عنوان: **درخت‌ها و گراف‌ها قسمت سوم**
نویسنده: علی یگانه مقدم
تاریخ: ۱۳۹۳/۱۲/۰۵ ۱۰:۴۰
آدرس: www.dotnettips.info
گروه‌ها: C#, Data Structures, Tree

همانطور که در قسمت قبلی گفتیم، در این قسمت قرار است به پیاده سازی درخت جست و جوی دو دویی مرتب شده بپردازیم. در مطلب قبلی اشاره کردیم که ما متدهای افزودن، جستجو و حذف را قرار است به درخت اضافه کنیم و برای هر یک از این متدها توضیحاتی را ارائه خواهیم کرد. به این نکته دقت داشته باشید درختی که قصد پیاده سازی آن را داریم یک درخت متوازن نیست و ممکن است در بعضی شرایط کارآیی مطلوبی نداشته باشد.

همانند مثال‌ها و پیاده سازی‌های قبلی، دو کلاس داریم که یکی برای ساختار گره است BinaryTreeNode<T> و دیگری برای ساختار درخت اصلی BinaryTree<T>.

کلاس BinaryTreeNode که در پایین نوشته شده‌است بعدا داخل کلاس BinaryTree قرار خواهد گرفت:

```
internal class BinaryTreeNode<T> :
    IComparable<BinaryTreeNode<T>> where T : IComparable<T>
{
    // مقدار گره
    internal T value;

    // شامل گره پدر
    internal BinaryTreeNode<T> parent;

    // شامل گره سمت چپ
    internal BinaryTreeNode<T> leftChild;

    // شامل گره سمت راست
    internal BinaryTreeNode<T> rightChild;

    /// <summary>سازنده</summary>
    /// <param name="value">مقدار گره ریشه</param>
    public BinaryTreeNode(T value)
    {
        if (value == null)
        {
            // از آن جا که نال قابل مقایسه نیست اجازه افزودن را از آن سلب می‌کنیم
            throw new ArgumentNullException(
                "Cannot insert null 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);
    }
}
```

تکلیف کدهای اولیه که کامنت دارند روشن است و قبلا چندین بار بررسی کردیم ولی کدها و متدهای جدیدتری نیز نوشته شده‌اند

که آن‌ها را بررسی می‌کنیم:
ما در مورد این درخت می‌گوییم که همه چیز آن مرتب شده است و گره‌ها به ترتیب چیده شده اند و اینکار تنها با مقایسه کردن گره‌های درخت امکان پذیر است. این مقایسه برای برنامه نویسان از طریق یک ذخیره در یک ساختمان داده خاص یا اینکه آن را به یک *نوع Type* قابل مقایسه ارسال کنند امکان پذیر است. در سی شارپ نوع قابل مقایسه با کلمه‌های کلیدی زیر امکان پذیر است:

```
T : IComparable<T>
```

در اینجا T می‌تواند هر نوع داده‌ای مانند Byte و int و ... باشد؛ ولی علامت : این محدودیت را اعمال می‌کند که کلاس باید از اینترفیس IComparable ارث بری کرده باشد. این اینترفیس برای پیاده‌سازی تنها شامل تعریف یک متد است به نام CompareTo(T) obj که عمل مقایسه داخل آن انجام می‌گردد و در صورت بزرگ بودن شیء جاری از آرگومان داده شده، نتیجه‌ی برگردانده شده، مقداری مثبت، در حالت برابر بودن، مقدار 0 و کوچکتر بودن مقدارمنفی خواهد بود. شکل تعریف این اینترفیس تقریبا چنین چیزی باید باشد:

```
public interface IComparable<T>
{
    int CompareTo(T other);
}
```

نوشتن عبارت بالا در جلوی کلاس، به ما این اطمینان را می‌بخشد که که نوع یا کلاسی که به آن پاس می‌شود، یک نوع قابل مقایسه است و از طرف دیگر چون می‌خواهیم گره‌هایمان نوعی قابل مقایسه باشند IComparable<T> را هم برای آن ارث بری می‌کنیم.
همچنین چند متد دیگر را نیز override کرده‌ایم که اصلی‌ترین آن‌ها GetHashCode و Equal است. موقعی که متد CompareTo مقدار 0 بر می‌گرداند مقدار برگشتی Equals هم باید True باشد.
... و یک نکته مفید برای خاطرسپاری اینکه موقعیکه دو شیء با یکدیگر برابر باشند، کد هش تولید شده آن‌ها نیز با هم برابر هستند. به عبارتی اشیاء یکسان کد هش یکسانی دارند. این رفتار سبب می‌شود که که بتوانید مشکلات زیادی را که در رابطه با مقایسه کردن پیش می‌آید، حل نمایید.

پیاده سازی کلاس اصلی BinarySearchTree

مهمترین نکته در کلاس زیر این مورد است که ما اصرار داشتیم، T باید از اینترفیس IComparable مشتق شده باشد. بر این حسب ما می‌توانیم با نوع داده‌هایی چون int یا string کار کنیم، چون قابل مقایسه هستند ولی نمی‌توانیم با int[] یا streamreader کار کنیم چرا که قابل مقایسه نیستند.

```
public class BinarySearchTree<T>    where T : IComparable<T>
{
    /// کلاسی که بالا تعریف کردیم
    internal class BinaryTreeNode<T> :
        IComparable<BinaryTreeNode<T>> where T : IComparable<T>
    {
        // …
    }

    /// <summary>
    /// ریشه درخت
    /// </summary>
    private BinaryTreeNode<T> root;

    /// <summary>
    /// سازنده کلاس
    /// </summary>
    public BinarySearchTree()
    {
        this.root = null;
    }

//پیاده سازی متدها مربوط به افزودن و حذف و جست و جو
}
```

در کد بالا ما کلاس اطلاعات گره را به کلاس اضافه می‌کنیم و یه سازنده و یک سری خصوصیت رابه آن اضافه کرده ایم.در این مرحله گام به گام هر یک از سه متد افزودن ، جست و جو و حذف را بررسی می‌کنیم و جزئیات آن را توضیح می‌دهیم.

افزودن یک عنصر جدید

افزودن یک عنصر جدید در این درخت مرتب شده، مشابه درخت‌های قبلی نیست و این افزودن باید طوری باشد که مرتب بودن درخت حفظ گردد. در این الگوریتم برای اضافه شدن عنصری جدید، دستور العمل چنین است: اگر درخت خالی بود عنصر را به عنوان ریشه اضافه کن؛ در غیر این صورت مراحل زیر را نجام بده:
اگر عنصر جدید کوچکتر از ریشه است، با یک تابع بازگشتی عنصر جدید را به زیر درخت چپ اضافه کن.
اگر عنصر جدید بزرگتر از ریشه است، با یک تابع بازگشتی عنصر جدید را به زیر درخت راست اضافه کن.
اگر عنصر جدید برابر ریشه هست، هیچ کاری نکن و خارج شو.

پیاده سازی الگوریتم بالا در کلاس اصلی:

```
public void Insert(T value)
{
    this.root = Insert(value, null, root);
}

/// <summary>
/// متدی برای افزودن عنصر به درخت
/// </summary>
/// <param name="value">مقدار جدید</param>
/// <param name="parentNode">والد گره جدید</param>
/// <param name="node">گره فعلی که همان ریشه است</param>
/// <returns>گره افزوده شده</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;
}
```

متد درج سه آرگومان دارد، یکی مقدار گره جدید است؛ دوم گره والد که با هر بار صدا زدن تابع بازگشتی، گره والد تغییر خواهد کرد و به گره‌های پایین‌تر خواهد رسید و سوم گره فعلی که با هر بار پاس شدن به تابع بازگشتی، گره ریشه‌ی آن زیر درخت است.
در مقاله قبلی اگر به یاد داشته باشید گفتیم که جستجو چگونه انجام می‌شود و برای نمونه به دنبال یک عنصر هم گشتیم و جستجوی یک عنصر در این درخت بسیار آسان است. ما این کد را بدون تابع بازگشتی و تنها با یک حلقه while پیاده خواهیم کرد. هر چند مشکلی با پیاده سازی آن به صورت بازگشتی وجود ندارد.
الگوریتم از ریشه بدین صورت آغاز می‌گردد و به ترتیب انجام می‌شود:
اگر عنصر جدید برابر با گره فعلی باشد، همان گره را بازگشت بده.
اگر عنصر جدید کوچکتر از گره فعلی است، گره سمت چپ را بردار و عملیات را از ابتدا آغاز کن (در کد زیر به ابتدای حلقه برو).
اگر عنصر جدید بزرگتر از گره فعلی است، گره سمت راست را بردار و عملیات را از ابتدا آغاز کن.

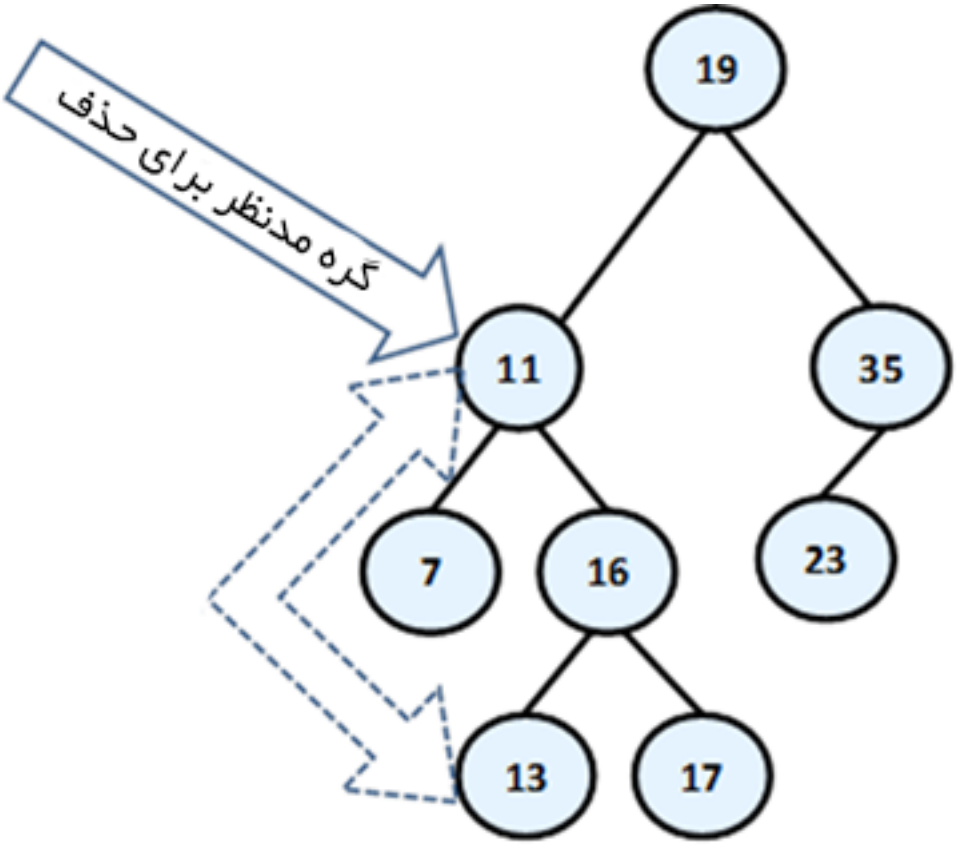
در انتها اگر الگوریتم، گره را پیدا کند، گره پیدا شده را باز می‌گرداند؛ ولی اگر گره را پیدا نکند، یا درخت خالی باشد، مقدار برگشتی نال خواهد بود.

حذف یک عنصر

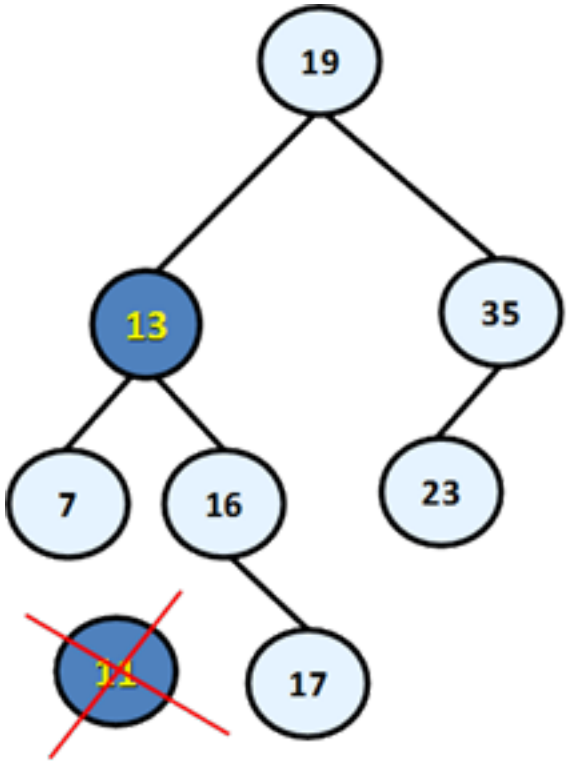
حذف کردن در این درخت نسبت به درخت دودویی معمولی پیچیده‌تر است. اولین گام این عمل، جستجوی گره مدنظر است. وقتی گره‌ایی را مدنظر داشته باشیم، سه بررسی زیر انجام می‌گیرد:

اگر گره برگ هست و والد هیچ گره‌ای نیست، به راحتی گره مد نظر را حذف می‌کنیم و ارتباط گره والد با این گره را نال می‌کنیم.
اگر گره تنها یک فرزند دارد (هیچ فرقی نمی‌کند چپ یا راست) گره مدنظر حذف و فرزندش را جایگزینش می‌کنیم.
اگر گره دو فرزند دارد، کوچکترین گره در زیر درخت سمت راست را پیدا کرده و با گره مدنظر جابجا می‌کنیم. سپس یکی از دو عملیات بالا را روی گره انجام می‌دهیم.

اجازه دهید عملیات بالا را به طور عملی بررسی کنیم. در درخت زیر ما می‌خواهیم گره 11 را حذف کنیم. پس کوچکترین گره سمت راست، یعنی 13 را پیدا می‌کنیم و با گره 11 جابجا می‌کنیم.



بعد از جابجایی، یکی از دو عملیات اول بالا را روی گره 11 اعمال می‌کنیم و در این حالت گره 11 که یک گره برگ است، خیلی راحت حذف و ارتباطش را با والد، با یک نال جایگزین می‌کنیم.




```
/// عنصر مورد نظر را جست و جوی می‌کند و اگر مخالف نال بود گره برگشتی را به تابع حذف ارسال می‌کند
public void Remove(T value)
{
    BinaryTreeNode<T> nodeToDelete = Find(value);
    if (nodeToDelete != null)
    {
        Remove(nodeToDelete);
    }
}

private void Remove(BinaryTreeNode<T> node)
{
    //بررسی می‌کند که آیا دو فرزند دارد یا خیر
    // این خط باید اول همه باشد که مرحله یک و دو بعد از آن اجرا شود
    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;
    }

    // مرحله یک و دو اینجا بررسی میشه
    BinaryTreeNode<T> theChild = node.leftChild != null ?
            node.leftChild : node.rightChild;

    // اگر حداقل یک فرزند داشته باشد
    if (theChild != null)
    {
        theChild.parent = node.parent;

        // بررسی می‌کند گره ریشه است یا خیر
        if (node.parent == null)
        {
            root = theChild;
        }
        else
        {
            // جایگزینی عنصر با زیر درخت فرزندش
            if (node.parent.leftChild == node)
            {
                node.parent.leftChild = theChild;
```

```
            }
            else
            {
                node.parent.rightChild = theChild;
            }
        }
    }
    else
    {
        // کنترل وضعیت موقعی که عنصر ریشه است
        if (node.parent == null)
        {
            root = null;
        }
        else
        {
            // اگر گره برگ است آن را حذف کن
            if (node.parent.leftChild == node)
            {
                node.parent.leftChild = null;
            }
            else
            {
                node.parent.rightChild = null;
            }
        }
    }
}
```

در کد بالا ابتدا جستجو انجام می‌شود و اگر جواب غیر نال بود، گره برگشتی را به تابع حذف ارسال می‌کنیم. در تابع حذف اول از همه بررسی می‌کنیم که آیا گره ما دو فرزند دارد یا خیر که اگر دو فرزنده بود، ابتدا گره‌ها را تعویض و سپس یکی از مراحل یک یا دو را که در بالاتر ذکر کردیم، انجام دهیم.

دو فرزندی

اگر گره ما دو فرزند داشته باشد، گره سمت راست را گرفته و از آن گره آن قدر به سمت چپ حرکت می‌کنیم تا به برگ یا گره تک فرزنده که صد در صد فرزندش سمت راست است، برسیم و سپس این دو گره را با هم تعویض می‌کنیم.

تک فرزندی

در مرحله بعد بررسی می‌کنیم که آیا گره یک فرزند دارد یا خیر؛ شرط بدین صورت است که اگر فرزند چپ داشت آن را در theChild قرار می‌دهیم، در غیر این صورت فرزند راست را قرار می‌دهیم. در خط بعدی باید چک کرد که theChild نال است یا خیر. اگر نال باشد به این معنی است که غیر از فرزند چپ، حتی فرزند راست هم نداشته، پس گره، یک برگ است ولی اگر مخالف نال باشد پس حداقل یک گره داشته است.

اگر نتیجه نال نباشد باید این گره حذف و گره فرزند ارتباطش را با والد گره حذفی برقرار کند. در صورتیکه گره حذفی ریشه باشد و والدی نداشته باشد، این نکته باید رعایت شود که گره فرزند بری متغیر root که در سطح کلاس تعریف شده است، نیز قابل شناسایی باشد.

در صورتی که خود گره ریشه نباشد و والد داشته باشد، غیر از اینکه فرزند باید با والد ارتباط داشته باشد، والد هم باید از طریق دو خاصیت فرزند چپ و راست با فرزند ارتباط برقرار کند. پس ابتدا بررسی می‌کنیم که گره حذفی کدامین فرزند بوده: چپ یا راست؟ سپس فرزند گره حذفی در آن خاصیت جایگزین خواهد شد و دیگر هیچ نوع اشاره‌ای به گره حذفی نیست و از درخت حذف شده است.

بدون فرزند (برگ)

حال اگر گره ما برگ باشد مرحله دوم، کد داخل else اجرا خواهد شد و بررسی می‌کند این گره در والد فرزند چپ است یا

راست و به این ترتیب با نال کردن آن فرزند در والد ارتباط قطع شده و گره از درخت حذف می‌شود.

پیمایش درخت به روش DFS یا LVR یا In-Order

```
public void PrintTreeDFS()
{
    PrintTreeDFS(this.root);
    Console.WriteLine();
}


private void PrintTreeDFS(BinaryTreeNode<T> node)
{
    if (node != null)
    {
        PrintTreeDFS(node.leftChild);
        Console.Write(node.value + " ");
        PrintTreeDFS(node.rightChild);
    }
}
```

در مقاله بعدی درخت دودویی متوازن را که پیچیده‌تر از این درخت است و از کارآیی بهتری برخوردار هست، بررسی می‌کنیم.