Технически университет- София

Виктор Руменов Костадинов

Група 43, №121224145

Курсова работа по Синтез и анализ на алгоритми(САА)

Дървовидни структури (дървета) – задаване и обхождане

Знаем, че редиците и списъците могат лесно да се дефинират по следния начин: редица(списък) от базов тип T е едно от двете:

(1) празната редица(списък);
(2) съчленение(верига) на T с редица от базов тип T.

Тук рекурсията се използва като помощно средство за дефиниране на принципите на структуриране, а именно- подреждане в редица или итерация. Редиците и итерациите са толкова общи, че те обикновено се разглеждат като фундаментални модели на структура и поведение. Но трябва да се има в предвид, че те могат да се дефинират посредством рекурсия, докато обратното не е вярно, защото рекурсията може да се използва ефективно и елегантно за дефиниране на много по-сложни структури. Дърветата са добре познат пример. Да дефинираме понятието *"дървовидна структура"* така: дървовидна структура с базов тип T е едно от двете:

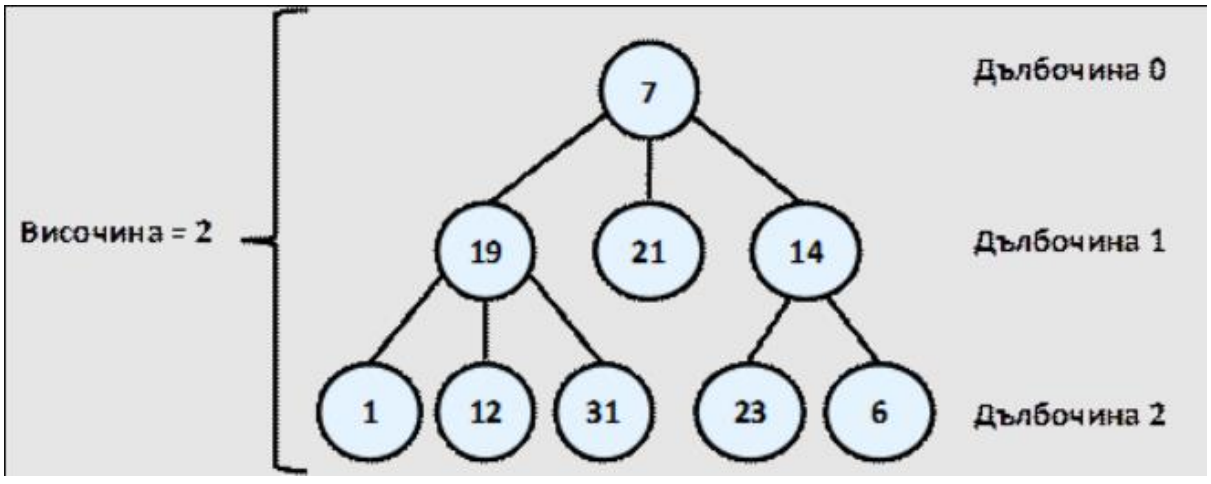
(1) празната структура;
(2) възел от тип T, свързан с краен брой несвързани помежду си дървовидни структури от базов тип T, наречени поддървета.

От сходството между рекурсивните дефиниции на редици и дървовидни структури, редицата(списъкът) понякога може да се нарича също *изродено дърво*.

Има няколко начина за представяне на дървовидни структури. Някои от тях са вложени множества, вложени скоби, стъпаловидно отместване, граф. Макар и странен, вече се е установил навик за изобразяване на дърветата от горе до долу или, с други думи, така, че да се виждат корените им. Тази формулировка обаче е заблуждаваща, тъй като корен обикновено се нарича най-горният възел. Макар да знаем, че дърветата в природата са много по-сложни създания от нашите абстракции,

отсега нататък ще наричаме дървовидните структури просто дървета.

Дърветата представляват основополагащ елемент в алгоритмите и структурите от данни. Те осигуряват ефикасно съхранение, извличане и манипулиране на йерархични данни. Намират приложение в компютърни файлови системи, компилатори, бази данни, изкуствен интелект, мрежови архитектури, разработка на игри и други области. В програмирането дърветата са изключително често срещана структура от данни, защото те естествено стимулират йерархичните структури на различни обекти, които постоянно съществуват около нас в реалния свят. Формалната дефиниция на дървовидна структура/дърво е свързан, ацикличен граф, състоящ се от възли/върхове, свързани посредством ребра, притежаващ точно един специален връх, който няма предшественици и се дефинира като корен. Познато е, че между всеки два върха съществува точно един прост път, докато всеки връх е способен да има нула или повече преки наследници. Броят на възлите е пряко свързан с броят на ребрата (при n възли, то имаме n – 1 ребра).



За по-прегледно и ясно обяснение ще разгледаме примерната дървовидна структура, точките на която са номерирани с произволни числа. Както казахме малко по-нагоре, всяка една

такава ще наричаме *връх*, а отсечката между тях- *ребро*. Мисля че поне веднъж на всеки един от нас се е налагало да изобразим и представим своето родословно дърво. В повече случаи на него са изобразени нашите кръвни роднини като родители, прародители, братя, сестри и т.н. Тук положението е подобно, а може да се каже и същото. Върховете “7”, “19”, “14” са *родителски върхове(бащи)*, съответно на своите преки *наследници(деца)*- “19”, “21”, “14” са заели позиция под “7” и са директно свързани с него, докато същото може да се каже за “1”, “12”, “31” и техния родител “19”, както и “14”- баща на “23” и “6”. Следвайки тази логика няма да сгрешим ако кажем че един наследник на връх е *“брат”* с останалите негови наследници. След толкова роднински връзки е разбираемо да се оплетем в техните взаимоотношения, но важното е да разберем логиката и структурата на дърветата. *Корен* е върхът, който няма предшественици. В нашия случай той е “7”. Всички върхове, които нямат наследници(“1”, “12”, “31”, “21”, “23”, “6”) могат да се нарекат *листа*. Останалите върхове, различни от корена и листата се наричат *вътрешни върхове(*“19”, “14”). *Път* ще наричаме последователност от свързани чрез ребра върхове, в която няма повтарящи се върхове. Пример за това е последователността “23”, “14”, “7”, “21”. *Дължина на път* е броят на ребрата, свързващи последователността от върхове в пътя. Дължината на примера ни за път е три(може да се каже че дължината е броят на върховете в пътя минус едно). *Степен на връх* можем да дефинираме като броят на преките наследници на дадения връх, а *височината на дървото* е максималната от дълбочините на всички върхове. Както виждаме на изображението, за нашия пример височината е две.

Съществуват различни видове дървовидни структури от данни, които се класифицират според броя на наследниците, степента на балансираност и предназначението им. Най-често използвани са двоичните дървета, балансираните дървета и

специализираните структури като heap и trie, които намират широко приложение в съвременните алгоритми и софтуерни системи. Едно дърво може да бъде представено по различни начини:

- *Йерархично задаване*- при него дървото се описва чрез корен, върхове, деца и поддървета. Използва се в теорията и при абстрактно описание на алгоритми.
- *Задаване чрез указатели*- всеки връх съдържа данни, указател към родител и указател към деца. Това е най-често срещаното представяне в програмирането(двоично и n-арно дърво).
- *Задаване чрез масив*- върховете се съхраняват в масив. Използва се главно при пълни двоични дървета и heap структури.
- *Задаване чрез родителски масив*- за всеки връх се пази индекс на неговия родител. Този вид е подходящ за статични дървета.

По начина на обхождане на едно дърво също може да разграничим няколко вида:

- *Обхождане в дълбочина(DFS- Depth First Search)*- Preorder, Inorder, Postorder. За n-арни дървета имаме при Preorder- от корен до всички деца(отляво надясно), а при Postorder- от всички деца до корен.
- *Обхождане в широчина(BFS- Breadth First Search)*- обхождане по нива. Реализира се с опашка и е подходящо за търсене на най-къс път по нива.
- *Рекурсивно и итеративно обхождане*

След като вече знаем какво представлява дървовидната структура и сме запознати с основните понятия, описващи я, нека

сега да разгледаме как точно се представя тя в програмирането. Със следния код(C#) ще реализираме дърво:

```
using System;
using System.Collections.Generic;

//Дървовидна структура
10 references
public class TreeNode<T>
{
    private T value;
    private bool hasParent;
    private List<TreeNode<T>> children;

    1 reference
    public TreeNode(T value)
    {
        if (value == null)
        {
            throw new ArgumentNullException("Cannot insert null value!");
        }
        this.value = value;
        this.children = new List<TreeNode<T>>();
    }

    1 reference
    public T Value
    {
        get { return this.value; }
        set { this.value = value; }
    }

    1 reference
    public int ChildrenCount
    {
        get { return this.children.Count; }
    }
```

```
    1 reference
    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);
    }

    1 reference
    public TreeNode<T> GetChild(int index)
    {
        return this.children[index];
    }
}
```

```
15 references
public class Tree<T>
{
    private TreeNode<T> root;

    7 references
    public Tree(T value)
    {
        if (value == null)
        {
            throw new ArgumentNullException("Cannot insert null value!");
        }
        this.root = new TreeNode<T>(value);
    }

    3 references
    public Tree(T value, params Tree<T>[] children) : this(value)
    {
        foreach (Tree<T> child in children)
        {
            this.root.AddChild(child.root);
        }
    }

    0 references
    public TreeNode<T> Root
    {
        get { return this.root; }
    }
```

```
    2 references
    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 + "   ");
        }
    }

    1 reference
    public void PrintDFS()
    {
        this.PrintDFS(this.root, string.Empty);
    }
}
```

```
0 references
public static class TreeExample
{
    0 references
    static void Main()
    {
        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))
            );

        tree.PrintDFS();
    }
}
```

С този код представяме реализацията на дървовидна структура от данни в програмирането. Той имплементира общо(n-арно) кореново дърво с preorder обхождане. Всеки връх от дървото е дефиниран рекурсивно чрез себе си. Връх от дървото дефинираме посредством TreeNode<T>, който съдържа в себе си списък от наследници- също върхове от дървото. Използваме два класа: Tree<T>, който представя самото дърво и <TreeNode>, който описва отделен възел. Всички операции, свързани с отделен връх- като създаването му, добавяне на негов наследник, получаване на броя на наследниците и др.- са реализирани в класа TreeNode<T>. Операции, които засягат дървото като цялостна структура(например обхождане или търсене по стойност), се намират в класа Tree<T>. Това логическо разделение прави кода по-структуриран и по-гъвкав, тъй като част от функционалността се отнася до конкретни върхове, а друга- до дървото като цяло. За улеснение при конструиране на дървовидни структури има създаден специален конструктор, който приема стойност за връх и списък от неговите поддървета. Така може да се подават произволен брой аргументи от тип Tree<T>. В примера именно този конструктор се използва, което прави структурата на дървото лесна за онагледяване. При подобна архитектура дървото знае кой е неговият корен, а всеки връх познава своите деца. Също така е възможно да съществува и празно дърво(когато root = null). Всеки връх съдържа две основни части, а именно частно поле value, което пази неговата стойност, и списък children, включващ

наследниците му. Този списък съдържа елементи от същия тип- така всеки връх държи референции към своите преки деца. В имплементацията се съдържат и публични свойства за достъп до стойността на върха. Външен за класа код може да извършва следните операции върху децата:

- AddChild(TreeNode<T> child)- добавя нов наследник.
- TreeNode<T> GetChild(int index)- връща дете по зададен индекс.
- ChildrenCount- връща броя на наследници на даден връх.

За да гарантираме, че всеки връх има точно един родител, използваме частно поле hasParent, което показва дали върхът вече е присвоен като дете. В метода AddChild(TreeNode<T> child) проверяваме дали този връх няма друг родител. Ако има, се генерира изключение, тъй като това е недопустимо. В класа Tree<T> е налично едно публично свойство- Root, което връща корена на дървото.

Реализацията на рекурсивното обхождане на дървото в дълбочина е посредством методът TraverseDFS() в класа Tree<T>, който използва частния метод PrintDFS(TreeNode<T> root, string spaces). Именно той извършва обхождането и извежда елементите на дървото на стандартния изход. Чрез добавяне на интервали визуално се представя йерархията на дървовидната структура, като всеки по-долен връх се измества надясно. Алгоритъмът за обхождане в дълбочина(DFS/Depth-First Search) започва от даден връх и се стреми да достигне максимално колкото се може по-надолу в дървовидната йерархия. Когато достигне връх без наследници, алгоритъмът се връща назад към предходните върхове. Процесът може да бъде описан така:

1. Обработваме текущия връх.

2. Рекурсивно обхождаме всяко от поддърветата му и съответно извикваме същия метод за всеки негов наследник.

Накрая създаваме клас TreeExample, където тестваме нашата имплементация на дървовидна структура. Като резултат получаваме дървото, което показахме в горния пример, принтирано в конзолата.

Дърветата заемат важно място в информатиката и програмирането, както и в много други сфери на работа, тъй като позволяват ефективното представяне и обработка на йерархично организирана информация. В резултат на показаното може да се направи извод, че правилният избор на структура от тип дърво и подходящ алгоритъм за обхождане е от съществено значение за ефективността на решенията. Усвояването на концепции като дървовидната структура създава стабилна основа за по-нататъшно изучаване на още по-сложни структури от данни и алгоритми и намира широко приложение в реални софтуерни системи.

GitHub:
https://github.com/VikiBG9/SAA/tree/e600bb14bbde0a975fb743dcfb919a9569cd2eee/SAA/TreeNode/TreeNodeProject

Използвана литература:

- Синтез и анализ на алгоритми- Стойчо Д. Стойчев;
- introProgramming.info - глава 17. Дървета и графи;
- pg-vpeev.com - 19.дървовидни структури.