

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، موقعی که 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 کارایی 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 - ADT - Abstraction Data Type

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 فروشگاه‌ای 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:

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

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

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. دقیقاً 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;
}

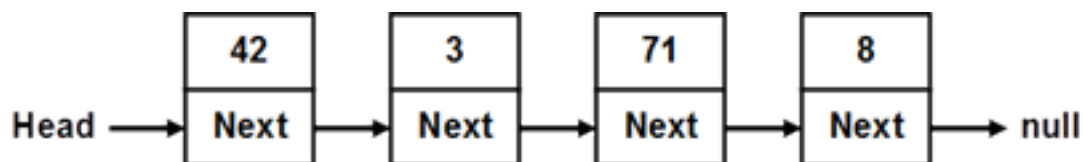
```

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 `Head`، `Tail`، `Count` به ترتیب برای اشاره به اولین گره، آخرین گره و تعداد گره‌ها، به کار می‌روند که در ادامه کد آن را 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;
    }

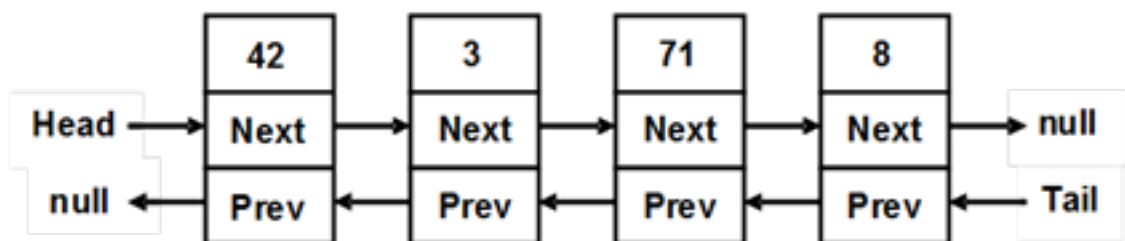
    if (object.ReferenceEquals(this.tail, node))
    {
        this.tail = prevNode;
    }
}
```

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 ذخیره داده‌ها در حافظه روش‌های متفاوتی وجود دارند که بیشتر تفاوت آن در مورد استفاده از حافظه و کارایی 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.