефинираме дърво и по по-прост начин: всеки единичен връх наричаме дърво и той може да има нула или повече наследници, които също са дървета.

**Височина на дърво**е максималната от дълбочините на всички върхове. В горния пример височината е 2.

**Степен на връх** ще наричаме броят на преките наследници (деца) на дадения връх. Степента на "19" и "7" е три, докато тази на "14" е две. Листата са от нулева степен.

**Разклоненост на дърво** се нарича максималната от степените на всички върхове в дървото. В нашият пример степента на върховете е най-много 3, следователно разклонеността на дървото ни е 3.

### Реализация на дърво – пример

Нека сега разгледаме как можем да представяме дърветата като структури от данни в програмирането. Ще реализираме дърво, което съдържа числа във върховете си и всеки връх може да има 0 или повече наследници, които също са дървета (следвайки рекурсивната дефиниция). Всеки връх от дървото е рекурсивно-дефиниран чрез себе си. Един връх от дървото (**TreeNode<T>**) съдържа в себе си списък от наследници, които също са върхове от дървото (**TreeNode<T>**). Нека разгледаме сорс кода:

|  |
| --- |
| using System;using System.Collections.Generic; /// <summary>/// Represents a tree node/// </summary>/// <typeparam name="T">the type of the values in/// nodes</typeparam>public class TreeNode<T>{      // Contains the value of the node      private T value;       // Shows whether the current node has parent      private bool hasParent;       // Contains the children of the node      private List<TreeNode<T>> children;       /// <summary>      /// Constructs a tree node      /// </summary>      /// <param name="value">the value of the node</param>      public TreeNode(T value)      {            if (value == null)            {                  throw new ArgumentNullException(                        "Cannot insert null value!");            }            this.value = value;            this.children = new List<TreeNode<T>>();      }       /// <summary>      /// The value of the node      /// </summary>      public T Value      {            get            {                  return this.value;            }            set            {                  this.value = value;            }      }       /// <summary>      /// The number of node's children      /// </summary>      public int ChildrenCount      {            get            {                  return this.children.Count;            }      }       /// <summary>      /// Adds child to the node      /// </summary>      /// <param name="child">the child to be added</param>      public void AddChild(TreeNode<T> child)      {            if (child == null)            {                  throw new ArgumentNullException(                        "Cannot insert null value!");            }             if (child.hasParent)            {                  throw new ArgumentException(                        "The node already has a parent!");            }             child.hasParent = true;            this.children.Add(child);      }       /// <summary>      /// Gets the child of the node at given index      /// </summary>      /// <param name="index">the index of the desired child</param>      /// <returns>the child on the given position</returns>      public TreeNode<T> GetChild(int index)      {            return this.children[index];      }} /// <summary>/// Represents a tree data structure/// </summary>/// <typeparam name="T">the type of the values in the/// tree</typeparam>public class Tree<T>{      // The root of the tree      private TreeNode<T> root;           /// <summary>      /// Constructs the tree      /// </summary>      /// <param name="value">the value of the node</param>      public Tree(T value)      {            if (value == null)            {                  throw new ArgumentNullException(                        "Cannot insert null value!");            }             this.root = new TreeNode<T>(value);      }           /// <summary>      /// Constructs the tree      /// </summary>      /// <param name="value">the value of the root node</param>      /// <param name="children">the children of the root      /// node</param>      public Tree(T value, params Tree<T>[] children)            : this(value)      {                foreach (Tree<T> child in children)            {                  this.root.AddChild(child.root);            }      }       /// <summary>      /// The root node or null if the tree is empty      /// </summary>      public TreeNode<T> Root      {            get            {                  return this.root;            }      }       /// <summary>      /// Traverses and prints tree in Depth      /// First Search (DFS) manner      /// </summary>      /// <param name="root">the root of the tree to be      /// traversed</param>      /// <param name="spaces">the spaces used for      /// representation of the parent-child relation</param>      private void PrintDFS(TreeNode<T> root, string spaces)      {            if (this.root == null)            {                  return;            }                               Console.WriteLine(spaces + root.Value);                             TreeNode<T> child = null;            for (int i = 0; i < root.ChildrenCount; i++)            {                  child = root.GetChild(i);                  PrintDFS(child, spaces + "   ");            }      }       /// <summary>      /// Traverses and prints the tree in      /// Depth First Search (DFS) manner      /// </summary>      public void PrintDFS()      {            this.PrintDFS(this.root, string.Empty);      }} /// <summary>/// Shows a sample usage of the Tree<T> class/// </summary>public static class TreeExample{      static void Main()      {            // Create the tree from the sample            Tree<int> tree =                  new Tree<int>(7,                        new Tree<int>(19,                              new Tree<int>(1),                              new Tree<int>(12),                              new Tree<int>(31)),                        new Tree<int>(21),                        new Tree<int>(14,                              new Tree<int>(23),                              new Tree<int>(6))                  );             // Traverse and print the tree using Depth-First-Search            tree.PrintDFS();             // Console output:            // 7            //       19            //          1            //          12            //          31            //       21            //       14            //          23            //          6      }} |

#### Как работи нашата имплементация на дърво?

Нека кажем няколко думи за предложения код. В примера имаме клас **Tree<Т>**, който е имплементация на самото дърво. Дефиниран е и клас – **TreeNode<T>**, който представлява един връх от дървото.

Функционалността, свързана с връх, като например създаване на връх, добавяне на наследник на връх, взимане на броя на наследниците и т.н. се реализират на ниво **TreeNode<T>**.

Останалата функционалност (например обхождане на дървото) се реализира на ниво **Tree<Т>**. Така функционалността става логически разделена между двата класа, което прави имплементацията по гъвкава.

Причината да разделим на два класа имплементацията е, че някои опера­ции се отнасят за конкретен връх (например добавяне на наслед­ник), докато други се отнасят за цялото дърво (например търсене на връх по неговата стойност). При такова разделяне дървото е клас, който знае кой му е коренът, а всеки връх знае наследниците си. При такава имплемен­тация е възможно да имаме и празно дърво (при **root=null**).

Ето и някои подробности от реализацията на **TreeNode<T>**. Всеки един връх (node) на дървото представлява съвкупност от частно поле **value**, което съдържа стойността му, и списък от наследници **children**. Списъкът на наследниците е от елементи на същия тип. Така всеки връх съдържа списък от референции към неговите преки наследници. Предоставени са също и публични свойства за достъп до стойността на върха. Операциите, които могат да се извършват от външен за класа код върху децата, са:

-     **AddChild(TreeNode<T> child)** - добавя нов наследник.

-     **TreeNode<T> GetChild(int index)** - връща наследник по зададен индекс.

-     **ChildrenCount** - връща броя на наследници на даден връх.

За да спазим изискването всеки връх в дървото да има точно един родител, сме дефинирали частното поле **hasParent**, което показва дали даденият връх има родител. Тази информация се използва вътрешно в нашия клас и ни трябва в метода **AddChild(Tree<T> child)**. В него правим проверка дали кандидат детето няма вече родител. Ако има, се хвърля изключение, показващ, че това е недопустимо.

В класа **Tree<Т>** сме предоставили едно единствено **get** свойство - **TreeNode<T>** **Root,** което връща корена на дървото.

#### Рекурсивно обхождане на дърво в дълбочина

В класа **Tree<Т>** e реализиран и методът **TraverseDFS()**, който извиква частния метод**PrintDFS(TreeNode<T> root, string spaces)**, който обхожда дървото в дълбочина и извежда на стандартния изход елементите му, така че нагледно да се изобрази дървовидната структура чрез отместване надясно (с добавяне на интервали).

Алгоритъмът за **обхождане в дълбочина** (**Depth-First-Search** или **DFS**) започва от даден връх и се стреми да се спусне колкото се може по-надолу в дървовидната йерархия. Когато стигне до връх, от който няма продължение се връща назад към пред­ходния връх. Алгоритъма можем да опишем схематично по следния начин:

1.  Обхождаме текущия връх.

2.  Последователно обхождаме рекурсивно всяко едно от поддър­ветата на текущия връх (обръщаме се рекурсивно към същия метод после­дователно за всеки един от неговите преки наслед­ници).

#### Създаване на дърво

За да създаваме по-лесно дървета сме дефинирали специален конструк­тор, който приема стойност на връх и списък от поддървета за този връх. Така позволяваме подаването на произволен брой аргументи от тип **Tree<T>** (поддървета). При създаването на дървото за нашия пример използваме точно този конструктор и той ни позволява да онагледим структурата му.

### Обхождане на директориите по твърдия диск

Нека сега разгледаме още един пример за дърво – файловата система. Замисляли ли сте се, че директориите върху твърдия ви диск образуват йерархична структура, която е дърво? Можете да се сетите и за много други реални примери, при които се използват дървета.

Нека разгледаме по-подробно файловата система в Windows. Както знаем от нашия всекидневен опит, ние създаваме папки върху твърдия диск, които могат да съдържат от своя страна подпапки или файлове. Подпап­ките отново може да съдържат подпапки и т. н. до някакво разумно огра­ничение (максимална дълбочина).

Дървото на директориите на файловата система е достъпно чрез стандартни функции от класа**System.IO.DirectoryInfo**. То не съществува като структура от данни в явен вид, но съществува начин да извличаме за всяка директория файловете и директориите в нея и следователно можем да го обходим чрез стандартен алгоритъм за обхождане на дървета.

Ето как изглежда типичното дърво на директориите в Windows:



#### Рекурсивно обхождане на директориите в дълбочина

Следващият пример показва как да обходим рекурсивно (в дълбочина, по алгоритъмa Depth-First-Search) дървовидната структура на дадена папка и да изведем на стандартния изход нейното съдържание:

|  |
| --- |
| **DirectoryTraverserDFS.cs** |
| using System;using System.IO; /// <summary>/// Sample class, which traverses recursively given directory/// based on the Depth-First-Search (DFS) algorithm/// </summary>public static class DirectoryTraverserDFS{      /// <summary>      /// Traverses and prints given directory recursively      /// </summary>      /// <param name="dir">the directory to be traversed</param>      /// <param name="spaces">the spaces used for representation      /// of the parent-child relation</param>      private static void TraverseDir(DirectoryInfo dir,            string spaces)      {            // Visit the current directory            Console.WriteLine(spaces + dir.FullName);             DirectoryInfo[] children = dir.GetDirectories();             // For each child go and visit its subtree            foreach (DirectoryInfo child in children)            {                  TraverseDir(child, spaces + "  ");            }      }       /// <summary>      /// Traverses and prints given directory recursively      /// </summary>      /// <param name="directoryPath">the path to the directory      /// which should be traversed</param>      public static void TraverseDir(string directoryPath)      {            TraverseDir(new DirectoryInfo(directoryPath),                  string.Empty);      }       public static void Main()      {            TraverseDir("C:\\");      }} |

Както се вижда от примера, рекурсивното обхождане на съдържанието на директория по нищо не се различава от обхождането на нашето дърво.

Ето как изглежда резултатът от обхождането (със съкращения):

|  |
| --- |
| C:\  C:\Config.Msi  C:\Documents and Settings    C:\Documents and Settings\Administrator      C:\Documents and Settings\Administrator\.ARIS70      C:\Documents and Settings\Administrator\.jindent      C:\Documents and Settings\Administrator\.nbi        C:\Documents and Settings\Administrator\.nbi\downloads        C:\Documents and Settings\Administrator\.nbi\log        C:\Documents and Settings\Administrator\.nbi\cache        C:\Documents and Settings\Administrator\.nbi\tmp        C:\Documents and Settings\Administrator\.nbi\wd      C:\Documents and Settings\Administrator\.netbeans        C:\Documents and Settings\Administrator\.netbeans\6.0... |

#### Обхождане на директориите в ширина

Нека сега разгледаме още един начин да обхождаме дървета. **Обхожда­нето в ширина** (**Breath-First-Search** или **BFS**) е алгоритъм за обхож­дане на дървовидни структури от данни, при който първо се посещава началния връх, след това неговите преки деца, след тях преките деца на децата и т.н. Този процес се нарича **метод на вълната**, защото прилича на вълните, образувани от камък, хвърлен в езеро.

Алгоритъмът за обхождане на дърво в ширина по метода на вълната можем да опишем схематично по следния начин:

1.  Записваме в опашката **Q** началния връх.

2.  Докато **Q** не е празна повтаряме следните две стъпки:

-     Изваждаме от **Q** поредния връх **v** и го отпечатваме.

-     Добавяме всички наследници на **v** в опашката.

Алгоритъмът BFS е изключително прост и има свойството да обхожда първо най-близките до началния връх върхове, след тях по-далечните и т.н. и най-накрая – най-далечните върхове. С времето ще се убедите, че BFS алгоритъмът има широко приложение при решаването на много задачи, като например при търсене на най-кратък път в лабиринт.

Нека сега приложим BFS алгоритъма за отпечатване на всички директории от файловата система:

|  |
| --- |
| **DirectoryTraverserBFS.cs** |
| using System;using System.Collections.Generic;using System.IO; /// <summary>/// Sample class, which traverses given directory/// based on the Breath-First-Search (BFS) algorithm/// </summary>public static class DirectoryTraverserBFS{      /// <summary>      /// Traverses and prints given directory with BFS      /// </summary>      /// <param name="directoryPath">the path to the directory             /// which should be traversed</param>      public static void TraverseDir(string directoryPath)      {            Queue<DirectoryInfo> visitedDirsQueue =                  new Queue<DirectoryInfo>();            visitedDirsQueue.Enqueue(new DirectoryInfo(directoryPath));            while (visitedDirsQueue.Count > 0)            {                  DirectoryInfo currentDir = visitedDirsQueue.Dequeue();                  Console.WriteLine(currentDir.FullName);                   DirectoryInfo[] children = currentDir.GetDirectories();                  foreach (DirectoryInfo child in children)                  {                     visitedDirsQueue.Enqueue(child);                  }            }      }       public static void Main()      {            TraverseDir("C:\\ ");      }} |

Ако стартираме програмата, ще се убедим, че обхождането в ширина първо открива най-близките директории до корена (дълбочина 1), след тях всички директории на дълбочина 2, след това директориите на дълбо­чина 3 и т.н. Ето примерен изход от програмата:

|  |
| --- |
| C:\C:\Config.MsiC:\Documents and SettingsC:\InetpubC:\Program FilesC:\RECYCLERC:\System Volume InformationC:\WINDOWSC:\wmpubC:\Documents and Settings\AdministratorC:\Documents and Settings\All UsersC:\Documents and Settings\Default User... |

###

### Двоични дървета

В предишната точка от темата разгледахме обобщената структура дърво. Сега ще преминем към един неин полезен частен случай, който се оказва изключително важен за практиката – **двоично дърво**. Важно е да отбележим, че термините, които дефинирахме до момента, важат с пълна сила и при този вид дърво. Въпреки това, по-долу ще дадем и някои допълнителни, специфични за дадената структура определения.

**Двоично дърво (binary tree)** – дърво, в което всеки връх е от степен не надвишаваща две т.е. дърво с разклоненост две. Тъй като преките наследници (деца) на всеки връх са най-много два, то е прието да се въвежда наредба между тях, като единият се нарича **ляв наследник**, а другият –**десен наследник**. Те, от своя страна, са корени съответно на **лявото поддърво** и на **дясното поддърво** на техния родител.

#### Двоично дърво – пример

Ето и едно примерно двоично дърво, което ще използваме за изложението по-нататък. В този пример отново въвеждаме номерация на върховете, която е абсолютно произволна и която ще използваме, за да може по-лесно да говорим за всеки връх.

На примера са изобразени съответно корена на дървото "14", пример за ляво поддърво (с корен "19") и дясно поддърво (с корен "15"), както и ляв и десен наследник – съответно "3" и "21".



Следва да отбележим обаче, че двоичните дървета имат едно много сериозно различие в дефиницията си, за разлика от тази на обикновеното дърво – наредеността на наследниците на всеки връх. Следващият пример ясно показва това различие:



На схемата са изобразени две абсолютно различни **двоични дървета** – в единия случай коренът е "19" и има **ляв наследник** "23", а в другия имаме двоично дърво с корен отново "19", но с "23" за**десен наследник**. Ако разгледаме обаче двете структури като **обикновени** **дървета**, те ще са абсолютно еднакви и неразличими. Затова такова **дърво** бихме изобра­зили по следния начин:



|  |  |
| --- | --- |
| clip_image014 | **Запомнете! Въпреки, че разглеждаме двоичните дървета като подмножество на структурата дърво, трябва да се отбележи, че условието за нареденост на наследниците ги прави до голяма степен различни като структури.** |

#### Обхождане на двоично дърво

Обхождането на дърво по принцип е една класическа и често срещана задача. В случая на двоичните дървета има няколко основни начина за обхождане:

-     **ЛКД (Ляво-Корен-Дясно/Inorder)** – обхождането става като първо се обходи лявото поддърво, след това корена и накрая дясното поддърво. В нашият пример последователността, която се получава при обхождането е: "23", "19", "10", "6", "21", "14", "3", "15".

-     **КЛД (Корен-Ляво-Дясно/Preorder)** – в този случай първо се обхожда корена на дървото, после лявото поддърво и накрая дясното. Ето и как изглежда резултатът от този вид обхождане: "14", "19", "23", "6", "10", "21", "15", "3".

-     **ЛДК (Ляво-Дясно-Корен/Postorder)** – тук по аналогичен на горните два примера начин, обхождаме първо лявото поддърво, после дясното и накрая коренът. Резултатът след обхождането е "23", "10", "21", "6", "19", "3", "15", "14".

#### Обхождане на двоично дърво с рекурсия – пример

В следващия пример ще покажем примерна реализация на двоично дърво, което ще обходим по схемата ЛКД:

|  |
| --- |
| using System;using System.Collections.Generic; /// <summary>/// Represents a binary tree node/// </summary>/// <typeparam name="T">the type of the values in nodes</typeparam>public class BinaryTreeNode<T>{      // Contains the value of the node      private T value;       // Shows whether the current node has parent      private bool hasParent;       // Contains the left child of the node      private BinaryTreeNode<T> leftChild;       // Contains the right child of the node      private BinaryTreeNode<T> rightChild;       /// <summary>      /// Constructs a binary tree node      /// </summary>      /// <param name="value">the value of the node</param>      /// <param name="leftChild">the left child of the node</param>      /// <param name="rightChild">the right child of the      /// node</param>      public BinaryTreeNode(T value,                  BinaryTreeNode<T> leftChild,                  BinaryTreeNode<T> rightChild)      {            if (value == null)            {                  throw new ArgumentNullException(                        "Cannot insert null value!");            }             this.value = value;            this.LeftChild = leftChild;            this.RightChild = rightChild;      }       /// <summary>      /// Constructs a binary tree node with no children      /// </summary>      /// <param name="value">the value of the node</param>      public BinaryTreeNode(T value) : this(value, null, null)      {      }       /// <summary>      /// The value of the node      /// </summary>      public T Value      {            get            {                  return this.value;            }            set            {                  this.value = value;            }      }       /// <summary>      /// The left child of the node      /// </summary>      public BinaryTreeNode<T> LeftChild      {            get            {                  return this.leftChild;            }            set            {                  if (value == null)                  {                        return;                  }                   if (value.hasParent)                  {                        throw new ArgumentException(                              "The node already has a parent!");                  }                   value.hasParent = true;                  this.leftChild = value;            }      }       /// <summary>      /// The right child of the node      /// </summary>      public BinaryTreeNode<T> RightChild      {            get            {                  return this.rightChild;            }            set            {                  if (value == null)                  {                        return;                  }                   if (value.hasParent)                  {                        throw new ArgumentException(                              "The node already has a parent!");                  }                   value.hasParent = true;                  this.rightChild = value;            }      }} /// <summary>/// Represents a binary tree structure/// </summary>/// <typeparam name="T">the type of the values in the/// tree</typeparam>public class BinaryTree<T>{      // The root of the tree      private BinaryTreeNode<T> root;       /// <summary>      /// Constructs the tree      /// </summary>      /// <param name="value">the value of the root node</param>      /// <param name="leftChild">the left child of the root node</param>      /// <param name="rightChild">the right child of the      /// root node</param>      public BinaryTree(T value, BinaryTree<T> leftChild,                  BinaryTree<T> rightChild)      {            if (value == null)            {                  throw new ArgumentNullException(                        "Cannot insert null value!");            }             BinaryTreeNode<T> leftChildNode =                  leftChild != null ? leftChild.root : null;             BinaryTreeNode<T> rightChildNode =                  rightChild != null ? rightChild.root : null;             this.root = new BinaryTreeNode<T>(                  value, leftChildNode, rightChildNode);      }       /// <summary>      /// Constructs the tree      /// </summary>      /// <param name="value">the value of the root node</param>      public BinaryTree(T value)            : this(value, null, null)      {      }       /// <summary>      /// The root of the tree.      /// </summary>      public BinaryTreeNode<T> Root      {            get            {                  return this.root;            }      }       /// <summary>      /// Traverses binary tree in pre-order manner      /// </summary>      /// <param name="root">the binary tree to be traversed</param>      private void PrintInorder(BinaryTreeNode<T> root)      {            if (root == null)            {                  return;            }                       // 1. Visit the left child            PrintInorder(root.LeftChild);                       // 2. Visit the root of this subtree            Console.Write(root.Value + " ");                       // 3. Visit the right child            PrintInorder(root.RightChild);      }       /// <summary>      /// Traverses and prints the binary      /// tree in pre-order manner      /// </summary>      public void PrintInorder()      {            PrintInorder(this.root);            Console.WriteLine();      }} /// <summary>/// Shows how the BinaryTree class can be used/// </summary>public class BinaryTreeExample{      public static void Main()      {            // Create the binary tree from the sample            BinaryTree<int> binaryTree =                  new BinaryTree<int>(14,                              new BinaryTree<int>(19,                                          new BinaryTree<int>(23),                                          new BinaryTree<int>(6,                                                      new BinaryTree<int>(10),                                                      new BinaryTree<int>(21))),                              new BinaryTree<int>(15,                                          new BinaryTree<int>(3),                                          null));             // Traverse and print the tree in in-order manner            binaryTree.PrintInorder();             // Console output:            // 23 19 10 6 21 14 3 15      }} |

#### Как работи примерът?

Тази примерна имплементация на двоично дърво не се различава съществено от реализацията, която показахме в случая на обикновено дърво. Отново имаме отделни класове за представяне на двоично дърво и на връх в такова – **BinaryTree<T>** и **BinaryTreeNode<T>**. В класа**BinaryTreeNode<T>** имаме частни полета **value** и **hasParent**. Както и преди, първото съдържа стойността на върха, а второто показва дали върха има родител. При добавяне на ляв или десен наследник (ляво/дясно дете) на даден връх, се прави проверка дали имат вече родител и ако имат, се хвърля изключение, аналогично на реализацията ни на дърво.

За разлика от реализацията на обикновеното дърво, сега вместо списък на децата, всеки връх съдържа по едно частно поле за ляв и десен наследник. За всеки от тях сме дефинирали публични свойства, за да могат да се достъпват от външен за класа код.

В **BinaryTree<T>** е реализирано едно единствено **get**свойство, което връща корена на дървото. Методът **PrintInоrder()**извиква вътрешно метода **PrintInоrder(BinaryTreeNode<T> root)**. Вторият метод, от своя страна, обхожда подаденото му дърво по схемата ляво-корен-дясно (ЛКД). Това става по следния тристъпков алгоритъм:

1.  Рекурсивно извикване на метода за обхождане за лявото поддърво на дадения връх.

2.  Обхождане на самия връх.

3.  Рекурсивно извикване на метода за обхождане на дясното поддърво.

Силно препоръчваме на читателя да се опита (като едно добро упраж­нение) да модифицира предложения алгоритъм и код самостоятелно, така че да реализира другите два основни типа обхождане.

### Наредени двоични дървета за претърсване

До момента видяхме как можем да построим обикновено дърво и двоично дърво. Тези структури сами по себе си са доста обобщени и трудно, в такъв суров вид, могат да ни свършат някаква по-сериозна работа. На практика в информатиката се прилагат някои техни разновидности, в които са дефинирани съвкупност от строги правила (алгоритми) за раз­лични операции с тях и с техните елементи. Всяка една от тези разновид­ности носи със себе си специфични свойства, които са полезни в различни ситуации.

Като примери за такива полезни свойства могат да се дадат бързо търсене на елемент по зададена стойност ([червено-черно дърво](http://www.introprogramming.info/intro-csharp-book/read-online/glava17-durveta-i-grafi/#_Балансирани_дървета)); нареденост (сортираност) на елементите в дървото; възможност да се организира голямо количество информация на някакъв файлов носител, така че търсенето на елемент в него да става бързо с възможно най-малко стъпки [(B-дърво)](http://en.wikipedia.org/wiki/B-tree), както и много други.

В тази секция ще разгледаме един по-специфичен клас двоични дървета – **наредените**. Те използват едно често срещано при двоичните дървета свойство на върховете, а именно съществуването на **уникален иденти­фикационен ключ**във всеки един от тях. Този ключ не се среща никъде другаде в рамките на даденото дърво. Друго основно свойство на тези ключове е, че са [сравними](http://www.introprogramming.info/intro-csharp-book/read-online/glava17-durveta-i-grafi/#_Сравнимост_между_обекти). Наредените двоични дървета позволяват бързо (в общия случай с приблизително log(n)на брой стъпки) търсене, добавяне и изтриване на елемент, тъй като поддържат елемен­тите си индиректно в сортиран вид.

#### Сравнимост между обекти

Преди да продължим, ще въведем следната дефиниция, от която ще имаме нужда в по-нататъшното изложение.

**Сравнимост**– два обекта A и B наричаме сравними, ако е изпълнена **точно една**от следните три зависимости между тях:

-     "A е по-малко от B"

-     "A е по-голямо от B"

-     "A е равно на B"

Аналогично два ключа A и B ще наричаме сравними, ако е изпълнена точно една от следните три възможности: A < B, A > B или A = B.

Върховете на едно дърво могат да съдържат най-различни полета. В по-нататъшното разсъждение ние ще се интересуваме само от техните уникални ключове, които ще искаме да са сравними. Да покажем един пример. Нека са дадени два конкретни върха A и B:



В примера ключът на A и B са съответно целите числа 19 и 7. Както знаем от математиката, целите числа (за разлика от комплексните например) са **сравними**, което според гореизложените разсъждения ни дава правото да ги използваме като ключове. Затова за върховете A и B можем да кажем, че "A е по-голямо от B" тъй като "19 е по-голямо от 7".

|  |  |
| --- | --- |
| clip_image014[1] | **Забележете! Този път числата изобразени във върховете са техни уникални идентификационни ключове, а не както досега произволни числа.** |

Стигаме и до дефиницията за **наредено двоично дърво за търсене:**

**Наредено двоично дърво (дърво за търсене, binary search tree)** e двоично дърво,в което всеки връх има уникален ключ, всеки два от ключовете са сравними и което е организирано така, че за всеки връх да е изпълнено:

-     Всички ключове в **лявото** му поддървоса**по-малки** от неговия ключ.

-     Всичкиключове в **дясното**му поддърво са**по-големи** от неговия ключ.

#### Свойства на наредените двоични дървета за претърсване

На фигурата е изобразен пример за наредено двоично дърво за претър­сване. Ще изпол­зваме този пример, за да дадем някои важни свойства на наредеността на двоично дърво:



По дефиниция имаме, че лявото поддърво на всеки един от върховете се състои само от елементи, които са по-малки от него, докато в дясното поддърво има само по-големи елементи. Това означава, че ако искаме да намерим даден елемент тръгвайки от корена, то или сме го намерили или трябва да го търсим съответно в лявото или дясното му поддърво, с което ще спестим излишни сравнения. Например, ако търсим в нашето дърво 23, то няма смисъл да го търсим в лявото поддърво на 19, защото 23 със сигурност не е там (23 е по-голямо от 19 следователно евентуално е в дясното поддърво). Това ни спестява 5 излишни сравнения с всеки един от елементите от лявото поддърво, които, ако използваме свързан списък, например, ще трябва да извършим.

От наредеността на елементите следва, че **най-малкият**елемент в дър­вото е най-левият наследник на корена, ако има такъв, или самият корен, ако той няма ляв наследник. По абсолютно същия начин **най-големият**елемент в дървото е най-десният наследник на корена, а ако няма такъв – самият корен. В нашия пример това са минималният елемент 7 и макси­малният – 35. Полезно и директно следващо свойство от това е, че всеки един елемент от лявото поддърво на даден връх е по-малък от всеки друг, който е в дясното поддърво на същия връх.

#### Наредени двоични дървета за търсене – пример

Следващият пример показва реализация на двоично дърво за търсене. Целта ни ще бъде да предложим методи за добавяне, търсене и изтриване на елемент в дървото. За всяка една от тези операции ще дадем подробно обяснение как точно се извършва.

#### Наредени двоични дървета: реализация на върховете

Както и преди, сега ще дефинираме вътрешен клас, който да опише структурата на един връх. По този начин ясно ще разграничим и капсулираме структурата на един връх като същност, която дървото ни ще съдържа в себе си. Този отделен клас сме дефинирали като частен и е видим само в класа на нареденото ни дърво. Ето и неговата дефиниция:

|  |
| --- |
| ...  /// <summary>/// Represents a binary tree node/// </summary>/// <typeparam name="T"></typeparam>private class BinaryTreeNode<T> : IComparable<BinaryTreeNode<T>>      where T : IComparable<T>{      // Contains the value of the node      internal T value;       // Contains the parent of the node      internal BinaryTreeNode<T> parent;       // Contains the left child of the node      internal BinaryTreeNode<T> leftChild;       // Contains the right child of the node      internal BinaryTreeNode<T> rightChild;            /// <summary>      /// Constructs the tree node      /// </summary>      /// <param name="value">The value of the tree node</param>      public BinaryTreeNode(T value)      {            this.value = value;            this.parent = null;            this.leftChild = null;            this.rightChild = null;      }       public override string ToString()      {            return this.value.ToString();      }       public override int GetHashCode()      {            return this.value.GetHashCode();      }       public override bool Equals(object obj)      {            BinaryTreeNode<T> other = (BinaryTreeNode<T>)obj;            return this.CompareTo(other) == 0;      }       public int CompareTo(BinaryTreeNode<T> other)      {            return this.value.CompareTo(other.value);      }}... |

Да разгледаме предложения код. Още в името на структурата, която разглеждаме – "наредено дърво за търсене", ние говорим за наредба, а такава можем да постигнем **само**ако имаме **сравнимост**между елемен­тите в дървото.

#### ****Сравнимост между обекти в**** ****C#****

Какво означава понятието "сравнимост между обекти" за нас като програ­мисти? Това означава, че трябва да задължим по някакъв начин всички, които използват нашата структура от данни, да я създават подавайки и **тип, който е сравним**. На C# изречението "тип, който е сравним" би "звучало" така:

|  |
| --- |
| T **:** IComparable<T> |

Интерфейсът **IComparable<T>**, намиращ се в пространството от имена **System**, се състои само от един метод int**CompareTo(T obj)**, който връща отрицателно цяло число, нула или положително цяло число съответно, ако текущият обект е по-малък, равен или по-голям от този, който е подаден на метода. Дефиницията му изглежда по приблизително следния начин:

|  |
| --- |
| public interface IComparable<T>{      // Summary:      //     Compares the current object with another object of the   //     same type.      int CompareTo(T other);} |

Имплементирането на този интерфейс от даден клас ни гарантира, че неговите инстанции са сравними.

От друга страна, на нас ни е необходимо и самите върхове, описани чрез класа **BinaryTreeNode,**също да бъдат сравними помежду си. Затова той също имплементира **IComparable<T>**. Както се вижда от кода, имплемента­цията на **IComparable<T>** на класа **BinaryTreeNode** вътрешно извиква тази на типа **T**.

В кода също сме припокрили и методите **Equals(Object obj)** и **GetHashCode()**. Добра (задължителна) практика е тези два метода да са съгласувани в поведението си т.е. когато два обекта са еднакви, хеш-кодът им да е еднакъв. Както ще видим в главата за [хеш-таблици](http://www.introprogramming.info/intro-csharp-book/read-online/glava18-rechnici-hesh-tablici-i-mnojestva/#_Хеш-таблици), обратното въобще не е задължително. Аналогично -  очакваното поведение на **Equals(Object obj)** е да връща истина, точно когато и **CompareTo(T obj)** връща 0.

|  |  |
| --- | --- |
| clip_image014[2] | **Задължително синхронизирайте работата на методите Equals(Object obj),CompareTo(T obj) и GetHashCode(). Това е тяхното очаквано поведение и ще ви спести много трудно откриваеми проблеми!** |

До тук разгледахме методите, предложени от нашият клас. Сега да видим, какви полета ни предоставя. Те са съответно за **value** (ключът) от тип **T** родител – **parent**, ляв и десен наследник – съответно **leftChild** и **rightChild**. Последните три са от типа на дефиниращия ги клас, а именно**BinaryTreeNode**.

#### Наредени двоични дървета - реализация на основния клас

Преминаваме към реализацията на класа, описващ самото наредено двоично дърво. Дървото само по себе си като структура се състои от един корен от тип **BinaryTreeNode**, който вътрешно съдържа наследниците си – съответно ляв и десен, те вътрешно също съдържат техните наследници и така рекур­сивно надолу докато се стигне до листата. Друго важно за отбелязване нещо е дефиницията**BinarySearchTree<T> where T:IComparable<T>**. Това ограничение на типа **T** се налага заради изискването на вътрешния ни клас, който работи само с типове, имплементиращи **IComparable<T>**.

|  |
| --- |
| public class BinarySearchTree<T>      where T : IComparable<T>{      /// <summary>      /// Represents a binary tree node      /// </summary>      /// <typeparam name="T">The type of the nodes</typeparam>      private class BinaryTreeNode<T> :            IComparable<BinaryTreeNode<T>>            where T : IComparable<T>      {            //...            //... The implementation from above goes here!!! ...            //...      }      /// <summary>      /// The root of the tree      /// </summary>      private BinaryTreeNode<T> root;       /// <summary>      /// Constructs the tree      /// </summary>      public BinarySearchTree()      {            this.root = null;      }       //...      //... The operation implementation goes here!!! ...      //...} |

Както споменахме по-горе, ще разгледаме следните операции:

-     добавяне на елемент;

-     търсене на елемент;

-     изтриване на елемент.

#### Добавяне на елемент в подредено двоично дърво

След добавяне на нов елемент, дървото трябва да запази своята нареде­ност. Алгоритъмът е следният: ако дървото е празно, то добавяме новия елемент като корен. В противен случай:

-     Ако елементът е по-малък от корена, то се обръщаме рекурсивно към същия метод, за да включим елемента в лявото поддърво.

-     Ако елементът е по-голям от корена, то се обръщаме рекурсивно към същия метод, за да включим елемента в дясното поддърво.

-     Ако елементът е равен на корена, то не правим нищо и излизаме от рекурсията.

Ясно се вижда как алгоритъмът за включване на връх изрично се съобра­зява с правилото елементите в лявото поддърво да са по-малки от корена на дървото и елементите от дясното поддърво да са по-големи от корена на дървото. Ето и примерна имплементация на този метод. Забележете, че при включването се поддържа референция към родителя, защото родите­лят също трябва да бъде променен.

|  |
| --- |
| /// <summary>/// Inserts new value in the binary search tree/// </summary>/// <param name="value">the value to be inserted</param>public void Insert(T value){      if (value == null)      {            throw new ArgumentNullException(                  "Cannot insert null value!");      }       this.root = Insert(value, null, root);} /// <summary>/// Inserts node in the binary search tree by given value/// </summary>/// <param name="value">the new value</param>/// <param name="parentNode">the parent of the new node</param>/// <param name="node">current node</param>/// <returns>the inserted node</returns>private BinaryTreeNode<T> Insert(T value,            BinaryTreeNode<T> parentNode,            BinaryTreeNode<T> node){      if (node == null)      {            node = new BinaryTreeNode<T>(value);            node.parent = parentNode;      }      else      {            int compareTo = value.CompareTo(node.value);            if (compareTo < 0)            {                  node.leftChild =                        Insert(value, node, node.leftChild);            }            else if (compareTo > 0)            {                  node.rightChild =                        Insert(value, node, node.rightChild);            }      }       return node;} |

#### Търсене на елемент в подредено двоично дърво

Търсенето е операция, която е още по-интуитивна. В примерния код сме показали как може търсенето да се извърши без рекурсия, a чрез итерация. Алгоритъмът започва с елемент **node**, сочещ корена. След това се прави следното:

-     Ако елементът е равен на **node**, то сме намерили търсения елемент и го връщаме.

-     Ако елементът е по-малък от **node**, то присвояваме на **node** левия му наследник т.е. продължаваме търсенето в лявото поддърво.

-     Ако елементът е по-голям от **node**, то присвояваме на **node** десния му наследник т.е. продължаваме търсенето в дясното поддърво.

При приключване, алгоритъмът връща или намерения връх или **null,** ако такъв връх не съществува в дървото.

Следва примерен код:

|  |
| --- |
| /// <summary>/// Finds a given value in the tree and returns the node/// which contains it if such exsists/// </summary>/// <param name="value">the value to be found</param>/// <returns>the found node or null if not found</returns>private BinaryTreeNode<T> Find(T value){      BinaryTreeNode<T> node = this.root;       while (node != null)      {            int compareTo = value.CompareTo(node.value);            if (compareTo < 0)            {                  node = node.leftChild;            }            else if (compareTo > 0)            {                  node = node.rightChild;            }            else            {                  break;            }      }       return node;} |

#### Изтриване на елемент от подредено двоично дърво

Изтриването е най-сложната операция от трите основни. След нея дървото трябва да запази своята нареденост. Първата стъпка преди да изтрием елемент от дървото е да го намерим. Вече знаем как става това. След това се прави следното:

-     Ако върхът е листо – насочваме референцията на родителя му към **null**. Ако елементът няма родител следва, че той е корен и просто го изтриваме.

-     Ако върхът има само едно поддърво – ляво или дясно, то той се замества с корена на това поддърво.

-     Ако върхът има две поддървета. Тогава намираме най-малкият връх в дясното му поддърво и го разменяме с него. След тази размяна върхът ще има вече най-много едно поддърво и го изтриваме по някое от горните две правила. Тук трябва да отбележим, че може да се направи аналогична размяна, само че взимаме лявото поддърво и най-големият елемент от него.

Оставяме на читателя, като леко упражнение, да провери коректността на всяка една от тези три стъпки.

Нека разгледаме едно примерно изтриване. Ще използваме отново нашето наредено дърво, което показахме в началото на тази точка. Да изтрием например елемента с ключ 11.



Върхът 11 има две поддървета и, съгласно нашия алгоритъм, трябва да бъде разменен с най-малкия елемент от дясното поддърво, т.е. с 13. След като извършим размяната вече можем спокойно да изтрием 11, който е листо. Ето крайния резултат:



Предлагаме следния примерен код, който реализира описания алгоритъм:

|  |
| --- |
| /// <summary>/// Removes an element from the tree if exists/// </summary>/// <param name="value">the value to be deleted</param>public void Remove(T value){      BinaryTreeNode<T> nodeToDelete = Find(value);      if (nodeToDelete == null)      {            return;      }       Remove(nodeToDelete);} private void Remove(BinaryTreeNode<T> node){      // Case 3: If the node has two children.      // Note that if we get here at the end      // the node will be with at most one child      if (node.leftChild != null && node.rightChild != null)      {            BinaryTreeNode<T> replacement = node.rightChild;            while (replacement.leftChild != null)            {                  replacement = replacement.leftChild;            }            node.value = replacement.value;            node = replacement;      }       // Case 1 and 2: If the node has at most one child      BinaryTreeNode<T> theChild = node.leftChild != null ?                  node.leftChild : node.rightChild;       // If the element to be deleted has one child      if (theChild != null)      {            theChild.parent = node.parent;             // Handle the case when the element is the root            if (node.parent == null)            {                  root = theChild;            }            else            {                  // Replace the element with its child subtree                  if (node.parent.leftChild == node)                  {                        node.parent.leftChild = theChild;                  }                  else                  {                        node.parent.rightChild = theChild;                  }            }      }      else      {            // Handle the case when the element is the root            if (node.parent == null)            {                  root = null;            }            else            {                  // Remove the element - it is a leaf                  if (node.parent.leftChild == node)                  {                        node.parent.leftChild = null;                  }                  else                  {                        node.parent.rightChild = null;                  }            }      }} |

### Балансирани дървета

Както видяхме по-горе, наредените двоични дървета представляват една много удобна структура за търсене. Така дефинирани операциите за създаване и изтриване на дървото имат един скрит недостатък. Какво би станало ако в дървото включим последователно елементите 1, 2, 3, 4, 5, 6? Ще се получи следното дърво:



В този случай двоичното дърво се е изродило в свързан списък. От там и търсенето в това дърво ще е доста по-бавно (с **N** на брой стъпки, а не с **log(N)**), тъй като, за да проверим дали даден елемент е вътре, в най-лошия случай ще трябва да преминем през всички елементи.

Ще споменем накратко за съществуването на структури от данни, които в общия случай запазват логаритмичното поведение на операциите добавяне, търсене и изтриване на елемент. Преди да кажем как се постига това, ще въведем следните две дефиниции:

**Балансирано двоично дърво**– двоично дърво, в което никое листо не е на "много по-голяма" дълбочина от всяко друго листо. Дефиницията на "много по-голяма" зависи от конкретната балансираща схема.

**Идеално балансирано двоично дърво**– двоично дърво, в което разликата в **броя на върховете на лявото и дясното поддърво** на всеки от върховете е най-много единица.

Без да навлизаме в детайли ще споменем, че ако дадено двоично дърво е балансирано, дори и да не е идеално балансирано, то операциите за добавяне, търсене и изтриване на елемент в него са с логаритмична сложност дори и в най-лошия случай. За да се избегне дисбаланса на дървото за претърсване, се прилагат операции, които пренареждат част от елементите на дървото при добавяне или при премахване на елемент от него. Тези операции най-често се наричат **ротации**. Конкретният вид на ротациите, се уточнява допълнително и зависи от реализацията на конкрет­ната структура от данни. Като примери за такива структури, можем да дадем **червено-черно** дърво, **AVL**-дърво, **AA**-дърво, **Splay**-дърво и др.

За по-детайлно разглеждане на тези и други структури препоръчваме на читателя да потърси в строго специализираната литература за алгоритми и структури от данни.

#### ****Скритият клас**** ****TreeSet<T> в**** ****.NET Framework****

След като вече се запознахме с наредените двоични дървета и с това какво е предимството те да са балансирани, идва момента да покажем и какво C# има за нас по този въпрос. Може би всеки от вас тайно се е надявал, че никога няма да му се налага да имплементира балансирано наредено двоично дърво за търсене, защото изглежда доста сложно. Това най-вероятно наистина е така.

До момента разгледахме какво представляват балан­сира­ните дървета, за да добиете представа за тях. Когато ви се наложи да ги ползвате, винаги можете да разчитате да ги вземете от някъде наготово. В стандартните библиотеки на .NET Framework има готови имплементации на баланси­рани дървета, а освен това по Интернет можете да намерите и много външни библиотеки.

В пространството от имена **System.Collections.Generic** се поддържа класът **TreeSet<T>**, който вът­решно представлява имплементация на червено-черно дърво. Това, както вече знаем, означава, че добавянето, търсенето и изтриването на еле­менти в дървото ще се извърши с логаритмична сложност (т.е. ако имаме 1 000 000 елемента операцията ще бъде извършена за около 20 стъпки). Лошата новина е, че този клас е **internal** и е видим само в тази библиотека. За щастие обаче, този клас се ползва вътрешно от друг, който е публично достъпен – **SortedDictionary<T>**. Повече информация за класа **SortedDictionary<T>** можете да намерите в секцията "[Множества](http://www.introprogramming.info/intro-csharp-book/read-online/glava18-rechnici-hesh-tablici-i-mnojestva/#_Структура_от_данни)" на главата"[Речници, хеш-таблици и множества](http://www.introprogramming.info/intro-csharp-book/read-online/glava18-rechnici-hesh-tablici-i-mnojestva)".