آموزشکده فنی شهید رجایی کاشان

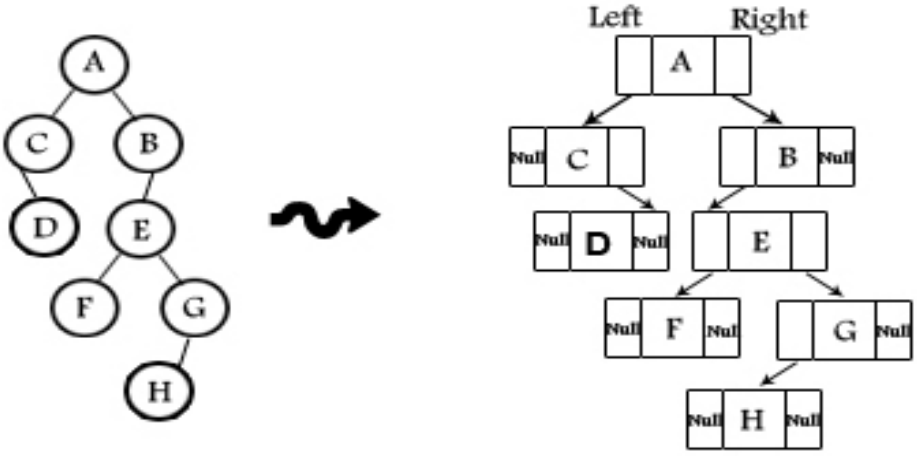
جزوه درس

ساختمان داده ها

تهیه و تنظیم:

مهندس محمدی

گروه کامپیوتر



تابستان ۸۷

سرفصل مطالبی که در این درس مورد بررسی قرار می گیرد:

فصل اول: بررسی پیچیدگی زمانی یک برنامه

- Big Oh
- Omega
- Teta

فصل دوم:آرایه ها

- **آرایه ها و آدرس نسبی**
- **کاربرد آرایه ها**
- **ماتریس اسپارس**
- **نحوه نمایش ماتریس اسپارس و بالامثلثی و پایین مثلثی**

فصل سوم: پشته و صف

- **پیاده سازی پشته به کمک آرایه ها**
- **کاربرد های پشته**
- **Infix-Postfix-Prefix ارزیابی عبارات**
- **پیاده سازی صف به کمک آرایه ها**
- **کاربردهای صف**

فصل چهارم: لیست های پیوندی

- **لیست پیوندی ساده**
- **پیاده سازی پشته به کمک لیست پیوندی**
- **پیاده سازی صف به کمک لیست پیوندی**
- **کاربردهای لیست پیوندی و مقایسه با آرایه**
- **لیست پیوندی دو طرفه**
- **لیست پیوندی حلقوی**
- **لیست عمومی و پیمایش آن**

فصل پنجم: درخت ها

- **تعریف ها ، درخت عمومی و دودویی و تبدیل آنها**
- **نمایش درخت در حافظه**
- **درخت دودویی پر و کامل**
- **پیمایش درخت دودویی**
- **درخت دودویی نخی**
- **Heap هرم و کاربردهای آن**
- **درخت جستجوی دودویی**
- **جنگل و تبدیل جنگل به درخت دودویی**

فصل ششم: گراف ها

- **گراف و تعاریف اولیه**
- **کاربردهای گراف**
- **نمایش گراف در حافظه(ماتریس مجاورتی – لیست مجاورتی)**
- **پیمایش گراف(عمقی – سطحی)**
- **گراف متصل و نقاط بحرانی**
- **درخت پوشا**
- **درخت پوشای مینیمم والگوریتمهای آن**

فصل هفتم: مرتب سازی و جستجو(تحلیل)

- **مرتب سازی درجی**
- **مرتب سازی سریع**
- **مرتب سازی ادغامی**
- **مرتب سازی مبنایی**
- **مرتب سازی هرمی**
- **مرتب سازی درختی**

دانشجوی گرامی این جزوه فقط برای کمک به شما در یادگیری بهتر درس و بهتر ارائه شدن درس تهیه شده است و شامل همه مطالب درس نیست. لذا حضور در کلاس و یادداشت برداری از مطالبی که ارائه می شود ضروری است.

نحوه تجزیه و تحلیل و سنجش کارایی یک برنامه

میزان حافظه یا پیچیدگی فضای یک برنامه مقدار حافظه مورد نیاز برای اجرای کامل یک برنامه است. مقدار زمان یا پیچیدگی زمانی یک برنامه مقدار زمانی از کامپیوتر است که برای اجرای کامل برنامه لازم است.

میزان حافظه (پیچیدگی حافظه)

فضای مورد نیاز برای یک برنامه شامل موارد زیر است:

- بخش ثابت که مستقل از بعضی خصیصه های ورودی و خروجی مانند تعداد و اندازه است. این بخش شامل فضای دستورالعمل (کد برنامه) ، فضای لازم برای متغیرهای ساده و ثابت ها می باشد.
- بخش متغیر که شامل فضای مورد نیاز متغیرهای ساختاری برنامه که اندازه آن بستگی به نمونه مسئله ایی که حل می شود، دارد و فضای لازم برای متغیرهای مرجع که اندازه آنها نیز به خصیصه های نمونه بستگی دارد و فضای لازم برای پشته بازگشتی می باشد.

پیچیدگی زمانی

زمان برنامه (T(P)) مجموع زمان کامپایل و زمان اجرای برنامه است. زمان کامپایل برای یک برنامه فقط یک بار رخ می دهد در ضمن به خصیصه های نمونه بستگی ندارد. بنابراین زمان اجرا یک برنامه Tp مورد مورد بررسی قرار می گیرد.

برای بدست آوردن زمان اجرای برنامه راه های مختلفی وجود دارد مثلا تعداد جمع ها و ضرب ها و تقسیمات و تفریقات یک برنامه **را بد**ست آوریم و زمان را به صورت زیر محاسبه کنیم:

$$Tp= C_1ADD(n) + C_2Sub(n) + C_3Mul(n) + C_4Div(n)$$

بدست آوردن چنین فرمول دقیقی یک کار غیر ممکن است زیرا زمان مورد نیاز برای جمع و ضرب و تفریق و تقسیم به عوامل متعددی بستگی دارد. در ضمن بین کامپیوتر های مختلف نیز متعدد است.

روش دیگری که برای بدست آوردن پیچیدگی زمانی یک برنامه استفاده می شود این است که تعداد مراحل برنامه را بشماریم. پیچیدگی زمانی یک برنامه را بر اساس تعداد مراحل آن بدست می آوریم.

برای بدست آوردن تعداد مراحل یک برنامه با توجه به نوع دستورات عمل می کنیم. مثلا دستورات انتساب را یک مرحله محسوب می کنیم. دستورات تکرار را با توجه به تعداد تکرارشان محاسبه می کنیم. دستورات تعریفی را جزء مراحل محسوب نمی کنیم. دستور شرط نیز یک مرحله محسوب می شود.

تعداد مراحل فراخوانی یک تابع مساوی مجموع تعداد مراحل فراخوانی هر تابع است.

یک از روشهایی که برای محاسبه تعداد مراحل یک برنامه استفاده می شود استفاده از متغیری به نامCount است که در ازای هر مرحله یکی به آن اضافه می شود.

مثال)

```
Float Sum(Float *A , Const int n)
{
  Float S=0;   Int i;
    Count ++;
  For (I=0 ; I<n ; I++) {
      Count ++;
       S+ = A[ I ];
     Count++;
   }
Coumt++;  // For Last time of for
Return (S);
Count++;
}
```

```
Float Sum(Float *A , Const int n)
{
  Float S=0;   Int i;
   For (I=0 ; I<n ; I++)
      S+ = A[ I ];
   Return (s);
}
```

که پیچیدگی زمانی آن 2n+3 است.

مثال) برنامه جمع دو ماتریس

```
For ( I=0 ; I<m ; I++)
  For  ( j=0 ; j<n ; j++)
    C[I][j]=A[I][j]+B[I][j];
```

```
For ( I=0 ; I<m ; I++){
   Count++;
  For  ( J=0 ; J<n; J++) {
    Count++;
    C[I][j]=A[I][j]+B[I][j];
     Count++;
   }
  Count++; // For Last time of for
}
```

که پیچیدگی زمانی آن 2mn+2m+1 می شود.

البته این روش نیز چندان دقیق نیست. مثلا ما دستور A:=5*2+4 –B رایک مرحله می گیریم و A:=8 را نیز یک مرحله می گیریم در صورتیکه از نظر زمانی با هم متفاوتند.

تعداد مراحل لزوما پیچیدگی جمله را منعکس نمی کند.

ولی در یک برنامه بزرگ می توان تعداد مراحل را با کمی اغماض پیچیدگی برنامه دانست و می توان برنامه ها را براساس پیچیدگی زمانی آنها با هم مقایسه کرد.

روش دیگر روش S/e نام دارد که شباهت زیادی به روش Count دارد.

تمرین: چرا پیچیدگی الگوریتم فیبوناچی 4n+1 است؟

هدف از تعیین تعداد مراحل این است که بتوانیم پیچیدگی زمانی دوبرنامه که عمل مشابهی را انجام می دهند مقایسه کنیم و همچنین بتوانیم میزان افزایش زمان اجرا را وقتی مشخصات نمونه تغییر می کنند پیشگویی کنیم.

مثلا اگر پیچیدگی زمانی برنامه ایی 2n باشد و پیچیدگی زمانی برنامهٔ دیگری n2 است کدام یک سریعتر است؟

البته هر دوبرنامه یک الگوریتم است.

در مورد پیچیدگی زمانی یک برنامه تعاریف ریاضی(نشانه گذاری مجانبی) وجود دارد که در اینجا به اختصار بحث می شود تا بتوانیم عبارات با معنی(ولی غیردقیق) در مورد پیچیدگی زمانی یک برنامه بدست آوریم.

(O) Big Oh

$f(n) = O(g(n))$ است اگر مقادیر ثابتی مانند C و n_0 باشد که رابطه زیر برقرار باشد:

$f(n) \le C\, g(n)$ برای تمام $n \ge n_0$

$$f(n) = O(g(n)) \Leftrightarrow \exists C, n_0 \rangle 0 \quad \forall n \ge n_0 \quad |f(n)| \le C|g(n)|$$

مثال) فرض کنید f(n)=3n+2 باشد که دراین صورت اگر n_0=2 باشد و C=4 باشد و g(n)=n رابطه زیر برقرار است

$3n+2 \le 4\, n$

پس 3n+2=O(n)

البته می توان g(n) های دیگری نیز پیدا کرد که این خاصیت را داشته باشند.

$3n+2 \le 2n$

$C=1$ و $n_0=4$

ولی معمولا تابعی را به عنوان g(n) در نظر می گیرند که از بقیه کوچکتر باشد $n \le n2$

$f(n)=O(g(n))$ فقط نشان می دهد که $g(n)$ حد بالای مقدار $f(n)$ است برای تمام $n\geq n0$

قضیه: اگر $f(n)=a_mn^m+\ldots+a_1n+a_0$ در این صورت $f(n)=O(n^m)$

امگا(Ω)

$f(n)=\Omega(g(n))$ اگر به ازای مقادیر ثابت C و n_0 رابطه $f(n)\geq C.g(n)$ برای $n\geq n_0$ همیشه برقرار باشد.

مثال) مثلا $3n+2\geq 3.n$ است در ازای $n\geq 1$ $3n+2=\Omega(n)$

$10n2+4n+7\geq n2$ در ازای $n\geq 1$ یعنی $10n2+4n+7=\Omega(n2)$ البته می توان نوشت که $10n2+4n+7=\Omega(n)$

اما معمولا تابعی را در نظر می گیرند که از بقیه بزرگتر باشد.

عبارت $f(n)=\Omega(g(n))$ فقط نشان می دهد که $g(n)$ یک حد پایین برای تابع $f(n)$ است.

قضیه: اگر $f(n)=a_mn^m+\ldots+a_1n+a_0$ در این صورت $f(n)=\Omega(n^m)$

تتا(θ)

$f(n)=\theta(g(n))$ است اگر به ازای مقادیر $c1$ و $c2$ و n_0 رابطه زیر درست باشد

$c1.g(n)\leq f(n)\leq c2.g(n)$ در ازای $n\geq n_0$

قضیه: اگر $f(n)=a_mn^m+\ldots+a_1n+a_0$ در این صورت $f(n)=\theta(n^m)$

نشانه گذاری تتا از امگا و Big oh دقیق تر است زیرا هم بعنوان کران بالا و هم بعنوان کران پایین $f(n)$ می باشد.

مثال) جمع دو ماتریس:

حال حداکثر پیچیدگی زمانی را به عنوان پیچیدگی در نظر بگیریم یعنی $\theta(Rows.Cols)$.

```
Void   ADD(int  *a)
{
 int  I , j;
For   (I=0 ;  I <Rows ; I++)     //➔   θ(Rows)
  For  (j=0 ; j<Cols ; j++)      // ➔    θ(Rows.Cols)
   C[I][j]=a[I][j]+b[I][j]   //➔ θ(Rows.Cols)
}
```

بوسیله این پیچیدگی ها می توان دو برنامه که عمل یکسانی را انجام می دهند با هم مقایسه کرد.

مثلا برنامه ایی که $O(n^2)$ است و برنامه ایی که $O(2^n)$ را با هم مقایسه می کنیم. اگر $n=10$ باشد $n^2=100$ و $2^n=1024$ می شود.

تمرین : برنامه ضرب دو ماتریس $n*n$ را نوشته و پیچیدگی زمانی آنرا بدست آورید .

تمرین: برنامه بدست اوردن سری فیبوناچی تا عدد کمتر از N بررسی کنید و پیچیدگی زمانی آن را بدست آورید.

```
K=0;
For (i=0;i>0;i=i/2)
    K++;
```

```
k=0;
For (i=0;i<n;i=i*2)
    K++;
```

این حلقه ها لگاریتمی هستند و دارای واحد زمانی **O(log n)** هستند.

```
K=0;
For (i=0;i<n;i++)
    For (j=I;j>0;j=j/2)
        K++;
```

```
K=0;
For (i=1;i<=n;i=i*2)
    For (j=1;j<=i;j++)
        K++;
```

مثال) مرتبهٔ اجرائی الگوریتم زیر را محاسبه کنید:

```
i=n;
while (i>1)  {
      i=i/2;      → O(log_2^n)
      j=i;
      while (j>1)
        j=j/3;  → O(log_3^n)
  }
```

چون دو حلقهٔ تودر تو داریم Order کل به صورت حاصلضرب دو Order محاسبه می شود

$$O\left(\log_2^n * \log_3^n\right)$$

آرایه ها:

از دیدگاه برنامه نویسان آرایه مجموعه ایی از خانه های پشت سر هم در حافظه است. ولی همیشه این طور تصور نمی شود.
اگر آرایه را به عنوان یک ADT در نظر بگیریم مجموعه ای از زوج های <index , Value> است طوری که یک تناظر یک به یک بین اندیس و محتوای سلول دارد.
این تعریف به وضوح مشخص می کند که آرایـه یک ساختـار کلی تراز " یک مجموعه از داده های پشت سرهم در حافظه" است.همچنین درموردترکیب مجموعه اندیس ها قابل انعطاف تر است.
آرایه را می توان یک ADT با عمل های Stroe , Retrieve تعریف کرد به صورت زیر:

```
#include <stdio.h>
#define  Max  1024
class Array{
  float  Ar[Max];
  int end;
  public:
   Array(int n){
                    end=n-1;
                  }
  void  store(int i, int value)
    {
     if (i>end){
           printf("Error in index");
           return;}
      else
            Ar[i]=value;
     }
  float  Retrieve(int i)
    {
      if (i>end){
            printf("Error in index");
            return(0);
           }
           else
            return(Ar[i]);
    }
} ;
```

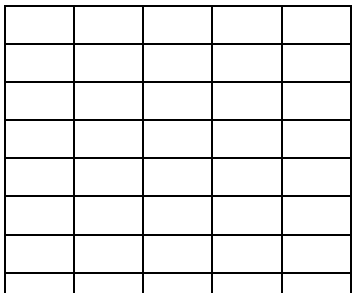
وقتی در برنامه ای به نوعی داده نیاز باشد که در آن زبان وجود ندارد برنامه نویس باید نوع مورد نظرش را ایجاد کند. نوع داده ای را که برنامه نویس ایجاد می کند نوع داده انتزاعی می گویند.
ADT یک مدل ریاضی است که عملیاتی را بر روی آن مدل تعریف می کند. هرنوع داده متشکل از چند مقدار و مجموعه ای از عملیات بر روی آنها است.مانند نوع داده int که در زبان C عملیاتهایی مانند > < / = - + * و غیره برای آنها تعریف شده است.

نمایش آرایه های چند بعدی:

آرایه های چند بعدی معمولا با ذخیره عناصر در یک آرایه یک بعدی پیاده سازی می شوند.
به عنوان مثال ماتریس را می توان یک آرایه یک بعدی در نظر گرفت.

0,0	**0,1**	**0,2**	**0,3**	**1,0**	**1,1**	**1,2**	**1,3**	**2,0**	**2,1**	**2,2**	**2,3**

آرایه بالا نشان دهنده یک ماتریس **۳×۴** است.



A[0][0]= A+0+0*5
A[0][1]= A+1+0*5
A[0][2]= A+2+0*5
A[0][4]= A+4+0*5
A[1][0]= A+0+1*5
A[1][1]= A+1+1*5

A[1][2]= A + 2 + 1*5
A[1][3] = A+ 3 + 1*5
A[2][4] = A+ 4 + 2*5
A[6][2] = A+ 2 + 6*5

A[X][Y] = Base + Y +X*N
N ➔ Number of Columns

فرض می‌کنیم آرایهa در حافظه به صورت سطری ذخیره شده است:

$$Var\ a : array[1\ldots m][1\ldots n]\ of\ type$$

اگر بخواهیم آدرس خانهٔ $[i, j]$ ام را در آرایهٔ a در حافظه پیدا کنیم می‌توان از فرمول زیر استفاده کرد

$$\boxed{Addr\{a[i, j]\} = \alpha + \{(i-1)*n + (j-1)\}*t}$$

این روش ذخیره‌سازی در زبان‌های برنامه‌نویسی مانند c و پاسکال بکاررفته است.

آرایهٔ سه بعدی

فرض کنید آرایه سه بعدی زیر را داشته باشیم : در اینجا با صفحه هایی سروکار داریم که در راستای فلش نشان داده شده قرار گرفته‌اند وبه ترتیب نشان داده شده در شکل زیر ذخیره می‌شوند:

فرض می‌کنیم آرایهٔ a در حافظه به صورت سطری ذخیره شده است:

$$Var\ a : array[1\ldots m][1\ldots n][1\ldots p]\ of\ type$$

اگر بخواهیم آدرس خانهٔ $[i, j, k]$ام را در آرایهٔ a در حافظه پیدا کنیم می‌توان از فرمول زیر استفاده کرد

$$\boxed{Addr\{a[i, j, k]\} = \alpha + \{(i-1)*n*p + (j-1)*p + (k-1)\}*t}$$

فرض می‌کنیم آرایهٔa در حافظه به صورت ستونی ذخیره شده است:

اگر بخواهیم آدرس خانهٔ $[i, j, k]$ام را در آرایهٔ a در حافظه پیدا کنیم می‌توان از فرمول زیر استفاده کرد

$$\boxed{Addr\{a[i, j, k]\} = \alpha + \{(k-1)*m*n + (j-1)*m + (i-1)\}*t}$$

مثال ۱)

فرض کنید آرایهٔ زیر که هر عنصر آن 4 بایت طول دارد به شکل ستونی از آدرس 20000 حافظه ذخیره شده است ؛ آدرس خانهٔ زیر را بدست آورید:

$$M : array[1\ldots 30, 1\ldots 20, 1\ldots 10]\ of\ \ type$$

$$Addr\{M[21,11,9]\} = 20000 + \{(9-1)*30*20 + (11-1)*30 + (21-1)\}*4$$
$$Addr\{M[21,11,9]\} = 20000 + \{4800 + 300 + 20\}*4 = 40480$$

نمایش چند جمله ایی ها بوسیله آرایه: $a_mX^m+a_{m-1}X^{m-1}+...+a_0$

جمع چند جمله ایی ها:

مشتق یک چند جمله ایی:

ماتریس اسپارس Sparce Matrix

ماتریسی است بسیار بزرگ (مثلا ۱۰۰۰*۱۰۰۰) که تعداد زیادی از عناصر آن مثل هم است.به همین دلیل حافظه زیادی به هدر می رود.مثلا فرض کنید یک ماتریس ۱۰۰*۱۰۰ که ۵۰ عنصر غیر صفر دارد و مابقی آن صفر است.
اگر این ماتریس مانند ماتریس های دیگر در حافظه ذخیره شود حافظه بسیار زیادی به هدر می رود.

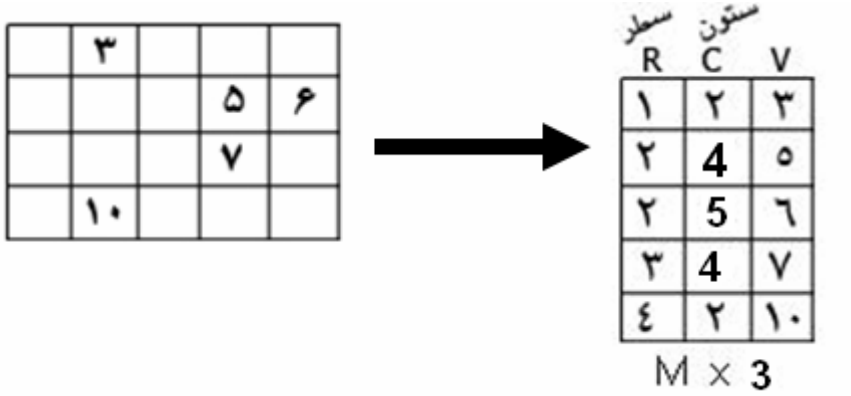
کاربرد ماتریس اسپارس

تصویر در کامپیوتر یک ماتریسی است که هر عنصر آن در واقع کد رنگ پیکسلی است.برای ذخیره کردن عکسهای رادیولوژی که بیشتر عکس سیاه است یک ماتریس اسپارس حاصل می شود که حافظه زیادی را به هدر می دهد.
برای رفع این مشکل می توان ماتریس اسپارس را به روش m*3 ذخیره کرد.

روش M * 3

در این روش برای ذخیره کردن ماتریس اسپارس از یک ماتریس M * ۳ استفاده می کنیم (M تعداد عناصر غیر صفر است).

مثال)



0	3	5	0	0	17	0	11	4	0
6	0	0	14	0	0	0	0	0	0
0	4	0	0	2	15	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	19	0	0	0	13	0
1	9	0	0	0	0	11	0	0	0
0	0	0	0	0	0	16	0	20	0
6	7	0	12	0	0	0	0	0	15
0	0	0	0	0	0	1	0	0	0
0	9	0	0	12	0	0	20	0	18

R	C	V
0	1	3
0	2	5
0	5	17
0	7	11
0	8	4
1	0	6
1	3	14
2	1	4
2	4	2
2	5	15
4	4	19
4	8	13
5	0	1
5	1	9
5	6	11
6	6	16

ماتریس

R	C	V
0	1	6
0	5	1
1	0	3
1	2	4
1	5	9
2	0	5
3	1	14
4	2	2
4	4	19
5	0	17
5	2	15
6	5	11
6	6	16
7	0	11
8	0	4
8	4	13

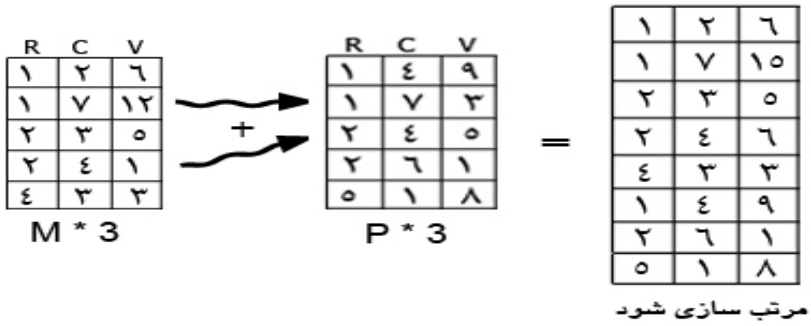
ترانهاده ماتریس

معایب روش 3 * M

۱- به علت ایستا بودن آرایه ها عدد M (تعداد عناصر غیر صفر) از همان ابتدا باید مشخص باشد.

۲- به دلیل اینکه شکل ظاهری ماتریس تغییر کرده الگوریتم های مربوط به ماتریس تغییر خواهد کرد.

جمع دو ماتریس اسپارس



نکته: حاصل جمع دو ماتریس اسپارس دارای پیچیدگی زمانی O(M*P) است در صورتی که پیچیدگی زمانی دو ماتریس N*N برابر O(N^2) است (که N نیز معمولا عدد بزرگی است) .

تمرین: برنامه ای بنویسید که دو ماتریس اسپارس حداکثر 10 عضوی غیر صفر را دریافت و ذخیره و سپس حاصل جمع آن را نمایش دهد.

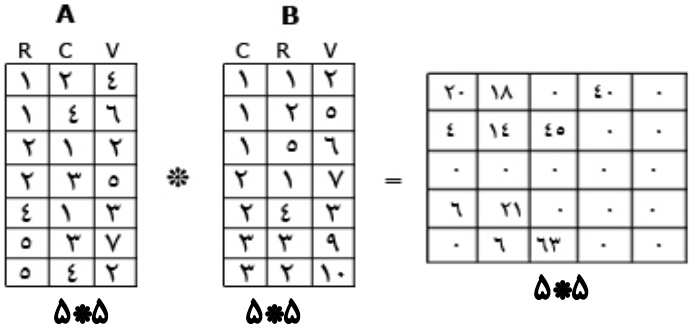
ترانهاده یک ماتریس

برای محاسبه ترانهاده یک ماتریس ۳ * M جای Cو R را عوض می کنیم و سپس آن را دوباره مرتب می کنیم.

ضرب دو ماتریس اسپارس

برای ضرب دو ماتریس اسپارس A , B (A*B) ابتدا ترانهاده ماتریس B را محاسبه کرده تا ماتریس B بر اساس ستونی مرتب شود سپس عناصر متناظر (مساوی) A را در B ضرب می کنیم و آنها که متناظر نیستند صفر در نظر گرفته می شود.

نکته: ممکن است حاصل ضرب دو ماتریس اسپارس دیگر اسپارس نشود.



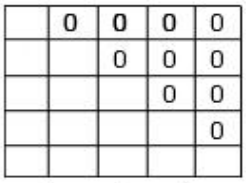
اگر ماتریس اسپارس را به صورت آرایه دو بعدی ذخیره کنیم برای جابجا کردن سطرها و ستونها زمانی معادل O(Rows.Cols)نیاز است که بسیار زیاد است.

اما اگر ماتریس اسپارس به صورت سه گانه ها (سطری) ذخیره شود با یک الگوریتم مناسب می توان ترانهاده آن را محاسبه کرد که دارای پیچیدگی زمانی O(Terms.Columns) خواهد بود. Terms نشان دهنده تعداد عناصر غیر صفر در ماتریس اسپارس است.

حاصل ضرب دو ماتریس از ضرب سطرها در ستونها بدست می آید. اگر دوماتریس اسپارس A,B را به صورت سه گانه ها (سطری)ذخیره کنیم برای محاسبه A*B ابتدا ترانهاده ماتریس B را به روشی که بحث شد بدست آوریم حال می توانیم

سطرها را در ستونها ضرب کنیم. دلیل این کار این است که اگر برای پیدا کردن ستون J در ماتریس B باید کل سه گانه ها پویش (جستجو) شود اما وقتی ترانهاده B بدست می آید عناصر ستون J کنار هم قرار می گیرند.

ماتریس پایین مثلثی و بالا مثلثی



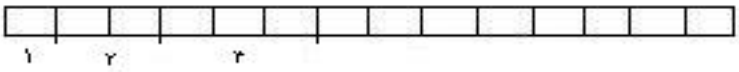
ماتریس پایین مثلثی

نکته: ماتریس های پایین مثلثی و بالا مثلثی همیشه مربعی هستند.

ماتریس بالا مثلثی ماتریسی است که عناصر بالای قطر اصلی آن غیر صفر باشد و عناصر پایین قطر اصلی صفر باشد.

برای ذخیره کردن این ماتریس که باعث اتلاف حافظه می شود به این روش عمل می کنیم :

اگر ماتریس دارای n تا سطر و ستون باشد تعداد عناصر پر آن برابر $n(n+1)/2$ خواهد بود که برای ذخیره کردن آن می توانیم از یک آرایه یک بعدی به اندازه $n(n+1)/2$ استفاده کنیم.

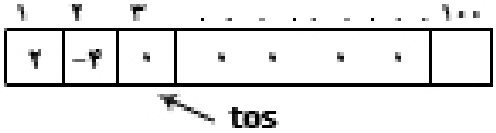


پیاده سازی stack (پشته) به کمک آرایه ها

Stack ساختمان داده ای است که حالت LIFO دارد یعنی آخرین ورودی اولین خروجی است.

به عمل ریختن اطلاعات داخل stack , push و به عمل برداشت اطلاعات از آن pop گفته می شود.برای پیاده سازی یک stack به کمک آرایه یک متغیر tos تعریف می کنیم که همیشه عنصر بالای stack را تعیین می کند و به دو زیر برنامه push , pop نیاز داریم که در زیر برنامه push ابتدا tos یک واحد افزایش می یابد(به شرط اینکه stack پرنباشد) و سپس یک عنصر اضافه می کنیم.

در زیر برنامه pop ابتدا عنصر را خارج کرده و سپس tos را یک واحد کمک می کنیم.



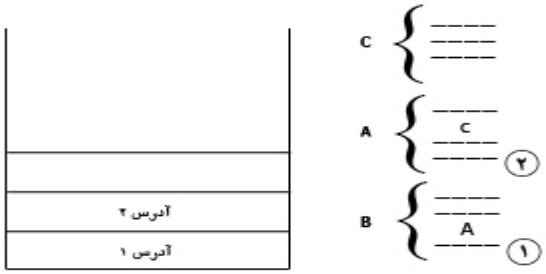
عیب این روش آن است که چون از آرایه اسفاده می شود حجم پشته محدود است.

مزیت این روش آن است که زیر برنامه های pop , push دارای پیچیدگی زمانی O(1) هستند و بسیار سریع عمل می کنند.

کاربرد های stack

یکی از مهمترین کاربردهای stack در زبانهای برنامه نویسی فراخوانی زیر برنامه هاست.

هنگامی که یک تابع فراخوانی می شود آدرس فعلی در stack , push می شود و بعد از اتمام اجرای آن زیر برنامه برای برگشت به محل اولیه آن آدرس از stack برداشته می شود.



برنامه های بازگشتی از stack برای بازگشت به عقب استفاده می کنند.

نکته: در بیشتر الگوریتم ها که نیاز به حفظ مسیر است(جهت برگشت) از stack استفاده می شود.

Prefix – infix – postfix

۱- روش میانوندی(infix)

در این روش عملگر وسط قرار گرفته و عملوند ها سمت چپ و راست قرار می گیرند.

مثال) A+B

۲- روش پسوندی (postfix) :

در این روش عملگر بعد از عملوند ها ظاهر می شود.

مثال) AB+

۳- روش پیشوندی (prefix) :

در این روش عملگر قبل از عملوند ظاهر می شود.

مثال) +AB

در زیر مثال هایی از هر سه نوع عبارت نوشته شده است.

1) Infix= A-B*C postfix= ABC*- prefix= -A*BC

---

2) Infix= (A-B)*C postfix= AB-C* prefix= *-ABC

عبارتهای prefix , postfix دو مزیت نسبت به infix دارند:

۱- در عبارات prefix , postfix جهت به دست آوردن حاصل نیازی به دانستن اولویت ها نداریم.

۲- در عبارات prefix , postfix پرانتز وجود ندارد.

بیشتر زبان های برنامه نویسی عبارت infix از برنامه نویس تحویل میگیرند و خود به postfix یا prefix تبدیل می کنند.

الگوریتم تبدیل infix به postfix

۱-روش ایجاد پرانتز بندی کامل :

در این روش ابتدا عبارت infix را پرانتز بندی کامل می کنیم (بر اساس اولویت انجام) سپس از سمت چپ عبارت به دست آمده کراکتر به کراکتر عملیات زیر را انجام می دهیم:

الف) اگر کراکتر به دست آمده پرانتز باز است هیچ عملی انجام نمی شود.

ب) اگر کراکتر مورد نظر عملوند است خودش را چاپ می کنیم.

ج) اگر کراکتر مورد نظر عملگر است آن را داخل یک پشته push می کنیم.

د) اگر کراکتر مورد نظر پرانتز بسته است از پشته pop کرده و چاپ می کنیم.

مثال)عبارات infix زیر را به Prefix تبدیل کنید.

عبارت infix	شکل پرانتز بندی	عبارت Prefix
A+(B*C)/(M-H)+F	((A +((B*C)/(M-H)))+F)	++A/*BC-MHF
A+B/((D+E)*(A-C))	(A+(B/((D+E)*(A-C))))	+A/B*+DE-AC

پیاده سازی صف به کمک آرایه ها

صف ساختمان داده ای است که به صورت FIFO کار میکند یعنی هر عنصری که اول وارد شود اول نیز خارج می شود. برای پیاده سازی صف به کمک آرایه راه های زیادی وجود دارد.

روش اول :

در این روش یک متغیر به نام rear تعریف می کنیم که در ابتدا صفر است(یعنی صف خالی است). برای اضافه کردن یک عنصر ابتدا rear را یک واحد افزایش و سپس داخل صف می ریزیم و برای خارج کردن از صف عنصر داخل اندیس 0 را برداشته و بقیه داده های داخل صف را شیفت به چپ می دهیم. rear نیز یک واحد کم می شود. در این روش پیچیدگی زمانی خارج کردن از صف O(N) است.

روش دوم :

در این روش دو متغیر rear , front را برای نمایش ابتدا و انتهای صف در نظر می گیریم.

ابتدا rear = front = -1 است. بنابراین شرط خالی بودن صف مساوی بودن rear , front است و شرط پر بودن صف rear = n است(n تعداد عناصر آرایه است). تعداد عناصر موجود در صف برابر rear – front است.

تعداد عناصر خالی موجود در صف برابر n-(rear – front) است. مشکل اصلی این روش این است که متغیر های rear , front فقط به سمت جلو حرکت می کنند و به همین دلیل عناصر آرایه فقط یک بار استفاده می شوند. پیچیدگی زمانی این روش در add , delete برابر با O(1) است.

صف حلقوی

در این صف ابتدا rear = front = 0 است. در این صف هرگاه rear به انتهای آرایه برسد و front نیز صفر نباشد rear دوباره صفر می شود یعنی سلول های آرایه یکبار مصرف نیستند.

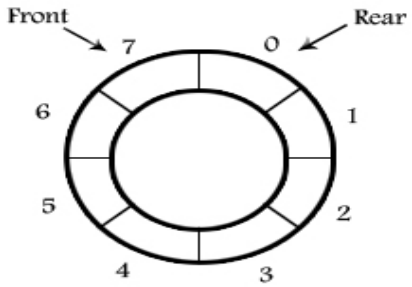
شرط خالی بودن این صف این است که front =rear باشد.

شرط پر بودن صف این است که front = (rear+1) mod n باشد.

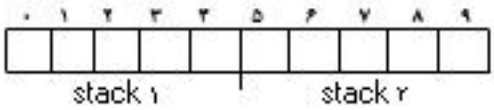
نکته: در صف حلقوی در هر لحظه از n تا سلول آرایه حداکثر n-1 عنصر می تواند مقدار دهی شود زیرا بین حالت پر و خالی باید اختلافی وجود داشته باشد یعنی اگر n تا عنصر داخل صف بریزیم آنگاه rear = front شده که در این صورت به نظر می رسد که صف خالی باشد.

نکته: پیچیدگی زمانی اضافه و حذف در صف حلقوی O(1) است.

عیب صف حلقوی این است که تعداد عناصر آن محدود است زیرا از آرایه ها استفاده می کنیم.



پشته های چندگانه



می توانیم از یک آرایه برای ۲ پشته استفاده کنیم که در ابتدا آرایه را به دوقسمت مساوی تقسیم کرده و از آن استفاده می کنیم.

اگر پشته ۱ پر شود و پشته ۲ عنصر خالی داشته باشد می توانیم پشته ۱ را کمی گسترش دهیم. این روش پشته ها را تا حدی پویا می کند در عوض پیچیدگی زمانی برنامه را به دلیل شیفت دادن زیاد می کند.

الگوریتم تبدیل infix به postfix به کمک پشته

۱- شروع
۲- یک token از عبارت infix خارج کن.
۳- اگر token=')' آن گاه آن را در پشته درج کن و برو به ۲
۴- اگر token = "عملوند" آن گاه آن را در خروجی چاپ کن و برو به ۲
۵- اگر token = "عملگر" آن گاه تا زمانی که عملگرهای بالای پشته از عملگر , اولویت یکسان یا بیشتری دارند آن ها را pop کن و چاپ کن سپس عملگر را در پشته درج کن و برو به ۲
۶- اگر token = '(' آن گاه تا زمانی که به ')' نرسیده ایم از پشته pop کن و چاپ کن. فقط ')' را از پشته خارج می کنیم ولی چاپ نمی کنیم. برو به ۲
۷- اگر token = NULL آن گاه تا زمانی که پشته خالی نشده pop کن و چاپ کن.
۸- پایان

تمرین: برنامه ای بنویسید که:

الف) یک عبارت infix تک رقمی در یافت کندو prefix آن را نمایش دهد.

ب) عبارت prefix به دست آمده را حساب کند.

ج) پرانتز بندی کامل infix آن را به دست آورید. (توضیح: به کمک prefix حاصل میتوان پرانتز بندی را انجام داد)

محاسبه عبارت postfix

از رشته postfix هربار یک token را خارج کرده و اگر token = عملوند بود آن را داخل یک پشته push می کنیم اما اگر token = عملگر بود دو عملوند از پشته pop کرده و عملیات مربوطه را انجام می دهیم. آخرین عددی که داخل پشته می ماند جواب عبارت است.

نکته: به کمک این روش میتوان معتبر بودن یه عبارت postfix را چک کرد.

کاربردهای صف:

در سیستم عامل های چند کاره (multi task) هم زمان چندین برنامه در حال اجرا هستند در صورتی که یک cpu بیشتر نداریم. سیستم عامل از یک صف برای زمان بندی فرآیندها استفاده می کند. در واقع هر فرآیند که می خواهد اجرا شود وارد یک صف می شود. هر بار سیستم عامل یک فرآیند را خارج کرده و تحویل cpu می دهد و برای مدت زمان کوتاهی اجرا می شود. دوباره در صورت پایان نیافتن برنامه , برنامه به انتهای صف اضافه می شود تا زمانی که برنامه کارش تمام شود.

یکی دیگر از کاربردهای صف در پرینتر می باشد که طبق صف یکی یکی چاپ می کند.

نمونه سئوال تست:

۱-اگر a=2 و b=2 و C=3 و d=4 باشد آنگاه ارزش عبارت پسوندی **ab*c+dc-/** کدام است؟

الف)7 ب)6 ج)5 د)4

۲-اگر رشته اعداد ۱و ۳و ۴و۵و۷را به ترتیب از سمت راست وارد یک Stack کنیم کدامیک از خروجیهای زیر این Stack امکان پذیر نیست؟(از سمت چپ)

⟵

الف)7 5 4 3 1 ب)1 3 7 5 4 ج)1 7 3 5 4 د)1 4 3 7 5

۳-کدام یک از تساوی های زیر درست است؟

الف) $5n^2-6n=O(n^3)$ ب) $5n^2-6n=\theta(n^3)$ ج) $5n^2-6n=\Omega(n^3)$ د) $5n^2-6n=\theta(n)$

۴-اگروفقط اگر به ازای مقادیر ثابت و مثبت C و n_0 داشته باشیم $f(n) >= c.g(n)$ به ازای همه $n>n_0$ آنگاه:

الف) $f(n)=O(g(n))$ ب) $f(n) =\theta(g(n))$ ج) $f(n)= \Omega(g(n))$ د) $g(n) = \Omega(f(n))$

۵-کدام یک از تساوی های زیر درست نیست؟

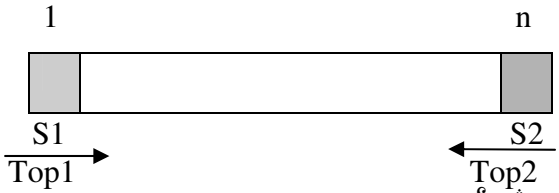
الف) $3^n=O(2^n)$ ب) $n^2 = O(2^n)$ ج) $n!=O(n^n)$ د) $2n^2+n\text{Log } n =\theta(n^2)$

۶-پیچیدگی زمانی الگوریتم روبرو کدام گزینه است؟

الف) $O((m+1)/2)$ ب) $O(m^2)$

ج) $O(m.\log m)$ د) $O(mk)$

۷-دو پشته را به وسیله یک آرایه پیاده سازی می کنیم به صورتی که در خلاف جهت یکدیگر رشد نمایند. شرط پر بودن دوپشته چیست؟



الف) Top1=Top2 ب) Top1 + 1 =Top2

ج) Top1>Top2 د) Top1>= Top2

۸-در نمایش صف حلقوی به کمک آرایه، چرا از یک خانه آرایه استفاده نمی شود؟

الف) اندیس خانه مزبور صفر است ب) به عنوان رزرو برای مواقع خاص نگه داشته می شود

ج)برای ارتباط خانه آخر با خانه اول باید از یک خانه استفاده کنیم

د) درصورت استفاده، پر وخالی بودن صف با یکدیگر اشتباه می شود

۹- تابع ACK به صورت روبرو تعریف شده است مقدار ACK(1,1) برابر است با:

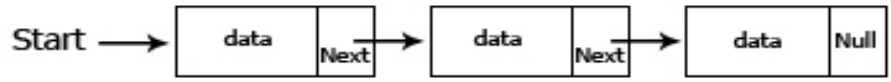
```
int  ACK(int m  ,  int   n)
{
if   (m<0  ||  n<0) return  0;
else  if (m = = 0) return n+1 ;
else  if (n = = 0) return ACK(m-1,1);
else  return ACK(m-1 , ACK(m , n-1));
}
```

الف) 3 ب) 4

ج) 5 د) 6

لیست پیوندی ساده

لیست پیوندی از تعدادی گره که منطقا پشت سر هم قرار گرفته اند تشکیل شده است. آدرس ابتدای این لیست در یک اشاره گر به نام start ذخیره می شود مانند شکل زیر :



نکته: ساختمان داده هر گره حداقل از دو فیلد تشکیل شده است. یک فیلد به نام data که داده گره و فیلد دیگر اشاره گری به نام next است که آدرس گره بعدی را نگه داری می کند.

نکته: فیلد next آخرین گره NULL است.

نکته: یکی از کاربرد های لیست پیوندی در ساختار فایل سیستم عامل است.

هر فایل در واقع مجموعه ای از گره هاست که به صورت یک لیست پیوندی بر روی حافظه جانبی ذخیره می شود و آدرس ابتدای این لیست پیوندی در جدول FAT ذخیره می شود.

مزایای لیست پیوندی نسبت به آرایه

۱- لیست پیوندی پویا است ولی آرایه ها پویا نیستند.

۲- لیست پیوندی از فضای حافظه به صورت بهینه استفاده می کند.

معایب لیست پیوندی نسبت به آرایه

۱- دسترسی به عناصر آرایه بسیار سریعتر از دسترسی به عناصر لیست پیوندی است.

۲- الگوریتم هایی مانند مرتب سازی و جستجو در مورد لیست های پیوندی با مشکلاتی مواجه است.

مثال) فرض کنید یک لیست پیوندی ساده با ابتدای start داریم. می خواهیم داده x را در آن جستجو کنیم. برنامه ای بنویسید که این کار را انجام دهد.

```
Struct node
{
  int data;
  Struct node * next;
};
Struct node * start;
Int search(struct node * start)
   {
    struct node * temp;
    Temp=start;
    Int find=0;
    While (temp !=NULL)
         {
         if (temp->data == x)
                  {
                  Find=1
                  Break;
                  }
         Temp=temp->next;
         }
 Return(find);
 }
```

مثال) برنامه ای بنویسید که یک لیست پیوندی ساده ایجاد و ۳ گره به آن اضافه کند.

```
#include <stdio.h>
#include <stdlib.h>
Struct node
   {
    Int data;
    Struct node * next;
   };
Struct node * start;
Void main()
{
Start=(struct node *)malloc(sizeof(node));
Start->data=200;
Start->next=NULL;

Start->next = (struct node *)malloc(sizeof(node));
Start->next->data=100;
Start->next->next=NULL;

Start->next->next = (struct node *)malloc(sizeof(node));
Start->next->next->data=170;
Start->next->next->next=NULL;
}
```

مثال) برنامه ای بنویسید که به کمک یک لیست پیوندی یک stack (پشته) را شبیه سازی کند.

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <conio.h>
Struct node
   {
    Int data;
    Struct node * next;
   };
Struct node * start;
//*****************************
Void push()
{
If (start==NULL)
        {
        Start=(struct node *)malloc(sizeof(node));
        Cout<<"enter a number";
        Cin>>start->data;
        Start->next=NULL;
        }
Else
        {
        Struct node * temp;
        Temp=(struct node *)malloc(sizeof(node));
        Temp->next=start;
        Start=temp;
        Cout<<"enter a number";
        Cin>>start->data;
        }
}
//*****************************
```

```
Void list()
{
Struct node * temp;
Temp=start;
If (temp==NULL)
        Cout<<"stack is empty";
Else
        While (temp!=NULL)
                {
                Cout<<temp->data<<"   ";
                Temp=temp->next;
                }
}
//*****************************
Void pop()
{
Struct node * temp;
If (start==NULL)
        Cout<<"stack is empty";
Else
        {
        temp=start;
        Cout<<start->data<<endl;
        Start=start->next;
        Free(temp);
        }
}
//*****************************
Void main()
{
Char c;
Start=NULL;
Clrscr();
Do
        {
        Printf("\nEnter A for push\n");
        Printf("\nEnter D for pop\n");
        Printf("\nEnter L for list\n");
        Printf("\nEnter Q for exit\n");
        Cin>>c;
        If (c=='a') push();
        If (c=='d') pop();
        If (c=='l') list();
        }
while (c!='q');
}
```

مثال) برنامه ای بنویسید که عنصری به آخر یک لیست پیوندی ساده اضافه کند.

```
Struct node * temp;
Temp=start;
If (start ==NULL)    اگر اولین عنصر بود
        {
        Start=(struct node *)malloc(sizeof(node));
        Cin>>start->data;
        Start->next=NULL
```

```
        }
Else
        {
        While (temp->next != NULL)
                Temp=temp->next;
        Temp->next=(struct node *)malloc(sizeof(node));
        Temp->next->next=NULL;
        Cin>>temp->next->data;
        }
}
```

مثال) برنامه ای بنویسید که گره آخر یک لیست پیوندی ساده را حذف کند.

```
Struct node * temp;
If (start ==NULL)
        Return;
If  (start->next == NULL)
        {
        Free(start);
        Start=NULL;
        Return;
        }
Temp=start;
While (temp->next->next != NULL)
        Temp=temp->next;
Free(temp->next);
Temp->next=NULL;
```

تمرین: برنامه ای بنویسید که با فرض وجود یک لیست پیوندی ساده عدد x را در گره ها جستجو کرده و در صورت پیدا شده آن گره را حذف کند.

تمرین: برنامه ای بنویسید که در یک لیست پیوندی موجود عدد x را جستجو کند. در صورت پیدا شدن گره جدیدی با محتوای y به جلوی آن اضافه کند.

پیاده سازی صف به کمک لیست پیوندی

برای این کار جهت اضافه شدن به صف یک گره به انتهای لیست پیوندی اضافه می کنیم و جهت برداشتن یا خارج کردن از صف یک گره از اول لیست پیوندی بر می داریم.

مثال) برنامه ای بنویسید که به کمک لیست پیوندی ساده صف را پیاده سازی کند.

```
#include  <stdio.h>
#include <iostream.h>
#include <stdlib.h>
#include <conio.h>
struct node{
     int data;
     struct node * next;
     };
struct node * start;
//*****************************
void ADD()
{
  if (start==NULL)
     {
          start=(struct node *)malloc(sizeof(node));
```

```
            cout<<"Enter A NUMBER  " ;
            cin>> start->data;
            start->next=NULL;
       }
   Else
            {
            struct node * temp;
            temp=start;
            while(temp->next!=NULL)
             temp=temp->next;
            temp->next=(struct node * ) malloc (sizeof(node));
            cout<< " Enter a NUMBER";
            cin>>temp->next->data;
            temp->next->next=NULL;
            }
}
//*****************************
void list()
{
 struct node * temp;
 temp=start;
 if (temp==NULL)
    cout<<"Queue is empty"<<endl;
 else
    while (temp!=NULL)
            {
             cout<<temp->data<<"   ";
             temp=temp->next;
            }
}
//*****************************
void del()
{
  struct node * temp;
  if (start==NULL)
     cout<<"Queue  Is Empty"<<endl;
  else
            {
            temp=start;
            cout<<start->data<<endl;
            start=start->next;
            free(temp);
            }
}
//*****************************
void main()
{
 char c;
 start=NULL;
      clrscr();
   do{
    printf("\nEnter A for ADD\n");
    printf("Enter D for delete \n");
    printf("Enter L for List \n");
    printf("Enter Q for Exit \n");
```

```
    cin>>c;
    if (c=='a') ADD();
    if (c=='l') list();
    if (c=='d') del();
  }while (c!='q');
}
```

مثال) برنامه ای بنویسید که با فرض وجود یک لیست پیوندی ساده یک کپی از آن ایجاد کند.

```
Struct node * temp1;
Struct node * temp2;
Struct node * s2;
Temp1=start;
S2=(struct node *)malloc(sizeof(node));
S2->data=temp->data;
Temp2=s2;
Temp1=temp1->next;
While (temp1 !=NULL)
        {
        temp2->next=(struct node *)malloc(sizeof(node));
        Temp2=temp2->next;
        Temp2->data=temp1->data;
        Temp1=temp1->next;
        }
Temp2->next=NULL;
```

تمرین: برنامه ای بنویسید که با فرض وجود ۲ لیست پیوندی مرتب لیست پیوندی سوم را از ادغام این دو لیست به دست آورد به طوری که مرتب باشد.

مثال) برنامه ای بنویسید که محتوای یک لیست پیوندی ساده را از آخر به اول نمایش دهد.

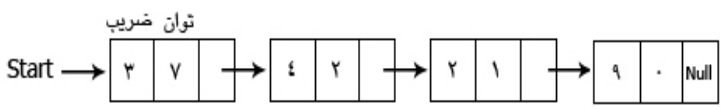
```
Void test(struct node * ptr)
{
if (ptr==NULL)
        Return;
Else
        {
        cout<<ptr->data;        {1}
        Test(ptr->next);        {2}
        }
}
```

نکته: برنامه بالا محتوای یک لیست پیوندی ساده را از اول به آخر نمایش می دهد. اگر جای خط {۱} و {۲} را عوض کنیم آن گاه محتوای لیست پیوندی از آخر به اول نمایش داده می شود.

تمرین: برنامه ای بنویسید که محتوای یک لیست پیوندی را از آخر به اول در لیست پیوندی دیگری کپی کند.

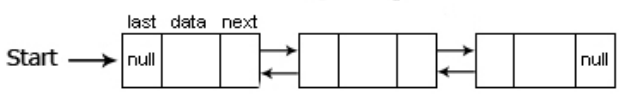
نکته: یکی از کاربردهای لیست پیوندی نمایش چند جمله ای های ریاضی است. به صورت شکل زیر:

$3x^7-4x^2+2x-9$



لیست های دو پیوندی (دو طرفه)

ساختار هر گره در لیست پیوندی دو طرفه بدین صورت است که هر گره آدرس گره قبلی و بعدی خود را دارد.



ساختار رکورد این نوع لیست در زبان c به صورت زیر است:

```
Struct node {
        Struct node * last;
        Int data;
        Struct node * next;
};
```

مثال) برنامه ای بنویسید که یک گره به ابتدای لیست دو پیوندی اضافه کند.

```
Temp=(struct node *)malloc(sizeof(node))
Temp->next=start;
Start->last=temp;
Temp->last=NULL;
Start=temp;
```

مثال) برنامه ای بنویسید که یک گره از اول لیست حذف کند.

```
Start=start->next;
Free(start->last);
Start->last=NULL;
```

نکته: برای حذف کردن یک گره لازم نیست که مانند لیست های پیوندی ساده روی گره قبلی توقف کنیم زیرا در خود گره نیز آدرس گره قبلی وجود دارد.

مثال) برنامه ای بنویسید که ابتدا یک لیست پیوندی دو طرفه ایجاد و سپس یک گره از وسط یک لیست دو پیوندی حذف کند.

```
#include <stdio.h>
#include <iostream.h>
#include <stdlib.h>
struct node{
   struct node * last;
   int data;
   struct node * next;
};
struct node * start;
void main()
{
  struct node * temp;
  start=(struct node *)malloc(sizeof(node));
  start->data=10;
  start->last=NULL;

  start->next=(struct node *)malloc(sizeof(node));
  temp=start->next;
  temp->last=start;
  temp->data=20;

  temp->next=(struct node *)malloc(sizeof(node));
  temp->next->last=temp;
  temp=temp->next;
  temp->data=30;

 temp->next=NULL;
 temp=start->next;
 temp->last->next=temp->next;
 temp->next->last=temp->last;
 free(temp);}
```

مثال) برنامه ای بنویسید که گره ای را به وسط یک لیست دو پیوندی اضافه کند.

```
Temp=(struct node *)malloc(sizeof(node));
Temp->last=ptr;
Temp->next=ptr->next;
Ptr->next=temp;
Ptr->next->last=temp;
```

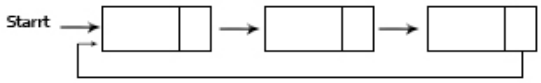
ایجاد لیست گره های پاک شده

برای بالا بردن سرعت برنامه های لیست پیوندی که به طور مکرر گره ها را حذف و درج می کنند می توانیم از تکنیک حذف منطقی استفاده کنیم.

در این تکنیک هر گره ای که قرار است free شود به جای free شدن آن را به یک لیست پیوندی دیگر منتقل می کنیم.(لیست سطل زباله)

هنگامی که می خواهیم عنصر جدیدی ایجاد کنیم به جای malloc , از لیست حذف شده ها عنصری را بر می داریم.

لیست پیوندی یک طرفه حلقوی



نکته: در این نوع لیست ها آخرین گره به اولین گره متصل است و در این نوع لیست ها NULL وجود ندارد.

در این لیست با داشتن آدرس هر گره دلخواه می توان به تمام گره ها دسترسی داشت و آن را پیمایش کرد که مزیت آن است.

مثال) برنامه ای بنویسید که یک لیست پیوندی حلقوی ساده را پیمایش کند.

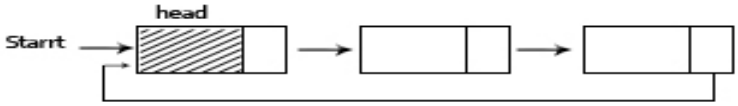
```
Void listH()
{
struct node * temp=start;
Do
        {
        Cout<<temp->data;
        Temp=temp->next;
        }
While(temp != start)
}
```

نکته: در لیست حلقوی , حذف و درج در ابتدای لیست مستلزم اصلاح کردن گره آخر است که این کار دارای پیچیدگی زمانی O(n) است.

نکته: برای حذف و درج یک عنصر بهترین جا بعد از گره اول یعنی گره دوم است زیرا نیازی به تغییر دادن گره آخر نیست.

نکته: تهی بودن لیست پیوندی حلقوی نمایشش با مشکل مواجه می شود.

برای رفع مشکلات حذف و درج در ابتدای لیست پیوندی حلقوی , یک گره کمکی به نام head در ابتدای لیست اضافه می کنیم.گره head داده ای ندارد و فقط برای رفع مشکل حلقوی استفاده می شود. در این صورت اولین گره داخل لیست, دومی می شود و مشکل تغییر دادن گره آخر حل می شود.



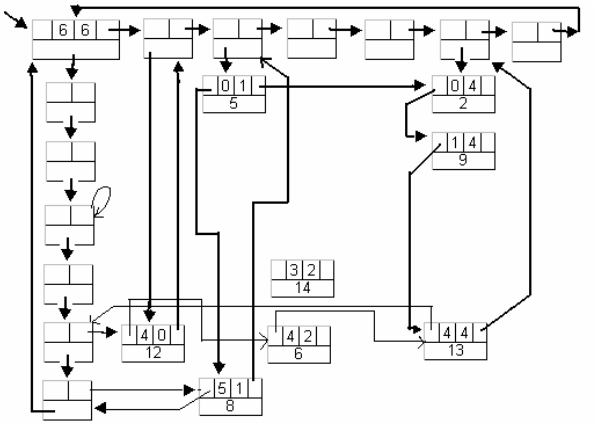
مثال) زیربرنامه ای بنویسید که با فرض وجود ۲ چند جمله ای مرتب شده در دو لیست s1 وs2 , حاصل جمع این دو چند جمله ای را به دست آورد.

```
Void add(struct node * s1 ,struct node * s2)
{
Struct node * temp1;    Struct node * temp2;
Struct node * ptr;          Struct node * s3;
temp1=s1;
temp2=s2;
int find;
S3=new    node;
ptr=s3;
While (temp1 !=NULL  &&   temp2 !=NULL )
    {
        If  (temp1 -> tavan = = temp2-> tavan )
            {
               Ptr->tavan = temp1 -> tavan;
               Ptr-> zarib= temp1-> zarib  +   temp2 -> zarib;
               Temp1 = temp1 -> next;
                Temp2 = temp2 -> next;
            }
          Else
              If (temp1-> tavan > temp2-> tavan)
                    {
                        Ptr->tavan = temp1 -> tavan;
                         Ptr-> zarib= temp1-> zarib ;
                         Temp1 = temp1 -> next;
                    }
               Else
                   {
                         Ptr->tavan = temp2 -> tavan;
                         Ptr-> zarib= temp2-> zarib ;
                         Temp2 = temp2 -> next;
                    }
           Ptr-> next = new   node ;
            Ptr= ptr- > next;
        }
```

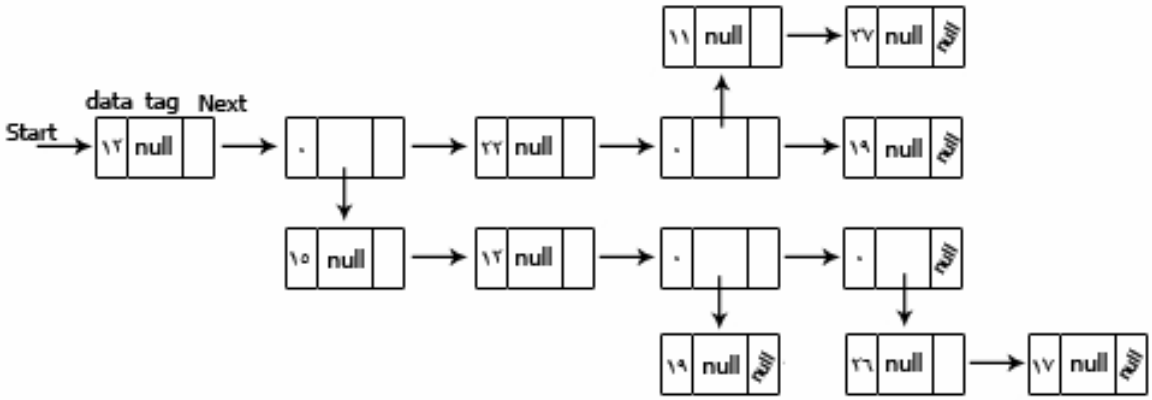
```
If  (temp1 !=NULL)
 {
   While (temp1 != NULL)
     {
       Ptr->tavan = temp1 -> tavan;
       Ptr-> zarib= temp1-> zarib ;
       Temp1 = temp1 -> next;
      }
If  (temp2 !=NULL)
 {
   While (temp2 != NULL)
     {
       Ptr->tavan = temp2 -> tavan;
       Ptr-> zarib= temp2-> zarib ;
       Temp2= temp2 -> next;
      }
Ptr->next = NULL;
}
```

نحوه نمایش ماتریس های اسپارس به کمک لیست پیوندی

می توانیم یک ماتریس اسپارس را به کمک لیست های پیوندی حلقوی نمایش دهیم که به تعداد ستون ها یک لیست پیوندی حلقوی و به تعداد سطر ها نیز یه لیست پیوندی حلقوی دیگر خواهیم داشت.



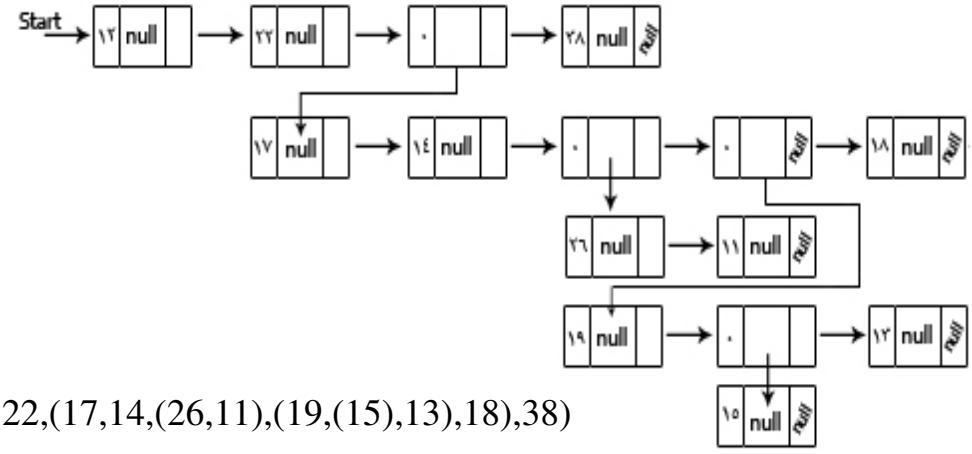
لیست عمومی



در لیست های عمومی هر گره دو حالت دارد. یا حاوی اطلاعات است یا خود یک زیر لیست است.
در لیست های عمومی اگر tag=NULL باشد آن گاه آن گره فقط حاوی data است اما اگر tag <> NULL باشد آن گاه آن گره data ندارد و خود یک زیر لیست است.
هر لیست عمومی شامل پرانتزهایی است.اگر بخواهیم لیست عمومی بالا را به صورت پرانتز بندی نمایش دهیم به صورت زیر است:

(12,(15,16,(19),(26,17)),22,(11,37),19)

مثال) لیست عمومی پرانتز بندی زیر را رسم کنید.



(12,22,(17,14,(26,11),(19,(15),13),18),38)

کاربردهای لیست عمومی

۱- در سیستم عامل برای ایجاد یک ساختار فایل تو در تو (سلسله مراتبی) از لیست عمومی استفاده می شود. به طوری که هر فایل یک گره معمولی (حاوی data) و هر فولدر یک زیر لیست است.
۲- کمپایلرهایی مانند پاسکال برنامه نوشته شده را به یک لیست عمومی تبدیل می کنند به طوری که هر دستور ساده یک گره حاوی data و هر بلاک به یک زیر لیست تبدیل می شود.
۳- یکی دیگر از کاربرد های لیست عمومی نمایش درخت های عمومی است.

برنامه پیمایش یک لیست عمومی

```
Void listgeneral (struct node * temp)
        {
        if (temp !=NULL)
                {
                if (temp->tag == NULL)
                        {
                        cout<<temp->data;
                        Listgeneral(temp->next);
                        }
                Else
                        {
                        listgeneral (temp->tag);
                        Listgeneral (temp->next);
                        }
                }
        }
```

تمرین:

۱- زیر برنامه ای بنویسید که تعداد گره های حاوی data یک لیست عمومی را حساب کند.

۲- زیر برنامه ای بنویسید که maximum عمق یک لیست عمومی را به دست آورد.

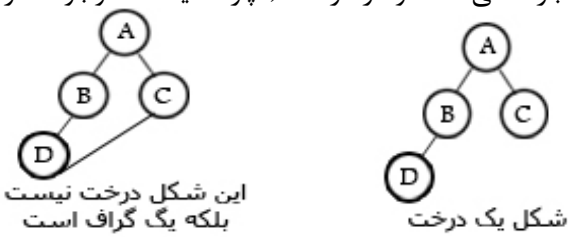
درخت عمومی

درخت , مجموعه ای محدود از یک یا چند گره می باشد به طوری که:

الف) دارای گره خاصی به نام ریشه است (root)

ب) بقیه گره ها خود , زیر درخت ریشه هستند.

نکته: درخت عمومی یک تعریف بازگشتی است و در درخت , چرخه یا حلقه وجود ندارد.



درجه گره: تعداد زیر درخت های یک گره (تعداد فرزندان آن) , درجه آن گره نام دارد.

برگ (Leaf): گره هایی که درجه صفر دارند برگ یا گره پایانی نامیده می شوند.

درجه درخت: حداکثر درجه گره های یک درخت را درجه درخت می گویند.

گره هم زاد: فرزندان یک گره هم زاد هستند.

سطح یک گره: اولین گره که ریشه است , دارای سطح ۱ و ما بقی سطوح , به ترتیب شماره گذاری می شود.

ارتفاع یا عمق درخت (Depth): به بیشترین سطح گره های یک درخت عمق درخت گفته می شود.

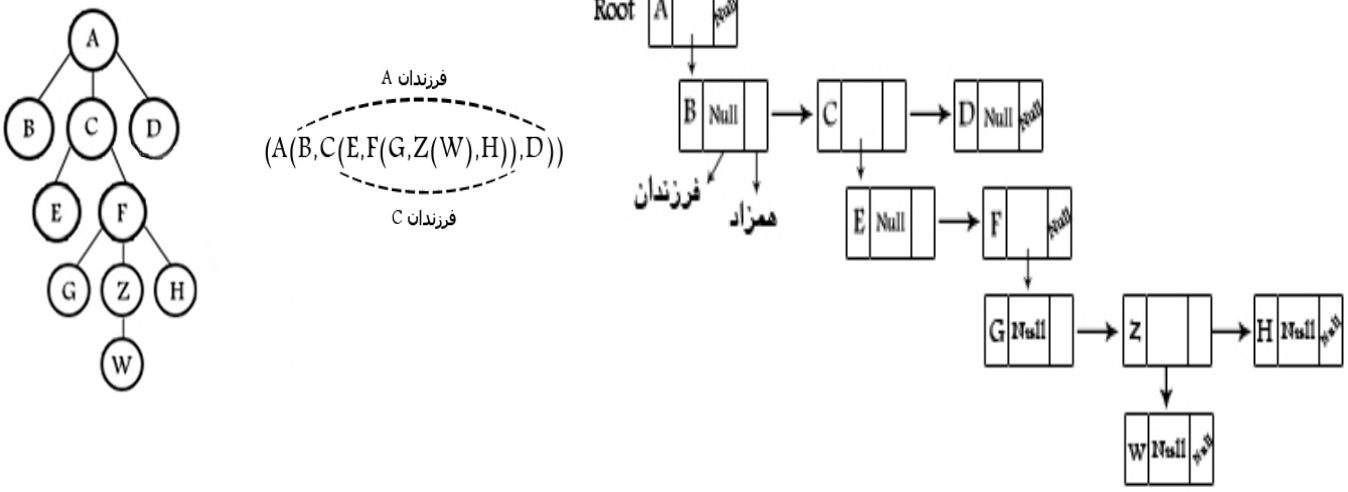
نکته: درخت یک ساختمان داده غیر خطی است.

در درخت عمومی , درجه درخت می تواند هر عددی باشد.

نحوه نمایش درخت عمومی به وسیله لیست عمومی

می توان درخت عمومی را به وسیله لیست عمومی نمایش داد.

مثال) درخت زیر را به شکل لیست عمومی رسم کنید.



درخت kتایی (k-ary tree): درختی که تعداد فرزندان یک گره حداگثرk باشد.
درخت متوازن (balanced tree): درختی است که اختلاف سطح برگ‌های آن حداکثر یک باشد.
درخت متوازن کامل: درختی است که اختلاف سطح برگ های آن صفر باشد.
درخت k تایی کامل (complete k-ary tree): درختی است که تعداد فرزندان هر گره (بجز برگ ها) دقیقا k باشد.

نکته ۱: در هر درخت k تایی کامل که n گره داشته باشد تعداد برگ‌ها $n_0 = \frac{(k-1)n+1}{k}$ می باشد.

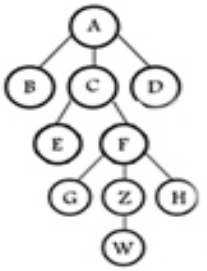
نکته ۲: اگر درخت k تایی کامل باشد تعداد کل گره ها $n = n_0 + n_1 + n_2 + n_3 + \cdots + n_k$ می باشد.

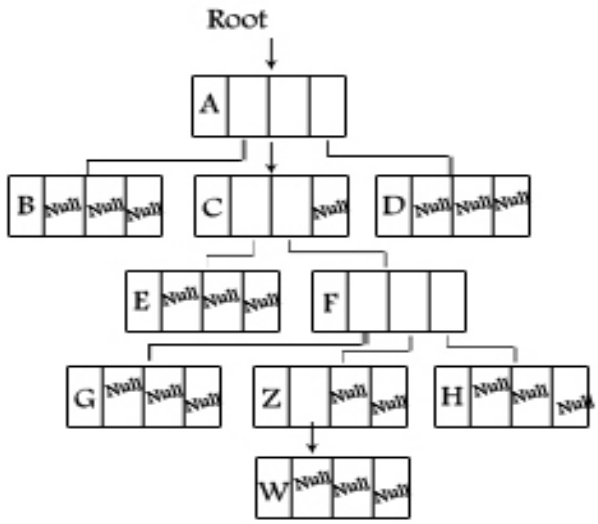
نکته ۳: در یک درخت k تایی کامل تعداد برگها از رابطهٔ $n_0 = (k-1)n_k + (k-2)n_{k-1} + \cdots + n_2 + 1$ به دست می آید
n_0 تعداد برگ‌ها ، n_1 تعداد گره‌هایی که یک فرزند دارند و n_k تعداد گره‌هایی است که k فرزند دارند.

نکته ۴: تعداد پیوند های بلا استفاده (nill) در نمایش لیست پیوندی یک درخت از رابطهٔ $n(k-1)+1$ بدست می آید.

نحوه نمایش درخت عمومی به کمک لیست های پیوندی

مثال) درخت عمومی روبرو را به کمک لیست پیوندی نمایش دهید.





مزیت:

مزیت این روش این است که ساختمان داده ایجاد شده شبیه درخت است.

معایب:

۱- به دلیل ثابت بودن تعداد اشاره گرهای هر گره , درجه درخت غیر قایل تغییر است.

۲- به دلیل اینکه بیشتر درخت ها اسپارس هستند (فرزندان زیادی ندارند) تعداد اشاره گرهای NULL زیاد میشود که باعث اتلاف حافظه می شود.

درخت دودویی (Binary)

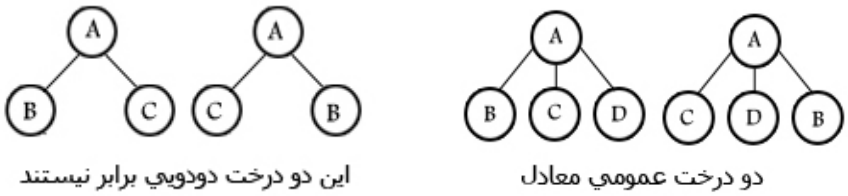
درختی که حداکثر درجه آن دو باشد درخت دودویی است.در این درخت هر گره می تواند یک زیر درخت چپ و یک زیر درخت راست داشته باشد.

نکته: درخت دودویی می تواند تهی باشد یعنی هیچ گره ای و حتی ریشه هم نداشته باشد.

تفاوت درخت دودویی و درخت عمومی

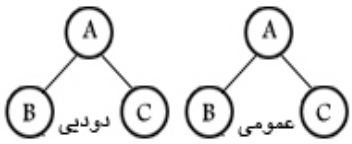
۱- درخت دودویی تهی وجود دارد ولی درخت عمومی حداقل باید یک گره داشته باشد.

۲- در درخت عمومی ترتیب فرزندان اهمیتی ندارد اما ترتیب فرزندان در درخت دودویی بسیار مهم است.



نکته: اگر درخت دودویی بالا را به چشم درخت عمومی نگاه کنیم دو درخت با هم برابرند.

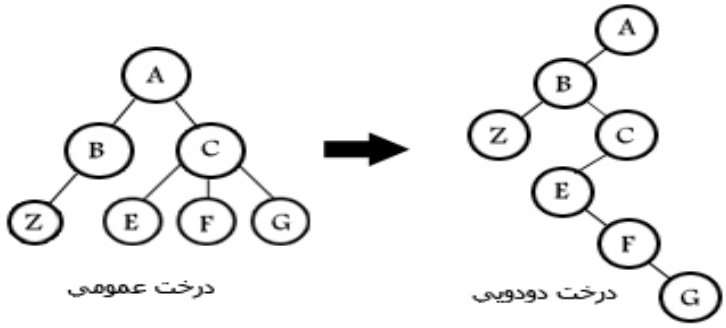
نکته: درخت های روبرو معادل نیستند.



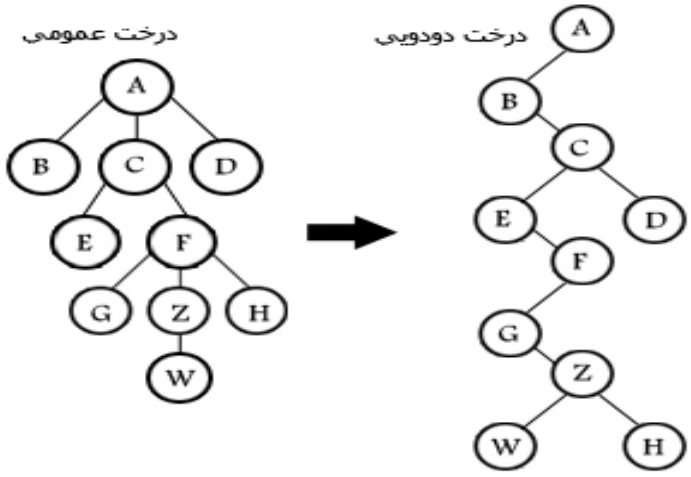
الگوریتم تبدیل درخت عمومی به دودویی (فرزند چپ – همزاد راست)

در این الگوریتم هر گره را به همراه فرزند چپش می نویسیم.اگر آن گره همزاد راست داشت به عنوان فرزند راست آن در نظر می گیریم.

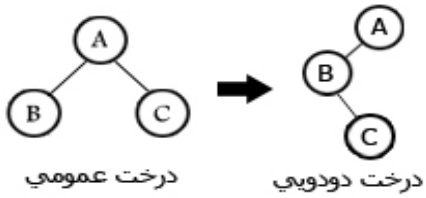
مثال) درخت عمومی زیر را به درخت دودویی تبدیل کنید.



مثال) درخت عمومی زیر را به دودویی تبدیل کنید.



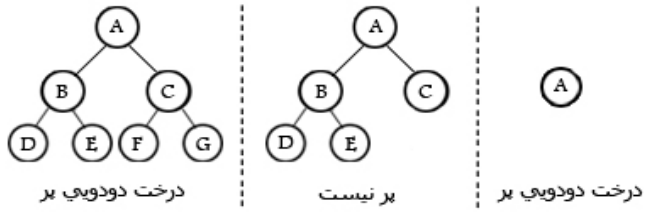
مثال) درخت عمومی زیر را به دودویی تبدیل کنید.



درخت دودوی پر

درخت دودویی پر , درختی است که همه گره ها به جزء گره های سطح آخر دقیقا دو فرزند دارند (درجه دو هستند).

مثال)

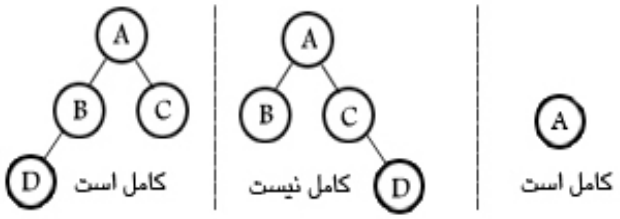


نکته: اگر یک درخت دودویی پر با عمق K داشته باشیم تعداد گره های آن دقیقا $2^{k}-1$ است.

نکته: اگر یک درخت دودویی پر با n گره داشته باشیم عمق آن درخت $\log_2^{n+1}$ است.

درخت دودویی کامل

یک درخت دودویی کامل است هر گاه تمام سطح های آن به غیر از آخرین سطح پر باشد و سطح آخر از سمت چپ در حال پر شدن باشد.



نکته: هر درخت پری کامل است ولی هر درخت کاملی پر نیست.

مثال) یک درخت دودویی کامل با عمق K حداکثر چند گره و حداقل چند گره می تواند داشته باشد؟

حداکثر $2^{k}-1$ و حداقل 2^{k-1}

نکته: در یک درخت دودویی در سطح i ام حداکثر می تواند 2^{i-1} گره وجود داشته باشد.

قضیه: در یک درخت دودویی اگر تعداد گره های درجه دو , n_2 باشد و تعداد گره های پایانی n_0 باشد همیشه رابطه زیر برقرار است:

$$n_0=n_2+1$$

نکته: اگر یک درخت دودویی کامل با تعداد n گره داشته باشیم عمق آن $[\log_2^n]+1$ است.

مثال) درخت دودویی کاملی دارای n گره است.تعداد برگهای آن چقدر است؟

$(n+1)/2$

مثال) یک درخت دودویی با n گره حداقل و حداکثر دارای چه عمقی است؟

حداکثر عمق زمانی ایجاد می شود که درخت دودویی مورب ایجاد کنیم و حداقل آن این است که درخت پر(کامل) شود. پس:

حداقل:$[\log_2^{n+1}]$ و حداکثر:n

نحوه نمایش درخت دودویی

الف) نحوه نمایش درخت دودویی به کمک آرایه ها

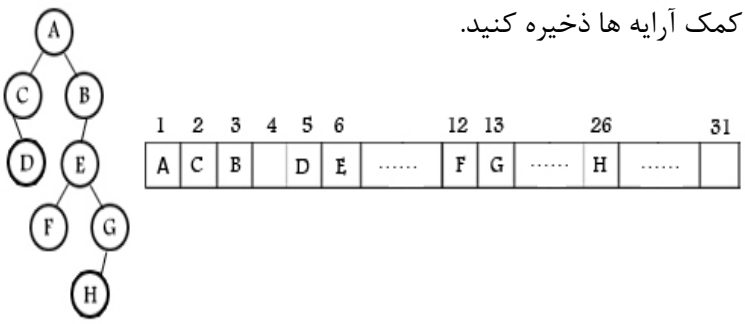
اگر یک درخت دودویی با عمق k داشته باشیم یک آرایه به طول $2^{k}-1$ ایجاد می کنیم.

سپس ریشه را در اندیس ۱ قرار می دهیم و سپس برای به دست آوردن محل قرار گیری فرزندان هر گره , از دو فرمول زیر استفاده می کنیم:

محل قرار گیری فرزند سمت چپ = i*2

محل قرار گیری فرزند سمت راست = i*2+1

مثال) درخت دودویی زیر را به کمک آرایه ها ذخیره کنید.



مزیت:

یکی از مزایای این روش این است که به راحتی می توان فرزندان و پدر هر گره را دانست.

اندیس پدر = i div 2

معایب:

۱- معمولا درخت ها حالت اسپارس دارند یعنی بیشتر گره ها فرزند ندارند به همین دلیل آرایه استفاده شده فضای زیادی را به هدر می دهد. مانند درختان مورب

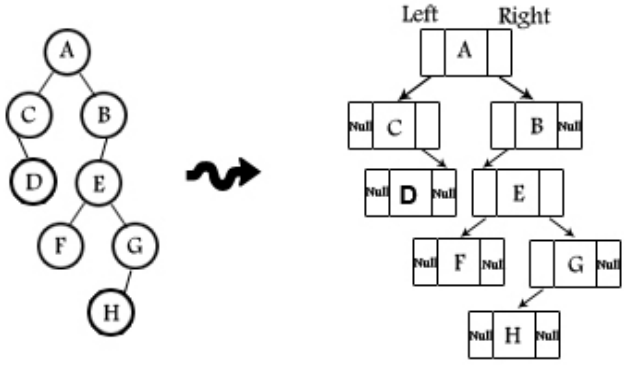
۲- اگر بخواهیم عمق درخت را زیاد کنیم باید طول آرایه را نیز تغییر دهیم که این امکان پذیر نیست (چون آرایه ها ایستا هستند).

ب) نمایش درخت دودویی به کمک لیست های پیوندی

در این روش می توانیم یک رکورد به صورت زیر برای هر گره درخت ایجاد کنیم.

```
Struct node
        {
        struct node * left;
        Int data;
        Struct node * right;
        };
```

مثال) درخت دودویی زیر را به کمک لیست پیوندی نمایش دهید.

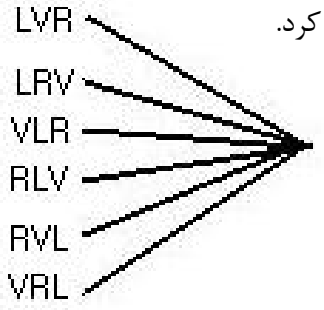


این نوع نمایش ظاهری شبیه درخت دارد . در این نمایش نمی توانیم با داشتن فرزند , پدر آن را به دست آوریم.در این روش فضای اضافه زیادی نخواهیم داشت.

پیمایش درخت دودویی

پیمایش درخت یعنی ملاقات کردن(چاپ کردن) همه گره ها با ترتیب و نظم خاص.

به دو روش کلی می توان درخت دودویی را پیمایش کرد.



۱- پیمایش عمقی :

۲- پیمایش سطحی

روش LVR (inorder) **:**

در این روش ابتدا زیر درخت سمت چپ پیمایش می شود سپس ریشه چاپ شده و در نهایت زیر درخت سمت راست پیمایش می شود.

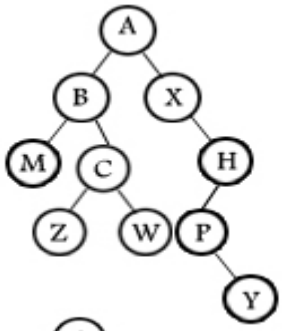
روش LRV (postorder) **:**

در این روش ابتدا زیر درخت سمت چپ سپس زیر درخت سمت راست و در نهایت ریشه چاپ می شود.

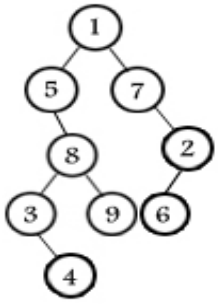
روش VLR (preorder) **:**

در این روش ابتدا ریشه , سپس زیر درخت سمت چپ ودر نهایت زیر درخت سمت راست چاپ می شود.

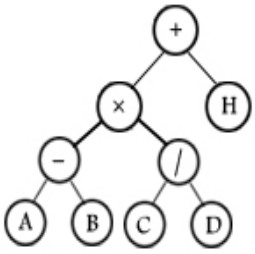
مثال) درخت های دودویی زیر را به سه روش LVR ,LRV,VLR پیمایش کنید.



LVR = MBZCWAXPYH
LRV = MZWCBYPHXA
VLR = ABMCZWXHPY



LVR = 534891762
LRV = 439856271
VLR = 158349726



LVR = (A-B)*(C/D)+H
LRV = AB-CD/*H+
VLR = +*-AB/CDH

نکته: در روش LRV آخرین گره ای که ملاقات می شود ریشه است ولی در روش VLR اولین گره ملاقاتی ریشه است.

برنامه پیمایش درخت به روش LVR (به روش بازگشتی)

```
Void LVR(struct node * ptr)
        {
        if (ptr != NULL)
                {
                LVR(ptr->left);          L
                Cout<<ptr->data;         V
                LVR(ptr->right);         R
                }
        }
```

نکته: با جابجایی دستورات خطوط L,R,V می توان پیمایش های دیگر را نیز به دست آورد.

این روش بسیار کند عمل می کند زیرا به تعداد اشاره گرهای درخت , n*2 فراخوانی انجام می شود.در پیمایش عمقی یک درخت دودویی مجبوریم از استک استفاده کنیم.

در برنامه بالا نیز از روش بازگشتی استفاده شده که خود دارای استک است.

مثال) برنامه پیمایش LVR یک درخت دودویی به صورت غیر بازگشتی و به کمک یک استک را بنویسید.

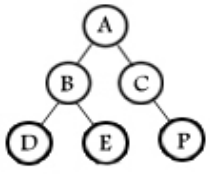
```
Void LVR(struct node * ptr)
{
Stack s;
While(1)
        {while (ptr!=NULL)
                {
                s.push(ptr);
                ptr=ptr->left;
                }
        If (s.isempty==1) break;
        Ptr=s.pop();
        Cout<<ptr->data;
        Ptr=ptr->right;
        }
}
```

نکته: مزیت این برنامه نسبت به برنامه قبل این است که سریعتر عمل می کند.

پیمایش سطحی(ترتیبی)

در این روش هر سطح به ترتیب از بالا به پایین نمایش داده می شود.در هر سطر از چپ به راست عمل پیمایش انجام می شود.

مثال) پیمایش سطحی درخت روبرو را بنویسید.



(A,B,C,D,E,P)

در این روش از یک صف استفاده می کنیم.ابتدا ریشه را چاپ کرده , فرزندانش را در صورت وجود به ترتیب از چپ به راست داخل صف

می ریزیم.هر بار یک گره از صف خارج کرده و آن را چاپ می کنیم و فرزندانش را به ترتیب از چپ به راست داخل صف می ریزیم.این عمل تا زمانی که صف خالی شود تکرار می شود.

برنامه پیمایش سطحی

```
Void levelorder(struct node * ptr)
{ queue q;
If (ptr==NULL) return;
q.add(ptr);
while (q.isempty==0)
        {ptr=q.delete();
        Cout<<ptr->data;
        If (ptr->left !=NULL)
                q.add(ptr->left);
        if (ptr->right !=NULL)
                q.add(ptr->right);
        }
}
```

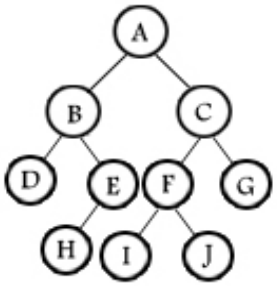
ترسیم درخت با توجه به پیمایش های آن

اگر پیمایش های میانوندی (LVR) و پسوندی (LRV) یک درخت را داشته باشیم می توانیم آن درخت را به صورت منحصر به فرد رسم کنیم.هم چنین با داشتن پیمایش های میانوندی و پیشوندی می توان درخت منحصربه فرد ترسیم کرد.

نکته: در صورت داشتن پیمایش پیشوندی و پسوندی نمی توانیم درخت منحصر به فرد رسم کنیم.

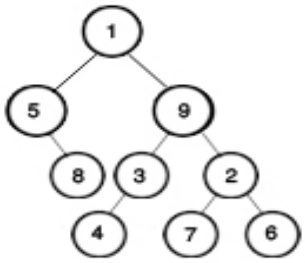
نکته: وجود پیمایش میانوندی (LVR) الزامی است.

مثال) پیمایش LVR و LRV یک درخت داده شده است.درخت آن را رسم کنید.



LVR = DBHEAIFJCG
VLR = ABDEHCFIJG

مثال) پیمایش LVR و LRV یک درخت داده شده است.درخت آن را رسم کنید.



LVR = 581439726
LRV = 854376291

تمرین: با توجه به پیمایش های Inorder و Preorder درخت دودویی درخت را رسم کنید؟

Inorder➔ D B H E A I F J C G
Preorder ➔ A B D E H C F I J G

تمرین) با توجه به پیمایش های Inorder و Postorder درخت دودویی درخت را رسم کنید؟

Inorder ➔ n1 n2 n3 n4 n5 n6 n7 n8 n9
Postorder ➔ n1 n3 n5 n4 n2 n8 n7 n9 n6

تمرین) با توجه به پیمایش Inorder و Postorder درخت دودویی درخت را رسم کنید؟

Inorder➔ E I C F J B G D K H L A
Postorder➔ I E J F C G K L H D B A

مثال) زیر برنامه ای بنویسید که با فرض وجود یک درخت دودویی یک کپی از آن ایجاد شود.

```
Struct node * copy(struct node * ptr)
{struct node * temp;
If (ptr !=NULL)
        {
        Temp=new node;
        Temp->data=ptr->data;
        Temp->left=copy(ptr->left);
        Temp->right=copy(ptr->right);
        Return(temp);
        }
```

```
Else
        Return(NULL);
}
```

مثال) برنامه ای بنویسید که با فرض وجود یک درخت دودویی عمق آن را محاسبه کند.

```
Int depth(struct node * ptr)
{if (ptr==NULL)
        Return(0);
Else
        Return(1+max(depth(ptr->left),depth(ptr->right))
}
```

مثال) زیر برنامه ای بنویسید که با فرض وجود یک درخت دودویی تعیین کند که آیا پر است یا نه؟

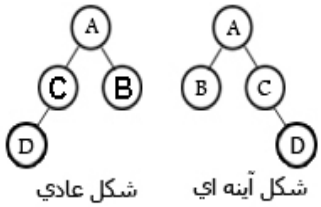
برای نوشتن این برنامه می توانیم ابتدا تعداد گره های یک درخت را محاسبه کنیم.سپس به کمک برنامه قبل عمق آنرا محاسبه کنیم.اگر رابطه روبرو برقرار باشد درخت پر است2^n-1=k

با توجه به این سوال و رابطه بالا برنامه ای بنوسید که با فرض وجود یک درخت دودویی تعداد گره های آن را محاسبه کند.

```
Int numberofnode(struct node * ptr)
{if (ptr !=NULL)
        Return(1+numberofnode(ptr->left)+numberofnode(ptr->right);
Else
        Return(0);
}
```

تمرین: زیر برنامه ای بنویسید که با فرض وجود دو درخت دودویی تساوی آنها را بررسی کند.(هم محتوا و هم گره)

تمرین: زیر برنامه ای بنویسید که با فرض وجود یک درخت دودویی تصویر آینه ای آن را ترسیم کند.



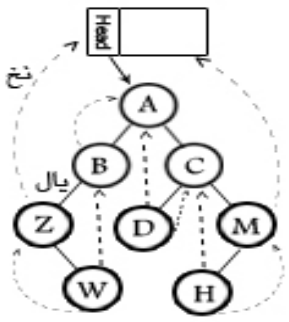
تمرین: برنامه ای بنویسید که تعداد برگهای یک درخت را بشمارد.

درخت نخی دودویی (Thread tree)

اگر یک درخت با n تا گره داشته باشیم n*2 تا اشاره گر وجود دارد که از این تعداد همیشه n+1 از ان تهی (NULL) و n-1 از آن غیر تهی است.بنابراین تعداد اشاره گرهای بدون استفاده بیشتر از تعداد اشاره گرهای استفاده شده است.

پیمایش درخت دودویی بدون استفاده از استک

هدف از ایجاد درخت نخی دودویی این است که در پیمایش عمقی آن از استک استفاده نکنیم. بدین ترتیب پیمایش درخت سریعتر انجام خواهد شد.



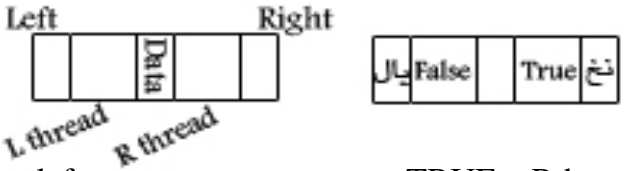
LVR = ZWBADCHM

می توانیم از اشاره گرهای NULL برای اتصال به گره قبلی و بعدی در پیمایش LVR استفاده کنیم.مثلا در شکل بالا گره W که هر دو طرف آن تهی (NULL) است می تواند به گره قبلی و بعدی خود در پیمایش LVR وصل شود که در این صورت به این اتصالات نخ می گویند.

نحوه نخ کشی درخت:

برای این کار ابتدا پیمایش LVR درخت را به دست آورده و سپس , هر اشاره گر چپی که تهی بود به گره قبلی خود در LVR متصل می شود.البته یک گره head نیز وجود دارد که اولین گره که قبلی ندارد به آن متصل می شود و هر اشاره گر سمت راستی که تهی بود به گره بعدی خود در پیمایش LVR متصل می شود.البته گره آخر به head متصل می شود.بدین ترتیب اتصالات تهی کمک می کند تا به سمت بالای درخت حرکت کنیم و نیاز به استک نداشته باشیم.

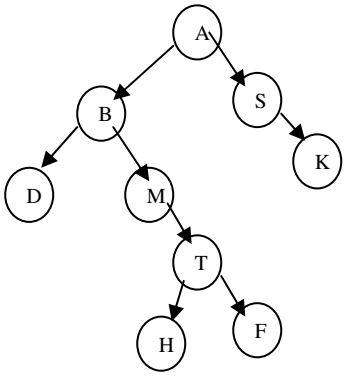
نکته: ساختمان داده هر گره در درخت نخی به صورت شکل زیر است که در این حالت می توان نخ و یال را از هم جدا و تشخیص داد.



در ساختمان داده بالا اگر Lthread و Rthread , TRUE باشند به معنای این است که left و right بیانگر نخ هستند و اگر این دو فیلد FALSE باشند یعنی اینکه left و right یال هستند.

پیمایش درخت نخی به روش LVR

```
Struct node * next(struct node * ptr)
{
Struct node * temp;
Temp=ptr->right;
If (ptr->rthread == false)
        While (temp->lthread == false)
                temp=temp->left;
return (temp);
}
Void threadorder(struct node * ptr)
        {
        Struct node temp1=root;
                Do
                {
                Temp1=next(temp1);
                Printf("%d",temp1->data);
                }
                While(temp1 !=NULL)
}
```



هرم (Heap)

max tree : درختی است که مقدار هر گره آن بزرگتر یا مساوی فرزندانش باشد.

Min tree : درختی است که مقدار هر گره آن کوچکتر یا مساوی فرزندانش باشد.

Max heap : درخت دودویی کاملی است که max tree نیز باشد.

نکته: درخت max heap حتما باید دودویی کامل باشد.

Min heap : درخت دودویی کاملی است , که min tree هم باشد.

نکته: در درخت max heap بزرگترین عنصر درخت در ریشه قرار می گیرد و در min heap کوچکترین عنصر درخت در ریشه قرار می گیرد.

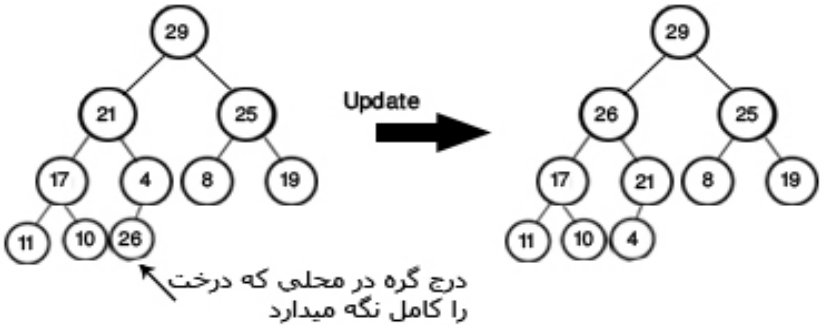
نحوه درج و حذف یک گره در max heap

معمولا گره ای که در یک heap حذف می شود ریشه است.برای حذف کردن ریشه , عنصری از درخت که در پایین ترین سطح و راست ترین قسمت وجود دارد را پیدا کرده و ان را جایگزین ریشه می کنیم. سپس درخت را update می کنیم به طوری که درخت دوباره به یک heap تبدیل شود.

مثال)

مراحل update درخت heap

مثال)

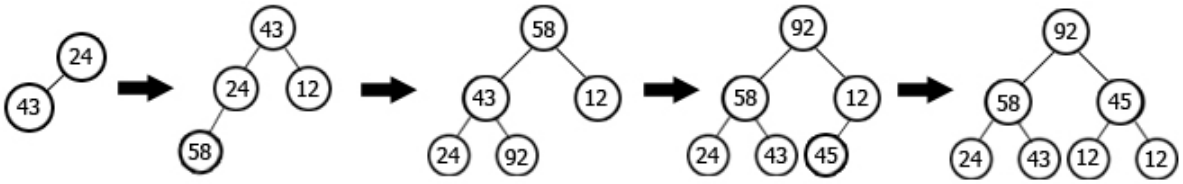


مراحل update درخت heap

الگوریتم درج

ابتدا در سطر آخر , راست ترین قسمت NULL را پیدا کرده و گره جدید را در آن محل درج می کنیم. سپس باید درخت را update کنیم.

مثال) اعداد ۲۴ و ۴۳ و ۱۲ و ۵۸ و ۹۲ و ۴۵ و ۱۲ به ترتیب (از راست به چپ) وارد یک درخت max heap تهی می شود. درخت را مرحله به مرحله رسم کنید.



نکته: حذف و درج یک گره در درخت heap دارای پیچیدگی زمانی $O(\log^{n}_{2})$ است که n تعداد عناصر داخل heap است.

کاربرد heap

۱- heap sort

یکی از کاربرد های درخت heap در مرتب سازی داده ها است.(الگوریتم heap sort).
در این حالت مجموعه ای از اعداد نا مرتب داریم .اعداد را به ترتیب وارد یک درخت heap می کنیم (insert) که این عمل $O(\log^{n}_{2})$ زمان می برد. حال n گره را از ریشه درخت خارج می کنیم(delete). بدین ترتیب داده ها منظم و مرتب شده از درخت خارج می شوند که مجموعا دارای پیچیدگی زمانی $O(n.\log^{n}_{2})$ است.(یکی از سریعترین روش های مرتب سازی)

۲- صف اولویت

صف اولویت صفی است که داده ها به ترتیبی که وارد شده اند خارج نمی شوند بلکه بسته به اولویت آنها از صف خارج می شوند یعنی هر عنصری که وارد می شود به ته صف اضافه می شود اما هر عنصری که اولویت بیشتری دارد از صف خارج می شود.

نکته: می توانیم صف اولویت را به کمک آرایه ها یا لیست های پیوندی ایجاد کنیم اما دارای پیچیدگی زمانی بسیار زیاد $O(n)$ خواهد بود.

درخت max heap می تواند نقش یک صف اولویت را داشته باشد.هر عنصری که به پایین درخت اضافه می شود درخت update شده و به سمت بالا حرکت می کند و همیشه عنصری که از همه بزرگتر باشد در ریشه قرار می گیرد و همیشه ریشه را حذف می کنیم.

یکی ار کاربرد های صف اولویت در سیستم عامل های real time (بلادرنگ) است.

این سیستم عامل ها فرآیند ها را طبق اولویتشان اجرا می کنند و تقسیم زمان رخ نمی دهد.

در این سیستم عامل از درخت heap استفاده می شود.

ایجاد صف اولویت به کمک heap دارای پیچیدگی زمانی $O(\log^{n}_{2})$ است.

تمرین: درج و حذف عنصر در پیاده سازی صف اولویت به کمک:

الف) درخت heap ب) آرایه نامرتب ج) لیست نامرتب د) آرایه مرتب ه) لیست مرتب

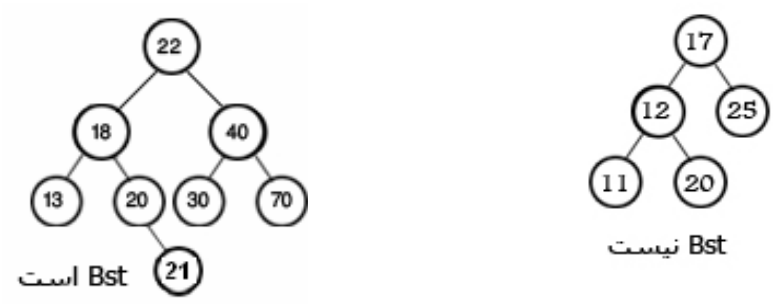
را از نظر پیچیدگی زمانی با هم مقایسه کنید.

نکته: برای مرتب کردن نزولی از max heap و برای مرتب کردن صعودی از min heap استفاده می شود.

مثال) کدام درخت را نمی توان با آرایه نمایش داد؟

۱) اریب ۲) پر ۳) کامل ۴) max heap

(Binary search tree) BST
درخت جستجوی دودویی



درخت BST , درخت دودویی است که ممکن است تهی باشد.اگر تهی نباشد آن گاه هر عنصر آن باید از زیر درخت سمت چپ بزرگتر و از زیر درخت سمت راست کوچکتر باشد.

نکته: درخت BST ممکن است کامل نباشد.

نکته: درخت BST فاقد داده تکراری است.

کاربردهای درخت BST

۱- یکی از کاربرد های درخت BST در جستجوی داده ها است.

مثال) با فرض وجود یک درخت BST , برنامه ای بنویسید که یک عنصر در آن جستجو کند.

(برنامه بازگشتی)

```
Struct node * search(struct node * ptr , int key)
        {if (ptr == NULL)
                Return(NULL);
        Else
                If (ptr-> data == key)
                        Return(ptr);
                Else
                        If (ptr-> data > key)
                                Return(search(ptr->left , key));
                        Else
                                Return(search(ptr->right , key));
        }
```

(برنامه غیر بازگشتی)

```
Struct node * search(struct node * ptr , int key)
        {while (ptr <> NULL)
                {if (key == ptr->data)
                        Return(ptr);
                If (key <ptr->data)
                        Ptr=ptr->left;
                Else
                        Ptr=ptr->right;
                }
        Return(NULL);
        }
```

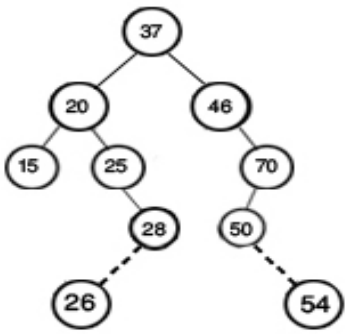
نکته: جستجو در درخت BST دارای پیچیدگی زمانی $O(\log^{n}_{2})$ است.

درج یک عنصر در درخت BST

برای درج یک گره جدید در درخت BST ابتدا گره جدید را در درخت BST جستجو می کنیم. اگر گره جدید در درخت BST پیدا شد نمی توانیم آن را درج کنیم (زیرا BST عنصر تکراری ندارد).
در غیر این صورت (در صورت پیدا نشدن) آخرین گره ای که مورد جستجو واقع شده است را به دست می آوریم. حال گره جدید را بسته به اندازه اش به چپ یا راست اضافه می کنیم.

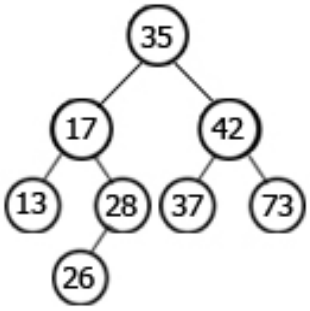
نکته: عمل درج دارای پیچیدگی زمانی $O(\log_2^n)$ است.

مثال) گره های ۲۶ و ۵۴ را به درخت روبرو اضافه کنید.



۲- یکی دیگر از کاربردهای درخت BST , حذف عناصر تکراری و مرتب سازی داده ها است.

مثال) اعداد ۳۵ و ۴۲ و ۱۷ و ۱۳ و ۲۸ و ۴۲ و ۷۳ و ۲۶ و ۳۷ را به ترتیب (از راست به چپ) در یک درخت تهی BST درج کنید.



LVR: 13,17,26,28,35,37,42,73
RVL: 73,42,37,35,28,26,17,13

هنگامی که داده های جدید وارد می شوندعنصر تکراری حذف می شوند. اگر LVR درخت BST را بنویسیم داده های مرتب شده به دست می آید.

نکته: پیچیدگی زمانی ساخت درخت BST , $O(n.\log_2^n)$ است.

نکته: اگر LVR درخت BST مرتب نباشد یعنی اینکه درخت را اشتباه رسم کرده ایم.

نکته: اگر بخواهیم داده ها نزولی مرتب شوند باید شکل RVL آن را بنویسیم.

حذف یک عنصر از درخت BST

برای حذف یک گره از درخت BST ۳ حالت ممکن است رخ دهد.

الف) گره ای که می خواهیم حذف کنیم برگ باشد که در این صورت به راحتی قابل حذف است.

ب) گره ای که می خواهیم حذف کنیم تک فرزندی باشد که در این صورت فرزند جانشین گره خواهد شد.

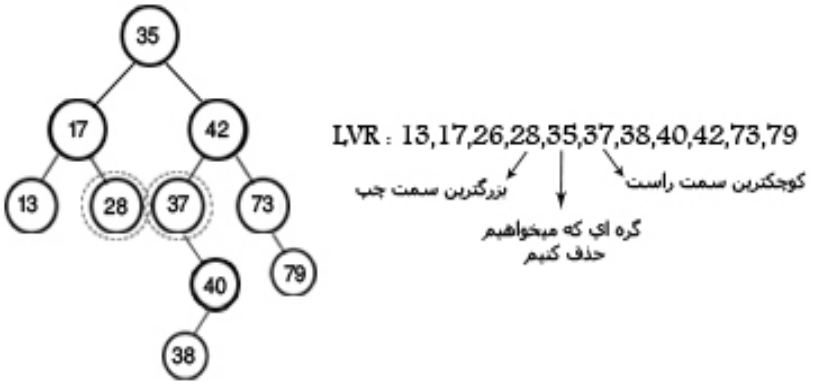
ج) گره ای که می خواهیم حذف کنیم دو فرزندی باشد (درجه ۲ باشد) که در این صورت ۲ راه حل داریم:

۱- می توانیم بزرگترین عنصر سمت چپ را جانشین آن کنیم.

۲- می توانیم کوچکترین عنصر سمت راست را جانشین کنیم.البته ممکن است هر کدام از حالتهای ۱ و۲ منجر به حذف عنصر تک فرزندی شود که مانند قسمت ب عمل می کنیم.

نکته: برای پیدا کردن کوچکترین عنصر سمت راست یا بزرگ ترین عنصر سمت چپ می توانیم LVR درخت را به دست آورده , که گره ما قبل و بعدی گره مربوطه , کوچکترین و بزرگترین خواهند بود.

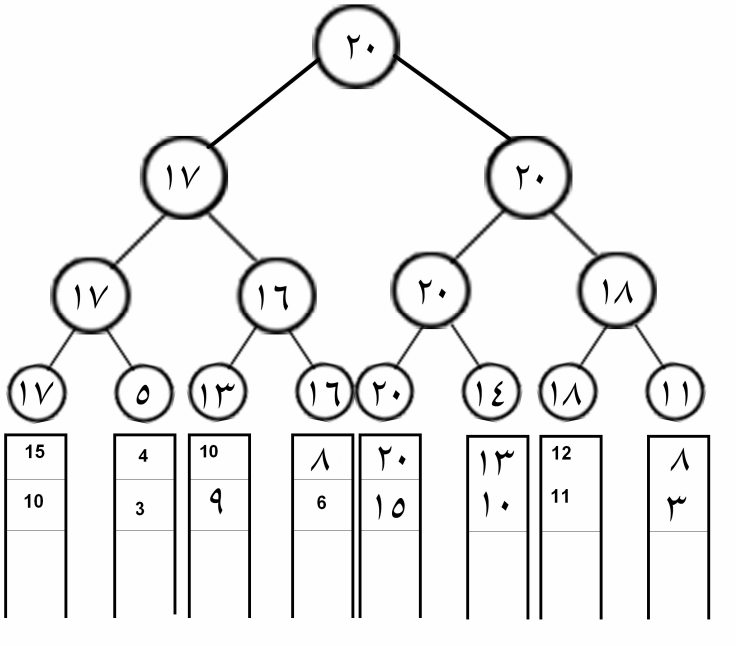
مثال)



درخت انتخابی

در این درخت بین برگ های سطح آخر مسابقه برگزارمی شود و برنده ها (اعداد بزرگتر) به سمت بالای درخت حرکت می کنند. عنصری که به عنوان برنده اصلی از درخت خارج می شود از پایین درخت در محل قبلی اش یک داده جدید وارد می شود (RUN می شود).

نکته: یکی از کاربرد های این درخت در ادغام کردن تعدادی آرایه است.



درخت دودویی متوازن AVL (Balanced Binary Tree)

درخت AVL یک درخت BST است که ارتفاع آن توازن دارد یعنی حداکثر اختلاف ارتفاع بین زیردرخت های فرعی چپ و راست، یک می باشد .

نکته ۱: حداکثر ارتفاع درخت AVL برا بر است با: $h = \lfloor \log_2^n \rfloor$

نکته ۲: زمان درج و حذف یک عنصر در AVL برا بر است با : $O(\log_2^n)$

نکته ۳: در درخت های AVL رابطه زیر برقرار است :

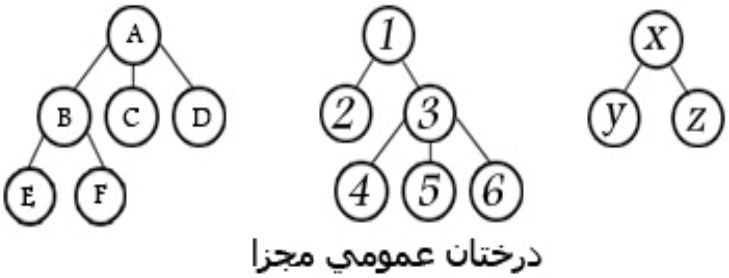
$$Avl_{\min}(n) = Avl_{\min}(n-1) + Avl_{\min}(n-2) + 1$$

جنگل

به مجموعه ای از درختان عمومی مجزا (N >=0) جنگل گفته می شود.

نکته: یکی از کاربرد های جنگل در مجموعه های ریاضی است.

نمایش جنگل در کامپیوتر با مشکلاتی مواجه است به همین دلیل آن را به درخت دودویی تبدیل می کنیم.



درختان عمومي مجزا

الگوریتم تبدیل جنگل به درخت دودویی

ابتدا همه درختان را به کمک الگوریتم فرزند چپ – همزاد راست به درختان دودویی تبدیل می کنیم. سپس درختان دودویی را از سمت راست ریشه به یکدیگر متصل می کنیم.

نکته: حالتهای متعددی ممکن است رخ دهد زیرا چیدمان درختان دودویی ممکن است متفاوت باشد.

نمونه سئوال تست:

1- چنانچه بخواهیم داده های تکراری را از لیستی حذف کنیم از کدام ساختار داده یی برای لیست مزبور استفاده می کنیم؟

الف) Graph ب) Min Heap ج) BST د) Max Heap

۲-درخت دودویی پر، کدام است؟

الف) درخت دودویی به عمق h که دارای 2^{h-1} گره است ب) درخت دودویی به عمق h که دارای $2^{h}-1$ گره است.

ج) درخت دودویی که حداقل گره های سطح آخر آن پر باشد د) درخت دودویی که فقط گره های سطح آخر آن پر باشد.

۳-اگر تعداد برگهای یک درخت دودویی کامل ۷۲ عدد باشد حداکثر ارتفاع آن درخت چقدر است؟

الف) 14 ب) 36 ج) 8 د) 10

۴-زیربرنامه روبرو کدام ویژگی از درخت دودویی T را محاسبه می کند؟

```
int  test(struct node *T)
{
  If (T==null)
     return 0;
  else
     return(1+max( test(T->Left)+test(T->Right));
}
```

الف) تعداد عناصر

ب) تعداد گره های درجه دو

ج) تعداد برگ ها

د) ارتفاع درخت

۵-پیمایش درخت دودویی به روش روبرو معادل کدام گزینه است؟

```
Void abc(struct node * P)
   {
     Cout<<P->Data;
     abc(P->Right);
     abc(P->left);
}
```

الف) گره های ملاقات شده معکوس گره های ملاقات شده توسط Preorder است

ب) گره های ملاقات شده معکوس گره های ملاقات شده توسط Postorder است

ج) گره های ملاقات شده معکوس گره های ملاقات شده توسط Inorder است

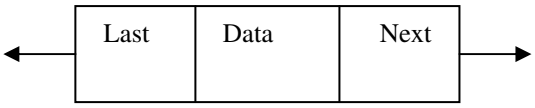
د) هیچکدام گزینه ها صحیح نیست.

۶-گره های 7 , 5 , 2 , 9, 6 ,4 , 1 , 3 (به ترتیب از چپ به راست) را دریک درخت جستجوی دودویی خالی به نام T درج می کنیم. پیمایش Postoredr آن درخت کدام گزینه است؟ (از چپ به راست)

الف) 3, 4 , 6 , 9, 7 , 5 , 1 , 2 ب) 9, 7, 6 , 5 , 4, 3 , 1 , 2
ج) 9, 7, 5, 6 , 4, 3, 1 , 2 د) 7, 9 , 5, 6 , 4, 3 , 2 , 1

۷-برای درج یک عنصر در ابتدای یک لیست پیوندی دوطرفه ساده چه تعداد از اشارگرهای Next و Last را باید تغییر دهیم؟

الف) دو عدد Next و یک عدد Last
ب)یک عدد Next و دو عدد Last
ج)یک عدد Next و یک عدد Last
د) دو عدد Next و دو عدد Last



۸- در یک درخت دودویی تعداد گره های پایانی ۳۲ تا است تعداد گره های درجه ۲ آن چند تا است؟

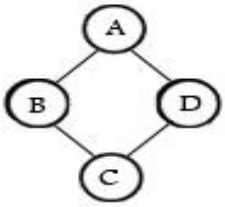
الف) ۱۶ ب) ۳۳ ج) ۳۱ د) ۱۵

۹-حداقل تعداد گره یکدرخت باینری کامل و یک درخت باینری کامل با عمق K به ترتیب(راست به چپ) برابر است با:

الف) 2^{k-1} و 2^{k-1} ب) 2^{k} و 2^{k-1} ج)2^{k} -1 و2^{k-1} د) 2^{k-1} و 2^{k} -1

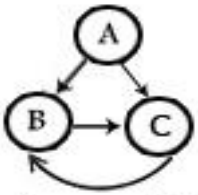
گراف

گراف G شامل دو مجموعه V , E است که V مجموعه محدود و غیر تهی از راس ها است و E مجموعه ای محدود (احتمالا تهی) از لبه ها(یال ها) است و به صورت زیر نوشته می شود:



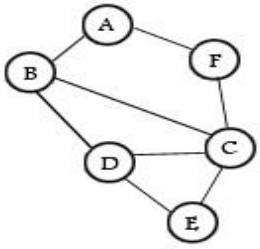
گراف

V={A,B,C,D} نمی تواند تهی باشد
E={<A,B><A,D><D,C><B,C>} می تواند تهی باشد



گراف جهت دار

دو نوع گراف وجود دارد :
۱- گراف جهت دار
۲- گراف بدون جهت



گراف بدون جهت

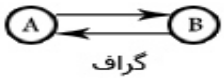
نکته: درخت , حالت خاصی از گراف است.

نکته: درخت , گراف بدون سیکل است.

نکته: گراف یا جهت دار است یا بدون جهت.

نکته: گراف فاقد لبه ای است که از یک راس به خود متصل است.

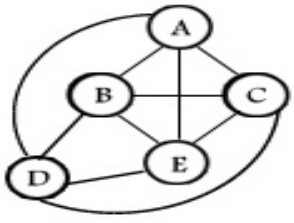
نکته: گراف فاقد لبه های چند گانه است.



گراف

گراف ساده: گرافی است که فاقد حلقه و لبه چندگانه باشد.

گراف کامل: گرافی است که دارای حداکثر تعداد یال ها باشد.به عبارت دیگر یعنی اینکه همه راس ها مجاور یکدیگر باشند.



گراف کامل

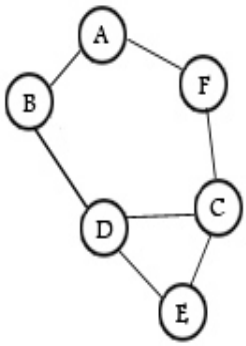
گره مجاور: i مجاور راس j است اگر لبه <i , j> در گراف وجود داشته باشد.

نکته: یک گراف کامل بدون جهت دارای n(n-1)/2 لبه(یال) است اما یک گراف کامل جهت دار دارای n(n-1) لبه (یال) است.

نکته: گراف یک ساختمان داده غیر خطی است.

مسیر

رفتن از یک راس به راس دیگر مسیر نام دارد.

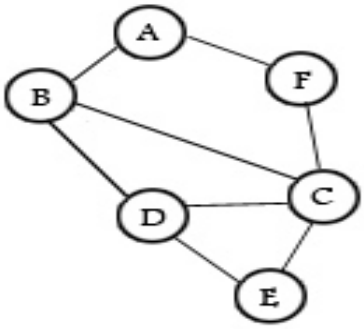


مسیر= {<A,B>,<B,D>,<D,C>}

طول مسیر= 3

مسیر ساده

مسیری که همه رئوس آن احتمالا به جزء اولی و آخری مجزا باشد.



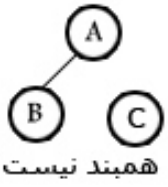
1) {<B,C><C,D><D,E>} ساده است
2) {<F,C><C,D><D,B><B,C>} ساده نیست
3) {<A,F><F,C><C,B><B,A>} ساده است(حلقه یا سیکل)

حلقه به مسیر ساده ای گفته می شود که اولین و آخرین راس آن یکی باشد (مثل حالت ۳).

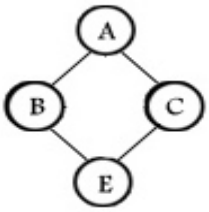
در یک گراف بدون جهت راس i , j متصل هستند اگر مسیری از i به j وجود داشته باشد.

گراف متصل یا هم بند

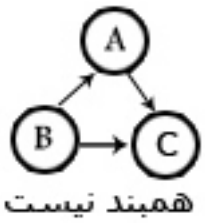
گرافی است که برای هر راس i و j در آن مسیری از i به j وجود داشته باشد (همه گره ها به هم متصل باشند).



همبند نیست



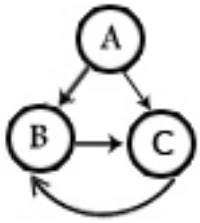
همبند است



همبند نیست

نکته: به گرافهای جهت دار گاهی اوقات دایگراف گفته می شود.

مثال) در گراف مقابل چند مسیر ساده به طول ۳ وجود دارد؟



جواب : در شکل مقابل هیچ مسیر ساده ای به طول ۳ وجود ندارد.

نکته: درخت یک گراف متصل بدون سیکل است.

نکته : در هر گراف ساده طول هر مسیر ساده کوچکتر یا مساوی n-1 است. (n تعداد گره‌های گراف می باشد)

درجه یک راس

در یک گراف بدون جهت تعداد لبه های متصل به راس i , درجه آن راس محسوب می شود اما در گراف جهت دار هر راس دارای درجه ورودی و درجه خروجی است.

تعداد یال هایی که به راس i وارد شده است درجه ورودی و تعداد یال هایی که از آن خارج شده است درجه خروجی آن است.

درجه گراف

درجه گراف برابر بزرگترین درجه راس های آن گراف است.

نکته: در یک گراف بدون جهت تعداد راس هایی که دارای درجه فرد هستند زوج خواهند شد.

نکته: اگر در یک گراف با n راس که درجه راس i آن d_i باشد آنگاه به وسیله فرمول زیر تعداد لبه های آن را می توان محاسبه کرد:

$$e = \frac{1}{2}\left(\sum_{i=1}^{n} di\right)$$ ⟵ جمع درجه ها

مثال) گرافی راس های آن دارای درجه های زیر است.تعداد یال های آن را محاسبه کنید.

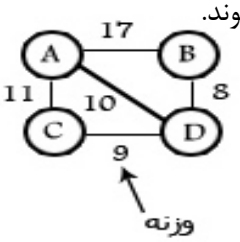
1-> 2 **2**-> 3 **3**->5 **4**-> 2 **5**-> 2 ⟵ یال 7 = 2/ (2+3+5+2+2)

گراف منظم

اگر در گرافی درجه تمام راس ها دقیقا برابر ۲ باشد به آن گراف منظم می گویند.

گراف منظم

نکته: گرافی که لبه هایش دارای وزن باشد شبکه نامیده می شوند.

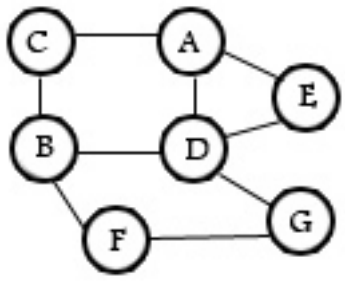


شبکه

نمایش گراف

نمایش گراف به دو صورت ماتریس مجاورتی (همجواری) و لیست مجاورتی است.

ماتریس مجاورتی (همجواری)



V=

	A	B	C	D	E	F	G
A	•	•	1	1	1	•	•
B	•	•	1	1	•	1	•
C	1	1	•	•	•	•	•
D	1	1	•	•	1	•	1
E	1	•	•	1	•	•	•
F	•	1	•	•	•	•	1
G	•	•	•	1	•	1	•

N•N تعداد راس ها

در این روش برای مجاور بودن دو راس i , j کافی است v[i,j] را بررسی کنیم.اگر ۱ بود یعنی مجاورند و اگر 0 بود یعنی مجاور نیستند.

در این روش اگر تعداد راس ها بسیار زیاد باشد و تعداد یال ها کم باشد ماتریس مجاورتی یک ماتریس اسپارس خواهد شد.

در گراف بدون جهت ماتریس مجاورتی همواره متقارن است لذا فقط مثلث بالا یا پایین را ذخیره می کنیم تا فضای کمتری مصرف شود ولی در گراف جهت دار ماتریس مجاورتی متقارن نیست.

مجموع عناصر سطر i یا ستون i در ماتریس مجاورتی برای گراف بدون جهت برابر درجه راس i می شود اما در گراف جهت دار جمع سطر i برابر درجه خروجی و جمع ستون i برابر درجه ورودی است.

الگوریتم های مربوط به گراف ها با نمایش گراف به صورت ماتریس مجاورتی ساده می شوند.

در روش ماتریس مجاورتی به راحتی می توان یال اضافه یا حذف کرد اما نمی توان راس اضافه کرد.ماتریس مجاورتی V^1 بیان گر مسیر های با طول ۱ در گراف است.اگر V^2 را محاسبه کنیم تعداد مسیرهای با طول ۲ بین هر دو راس به دست می آید.

مثلا اگر $V^2[i , j]=3$ باشد یعنی بین راس i , j , ۳ مسیر ساده با طول ۲ وجود دارد. اگر V^3 محاسبه شود تعداد مسیر های ساده با طول ۳ بین هر دو راس به دست می آید.

نکته: اگر یک گراف با n راس داشته باشیم حداکثر طول یک مسیر ساده n-1 خواهد شد.

بنابراین V^{n-1} مسیر های ساده بدون سیکل با حداکثر طول را مشخص می کند.

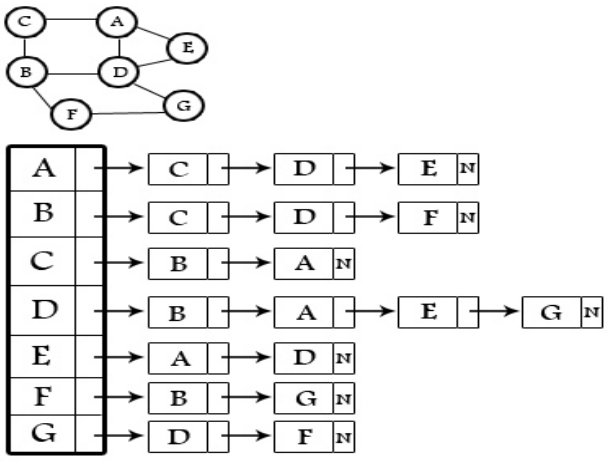
بررسی یک گراف برای هم بند بودن(متصل بودن)

اگر برای هر دو راس i , j در گراف , $V^{n-1}[i , j]$ و ..... و $V^2[i,j]$ و $V^1[i,j]$ را محاسبه می کنیم و همگی صفر باشند به معنای این است که گره i , j به هم متصل نیستند واگریکی از موارد ذکر شده غیر صفر بود دو راس i , j به هم متصل اند.

حال برای متصل بودن یک گراف باید این عمل را برای همه رئوس انجام دهیم. اگر همه راس ها به هم متصل بودند آن گاه گراف هم بند است.

حداقل مقدار فضایی که ماتریس مجاورتی می تواند اشغال کند n^2 بیت است. یکی از مشکلات ماتریس مجاورتی این است که , وزنه های یال ها و اطلاعات داخل راس ها را ذخیره نمی کند که می توانیم برای ذخیره وزنه ها به جای ۱ های داخل ماتریس , مقدار وزنه ها را قرار دهیم.

لیست مجاورتی



در این روش یک لیست پیوندی به نام head داریم که مشخصات هر راس را به همراه یک لیست پیوندی برای هر راس نگهداری می کند. هر لیست پیوندی گره های مجاور هر راس است. در این روش برای چک کردن مجاور بودن دو راس i , j باید لیست پیوندی را پیمایش کنیم که این کار دارای پیچیدگی زمانی O(n) است که نسبت به ماتریس مجاورتی بیشتر است.

نکته: در این روش می توان یال و راس ها را اضافه یا حذف کرد.

نکته: برای محاسبه درجه هر راس تعداد گره های هر لیست پیوندی را می شماریم.

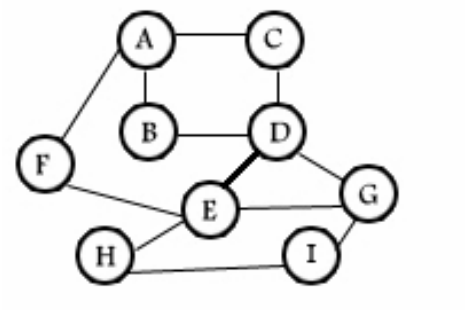
در یک گراف بدون جهت , لیست مجاورتی آن دارای n گره head و 2e عنصر در لیست پیوندی خواهد بود که e , تعداد لبه ها می باشد.

یکی از مشکلات این روش فضای اضافه مصرف شده برای یال ها است.الگوریتم های گراف که به این روش نمایش داده می شوند بسیار کند خواهد بود.

پیمایش گراف

منظور از پیمایش گراف ملاقات کردن همه راس ها با شروع از یک راس است.

برای پیمایش گراف دو روش عمقی (dfs) و سطحی (bfs) وجود دارد.



پیمایش عمقی (DFS)

در این روش از یک راس شروع می کنیم و آن را ملاقات می کنیم سپس به طرف یکی از فرزندان آن می رویم و در عمق , آن را پیمایش می کنیم.

نکته: چون در گراف ترتیب فرزندان اهمیتی ندارد , پیمایش گراف می تواند حالتهای بسیار زیادی داشته باشد.

مثلا با شروع از راس A سه حالت زیر می تواند پیمایش عمقی باشد.

1) ABDCEFGIH 2) ABDEHIGFC 3) AFEHIGDBC

نکته: پیمایش ABDGCEFHI به دلیل وجود C بعد از G نمی تواند پیمایش عمقی باشد.

برای پیمایش عمقی نیاز به یک آرایه به نام visit داریم (به طول تعداد راس ها) که ابتدا با صفر پر می شود. هر گره ای که ملاقات می شود اندیس متناظر آن را ۱ می کنیم.

Visit =

A	B	C	D	E	F
0	0	1	0	0	1

این آرایه به ما کمک می کند تا راس هایی که تا به حال ملاقات شده اند را به دست آوریم.

الگوریتم پیمایش عمقی گراف

از راس x شروع می کنیم و آن را ملاقات می کنیم (یعنی آن را چاپ می کنیم و visit آن را ۱ می کنیم). سپس فرزندان آن را که ملاقات نشده اند (به کمک visit) به هر ترتیب دلخواهی در یک پشته می ریزیم. حال هر بار یک عنصر از پشته خارج می کنیم و در صورتی که قبلا ملاقات نشده است , آن را ملاقات می کنیم و فرزندان آن را دوباره در صورتی که قبلا ملاقات نشده اند در پشته می ریزیم. این عمل را تا زمانی که پشته خالی نشده است تکرار می کنیم.

برنامه پیمایش عمقی (DFS) گراف

```
Void dsf (int v)
{
Visit[v]= true;
Cout<<v;
For (w=graph[v] ; w<>NULL;w=w->link)
        If (! Visit [w])
                Dfs(w);
}
```

نکته: در برنامه بالا که به صورت شبه کد است , فرض بر این است که گراف به وسیله یک لیست مجاورتی نمایش داده شده است.

نکته: آرایه graph همان head است.

پیمایش سطحی (BFS)

در پیمایش سطحی گره شروع می تواند هر گره ای باشد و ترتیب ملاقات فرزندان می تواند متفاوت باشد به همین دلیل حالتهای مختلفی رخ می دهد. در این پیمایش از یک آرایه به نام visit جهت نگه داری گره های ملاقات شده و یک صف استفاده می کنیم.

برای پیمایش سطحی ابتدا گره x را ملاقات کرده و فرزندان آن را به هر ترتیب دلخواهی در یک صف می ریزیم. سپس هر بار یک عنصر از صف خارج کرده در صورتی که قبلا ملاقات نشده اند , آن را ملاقات کرده و فرزندان آن را هر کدام که ملاقات نشده اند در صف می ریزیم .این کار را تا زمانی که صف خالی شود تکرار می کنیم.

برنامه پیمایش سطحی (BFS) گراف

```
Void bfs(int v)
{visit [v] =true;
Cout<<v;
For (w=graph [v] ; w<>NULL ; w=w->link)
        Queue.add(w);
While (queue.is empty ==false)
        {v=queue.exit;
           If (!visit [v])
                {visit [v]=true;
                Cout<<v;
                For (w=graph [v]; w<> NULL ; w=w->link)
                        If (! Visit [w])
                                Queue.add(w);
                }
        }
}
```

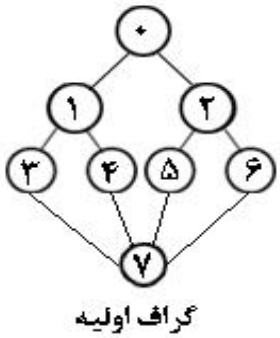
گراف متصل

گراف متصل , گرافی است که از هر راس آن به راس دیگر حداقل یک مسیر وجود داشته باشد.
الگوریتم های bfs , dfs می توانند تعیین کنند که آیا گرافی متصل است یا خیر. اگر الگوریتم های پیمایش گراف را با نقطه شروع هر راس دلخواهی اجرا کنیم و در نهایت یکی از راس ها ملاقات نشود یعنی گراف متصل نیست , اما اگر گراف را از هر راس که پیمایش می کنیم همه راس ها ملاقات شوند به معنای این است که گراف متصل است.

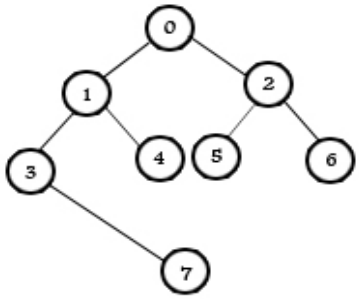
درخت پوشا (spanning tree)

درختی که تعدادی از لبه ها (یال ها) و تمامی رئوس G را در بر دارد , درخت پوشا نامیده می شود.
پیمایش های عمقی و سطحی , باعث ایجاد درختهای پوشای متفاوتی از گراف می شوند.درخت پوشای حاصل از پیمایش عمقی گراف ,درخت پوشای عمقی و درخت پوشای حاصل از پیمایش سطحی گراف , درخت پوشای سطحی نامیده می شود.

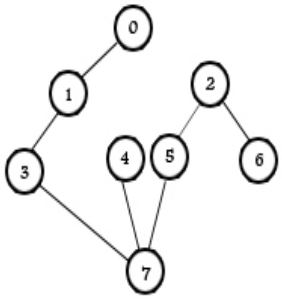
مثال) درخت پوشای dfs و bfs گراف زیر را ترسیم کنید.



گراف اولیه



پوشای Bfs



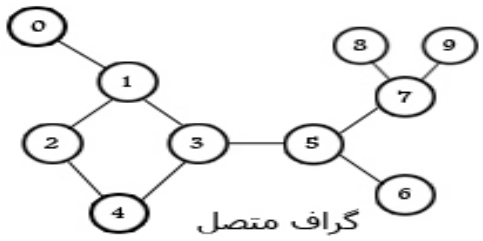
پوشای Dfs

نکته: هر گراف متصل با n راس , بایستی حداقل n-1 لبه داشته باشد و همه گراف های متصل با n-1 لبه , درخت هستند.بنابراین می توان نتیجه گرفت که درخت پوشا , دارای n-1 لبه است.

نقطه اتصال(بحران) و گراف دو اتصالی

نقطه اتصال , یک راس مانند V از گراف G می باشد, به نحوی که حذف راس V , همراه با تمام لبه های متلاقی با V , گرافی به نام G' ایجاد می کند که حداقل دارای دو جزء متصل است.

مثال) گراف متصل شکل زیر دارای چهار نقطه اتصال یعنی رئوس ۱و۳و۵و۷ می باشد.



گراف متصل

نکته: گراف **دو اتصالی** , یک گراف متصل است , اگر فاقد نقطه اتصال (بحرانی) باشد.
در خیلی از کاربرد های گراف به کار گیری نقطه اتصال رضایت بخش نیست. فرض کنید شکل گراف مثال قبل یک شبکه ارتباطات باشد. حال اگر یکی از ایستگاههایی که نقطه اتصال است خراب شود , حاصل کار نه تنها فقدان ارتباط با آن ایستگاه است بلکه بین ایستگاههای دیگر نیز ارتباطاتی قطع می شود.

درخت پوشای با حداقل هزینه

هزینه درخت پوشای یک گراف جهت دار دارای وزن, مجموع هزینه ها (وزن های) لبه ها در درخت پوشا می باشد.درخت پوشای حداقل هزینه , درخت پوشایی است که دارای کمترین هزینه باشد.در اینجا برای به دست آوردن درخت پوشای حداقل هزینه یک گراف جهت دار متصل , الگوریتم های راشال , پریم و سولین را شرح می دهیم.
روش ما برای تعیین درخت پوشا با حداقل هزینه باید سه شرط زیر را داشته باشد:

۱- باید فقط از لبه های داخل گراف استفاده کنیم.
۲- باید دقیقا از n-1 لبه استفاده کنیم.
۳- نباید از لبه هایی که ایجاد یک حلقه می کنند , استفاده کنیم.

الگوریتم راشال (کروسکال)

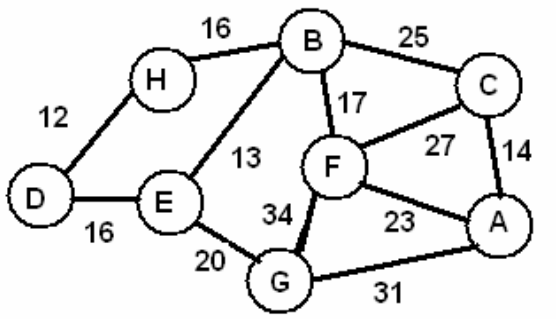
در این روش هر بار یک یال با کمترین هزینه را انتخاب می کنیم به طوری که باعث سیکل نشود و دقیقا n-1 لبه انتخاب می کنیم. در این الگوریتم ابتدا جنگل تولید می شود و در نهایت جنگل به درخت پوشا تبدیل می شود.

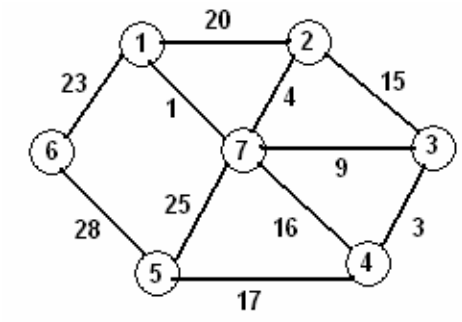
الگوریتم سولین

در این الگوریتم در مرحله اول برای هر راس یالی را انتخاب می کنیم که حداقل باشد و باعث سیکل نیز نشود. در پایان مرحله اول جنگل ایجاد می شود. در مرحله دوم درختان جنگل را با یال های حداقل به یکدیگر متصل می کنیم به طوری که باعث سیکل نشود.

الگوریتم پریم

در این الگوریتم جنگل تولید نمی شود بلکه از همان ابتدا درخت ایجاد می شود.در این الگوریتم از یک راس دلخواه شروع می کنیم و کمترین یال (یال حداقل) را انتخاب می کنیم و بقیه یال های آن را در یک مجموعه نگه داری می کنیم. یال انتخاب شده یک راس به مجموعه اضافه می کند .از بین یال های جدید و یال های مجموعه دیده شده کمترین یال را انتخاب می کنیم. این عمل n-1 بار انجام می شود.





مرتب سازی:

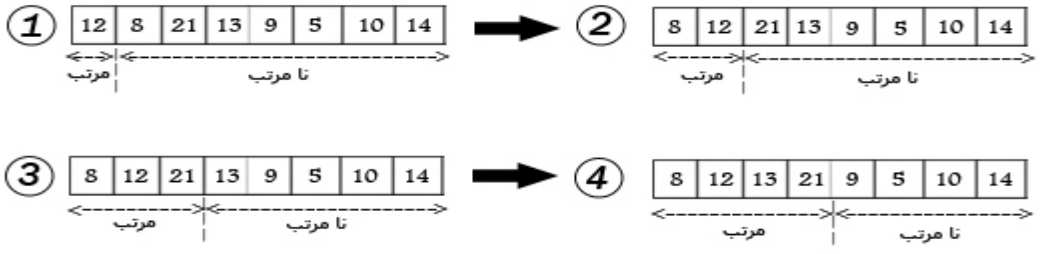
الگوریتم‌های مختلفی برای مرتب سازی n عنصر وجود دارد که می توان آن ها را از جهات گوناگون مورد بررسی قرار داد. هریک از این الگوریتم ها، براساس خواص خود ممکن است در مسایل ویژه کارکرد بهتری داشته باشند. در عناصری که مرتب می‌شوند، بخشی از هر عنصر مرتب شونده را که مرتب سازی بر اساس آن انجام می گیرد کلید (key) می گویند.
نکته: الگوریتم هایی که عناصری با کلید یکسان را جا بجا نمی کنند، الگوریتمهای متعادل (Stable) می گویند.
در هر الگوریتم مرتب سازی همیشه اعمال مقایسه و تعویض (Compare & Exchange) انجام می گیرد که تعداد این اعمال مرتبه اجرایی الگوریتم را مشخص می کنند .

رده بندی الگوریتم ها

الف- الگوریتم های مرتب کننده داخلی (in-place یا Internal)
در این الگوریتم‌ها، عناصر مرتب شونده همه در حافظه هستند و مقایسهٔ کلید، مهمترین عمل الگوریتم است .
ب- الگوریتم های مرتب کننده خارجی (out-place یا External)
در این الگوریتم ها عناصر مرتب شونده همه در حافظه اصلی قرار ندارند، و دسترسی به عناصر از مهمترین عوامل تعیین کننده زمان اجرای الگوریتم ها است
نکته: الگوریتم ها از نظر تعداد مقایسه در وضعیت های متوسط بهترین وبدترین حالت قرارمی گیرند .

مرتب سازی درجی (insertion sort)

این مرتب سازی با جابه جا کردن عناصر باعث مرتب شدن می شود.در این مرتب سازی محور در خانه دوم قرار می گیرد و فرض می کنیم که خانه اول مرتب است.سپس این محور به جلو می رود و سلول جدید را در مکان خود درج می کنیم.
این الگوریتم مرتب سازی الگوریتمی است که مرتب سازی را با درج رکورد ها (عناصر) در یک آرایه مرتب شده موجود , مرتب سازی می کند.
مثال) آرایه زیر را به روش درجی مرتب سازی کنید.



نکته: این نوع مرتب سازی هیچ گونه فضای اضافی مصرف نمی کند.
نکته) این الگوریتم متعادل پوده و در آرایه های مرتب بهترین حالت و برای آرایه های مرتب معکوس، بدترین عملکرد را دارد.

برنامه مرتب سازی درجی

```
Void insertsort(void)
{int I,j,x;
For (i=2 ; i<=n ; i++)
        {x=s[i];
        J=i-1;
        While (j>0   &&   s[j]>x)
```

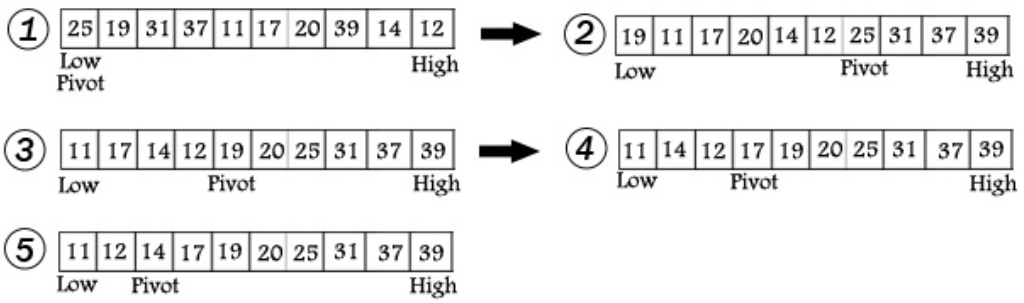
```
            {s[j+1]=s[j];
            j--;
            }
        S[j+1]=x;
        }
}
```

جدول پیچیدگی زمانی این نوع مرتب سازی به صورت زیر است.

	بهترین حالت	حالت متوسط	بدترین حالت
تعویض	$O(1)$	$O(n^2)$	$O(n^2)$
مقایسه	$O(n)$	$O(n^2)$	$O(n^2)$

مرتب سازی سریع (Quick sort)

در این روش آرایه به دو قسمت تقسیم می شود.در این روش یک عنصر به عنوان عنصر محور (pivot) انتخاب می شود و عناصر کوچکتر از محور در سمت چپ و عناصر بزرگتر از محور در سمت راست محور قرار می گیرند و آرایه تقسیم می شود (معمولا عنصر اول به عنوان محور در نظر گرفته می شود). بعد از این عمل که آرایه به دو قسمت چپ و راست تقسیم شد و محور در بین چپ و راست قرار گرفت دوباره برای قسمت چپ و راست نیز یک محور در نظر گرفته و سپس قسمتهای چپ و راست به دو قسمت چپ و راست تقسیم می شوند که این کار به صورت بازگشتی انجام می شود.این اعمال را تا زمانی که آرایه مرتب شود انجام می دهیم.



نکته: برای مرتب کردن قسمت چپ و راست باید دوباره pivot تعریف کنیم.

نکته: انتخاب محور (pivot) تاثیر بسیار زیادی در پیچیدگی زمانی دارد.

پیچیدگی زمانی این نوع مرتب سازی در حالتهای مختلف در جدول زیر شرح داده شده است.

	بهترین حالت	حالت متوسط	بدترین حالت
پیچیدگی زمانی	$O(n.\log n)$	$O(n.\log n)$	$O(n^2)$

برنامه مرتب سازی سریع

```
Void quicksort(int low , int high)
{int pivot=low;
If (high > low)
        {partition(low , high, pivot);
        Quicksort(low , pivot -1);
        Quicksort(pivot +1 ;high);
}       }
```

نکته: این برنامه نیاز به فضای اضافه دارد (استفاده از استک) زیرا حالت بازگشتی دارد.

نکته) در مرتب سازی سریع بدترین حالت زمانی رخ می دهد که عنصر محور کوچکترین یا بزرگترین عنصر آرایه باشد.

نکته) در مرتب سازی سریع اگر آرایه از قبل مرتب باشد بدترین حالت می تواند رخ دهد.

نکته) اگر عنصر محوری آرایه را به دو زیرآرایه تقریبا یکسان تقسیم کند در این صورت بهترین حالت رخ می دهد که بهترین روش مرتب سازی خواهد بود.

نکته) این الگوریتم متعادل(پایدار) نیست.یعنی عناصر با کلید یکسان را جابجا می کنند.

نکته) بطور کلی در آرایه های مرتب بدترین عملکرد و در آرایه های نامرتب بهترین عملکرد را دارد.

نکته: الگوریتم سورت سریع از نوع تقسیم وغلبه (Divide Conquce) است که بازه و محدوده عمل را به چند بازه کوچک تقسیم می کند.

مرتب سازی ادغامی (merge sort)

در این مرتب سازی از مساله ای به نام merge استفاده می شود.

اگر دو آرایه مرتب شده با طول های n/2 و n/2 داشته باشیم می توانیم آنها را طوری با هم ادغام کنیم که یک آرایه n تایی مرتب حاصل شود.

تمرین: برنامه ای بنویسید که دو آرایه ۱۰ عنصری مرتب را از کاربر دریافت کند وسپس این دو را با هم merge کرده تا یک آرایه ۲۰ تایی مرتب حاصل شود.(از الگوریتم های مرتب سازی استفاده نکنید).

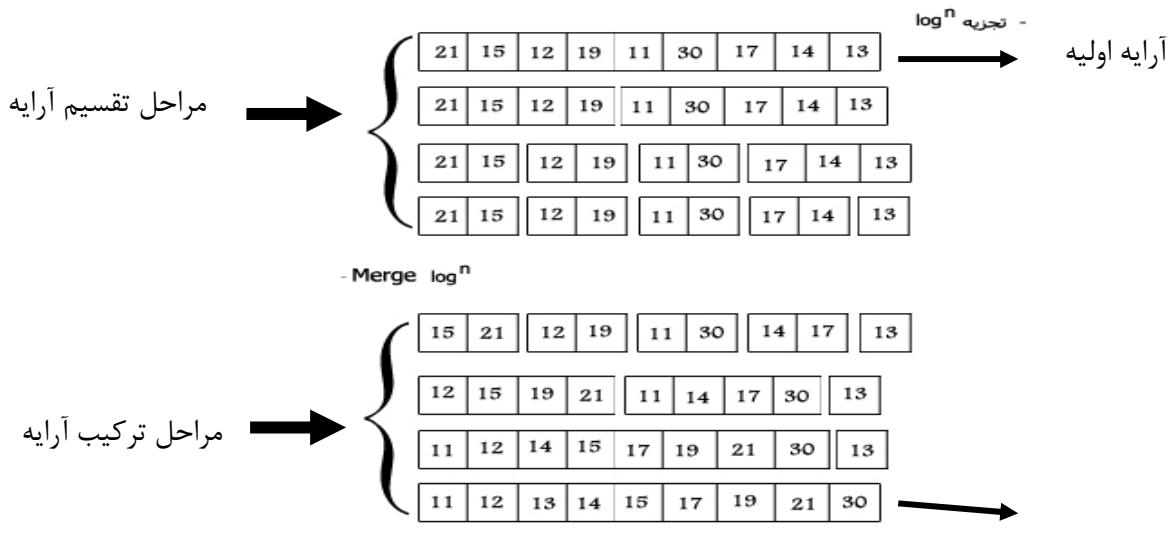
برای مرتب سازی یک آرایه به روش ادغامی ابتدا آرایه را نصف کرده و سپس ۲ آرایه را مرتب می کنیم.بعد از آنکه این دو آرایه کوچک مرتب شدند آنها را ادغام کرده تا آرایه اصلی مرتب شود.

برای مرتب کردن آرایه های کوچک به دست آمده دوباره آنها را نصف می کنیم. عمل نصف کردن را تا زمانی که تعداد عناصر هر آرایه بیشتر از ۱ است تکرار می کنیم.

جدول پیچیدگی زمانی این روش به صورت زیر است.

پیچیدگی زمانی	بهترین حالت	حالت متوسط	بدترین حالت
O (n.log n)	O (n.log n)	O (n.log n)	O (n.log n)

مثال) آرایه زیر را به روش ادغامی مرتب کنید.



آرایه مرتب

نکته: در مرتب سازی ادغامی با توجه به اینکه آرایه همیشه به دو قسمت تقسیم می شود به فضای اضافه جهت مرتب سازی نیاز دارد.

برنامه مرتب سازی ادغامی

```
Void mergesort( int n, int s[])
{int h=n/2;
Int m=n-h;
Int u[1..h] , v[1..m];
If (n>1)
        {copy s[i] to s[h] in u;
        Copy s[h+1] to s[n] in v;
        Mergesort (h,u);
        Mergesort (m,v);
        Merge(h,m,u,v,s);
        }
}
```

نکته) مرتب سازی ادغامی پایدار است.

نکته) عیب این الگوریتم این است که برای مرتب کردن به یک آرایه کمکی با n عنصر نیازمند است.

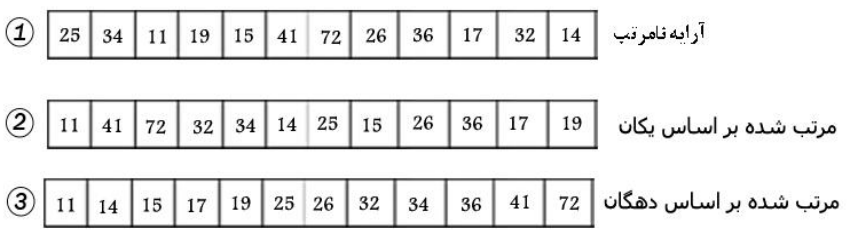
مرتب سازی مبنایی (Base sort)(Radio Sort)

در این روش برای مرتب کردن اعداد به ۱۰ صف (q_0 .... q_9) نیاز داریم. در این روش ابتدا اعداد را بر اساس یکان داخل صف ها می ریزیم.سپس به ترتیب اولویت از q_0 تا q_9 خارج می کنیم و در آرایه می ریزیم. دوباره این عمل را برای دهگان انجام می دهیم . در این صورت آرایه مرتب می شود.

نکته: تعداد دفعاتی که باید در صف بریزیم و خارج کنیم بستگی به تعداد ارقام عدد دارد.

مثال) آرایه زیر را به روش مبنایی مرتب کنید.

صف	بر اساس یکان	بر اساس دهگان
q0		
q1	11,41	11,14,15,17,19
q2	72,32	25,26
q3		32,34,36
q4	34,14	41
q5	25,15	
q6	26,36	
q7	17	72
q8		
q9	19	



نکته: برای مرتب کردن رشته ها به روش مبنایی به ۲۶ صف نیاز داریم.

نکته: پیچیدگی زمانی این روش مرتب سازی بستگی به محتوای آرایه دارد (عدد باشد یا حرف).

جدول پیچیدگی زمانی این روش به صورت زیر است.

	بهترین حالت	حالت متوسط	بدترین حالت
پیچیدگی زمانی	$O(S * n)$	$O(n.\log n)$	$O(n^2)$

تعداد ارقام

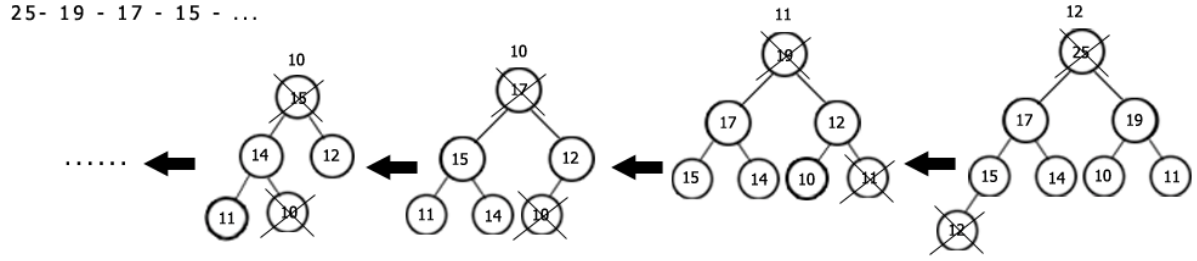
نکته: در این روش برای ایجاد صف ها فضای اضافی مصرف می کنیم.

مرتب سازی heap (heap sort)

در این الگوریتم مجموعه ای از اعداد نامرتب داریم. آنها را یکی یکی وارد یک درخت max heap می کنیم به طوری که در نهایت یک درخت max heap کامل به دست آید.

نکته: برای ساختن یک درخت max heap با n تا عنصر $O(n.\log n)$ زمان مصرف می شود.

حال اگر از ریشه یکی یکی حذف کنیم داده ها مرتب می شوند.



نکته: برای حذف کردن n تا عنصر , پیچیدگی زمانی اجرا $O(n.\log n)$ می شود.

جدول پیچیدگی زمانی این روش به صورت زیر است.

بدترین حالت	حالت متوسط	بهترین حالت	پیچیدگی زمانی
$O(n.\log n)$	$O(n.\log n)$	$O(n.\log n)$	

نکته: برای مرتب سازی صعودی به جای max heap از min heap استفاده می کنیم.

نکته: با توجه به اینکه می توان یک درخت را به کمک آرایه نمایش داد در الگوریتم heap sort نیازی به فضای اضافه نخواهیم داشت یعنی نیازی به ساختن یک درخت به کمک اشاره گرها نیست بلکه درخت را درآرایه مربوطه نگه داری می کنیم که این روش در درسهای قبل بیان شده است.

مرتب سازی درختی:BST

در این روش از درخت جستجوی دودویی BST برای مرتب سازی استفاده می شود.اگر درخت BST را پیمایش Inorder شود دنباله بدست آمده مرتب صعودی خواهد بود.

کارایی نسبی این روش به ترتیب اولیه داده ها بستگی دارد.اگرآرایه ورودی کاملا مرتب باشد درخت جستجوی دودیی مورب بدست می آید که در این صورت برای ساخت درخت و مرتب کردن آن $O(n^2)$ زمان نیاز خواهیم داشت.

بدترین حالت	حالت متوسط	بهترین حالت	پیچیدگی زمانی
$O(n^2)$	$O(n.\log n)$	$O(n.\log n)$	

مشکل اصلی مرتب سازی درخت دودویی نیز مانند مرتب سازی ادغامی است که نیاز به فضایی به طول N برای ایجاد درخت است.

نکته: مقایسه تعداد جابه جایی ها در سه الگوریتم مختلف مرتب سازی به صورت زیر است:

حبابی (تعویضی) > درجی (insertion) > انتخابی (selection)

نکته: معمولا مرتب سازی سریع (quick sort) بهتر از مرتب سازی های دیگر عمل می کند.

مرتب سازی انتخابی:

مرتب سازی حبابی:

مرتب سازی حبابی عمل مرتب سازی را با تعویض مرتب عناصر در یک بردار انجام می دهد. در این الگـوریتم n بـار در طـول بردار حرکت انجام می شود و یک عنصر با عنصر بعدی مقایسه و در صورت لزوم جابجایی انجام می گیرد. پس در نخستین بار حرکت در بردار، بزرگترین یا کوچکترین عنصر در انتهای بردار قرار می گیرد و در i امین بار، عناصـر مکـان هـای n-i تـا n در جای صحیح قرار گرفته‌اند. برای بهبود الگوریتم می توان در مرتبه i ام، طی کردن بردار عناصر را تا عنصر n-i انجام داد و بـه کمک یک flag، در صورتی که هیچ تعویضی در یک بار طی کردن انجام نشود، بردار مرتب شده است و الگوریتم خاتمـه مـی یابد.

1) مرتبه اجرایی تکه برنامه روبرو کدام گزینه است؟

```
a:=n;
while (a>1) {
      for  b=1  to  m     print ″Yes″
      For  c=1  to  K     Print ″No″
      a:= a  div  2
  }
```

الف) O(m.k.Log n)

ب) O(m.k.n)

ج) O((m+k).Log a)

د) O((m+k).Log n)

2)فرض کنید صف Q دارای تعدادی عنصر است شبه کد روبرو چه عملی را انجام می دهد؟S نیز یک استک است که ابتدا خالی است.

```
While ( Q.Isempty=Flase) {
     S.Push( Q.Delete)
}
While (S.Isempty=Flase) {
     Q.ADD(S.POP)
}
```

الف) عناصر داخل صف را معکوس می کند

ب) عناصر را به صورت معکوس داخل استک می ریزد

ج) داده های داخل صف را مرتب می کند

د)داده ها را از صف داخل استک می ریزد

3)کدام گزینه صحیح است؟

الف) اتلاف حافظه و ایستا بودن لیست های پیوندی از معایب آن محسوب می شود

ب) زمان دسترسی به عنصر اول و عنصر آخر آرایه با یکدیگر یکسان است

ج) دسترسی به عناصر لیست پیوندی نسبت به آرایه سریع تر انجام می شود

د) عناصر یک لیست پیوندی در حافظه پشت سر هم هستند.

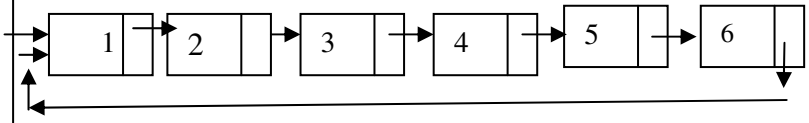
4)دلیل استفاده از نمادگذاری پسوندی بجای میانوندی برای عبارات در کامپیوتر چیست؟

الف) نمایش بهتر و آسان تر ب) حذف پرانتزها از عبارت میانوندی

ج) بی اثر کردن الویت عملگرها د) ب و ج

5)فرض کنید قطعه برنامه روبرو روی لیست یک طرفه حلقوی زیر اعمال شود خروجی نهایی چیست؟

```
While (L->next != L)
  {
    L = L->Next->Next;
    P = L->Next ;
    L->Next = P->Next;
  }
Cout<< L->Data;
```



L اشاره گری به ابتدای لیست است.

الف) 1 ب) 2 ج) 6 د) 5

6)اگر Start اشاره گری به ابتدای یک لیست یکطرفه باشد اثر اجرای قطعه دستورات زیر چیست؟

```
P=Start;
T=Null;
While (P != NULL)
  {
    R=T;
    T=P;
    P=P->Next;
    T->Next=R;
  }
```

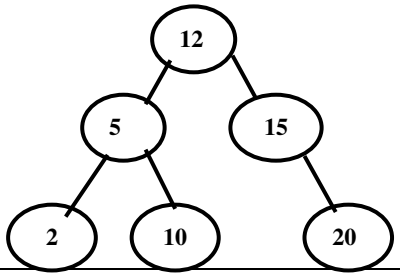
الف) لیست را با سه اشاره گر پیمایش کرده و Strat را به گره آخر اشاره می دهد

ب) لیست را مرتب می کند

ج) لیست یکطرفه را به لیست حلقوی تبدیل می کند

د) لیست را معکوس می کند

7) درخت روبرو چه نوع درختی است؟



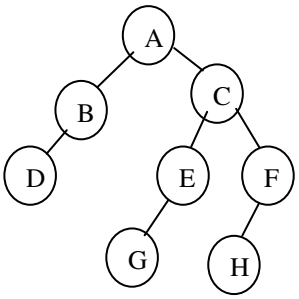
الف) درخت جستجوی دودویی

ب) Max Heap

ج) درخت دودویی کامل

د) درخت دودویی پر

8) پیمایش Postorder درخت روبرو کدام گزینه است؟



الف)DBAGECHF

ب)ABDCEGFH

ج) DBGEHFCA

د) DBGEFHCA

9)گراف کاملی دارای ۱۰ راس است چند یال از آن حذف شود درخت پوشا ایجاد می شود؟

الف) 9 ب) 45 ج)36 د)10

۱۰) الگوریتم زیر مربوط به کدام مرتب سازی است؟

```
Void   what(int   X[], int    n)
{
    int  i, k , y;
  for  ( i=1 ; i<= n ;  i++)
 {
      y= x[i] ; k= i -1 ;
    while ( k> 0  && y>X[i] )
       {
          X[ k+1] =X[ k] ;
          k=k-1;
       }
   }
}
```

الف) مرتب سازی درجی صعودی

ب) مرتب سازی درجی نزولی

ج) مرتب سازی حبابی صعودی

د) مرتب سازی حبابی نزولی

ضمیمه : ساختمان داده در پاسکال:

استفاده از اشاره گرها در دسترسی به متغیرها:

```
var
  ptr:^integer;
  z:integer;
begin
   z:=250;
   ptr^:=z;
   writeln(z, ptr^);
   ptr^:=ptr^+3;
   writeln(ptr^);
end.
```

یک مثال دیگر:

```
var
  ptr:^integer;
begin
  new(ptr);
  ptr^:=200;
  writeln(ptr^);
  dispose(ptr);
  new(ptr);
  ptr^:=450;
  writeln(ptr^);
  dispose(ptr);
end.
```

استفاده از حافظه پویا در ایجاد رکورد درحین اجرای برنامه:

```
type
   node=RECORD
     name:string[20];
     avg:real;
    end;
var
  student:^node;
begin
   new(student);
   student^.name:='ALI';
   student^.avg:=13.5;
   dispose(student);
end.
```

یک برنامه جامع برای لیست های پیوندی:

```
type
  addres=^node;
  node=record
    name:string[20];
    avg:real;
    next:addres;
```

```
  end;
var
  first:addres; c:char;
{**********************}
```
procedure addnode;
```
  var
    ptr:addres;
begin
  if first=nil then    اگر گره اول باشد
     begin
        new(first);
        write('enter a name and nuber=>');
        readln(first^.name, first^.avg);
        first^.next:=nil;
     end
     else
     begin
          new(ptr);
          write('enter a name and nuber=>');
          readln(ptr^.name, ptr^.avg);
          ptr^.next:=first;
          first:=ptr;
     end;
end;
{************************}
```
procedure list;
```
   var
      ptr:addres;
  begin
     ptr:=first;
     while ptr<>nil do
       begin
          writeln(ptr^.name , ptr^.avg:0:2);
          ptr:=ptr^.next;
       end;
  readln;
  end;
{**********************************}
```
procedure Deletenode;
```
 var
    ptr1,ptr2:addres;    n:string[20];
 begin
    writeln(' enter a name for delete');
    readln(n);
    if first=nil then
      exit;
    if first^.name=n then
      begin
        ptr1:=first;
        first:=first^.next;
```

```
      dispose(ptr1);
    end
    else
     begin
        ptr1:=first;
        ptr2:=first;
      while (ptr2<>nil) do
        begin
         if ptr2^.name=n then
          begin
            ptr1^.next:=ptr2^.next;
            dispose(ptr2);
            break;
          end;
          ptr1:=ptr2;
          ptr2:=ptr2^.next;
        end;  // end while پایان
    end;   // end else پایان
end;
{**********************************}
```

procedure search;

```
 var
  ptr:addres;
  m:string[20]; B:boolean;
 begin
  ptr:=first;
  writeln(' enter a name for search');
  readln(m);
  B:=false;
  while ptr<> nil do
   begin
     if ptr^.name=M then
       begin
         B:=True;
         writeln(ptr^.name , ptr^.avg:0:2);
         break;
       end;
      ptr:=ptr^.next;
   end;
  if B=False then
   writeln(' Not Found');
{************************}
```

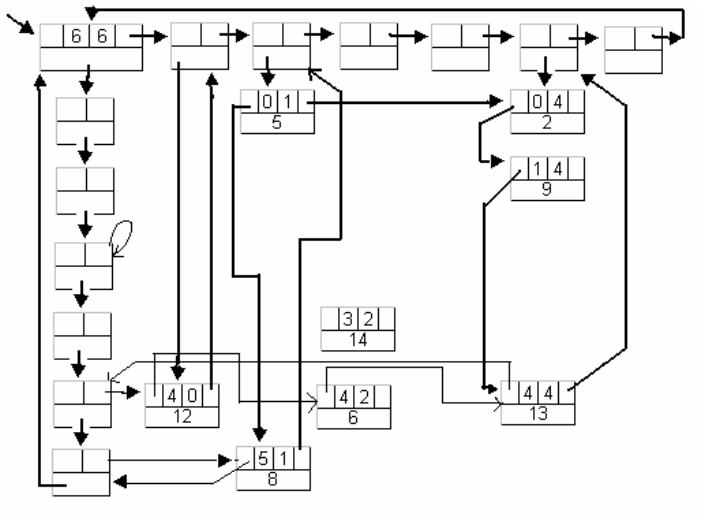
begin بلاك اصلي

```
 repeat
   writeln(' [A] for add node');
   writeln(' [L] for list of node');
   writeln(' [D] for delete a node');
   writeln(' [S] for search a node');
   writeln(' [Q] for exit');
   readln(c);
```

```
    case c of
      'A', 'a'  : addnode;
      'L', 'l'  :list;
      'D', 'd'  :deletenode;
      'S', 's'  :search;
    end;
   until ((c='q') or (c='Q'));
end.
```

نمایش ماتریس اسپارس بوسیله لیست های پیوندی:

برای نمایش یک ماتریس اسپارس می توان از لیست های پیوندی حلقوی استفاده کرد



0	5	0	0	2	0
0	0	0	0	9	0
0	0	0	0	0	0
0	0	14	0	0	0
12	0	6	0	13	0
0	8	0	0	0	0

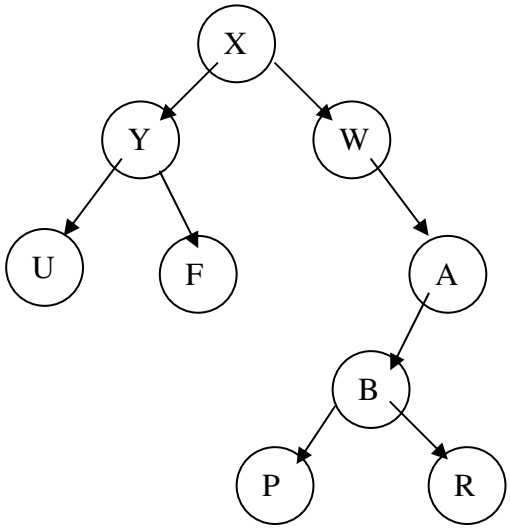
در شکل روبرو نحوهٔ اتصال گره های
هر ستون و هر سطر نمایش داده شده است.
البته کامل نیست آنرا کامل کنید؟
هر سطر یک لیست پیوندی حلقوی است
و هر ستون نیز یک لیست پیوندی حلقوی است.

درخت ها و پیمایش آنها:

Lchild	Data	RChild

```
type
  Adress=^ node;
  Node=Record
    Data:integer;
    Lchild:Adress;  // اشاره گر مربوط به فرزند سمت چپ
    Rchild:Adress; // اشاره گر مربوط به فرزند سمت راست
  end;
var
 ROOT:Adress;
{*****************************}
procedure Preorder(Ptr:Adress); // پیمایش پیشوندي
   begin
      if Ptr<>NIL  then
        begin
           Write(Ptr^.Data);
           Preorder(Ptr^.Lchild);
           Preorder(Ptr^.Rchild);
        end;

{************************************}
```



```
procedure Postorder(Ptr:Adress); //نمايش پسوندي
  begin
     if Ptr<>NIL  then
       begin
          Postorder(Ptr^.Lchild);
          Postorder(Ptr^.Rchild);
          Write(Ptr^.Data);
       end;
  end;
{*************************************}
procedure Inorder(Ptr:Adress);//پيمايش ميانوندي
  begin
     if Ptr<>NIL  then
       begin
          Inorder(Ptr^.Lchild);
          Write(Ptr^.Data);
          Inorder(Ptr^.Rchild);
       end;
  end;
{*************************************}

begin
  ROOT:=nil;
end.
```

پیمایش غیر بازگشتی درخت دودویی :

با استفاده از یک پشته می توان این عمل را انجام داد.

ساخت درخت دودویی با استفاده از پیمایش آن:

اگر پیمایش Inorder و یکی از دو پیمایش PostOrder و Preorder یک درخت داشته باشیم می توانیم خود درخت را رسم کنیم.

- اگر پیمایش Preorder مشخص باشد اولین گره ، ریشه است. اگر نمایش Postorder مشخص باشد آخرین گره، ریشه است.
- وقتی گره ریشه مشخص شد تمام گره های زیر درخت چپ و زیر درخت راست آن را می توان پیدا کرد.

مثال: با توجه به پیمایش های Inorder و Preorder درخت دودویی درخت را رسم کنید؟

Inorder➔ D B H E A I F J C G
Preorder ➔ A B D E H C F I J G

مثال) با توجه به پیمایش های Inorder و Postorder درخت دودویی درخت را رسم کنید؟

Inorder ➔ n1 n2 n3 n4 n5 n6 n7 n8 n9
Postorder ➔ n1 n3 n5 n4 n2 n8 n7 n9 n6

سئوال) با توجه به پیمایش Inorder و Postorder درخت دودویی درخت را رسم کنید؟

Inorder➔ E I C F J B G D K H L A
Postorder➔ I E J F C G K L H D B A

تمرینهای ساختمان داده:

۱- برنامه ایی بنویسید که **صف** را با استفاده از لیست پیوندی شبیه سازی کند؟

```
type
  Addres=^Node;
  Node=Record
     Data:integer;
     Next:Addres;
  end;
var   First:Addres;
{********************}
procedure ADDQueue;
  var     D:integer;     ptr:Addres;
begin
   if First=NIL  then
      begin
         new(First);
         writeln('Enter a number');
         Readln(First^.Data);
         First^.Next:=Nil;
      end
      else
        begin
         new(Ptr);
         writeln('Enter a number');
         Readln(Ptr^.Data);
         Ptr^.Next:=First;
         First:=Ptr;
        end;
end;
{********************}
function DeletefromQueue:integer;
var   Ptr1,Ptr2:Addres;
begin
    Ptr1:=First; Ptr2:=First;
    if First=Nil then
     begin
       DeletefromQueue:=0;
       Writeln('Qeue is Empty');
     end
    else
       begin
          while Ptr2^.Next <> NIL do
            begin
              Ptr1:=Ptr2;
              Ptr2:=Ptr2^.Next;
            end;
          DeletefromQueue:=Ptr2^.Data;
          Dispose(Ptr2);
          Ptr1^.next:=Nil;
```

```
      end;
end;
begin
  First:=NIL;
  AddQueue;
  AddQueue;
  AddQueue;
  writeln(DeletefromQueue);
  AddQueue;
  writeln(DeletefromQueue);
  writeln(DeletefromQueue);
  writeln(DeletefromQueue);
end.
```

۲-زیربرنامه ایی بنویسید که برای یک لیست پیوندی ساده **تعداد** گره های آن را چاپ کند؟

```
Procedure   NumberOfNode;
   Var     Ptr:Addres;  C:integer;
Begin
    C:=0;
    Ptr:=First;
   While  Ptr <> NIL  Do
     Begin
       C:=C+1;
      Ptr:=Ptr^.Next;
     End;
  Writeln( C);
End;
```

۳-زیربرنامه ایی بنویسید که عمل اضافه کردن به لیست پیوندی ساده را در انتها انجام دهد؟

```
Procedure  AddToEnd;
  Var       Ptr :Addres;
Begin
      Ptr:=First;
   If    First = NIL then
   Begin
      New(First);
      Writeln('Enter a number ');
      Readln(First^.Data);
      First^.Next:=Nil;
   End
   Else
       Begin
            While  Ptr^.Next<> NIL  Do
                 Ptr:= Ptr^.Next;
             New(Ptr^.Next);
             Ptr:=Ptr^.Next;
             Writeln('Enter a number ');
              Readln(Ptr^.Data);
              Ptr^.Next:=Nil;
       End;
End;
```

٤-زیربرنامه ایی بنویسید که یک گره با آدرس Ptr را در یک لیست پیوندی **دو** طرفه حذف کند.

این برنامه با فرض این نوشته شده است که گره ایی که قرار است حذف شود قبلا جستجو شده و آدرس آن را که Ptr است را داریم.

```
Procedure   Delete( Ptr : Address);
Begin
  If   First = Ptr  Then   //گره ايي كه قرار است حذف شود گره اول است
     Begin
       First:=First^.Next;
       First^.Last := Nil;
       Dispose(Ptr);
     End
  Else
     Begin
         Ptr^. Last ^ . Next := Ptr ^ .Next;
         Ptr^ . Next ^ .Last:=Ptr^ . Last;
         Dispose(Ptr);
     End;
End;
```

٥-زیر برنامه ایی بنویسید که با فرض اینکه یک لیست پیوندی ساده داریم **همه گره های آن را حذف** کند؟

```
Procedure   DeleteAll;
 Var    Ptr:Addres;
 Begin
     While  First <> NIL  Do
     Begin
          Ptr:=First;
          First:=First^.Next
          Dispose(Ptr);
     End;
End;
```

٧-الگوریتمی را طراحی کنید که یک لیست پیوندی ساده را معکوس کند؟

با فرض اینکه یک پشته داریم که می توان عملیات Push و Pop و Isempty را دارد این زیربرنامه را می نویسیم.

```
Procedure  INVERSE;
Var  Ptr: Addres;
Begin
  Ptr := First;
   While  Ptr <> NIL  Do
   Begin
      Push(Ptr);  // آدرس ها را داخل پشته مي ريزيم
       Ptr:= Ptr ^ . Next;
   End;
First := POP;  // آخرين گره از پشته خارج شده و در ابتداي ليست قرار مي گيرد
Ptr := First;
While   Not  Isempty  Do   //تا زماني كه پشته خالي نشده است
   Begin
       Ptr^.Next:= POP;
        Ptr:=Ptr ^. Next;
```

```
    End;
  Ptr ^. Next := NIL; // اشاره گر آخرين گره را تهي مي كنيم
  End;
```

۳-برنامه ایی بنویسید که عملیات مربوط به **پشته**(Pop , Push) را با یک لیست پیوندی انجام دهد؟

```
type
  Addres=^Node;
  Node=Record
    Data:integer;
    Next:Addres;
  end;
var
   First:Addres;
{********************}
procedure Push;
  var D:integer;  ptr:Addres;
begin
   if First=NIL  then
     begin
        new(First);
        writeln('Enter a number');
        Readln(First^.Data);
        First^.Next:=Nil;
     end
     else
       begin
        new(Ptr);
        writeln('Enter a number');
        Readln(Ptr^.Data);
        Ptr^.Next:=First;
        First:=Ptr;
       end;
end;
{********************}
function Pop:integer;
var   Ptr:Addres;
begin
    Ptr:=First;
    if First=Nil then
     begin
       Pop:=0;
       Writeln('Stack is Empty');
     end
    else
        begin
          pop:=First^.Data;
          First:=First^.Next;
          Dispose(Ptr);
        end;
end;
```

```
begin
  First:=NIL;
  Push;
  Push;
  Push;
  writeln(Pop);
  Writeln(Pop);
  push;
  Writeln(Pop);
  writeln(Pop);
end.
```

۸-زیربرنامهٔ بازگشتی بنویسید که یک لیست پیوندی ساده را به صورت **معکوس چاپ** کند؟

این زیربرنامه در بلوک اصلی به صورت ;PRINTINVERSE(First) فراخوانی می شود.

```
Procedure   PRINTINVERSE( PTR : Addres);
 Begin
    IF  PTR^.Next = NIL  THEN
       Wrietln(PTR^. DATA)
   Else
    Begin
          PRINTINVERSE(PTR^.Next);
        Writeln(PTR^.DATA);
    End;
 End;
```