دانشگاه آزاد اسلامی واحد رودهن

ساختمان های داده

استاد: ساسان آزاد

تالیف مطالب بر اساس کتاب:

An Introduction to Data Structures with Applications
(Tremblay – Sorenson)

جلسه ۱

ساختمان های داده و الگوریتم Data Structures and Algorithm

با توجه به عنوان فوق در می یابیم که در این مجموعه از مطالب با دو مقوله کلی مواجه هستیم:

۱- ساختمان های داده یا ساختارهای داده یا **Data Structures**، که در واقع چگونگی استفاده از حافظه اصلی است و تا کنون با تعریف متغیرها و تعریف آرایه ها در برنامه های نوشته شده قبلی انجام شده است، و در ادامه، با بررسی ساختارهای داده مختلف، در واقع روشهای مختلف بکارگیری حافظهٔ اصلی بررسی می شود.

۲- الگوریتم **Algorithm**

عملیات مختلف بر روی هر ساختار داده، طی یک روند یا رویه انجام می شود که به آن، الگوریتم انجام آن کار می گوییم.

کتاب مرجع :

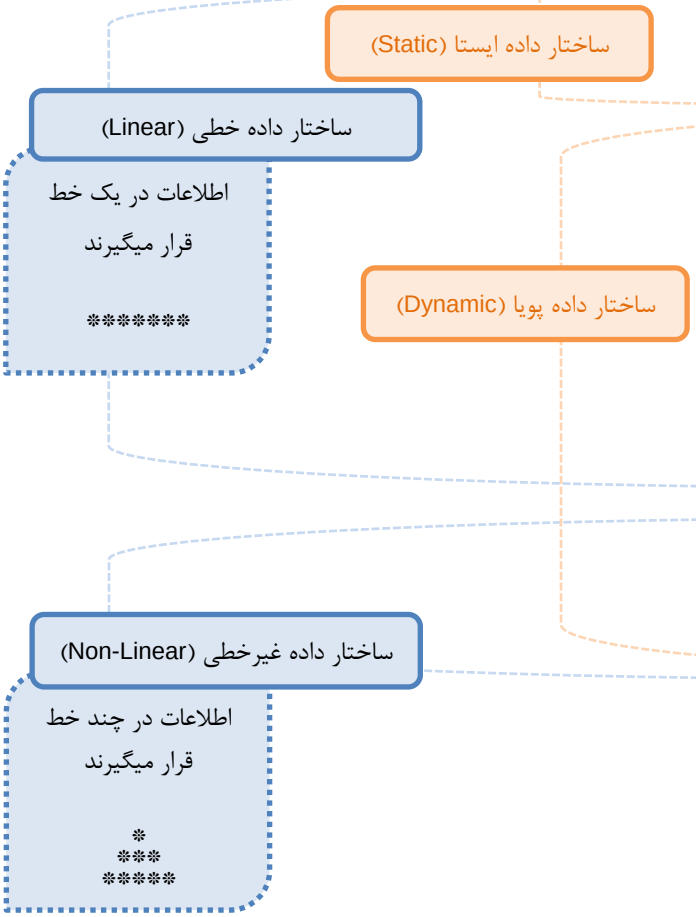
An Introduction to Data Structures with Applications (Tremblay – Sorenson)

دلیل انتخاب این کتاب به عنوان مرجع، گویش مناسب و ساده و عدم اتکا به زبان برنامه سازی خاص در ارائه ی الگوریتم ها می باشد. این کتاب از یک شبه زبان (**Pseudo Language**) بسیار ساده و روان در این راستا استفاده می نماید که قبل از شروع به بررسی ساختارهای داده ی مختلف، بررسی می گردد.

شرایط کلاس و تذکرها:

۱- این درس ابداً شب امتحانی یا یک هفته به امتحانی یا یک ماه مانده به امتحانی نیست
۲- تمرینات و لزوم حل آنها از سوی دانشجویان

فهرست مطالب :



خصوصیات ساختار دادهٔ ایستا (Static):

۱- اندازه (سایز) ثابت

۲- هرزروی حافظه بسیار بالاست

۳- کارکرد بسیار ساده و آسان است

نماینده: آرایه (**Array**)

- ✓ نام (**Name**)
- ✓ نوع (**Type**)
- ✓ بعد (**Dimension**)
- ✓ اندازه (**Size**)

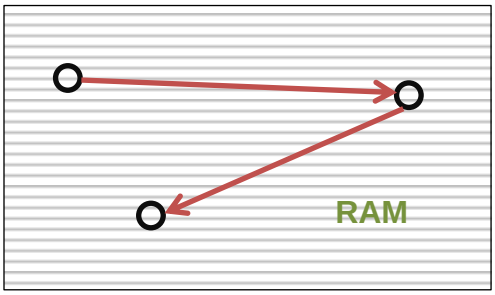
خصوصیات ساختار دادهٔ پویا (Dynamic):

۱- اندازه (سایز) متغیر

۲- هرزروی حافظه اصلاً وجود ندارد

۳- کارکرد نسبتاً مشکل تر است

نماینده: لیست پیوندی



جلسه ۲

ارائه ی الگوریتمی یا Algorithmic Notation

در بحث ارائه ی الگوریتمی و به منظور آشنایی با نحوه ی الگوریتم نویسی و نیز نحوه ی نوشتار، کار را با بررسی یک الگوریتم نمونه که قصد دارد عملیات خاصی را بر روی ساختار داده ی آشنای آرایه انجام دهد، آغاز می کنیم. عملیات خاص این الگوریتم عبارت است از یافتن یا پیدا نمودن بزرگترین عنصر از میان عناصر ذخیره شده در یک آرایه که باید ازدو دیدگاه بررسی شود:

۱- **قالب (Format) ارائه ی الگوریتم**

۲- نحوهٔ نوشتار (Syntax) ارائه ی الگوریتم

قالب (Format)

۱- هر الگوریتم با یکی از کلمات کلیدی: **Procedure** یا **Function** آغاز می شود که نشان دهنده نوع الگوریتم است. نوع **Procedure** در پایان، مقداری را به خارج از خود ارسال نمی کند اما نوع **Function** ارسال می کند.

۲- نام هر الگوریتم متناسب با عملکرد آن انتخاب، و پس از کلمات کلیدی فوق نوشته می شود.

۳- پارامترها در صورت وجود، در داخل پرانتز ( ) و پس از نام نوشته می شوند. لازم به ذکر است که پارامترها، مقادیری هستند که به الگوریتم یا برنامه وارد می شوند، و عملیات و محاسبات بر اساس آنها انجام می پذیرد: Sin(x)، Sin(۰)، Sin(۹۰) ...

۴- در این روش ارائهٔ الگوریتمی، نیاز به تعریف متغیرها نیست (**Variable Declaration**).

۵- هر الگوریتم طی مراحل (**Steps**) متعدد نوشته می شود.

۶- هر مرحله با یک توضیح (**Comment**) آغاز می گردد که درون یک کروشه [ ] نوشته می شود.

۷- معمولاً (۹۰٪ اوقات) مرحله ی اول الگوریتم اختصاص به بررسی حالات خطا و یا حالات خاص دارد.

ب) نحوهٔ نوشتار (Syntax)

۱- عبارت جایگزینی (**Assignment Statement**) : عبارت جایگزینی، مقداری را در متغیری یا متغیری را در متغیری دیگر قرار می دهد: (معادل نماد تساوی (=) در زبان های برنامه سازی)
نماد مورد استفاده: (←)

A ← ۵ B ← A

در راستای خلاصه نویسی در الگوریتم ها، می توان از موارد زیر استفاده کرد:

عبارت جایگزینی: A ←→ B تعویض مقادیر

مقداردهی همزمان: A ← B ← C ← ۰

۲- عبارت شرطی (If Statement)

```
IF Condition (شرط)
Then ..... Statement (عبارت)
     .....
     .....
     .....
Else ..... Statement (عبارت)
     .....
     .....
```

۳- حلقه (Loop)

برای انجام عملیات تکرار شونده از حلقه استفاده می کنیم که دارای دو نوع کلی زیر است :

I. حلقه با تعداد دفعات تکرار معلوم
II. حلقه با تعداد دفعات تکرار نامعلوم

I. Repeat thru step X For Index = Sequence

تا مرحلهٔ X الگوریتم

نشان دهنده شماره مرحله پایانی حلقه است

I = ۱ to ۱۰۰
J = ۵ to ۱۵

II. Repeat thru step X While Logical Expression

تا مرحلهٔ X الگوریتم

عبارت منطقی

- **نکته ۱ : بدنهٔ حلقه کمی جلوتر از واژه ی Repeat نوشته می شود.**
- **نکته ۲ : چنانچه بدنهٔ حلقه تنها یک مرحله باشد، نیاز به نوشتن Thru Step X نیست:**

```
Repeat While A<B
        A ← A-۱
        B ← B+۱
        C ← A+B
```

۴- چند واژهٔ کلیدی **Keywords**:

Go to Step X: موجب پرش و تغییر اجراء به مرحلهٔ X می شود

Exit / Return (Y): موجب اتمام الگوریتم و خروج از آن می شود

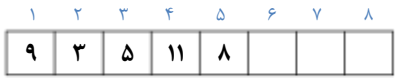
A [I]: A دسترسی به عنصر I اُم از آرایهٔ

Write (A): موجب به نمایش درآمدن مقدار یا پیغامی می شود Write (‘A’) یا

مثال: الگوریتم یافتن بزرگترین عضو از میان اعضای ذخیره شده در یک آرایه

فرض بر آن است که N، تعداد عناصر ذخیره شده می باشد:

```
Procedure Greatest
۱-  [Is the array empty?]
    IF  N < ۱
    Then Write ('Empty Array')
         Exit
۲-  [Initialization]
    Max ← A[۱]
    I ← ۲
۳-  [Examine all elements]
    Repeat thru step ۵  while I ≤ N
۴-  [Change Max if necessary]
         IF Max < A[I]
         Then Max ← A[I]
۵-  [Prepare for next examination]
         I ← I+۱
۶-  [Finished]
    Exit
```



Size = ۸

N = ۵

Max = ~~۹~~ → ۱۱

I = ~~۲~~ → ~~۳~~ → ~~۴~~ → ~~۵~~ → ۶

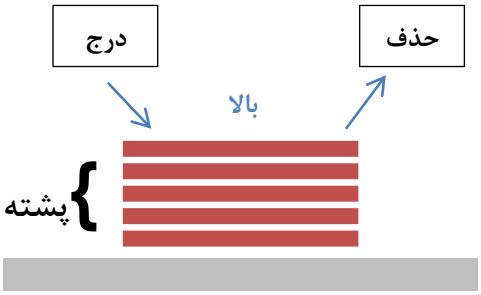
جلسهٔ ۳

ساختارهای داده ی ایستا یا Static

❶ ساختار دادهٔ پشته (Stack)

ساختار دادهٔ پشته یک ساختار **Last In First Out** یا به اختصار **LIFO** و یا به عبارت دیگر آخرین داده ی وارد شونده، اولین دادهٔ خارج شونده است.

همانگونه که در شکل ملاحظه میشود، کلیهٔ عملیات درج و حذف در نقطه ی بالای آن انجام می گردد و لذا نقطه ی بالا در پشته، از اهمیت زیادی برخوردار است.

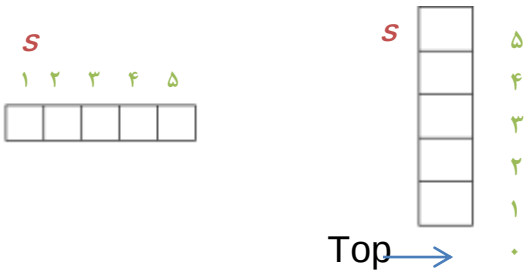


عمل درج: نقطه ی بالا

عمل حذف: نقطه ی بالا

نحوه ی شبیه سازی:

برای شبیه سازیِ خودِ پشته، از ساختار داده ی آرایه استفاده می کنیم:



برای شبیه سازی نقطهٔ مهم بالای پشته از متغیر کمکی با نام مناسبِ TOP استفاده می کنیم. TOP در شبیه سازی نشان دهنده ی بالاترین عنصر پشته است اما در واقع حاوی اندیسی از آرایه خواهد بود که بالاترین عضو و در واقع آخرین عضو اضافه شده به پشته در آن قرار گرفته است.

پشتهٔ خالی : **TOP = ۰**

پشتهٔ پر : **TOP = Size (N)**

مراحل پیاده سازی یک الگوریتم

برای طراحی و پیاده سازی یک الگوریتم تجربه به ما می گوید بهتر است مراحل زیر طی شود:

۱- رسم یک شکل فرضی از ساختار داده ی مورد بحث

۲- تحلیل عملیات انجام شونده بر روی شکل فرضی

۳- نوشتن تحلیل انجام شده به فرم محاوره ای (فارسی) [البته در یکی دو ساختار داده ی ابتدا]

۴- تبدیل فرم محاوره ای به فرم ارائه ی الگوریتمی

الگوریتم درج در پشته یا Push:

۱- به روز رسانی اشاره گر **TOP** به فرم افزایشی (**Incremental**).

۲- درج عضو مورد نظر در نقطه ای که **TOP** نشان می دهد.

خطا : چنانچه پشته پر باشد و بخواهیم درج کنیم خطای سرریزی پشته **Overflow** رخ می دهد.

فرض بر آن است که **N** سایز پشته است.

```
Procedure Push (S, TOP, X)
۱- [Overflow?]
   IF  TOP ≥ N
   Then Write ('Stack Overflow')
        Exit
۲- [Increment TOP Pointer]
   TOP ← TOP + ۱
۳- [Insert Element]
   S[TOP] ← X
۴- [Finished]
   Exit
```

الگوریتم حذف از پشته یا Pop: (بازیابی)

۱- حذف عنصری که TOP به آن اشاره دارد (در واقع ذخیره ی عنصری که **TOP** به آن اشاره دارد در متغیری کمکی جهت ارسال به خارج از الگوریتم در انتها).

۲- به روز رسانی اشاره گر TOP به صورت کاهشی **Decremental**

خطا : چنانچه پشته خالی باشد **و** بخواهیم عضوی حذف کنیم خطای **Underflow** رخ می دهد.

```
Function Pop (S, TOP)
    ۱-  [Underflow?]
        IF  TOP = ۰
        Then Write ('Stack Underflow')
             Return(۰)     به منزلهٔ عدم توانایی حذف عضو (۰)
    ۲-  [Delete Element]
        Y ← S[TOP]
    ۳-  [Decrement Top Pointer]
        TOP ← TOP -۱
    ۴-  [Finished]
        Return(Y)
```

تمرین: الگوریتمی طراحی کنید که با کمک خصوصیت LIFO پشته و الگوریتم های PUSH و POP، محتویات ذخیره شده در یک پشته فرضی به نام S را برعکس یا به عبارت دیگر معکوس نماید و در خروجی اصل و برعکس محتویات به نمایش در آید. به عنوان مثال اگر اعداد از ۱ تا ۱۰ به عنوان محتویات پشته در آن درج شده باشد، یک بار از ۱ تا ۱۰ و یک بار از ۱۰ تا ۱ نمایش داده شود.

تذکر: برای استفاده از الگوریتم های نوشته شده ی قبلی و موجود در الگوریتم های جدید (به عبارت دیگر فراخوانی آنها)، با توجه به نوع الگوریتم می توان به یکی از دو روش زیر عمل نمود:

۱- استفاده (فراخوانی) از نوع Procedure:

Call ProcedureName / Call Push (S, Top, X) / Call Push (S, ۲, 'w')

۲- استفاده (فراخوانی) از نوع Function:

X ← FunctionName / X ← Pop (S, Top) / Y ← Pop (S, ۳)

جلسهٔ ۴

❷ ساختار دادهٔ صف Queue

ساختار دادهٔ صف یک ساختار **First In First Out** یا به اختصار **FIFO** است. به عبارت دیگر اولین دادهٔ وارد شونده اولین دادهٔ خارج شونده است.

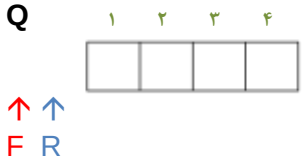
در صف دو نقطهٔ مهم وجود دارد :

- نقطه ی ابتدا، که عمل حذف از آن انجام می شود.
- نقطه ی انتها، که عمل درج در آن انجام می شود.

برای شبیه سازی صف از ساختار داده ی آرایه بهره می گیریم.

برای شبیه سازی نقاط مهم (طبق معمول) از متغیرهای کمکی با نام های مناسب استفاده می کنیم:

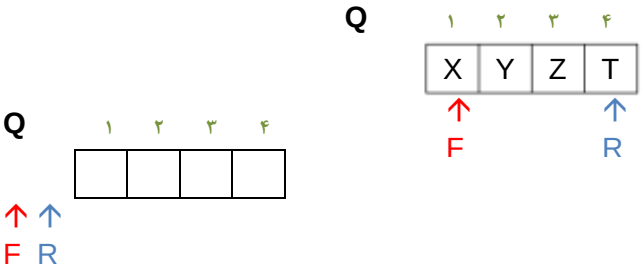
- ابتدا : **Front** یا به اختصار **F**
- انتها : **Rear** یا به اختصار **R**



در واقع **F** حاوی اندیسی از آرایه است که عضو ابتدایی صف در آن قرار دارد و **R** حاوی اندیسی از آرایه است که عضو انتهایی صف در آن قرار دارد.

صف خالی : $F = R = 0$ صف تک عضوی : $F = R \neq 0$

الگوریتم درج در صف



۱- به روز رسانی اشاره گر **R** به صورت افزایشی.

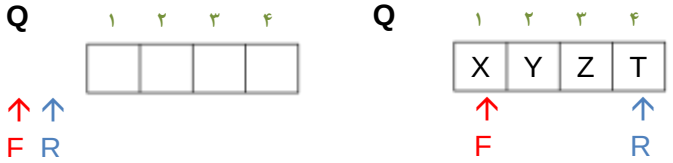
۲- درج عضو مورد نظر در نقطه ای که **R** نشان می دهد.

استثنا: در زمان درج اولین عضو، اشاره گر **F** نیز باید به روز رسانی شده و برابر **۱** گردد.

خطا: چنانچه صف پر باشد (**R=Size**) **و** بخواهیم عضوی درج نماییم خطای سرریزی رخ می دهد.

```
Procedure Qinsert (Q, F, R, N, X)
۱-  [Overflow?]
    IF  R ≥ N
    Then Write ('Overflow')
         Exit
۲-  [Increment Rear Pointer]
    R ← R + ۱
۳-  [Insert Element]
    Q[R] ← X
۴-  [Is front pointer proper?]
    IF  F = ۰
    Then  F ← ۱
    Exit
```

الگوریتم حذف از صف (بازیابی)



۱- حذف عنصری که **F** به آن اشاره دارد (در واقع ذخیره ی عنصری که **F** به آن اشاره دارد در یک متغیر کمکی جهت ارسال به خارج در پایان الگوریتم).

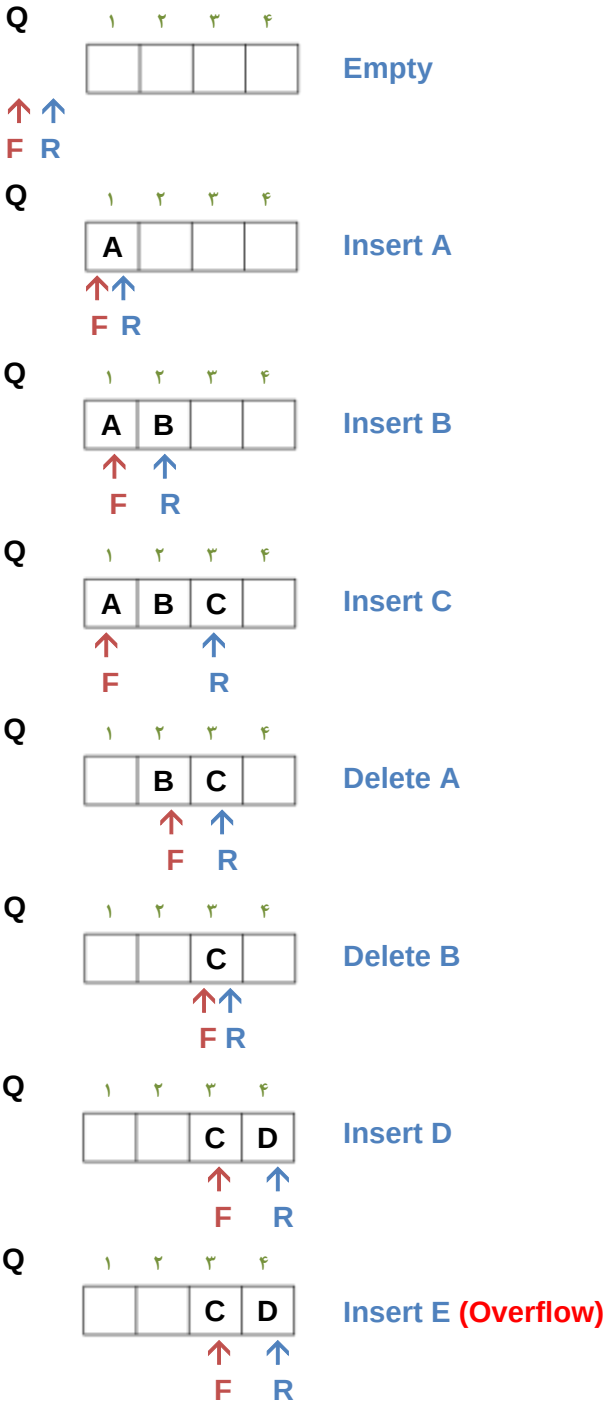
۲- به روز رسانی اشاره گر **F** به صورت افزایشی.

استثنا: در زمان حذف تک عضو، هردو اشاره گر **R** و **F** باید به روز رسانی شده و برابر صفر گردند.

خطا: چنانچه صف خالی باشد **و** بخواهیم عنصری حذف نماییم، خطای **Underflow** رخ می دهد.

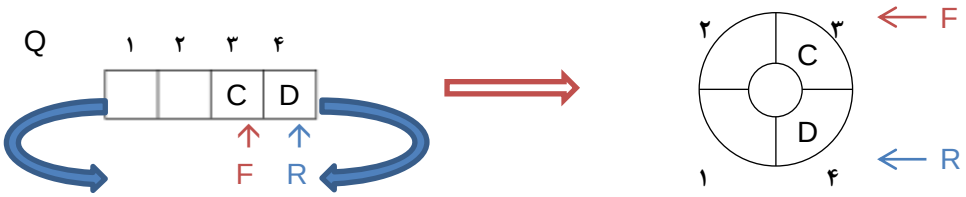
```
Function Qdelete (Q, F, R)
۱- [Underflow?]
   IF  F = ۰
   Then Write ('Underflow')
        Return(۰)     به منزلهٔ عدم توانایی حذف عضو (۰)
۲- [Delete Element]
   Y ← Q[F]
۳- [Queue Empty?]
   IF  F = R
   Then  F ← R ← ۰
   Else  F ← F + ۱
۴- [Finished]
   Return(Y)
```

تمرین: الگوریتمی طراحی کنید که اعداد از ۱ تا ۵ را در یک صف قرار داده (درج نماید) و سپس با کمک گرفتن از ساختار داده ی پشته، اعداد موجود در صف را بر عکس چاپ نماید.



با بررسی یک مثال و چند عمل درج و حذف تصادفی (شکل مقابل) در صف، به یک اشکال بالقوه برخورد میکنیم. همانگونه که ملاحظه میشود در این گونه صف، یعنی صف خطی (**Linear Queue**) و روش به کار گرفته شده برای به روز رسانی اشاره گرها (یعنی فقط افزایشی)، این امکان وجود دارد که پس از تعدادی درج و حذف تصادفی، علیرغم وجود فضای فیزیکی خالی در ابتدای صف، با سرریزی مواجه شویم. برای جلوگیری از این موضوع و هرز نرفتن حافظه، می توان روش به روز رسانی اشاره گر ها را تغییر داد و در واقع به ساختار جدیدی به نام صف مدور یا دورانی یا **Circular Queue** دست یافت که در آن تا زمانی که فضای فیزیکی خالی وجود دارد، قابلیت استفاده از آن نیز هست :

❸ صف مدور یا دورانی یا Circular Queue:



در به روز رسانی اشاره گر های **F** و **R** در صف دورانی، چنانچه اشاره گرها به سایز صف رسیدند، در واقع **Reset** شده و برابر با **۱** می گردند، که در رابطه با عمل درج و مشکل فوق الذکر، در صورتی که به فضای خالی و قابل استفاده اشاره نمایند، قابلیت استفاده از آن بدین صورت فراهم می گردد.

الگوریتم درج در صف دورانی

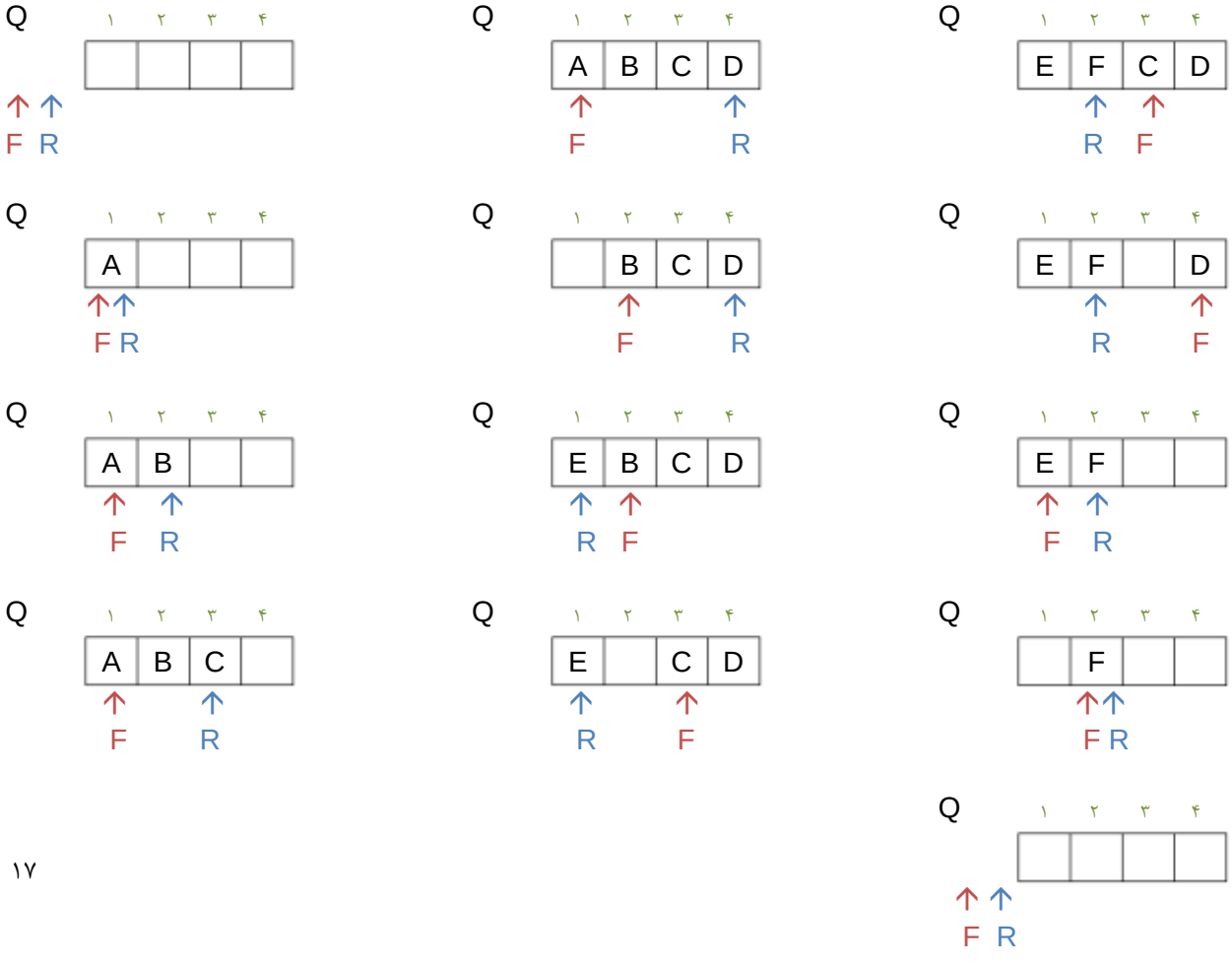
```
Procedure CQinsert (Q, F, R, N, X)
۱-  [Reset Rear Pointer]
    IF  R = N
    Then R ← ۱
    Else  R ← R + ۱
۲-  [Overflow?]
    IF F = R
    Then Write ('Overflow')
         Exit
۳-  [Insert Element]
    Q[R] ← X
۴-  [Is front pointer proper?]
    IF  F = ۰
    Then  F ← ۱
    Exit
```

مانند Qinsert

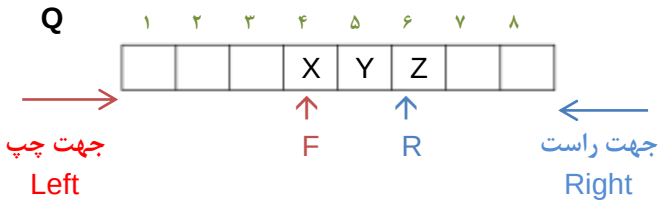
الگوریتم حذف از صف دورانی

```
Function CQdelete (Q, F, R)
۱-  [Underflow?]                          ┐
    IF  F = ۰                             │
    Then Write ('Underflow')              │ مانند Qdelete
         Return(۰)                        │
۲-  [Delete Element]                      │
    Y ← Q[F]                              ┘
۳-  [Queue Empty?]
    IF  F = R
    Then  F ← R ← ۰
          Return(Y)
۴-  [Update Front Pointer]
    IF  F=N
    Then  F ← ۱
    Else  F ← F + ۱
    Return(Y)
```

مثال قبل به طریق صف مدور (**Circular Queue**)



تمرین : یک صف دو جهته یا **Double Queue** یا به اختصار **Dequeue** صفی است که انجام عملیات درج و حذف در آن، علاوه بر اتکا به مفاهیم اصلی صف، یعنی ابتدا و انتها، متکی بر پارامتر جدیدی به نام جهت می باشد که معرف جهت انجام عملیات است :



الف) الگوریتمی طراحی کنید که یک عنصر در این ساختار درج نماید : **DQinsert**

ب) الگوریتمی طراحی کنید که یک عنصر از این ساختار حذف نماید : **DQdelete**

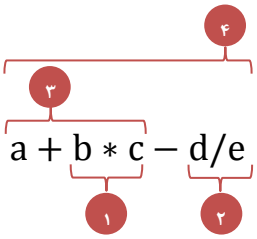
Procedure DQinsert (Q, F, R, N, Y, Jahat) پارامتر جدید

- نکته : چنانچه 'جهت' = **Right** باشد، در واقع مانند آن است که نقش اشاره گرهای **F** و **R** جابجا شده است : **F** انتها و **R** ابتدا میباشد.

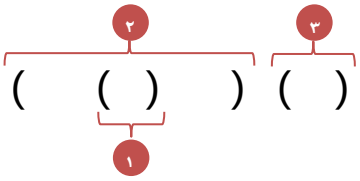
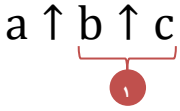
جلسهٔ ۵

کاربرد پشته در کامپایلرها

برای محاسبهٔ یک عبارت ریاضی مفروض مانند $a + b * c - d/e$ با توجه به اولویت های اپراتورها عمل می نماییم و تشخیص اولویت اپراتورها با توجه به امکان رؤیت همزمان آنها می باشد:



یادآوری اولویتها

اولویت	اپراتور	ملاحظات
۱	( )	( ) یک اپراتور نیست بلکه یک جداکننده یا Separator و اولویت دهنده است به این صورت که از چپ به راست و از داخل به خارج اولویت بالاتر است : 
۲	↑ توان	اولویت از راست به چپ بالاتر 
۳	/ و *	اولویت از چپ به راست بالاتر 
۴	− و +	اولویت از چپ به راست بالاتر 

اما در کامپایلرها امکان رؤیت همزمان اپراتورها وجود ندارد لذا هم اکنون هدف ما رسیدن به فرمی از عبارت ریاضی است که در محاسبهٔ آن نیازی به رؤیت همزمان اپراتورها نباشد.

فرم های معتبر مختلف عبارتهای ریاضی :

۱- **Prefix** (پیشوندی) Polish:

operator	operand	operand
+	a	b

۲- **Infix** (میانوندی):

operand	operator	operand
a	+	b

۳- **Postfix** (پسوندی) Reverse Polish:

operand	operand	operator
a	b	+

چند مثال ...

Infix	Prefix	Postfix
a	a	a
a + b	+ab	ab +
a + b + c	+ + ab c (T)	ab + c + (T)
a + (b + c)	+a + bc (T)	a bc + + (T)

توضیح : در زیر مراحل تبدیل یک عبارت Infix را به حالت های Prefix و Postfix مشاهده می کنید:

Prefix	Postfix
a + (b + c)	a + (b + c)
۱ : +bc	۱ : bc +
a + T	a + T
+aT	aT +
+a + bc	abc + +

Prefix	Postfix
a + b + c	a + b + c
۱ : +ab	۱ : ab +
T + c	T + c
+Tc	Tc +
+ + abc	ab + c +

- نکتهٔ ۱ : برای محاسبهٔ عبارت Prefix کافی است آن را از راست به چپ اسکن نموده و به محض برخورد با اپراتور، دو اپرند سمت راستی را عمل نماییم.
- نکتهٔ ۲ : برای محاسبهٔ عبارت Postfix کافی است آن را از چپ به راست اسکن نموده و به محض برخورد با اپراتور، دو اپرند سمت چپی را عمل نماییم.
- نکتهٔ ۳ : چنانچه عبارت Infix مفروض دارای پرانتز ( ) باشد، معادلهای Prefix و Postfix آن فاقد حضور فیزیکی آن ها می باشند، اما نقش اولویت دهی پرانتز ها همچنان وجود دارد.

از میان دو انتخاب Prefix و Postfix شیوهٔ Postfix را انتخاب می کنیم چون در آن، عملیات محاسبه از چپ به راست انجام می شود که با ماهیت عملکرد یک کامپایلر (اسکن از چپ به راست) هم خوانی دارد.

در مثالهای بالا همچنان نقش رؤیت همزمان اپراتورها در یافتن معادل های Prefix و Postfix وجود دارد. هم اکنون به دنبال روشی هستیم که در آن بدون نیاز به رؤیت همزمان اپراتورها، عملیات تبدیل انجام شود.

دانشمندی لهستانی (Polish) به نام «لوکا زویچز» روشی را ابداع کرد که به کمک یک جدول ۴ ستونی و با استفاده از خصوصیت LIFO پشته، یک عبارت Infix را بدون نیاز به رؤیت همزمان اپراتورها به Postfix تبدیل نمود:

Character Scanned	Stack	Reverse polish	Rank

۱- کاراکترهای اسکن شونده : کلیهٔ اپراتورها و اپرندهای عبارت ریاضی Infix
۲- پشته ای که به کمک آن عملیات تبدیل انجام میشود (درج و حذف)
۳- عبارت مقصد Postfix
۴- عدد کنترل کنندهٔ صحت عبارت حاصل

طبق تعریف، عبارت ریاضی معتبر، دارای Rank = ۱ است و برای محاسبه Rank کلی یک عبارت باید Rank هر کاراکتر آن را محاسبه و همه را باهم جمع نماییم.

Rank (عبارت ریاضی) = جمع جبری Rankهای کاراکترها

Rank (Operator) = -۱

Rank (Operand) = +۱

مثال :

عبارت معتبر

$$a + b - c * d / e$$

$$(+1) + (-1) + (+1) + (-1) + (+1) + (-1) + (+1) + (-1) + (+1) = +1$$

عبارت نا معتبر

$$a + b - c * d / e +$$

$$(+1) + (-1) + (+1) + (-1) + (+1) + (-1) + (+1) + (-1) + (+1) + (-1) = 0$$

شرح عملیات و نحوه ی عملکرد: (قوانینی در نحوه استفاده از این جدول)

۱. هر کاراکتر موجود در عبارت Infix باید به طور جداگانه و تک تک اسکن شده (یعنی در ستون اول جدول قرار گیرد) و بلافاصله در پشته یا Stack درج گردد (یعنی در ستون دوم جدول نوشته شود).

۲. هرکاراکتر درج شونده در پشته از دو حالت خارج نیست: یا اپرند است و یا اپراتور می باشد که بسته به این موضوع، به یکی از دو حالت زیر عمل می نماییم:

الف) اگر کاراکتر درج شونده در پشته اپرند باشد، عمل درج بدون مشکل انجام می پذیرد.

ب) اگر کاراکتر درج شونده در پشته اپراتور باشد، به دلیل لزوم حفظ اولویت های اپراتورهای موجود در عبارت اصلی، انجام عمل درج با توجه به نوع کاراکتر موجود در بالای پشته صورت می پذیرد (که باز یا اپرند است و یا اپراتور).

ب۱) اگر اپراتور اسکن شد، و در بالای پشته اپرند وجود داشته باشد، ابتدا اپرند از پشته POP شده (حذف گردیده) و در ستون Reverse Polish نوشته می شود، سپس می توان اپراتور اسکن شده را در پشته درج نمود.

ب۲) اگر اپراتور اسکن شد، و در بالای پشته اپراتور وجود داشته باشد، باید با توجه به اولویت ها عمل نمود. به این ترتیب که:
اگر تقدم اپراتور موجود در بالای پشته، کمتر از تقدم اپراتور اسکن شده بود، آن اپراتور موجود، در پشته باقی مانده و درج اپراتور اسکن شده در پشته انجام می گردد. اما اگر تقدم اپراتور موجود در بالای پشته، بیشتر از تقدم اپراتور اسکن شده بود، ابتدا اپراتور موجود در پشته را POP نموده و در ستون Reverse Polish می نویسیم، سپس اقدام به درج اپراتور اسکن شده در پشته می نماییم.

۳. چنانچه عبارت Infix مفروض فاقد پرانتز باشد، عملیات در جدول را با قرار دادن یک # (Hash Tag / Number Sign) در پشته، و در صورتی که دارای پرانتز باشد، عملیات در جدول را با قرار دادن یک) یا پرانتز باز در پشته آغاز می نماییم. ضمناً در زمان رسیدن به انتهای عبارت Infix، هر آنچه که در پشته باقی مانده باشد را POP نموده و در Reverse Polish می نویسیم.

✎ مثال : عبارت ریاضی میانوندی زیر را با استفاده از روش جدول چهارستونی زویچز به یک عبارت پسوندی معتبر تبدیل کنید:

$$a + b * c - d / e * h$$

Character Scanned	Stack	Reverse polish	Rank
	#	--------	----
a	#a	--------	----
+	# +	a	۱
b	# + b	a	۱
*	# + *	ab	۲
c	# + * c	ab	۲
−	# −	abc * +	۱
d	# − d	abc * +	۱
/	# − /	abc * +d	۲
e	# − / e	abc * +d	۲
*	# − *	abc * +de/	۲
h	# − * h	abc * +de/	۲
		abc * +de/h * −	۱

☜ تمرین : عبارت ریاضی میانوندی زیر را با استفاده از روش جدول چهارستونی زویچز به یک عبارت پسوندی معتبر تبدیل کنید:

$$(a + b \uparrow c \uparrow d) * (e + f / d)$$

جلسهٔ ۶

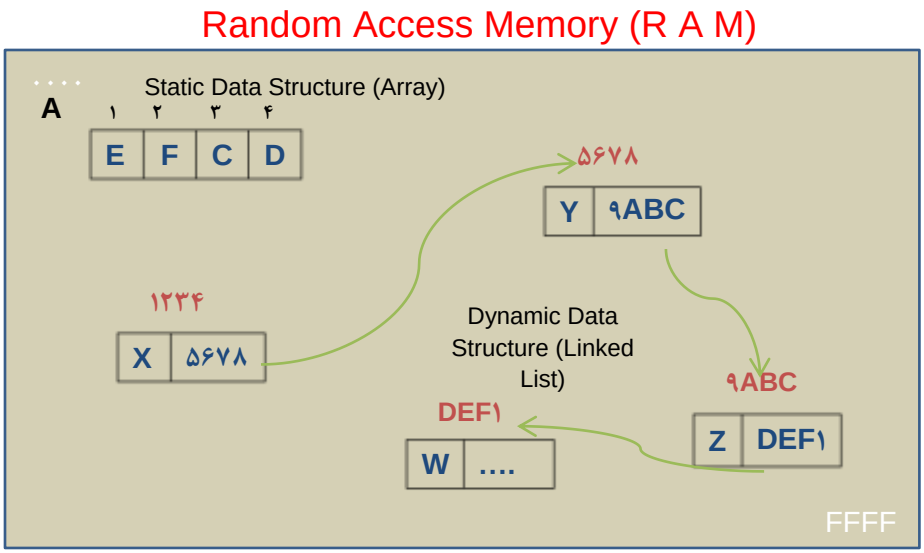
ساختارهای داده ی پویا (Dynamic Data Structures)

لیستهای پیوندی Linked Lists

❶ لیستهای پیوندی یک جهته : Single Linked Lists

همانگونه که قبلاً هم اشاره شده است، ساختارهای داده ی پویا، ساختارهایی هستند که اندازهٔ آنها ثابت نیست و همواره در حال ازدیاد عنصر یا کاهش عنصر (Grow and Shrink) می باشند.

روند استفاده از این ساختار بدین ترتیب است که در زمان نیاز به استفاده از حافظهٔ اصلی، درخواستی را مبنی بر این نیاز صادر می نماییم و در پاسخ، بخشی از حافظهٔ اصلی در اختیارمان قرار می گیرد و ما استفاده ی لازم را می بریم. در مقابل و در زمان عدم نیاز به هر بخش، آن را رها می سازیم تا به مجموعهٔ قابل استفادهٔ حافظهٔ اصلی باز گردد. ذیلاً با یک مثال این موضوع را دنبال می کنیم: فرض کنید اولین نیاز به استفاده از حافظه ی اصلی بوجود آمد و لذا یک درخواست صادر می کنیم و در پاسخ، با توجه به ماهیت تصادفی بودن حافظه ی اصلی، بخشی از آن با آدرس فرضی **۱۲۳۴** به شکل تصادفی دراختیارمان قرار می گیرد و ما استفاده ی لازم را می نماییم. مجدداً فرض کنید بار دیگر نیاز به استفاده از حافظه ی اصلی بوجود آمد و لذا یک درخواست دیگر صادر می کنیم و در پاسخ، باز با توجه به ماهیت تصادفی بودن حافظه ی اصلی، بخشی دیگر از آن با آدرس فرضی **۵۶۷۸** به شکل تصادفی در اختیارمان قرار گرفته و استفاده ی لازم را می نماییم. در نیاز سوم مجدداً یک درخواست دیگر صادر می کنیم و در پاسخ، باز با توجه به ماهیت تصادفی بودن حافظه ی اصلی، بخشی دیگر از آن با آدرس فرضی **۹ABC** به شکل تصادفی در اختیارمان قرار گرفته و استفاده ی لازم را می نماییم.



با توجه به شکل بالا، در رابطه با یک ساختار دادهٔ ایستا و ساختار دادهٔ پویا، آنچه که کاملاً قابل مشاهده است، وجود توالی در ساختار داده ی ایستا و عدم توالی در ساختار داده ی پویا می باشد. از آنجایی که مسئلهٔ توالی عناصر امری مهم است، در ساختار داده ی پویا باید آن را خود بوجود آوریم و لذا لازم است در کنار اطلاعات ذخیره شونده، آدرس عضو بعد را نیز ذخیره کنیم. بنابراین، هر عضو از یک ساختار دادهٔ لیست پیوندی یک جهته دارای دو بخش است:

۱- بخش ذخیرهٔ اطلاعات: Info

۲- بخش ذخیرهٔ آدرس عضو بعدی: Link

شکل تمیزتر:

نکات مهم:

۱- در لیست پیوندی یک جهته، نقطه ی مهم عبارت است از آدرس عضو اول، که برای حفظ آن طبق معمول ساختارهای دادهٔ قبلی عمل کرده و آن را در متغیری کمکی با نام مناسب حفظ می کنیم:

First / Start / Begin

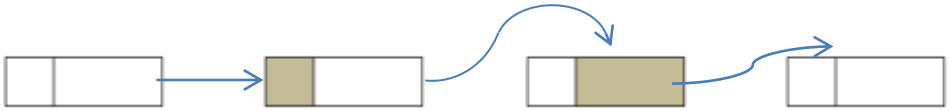
۲- لیست پیوندی یک جهته تهی، دارای هیچ عضوی نیست و لذا در آن داریم:

First = Null

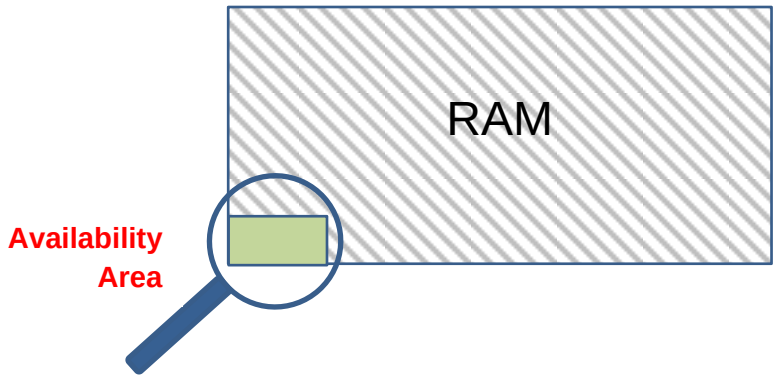
۳- در لیستهای پیوندی یک جهته، عنصر آخر دارای عضو بعدی نبوده و لذا بخش Link آن تهی (Null) میباشد.

۴- عملیات درج و حذف در لیستهای پیوندی، عملیاتی فیزیکی بوده و لذا در هر نقطه از آن یعنی ابتدا، وسط و انتها امکان پذیر است.

۵- هر عنصر دارای یک آدرس و آن آدرس دارای دو بخش است که یکی حاوی اطلاعات و یکی حاوی آدرس عضو بعدی است. در رسم شکل فرضی، هیچ تفاوتی نمی کند نوک پیکان به کجای عضو بعد اشاره کند. آنچه که مهم است، این می باشد که عضو بعدی هر عنصر، با این روش به نمایش در می آید.



۶- **شکل رسم شده از حافظهٔ اصلی تنها مربوط به بخشی از آن به نام محدودهٔ موجودیت یا Availability Area می باشد که تا زمانی که فضای خالی دارد می توان از آن در لیستهای پیوندی استفاده نمود.**



۷- برای انجام درخواست حافظه ی اصلی در الگوریتم نویسی، از نماد درخواست (← یا == >) استفاده می نماییم:

New← Node (New < == Node)

و برای انجام رهاسازی در الگوریتم نویسی، از تابع Restore بهره میگیریم:

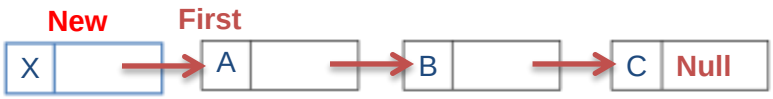
Restore (آدرس) Restore (۱۲۳۴) Restore (X)

همانطور که گفته شد امکان انجام عملیات درج و حذف در هر نقطه از یک لیست پیوندی وجود دارد و لذا برای درج عضو در لیست پیوندی یک جهته، ۳ الگوریتم مجزای زیر را ارائه می دهیم:

۱- درج در ابتدای لیست پیوندی یک جهته
۲- درج در انتهای لیست پیوندی یک جهته
۳- درج عضو در میانه ی لیست پیوندی یک جهته

۱- الگوریتم درج در ابتدای لیست پیوندی یک جهته:

با توجه به اهمیت آدرس شروع (عضو اول) در لیست های پیوندی یک جهته و برای تأکید بر این اهمیت، کلیهٔ الگوریتم های لیست های پیوندی یک جهته اعم از درج و حذف را از نوع Function طراحی کرده و آنچه را که به خارج ارسال می کنیم، همان آدرس عضو اول خواهد بود.




```
Function Insert (First, X)
    ۱-  [Obtain new node]
        New ← Node          ( < == )
    ۲-  [Set Information field]
        Info(New) ← X
    ۳-  [Set Link field]
        Link(New) ← First
    ۴-  [Return address of first node]
        Return(New)
```

OR
- First ← New
- Return (First)

جلسهٔ ۷

۲- الگوریتم درج در انتهای لیست پیوندی یک جهته:

برای درج یک عضو در انتهای لیست پیوندی یک جهته، طبق روال، درخواست حافظه را انجام داده که در پاسخ بخشی از حافظهٔ اصلی در اختیارمان قرار می گیرد. مقداردهی دو بخش این عضو نیز به سادگی قابل انجام است چون اطلاعات در بخش Info قرار گرفته و با توجه به این که عضو جدید، خود عنصر آخر خواهد بود، Null در بخش Link قرار می گیرد. اما این عضو هنوز به لیست پیوندی متصل نشده که این کار باید انجام شود.

از طرفی، در لیستهای پیوندی یک جهته، آدرس عضو اول مهم است که آن را در متغیر کمکی به نام **First** حفظ می نمودیم و به جز این آدرس، اطلاعات دیگری از لیست پیوندی یک جهته در اختیار نداریم و در جایی ذخیره نمی نماییم. اما در این الگوریتم، نیاز به آن داریم که به عضو آخر دسترسی پیدا نموده و آدرس عضو جدید را در بخش Link آن قرار دهیم تا اتصال به وجود آید. بنا بر این در این مسئله و هر مسئلهٔ دیگری که در رابطه با لیستهای پیوندی یک جهته مطرح می شود و با عنصری به غیر از عضو اول کاری داشته باشیم، لازم است ابتدا آن عنصر را بیابیم که این عمل تحت فرایندی به نام پیمایش یا Traverse انجام می گردد که از ابتدای لیست پیوندی آغاز شده و به سمت انتهای آن به پیش می رود. البته ممکن است بنا به نیاز، عمل پیمایش در میانهٔ لیست متوقف گردد.

برای انجام عمل پیمایش ۲ مورد زیر الزامی است:

۱- ذخیرهٔ آدرس عضو اول در یک متغیر کمکی و در واقع استفاده از یک نسخهٔ کپی از آن به علت اهمیت آدرس شروع:

Save ← First

۲- به علت تکراری بودن عملیات پیمایش، استفاده از یک حلقه با دفعات تکرار نامعلوم برای بررسی رسیدن به عضو هدف یا خیر و نیز رفتن به عضو بعدی (البته حداکثر تا انتهای لیست):

Repeat While Link (Save) ≠Null

Save ← Link (Save)

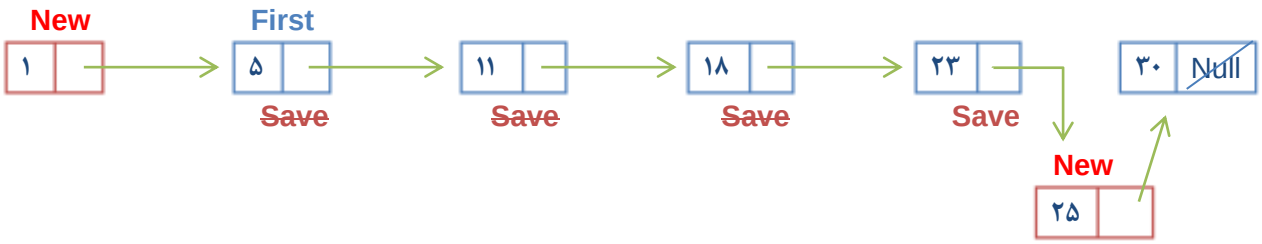
* آدرس عضو بعدی Save را در متغیر Save قرار بده (حرکت کن به سمت عضو بعد)

ضمناً برای توقف در میانهٔ لیست پیوندی یک جهته در صورت لزوم، می توان با یک and شرط مورد نظر دوم را به شرط اول اضافه نمود و به این ترتیب در واقع هر یک از شرط ها نقض شود، به عضو هدف رسیده ایم (یا به انتهای لیست و یا به شرط دوم):

Repeat While Link (Save) ≠ Null and شرط دوم

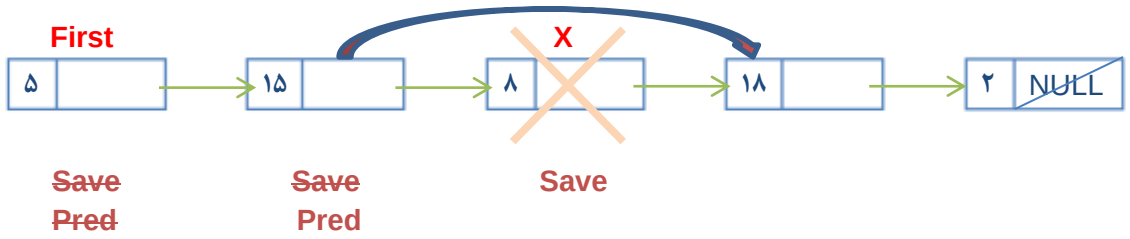

```
Function InsEnd (X, First)

۱- [Obtain new node]
   New ← Node                    ( < == )
۲- [Set fields of new node]
   Info(New) ← X
   Link(New) ← Null
۳- [Is the list empty?]
   IF First=Null
   Then Return(New)
۴- [Initialize search]
   Save ← First
۵- [Search for end of list]
   Repeat While Link(Save)≠Null
            Save ← Link(Save)
۶- [Set link field of last node to New]
   Link(Save) ← New
۷- [Return address of first node]
   Return(First)
```

۳- الگوریتم درج در لیست پیوندی مرتب یا دارای ترتیب (میانه لیست): **Ordered List**




```
Function InsOrd (X, First)
    ۱- [Obtain new node]
       New ← Node                    ( < == )
    ۲- [Set information field]
       Info(New) ← X
    ۳- [Is the list empty?]
       IF First = Null
       Then  Link(New) ← Null
             Return(New)
    ۴- [Does the new node precede others?]
       IF X ≤ Info(First)        (Info(New) ≤ Info(First) )
       Then  Link(New) ← First
             Return(New)
    ۵- [Initialize search]
       Save ← First
    ۶- [Search for predecessor of new node]
       Repeat While   Link(Save) ≠ Null   and   Info(Link(Save)) ≤ Info(New)
                Save ← Link(save)
    ۷- [Set link fields of new node and its predecessor]
       Link(New) ← Link(Save)
       Link(Save) ← New
    ۸- [Return address of first node]
       Return(First)
```

۴- الگوریتم حذف یک عضو با آدرس فرضی X از لیست پیوندی یک جهته



توجه : در الگوریتم زیر، X یک آدرس است:

```
Function Delete (First, X)

۱- [Is the list empty?]
   IF First = Null
   Then  Write('Empty list')
         Return(First)
۲- [Initialize search]
   Save ← First
۳- [Search for X]
   Repeat While   Link(Save) ≠ Null   AND   Save ≠ X
          Pred ← Save
          Save ← Link(save)
۴- [End of list?]
   IF Save ≠ X
   Then   Write('Address not found')
          Return(First)
۵- [Delete X]
   IF X = First
   Then First ← Link(First)
   Else  Link(Pred) ← Link(Save)
۶- [Return deleted node to availability area]
   Restore(X)
   Return(First)
```

تمرین:

۱- الگوریتمی طراحی کنید که تعداد عناصر موجود در یک لیست پیوندی یک جهته را شمارش نماید.

۲- الگوریتمی طراحی کنید که مقدار بخش Info از عنصر K اُم (K یک عدد است. مثلاً K = ۲ یعنی عنصر دوم، K = ۵ یعنی عنصر پنجم) در یک لیست پیوندی یک جهته را به مقدار دلخواه Y تغییر دهد.

۳- الگوریتمی طراحی کنید که در سمت چپ عضو K اُم (K یک عدد است. مثلاً K = ۲ یعنی عنصر دوم، K = ۵ یعنی عنصر پنجم) در یک لیست پیوندی یک جهته، یک عضو جدید درج نماید.

۴- الگوریتمی طراحی کنید که دو لیست پیوندی یک جهته با آدرس های شروع First۱ و First۲ را به یکدیگر متصل نموده و در انتها یک لیست پیوندی یک جهته وجود داشته باشد.

۵- الگوریتمی طراحی کنید که یک لیست پیوندی یک جهته را از آدرس فرضی S به دو بخش تقسیم نماید به طوری که S، آدرس عضو آغازین لیست پیوندی دوم باشد.

۶- الگوریتمی طراحی کنید که از روی یک لیست پیوندی یک جهته، یک نسخه کپی تهیه نموده و در انتها دو لیست پیوندی یک جهته که اطلاعات (بخش Info عناصر) آنها عیناً مانند یکدیگر است وجود داشته باشد.

فراموش نفرمایید که برای طراحی یک الگوریتم، ابتدا شکل فرضی از ساختار داده ی مربوطه را رسم کرده و بر روی شکل تحلیل نمایید، سپس اقدام به طراحی الگوریتم کنید.

جلسهٔ ۸

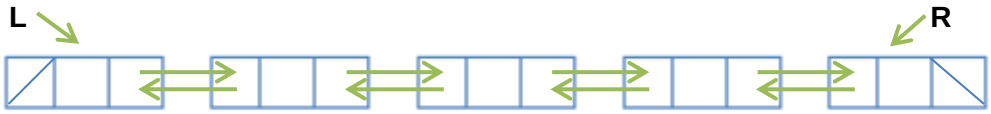
❷ لیست پیوندی دوجهته: Double Linked List

قبل از آغاز به بررسی لیست های پیوندی دوجهته، باید گفت که در واقع تمام مفاهیم اساسی در لیست های پیوندی چه از نوع یک جهته و چه از نوع دوجهته، با هم مشابه هستند و تنها تفاوت موجود میان آنها بخشهای تشکیل دهندهٔ هریک از عناصر آن هاست:

LPTR: حاوی آدرس عضو سمت چپ

Info: حاوی اطلاعات

RPTR: حاوی آدرس عضو سمت راست



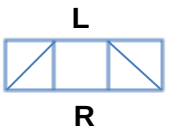
در لیست های پیوندی دوجهته آدرس دو عضو مهم است که باید آنها را حفظ نمود و برای این کار طبق معمول، از دو متغیر کمکی با اسامی مناسب بهره می گیریم. این دو آدرس عبارتند از آدرس سمت چپ ترین عضو و آدرس سمت راست ترین عضو:

L (Left Most): متغیر کمکی برای حفظ آدرس سمت چپ ترین عضو

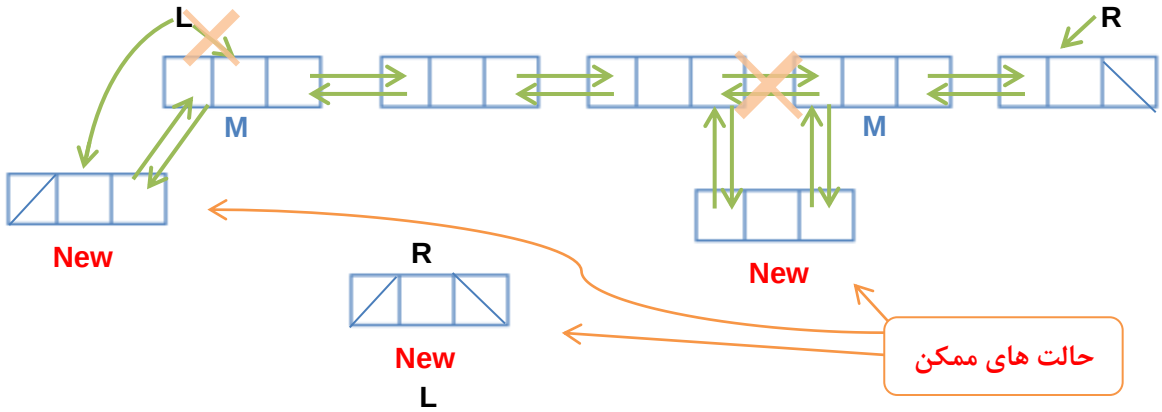
R (Right Most): متغیر کمکی برای حفظ آدرس سمت راست ترین عضو

لیست پیوندی دوجهتهٔ تهی: **L = R = Null**

لیست پیوندی دوجهته تک عضو: **L = R ≠ Null**



الگوریتم درج یک عضو در سمت چپ آدرس M در لیست پیوندی دوجهته:

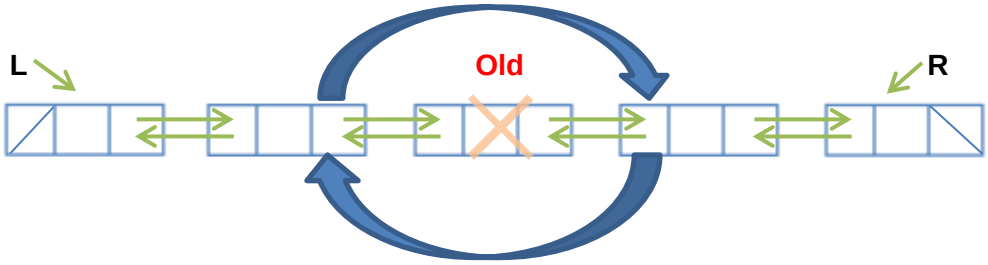


(نکته: این الگوریتم امکان درج یک عضو در سمتِ راستِ راست ترین عنصر لیست را ندارد)

تمرین: الگوریتمی طراحی کنید که در سمت راست آدرس M در یک لیست پیوندی دو جهته یک عنصر درج نماید.

```
Procedure DoubIns (L, R, M, X)

۱-  [Obtain New Node]
    New ← Node
۲-  [Set information field]
    Info(New) ← X
۳-  [Is the list empty?]
    IF L = Null
    Then   L ← R ← New
           LPTR(New) ← RPTR(New)← Null
           Exit
۴-  [Leftmost Insertion?]
    IF   M=L
    Then   LPTR(New) ← Null
           RPTR(New) ← M
           LPTR(L) ← New
           L ← New
           Exit
۵-   [Insert in the middle]
    RPTR(New) ← M
    LPTR(New) ← LPTR(M)
    LPTR(M) ← New
    RPTR(LPTR(New)) ← New
    Exit
```

الگوریتم حذف یک عنصر با آدرس Old از یک لیست پیوندی دوجهته:



Procedure DoubDel (L, R, Old)

۱. [Is the list empty?]

If L = Null
Then write ('Empty List')
Exit

۲. [Delete Node]

If L = R حذف تک عنصر

Then L ← R ← Null

Else If Old = L حذف سمت چپ ترین عنصر

Then L ← RPTR (L)
LPTR (L) ← Null

Else If Old = R حذف سمت راست ترین عنصر

Then R ← LPTR(R)
RPTR(R) ← Null

Else RPTR (LPTR (Old)) ← RPTR (Old) حالت کلی حذف از میانه لیست

LPTR (RPTR (OLD)) ← LPTR (Old)

۳. [Return Deleted Node to Availability Area]

Restore (Old)
Exit

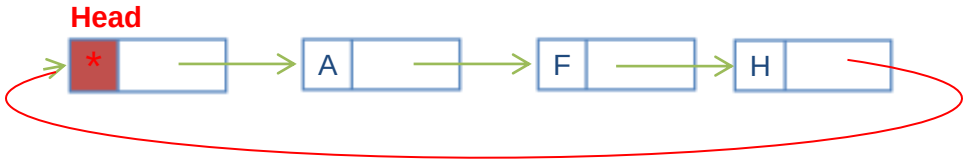
دوران در لیست های پیوندی – لیست های مدور: Circular Lists

بطور کلی بحث دوران در لیست های پیوندی، هم در نوع یک جهته و هم در نوع دو جهته مطرح است:

الف) لیست های پیوندی یک جهتهٔ مدور: Single Circular Linked Lists

با اولین نگاه به لیستهای پیوندی یک جهته و بررسی دوران، ساده ترین روش برای ایجاد آن، قرار دادن آدرس عضو اول یا First در بخش Link آدرس آخر به جای Null است :

امّا این روش ایجاد دوران، اولاً قابلیت اطمینان پایینی دارد (چون ممکن است درگیر یک دوران بی انتها شویم) و ثانیاً به عنوان یک ساختار مستقل از لیستهای پیوندی عادی به حساب نمی آید (چون در یک لیست یک جهته عادی تهی داریم First = Null است و نیز در یک لیست پیوندی یک جهته که با این روش مدور شده، تهی بودن همچنان First = Null است). لذا برای حل این مشکل، در هنگام ایجاد دوران در لیستهای پیوندی، از یک عنصر جدید که به آنها اضافه شده و به آن سرلیست یا Head می گویند، استفاده می شود:



خصوصیات سرلیست:

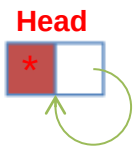
۱- آدرس سرلیست در متغیر کمکی با نام مناسب Head ذخیره می شود.

۲- بخش اطلاعات Head مهم نبوده و اساساً سرلیست جزو داده های لیست پیوندی به حساب نمی آید.

۳- در بخش Link(Head)، آدرس عضو اول قرار می گیرد و دیگر نیاز به متغیر کمکی First نیست.

۴- آدرس Head در بخش Link عضو آخر (به جای Null) قرار می گیرد.

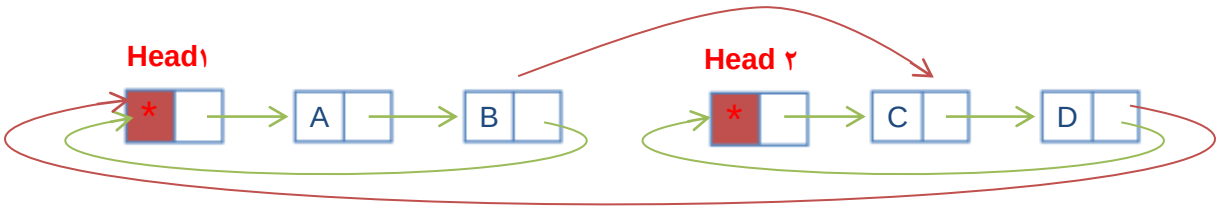
۵- لیست پیوندی یک جهته مدور تهی : Link(Head)=Head



تفاوت ها	تهی بودن	عنصر آخر
لیست پیوندی یک جهته عادی	First = Null	Link(عضو آخر) = Null
لیست پیوندی یک جهته مدور	Link(Head) = Head	Link(عضو آخر) = Head

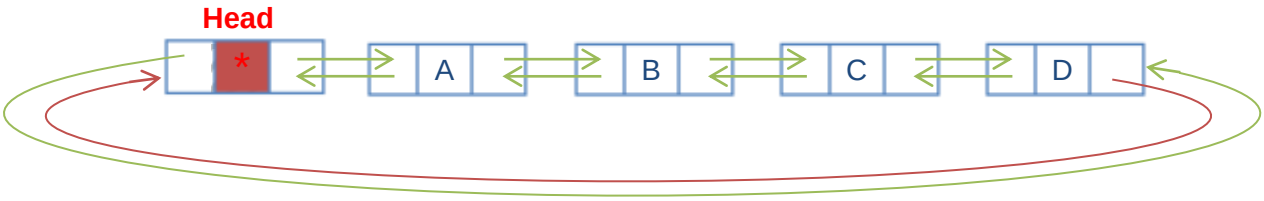
☜ تمرین : کلیهٔ الگوریتم های لیستهای پیوندی یک جهته را با فرض مدور بودن لیست یک جهته، اعم از متن درس و تمرینات مجدداً بررسی و طراحی نمایید. (جمعاً ۱۰ تمرین)

- توضیح واضح : یک عدد لیست پیوندی یک جهته مدور دارای یک سرلیست و دو عدد لیست پیوندی یک جهته مدور دارای دو سرلیست هستند.



(طرحی از تغییرات لازم در هنگام اتصال دو لیست پیوندی یک جهته مدور)

ب) لیستهای پیوندی دوجهتهٔ مدور: Double Circular Linked Lists



خصوصیات سرلیست

۱- آدرس سرلیست در متغیر کمکی با نام مناسب Head ذخیره می شود.

۲- بخش اطلاعات Head مهم نبوده و اساساً جزو داده های لیست پیوندی به حساب نمی آید.

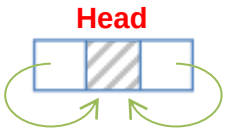
۳- RPTR(Head) = L

۴- LPTR(Head) = R

۵- LPTR(L) = Head

۶- RPTR(R) = Head

۷- لیست پیوندی دوجهته مدور تهی: LPTR(Head) = RPTR(Head) = Head



☜ تمرین : کلیهٔ الگوریتم های لیست های پیوندی دوجهته را با فرض اینکه لیست پیوندی دوجهته مدور است، مجدداً بررسی و طراحی نمایید.

جلسهٔ ۹

ساختارهای داده غیر خطی (Non Linear Data Structures)

تئوری گراف

تعریف عمومی گراف: به مجموعهٔ اعضا و ارتباط های موجود میان آنها در هر سیستم فرضی، گراف می گویند.

تعریف ریاضی گراف: G = <V, E> گراف G شامل دو مجموعهٔ V و E می باشد.

مجموعهٔ V که مخفف Vertex می باشد، در واقع همان مجموعهٔ اعضا یا نقاط یا گره ها یا Node ها بوده و این مجموعه حتماً غیر تهی است.

مجموعه E که مخفف Edge می باشد، در واقع همان مجموعه ارتباط ها یا راه ها یا اضلاع و یا لبه ها بوده و این مجموعه می تواند تهی باشد.

***نتیجه آنکه برای بوجود آمدن یک گراف حداقل یک گره لازم است.**

تعریف لبه یا Edge: ارتباط میان گره ها توسط لبه ها انجام می شود.

انواع لبه:

۱- لبه جهت دار یا Directed Edge: لبه ای است که جهت حرکت در آن معلوم و مشخص می باشد.

۲- لبه بدون جهت یا Undirected Edge: لبه ای است که جهت حرکت در آن معلوم و مشخص نمی باشد.

دسته بندی گراف ها بر حسب نوع لبه:

۱- گراف جهت دار یا Directed Graph یا به اختصار Digraph: گرافی است که در آن تمامی لبه ها جهت دار هستند.

۲- گراف بدون جهت یا Undirected Graph: گرافی است که در آن تمامی لبه ها بدون جهت هستند.

۳- گراف مخلوط یا Mixed Graph: گرافی است که در آن بعضی از لبه ها جهت دار و بعضی دیگر بدون جهت هستند.

تعریف گره آغازین یا Initial Node: گره ای است که در یک لبه حرکت از آن شروع می گردد.

تعریف گره پایانی یا Terminal Node: گره ای است که در یک لبه حرکت به آن ختم می گردد.

تعریف حلقه یا Loop: چنانچه در یک لبه، گره های آغازین و پایانی یکی باشند، یک حلقه به وجود می آید.

تعریف چرخه یا Cycle: چنانچه در یک گراف از یک گره شروع به حرکت نموده و پس از پیمودن چندین لبه، مجدداً به همان گره برسیم، یک چرخه را طی نموده ایم.

تعریف لبه موازی یا Parallel Edge: چنانچه میان دو گره، بیش از یک لبه وجود داشته باشد، بدون توجه به جهت های احتمالی موجود، همگی موازی تلقی می شوند.

دسته بندی گراف ها بر حسب وجود یا عدم چرخه:

۱- گراف چرخه دار یا Cyclic Graph: گرافی است که در آن حداقل یک چرخه وجود داشته باشد.

۲- گراف بدون چرخه یا Acyclic Graph: گرافی است که در آن هیچ چرخه ای یافت نشود.

دسته بندی گراف ها بر حسب وجود یا عدم لبه موازی:

۱- گراف ساده یا Simple Graph: گرافی است که در آن هیچ لبه موازی وجود نداشته باشد.

۲- گراف چندگانه یا Multi Graph: گرافی است که در آن حداقل دو لبه موازی وجود داشته باشد.

تعریف گره تک (منفرد) یا Isolated Node: گره ای است که به آن هیچ لبه ای متصل نشده باشد.

تعریف گراف تهی یا Null Graph: گرافی است که در آن اصولاً هیچ لبه ای وجود ندارد.

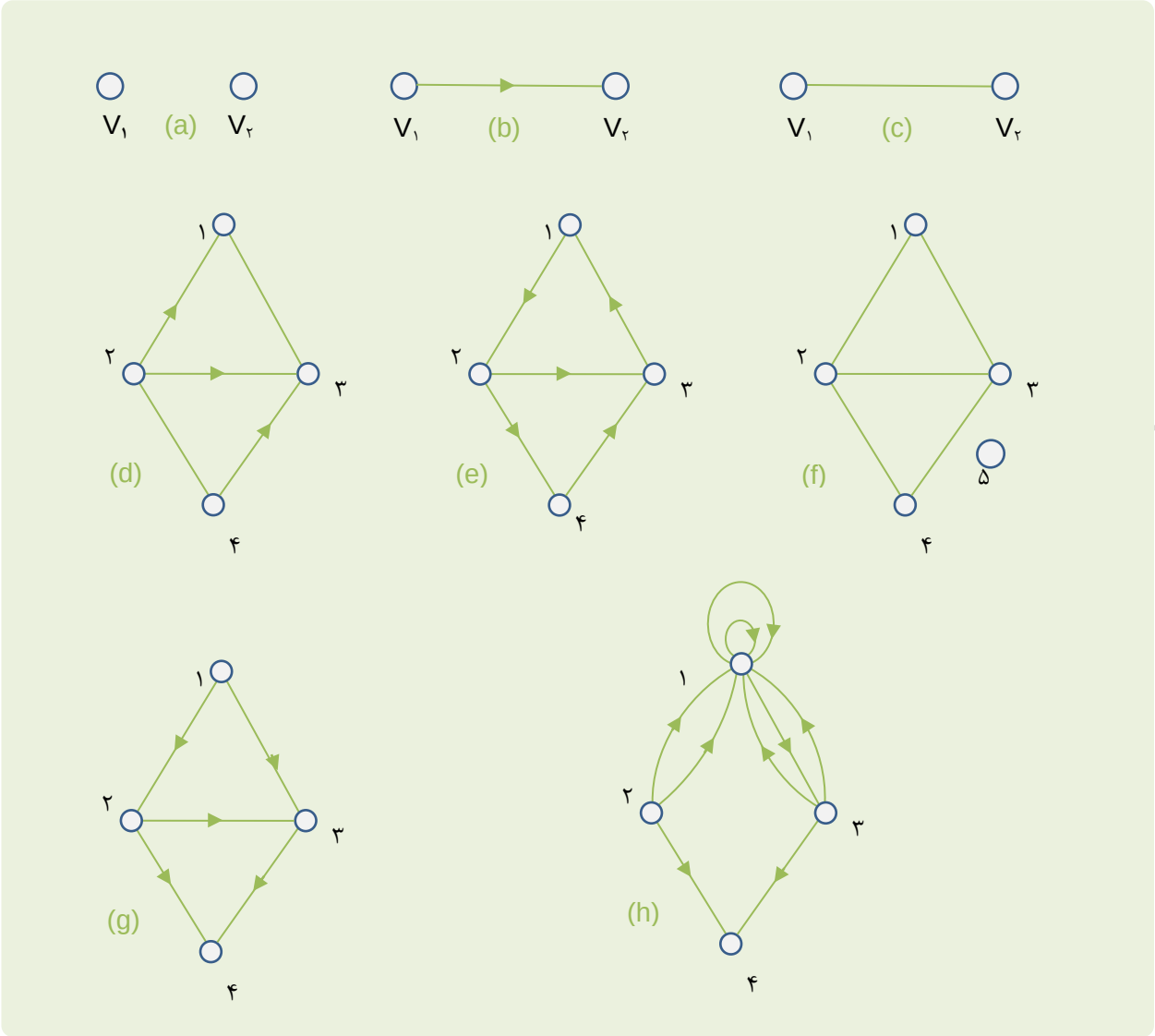
تعریف درخت یا Tree: یک گراف جهت دار بدون چرخه ی ساده یا A Simple Acyclic Digraph

تعریف درجه ورودی یا In Degree: به تعداد لبه های وارد شونده به یک گره درجه ورودی آن گره می گویند.

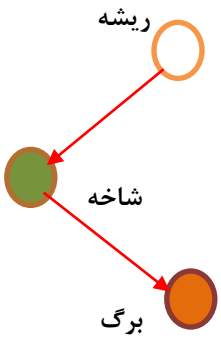
تعریف درجه خروجی یا Out Degree: به تعداد لبه های خارج شونده از یک گره درجه خروجی آن گره می گویند.

تعریف درجه کل یا Total Degree: به مجموع لبه های وارد شونده به یک گره یا In Degree و خارج شونده از یک گره یا Out Degree، درجــه کـــل آن گره می گویند.

نمونه هایی از انواع گرافها



- انواع گره در درخت

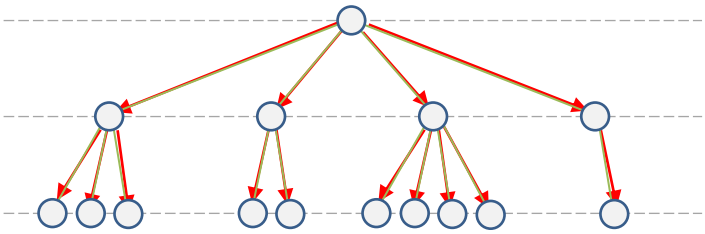


- ریشه (Root) : $outdegree \geq 0$ $indegree = 0$
- شاخه (Branch) : $outdegree > 0$ $indegree > 0$
- برگ (Leaf) : $outdegree = 0$ $indegree > 0$

- درخت m تایی یا m-ary Tree: درختی است که در آن رابطهٔ زیر حاکم است :

$$0 \leq \text{outdegree}\left(\text{هر گره}\right) \leq m$$

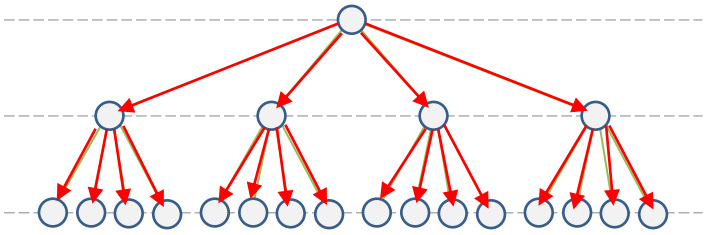
مثال : $\mathbf{m = 4}$ درخت ۴ تایی



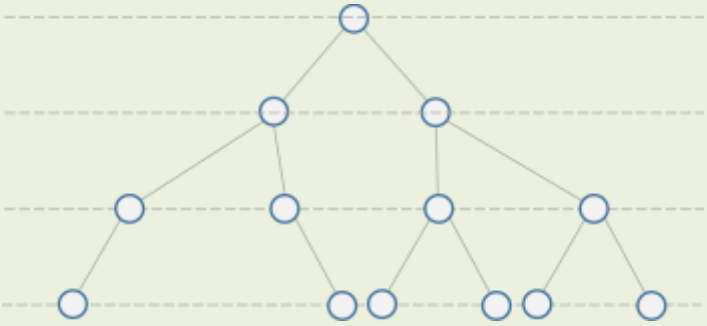
—

- مفهوم سطح (**Level**) : فاصلهٔ هر گره تا ریشه به سطح معروف است. بنابراین ریشه در سطح صفر یا ۰ **Level**، عناصر زیرین آن سطح یک یا ۱ **Level**، و به همین ترتیب الی آخر می باشد که هیچ محدودیتی در تعداد سطح ها وجود ندارد. فقط باید دقت داشت که عناصر هم سطح را در یک خط یا یک سطح رسم نمود.

- درخت **m** تایی کامل یا **Full m-ary Tree**: درختی است که در آن ***Out Degree*** هر گره یا برابر صفر یا **m** است.
- **مثال :** $\mathbf{m = 4}$ درخت ۴ تایی کامل



- **درخت دودویی (Binary Tree): درختی است که در آن m = ۲ می باشد و درخت دودویی کامل درختی است که درجه خروجی تمام گره های آن یا صفر و یا ۲ می باشد. ضمناً در رسم شکل با توجه به آنکه جهت تمام لبه ها از ریشه به سمت برگ ها می باشد، نیاز به رسم جهت نیست.**



◆ درخت دودویی مرتب (ایجاد)

در ایجاد یک درخت دودویی و چینش اطلاعات در آنها در واقع به طور کلی، هیچ قاعده و قانونی می تواند وجود نداشته باشد. اما به دلیل استفاده ای که می توانیم از نحوه خاص چینش اطلاعات و قانونمند شدن آن در زمان بازیابی اطلاعات ببریم، این کار را انجام داده و یک درخت دودویی مرتب را با توجه به قوانین زیر بوجود می آوریم:

برای ایجاد یک درخت دودویی مرتب، با توجه به تعدادی داده مفروض، به ترتیب ورود داده ها، به شرح زیر عمل می نماییم:

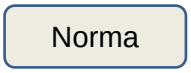
۱- داده وارد شونده اول را به عنوان ریشه قرار می دهیم.

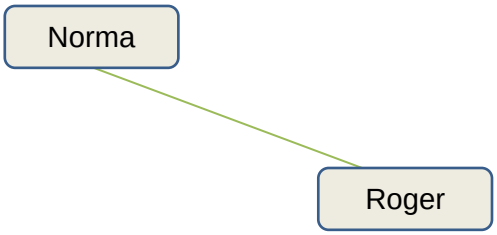
۲- داده های وارد شونده بعدی را به ترتیب ورودشان، ابتدا با ریشه و سپس با عناصر سطوح بعدی مقایسه می نماییم. چنانچه از ریشه کوچکتر یا مساوی آن باشند، در سمت چپ و چنانچه از ریشه بزرگتر باشند، در سمت راست درج می نماییم.

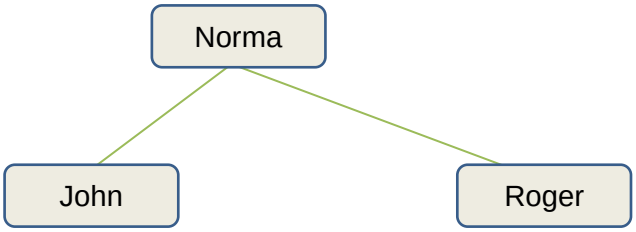
مثال: داده های رشته ای زیر مفروض می باشند، یک درخت دودویی مرتب با آنها می سازیم:

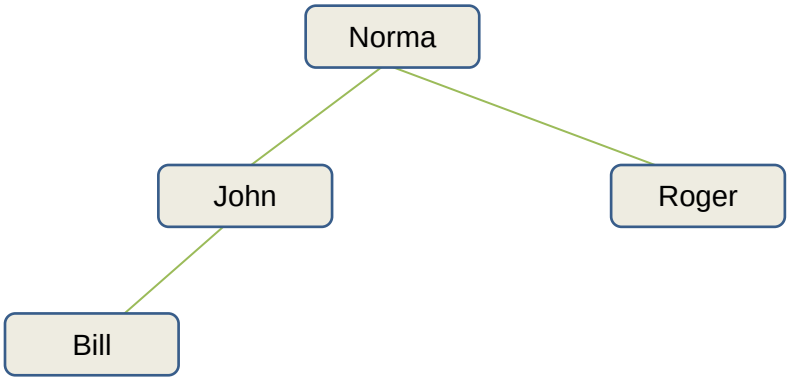
Norma – Roger – John – Bill – Leo – Paul – Ken – Maurice

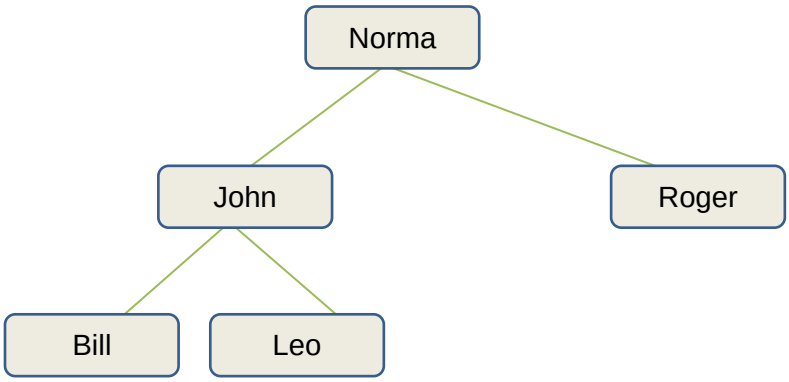
A-B-C-D-E-F-G-H-I-J-K-L-M-N-O-P-Q-R-S-T-U-V-W-X-Y-Z

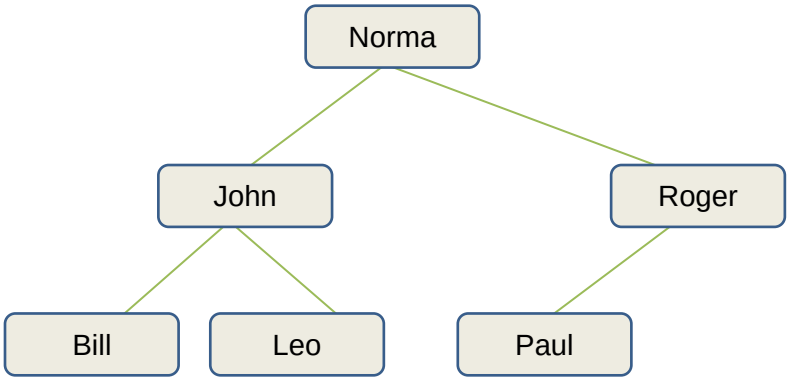


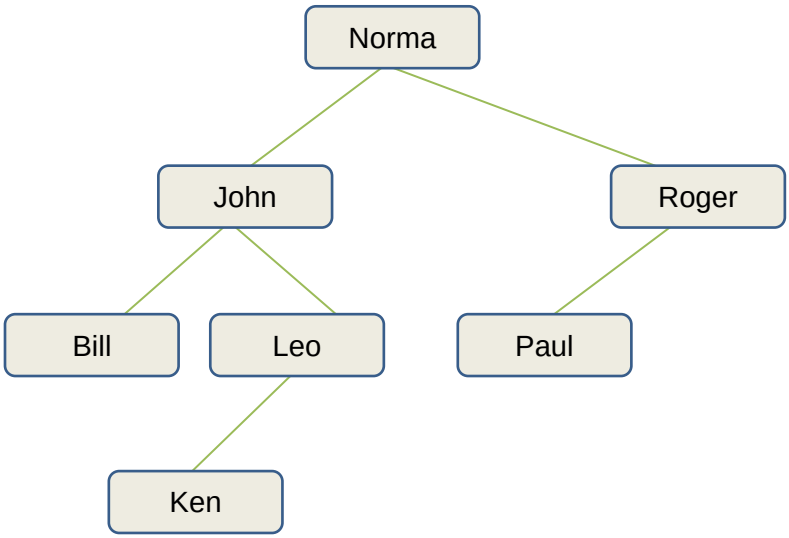


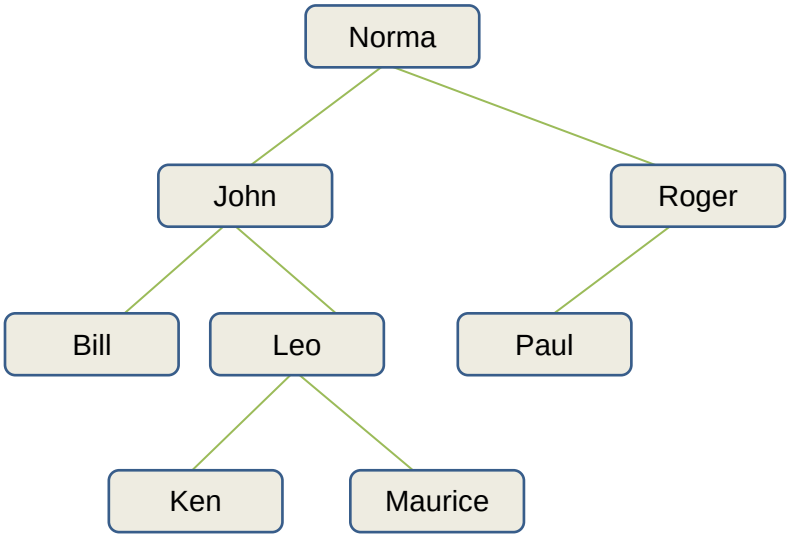





Norma
John
Roger
Bill
Leo


Norma
John
Roger
Bill
Leo
Paul


Norma
John
Roger
Bill
Leo
Paul
Ken



هم اکنون داده ها در درخت دودویی به شکل مرتب قرار گرفتند. حال بحث بازیابی اطلاعات و اصطلاحاً پیمایش یا Traverse درخت دودویی مطرح است و ببینیم استفاده ای که می توانیم از نحوه خاص چینش اطلاعات و قانونمند شدن آن در زمان بازیابی اطلاعات ببریم، چیست.

ابتدا سه روش از مشهورترین روش ها را در رابطه با پیمایش درخت دودویی بررسی می نماییم:

اگر فرض کنیم یک درخت دودویی شکلی ساده همچون درخت زیر را دارد که شامل یک ریشه (Node یا N)، یک زیرشاخه چپ (Left یا L)، و یک زیرشاخه راست (Right یا R) می باشد، اگر اطلاعات موجود در N قبل از L و R نوشته شود، اصطلاحاً Preorder پیمایش شده، اگراطلاعات موجود درN پس از L و R نوشته شود، اصطلاحاً Postorder پیمایش شده، و اگر اطلاعات موجود در N بین L و R نوشته شود، اصطلاحاً Inorder پیمایش شده است.

۱- Preorder N L R

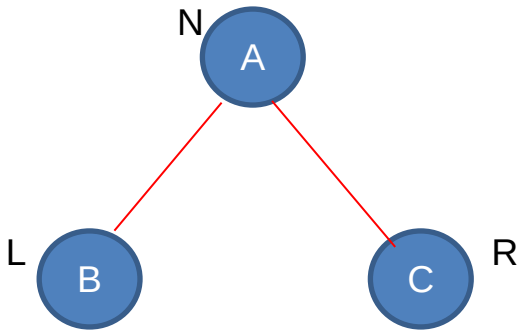
A B C

۲- Inorder L N R

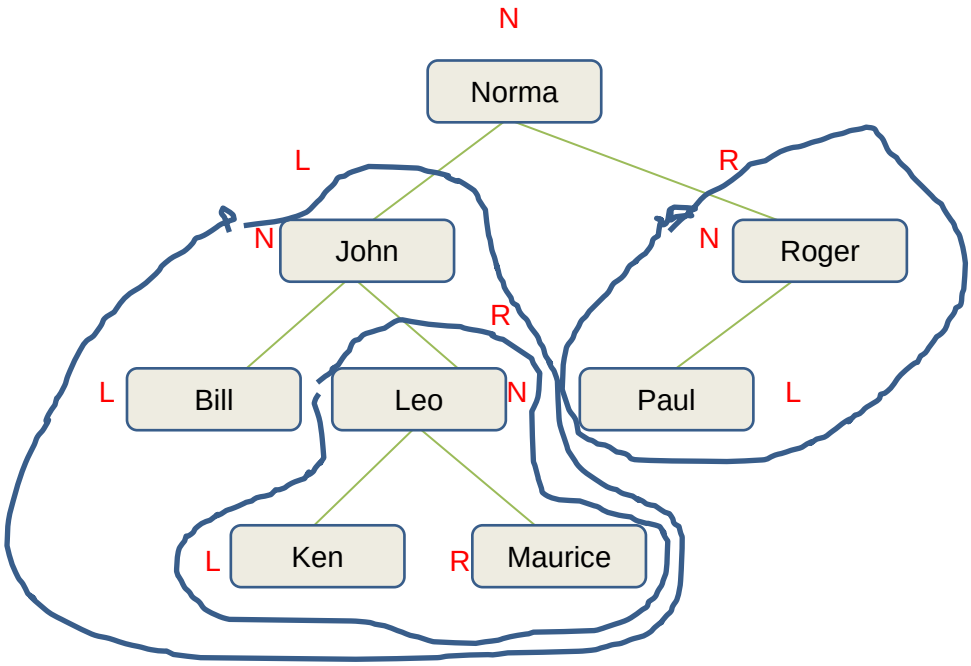
B A C

۳- Postorder L R N

B C A



اما همیشه درخت پیمایش شونده به سادگی و کوچکی درخت فوق نیست. در اینصورت بر روی درخت می بایست موارد N، L، و R را مشخص نمود و سپس به روش مورد نظر و دلخواه پیمایش کرد.



N L R: Norma, John, Bill, Leo, Ken, Maurice, Roger, Paul

L N R: Bill, John, Ken, Leo, Maurice, Norma, Paul, Roger

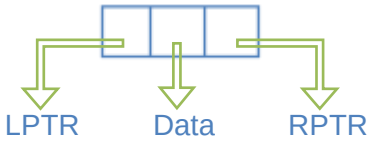
L R N: Bill, Ken, Maurice, Leo, John, Paul, Roger, Norma

اکنون اگر به حاصل پیمایش Inorder یا L N R درخت دودویی مرتب فوق دقت کنیم، می بینیم که اطلاعات مرتب و Sort شده ی صعودی است. این، همان استفاده ای بود که می توانیم از نحوه خاص چینش اطلاعات و قانونمند شدن آن در زمان بازیابی اطلاعات ببریم، یعنی مرتب سازی اطلاعات غیر خطی. حال این سؤال پیش می آید که پس چرا حاصل پیمایش به روش Inorder درخت دودویی ساده ی ابتدای بحث، مرتب نیست؟ پاسخ آن است که آن درخت، دودویی هست، اما مرتب نیست. لذا پیمایش به روش Inorder یک درخت دودویی مرتب منجر به مرتب سازی به Sort صعودی اطلاعات می گردد.

جلسهٔ ۱۰

الگوریتم های پیمایش :

قبل از آنکه به الگوریتم های پیمایش بپردازیم، ابتدا نحوه شبیه سازی یا ساخت یک درخت دودویی را در حافظه اصلی بررسی می نماییم. برای هر عنصر درخت، از یک Structure شبیه به لیست پیوندی دوجهته استفاده می کنیم:



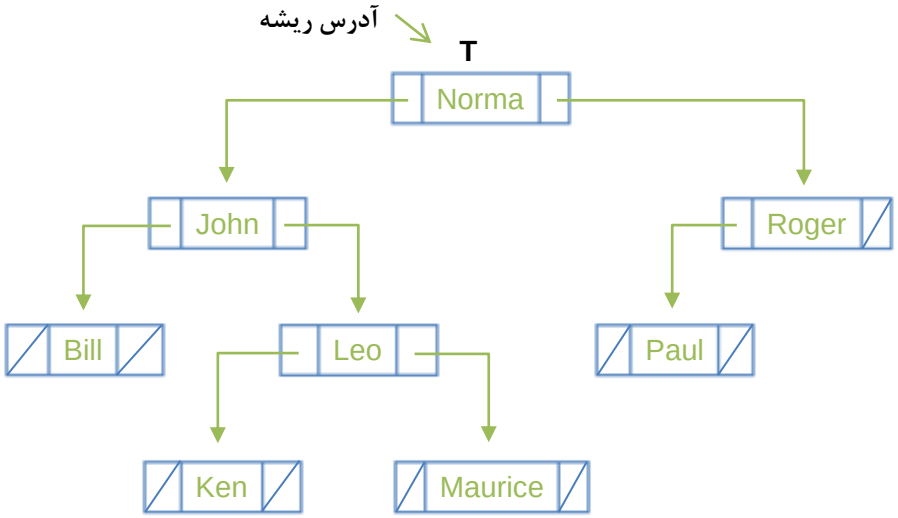
LPTR: حاوی آدرس زیر شاخه چپ

Data: حاوی اطلاعات

RPTR: حاوی آدرس زیر شاخه راست

نقطه ی مهم در ساختار داده درخت (Tree)، آدرس ریشه است که برای حفظ آن طبق معمول از یک متغیر کمکی با نام مناسب T استفاده می کنیم.

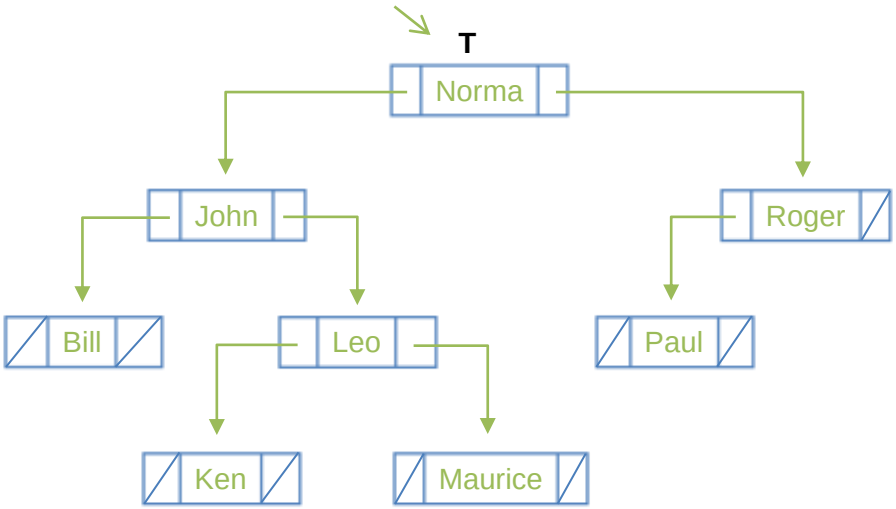
T: حاوی آدرس ریشه **T=Null: درخت تهی**



لازم به ذکر است که در سه الگوریتم مربوط به روش های Preorder، Inorder و Postorder، از ساختار داده پشته به عنوان ساختار داده ی کمکی بهره می گیریم و در آن آدرس هایی که بعداً قرار است مورد بررسی قرار گیرند را حفظ می کنیم.

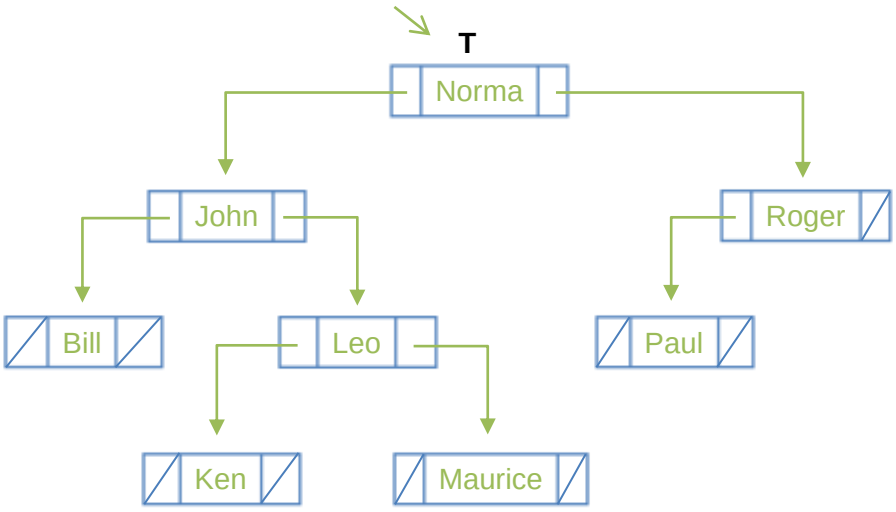
پیمایش درخت دودویی به روش Preorder یا NLR:

```
Procedure Preorder(T)
۱-  [Initialize]
    IF T=Null
    Then Write('Empty Tree')
         Exit
     Else  TOP ← ۰
         Call Push(S, TOP, T)
۲-  [Process each stacked branch address]
    Repeat thru Step ۳ While TOP > ۰
۳-  [Get Stored address and branch Left]
          P ← POP (S, TOP)
          Repeat while P ≠ Null
                Write(Data(P))
                IF RPTR(P) ≠ Null
                Then Call Push(S, TOP, RPTR(P))
                P ← LPTR(P)
۴-  [Finished]
    Exit
```



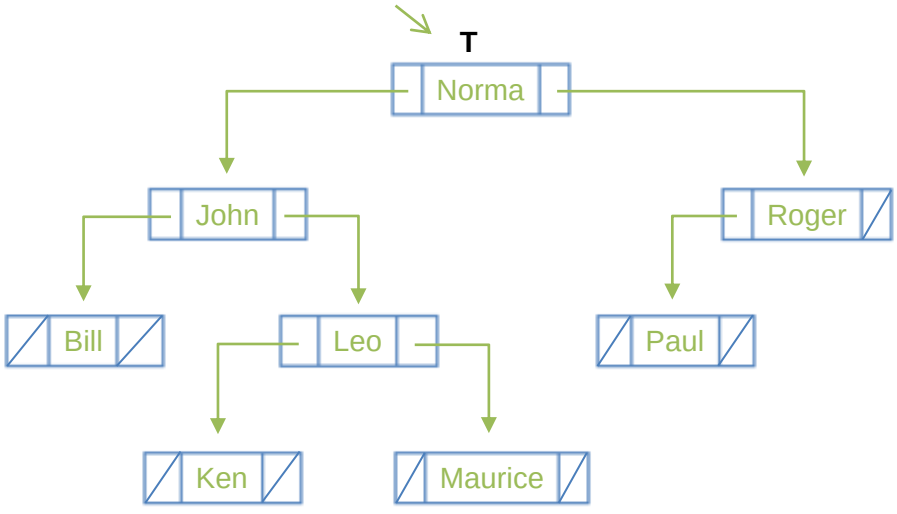
پیمایش درخت دودویی به روش Inorder یا LNR:

```
Procedure Inorder (T)
۱-  [Initialize]
    IF T = Null
    Then Write ('Empty Tree')
            Exit
     Else TOP ← ۰
            P ← T
۲-  [Traverse in Inorder]
    Repeat thru Step ۴ while True
۳-  [Descend Left]
            Repeat While P ≠ Null
                    Call Push (S, TOP, P)
                    P ← LPTR(P)
۴-  [Process Node and branch Right]
            Repeat While  P = Null
                    IF TOP = ۰
                    Then Exit
                    Else  P ← POP (S, TOP)
                        Write(Data(P))
                        P ← RPTR(P)
```



پیمایش درخت دودویی به روش Postorder یا LRN:

```
Procedure Postorder (T)
۱-  [Initialize]
    IF T = Null
    Then Write ('Empty Tree')
         Exit
     Else Top ← ۰
          P ← T

۲-  [Traverse in Postorder]
    Repeat thru Step ۵ While True
۳-  [Descend Left]
          Repeat While P ≠ Null
                Call Push (S, TOP, P)
                P ← LPTR(P)
۴-  [Process a Node whose Left and Right is traversed]
          Repeat While S[TOP] < ۰
                P ← POP (S, TOP)
                Write (Data(P))
                IF TOP = ۰
                Then Exit
۵-  [Branch Right and then mark Node]
          P ← RPTR (S[TOP])
          S[TOP] ←  - S[TOP]
```



تمرین: روش پیمایش سطح به سطح یا Level Order یک درخت دودویی بدین ترتیب است که عملیات از سطح صفر (ریشه) آغاز وسطح به سطح از چپ به راست ادامه یافته، اطلاعات موجود بازیابی می شوند. الگوریتمی طراحی کنید که یک درخت دودویی مفروض را به این روش پیمایش نماید. لازم به ذکر است که در این الگوریتم می توان از ساختار داده صف یا Queue به عنوان ساختار کمکی بهره برد. (حاصل پیمایش: Norma- John- Roger- Bill- Leo- Paul- Ken- Maurice است)