عنوان: **ساختار داده‌های خطی Linear Data Structure قسمت اول**
نویسنده: علی یگانه مقدم
تاریخ: ۱۳۹۳/۱۱/۳۰ ۱:۲۰
آدرس: www.dotnettips.info
گروه‌ها: C#, Data Structures, Linked List

بعضی از داده‌ها ساختارهای ساده‌ای دارند و به صورت یک صف یا یک نوار ضبط به ترتیب پشت سر هم قرار می‌گیرند؛ مثل ساختاری که صفحات یک کتاب را نگهداری می‌کند. یکی از نمونه‌های این ساختارها، List، صف، پشته و مشتقات آن‌ها می‌باشند.

ساختار داده‌ها چیست؟

در اغلب اوقات، موقعی‌که ما برنامه‌ای را می‌نویسیم با اشیاء یا داده‌های زیادی سر و کار داریم که گاهی اوقات اجزایی را به آن‌ها اضافه یا حذف می‌کنیم و در بعضی اوقات هم آن‌ها را مرتب سازی کرده یا اینکه پردازش دیگری را روی آن‌ها انجام میدهیم. به همین دلیل بر اساس کاری که قرار است انجام دهیم، باید داده‌ها را به روش‌های مختلفی ذخیره و نگه داری کنیم و در اکثر این روش‌ها داده‌ها به صورت منظم و پشت سر هم در یک ساختار قرار می‌گیرند.

ما در این مقاله، مجموعه‌ای از داده‌ها را در قالب ساختارهای متفاوتی بر اساس منطق و قوانین ریاضیات مدیریت می‌کنیم و بدیهی است که انتخاب یک ساختار مناسب برای هرکاری موجب افزایش کارآیی و کارآمدی برنامه خواهد گشت. می‌توانیم در مقدار حافظه‌ی مصرفی و زمان، صرفه جویی کنیم و حتی گاهی تعداد خطوط کدنویسی را کاهش دهیم.

نوع داده انتزاعی Abstraction Data Type -ADT

به زبان خیلی ساده لایه انتزاعی به ما تنها یک تعریف از ساختار مشخص شده‌ای را می‌دهد و هیچگونه پیاده سازی در آن وجود ندارد. برای مثال در لایه انتزاعی، تنها خصوصیت و عملگرها و ... مشخص می‌شوند. ولی کد آن‌ها را پیاده سازی نمی‌کنیم و این باعث می‌شود که از روی این لایه بتوانیم پیاده سازی‌های متفاوت و کارآیی‌های مختلفی را ایجاد کنیم.

ساختار داده‌های مختلف در برنامه نویسی:

خطی یا Linear: شامل ساختارهایی چون لیست و صف و پشته است: List ,Queue,Stack

درختی یا Tree-Like: درخت باینری ، درخت متوازن و B-Trees

Dictionary : شامل یک جفت کلید و مقدار است در جدول هش

بقیه: گراف‌ها، صف الویت، bags, Multi bags, multi sets

در این مقاله تنها ساختارهای خطی را دنبال می‌کنیم و در آینده ساختارهای پیچیده‌تری را نیز بررسی خواهیم کرد و نیاز است بررسی کنیم کی و چگونه باید از آن‌ها استفاده کنیم.

ساختارهای لیستی از محبوبترین و پراستفاده‌ترین ساختارها هستند که با اشیاء زیادی در دنیای واقعی سازگاری دارند. مثال زیر را در نظر بگیرید:

قرار است که ما از فروشگاهی خرید کنیم و هر کدام از اجناس (المان‌ها) فروشگاه را که در سبد قرار دهیم، نام آن‌ها در یک لیست ثبت خواهد شد و اگر دیگر المان یا جنسی را از سبد بیرون بگذاریم، از لیست خط خواهد خورد.

همان که گفتیم یک ADT میتواند ساختارهای متفاوتی را پیاده سازی کند. یکی از این ساختارها اینترفیس system.collection.IList است که پیاده سازی آن منجر به ایجاد یک کلاس جدید در سیستم دات نت خواهد شد. پیاده سازی اینترفیس‌ها در سی شارپ، قوانین و قراردادهای خاص خودش را دارد و این قوانین شامل مجموعه‌ای از متدها و خصوصیت‌هاست. برای پیاده سازی هر کلاسی از این اینترفیس‌ها باید این متدها و خصوصیت‌ها را هم در آن پیاده کرد.

با ارث بری از اینترفیس system.collection.IList باید رابطهای زیر در آن پیاده سازی گردد:

void Add(object)	افزودن المان به آخر لیست
void Remove(object)	حذف یک المان خاص از لیست
void Clear()	حذف کلیه المان‌ها
bool Contains(object)	شامل این داده میشود یا خیر؟
void RemoveAt(int)	حذف یک المان بر اساس جایگاه یا اندیسش
void Insert(int, object)	افزودن یک المان در جایگاهی (اندیس) خاص بر اساس مقدار position

(void Add(object	افزودن المان به آخر لیست
(int IndexOf(object	اندیس یا جایگاه یک عنصر را بر می‌گرداند
[this[int	ایندکسر ، برای دستریس به عنصر در اندیس مورد نظر

لیست‌های ایستا static Lists

آرایه‌ها می‌توانند بسیاری از خصوصیات ADT را پیاده کنند ولی تفاوت بسیار مهم و بزرگی با آن‌ها دارند و آن این است که لیست به شما اجازه می‌دهد به هر تعدادی که خواستید، المان‌های جدیدی را به آن اضافه کنید؛ ولی یک آرایه دارای اندازه‌ی ثابت Fix است. البته این نکته قابل تامل است که پیاده سازی لیست با آرایه‌ها نیز ممکن است و باید به طور خودکار طول آرایه را افزایش دهید. دقیقا همان اتفاقی که برای stringbuilder در این مقاله توضیح دادیم رخ می‌دهد. به این نوع لیست‌ها، *لیست‌های ایستایی که به صورت آرایه ای توسعه پذیر پیاده سازی میشوند* می‌گویند. کد زیر پیاده سازی چنین لیستی است:

```
public class CustomArrayList<T>
{
    private T[] arr;
    private int count;

    public int Count
    {
        get
        {
            return this.count;
        }
    }

    private const int INITIAL_CAPACITY = 4;

    public CustomArrayList(int capacity = INITIAL_CAPACITY)
    {
        this.arr = new T[capacity];
        this.count = 0;
    }
```

در کد بالا یک آرایه با طول متغیر INITIAL_CAPACITY که پیش فرض آن را 4 گذاشته ایم می‌سازیم و از متغیر count برای حفظ تعداد عناصر آرایه استفاده می‌کنیم و اگر حین افزودن المان جدید باشیم و count بزرگتر از INITIAL_CAPACITY رسیده باشد، باید طول آرایه افزایش پیدا کند که کد زیر نحوه‌ی افزودن المان جدید را نشان می‌دهد. استفاده از حرف T بزرگ مربوط به مباحث Generic هست. به این معنی که المان ورودی می‌تواند هر نوع داده‌ای باشد و در آرایه ذخیره شود.

```
public void Add(T item)
{
    GrowIfArrIsFull();
    this.arr[this.count] = item;
    this.count++;
}

public void Insert(int index, T item)
{
    if (index > this.count || index < 0)
    {
        throw new IndexOutOfRangeException(
            "Invalid index: " + index);
    }
    GrowIfArrIsFull();
    Array.Copy(this.arr, index,
        this.arr, index + 1, this.count - index);
    this.arr[index] = item;
    this.count++;
}

private void GrowIfArrIsFull()
{
    if (this.count + 1 > this.arr.Length)
    {
        T[] extendedArr = new T[this.arr.Length * 2];
        Array.Copy(this.arr, extendedArr, this.count);
        this.arr = extendedArr;
    }
```

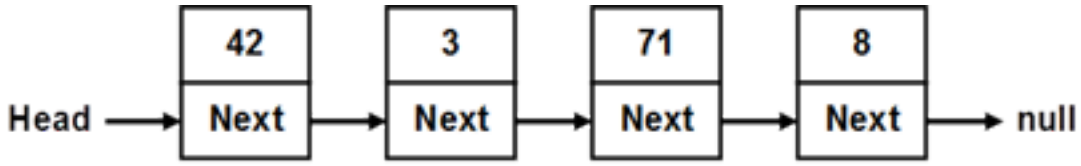
```
}

public void Clear()
{
    this.arr = new T[INITIAL_CAPACITY];
    this.count = 0;
}
```

در متد Add خط اول با تابع GrowIfArrIsFull بررسی می‌کند آیا خانه‌های آرایه کم آمده است یا خیر؟ اگر جواب مثبت باشد، طول آرایه را دو برابر طول فعلی‌اش افزایش می‌دهد و خط دوم المان جدیدی را در اولین خانه‌ی جدید اضافه شده قرار می‌دهد. همانطور که می‌دانید مقدار count همیشه یکی بیشتر از آخرین اندیس است. پس به این ترتیب مقدار count همیشه به خانه‌ی بعدی اشاره می‌کند و سپس مقدار count به روز میشود. متد دیگری که در کد بالا وجود دارد insert است که المان جدیدی را در اندیس داده شده قرار می‌دهد. جهت این کار از سومین سازنده‌ی array.copy استفاده می‌کنیم. برای این کار آرایه مبدا و مقصد را یکی در نظر می‌گیریم و از اندیس داده شده به بعد در آرایه فعلی، یک کپی تهیه کرده و در خانه‌ی بعد اندیس داده شده به بعد قرار می‌دهیم. با این کار آرایه ما یک واحد از اندیس داده شده یک خانه، به سمت جلو حرکت می‌کند و الان خانه index و index+1 دارای یک مقدار هستند که در خط بعدی مقدار جدید را داخل آن قرار می‌دهیم و متغیر count را به روز می‌کنیم. باقی موارد را چون پردازش‌های جست و جو، پیدا کردن اندیس یک المان و گزینه‌های حذف، به خودتان واگذار می‌کنم.

لیست‌های پیوندی Linked List - پیاده سازی پویا

همانطور که دیدید لیست‌های ایستا دارای مشکل بزرگی هستند و آن هم این است که با انجام هر عملی بر روی آرایه‌ها مانند افزودن، درج در مکانی خاص و همچنین حذف (خانه ای در آرایه خالی خواهد شد و خانه‌های جلوترش باید یک گام به عقب برگردند) نیاز است که خانه‌های آرایه دوباره مرتب شوند که هر چقدر میزان داده‌ها بیشتر باشد این مشکل بزرگتر شده و ناکارآمدی برنامه را افزایش خواهد داد.

این مشکل با لیست‌های پیوندی حل می‌گردد. در این ساختار هر المان حاوی اطلاعاتی از المان بعدی است و در لیست‌های پیوندی دوطرفه حاوی المان قبلی است. شکل زیر نمایش یک لیست پیوندی در حافظه است:



برای پیاده سازی آن به دو کلاس نیاز داریم. کلاس ListNode برای نگهداری هر المان و اطلاعات المان بعدی به کار می‌رود که از این به بعد به آن Node یا گره می‌گوییم و دیگری کلاس DynamicList<T> برای نگهداری دنباله ای از گره‌ها و متدهای پردازشی آن.

```
public class DynamicList<T>
{
    private class ListNode
    {
        public T Element { get; set; }
        public ListNode NextNode { get; set; }

        public ListNode(T element)
        {
            this.Element = element;
            NextNode = null;
        }

        public ListNode(T element, ListNode prevNode)
        {
            this.Element = element;
            prevNode.NextNode = this;
        }
    }

    private ListNode head;
    private ListNode tail;
    private int count;

    // …
```

```
}
```

از آن جا که نیازی نیست کاربر با کلاس ListNode آشنایی داشته باشد و با آن سر و کله بزند، آن را داخل همان کلاس اصلی به صورت خصوصی استفاده می‌کنیم. این کلاس دو خاصیت دارد؛ یکی برای المان اصلی و دیگر گره بعدی. این کلاس دارای دو سازنده است که اولی تنها برای عنصر اول به کار می‌رود. چون اولین بار است که یک گره ایجاد می‌شود، پس باید خاصیت NextNode یعنی گره بعدی در آن Null باشد و سازنده‌ی دوم برای گره‌های شماره 2 به بعد به کار می‌رود که همراه المان داده شده، گره قبلی را هم ارسال می‌کنیم تا خاصیت NextNode آن را به گره جدیدی که می‌سازیم مرتبط سازد. سه خاصیت کلاس اصلی به نام‌های Count,Tail,Head به ترتیب برای اشاره به اولین گره، آخرین گره و تعداد گره‌ها، به کار می‌روند که در ادامه کد آن‌را در زیر می‌بینیم:

```
public DynamicList()
{
    this.head = null;
    this.tail = null;
    this.count = 0;
}

public void Add(T item)
{
    if (this.head == null)
    {
        this.head = new ListNode(item);
        this.tail = this.head;
    }
    else
    {
        ListNode newNode = new ListNode(item, this.tail);
        this.tail = newNode;
    }
    this.count++;
}
```

سازنده مقدار دهی پیش فرض را انجام می‌دهد. در متد Add المان جدیدی باید افزوده شود؛ پس چک می‌کند این المان ارسالی قرار است اولین گره باشد یا خیر؟ اگر head که به اولین گره اشاره دارد Null باشد، به این معنی است که این اولین گره است. پس اولین سازنده‌ی کلاس ListNode را صدا می‌زنیم و آن را در متغیر Head قرار می‌دهیم و چون فقط همین گره را داریم، پس آخرین گره هم شناخته می‌شود که در tail نیز قرار می‌گیرد. حال اگر فرض کنیم المان بعدی را به آن بدهیم، اینبار دیگر Head برابر Null نخواهد بود. پس دومین سازنده‌ی ListNode صدا زده می‌شود که به غیر از المان جدید، باید آخرین گره قبلی هم با آن ارسال شود و گره جدیدی که ایجاد می‌شود در خاصیت NextNode آن نیز قرار بگیرد و در نهایت گره ایجاد شده به عنوان آخرین گره لیست در متغیر Tail نیز قرار می‌گیرد. در خط پایانی هم به هر مدلی که المان جدید به لیست اضافه شده باشد متغیر Count به روز می‌شود.

```
public T RemoveAt(int index)
{
    if (index >= count || index < 0)
    {
        throw new ArgumentOutOfRangeException(
            "Invalid index: " + index);
    }

    int currentIndex = 0;
    ListNode currentNode = this.head;
    ListNode prevNode = null;
    while (currentIndex < index)
    {
        prevNode = currentNode;
        currentNode = currentNode.NextNode;
        currentIndex++;
    }


    RemoveListNode(currentNode, prevNode);

    return currentNode.Element;
}
```

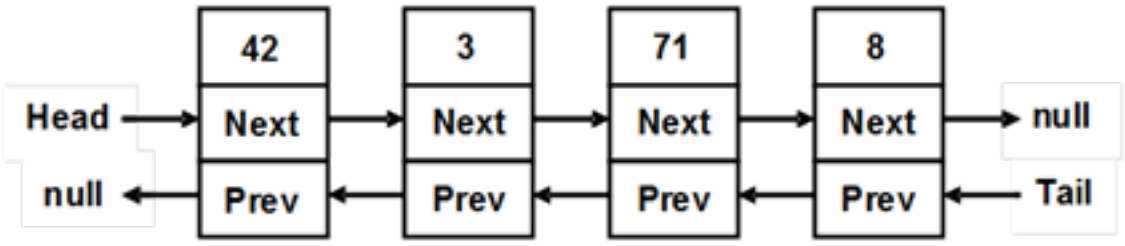
```
private void RemoveListNode(ListNode node, ListNode prevNode)
{
    count--;
    if (count == 0)
    {
        this.head = null;
        this.tail = null;
    }
    else if (prevNode == null)
    {
        this.head = node.NextNode;
    }
    else
    {
        prevNode.NextNode = node.NextNode;
    }

    if (object.ReferenceEquals(this.tail, node))
    {
        this.tail = prevNode;
    }
}
```

برای حذف یک گره شماره اندیس آن گره را دریافت می‌کنیم و از Head، گره را بیرون کشیده و با خاصیت nextNode آنقدر به سمت جلو حرکت می‌کنیم تا متغیر currentIndex یا اندیس داده شده برابر شود و سپس گره دریافتی و گره قبلی آن را به سمت تابع RemoveListNode ارسال می‌کنیم. کاری که این تابع انجام می‌دهد این است که مقدار NextNode گره فعلی که قصد حذفش را داریم به خاصیت Next Node گره قبلی انتساب می‌دهد. پس به این ترتیب پیوند این گره از لیست از دست می‌رود و گره قبلی به جای اشاره به این گره، به گره بعد از آن اشاره می‌کند. مابقی کد از قبیل جست و برگردان اندیس یک عنصر و ... را به خودتان وگذار می‌کنم.

در روش‌های بالا ما خودمان 2 عدد ADT را پیاده سازی کردیم و متوجه شدیم برای دخیره داده‌ها در حافظه روش‌های متفاوتی وجود دارند که بیشتر تفاوت آن در مورد استفاده از حافظه و کارآیی این روش هاست.

لیست‌های پیوندی دو طرفه Doubly Linked_List

لیست‌های پیوندی بالا یک طرفه بودند و اگر ما یک گره را داشتیم و می‌خواستیم به گره قبلی آن رجوع کنیم، اینکار ممکن نبود و مجبور بودیم برای رسیدن به آن از ابتدای گره حرکت را آغاز کنیم تا به آن برسیم. به همین منظور مبحث لیست‌های پیوندی دو طرفه آغاز شد. به این ترتیب هر گره به جز حفظ ارتباط با گره بعدی از طریق خاصیت NextNode، ارتباطش را با گره قبلی از طریق خاصیت PrevNode نیز حفظ می‌کند.



این مبحث را در اینجا می‌بندیم و در قسمت بعدی آن را ادامه می‌دهیم.

عنوان: **ساختار داده‌های خطی Linear Data Structure قسمت دوم**
نویسنده: علی یگانه مقدم
تاریخ: ۱۳۹۳/۱۲/۰۱ ۲:۱۰
آدرس: www.dotnettips.info
گروه‌ها: C#, Generics, Data Structures, Linked List

در قسمت قبلی به مقدمات و ساخت لیست‌های ایستا و پویا به صورت دستی پرداختیم و در این قسمت (مبحث پایانی) لیست‌های آماده در دات نت را مورد بررسی قرار می‌دهیم.

کلاس ArrayList

این کلاس همان پیاده سازی لیست‌های ایستایی را دارد که در مطلب پیشین در مورد آن صحبت کردیم و نحوه کدنویسی آن نیز بیان شد و امکاناتی بیشتر از آنچه که در جدول مطلب پیشین گفته بودیم در دسترس ما قرار می‌دهد. از این کلاس با اسم untyped dynamically-extendable array به معنی *آرایه پویا قابل توسعه بدون نوع* هم اسم می‌برند چرا که به هیچ نوع داده‌ای مقید نیست و می‌توانید یکبار به آن رشته بدهید، یکبار عدد صحیح، یکبار اعشاری و یکبار زمان و تاریخ، کد زیر به خوبی نشان دهنده‌ی این موضوع است و نحوه استفاده‌ی از این آرایه‌ها را نشان می‌دهد.

```
using System;
using System.Collections;

class ProgrArrayListExample
{
    static void Main()
    {
        ArrayList list = new ArrayList();
        list.Add("Hello");
        list.Add(5);
        list.Add(3.14159);
        list.Add(DateTime.Now);

        for (int i = 0; i < list.Count; i++)
        {
            object value = list[i];
            Console.WriteLine("Index={0}; Value={1}", i, value);
        }
    }
}
```

نتیجه کد بالا:

```
Index=0; Value=Hello
Index=1; Value=5
Index=2; Value=3.14159
Index=3; Value=29.02.2015 23:17:01
```

البته برای خواندن و قرار دادن متغیرها از آنجا که فقط نوع Object را برمی‌گرداند، باید یک تبدیل هم انجام داد یا اینکه از کلمه‌ی کلیدی dynamic استفاده کنید:

```
ArrayList list = new ArrayList();
list.Add(2);
list.Add(3.5f);
list.Add(25u);
list.Add(" ریال");
dynamic sum = 0;
for (int i = 0; i < list.Count; i++)
{
    dynamic value = list[i];
    sum = sum + value;
}
Console.WriteLine("Sum = " + sum);
// Output: Sum = 30.5ریال
```

مجموعه‌های جنریک Generic Collections

مشکل ما در حین کار با کلاس arrayList و همه کلاس‌های مشتق شده از system.collection.IList این است که نوع داده‌ی ما

تبدیل به Object می‌شود و موقعی‌که آن را به ما بر می‌گرداند باید آن را به صورت دستی تبدیل کرده یا از کلمه‌ی کلیدی dynamic استفاده کنیم. در نتیجه در یک شرایط خاص، هیچ تضمینی برای ما وجود نخواهد داشت که بتوانیم کنترلی بر روی نوع داده‌های خود داشته باشیم و به علاوه عمل تبدیل یا casting هم یک عمل زمان بر هست.
برای حل این مشکل، از جنریک‌ها استفاده می‌کنیم. جنریک‌ها می‌توانند با هر نوع داده‌ای کار کنند. در حین تعریف یک کلاس جنریک نوع آن را مشخص می‌کنیم و مقادیری که از آن به بعد خواهد پذیرفت، از نوعی هستند که ابتدا تعریف کرده‌ایم.
یک ساختار جنریک به صورت زیر تعریف می‌شود:

```
GenericType<T> instance = new GenericType<T>();
```

نام کلاس و به جای T نوع داده از قبیل int,bool,string را می‌نویسیم. مثال‌های زیر را ببینید:

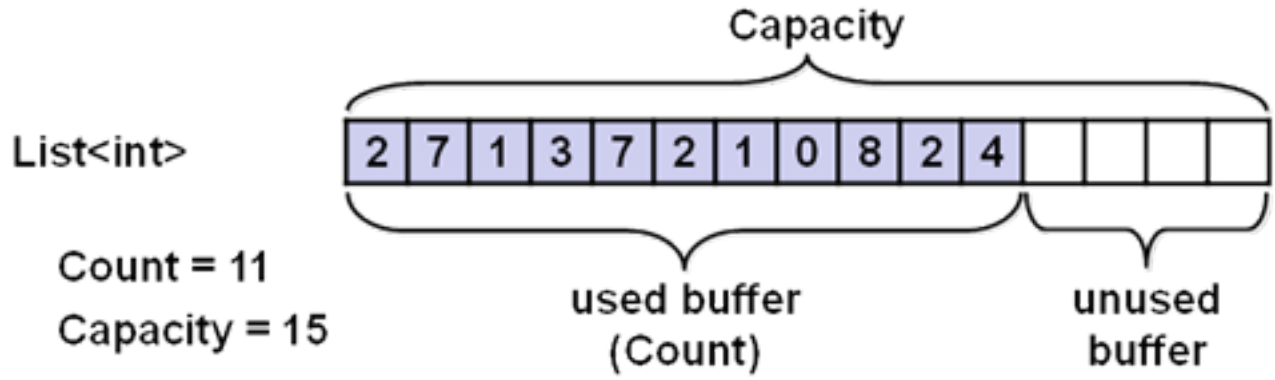
```
List<int> intList = new List<int>();
List<bool> boolList = new List<bool>();
List<double> realNumbersList = new List<double>();
```

کلاس جنریک List<T>

این کلاس مشابه همان کلاس ArrayList است و فقط به صورت جنریک پیاده سازی شده است.

```
List<int> intList = new List<int>();
```

تعریف بالا سبب ایجاد ArrayList ـی می‌باشد که تنها مقادیر int را دریافت می‌کند و دیگر نوع Object ـی در کار نیست. یک آرایه از نوع int ایجاد می‌کند و مقدار خانه‌های پیش فرضی را نیز در ابتدا، برای آن در نظر می‌گیرد و با افزودن هر مقدار جدید می‌بیند که آیا خانه‌ی خالی وجود دارد یا خیر. اگر وجود داشته باشد مقدار جدید، به خانه‌ی بعدی آخرین خانه‌ی پر شده انتقال می‌یابد و اگر هم نباشد، مقدار خانه از آن چه هست 2 برابر می‌شود. درست است عملیات resizing یا افزایش طول آرایه عملی زمان بر محسوب میشود ولی همیشه این اتفاق نمی‌افتد و با زیاد شدن مقادیر خانه‌ها این عمل کمتر هم می‌شود. هر چند با زیاد شدن خانه‌ها حافظه مصرفی ممکن است به خاطر زیاد شدن خانه‌های خالی بدتر هم بشود. فرض کنید بار اول خانه‌ها 16 تایی باشند که بعد می‌شوند 32 تایی و بعدا 64 تایی. حالا فرض کنید به خاطر یک عنصر، خانه‌ها یا ظرفیت بشود 128 تایی در حالی که طول آرایه (خانه‌های پر شده) 65 تاست و حال این وضعیت را برای موارد بزرگتر پیش بینی کنید. در این نوع داده اگر منظور زمان باشد نتجه خوبی را در بر دارد ولی اگر مراعات حافظه را هم در نظر بگیرید و داده‌ها زیاد باشند، باید تا حدامکان به روش‌های دیگر هم فکر کنید.



چه موقع از List<T> استفاده کنیم؟

استفاده از این روش مزایا و معایبی دارد که باید در توضیحات بالا متوجه شده باشید ولی به طور خلاصه:
استفاده از index برای دسترسی به یک مقدار، صرف نظر از اینکه چه میزان داده‌ای در آن وجود دارد، بسیار سریع انجام میگیرد.
جست و جوی یک عنصر بر اساس مقدار: جست و جو خطی است در نتیجه اگر مقدار مورد نظر در آخرین خانه‌ها باشد بدترین وضعیت ممکن رخ می‌دهد و بسیار کند عمل می‌کند. داده هر چی کمتر بهتر و هر چه بیشتر بدتر. البته اگر بخواهید مجموعه‌ای از مقدارهای برابر را برگردانید هم در بدترین وضعیت ممکن خواهد بود.

حذف و درج (منظور insert) المان‌ها به خصوص موقعی که انتهای آرایه نباشید، شیفت پیدا کردن در آرایه عملی کاملا کند و زمانبر است.
موقعی که عنصری را بخواهید اضافه کنید اگر ظرفیت آرایه تکمیل شده باشد، نیاز به عمل زمانبر افزایش ظرفیت خواهد بود که البته این عمل به ندرت رخ می‌دهد و عملیات افزودن Add هم هیچ وابستگی به تعداد المان‌ها ندارد و عملی سریع است.

با توجه به موارد خلاصه شده بالا، موقعی از لیست اضافه می‌کنیم که عملیات درج و حذف زیادی نداریم و بیشتر برای افزودن مقدار به انتها و دسترسی به المان‌ها بر اساس اندیس باشد.

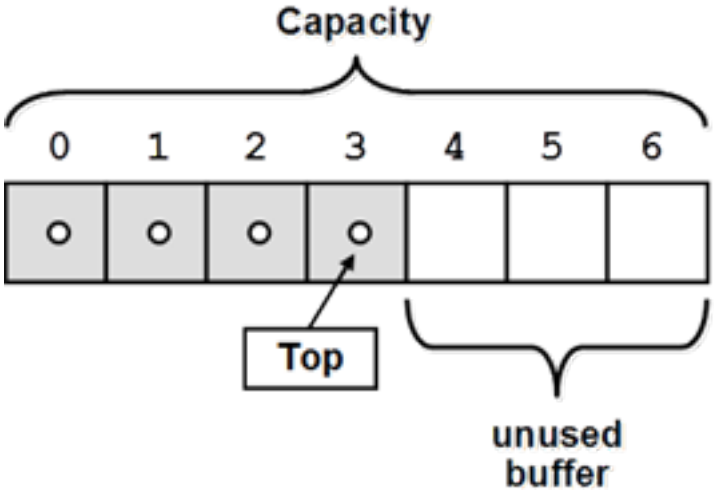
LinkedList<T>

یک کلاس از پیش آماده در دات نت که لیست‌های پیوندی دو طرفه را پیاده سازی می‌کند. هر المان یا گره یک متغیر جهت ذخیره مقدار دارد و یک اشاره گر به گره قبل و بعد. **چه موقع باید از این ساختار استفاده کنیم؟**

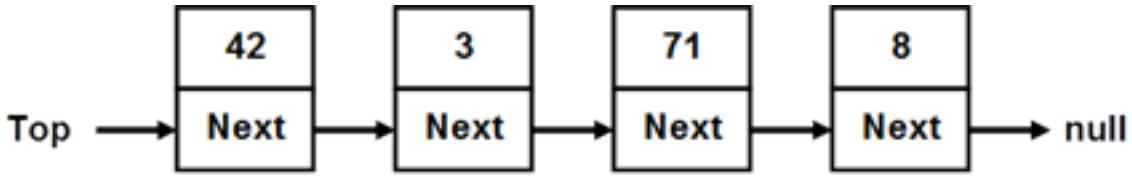
از مزایا و معایب آن :
افزودن به انتهای لیست به خاطر این که همیشه گره آخر در tail وجود دارد بسیار سریع است.
عملیات درج insert در هر موقعیتی که باشد اگر یک اشاره گر به آن محل باشد یک عملیات سریع است یا اینکه درج در ابتدا یاانتهای لیست باشد.
جست و جوی یک مقدار چه بر اساس اندیس باشد و چه مقدار، کار جست و جو کند خواهد بود. چرا که باید تمامی المان‌ها از اول به آخر اسکن بشن.
عملیات حذف هم به خاطر اینکه یک عمل جست و جو در ابتدای خود دارد، یک عمل کند است.

استفاده از این کلاس موقعی خوب است که عملیات‌های درج و حذف ما در یکی از دو طرف لیست باشد یا اشاره‌گری به گره مورد نظر وجود داشته باشد. از لحاظ مصرف حافظه به خاطر داشتن فیلدهای اشاره‌گر به جز مقدار، زیادتر از نوع List می‌باشد. در صورتی که دسترسی سریع به داده‌ها برایتان مهم باشد استفاده از List باز هم به صرفه‌تر است.

پشته Stack
یک سری مکعب را تصور کنید که روی هم قرار گرفته اند و برای اینکه به یکی از مکعب‌های پایینی بخواهید دسترسی داشته باشید باید تعدادی از مکعب‌ها را از بالا بردارید تا به آن برسید. یعنی بر خلاف موقعی که آن‌ها روی هم می‌گذاشتید و آخرین مکعب روی همه قرار گرفته است. حالا همان مکعب‌ها به صورت مخالف و معکوس باید برداشته شوند.
یک مثال واقعی‌تر و ملموس‌تر، یک کمد لباس را تصور کنید که مجبورید برای آن که به لباس خاصی برسید، باید آخرین لباس‌هایی را که در داخل کمد قرار داده‌اید را اول از همه از کمد در بیاورید تا به آن لباس برسید.
در واقع پشته چنین ساختاری را پیاده می‌کند که اولین عنصری که از پشته بیرون می‌آید، آخرین عنصری است که از آن درج شده است و به آن LIFO گویند که مخفف عبارت Last Input First Output آخرین ورودی اولین خروجی است. این ساختار از قدیمی‌ترین ساختارهای موجود است. حتی این ساختار در سیستم‌های داخل دات نت CLR هم به عنوان نگهدارنده متغیرها و پارامتر متدها استفاده می‌شود که به آن Program Execution Stack می‌گویند.
پشته سه عملیات اصلی را پیاده سازی می‌کند: **Push** جهت قرار دادن مقدار جدید در پشته، **POP** جهت بیرون کشیدن مقداری که آخرین بار در پشته اضافه شده و **Peek** جهت برگرداندن آخرین مقدار اضافه شده به پشته ولی آن مقدار از پشته حذف نمی‌شود.
این ساختار میتواند پیاده سازی‌های متفاوتی را داشته باشد ولی دو نوع اصلی که ما بررسی می‌کنیم، ایستا و پویا بودن آن است.
ایستا بر اساس آرایه است و پویا بر اساس لیست‌های پیوندی. شکل زیر پشته‌ای را به صورت استفاده از پیاده‌سازی ایستا با آرایه‌ها نشان می‌دهد و کلمه Top به بالای پشته یعنی آخرین عنصر اضافه شده اشاره می‌کند.



استفاده از لیست پیوندی برای پیاده سازی پشته:

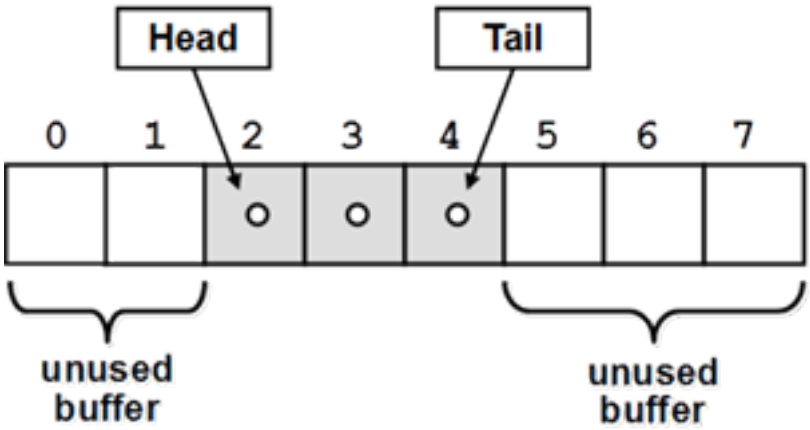


لیست پیوندی لازم نیست دو طرفه باشد و یک طرف برای کار با پشته مناسب است و دیگر لازم نیست که به انتهای لیست پیوندی عمل درج انجام شود؛ بلکه مقدار جدید به ابتدای آن اضافه شده و برای حذف گره هم اولین گره باید حذف شود و گره دوم به عنوان head شناخته می‌شود. همچنین لیست پیوندی نیازی به افزایش ظرفیت مانند آرایه‌ها ندارد.
ساختار پشته در دات نت توسط کلاس Stack از قبل آماده است:

```
Stack<string> stack = new Stack<string>();
stack.Push("A");
stack.Push("B");
stack.Push("C");
 while (stack.Count > 0)
    {
        string letter= stack.Pop();
        Console.WriteLine(letter);
    }
//خروجی
//C
//B
//A
```

صف Queue

ساختار صف هم از قدیمی‌ترین ساختارهاست و مثال آن در همه جا و در همه اطراف ما دیده می‌شود؛ مثل صف نانوایی، صف چاپ پرینتر، دسترسی به منابع مشترک توسط سیستمها. در این ساختار ما عنصر جدید را به انتهای صف اضافه می‌کنیم و برای دریافت مقدار، عنصر را از ابتدا حذف می‌کنیم. به این ساختار FIFO مخفف First Input First Output به معنی اولین ورودی و اولین خروجی هم می‌گویند.
ساختار ایستا که توسط آرایه‌ها پیاده سازی شده است:

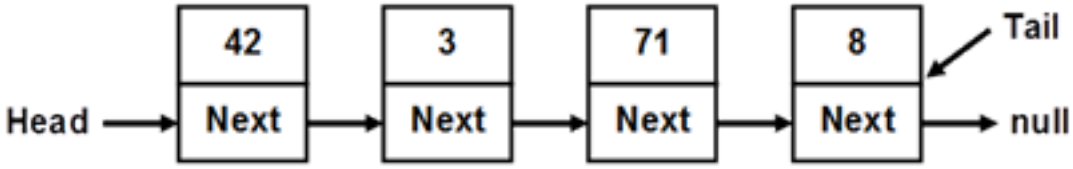


ابتدای آرایه مکانی است که عنصر از آنجا برداشته می‌شود و Head به آن اشاره می‌کند و tail هم به انتهای آرایه که جهت درج عنصر جدید مفید است. با برداشتن هر خانه‌ای که head به آن اشاره می‌کند، head یک خانه به سمت جلو حرکت می‌کند و زمانی که Head از tail بیشتر شود، یعنی اینکه دیگر عنصری یا المانی در صف وجود ندارد و head و Tail به ابتدای صف حرکت می‌کنند. در این حالت موقعی که المان جدیدی قصد اضافه شدن داشته باشد، افزودن، مجددا از اول صف آغاز می‌شود و به این صف‌ها، صف حلقوی می‌گویند.

عملیات اصلی صف دو مورد هستند enqueue که المان جدید را در انتهای صف قرار می‌دهد و dequeue اولین المان صف را بیرون می‌کشد.

پیاده سازی صف به صورت پویا با لیست‌های پیوندی

برای پیاده سازی صف، لیست‌های پیوندی یک طرفه کافی هستند:



در این حالت عنصر جدید مثل سابق به انتهای لیست اضافه می‌شود و برای حذف هم که از اول لیست کمک می‌گیریم و با حذف عنصر اول، متغیر Head به عنصر یا المان دوم اشاره خواهد کرد.

کلاس از پیش آمده صف در دات نت Queue<T> است و نحوه‌ی استفاده آن بدین شکل است:

```
static void Main()
{
    Queue<string> queue = new Queue<string>();
    queue.Enqueue("Message One");
    queue.Enqueue("Message Two");
    queue.Enqueue("Message Three");
    queue.Enqueue("Message Four");

    while (queue.Count > 0)
    {
        string msg = queue.Dequeue();
        Console.WriteLine(msg);
    }
```

```
}
خروجی//
//Message One
//Message Two
//Message Thre
//Message Four
```

عنوان: **درخت‌ها و گراف‌ها قسمت اول**
نویسنده: علی یگانه مقدم
تاریخ: ۱۳۹۳/۱۲/۰۲ ۱:۳۵
آدرس: www.dotnettips.info
گروه‌ها: C#, Data Structures, Tree

در این مقاله یکی از ساختارهای داده را به نام ساختارهای درختی و گراف‌ها معرفی کردیم و در این مقاله قصد داریم این نوع ساختار را بیشتر بررسی نماییم. این ساختارها برای بسیاری از برنامه‌های مدرن و امروزی بسیار مهم هستند. هر کدام از این ساختارهای داده به حل یکی از مشکلات دنیای واقعی می‌پردازند. در این مقاله قصد داریم به مزایا و معایب هر کدام از این ساختارها اشاره کنیم و اینکه کی و کجا بهتر است از کدام ساختار استفاده گردد. تمرکز ما بر **درخت هایی دودویی** ، **درخت‌های جست و جوی دو دویی** و **درخت‌های جست و جوی دو دویی متوازن** خواهد بود. همچنین ما به تشریح گراف و انواع آن خواهیم پرداخت. اینکه چگونه آن را در حافظه نمایش دهیم و اینکه گراف‌ها در کجای زندگی واقعی ما یا فناوری‌های کامپیوتری استفاده می‌شوند.

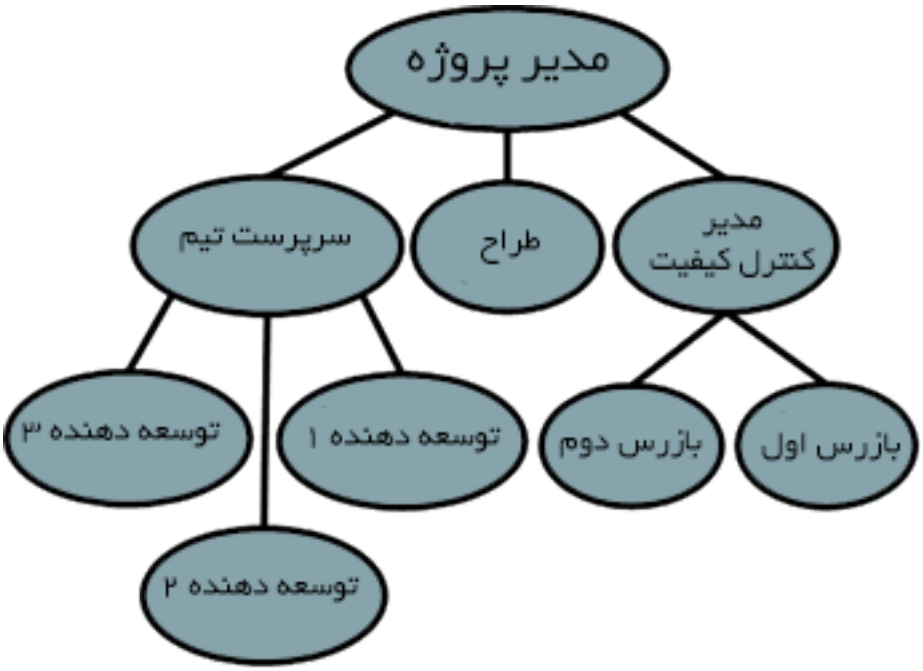
ساختار درختی

در بسیاری از مواقع ما با گروهی از اشیاء یا داده‌هایی سر و کار داریم که هر کدام از آن‌ها به گروهی دیگر مرتبط هستند. در این حالت از ساختار خطی نمی‌توانیم برای توصیف این ارتباط استفاده کنیم. پس بهترین ساختار برای نشان دادن این ارتباط **ساختار شاخه ای Branched Structure** است.

یک ساختار درختی یا یک ساختار شاخه‌ای شامل المان‌هایی به اسم گره Node است. هر گره می‌تواند به یک یا چند گره دیگر متصل باشد و گاهی اوقات این اتصالات مشابه یک *سلسه مراتب hierarchically* می‌شوند.

درخت‌ها در برنامه نویسی جایگاه ویژه‌ای دارند به طوری که استفاده‌ی از آن‌ها در بسیاری از برنامه‌ها وجود دارد و بسیاری از مثال‌های واقعی پیرامون ما را پشتیبانی می‌کنند.

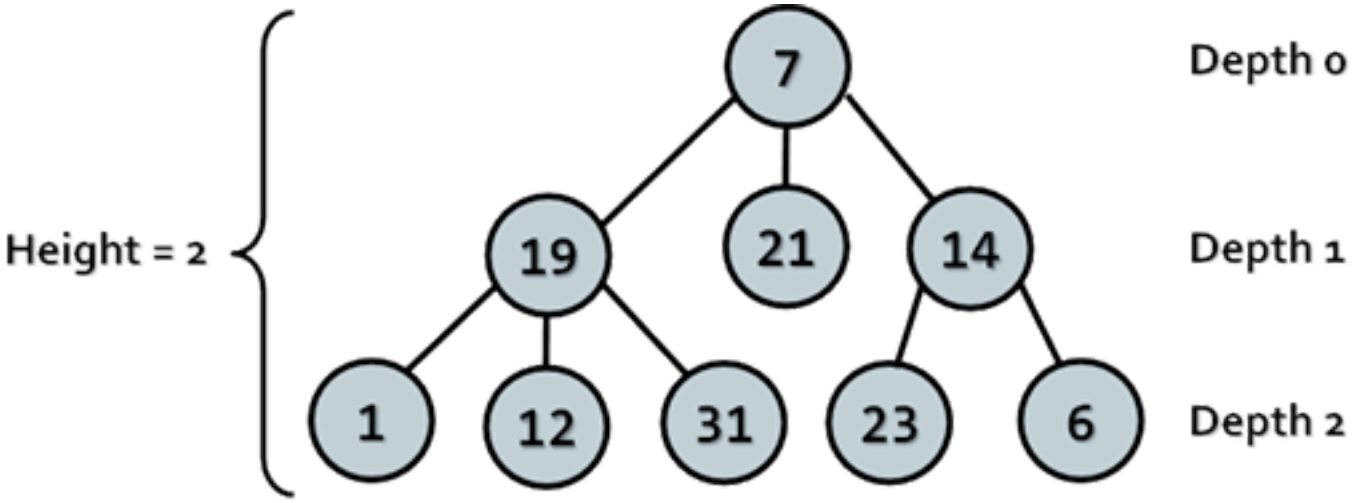
در نمودار زیر مثالی وجود دارد که در آن یک تیم نرم افزاری نمایش داده شده‌است. در اینجا هر یک از بخش‌ها وظایف و مسئولیت‌هایی را بر دوش خود دارند که این مسئولیت‌ها به صورت سلسله مراتبی در تصویر زیر نمایش داده شده‌اند.



ما در ساختار بالا متوجه می‌شویم که چه بخشی زیر مجموعه‌ی چه بخشی است و سمت بالاتر هر بخش چیست. برای مثال ما متوجه شدیم که مدیر توسعه دهندگان، "سرپرست تیم" است که خود نیز مادون "مدیر پروژه" است و این را نیز متوجه می‌شویم که مثلا توسعه دهنده‌ی شماره یک هیچ مادونی ندارد و مدیر پروژه در راس همه است و هیچ مدیر دیگری بالای سر او قرار ندارد.

اصطلاحات درخت

برای اینکه بیشتر متوجه روابط بین اشیا در این ساختار بشویم، به شکل زیر خوب دقت کنید:



در شکل بالا دایره‌هایی برای هر بخش از اطلاعت کشیده شده و ارتباط هر کدام از آن‌ها از طریق یک خط برقرار شده است. اعداد داخل هر دایره تکراری نیست و همه منحصر به فرد هستند. پس وقتی از اعداد اسم ببریم متوجه می‌شویم که در مورد چه چیزی صحبت می‌کنیم.

در شکل بالا به هر یک از دایره‌ها یک **گره Node** می‌گویند و به هر خط ارتباط دهنده بین گره‌ها **لبه Edge** گفته می‌شود. گره‌های 19 و 21 و 14 زیر گره‌های گره 7 محسوب می‌شوند. گره‌هایی که به صورت مستقیم به زیر گره‌های خودشان اشاره می‌کنند را **گره‌های والد Parent** می‌گویند و زیرگره‌های 7 را گره‌های **فرزند ChildNodes** . پس با این حساب می‌توانیم بگوییم گره‌های 1 و 12 و 31 را هم فرزند گره 19 هستند و گره 19 والد آن هاست. همچنین گره‌های یک والد را مثل 19 و 21 و 14 که والد مشترک دارند، گره‌های **خواهر و برادر** یا **حتی همنژاد Sibling** می‌گوییم. همچنین ارتباط بین گره 7 و گره‌های سطح دوم و الی آخر یعنی 1 و 12 و 31 و 23 و 6 را که والد بودن آن به صورت غیر مستقیم است را **جد** یا **ancestor** می‌نامیم و نوه‌ها و نتیجه‌های آن‌ها را **نسل descendants** .

ریشه Root : به گره‌ای می‌گوییم که هیچ والدی ندارد و خودش در واقع اولین والد محسوب می‌شود؛ مثل گره 7.

برگ Leaf : به گره‌هایی که هیچ فرزندی ندارند، برگ می‌گوییم. مثال گره‌های 1 و12 و 31 و 23 و 6

گره‌های داخلی Internal Nodes : گره هایی که نه برگ هستند و نه ریشه. یعنی حداقل یک فرزند دارند و خودشان یک گره فرزند محسوب می‌شوند؛ مثل گره‌های 19 و 14.

مسیر Path : راه رسیدن از یک گره به گره دیگر را مسیر می‌گویند. مثلا گره‌های 1 و 19 و 7 و 21 به ترتیب یک مسیر را تشکیل می‌دهند ولی گره‌های 1 و 19 و 23 از آن جا که هیچ جور اتصالی بین آن‌ها نیست، مسیری را تشکیل نمی‌دهند.

طول مسیر Length of Path : به تعداد لبه‌های یک مسیر، طول مسیر می‌گویند که می‌توان از تعداد گره‌ها -1 نیز آن را به دست آورد. برای نمونه : مسیر 1 و19 و 7 و 21 طول مسیرشان 3 هست.

عمق Depth : طول مسیر یک گره از ریشه تا آن گره را عمق درخت می‌گویند. عمق یک ریشه همیشه صفر است و برای مثال در درخت بالا، گره 19 در عمق یک است و برای گره 23 عمق آن 2 خواهد بود.

تعریف خود درخت Tree : درخت یک ساختار داده برگشتی recursive است که شامل گره‌ها و لبه‌ها، برای اتصال گره‌ها به

یکدیگر است.

جملات زیر در مورد درخت صدق می‌کند:

هر گره می‌تواند فرزند نداشته باشد یا به هر تعداد که می‌خواهد فرزند داشته باشد.
هر گره یک والد دارد و تنها گره‌ای که والد ندارد، گره ریشه است (البته اگر درخت خالی باشد هیچ گره ای وجود ندارد).
همه گره‌ها از ریشه قابل دسترسی هستند و برای دسترسی به گره مورد نظر باید از ریشه تا آن گره، مسیری را طی کرد.
ار تفاع درخت Height: به حداکثر عمق یک درخت، ارتفاع درخت می‌گویند. **درجه گره Degree** : به تعداد گره‌های فرزند یک گره، درجه آن گره می‌گویند. در درخت بالا درجه گره‌های 7 و 19 سه است. درجه گره 14 دو است و درجه برگ‌ها صفر است. **ضریب انشعاب Branching Factor:** به حداکثر درجه یک گره در یک درخت، ضریب انشعاب آن درخت گویند.

پیاده سازی درخت

برای پیاده سازی یک درخت، از دو کلاس یکی جهت ساخت گره که حاوی اطلاعات است TreeNode<T> و دیگری جهت ایجاد درخت اصلی به همراه کلیه متدها و خاصیت هایش Tree<T> کمک می‌گیریم.

```
public class TreeNode<T>
{
    // شامل مقدار گره است
    private T value;

    // مشخص می‌کند که آیا گره والد دارد یا خیر
    private bool hasParent;

    // در صورت داشتن فرزند ، لیست فرزندان را شامل می‌شود
    private List<TreeNode<T>> children;

    /// <summary>سازنده کلاس </summary>
    /// <param name="value">مقدار گره</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>خاصیتی جهت مقداردهی گره</summary>
    public T Value
    {
        get
        {
            return this.value;
        }
        set
        {
            this.value = value;
        }
    }

    /// <summary>تعداد گره‌های فرزند را بر میگرداند</summary>
    public int ChildrenCount
    {
        get
        {
            return this.children.Count;
        }
    }

    /// <summary>به گره یک فرزند اضافه می‌کند</summary>
    /// <param name="child">آرگومان این متد یک گره است که قرار است به فرزندی گره فعلی در آید</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>
    /// گره ای که اندیس آن داده شده است بازگردانده می‌شود
    /// </summary>
    /// <param name="index">اندیس گره</param>
    /// <returns>گره بازگشتی</returns>
    public TreeNode<T> GetChild(int index)
    {
        return this.children[index];
    }
}

/// <summary>این کلاس ساختار درخت را به کمک کلاس گره‌ها که در بالا تعریف کردیم میسازد</summary>
/// <typeparam name="T">نوع مقادیری که قرار است داخل درخت ذخیره شوند</typeparam>
public class Tree<T>
{
    // گره ریشه
    private TreeNode<T> root;

    /// <summary>سازنده کلاس</summary>
    /// <param name="value">مقدار گره اول که همان ریشه می‌شود</param>
    public Tree(T value)
    {
        if (value == null)
        {
            throw new ArgumentNullException(
                "Cannot insert null value!");
        }

        this.root = new TreeNode<T>(value);
    }

    /// <summary>سازنده دیگر برای کلاس درخت</summary>
    /// <param name="value">مقدار گره ریشه مثل سازنده اول</param>
    /// <param name="children">آرایه ای از گره‌ها که فرزند گره ریشه می‌شوند</param>
    public Tree(T value, params Tree<T>[] children)
        : this(value)
    {
        foreach (Tree<T> child in children)
        {
            this.root.AddChild(child.root);
        }
    }

    /// <summary>
    /// ریشه را بر میگرداند ، اگر ریشه ای نباشد نال بر میگرداند
    /// </summary>
    public TreeNode<T> Root
    {
        get
        {
            return this.root;
        }
    }

    /// <summary>پیمودن عرضی و نمایش درخت با الگوریتم دی اف اس </summary>
    /// <param name="root">ریشه (گره ابتدایی) درختی که قرار است پیمایش از آن شروع شود</param>
    /// <param name="spaces">یک کاراکتر جهت جداسازی مقادیر هر گره</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>متد پیمایش درخت به صورت عمومی که تابع خصوصی که در بالا توضیح دادیم را صدا
می‌زند</summary>
    public void TraverseDFS()
    {
        this.PrintDFS(this.root, string.Empty);
    }
}

/// <summary>
/// کد استفاده از ساختار درخت
/// </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))
            );

        // پیمایش درخت با الگوریتم دی اف اس یا عمقی
        tree.TraverseDFS();

        // خروجی
        // 7
        //       19
        //        1
        //        12
        //        31
        //       21
        //       14
        //        23
        //        6
    }
}
```

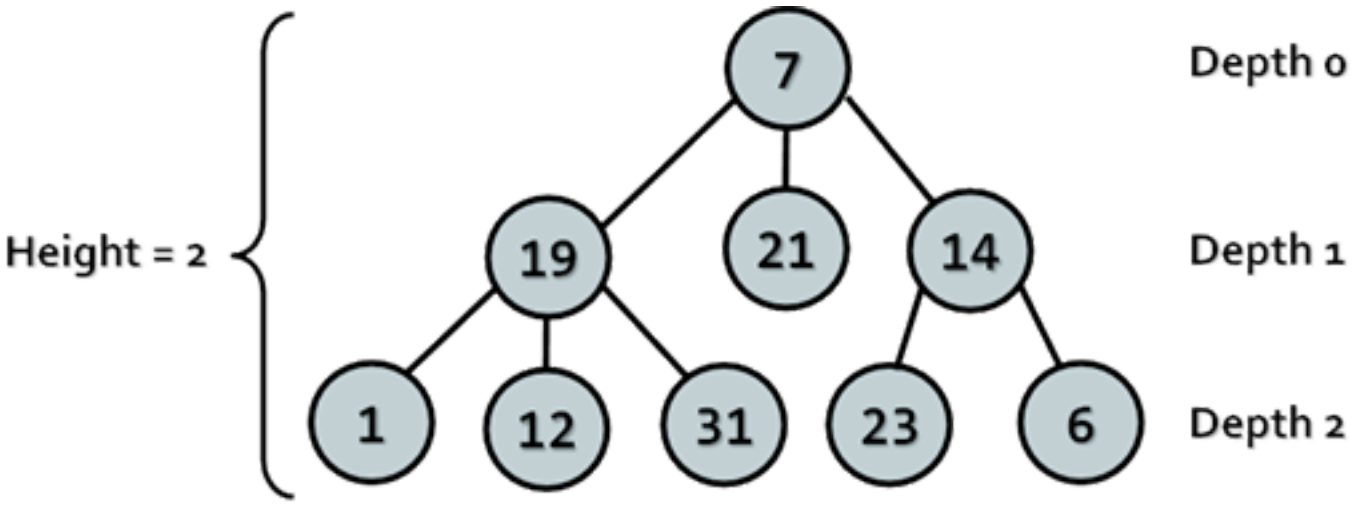
کلاس TreeNode وظیفه‌ی ساخت گره را بر عهده دارد و با هر شیءایی که از این کلاس می‌سازیم، یک گره ایجاد می‌کنیم که با خاصیت Children و متد AddChild آن می‌توانیم هر تعداد گره را که می‌خواهیم به فرزندی آن گره در آوریم که باز خود آن گره می‌تواند در خاصیت Children یک گره دیگر اضافه شود. به این ترتیب با ساخت هر گره و ایجاد رابطه از طریق خاصیت children هر گره درخت شکل می‌گیرد. سپس گره والد در ساختار کلاس درخت Tree قرار می‌گیرد و این کلاس شامل متدهایی است که می‌تواند روی درخت، عملیات پردازشی چون پیمایش درخت را انجام دهد.

پیمایش درخت به روش عمقی DFS (Depth First Search)

هدف از پیمایش درخت ملاقات یا بازبینی (تهیه لیستی از همه گره‌های یک درخت) تنها یکبار هر گره در درخت است. برای این کار الگوریتم‌های زیادی وجود دارند که ما در این مقاله تنها دو روش DFS و BFS را بررسی می‌کنیم.

روش DFS: هر گره‌ای که به تابع بالا بدهید، آن گره برای پیمایش، گره ریشه حساب خواهد شد و پیمایش از آن آغاز می‌گردد. در الگوریتم DFS روش پیمایش بدین گونه است که ما از گره ریشه آغاز کرده و گره ریشه را ملاقات می‌کنیم. سپس گره‌های فرزندش را به دست می‌آوریم و یکی از گره‌ها را انتخاب کرده و دوباره همین مورد را رویش انجام می‌دهیم تا نهایتا به یک برگ برسیم. وقتی که به برگی می‌رسیم یک مرحله به بالا برگشته و این کار را آنقدر تکرار می‌کنیم تا همه‌ی گره‌های آن ریشه یا درخت پیمایش شده باشند.

همین درخت را در نظر بگیرید:



پیمایش درخت را از گره 7 آغاز می‌کنیم و آن را به عنوان ریشه در نظر می‌گیریم. حتی می‌توانیم پیمایش را از گره مثلا 19 آغاز کنیم و آن را برای پیمایش ریشه در نظر بگیریم ولی ما از همان 7 پیمایش را آغاز می‌کنیم:

ابتدا گره 7 ملاقات شده و آن را می‌نویسیم. سپس فرزندانش را بررسی می‌کنیم که سه فرزند دارد. یکی از فرزندان مثل گره 19 را انتخاب کرده و آن را ملاقات می‌کنیم (با هر بار ملاقات آن را چاپ می‌کنیم) سپس فرزندان آن را بررسی می‌کنیم و یکی از گره‌ها را انتخاب می‌کنیم و ملاقاتش می‌کنیم؛ برای مثال گره 1. از آن جا که گره یک، برگ است و فرزندی ندارد یک مرحله به سمت بالا برمی‌گردیم و برگ‌های 12 و 31 را هم ملاقات می‌کنیم. حالا همه‌ی فرزندان گره 19 را بررسی کردیم، بر می‌گردیم یک مرحله به سمت بالا و گره 21 را ملاقات می‌کنیم و از آنجا که گره 21 برگ است و فرزندی ندارد به بالا باز می‌گردیم و بعد گره 14 و فرزندانش 23 و 6 هم بررسی می‌شوند. پس ترتیب چاپ ما اینگونه می‌شود:

```
7-19-1-12-31-21-14-23-6
```

پیمایش درخت به روش BFS (Breadth First Search)

در این روش (پیمایش سطحی) گره والد ملاقات شده و سپس همه گره‌های فرزندش ملاقات می‌شوند. بعد از آن یک گره انتخاب شده و همین پیمایش مجددا روی آن انجام می‌شود تا آن سطح کاملا پیمایش شده باشد. سپس به همین مرحله برگشته و فرزند بعدی را پیمایش می‌کنیم و الی آخر. نمونه‌ی این پیمایش روی درخت بالا به صورت زیر نمایش داده می‌شود:

```
7-19-21-14-1-12-31-23-6
```

اگر خوب دقت کنید می‌بینید که پیمایش سطحی است و هر سطح به ترتیب ملاقات می‌شود. به این الگوریتم، پیمایش موجی هم می‌گویند. دلیل آن هم این است که مثل سنگی می‌ماند که شما برای ایجاد موج روی دریاچه پرتاب می‌کنید.

برای این پیمایش از صف کمک گرفته می‌شود که مراحل زیر روی صف صورت می‌گیرد:

ریشه وارد صف Q می‌شود.
دو مرحله زیر مرتبا تکرار می‌شوند:

اولین گره صف به نام V را از Q در یافت می‌کنیم و آن را چاپ می‌کنیم.
فرزندان گره V را به صف اضافه می‌کنیم.

این نوع پیمایش، پیاده سازی راحتی دارد و همیشه نزدیک‌ترین گره‌ها به ریشه را می‌خواند و در هر مرحله گره‌هایی که می‌خواند از ریشه دورتر و دورتر می‌شوند.

نظرات خوانندگان

نویسنده: آقا ابراهیم
تاریخ: ۱۳۹۳/۱۲/۰۲ ۲۲:۴۶

سلام. ممنون بابت مطلب تون. کلا چند کاربرد سنگین و روز مره این ساختارهای درختی رو میخواستم.

نویسنده: محسن خان
تاریخ: ۱۳۹۳/۱۲/۰۲ ۲۳:۳۱

همین نظر تو در تویی که الان ذیل مطلب شما ارسال شد یک ساختار درختی هست.

نویسنده: علی یگانه مقدم
تاریخ: ۱۳۹۳/۱۲/۰۳ ۰:۵۶

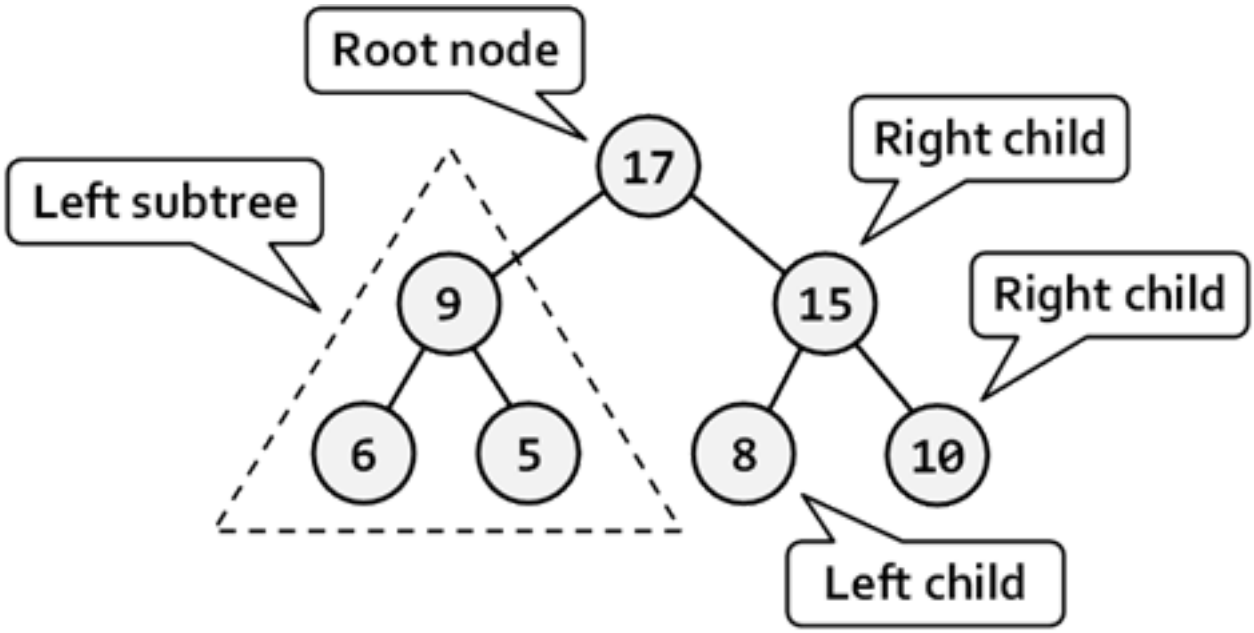
کاربرد این موارد زیاد هست و در قسمت‌های بعدی مواردی رو هم نام خواهیم برد
نمونه‌های این مثال مثل دیکشنری‌ها و جست و جوها ، نقشه‌های شهر و مسیریابی و بازی ها
سیستم‌های برق کشی و لوله کشی و .. در بعضی کشورها روی سیستم‌ها نظارت میشه و با ایجاد یک نقص فنی روی نقشه به اونها نشون میده
یا حتی سیستم فایل یا سیستم‌های جست و جو گر
همین موتور گوگل یا حتی موتورهای جدید که با روش خاص از گراف برای جست و جوی‌های مرتبط استفاده میکنن و داده‌ها مرتبط به هم متصل میشن رو میشه نمونه از این موارد دونست
در خیلی از موارد هم شما دارین ازشون استفاده می‌کنین ولی شاید به خاطر قابلیت‌های فریم ورک‌های جدید و پیشرفت زبان‌ها چنان محسوس نبودن

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.

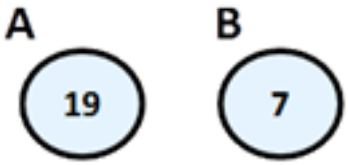
برای باقی روش‌های پیمایش تنها نیاز است که این سه خط را جابجا کنید:

```
// 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();
        }
```

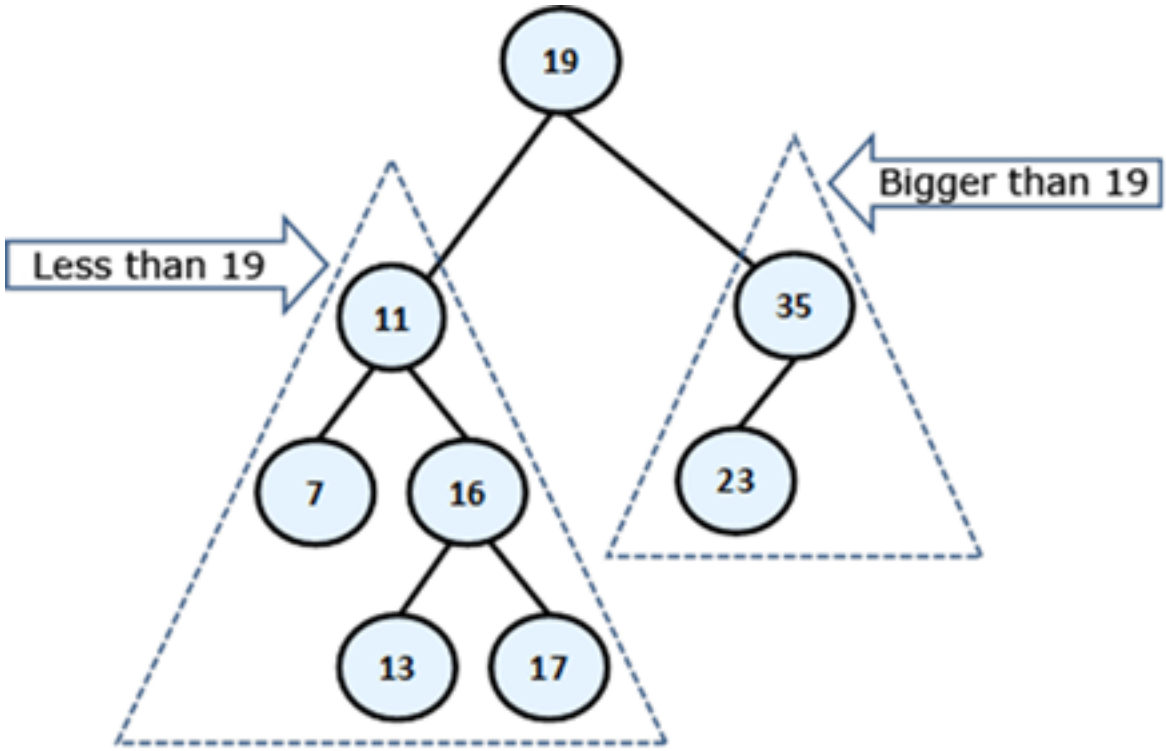
درخت دودویی مرتب شده Ordered Binary Search Tree

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.

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.

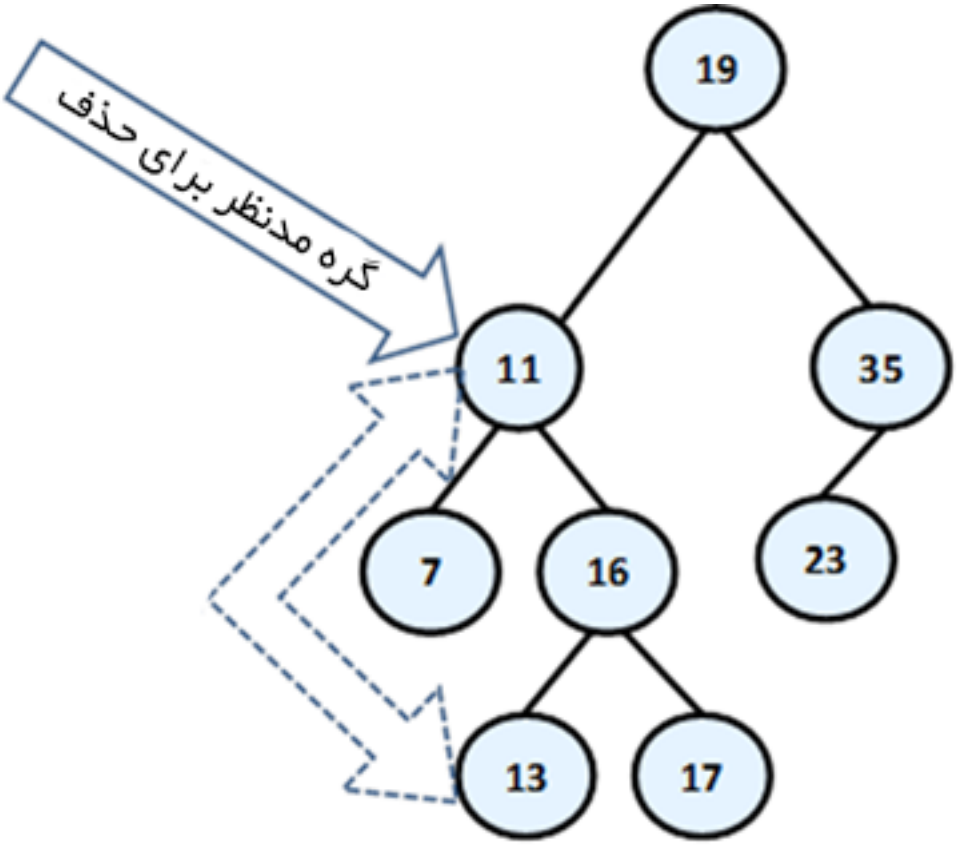
در انتها اگر الگوریتم، گره را پیدا کند، گره پیدا شده را باز می‌گرداند؛ ولی اگر گره را پیدا نکند، یا درخت خالی باشد، مقدار برگشتی نال خواهد بود.

حذف یک عنصر

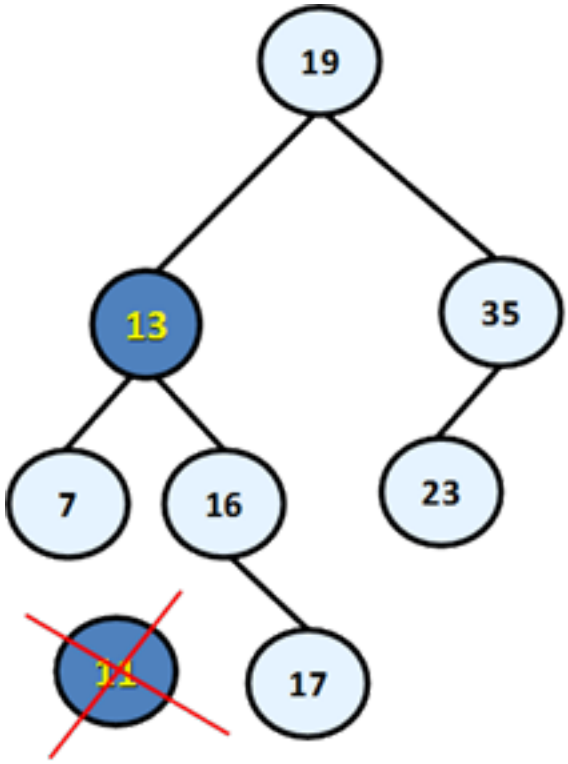
حذف کردن در این درخت نسبت به درخت دودویی معمولی پیچیده‌تر است. اولین گام این عمل، جستجوی گره مدنظر است. وقتی گره‌ایی را مدنظر داشته باشیم، سه بررسی زیر انجام می‌گیرد:

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);
    }
}
```

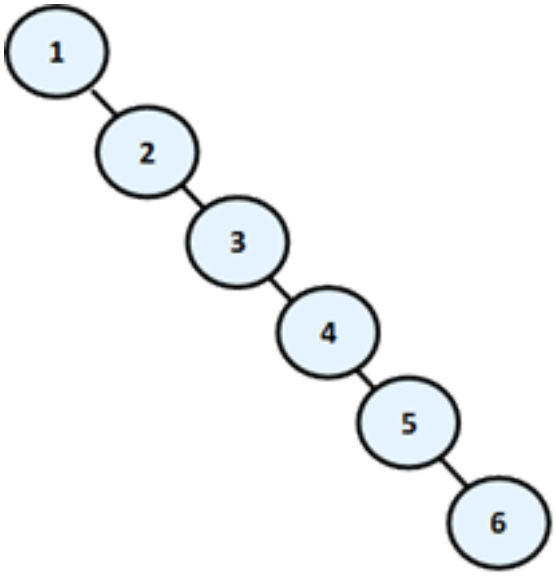
در مقاله بعدی درخت دودویی متوازن را که پیچیده‌تر از این درخت است و از کارآیی بهتری برخوردار هست، بررسی می‌کنیم.

عنوان: **درخت‌ها و گراف قسمت چهارم**
نویسنده: علی یگانه مقدم
تاریخ: ۱۳۹۳/۱۲/۰۶ ۱:۰
آدرس: www.dotnettips.info
گروه‌ها: C#, Data Structures, Tree

در قسمت قبلی مبحث پیاده سازی ساختمان (ساختار) درخت‌های جستجوی دودویی را به پایان رساندیم. در این قسمت قرار است بر روی درخت متوازن بحث کنیم و آن را پیاده سازی نماییم.

درخت متوازن

همانطور که دیدید، عملیات جستجو روی درخت جستجوی دو دویی به مراتب راحت و آسان‌تر است؛ ولی با این حال این درخت در عملیاتی چون درج و حذف، یک نقص فنی دارد و آن هم این است که نمی‌تواند عمق خود را کنترل کند و همینطور به سمت عمق‌های بیشتر و بزرگتر حرکت می‌کند. مثلا ساختار ترتیبی زیر را برای مقداری‌های 1 و 2 و 3 و 4 و 5و 6 در نظر بگیرید:



در این حالت دیگر درخت مانند یک درخت رفتار نمی‌کند و بیشتر شبیه لیست پیوندی است و عملیات جستجو همینطور کندتر و کندتر می‌شود و دیگر مثل سابق نخواهد بود، پیچیدگی برنامه بیشتر خواهد شد و از Log(n) به n می‌رسد. از آنجا که دوست داریم برای عملیات‌های رایجی چون درج و جستجو و حذف، همین پیچیدگی لگاریتمی را حفظ کنیم، از ساختاری جدیدتر بهره خواهیم برد.

درخت دودویی متوازن: درختی است که در آن هیچ برگی، عمقش از هیچ برگی بیشتر نیست.

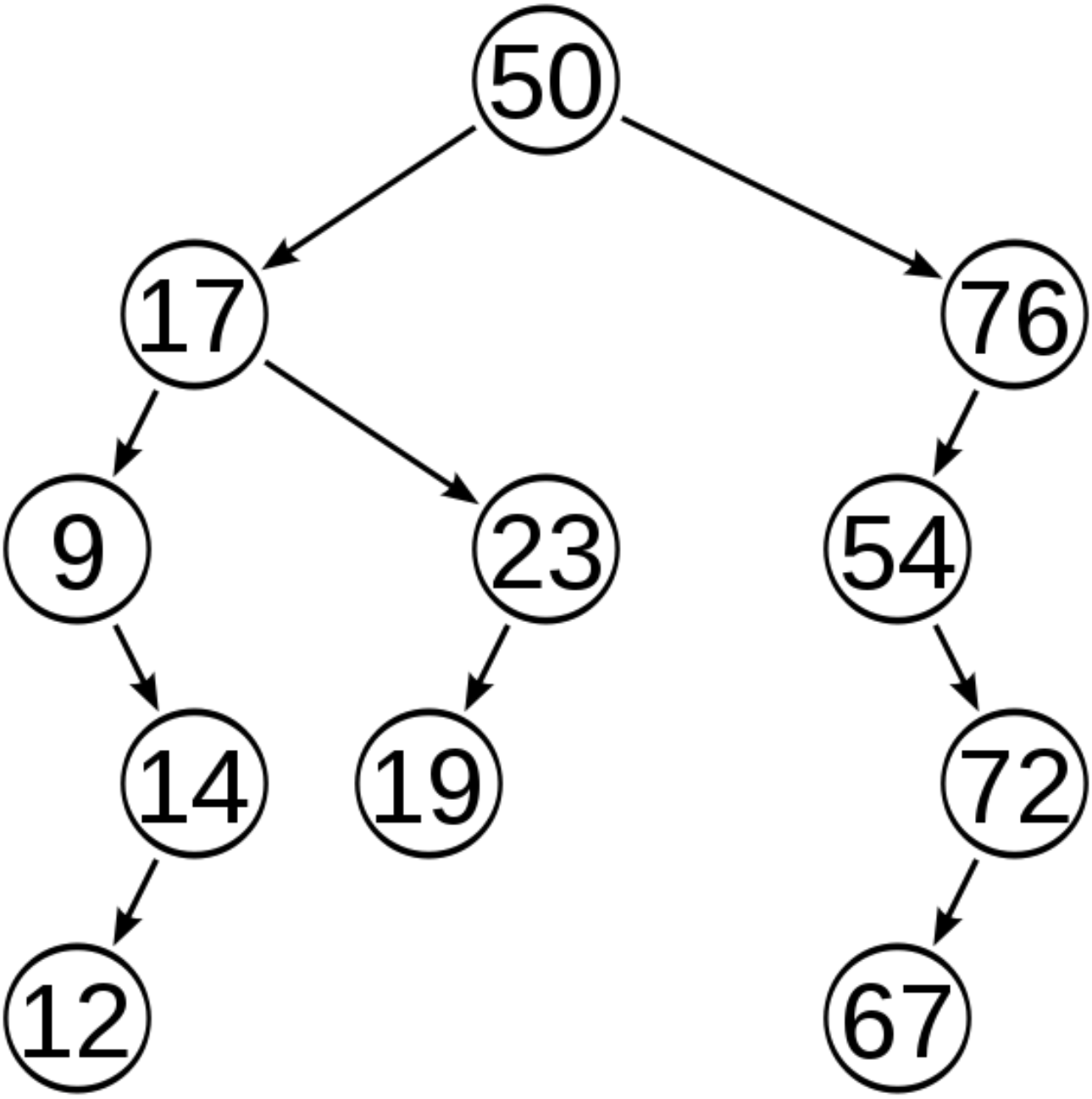
درخت دودویی متوازن کامل: درختی که تفاوتش در تعداد گره‌های چپ یا راست است و حداقل یک فرزند دارند.

درخت دودویی متوازن حتی اگر کامل هم نباشد، در عملیات پایه‌ای چون افزودن، حذف و جستجو در بدترین حالت هم با پیچیدگی لگاریتمی تعداد گام‌ها همراه است. برای اینکه این درخت با به هم ریختگی توازنش روبرو نشود، باید حین انجام عملیات پایه، جایگاه تعداد المان‌های آن بررسی و اصلاح شود که به این عملیات چرخش یا دوران Rotation می‌گویند. انجام این عملیات بستگی دارد که پیاده سازی این درخت به چه شکلی باشد و به چه صورتی پیاده سازی شده باشد. از پیاده سازی‌های این درخت می‌توان به درخت سرخ-سیاه Black Red Tree ، ای وی ال AVL Tree ، اسپلی Splay و ... اشاره کرد.

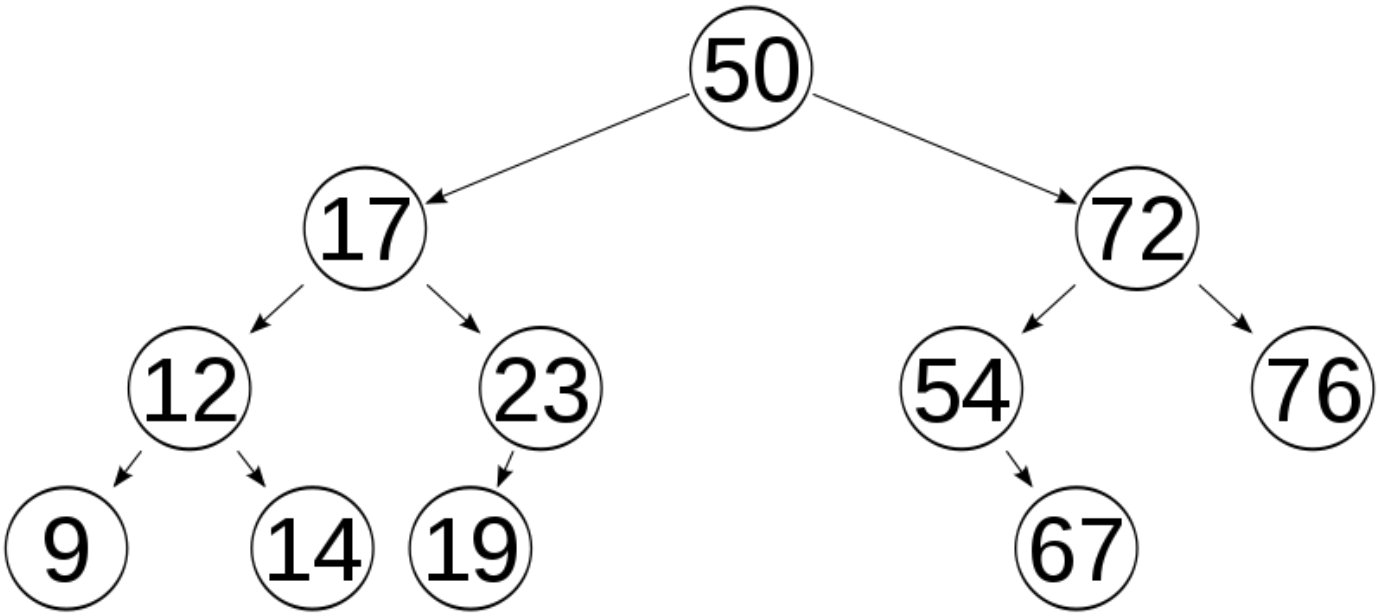
با توجه به موارد بالا میتوانیم به نتایج زیر برسیم که چرا این درخت در هر حالت، پیچیدگی زمانی خودش را در لگاریتم n حفظ

می‌کند:
المان‌ها و عناصرش را مرتب شده نگه می‌دارد.
خودش را متوازن نگه داشته و اجازه نمی‌دهد عمقش بیشتر از لگاریتم n شود.

نمونه ای از درخت جستجوی دو دویی



همان درخت ولی به صورت متوازن با پیاده سازی AVL



درخت‌های متوازن هم می‌متوانند **دو دویی یا باینری** باشند و هم **غیر باینری non-Binary**

درخت‌های دو دویی در انجام عملیات بسیار سریع هستند و در بالا به تعدادی از انواع پیاده سازی‌های آن اشاره کردیم.

درخت‌های غیر باینری نیز مرتب و متوازن بوده، ولی می‌توانند بیش از یک کلید داشته باشند و همچنین بیشتر از دو فرزند. این درخت‌ها نیز عملیات خود را بسیار سریع انجام می‌دهند. از نمونه‌های این درخت می‌توان به `B Tree,B+ Tree,Interval Tree` اشاره کرد.

از آنجا که پیاده‌سازی این نوع درخت کمی دشوار و پیچیده و طولانی است و همچنین پیاده سازی‌های مختلفی دارد؛ تعدادی از منابع موجود را در زیر معرفی می‌کنیم:

در خود دات نت در فضای نام `system.collection.generic` کلاس `TreeSet` یک نوع پیاده سازی از این درخت است که این پیاده سازی از نوع درخت سرخ سیاه می‌باشد و جالب است بدانید، در طی بیست گام می‌تواند در یک میلیون آیتم به جستجو بپردازد ولی خبر بد اینکه استفاده مستقیم از این کلاس ممکن نیست چرا که این کلاس به صورت داخلی `internal` برای استفاده‌ی خود کتابخانه طراحی شده است ولی خبر خوب اینکه کلاس `sortedDictionary` از این کلاس بهره برده است و به صورت عمومی در دسترس ما قرار گرفته است. همچنین کلاس `SortedSet` هم از دات نت 4 نیز در دسترس است.

کتابخانه‌های خارجی جهت استفاده در دات نت که به پیاده‌سازی درخت‌های متوازن پرداخته‌اند:

پیاده سازی Splay Tree با سی شارپ

پیاده سازی AVL به صورت بهینه و آسان برای استفاده

پیاده سازی درخت AVL با کارایی بالا

پیاده سازی درخت AVL به همراه نمایش گرافیکی آن

درخت سرخ-سیاه

کتابخانه ای متن باز برای درخت سرخ-سیاه

درخت متوازن به همراه جست و جو و حذف و پیمایش ها

غیر دودویی‌ها

پیاده سازی B Tree
پیاده سازی Interval Tree
پیاده سازی Interval Tree در سی ++