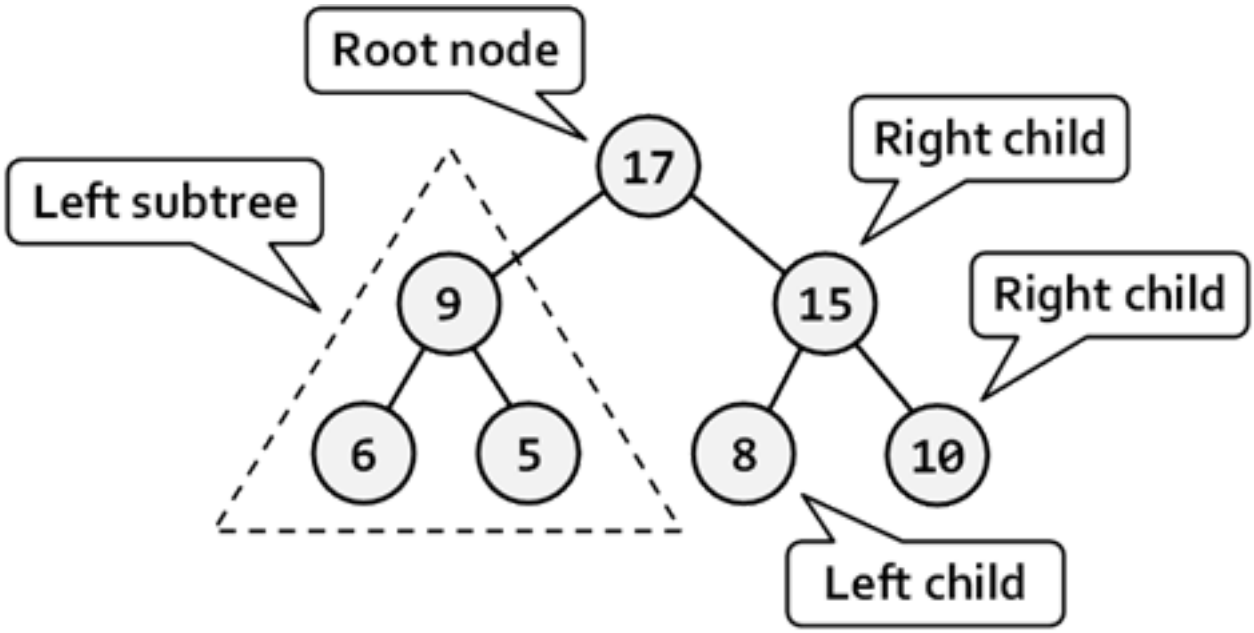
عنوان: **درخت‌ها و گراف‌ها قسمت دوم**
نویسنده: علی یگانه مقدم
تاریخ: ۱۳۹۳/۱۲/۰۳ ۱:۲۵
آدرس: www.dotnettips.info
گروه‌ها: C#, Data Structures, Tree

در قسمت قبلی ما به بررسی درخت و اصطلاحات فنی آن پرداختیم و اینکه چگونه یک درخت را پیمایش کنیم. در این قسمت مطلب قبل را با درخت‌های دودویی ادامه می‌دهیم.

درخت‌های دودویی Binary Trees

همه‌ی موضوعات و اصطلاحاتی را که در مورد درخت‌ها به کار بردیم، در مورد این درخت هم صدق می‌کند؛ تفاوت درخت دودویی با یک درخت معمولی این است که درجه هر گره نهایتا دو خواهد بود یا به عبارتی ضریب انشعاب این درخت 2 است. از آن جایی که هر گره در نهایت دو فرزند دارد، می‌توانیم فرزندانش را به صورت فرزند چپ Left Child و فرزند راست Right Child صدا بزنیم. به گره‌هایی که فرزند ریشه هستند اینگونه می‌گوییم که گره فرزند چپ با همه فرزندانش می‌شوند زیر درخت چپ Left SubTree و گره سمت راست ریشه با تمام فرزندانش زیر درخت راست Right SubTree صدا زده می‌شوند.



نحوه پیمایش درخت دودویی

این درخت پیمایش‌های گوناگونی دارد ولی سه تای آن‌ها اصلی‌تر و مهمتر هستند:

In-order یا LVR (چپ، ریشه، راست) : در این حالت ابتدا گره‌های سمت چپ ملاقات (چاپ) می‌شوند و سپس ریشه و بعد گره‌های سمت راست.

Pre-Order یا VLR (ریشه، چپ، راست) : در این حالت ابتدا گره‌های ریشه ملاقات می‌شوند. بعد گره‌های سمت چپ و بعد گره‌های سمت راست.

Post_Order یا LRV (چپ، راست، ریشه) : در این حالت ابتدا گره‌های سمت چپ، بعد راست و نهایتا ریشه، ملاقات می‌شوند.

حتما متوجه شده‌اید که منظور از v در اینجا ریشه است و با تغییر و جابجایی مکان این سه حرف RLV میتوانید به ترکیب‌های مختلفی از پیمایش دست پیدا کنید.

اجازه دهید روی شکل بالا پیمایش LVR را انجام دهیم: همانطور که گفتیم باید اول گره‌های سمت چپ را خواند، پس از 17 به سمت 9 حرکت می‌کنیم و می‌بینیم که 9، خود والد است. پس به سمت 6 حرکت می‌کنیم و می‌بینیم که فرزند چپی ندارد؛ پس خود 6 را ملاقات می‌کنیم، سپس فرزند راست را هم بررسی می‌کنیم که فرزند راستی ندارد پس کار ما اینجا تمام است و به سمت بالا حرکت می‌کنیم. 9 را ملاقات می‌کنیم و بعد عدد 5 را و به 17 بر می‌گردیم. 17 را ملاقات کرده و سپس به سمت 15 می‌رویم و الی آخر ...

```
6-9-5-17-8-15-10
```

:VLR

```
17-9-6-5-15-8-10
```

:LRV

```
6-5-9-8-10-15-17
```

نحوه پیاده سازی درخت دودویی:

```
public class BinaryTree<T>
{
    /// <summary>مقدار داخل گره</summary>
    public T Value { get; set; }

    /// <summary>فرزند چپ گره</summary>
    public BinaryTree<T> LeftChild { get; private set; }

    /// <summary>فرزند راست گره</summary>
    public BinaryTree<T> RightChild { get; private set; }

    /// <summary>سازنده کلاس</summary>
    /// <param name="value">مقدار گره</param>
    /// <param name="leftChild">فرزند چپ</param>
    /// <param name="rightChild">فرزند راست
    /// </param>
    public BinaryTree(T value,
        BinaryTree<T> leftChild, BinaryTree<T> rightChild)
    {
        this.Value = value;
        this.LeftChild = leftChild;
        this.RightChild = rightChild;
    }

    /// <summary>سازنده بدون فرزند
    /// </summary>
    /// <param name="value">the value of the tree node</param>
    public BinaryTree(T value) : this(value, null, null)
    {
    }

    /// <summary>LVR پیمایش</summary>
    public void PrintInOrder()
    {
        // ملاقات فرزندان زیر درخت چپ
        if (this.LeftChild != null)
        {
            this.LeftChild.PrintInOrder();
        }

        // ملاقات خود ریشه
        Console.Write(this.Value + " ");

        // ملاقات فرزندان زیر درخت راست
        if (this.RightChild != null)
        {
            this.RightChild.PrintInOrder();
        }
    }
```

```
}

/// <summary>
/// نحوه استفاده از کلاس بالا
/// </summary>
public class BinaryTreeExample
{
    static void Main()
    {
        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));

        binaryTree.PrintInOrder();
        Console.WriteLine();

        // خروجی
        // 23 19 10 6 21 14 3 15
    }
}
```

تفاوتی که این کد با کد قبلی که برای یک درخت معمولی داشتیم، در این است که قبلا لیستی از فرزندان را داشتیم که با خاصیت Children شناخته می‌شدند، ولی در اینجا در نهایت دو فرزند چپ و راست برای هر گره وجود دارند. برای جست و جو هم از الگوریتم In_Order استفاده کردیم که از همان الگوریتم DFS آمده‌است. در آنجا هم ابتدا گره‌های سمت چپ به صورت بازگشتی صدا زده می‌شدند. بعد خود گره و سپس گره‌های سمت راست به صورت بازگشتی صدا زده می‌شدند.

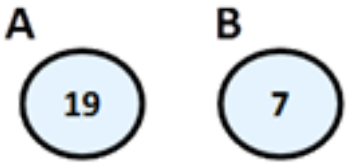
برای باقی روش‌های پیمایش تنها نیاز است که این سه خط را جابجا کنید:

```
// ملاقات فرزندان زیر درخت چپ
        if (this.LeftChild != null)
        {
            this.LeftChild.PrintInOrder();
        }

        // ملاقات خود ریشه
        Console.Write(this.Value + " ");

        // ملاقات فرزندان زیر درخت راست
        if (this.RightChild != null)
        {
            this.RightChild.PrintInOrder();
        }
```

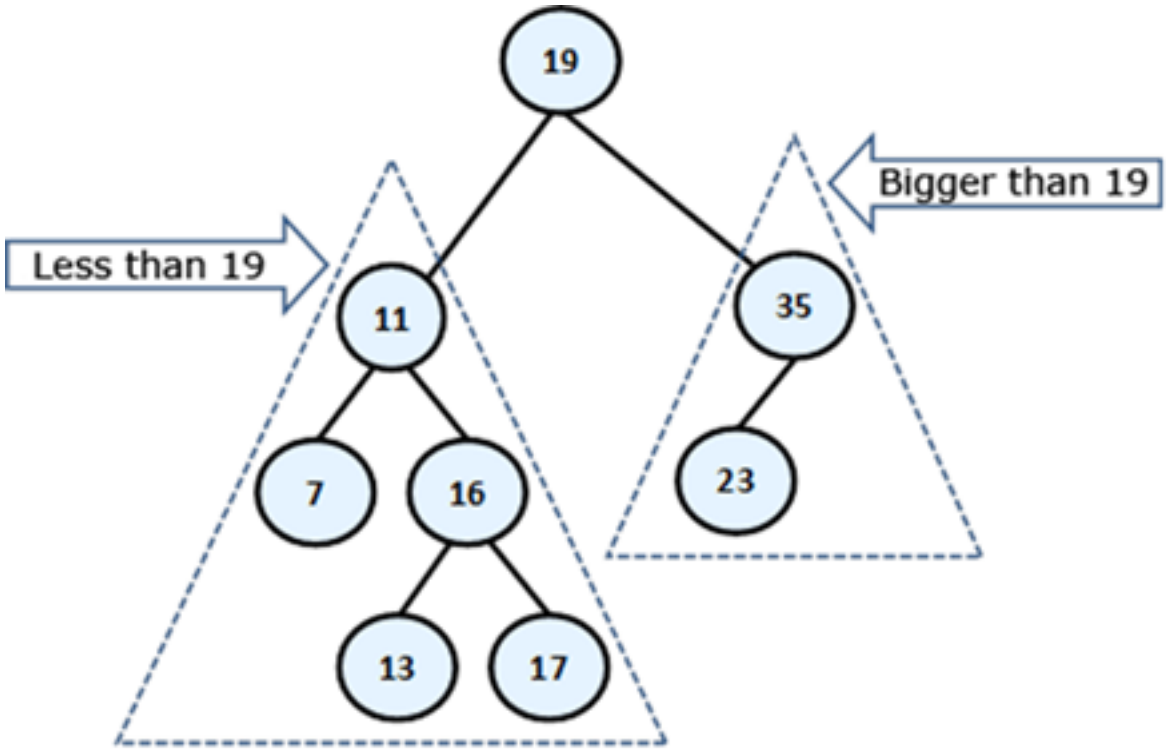
درخت دودویی مرتب شده Ordered Binary Search Tree

تا این لحظه ما با ساخت درخت‌های پایه آشنا شدیم: درخت عادی یا کلاسیک و درخت دو دویی. ولی در بیشتر موارد در پروژه‌های بزرگ از این‌ها استفاده نمی‌کنیم چرا که استفاده از آن‌ها در پروژه‌های بزرگ بسیار مشکل است و باید به جای آن‌ها از ساختارهای متنوع دیگری از قبیل درخت‌های مرتب شده، کم عمق و متوازن و کامل و پر و .. استفاده کرد. پس اجازه دهید که مهمترین درخت‌هایی را که در برنامه نویسی استفاده می‌شوند، بررسی کنیم.

همان طور که می‌دانید برای مقایسه اعداد ما از علامتهای <>= استفاده می‌کنیم و اعداد صحیح بهترین اعداد برای مقایسه هستند. در درخت‌های جست و جوی دو دویی یک خصوصیت اضافه به اسم **کلید هویت یکتا Unique identification Key** داریم که یک کلید قابل مقایسه است. در تصویر زیر ما دو گره با مقدارهای متفاوتی داریم که با مقایسه‌ی آنان می‌توانیم کوچک و بزرگ بودن یک گره را محاسبه کنیم. ولی به این نکته دقت داشته باشید که این اعداد داخل دایره‌ها، دیگر برای ما حکم مقدار ندارند و کلیدهای یکتا و شاخص هر گره محسوب می‌شوند.



خلاصه‌ی صحبت‌های بالا: در هر درخت دودویی مرتب شده، گره‌های بزرگتر در زیر درخت راست قرار دارند و گره‌های کوچکتر در زیر درخت چپ قرار دارند که این کوچکی و بزرگی بر اساس یک کلید یکتا که قابل مقایسه است استفاده می‌شود.



این درخت دو دویی مرتب شده در جست و جو به ما کمک فراوانی می‌کند و از آنجا که می‌دانیم زیر درخت‌های چپ مقدار کمتری دارند و زیر درخت‌های راست مقدار بیشتر، عمل جست و جو، مقایسه‌های کمتری خواهد داشت، چرا که هر بار مقایسه یک زیر درخت کنار گذاشته می‌شود.

برای مثال فکر کنید می‌خواهید عدد 13 را در درخت بالا پیدا کنید. ابتدا گره والد 19 را مقایسه کرده و از آنجا که 19 بزرگتر از 13 است می‌دانیم که 13 را در زیر درخت راست پیدا نمی‌کنیم. پس زیر درخت چپ را مقایسه می‌کنیم (بنابراین به راحتی یک زیر درخت از مقایسه و عمل جست و جو کنار گذاشته شد). سپس گره 11 را مقایسه می‌کنیم و از آنجا که 11 کوچکتر از 13 هست، زیر درخت سمت راست را ادامه می‌دهیم و چون 16 بزرگتر از 13 هست، زیر درخت سمت چپ را در ادامه مقایسه می‌کنیم که به 13 رسیدیم.

مقایسه گره‌هایی که برای جست و جو انجام دادیم:

```
19-11-16-13
```

درخت هر چه بزرگتر باشد این روش کارآیی خود را بیشتر نشان می‌دهد.

در قسمت بعدی به پیاده سازی کد این درخت به همراه متدهای افزودن و جست و جو و حذف می‌پردازیم.