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;
    }
```

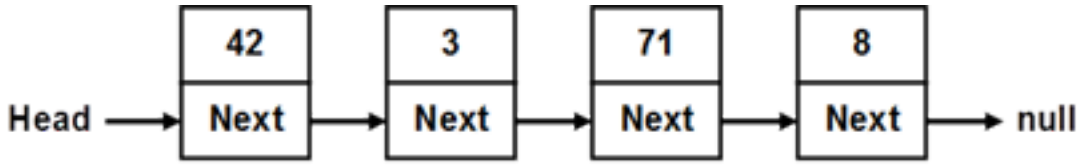
```
}

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;
}
```

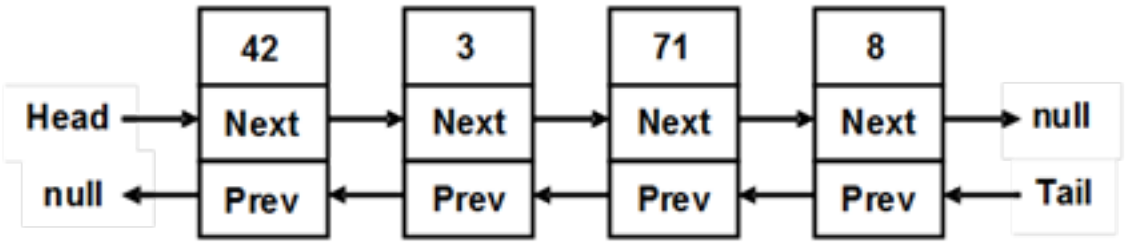
```
private void RemoveListNode(ListNode node, ListNode prevNode)
{
    count--;
    if (count == 0)
    {
        this.head = null;
        this.tail = null;
    }
    else if (prevNode == null)
    {
        this.head = node.NextNode;
    }
    else
    {
        prevNode.NextNode = node.NextNode;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

    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:

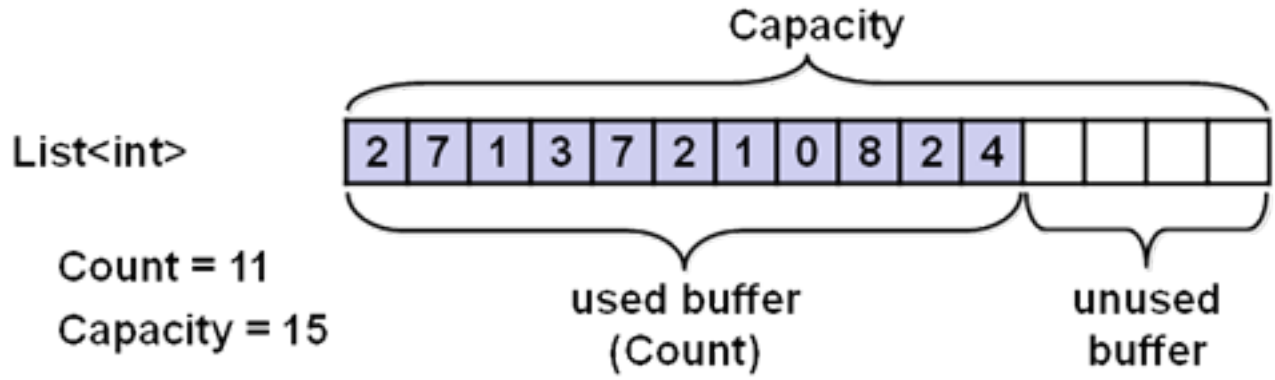
```
List<int> intList = new List<int>();
List<bool> boolList = new List<bool>();
List<double> realNumbersList = new List<double>();
```

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.

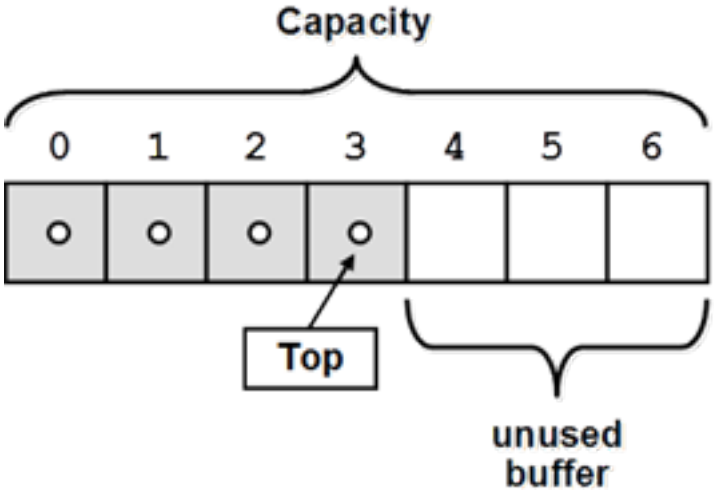
LinkedList<T>

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؟**

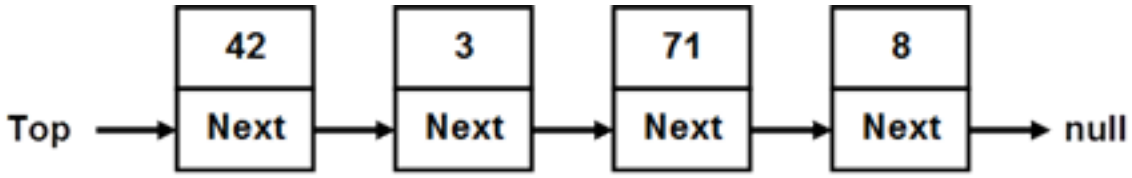
از مزایا و معایب آن :
افزودن به انتهای لیست به خاطر این که همیشه گره آخر در tail وجود دارد بسیار سریع است.
عملیات درج insert در هر موقعیتی که باشد اگر یک اشاره گر به آن محل باشد یک عملیات سریع است یا اینکه درج در ابتدا یاانتهای لیست باشد.
جست و جوی یک مقدار چه بر اساس اندیس باشد و چه مقدار، کار جست و 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:

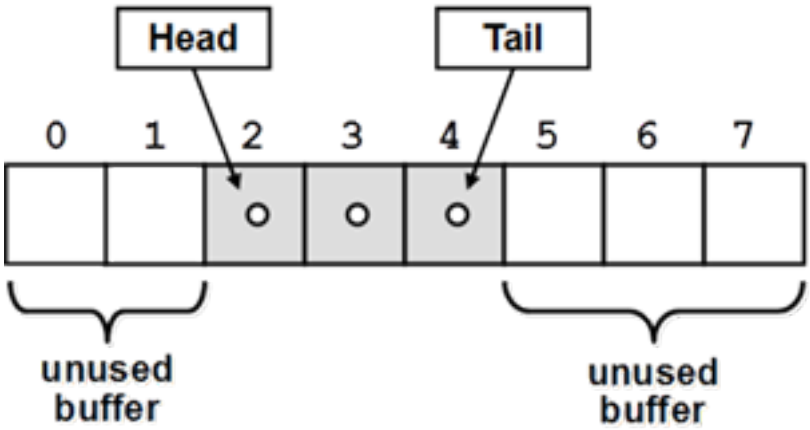


لیست پیوندی لازم نیست دو طرفه باشد و یک طرف برای کار با پشته مناسب است و دیگر لازم نیست که به انتهای لیست پیوندی عمل درج انجام شود؛ بلکه مقدار جدید به ابتدای آن اضافه شده و برای حذف گره هم اولین گره باید حذف شود و گره دوم به عنوان head شناخته می‌شود. همچنین لیست پیوندی نیازی به افزایش ظرفیت مانند آرایه‌ها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

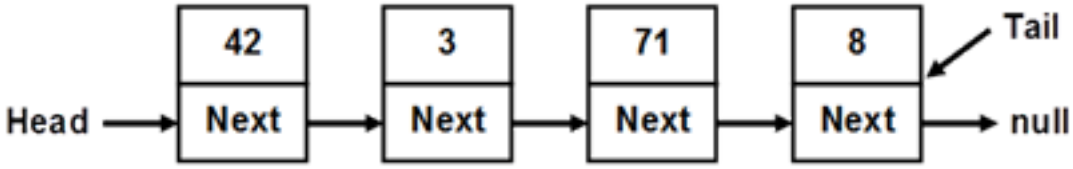


ابتدای آرایه مکانی است که عنصر از آنجا برداشته می‌شود و Head به آن اشاره می‌کند و tail هم به انتهای آرایه که جهت درج عنصر جدید مفید است. با برداشتن هر خانه‌ای که head به آن اشاره می‌کند، head یک خانه به سمت جلو حرکت می‌کند و زمانی که Head از tail بیشتر شود، یعنی اینکه دیگر عنصری یا المانی در صف وجود 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
```