ساختمان داده‌ها و الگوریتمها

مولفین

مهندس جعفر تنها   مهندس سید ناصر آیت

فصل اول

روش‌های تحلیل الگوریتم

اهداف

در پایان این فصل شما باید بتوانید:

- ✓ خصوصیات کلی یک الگوریتم را تعریف کنید.
- ✓ مرتبه زمانی یک الگوریتم را تعیین کنید.
- ✓ به تشریح نمادهای نشان دهنده کارایی یک الگوریتم بپردازید.
- ✓ مقادیر بازگشتی، یک الگوریتم بازگشتی را محاسبه کنید.
- ✓ به حل یک رابطه بازگشتی داده شده بپردازید.

سوالهای پیش از درس

۱- به نظر شما چگونه می توان فهمید یک برنامه نوشته شده از برنامه مشابه دیگر بهتر عمل می کند؟

....

....

۲- دلیل استفاده از الگوریتم بازگشتی به جای الگوریتم ترتیبی چیست؟

....

....

۳- به نظر شما در کامپیوترهایی با سرعت پردازش زیاد امروزی، آیا ارزش این را دارد که اثبات کنیم یک برنامه سریعتر از برنامه دیگری اجرا می شود؟

....

....

مقدمه

سوالی که در مورد یک الگوریتم یا الگوریتم‌های یک مسئله مطرح می شود اینست که کدام الگوریتم برای حل یک مسئله خاص بهتر عمل می‌کند؟ پاسخ دادن به این سوال به‌راحتی امکانپذیر نیست. مشخصه‌های زیادی از جمله سادگی، وضوح و زمان اجرا وغیره برای یک الگوریتم خوب می‌باشند. در این میان زمان اجرا، نقش بسیار مهمی ایفا می‌کند و غالباً کارایی برنامه را با زمان اجراء بررسی می‌کنند. در این فصل رفتار الگوریتم را قبل از پیاده‌سازی، از نظر زمان اجراء و کارایی بررسی می‌کنیم.

۱. ۱ زمان اجرای الگوریتم‌ها

همانطور که در بالا اشاره کردیم زمان اجرای یک الگوریتم از مسائل مهم طراحی الگوریتم می‌باشد. و غالباً کارایی الگوریتم‌ها را از روی زمان اجرای آنها بررسی می‌کنند (تنها معیار برای مقایسه نیست).

همانطور که می‌دانیم الگوریتم عبارتست از:

مجموعه‌ای از دستورات و دستورالعمل‌ها برای حل مسئله، که شرایط زیر را داراست:

- دقیق باشد
- مراحل آن به‌ترتیب انجام پذیرد
- پایان‌پذیر باشد

الگوریتم‌ها، توسط زبانهای برنامه‌نویسی پیاده‌سازی می‌شوند. و هر الگوریتم توسط یک برنامه (program) ارائه می‌شود (با هر زبان برنامه‌نویسی).

همچنین، هر برنامه مثل الگوریتم زمان اجرای خاص خود را دارد. بحث را از عوامل دخیل در زمان اجرای برنامه شروع می کنیم .

عوامل دخیل در زمان اجرای برنامه عبارتند از:

- سرعت سخت‌افزار

- نوع کامپایلر
- اندازه داده ورودی
- ترکیب داده‌های ورودی
- پیچیدگی زمانی الگوریتم
- پارامترهای دیگر که تأثیر ثابت در زمان اجرا دارند.

از این عوامل، سرعت سخت‌افزار و نوع کامپایلر به صورت ثابت در زمان اجرای برنامه‌ها دخیل هستند. پارامتر مهم، پیچیدگی زمانی الگوریتم است که خود تابعی از اندازه مسئله می‌باشد. ترکیب داده‌های ورودی نیز با بررسی الگوریتم در شرایط مختلف قابل اندازه‌گیری می‌باشد (در متوسط و بدترین حالات).

با توجه به مطالب بالا اهمیت زمان اجرای الگوریتم در یک برنامه، نرم‌افزار و غیره به وضوح مشاهده می‌گردد. لذا در ادامه سعی در بررسی پیچیدگی زمانی الگوریتم‌ها خواهیم داشت.

برای بررسی یک الگوریتم تابعی به‌نام $T(n)$ که تابع زمانی الگوریتم نامیده می شود، در نظر می‌گیریم. که در آن n اندازهٔ ورودی مسئله است. مسئله ممکن است شامل چند دادهٔ ورودی باشد. به‌عنوان مثال اگر ورودی یک گراف باشد علاوه بر تعداد راس‌ها (n)، تعداد یال‌ها (m) هم یکی از مشخصه‌های دادهٔ ورودی می‌باشد. دراینصورت زمان اجرای الگوریتم را با $T(n,m)$ نمایش می‌دهیم، در صورتی که تعداد پارامترها بیشتر باشند، آنهایی که اهمیت بیشتری در زمان اجرا دارند، را در محاسبات وارد می‌کنیم و از بقیه صرف‌نظر می‌کنیم.

برای محاسبه تابع زمانی $T(n)$ برای یک الگوریتم موارد زیر را باید در محاسبات در نظر بگیریم:

- زمان مربوط به اعمال جایگزینی که مقدار ثابت می‌باشند
- زمان مربوط به انجام اعمال محاسبات که مقدار ثابتی دارند.
- زمان مربوط به تکرار تعدادی دستور یا دستورالعمل (حلقه‌ها)

• زمان مربوط به توابع بازگشتی

از موارد ذکر شده در محاسبه زمان $T(n)$ یک الگوریتم محاسبه تعداد تکرار عملیات و توابع بازگشتی، اهمیت ویژه‌ای دارند. و در حقیقت در کل پیچیدگی زمانی مربوط به این دو می‌باشد.

۲. ۱ مرتبه اجرای الگوریتم

در ارزیابی الگوریتم دو فاکتور مهمی که باید مورد توجه قرار گیرد، یکی حافظه مصرفی و دیگری زمان اجرای الگوریتم است. یعنی الگوریتمی بهتر است که حافظه و زمان اجرای کمتری را نیاز داشته باشد. البته غالباً در الگوریتم‌های این کتاب فاکتور مهمتر، زمان اجرای الگوریتم می‌باشد. برای بررسی محاسبه اجرای الگوریتم‌ها کار را با چند مثال شروع می‌کنیم.

قطعه برنامه زیر را در نظر بگیرید:

```
(1)    x = 0 ;
(2)    for ( i = 0 ;  i < n ; i++ )
(3)        x++ ;
```

در قطعه کد بالا عملیات متفاوتی از جمله جایگزینی، مقایسه و غیره انجام می‌گیرد که هر کدام زمانهای متفاوتی را برای اجراشدن نیاز دارند. تابع زمانی قطعه کد بالا را می‌توان بصورت زیر محاسبه کرد:

سطر	زمان	تعداد
1	C_1	۱
2	C_2	$n+1$
3	C_3	n

با توجه به جدول، $T(n)$ برابر است با:

$$T(n) = C_1 + C_2(n+1) + C_3 n$$

حال C را بیشترین مقدار C_1، C_2، C_3 در نظر می‌گیریم بنابراین:

$$T(n) = C(2n + 2)$$

حال قطعه کد زیر را دو نقطه بگیرید:

```
(1)        x=0  ;
(2)        for ( i=0 ;  i < n ; i++ )
(3)            for ( j=0 ;  j < n  ; j++ )
(4)                  x++ ;
```

تابع زمانی قطعه کد بالا بصورت زیر محاسبه می‌شود:

سطر	هزینه	تعداد
1	C_1	۱
2	C_2	$n+1$
3	C_3	$n(n+1)$
4	C_4	$n \times n$

بنابراین $T(n)$ برابر است با:

$$T(n) = C_1 + C_2(n+1) + C_3 n(n+1) + C_4 n^2$$

C را بیشترین مقدار C_1، C_2، C_3 و C_4 در نظر می‌گیریم بنابراین خواهیم داشت:

$$T(n) = C(2n^2 + 2n + 2)$$

همانطور که مشاهده می‌کنید $T(n)$ برابر با یک چند جمله‌ای از درجه ۲ می‌باشد. اگر دقت کنید ضرایب چند جمله‌ای در تعداد تکرار، تأثیرگذاری کمتری دارند. ولی هدف ما از محاسبه مرتبه یک الگوریتم بدست آوردن زمان، در تعداد تکرارهای بزرگ یا خیلی بزرگ می‌باشد. بنابراین در حالت کلی ضرایب، تأثیر چندانی در زمان اجرا ندارند. به همین دلیل غالباً از آنها در محاسبات صرف‌نظر می‌کنند.

مثال۱۰۱: تابع زیر مربوط به محاسبه فاکتوریل عدد n را در نظر بگیرید:

```
(1)     int    Factorial( int  n )
        {
(2)              int  fact= 1 ;
(3)              for( int  i=2  ; i<= n ; i ++  )
(4)                    fact*= i ;
(5)              return fact ;
        }
```

تابع زمانی، تابع بالا بصورت زیر محاسبه می‌شود:

سطر	هزینه	تعداد
2	$C_۱$	۱
3	$C_۲$	n
4	$C_۳$	$n-۱$
5	$C_٤$	۱

بنابراین $T(n)$ برابر است با:

$$T(n) = C_۱ + C_۲n + C_۳(n-۱) + C_٤$$

C را بیشترین مقدار $C_۱$، $C_۲$، $C_۳$ و $C_٤$ در نظر می‌گیریم بنابراین خواهیم داشت:

$$T(n) = C(۲n+۱) .$$

مثال۱۰۲: تابع زیر مربوط به حاصل‌جمع دو ماتریس می‌باشد:

```
(1)    void   Add( a, b, c, int m,n )
       {
(2)          for( int i=0 ; i<n ; i ++ )
(3)              for( int j=0 ; j<m ; j ++ )
```

```
c[i,j]= a[i,j] + b[i,j] ;
}
```

تابع زمانی، الگوریتم بالا بصورت زیر محاسبه می‌شود:

سطر	هزینه	تعداد
1	C_1	$n+1$
2	C_2	$n(m+1)$
3	C_3	nm

بنابراین $T(n)$ برابر است با:

$$T(n) = C_1(n+1) + C_2 n(m+1) + C_3 nm$$

C را بیشترین مقدار C_1، C_2، C_3 در نظر می‌گیریم بنابراین خواهیم داشت:

$$T(n) = C(n+1+n(m+1)+nm)$$
$$= C(2nm+2n+1)$$

برای بررسی کارایی الگوریتم‌ها، نمادهایی معرفی شده است که در زیر آنها را بررسی می‌کنیم.

۱۰۲۰۱ نماد Big-oh

برای بررسی میزان رشد توابع زمانی الگوریتم‌ها، نماد Big-oh را بکار می‌گیرند و آنرا با علامت O نمایش می‌دهند. حال در زیر تعریف این علامت را ارائه می‌دهیم:

تعریف: گوئیم $T(n) \in O(f(n))$ اگر و فقط اگر ثابت C و ثابت صحیح n_0 وجود داشته باشند که برای همه مقادیر $n \ge n_0$ ، داشته باشیم $T(n) \le Cf(n)$ (رابطه $T(n) \in O(f(n))$ را بخوانید $T(n)$ متعلق به اوی بزرگ $f(n)$).

تعریف بالا به‌صورت زیر نیز بیان می‌شود :

$$T(n) \in O\big(f(n)\big) \Leftrightarrow \exists C, n_0 > 0 \text{ به‌طوریکه } \forall n \ge n_0 \quad T(n) \le Cf(n)$$

در تعریف بالا $T(n)$ زمان اجرای الگوریتم را مشخص می‌کند و تابعی از انـدازه داده‌ها می‌باشد.

در حالت کلی $f(n)$ مرتبه زمانی اجرای الگـوریتم نامیـده مـی‌شـود (اصـطلاحاً پیچیدگی زمانی الگوریتم هم گفته می‌شود) و با $O(f(n))$ نمایش داده می‌شود.

$T(n)$ مربوط به قطعه کد بالا که شامل فقط یک حلقه است را در نظر بگیرید:

$$T(n) = C(۲n + ۲)$$

C زمان اجرای عملیات، یک مقدار ثابت است بـا فـرض $C = ۱$ خـواهیم داشـت (قبلاً اشاره شد که C به نوع سخت‌افزار، زبان برنامه‌نویسی و غیره بستگی دارد):

$$T(n) = ۲n + ۲$$
$$\leq ۳n \Rightarrow T(n) \in O(n)$$

کـه در آن $n_۰ = ۲$ و $C = ۳$ مـی‌باشـد. بنـابراین بـازای n_0 و C مـشخص $T(n) \in O(n)$ خواهد بود.

مثال۱۰۳: زمان اجرای $T(n)$ مربوط به تعدادی الگوریتم موجود است مرتبـه یـا پیچیدگی زمانی این الگوریتم‌ها را محاسبه نمایید:

i) $T_۱(n) = ۲n^۲ + ٤n$

ii) $T_۲(n) = ۳n^۳ + ۳n$

iii) $T_۳(n) = ٤n + ٥n\,Logn + ۲$

حل:

i) $T_۱(n) = ۲n^۲ + ٤n \leq ۳n^۲$

که در آن اگر $C = ۳$ و $n_۰ = ٤$ باشد آنگاه $T_۱(n) \in O(n^۲)$ می‌باشد.

ii) $T_۲(n) = ۳n^۳ + ۳n$
$\leq ٤n^۳$

که در آن اگر $C = ٤$ و $n_۰ = ۲$ باشد، آنگاه $T_۲(n) \in O(n^۳)$.

iii) $T_۳(n) = ٤n + ٥n\,Log\,n + ۲ \leq ٤n\,Log\,n + ٥n\,Log\,n + ۲n\,Log\,n$

$$= ۱۱n \times Log\,n$$

که در آن اگر $C=11$ و $n_\circ=1$ باشد، آنگاه $T_3(n)\in O(n\,Log\,n)$ خواهد بود.

توجه داشته باشید که می‌توانید ضرایب مختلفی از C را بدست آورید.

نکته: وقتی $T(n)\in O(F(n))$ هست می‌گوییم $F(n)$ یک کران بالا برای $T(n)$ می‌باشد.

مثال۱۰۴: زمان اجرای $T(n)$ مربوط به مثال **۱۰۱** و مثال**۱۰۲ موجود** است مرتبه یا پیچیدگی زمانی این الگوریتم‌ها را محاسبه نمایید:

i) $T(n)=C(2n+1)$

ii) $T(n)=C(2nm+2n+1)$

حل:

i) $T(n)=C(2n+1)$

C زمان اجرای عملیات، یک مقدار ثابت است با فرض $C=1$ خواهیم داشت:

$$T(n)=C(2n+1)=2n+1$$
$$\le 3n$$

که در آن اگر $C=3$ و $n_\circ=1$ باشد آنگاه $T(n)\in O(n)$ می‌باشد.

ii) $T(n)=C(2nm+2n+1)$

C زمان اجرای عملیات، یک مقدار ثابت است با فرض $C=1$ و $m<n$ خواهیم داشت:

$$T(n)\le(2n^2+2n+1)$$
$$\le 3n^2$$

که در آن اگر $C=3$ و $n_\circ=3$ باشد آنگاه $T(n)\in O(n^2)$ خواهد بود.

مثال۱۰۵: درستی یا نادرستی عبارات زیر را ثابت کنید:

i) $T(n)=(2n+1)\in O(n^2)$

ii) $T(n)=(5n^2+n+1)\in O(n)$

iii) $T(n)=(4*2^n+n^2)\in O(2^n)$

حل:

$$\text{i)}\quad T(n) = (2n+1) \le 2n^2$$

که در آن اگر $C=2$ و $n_0=2$ باشد آنگاه $T(n)\in O(n^2)$ خواهد بود. بنابراین رابطه بالا یک رابطه صحیح می‌باشد.

$$\text{ii)}\quad T(n) = (5n^2+n+1) \le 6n^2$$

همانطور که ملاحظه می‌کنید $T(n)$ کمتر از $6n^2$ می‌باشد و به هیچ وجه در حالت کلی نمی‌تواند کمتر از Cn باشد. بنابراین رابطه بالا یک رابطه نادرست می‌باشد.

$$\text{iii)}\quad T(n) = (4*2^n+n^2) \le 5*2^n$$

که در آن اگر $C=5$ و $n_0=4$ باشد آنگاه $T(n)\in O(n^2)$ خواهد بود. بنابراین رابطه بالا یک رابطه صحیح می‌باشد.

همانطور که در مثال‌های بالا ملاحظه کردید در تابع زمانی باید جمله با بیشترین مرتبه را در نظر بگیریم و ضرایب جملات عملاً تاثیری در مرتبه زمانی الگوریتم ندارند. توجه به موضوع مذکور کمک زیادی به حل سریع مسئله می‌کند.

برای روشن شدن موضوع در حالت کلی قضایای زیر را ارائه می‌دهیم.

قضیه ۱۰۱: اگر $T(n)=a_m n^m + a_{m-1}n^{m-1}+...+a_1 n + a_0$ زمان اجرای یک الگوریتم باشد آنگاه $T(n)\in O(n^m)$.

اثبات :

به وضوح می‌توان نوشت:

$$
\begin{aligned}
T(n) &\le |T(n)| \\
&= \left|a_m n^m + a_{m-1} n^{m-1} + \ldots + a_1 n + a_0\right| \\
&\le \left|a_m n^m\right| + \left|a_{m-1} n^{m-1}\right| + \ldots + |a_1 n| + |a_0| \\
&\le n^m \sum_{i=0}^{m} |a_i|
\end{aligned}
$$

بنابراین بازای $C = \sum_{i=0}^{m} |a_i|$ و n_0 مشخص $T(n) \in O(n^m)$ خواهد بود.

بنابراین، در حالت کلی اگر $T(n)$ زمان اجرای یک الگوریتم باشد در اینصورت پیچیدگی زمانی الگوریتم متعلق به جمله‌ای خواهد بود که رشد بیشتری نسبت به بقیه جملات داشته باشد (با C و n_0 که محاسبه می‌شود).

مثال ۱۰۶: زمان اجرای $T(n)$ مربوط به تعدادی الگوریتم موجود است. درستی عبارات زیر را ثابت کنید.

i) $T(n) = (n+1)^2 \in O(n^2)$

ii) $T(n) = 3n^3 + 2n^2 \in O(n^3)$

iii) $T(n) = 5^n \notin O(2^n)$

حل: روابط زیر را طبق قضیه بالا حل می‌کنیم. لذا خواهیم داشت:

i) $T(n) = n^2 + 2n + 1 \le 4n^2$

بنابراین به ازای $n_0 = 1$ و $C = 4$ رابطه برقرار است.

ii) $T(n) = 3n^3 + 2n^2 \le 5n^3$

با توجه به تعریف به ازای $n_0 = 1$ و $C = 5$ رابطه برقرار است.

iii) $T(n) = 5^n \notin O(2^n)$

فرض کنید C و n_{0} موجود است بطوریکه به ازای هر $n \ge n_{0}$ داشته باشیم:

$$5^{n} \le C2^{n}$$

آنگاه عبارت $C \ge \left(\frac{5}{2}\right)^{n}$ حاصل می‌شود. در این رابطه به ازای nهـای بـزرگ C بسیار بزرگی تولید می‌شود بنابراین هیچ ثابت C به ازای هر n برای رابطـه بـالا وجـود ندارد.

۱۰۲۰۲ نماد Big-Omega

تعریف: گوئیم $T(n) \in \Omega(f(n))$ اگر و فقط اگـر ثابـت صـحیح C و n_0 وجـود داشـته باشـد کـه بـه ازای همـه مقـادیر $n \ge n_{0}$ داشـته باشـیم $T(n) \ge Cf(n)$ (رابطـه $T(n) \in \Omega(f(n))$ را بخوانید $T(n)$ امگای بزرگ $f(n)$).

تعریف بالا را بصورت زیر نیز ارائه می‌دهند:

$$T(n) \in \Omega(f(n)) \Leftrightarrow \exists C, n_{0} > 0 \quad \forall n \ge n_{0} \qquad Cf(n) \le T(n)$$

اگر دقت کنید ملاحظه می‌کنید که تعریف بالا یک کران پایین زمـان اجـرا بـرای $T(n)$ ارائه می‌دهد. بنابراین، در حالت کلی می‌توان گفت که $\Omega(f(n))$ بهترین حالـت اجرا برای یک الگوریتم می‌باشد.

برای درک بهتر نماد بالا در زیر چند مثال ارائه می‌دهیم.

مثال۱۰۷: زمان اجرای $T(n)$ الگوریتمی محاسبه شـده، $\Omega(f(n))$ آنـرا بدسـت آورید.

$$T(n) = an^{2} + bn + c \qquad \text{and} \quad a, b, c > 0$$

حل:

$$an^{2} + bn + c > an^{2} + b + c$$
$$> an^{2}$$

بنابراین اگر $n_{0} = 1$ و $C = a$ باشد، آنگاه $T(n) \in \Omega(n^{2})$ خواهد بود.

مثال۱۰۸:زمان اجرای $T(n)$ الگوریتمی محاسبه شده، $\Omega(f(n))$ آنرا بدست آورید.

$$T(n) = n^{٤} + ٥n^{٢}$$

حل:

$$T(n) = n^{٤} + ٥n^{٢} \geq n^{٤}$$

بنابراین اگر $n_{٠} = ١$ و $C = ١$ باشد، آنگاه $T(n) \in \Omega(n^{٤})$ خواهد بود.

مثال ۱۰۹: زمان اجرای $T(n)$ مربوط به تعدادی الگوریتم موجود است. درستی عبارات زیر را ثابت کنید.

i) $T(n) = ٦n + ٤ \in \Omega(n)$

ii) $T(n) = ٣n + ٢ \notin \Omega(n^{٢})$

iii) $T(n) = ٥^{n} + n^{٢} \in \Omega(٢^{n})$

حل:

i) $T(n) = ٦n + ٤ \geq ٦n$

بنابراین اگر $n_{٠} = ١$ و $C = ٦$ باشد، آنگاه $T(n) \in \Omega(n)$ خواهد بود.

ii) $T(n) = ٣n + ٢ \notin \Omega(n^{٢})$

فرض کنید C و $n_{٠}$ موجود است بطوریکه به ازای هر $n \geq n_{٠}$ داشته باشیم:

$$T(n) = ٣n + ٢ \geq Cn^{٢}$$

$$\Rightarrow Cn^{٢} - ٣n - ٢ \leq ٠$$

همانطور که ملاحظه می‌کنید در نامعادله بالا C با مقدار معین وجود ندارد

بنابراین $٣n + ٢ \notin \Omega(n^{٢})$ خواهد بود.

iii) $T(n) = ٥^{n} + n^{٢} \geq ٥^{n} \geq ٢^{n}$

بنابراین اگر $n_{٠} = ١$ و $C = ١$ باشد، آنگاه $T(n) \in \Omega(٢^{n})$ خواهد بود.

حال در حالت کلی قضیه زیر را ارائه می‌دهیم.

قضیه **102** : اگـر $T(n)=a_m n^m + a_{m-1}n^{m-1}+\ldots+a_1 n + a_0$ زمـان اجـرای یک الگوریتم بوده و $a_m > 0$ باشد آنگاه $T(n)\in\Omega(n^m)$.

اثبات :بعنوان تمرین به خواننده واگذار می‌شود.

۱,۲,۳ نماد θ

تا حالا یک کران پایین و یک کران بالا برای تـابع زمـانی یـک الگـوریتم توسـط نمادهای Ω و O ارائه دادیم. حال نماد θ را بصورت زیر تعریف می‌کنیم:

تعریف: گوئیم $T(n)\in\theta(f(n))$ اگر و فقط اگر ثابتهای C_1، C_2 و ثابت صحیح n_0 وجود داشته باشد بطوریکه برای همه مقادیر $n \ge n_0$:

$$C_1 f(n) \le T(n) \le C_2 f(n)$$

تعریف بالا بصورت زیر نیز ارائه می‌شود:

$$\exists C_1, C_2, n_0 > 0 \ \text{بطوریکه}\ \forall n \ge n_0 \quad C_1 f(n) \le T(n) \le C_2 f(n)$$

$$\Leftrightarrow T(n)\in\theta(f(n))$$

توسط نماد بالا تابع $T(n)$ هم از بالا و هم از پایین محدود می‌شود. توجه داشـته باشید که، درجه رشد تابع $f(n)$ و $T(n)$ یکسان است.

مثال ۱۰۱۰: اگر $T(n)=\frac{1}{2}n^2-3n$ باشد $\theta(f(n))$ را محاسبه کنید.

حل:

نشان می‌دهیم که $T(n)\in\theta(n^2)$.

طبق تعریف:

$$T(n)\in\theta(n^2) \quad \Leftrightarrow$$

$$\exists C_1 \text{و} C_2 \text{و} n_0 \ \text{بطوریکه}\ \forall n \ge n_0 \qquad C_1 n^2 \le \frac{1}{2}n^2 - 3n \le C_2 n^2 \quad (1)$$

حال رابطه (۱) را به $n^۲$ تقسیم می‌کنیم:

$$C_۱ \le \frac{۱}{۲} - \frac{۳}{n} \le C_۲$$

با توجه به عبارت بالا، قسمت راست، بـه ازای $C_۲ \ge ۱/۲$ و $n \ge ۱$ برقـرار اسـت.
به‌همین ترتیب برای قسمت چپ عبارت بالا به ازای $n \ge ۷$، $C_۱ \le ۱/٤$ حاصل می‌شود.

بنابراین به ازای $C_۱ = ۱/٤$ و $C_۲ = ۱/۲$ و $n_۰ \ge ۷$ عبارت $T(n) \in \theta(n^۲)$ خواهـد بود.

مثال ۱۰۱۱: فرض کنید $T(n) = ۷n^۳$ باشد . آیا $T(n) \in \theta(n^۲)$ می‌تواند باشد.

حل: طبق تعریف θ باید داشته باشیم:

$\exists C_۱, C_۲, n_۰ > ۰$ بطوریکه $\forall n \ge n_۰ \quad C_۱n^۲ \le ۷n^۳ \le C_۲n^۲$

طرف چپ رابطه بالا همیشه برقرار است اما طرف راست رابطه بـالا در صـورتی برقرار است که $n \le C_۲/۷$ باشد و این با تعریف θ که در آن رابطه برای هر n بزرگتـر از $n_۰$ برقرار است منافات دارد، لذا $T(n)$ نمی‌تواند متعلق به $\theta(n^۲)$ باشد.

مثال ۱۰۱۲: زمان اجـرای $T(n)$ مربـوط بـه تعـدادی الگـوریتم موجـود اسـت. درستی عبارات زیر را ثابت کنید.

i) $T(n) = ٦n + ٤ \in \theta(n)$

ii) $T(n) = ۳n + ٦ \notin \theta(n^۲)$

حل: طبق تعریف θ باید داشته باشیم:

$\exists C_۱, C_۲, n_۰ > ۰$ بطوریکه $\forall n \ge n_۰ \quad C_۱n \le ٦n + ٤ \le C_۲n$

با توجه به رابطـه بـالا، طـرف راسـت بـازای مقـادیر $C_۲ = ۷$ و $n_۰ = ۲$ برقـرار می‌باشد و طرف چپ رابطه بالا، بازای $C_۱ = ٦$ و $n_۰ = ۱$ برقرار خواهد بود.

بنابراین به ازای $C_۱ = ٦$ و $C_۲ = ۷$ و $n_۰ \ge ۲$ عبارت $T(n) \in \theta(n)$ خواهد بود.

ii) $T(n) = ۳n + ٦ \notin \theta(n^۲)$

طبق تعریف θ باید داشته باشیم:

$\exists C_1, C_2, n_0 > 0$ بطوریکه $\forall n \geq n_0 \quad C_1 n^2 \leq 3n + 6 \leq C_2 n^2$

طرف راست رابطه بالا همیشه برقرار است اما طرف چپ رابطه بالا بازای هر n برقرارنیست و این با تعریف θ که در آن رابطه برای هر n بزرگتر از n_0 برقرار است منافات دارد، لذا $T(n)$ نمی‌تواند متعلق به $\theta(n^2)$ باشد.

قضیه **۱۰۳:** اگر $T(n) = a_m n^m + a_{m-1} n^{m-1} + ... + a_1 n + a_0$ زمان اجرای یک الگوریتم بوده و $a_m > 0$ باشد آنگاه $T(n) \in \theta(n^m)$.

اثبات :بعنوان تمرین به خواننده واگذار می‌شود.

۱۰۲۰٤ مرتبه رشد

می‌خواهیم چند برنامه را از روی توابع زمان اجرای آنها با هم مقایسه کنیم.

فرض کنید که توابع زمان اجرای برنامه‌ها به صورت زیر باشند:

$T_1(n) = 2\mathrm{Log}_2^n$ ، $T_2(n) = 100n$

$T_3(n) = 5n^2$ ، $T_4(n) = \frac{1}{2} n^3$

$T_5(n) = 2^n$

حال نمودار توابع بالا را به ازای nهای مختلف تحلیل می‌کنیم:

شکل۱. ۱: زمان اجرای پنج برنامه

ملاحظه می‌کنید که برای تعداد ورودیهای کمتر، زمان اجراها به هم نزدیک‌اند. ولی وقتی تعداد ورودیها افزایش پیدا می‌کند رشد توابع زمان اجرا بسیار متفاوت از هم عمل می‌کنند. الگوریتم‌هایی که زمان اجرای نمایی دارند، رشد بسیار سریعی دارند.

در الگوریتم‌هایی که زمان اجرای آنها لگاریتمی می‌باشد رشد بسیار کمتری نسبت به بقیه توابع زمان اجرا دارند. بنابراین به وضوع می‌توان گفت، الگوریتم‌هایی که پیچیدگی زمانی آنها بصورت لگاریتمی می‌باشد، نسبت به بقیه خیلی بهتر عمل می‌کنند.

در شکل ۱۰۲ زمان اجرای چهار برنامه با پیچیدگی زمانی متفاوت نشان داده می‌شود. ماشین و کامپایلری که برای اجرای این برنامه‌ها بکار برده شده، خاص بوده و فرض می‌کنیم که معیار اجراء براساس ثانیه بوده و در مرحله اول اجراء با تعدادی ورودی، ۱۰۰۰ ثانیه زمان مصرف می‌شود.

زمان اجراء T(n)	بیشترین اندازه مسئله برای ۱۰^{۳} ثانیه	بیشترین اندازه مسئله برای ۱۰^{٤} ثانیه	بیشترین اندازه مسئله
۱۰۰n	۱۰	۱۰۰	۱۰۰۰
٥n^{۲}	۱٤	٤٥	۳۲۰
n^{۳}/۲	۱۲	۲۷	۲۳۰
۲^{n}	۱۰	۱۳	۱۳۰

شکل ۱۰۲: زمان اجرای ٤ برنامه

با مقایسه بیشترین اندازه مسئله برای چهار برنامه مشاهده می‌کنیم که اندازه مسئله برنامه‌ای که زمان اجرای خطی دارد، تقریباً ۱۰ برابر زمان اجرای برنامه‌ای است که بصورت نمایی می‌باشد.

۱۰۳ روش‌های تحلیل الگوریتم‌ها

برای حل مسائل معمولاً بیش از یک الگوریتم وجود دارد. سوالی که در اینگونه موارد مطرح می‌شود اینست که کدامیک از این الگوریتم‌ها بهتر عمل می کنند.

قبلاً اشاره کردیم که الگوریتم‌ها را براساس زمان اجراء و میزان حافظه مصرفی با هم مقایسه می‌کنند. (باز اشاره کردیم که در این کتاب معمولاً الگوریتم‌ها را براساس زمان با هم مقایسه می‌کنیم). بنابراین الگوریتمی کارا می‌باشد که زمان اجراء و حافظه مصرفی کمتری را هدر دهد.

با توجه به مباحث بالا در تحلیل الگوریتم‌ها، نیازمند محاسبه زمان اجراء هستیم به همین منظور روش‌های محاسبه زمان را در این مبحث بیان خواهیم کرد.

معمولاً الگوریتم‌هایی که برای حل مسائل بکار می‌بریم به دو دسته اصلی تقسیم می‌شوند:

۱) الگوریتم‌های ترتیبی

۲) الگوریتم‌های بازگشتی

1۰3۰1 الگوریتم‌های ترتیبی

برای بدست آوردن زمان اجرای یک الگوریتم ترتیبی، زمان اجرای دستورات جایگزینی، عملگرهای محاسباتی، شرطی و غیره را ثابت در نظر می‌گیریم (همانطور که قبلاً اشاره کردیم زمان این دستورات به نوع سخت‌افزار و کامپایلر بستگی دارد).

برای محاسبه زمان اجرای یک تکه برنامه زمانهای زیر را محاسبه می‌کنیم:

۱) اعمال انتساب، عملگرهای محاسباتی، شرط‌های if (ساده)و غیره زمان ثابت دارند.

۲) اگر تعدادی دستور تکرار شوند زمان اجراء، حاصلضرب تعداد تکرار در زمان اجرای دستورات خواهد بود. که معمولاً این قسمت از برنامه‌ها توسط حلقه‌ها نمایش داده می‌شوند.

۳) اگر برنامه شامل ساختار if و else باشد. که هر کدام زمانهای T_1 و T_2 داشته باشند، در اینصورت زمان اجرای این تکه برنامه برابر بیشترین مقدار T_1 و T_2 خواهد بود.

٤) زمان کل برنامه برابر حاصل جمع تکه برنامه‌ها می‌باشد.

بطور شهودی، معمولاً مرتبه زمان اجرای یک الگوریتم، مرتبه تکه‌ای از برنامه است که بیشترین زمان را دارا می‌باشد. برای اینکه ما همیشه برای تابع رشد، کران بالا یا بدترین حالت را در نظر می‌گیریم.

- **الگوریتم ۱۰۱ مرتب‌سازی حبابی**

مساله : پیچیدگی زمانی مرتب‌سازی حبابی را تحلیل کنید.

ورودی: آرایه A از عناصر و n تعداد عناصر آرایه

خروجی: لیست مرتب از داده‌ها

```
void    bubble ( elementtype  A[ ] , int    n )
{
          for ( i=0  ;   i < n – 1  ;  i++ )
             for (  j= n – 1   ;   j > i + 1  ;   j -- )
                  if (  A[ j-1 ] > A[ j ] )
                      exchange ( A[ j ] , A[ j – 1 ] )
}
```

دستور exchange محتویات دو خانه آرایه را با هم جابجا می‌کند و سه عمل جایگزینی در این دستور اجرا می‌شوند که زمان ثابتی برای آنها در نظر می‌گیریم.

در الگوریتم بالا حلقه داخلی در زمان $O(n-i)$ اجرا مـی‌شـود. همچنـین چـون زمان اجرای دستور exchange و شرط if ثابت است لذا زمان کل حلقه داخلی شـرط و دستور exchange برابر $O(n-i)$ می‌باشد. بنابراین زمان کل اجرای الگوریتم به‌صورت زیر می‌باشد:

$$T(n)=\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} d = d\sum_{i=1}^{n-1}(n-i)$$
$$= d\, n(n-1)/2 = d\left(\frac{n^2}{2}-\frac{n}{2}\right)$$

که در آن d ثابت می‌باشد. رابطه بالا را می‌توان بصورت زیر نوشت:

$$T(n)=d\left(\frac{n^2}{2}-\frac{n}{2}\right)\le\frac{d}{2}(n^2+n)$$
$$\le\frac{d}{2}(n^2+n^2)$$
$$\le d\times n^2$$

با در نظر گـرفتن C=d و $n_\circ=2$ (۱ یـا 0 = $n_\circ$ چـون n تعـداد ورودیهـا اسـت بی‌مفهوم می‌باشد) خواهیم داشت:

$$T(n)\in O(n^2) \qquad (1)$$

همچنین می‌توان $\Omega(f(n))$ را بصورت زیر محاسبه کرد:

به وضوح برای $n\ge 2$ داریم: $n-1\ge n/2$ بنابراین:

$$d\frac{n(n-1)}{2}\ge d\times\frac{n}{2}\times\frac{n}{2}=\frac{d}{4}n^2$$

با در نظر گرفتن $C=d/4$ و $n_\circ=2$ خواهیم داشت:

$$T(n)\in\Omega(n^2) \qquad (2)$$

با توجه به رابطه (۱) و (۲) و قضیه ۱۰۲ می‌توان گفت:

$T(n)\in\theta(n^2)$.

در مثال‌های بعدی برای سادگی محاسبه مقدار ثابت زمان اجراء را برابـر یـک در نظر خواهیم گرفت.

- **الگوریتم۱۰۲ جستجوی ترتیبی**

مساله : پیچیدگی زمانی الگوریتم جستجوی ترتیبی را تحلیل نمایید.

ورودی: A آرایه‌ای از عناصر، n تعداد عناصر ، x عنصر مورد جستجو.

خروجی: اندیس عنصر مورد جستجو در صورت وجود.

```
int   Seq_Search ( elementtype a[  ] ,  int    n, elementtype   x )
{
      int    i
      for (  i=0 ;  i < n  ;  i++ )
          if (  a[ i ] == x )
              return ( i )
       return (-1)
}
```

بهترین حالت الگوریتم زمانی اتفاق می‌افتد که عنـصر مـورد جـستجو بـا اولـین عنصر آرایه برابر باشد در اینصورت $T(n) \in O(۱)$ می‌باشد.

اما در حالت متوسط وضع متفاوت است.

در این حالت احتمال اینکه عنصر مورد جستجو در خانـه اول، دوم، ... و یـا nام آرایه باشد یکسان می‌باشد و مقدار آن برای هر یک از خانـه‌هـا برابـر $\frac{۱}{n}$ مـی‌باشـد. از طرف دیگـر اگـر عنـصر مـورد جـستجو در خانـه اول، دوم، ... و یـا nام باشـد تعـداد مقایسه‌ها به‌ترتیب برابر ۱، ۲، ... یا n خواهد بـود بنـابراین زمـان متوسـط اجـراء برابـر خواهد بود با:

$$T(n) = \frac{۱}{n} \times ۱ + \frac{۱}{n} \times ۲ + ... \frac{۱}{n} \times n = \sum_{i=۱}^{n} i/n$$

$$T(n) = \frac{۱}{n} \sum_{i=۱}^{n} i = \frac{۱}{n} \frac{n(n+۱)}{۲} = \frac{n+۱}{۲}$$

بنابراین:

$$T(n)=\frac{n+۱}{۲}\le\frac{۱}{۲}(n+n)=n$$

لذا به ازای $C=۱$ و $n_{\circ}=۱$ خواهیم داشت:

$$T(n)\in O(n)$$

الگوریتم بالا در بدترین حالت، که عنصر مورد جستجو با عنصر nام برابر باشد دارای پیچیدگی زمانی O(n) خواهد بود.

- **الگوریتم 5-1 یافتن بیشترین مقدار**

مساله : پیچیدگی زمانی پیدا کردن بیشترین مقدار در یک آرایه را تحلیل نمائید.

ورودی: آرایه A ، n تعداد عناصر.

خروجی: بیشترین مقدار آرایه

```
elementtype  Maximum( elementtype  A[ ] , int  n )
{
      Max = A[0] ;
      for(  i = 1  ;  i< n  ;  i++  )
           if( A[i] > Max )
                 Max = A[i] ;
       return( Max ) ;
}
```

در الگوریتم بالا حلقه for به تعداد $n-۱$، بار تکرار می‌شود و به همراه آن شرط if نیز بررسی می‌شود. و الگوریتم ربطی به ترکیب داده‌ها ندارد. بنابراین زمان اجرای الگوریتم به صورت زیر خواهد بود:

$$T(n)=\sum_{i=۱}^{n-۱}d=(n-۱)d$$

بنابراین می‌توان نوشت:

$$T(n)=(n-۱)d\le d(n+۱)$$
$$\le d(۲n)=۲d\times n$$

با در نظر گرفتن $C = ۲d$ و $n_{\circ} = ۲$ رابطه زیر برقرار خواهد بود:

$T(n) \in O(n)$

به وضوح می‌توان نشان داد که:

$T(n) \in \theta(n)$.

الگوریتم‌هایی که تا حال بررسی کردیم از نوع الگوریتم‌های ترتیبی بودنـد، حـال می‌خواهیم الگوریتم‌های نوع دوم که به الگوریتم‌های بازگشتی معروفند را بررسی کنیم.

۱۰3۰۲ الگوریتم‌های بازگشتی

معمولاً در الگوریتم‌های بازگشتی، مسئله را بـه دو یـا چنـد زیرمـسئله کـوچکتر تقسیم می‌کنیم. عمل تقسیم مسئله به زیرمسئله‌ها را تا زمانی که اندازه زیرمـسئله‌هـا بـه اندازه کافی کوچک شوند ادامه می‌دهـیم. بعـد از تقـسیم بـه انـدازه کـافی، بـرای حـل زیرمسئله‌ها از خود الگوریتم استفاده می‌کنـیم. سـپس حاصـل زیرمـسئله‌هـا را بـا هـم ترکیب می‌کنیم تا راه حل مسئله بزرگتر حاصل شود. اعمال ترکیب حاصل زیرمسئله‌هـا را تا زمانیکه مسئله اصلی حل نشده باشد ادامه می‌دهیم.

برای محاسبه زمان اجرای الگوریتم‌های بازگشتی به صورت زیر عمل می‌کنیم:

۱) زمان حل زیرمسئله‌ها را محاسبه می‌کنیم (که معمولاً مقدار ثابتی است)

۲) زمان لازم برای شکستن مسئله به زیرمسئله‌ها

۳) زمان لازم برای ادغام جوابهای زیرمسئله ها.

اگر مجموع سه زمان بالا را محاسبه کنیم، زمان اجرای الگوریتم بدسـت خواهـد آمد.

۱۰۳۰۳ محاسبه الگوریتم‌های بازگشتی (recursive algorithm)

همانطور که قبلاً اشاره کردیم، الگوریتمی را بازگشتی می‌نامند که برای محاسبه مقدار تابع نیاز به فراخوانی خود به تعداد لازم باشد. از خصوصیات الگوریتم‌های بازگشتی می‌توان به سادگی پیاده‌سازی و همچنین سادگی درک الگوریتم و غیره اشاره کرد.

در بسیاری از موارد با توجه به خصوصیات الگوریتم‌های بازگشتی ممکن است برای بکارگیری در مسائل نسبت به الگوریتم‌های ترتیبی ترجیح داده شوند ولی همیشه استفاده از آنها مفید نیست. در بعضی از مواقع ممکن است حافظه یا زمان اجرای زیادی را در مرحله اجرا هدر دهند. لذا غالباً بعد از تحلیل الگوریتم‌های بازگشتی در مورد بهتر بودن آنها در مرحله اجرا تصمیم می‌گیرند.

الگوریتم‌های بازگشتی شامل دو مرحله مهم هستند:

- عمل فراخوانی
- بازگشت از یک فراخوانی

با بکارگیری توابع بازگشتی دو مرحله بالا بترتیب انجام می‌گیرد. در مرحله فراخوانی اعمال زیر انجام می‌شود:

۱) کلیه متغیرهای محلی (Local Variable) و مقادیر آنها در پشته (Stack) سیستم قرار می‌گیرند.

۲) آدرس بازگشت به پشته منتقل می‌شود.

۳) عمل انتقال پارامترها (parameter passing) صورت می‌گیرد.

٤) کنترل برنامه (program counter) بعد از انجام مراحل بالا به ابتدای پردازه جدید اشاره می‌کند.

و در مراحل بازگشت عکس عملیات فوق، بصورت زیر انجام می‌شود:

۱) متغیرهای محلی از سرپشته حذف و در خود متغیرها قرار می‌گیرند.

۲) آدرس بازگشت از بالای پشته بدست می‌آید.

۳) آخرین اطلاعات از پشته حذف (pop) می‌شود.

٤) کنترل برنامه از آدرس بازگشت بند۲ ادامه می‌یابد.

نکته: پشته(Stack) ساختار داده‌ای است که آخرین ورودی اولین خروجی است(اطلاعات بیشتر را در فصول بعدی بحث خواهیم کرد) در این ساختار داده دو عملگر معروف بنام‌های pop و push وجود دارد که بترتیب اولی برای حذف از بالای پشته و دومی برای اضافه کردن به بالای پشته بکار می‌روند.

همانطور که اشاره کردیم با بکارگیری الگوریتم‌های بازگشتی اعمال فوق بترتیب انجام می‌شود. و همانطور که ملاحظه می‌کنید در بعضی از مواقع امکان استفاده از الگوریتم‌های بازگشتی بدلیل اینکه حافظه زیادی را هدر می‌دهند، وجود ندارد. (در بعضی از مواقع نیز زمان زیادی را برای اجرا نیاز دارند).

بنابراین در مسائلی که از الگوریتم‌های بازگشتی استفاده می‌کنیم. تحلیل و بررسی دقیقی از میزان حافظه مصرفی و زمان اجرا نیازمندیم.

١٠3٠٤ محاسبه مقادیر الگوریتم بازگشتی

همانطور که در بالا اشاره کریم برای محاسبه مقادیر الگوریتم‌های بازگشتی دو عمل فراخوانی و بازگشت از فراخوانی را نیاز داریم. که در بعضی مواقع ممکن است محاسبه مقدار الگوریتم بازگشتی مشکل به نظر برسد. بنابراین ترجیح دادیم که در این بخش مثال‌هایی را برای روشن شدن مطلب ارائه دهیم.

- **روش بازگشتی محاسبه فاکتوریل**

مساله محاسبه فاکتوریل یک عدد صحیح ساده‌ترین مثال، برای بیان الگوریتم‌های بازگشتی می‌باشد. همانطور که می‌دانیم فاکتوریل یک عدد صحیح n، بصورت بازگشتی زیر قابل تعریف است:

$$n! = \begin{cases} ۱ & \text{if} \quad n = ۰ \\ n \times (n - ۱)! & \text{if} \quad n > ۰ \end{cases}$$

حال دو مرحله اصلی در محاسبه الگوریتم‌های بازگشتی را در مثال بالا بررسی می‌کنیم.

همانطور که قبلاً اشاره کردیم در مرحله فراخوانی، مقادیر متغیرها در پشته قرار می‌گیرند یا اصطلاحاً در پشته push می‌شوند. بنابراین برای n=3 شکل زیر را خواهیم داشت:

شکل 3-10: مراحل محاسبه الگوریتم‌های بازگشتی برای فاکتوریل

در الگوریتم بالا نخست fact(3) فراخوانی می‌شود. بازای n=3 تابع دوباره فراخوانی می‌شود بنابراین مقادیر فراخوانی اول در پشته سیستم ذخیره می‌شود و عمل فراخوانی دوباره ادامه می‌یابد تا اینکه n=0 شود. در اینصورت برای محاسبه عملیات لازم در توابع فراخوانی شده، مقدار یک بازگشت داده می‌شود. بازای هر مرحله بازگشت یک عمل حذف از بالای پشته انجام می‌گیرد و در عین حال عملیات لازم برای بازگشت بعدی صورت می‌پذیرد. تا زمانیکه پشته خالی نشده باشد عمل بازگشت ادامه می‌یابد.

- **روش بازگشتی محاسبه سری فیبوناچی**

سری فیبوناچی یکی از مسائلی است که می‌توان آنرا بصورت غیربازگشتی نیز ارائه داد. ولی ذات آنبصورت بازگشتی است. همچنین ارائه آن بصورت بازگشتی به نظر ساده می‌رسد.

بصورت زیر می‌توان رابطه بازگشتی سری را نمایش داد:

$$fib(n) = \begin{cases} ۰ & \text{if} \quad n = ۱ \\ ۱ & \text{if} \quad n = ۲ \\ fib(n-۱) + fib(n-۲) & \text{if} \quad n > ۲ \end{cases}$$

در حالت کلی جملات سری عبارتند از:

۰ ۱ ۱ ۲ ۳ ۵ ۸ ۱۳ ...

حال، تابع بازگشتی زیر را برای تولید جملات سری فیبوناچی بکار می‌بریم:

```
int      fib (  int     n  )
{
        if  (  n = = 1 )
              return ( 0 ) ;
        else if ( n = = 2 )
              return (1) ;
        else
              return ( fib ( n – 1 )  +  fib ( n – 2 ) ) ;
}
```

مراحل الگوریتم‌های بازگشتی، برای الگوریتم بازگشتی بالا به ازای n=٤ در شکل (٤. ١) نمایش داده شده است.

شکل ٤. ١ : مراحل محاسبه الگوریتم‌های بازگشتی برای سری فیبوناچی

در مرحله اول اجراء، فراخوانی تابع شروع می‌شود که بـا فلـش تـو پـردر شـکل نشان داده شده است، بعد از انجام هر مرحله کامل از فراخوانی مرحله بازگشت شـروع می‌شود که با فلش های منقطع در شکل مشخص شده است .ترتیـب فراخـوانی هـا بـا حروف Aتا H در شکل مشخص می باشد.

در نهایت تابع مقدار ۳ را بعنوان خروجی برمی‌گرداند.

- **روش بازگشتی محاسبه برج هانوی**

یکی دیگر از مسائل کلاسیک که حل آن به روش بازگشتی قدرت ایـن روش را نـشان می‌دهد. مسأله‌ای بنام برج هانوی است. در این مسأله سه محور ثابت (میله) به نام‌هـای A، B و C داریم که در ابتدای کار هشت دیـسک (Disk) بـا انـدازه‌هـای متفـاوت و از بزرگ به کوچک حول محور A رویهم انباشته شده‌اند (به شکل توجه کنید).

در این مسأله هدف انتقال تمام دیسک‌های روی میلـه A بـه میلـه دیگـر مـثلاً C می‌باشد، بطوریکه قواعد زیر رعایت شود:

۱) هر بار بالاترین دیسک باید حرکت داده شود.

۲) دیسک بزرگتر بر روی دیسک کوچکتر قرار نگیرد.

۳) در هر بار حرکت فقط یک دیسک را می‌توان انتقال داد.

این معما را می‌توان تعمیم داد و تعداد دیسک‌ها را به جای هشت تا، nتا در نظـر گرفت. چنانچه n را برابر 2 بگیریم و معما را حل کنیم شناخت بهتری راجع بـه مـسأله پیدا خواهیم کرد. برای حل مسأله در این حالت ابتدا دیسک بالا را از محور A به محور B منتقل می کنیم، در مرحله بعـد دیـسک دیگـر حـول محـور A بـه محـور C منتقـل می‌گردد و در نهایت دیسک محور B را به محور C منتقل می‌کنیم.

شکل ۵. ۱ : وضعیت اولیه مسأله برج هانوی

حال مسأله را به n=۳ تعمیم می‌دهیم.

اگر به طریقی بتوانیم دو دیسک بالا از سه دیسک محور A را به محور B منتقل کنیم آنگاه دیسک آخر را می‌توان به محور C منتقل کرد و سپس دو دیسک موجود حول محور B را به محور C منتقل کرد. مراحل انجام کار بصورت زیر می‌باشد:

الف ـ دو دیسک بالا از سه دیسک محور A به محور B منتقل شود.

ب ـ آخرین دیسک محور A به محور C منتقل شود.

ج ـ دو دیسک حول محور B به محور C منتقل شود.

این روند را می‌توان ادامه داد و مسأله برج هانوی برای n دیسک را حل کرد. در واقع شاهکار روش بازگشتی در این است که حل یک مسأله بزرگتر را منوط به حل مسأله کوچکتر می‌کند و مسأله کوچکتر را به مسائل کوچکتر، تا بالاخره مسأله بسیار کوچک به روش ساده حل شود و از حل آن بترتیب عکس مسائل بزرگتر حل می گردد. در زیر الگوریتم Hanoi نمونه‌ای از هنر طراحی الگوریتم‌های بازگشتی را به نمایش می‌گذارد. مقدار n (تعداد دیسک ها) و محورهای A، B و C بعنوان ورودی الگوریتم زیر می‌باشند:

```
void  Hanoi ( int  n , peg  A ,  peg  B ,  pag  C )
```

```
{
        //دیسک‌های محور A به محور B منتقل شود
   if ( n = = 1 )
        move top Disk on A to C ;
   else{
        Hanoi (n-1, A, C, B) ;
        Move top Disk on A to C ;
        Hanoi (n-1, B, A, C)  ;
 }
```

الگوریتم بالا را برای n=۳ با توجه به مراحل اجرای یک الگوریتم بازگشتی در شکل٦. ١ نمایش می‌دهیم.

شکل ۱.٦: مراحل اجرای الگوریتم بازگشتی برج هانوی

در شکل بالا فلش‌های توپر مرحله فراخوانی تابع و فلش‌های بـا خطـوط منقطـع مرحله بازگشت را نمایش می‌دهند.

1٫3٫5 محاسبه تابع زمانی الگوریتم‌های بازگشتی

در اینجا قصد داریم طریقه محاسبه تابع زمانی الگوریتم‌هـای بازگـشتی را بحـث کنیم. برای روشن شدن مطلب از یک مثال استفاده می‌کنیم.

- **الگوریتم 1٫6 محاسبه فاکتوریل**

مساله: الگوریتم بازگشتی برای محاسـبه فاکتوریـل یـک عـدد نوشـته و زمـان اجرای الگوریتم را تحلیل کنید.

ورودی: عدد صحیح n

خروجی: محاسبه فاکتوریل عدد صحیح n

همانطور که می‌دانیم !n می‌تواند به صورت‌های زیر محاسبه شود:

$$(۱)\qquad n! = \begin{cases} ۱ & \text{if } n = ۰ \\ ۱\times ۲\times ۳\times \ldots \times (n-۱)\times n & \text{if } n > ۰ \end{cases}$$

$$(۲)\qquad n! = \begin{cases} ۱ & \text{if } n = ۰ \\ n\times (n-۱)! & \text{if } n > ۰ \end{cases}$$

شکل (۲) محاسبه فاکتوریل یک عدد در حقیقت شکل بازگشتی مسئله می‌باشد. همانطور که مشاهده می‌کنید برای محاسبه $n!$ نخست باید $(n-۱)!$ محاسبه گردد. همچنین برای محاسبه $(n-۱)!$ باید $(n-۲)!$ محاسبه شود. این تقسیم مسئله به زیرمسئله‌های کوچکتر تا زمانیکه n به صفر نرسیده باشد، ادامه پیدا می‌کند. وقتی n به صفر رسید، با توجه به اینکه $n!$ زمانیکه n به صفر رسیده باشد برابر یک است. زیرمسئله‌ها را حل می‌کنیم. سپس با ادغام جواب زیرمسئله‌ها در مراحل بالاتر، جواب مسئله اصلی حاصل می‌شود.

تابع بازگشتی محاسبه $n!$ به صورت زیر می‌باشد:

```
int      fact (  int   n )
{
        if ( n == 0 )
              return (1) ;
        else
              return ( n  *  fact ( n – 1 ) )  ;
}
```

$T(n)$ را زمان اجرای تابع fact(n) در نظر می‌گیریم. زمان اجرای دستور if برابر $O(۱)$ می‌باشد و زمان اجرای else دستور if برابر $O(۱)+T(n-۱)$ که در آن $O(۱)$ زمان مربوط به عمل ضرب و فراخوانی تابع می‌باشد. بنابراین:

$$T(n) = \begin{cases} O(۱) & \text{if } n = ۰ \\ O(۱)+T(n-۱) & \text{if } n > ۰ \end{cases}$$

$T(n)$ را به صورت زیر نیز می‌توان نوشت:

$$T(n)=\begin{cases} d & \text{if } n=0 \\ T(n-1)+C & \text{if } n>0 \end{cases}$$

بنابراین توانستیم تابع زمانی، الگوریتم بازگشتی fact را محاسـبه کنـیم. T(n) را یک رابطه بازگشتی می‌نامند. حال باید بتوانیم رابطه بازگشتی حاصل را حل کنیم.

در کل در این کتاب یک روش ساده برای حل روابط بازگشتی ارائه می‌دهیم. این روش می‌تواند برای برخی از مسائل جوابگو باشد.(بحث بیشتر در مـورد حـل روابـط بازگشتی در درس طراحی الگوریتم ارائه می‌شود.)

٤. ١ حل روابط بازگشتی

برای محاسبه زمان لازم برای اجرای یک الگوریتم بازگشتی و یا حافظه مورد نیاز آن در زمان اجرا، اغلب با رابطه‌های بازگشتی برخورد مـی‌کنـیم (همـانطور کـه در فـصل اول دیدیم). روابط بازگشتی معمولاً با توجه به انـدازهٔ ورودی بـه یـک معادلـه یـا نامعادلـه تبدیل می‌شوند.

در این اینجا قصد داریم روش‌هایی را برای حل روابط بازگشتی ارائه دهیم.یکـی از این روش‌ها، روش تکرار با جایگذاری می‌باشد. در این روش با توجـه بـه خاصـیت روابط بازگشتی به ازای nهای مختلف و جایگذاری آنها در هم، جواب مـسئله حاصـل می‌شود.

١.٤,١ روش تکرار با جای‌گذاری

این روش با استفاده از جای‌گذاری های متوالی می‌تواند، جواب مناسب را تولیـد کنـد. در این روش با توجه به خاصیت رابطه بازگشتی به ازای nهای مختلف (که در نهایـت به یک مقدار ثابت می‌رسد) و جای‌گذاری آنها در هم جواب مسئله حاصل می‌شود.

مثال١٣. ١: رابطه بازگشتی زیر را در نظر بگیرید:

$$T(n)=\begin{cases} C & \text{if} \quad n=2 \\ T(n-2)+d & \text{if} \quad n>2 \end{cases}$$

رابطه بالا را به روش تکرار با جایگذاری حل کنید.

طرف راست رابطه بالا را بسط می‌دهیم. بنابراین خواهیم داشت:

$$\begin{aligned} T(n) &= T(n-2)+d \\ &= T(n-4)+2d \\ &= \dots \\ &= T(n-2i)+i\times d \end{aligned}$$

رابطه بالا تا زمانیکه به $T(2)$ نرسیدیم ادامه می‌دهیم. بنابراین اگر $n-2i$ به عدد ۲ برسد آنگاه $T(2)$ حاصل می‌شود:

$$n-2i=2 \Rightarrow i={(n-2)}/{2}$$

با جایگذاری در رابطه بالا خواهیم داشت:

$$\begin{aligned} T(n) &= T(2)+\frac{(n-2)}{2}\times d \\ &= C+\frac{(n-2)}{2}\times d \end{aligned}$$

بنابراین $T(n)\in O(n)$.

مثال۱۴. ۱: رابطه بازگشتی زیر را در نظر بگیرید:

$$T(n)=2T\left(\lfloor n/2 \rfloor\right)+d \qquad (1.1)$$

که در آن d یک ثابت زمانی می‌باشد. روش تکرار با جای‌گذاری را بـرای رابطـه بازگشتی (۱. ۱) بصورت زیر بکار می‌بریم:

$$\begin{aligned} T(n) &= 2T\left(\lfloor n/2 \rfloor\right)+d \\ &= 4T\left(\frac{\lfloor n/2 \rfloor}{2}\right)+3d \end{aligned}$$

$$\le 4T\left(n/4\right)+3d$$
$$=\ldots$$
$$\le 2^i T\left(n/2^i\right)+\left(2^i-1\right)d \qquad (1.2)$$

رابطه بالا را آنقدر ادامه می‌دهیم تا به $T(1)$ برسیم بنابراین:

$$\frac{n}{2^i}=1 \Rightarrow i=\log_2^n$$

حال i را در رابطه **(۲. ۱)** جای‌گذاری می‌کنیم:

$$T(n)\le 2^{\log_2^n}\,T(1)+\left(2^{\log_2^n}-1\right)d$$

از آنجایی که $T(1)$ یک مقدار ثابت می‌باشد بنابراین خواهیم داشت:

$$T(n)\le Cn+d(n-1)$$

لذا $T(n)\in O(n)$.

مثال ۱۵. ۱: رابطه بازگشتی زیر را در نظر بگیرید:

$$T(n)\le\begin{cases}C_1 & \text{if} \quad n=1\\ 2T\left(n/2\right)+C_2 n & \text{if} \quad n>1\end{cases} \qquad (1.3)$$

روش تکرار با جای‌گذاری را برای حل رابطه بالا بکار ببرید.

در اولین گام برای حل n را با $n/2$ جایگزین می کنیم. تا $T\left(n/2\right)$ حال شود. بنابراین:

$$T\left(n/2\right)\le 2T\left(n/4\right)+C_2\, n/2 \qquad (1.4)$$

با جای‌گذاری **(۴. ۱)** در طرف راست رابطه **(۳. ۱)**، خواهیم داشت:

$$T(n)\le 4T\left(n/4\right)+2C_2 n$$
$$=\ldots$$
$$\le 2^i T\left(n/2^i\right)+iC_2 n \qquad (1.5)$$

رابطه بالا را آنقدر ادامه می‌دهیم تا به $T(1)$ برسیم، بنابراین:

(با فرض اینکه n توانی از ۲ می‌باشد)

$$\frac{n}{2^i}=1 \Rightarrow \quad i=\text{Log } n$$

حال i را در رابطه (۵. ۱) جای‌گذاری می‌کنیم:

$$T(n) \le 2^{\text{Log } n} T(1) + C_2 n \text{ Log } n$$
$$= C_1 n + C_2 n \text{ Log } n$$

بنابراین $T(n) \in O(n \text{ Log } n)$.

روش تکرار با جایگذاری، روش مناسبی برای حل روابط بازگشتی می‌باشد ولی در بعضی از موارد نمی‌توان از بازکردن فرمول، رابطه بازگشتی به جواب رسید.

۵. ۱ ارائه چند مثال

در این بخش قصد داریم با استفاده از چند مسئله و روش‌های تحلیل الگوریتم را مورد بررسی دقیق‌تر، قرار دهیم.

مثال۱۰۱۶: فرض کنید $T_1(n)$ و $T_2(n)$ زمان اجرای دو قطعه برنامه P_1 و P_2 باشد و داریم:

$$T_1(n) \in O(F(n))$$
$$T_2(n) \in O(g(n))$$

مقدار $T_1(n)+T_2(n)$، زمانی که قطعه برنامه P_2 در راستای قطعه برنامه P_1 اجرا می‌شود را محاسبه نمایید.

حل: می‌دانیم که $T_1(n) \in O(F(n))$ بنابراین C_1 و n_1 وجود دارد که برای:

$$\forall n \ge n_1 \qquad T_1(n) \le C_1 F(n)$$

و همچنین $T_2(n) \in O(g(n))$. بنابراین C_2 و n_2 وجود دارد که برای:

$$\forall n \ge n_1 \qquad T_2(n) \le C_2 g(n)$$

$$\Rightarrow T_1(n)+T_2(n) \le C_1 F(n) + C_2 g(n)$$

$$\le (C_1 + C_2)\max\{F(n), g(n)\}$$

که در آن با انتخاب $n_0 = \max\{n_1, n_2\}$ و $C = C_1 + C_2$ خواهیم داشت :

$$T_1(n) + T_2(n) \in O(\max\{F(n), g(n)\})$$

مثال ۱۷. ۳: الگوریتمی دارای تابع زمانی زیر می باشد:

$$T(n) = \begin{cases} 0 & n = 1 \\ 1 & n = 2 \\ T(n-2) + 3 & n > 2 \end{cases}$$

رابطه بازگشتی بالا را به روش تکرار با جایگذاری حل می‌کنید.

حل :

$$\begin{aligned} T(n) &= T(n-2) + 3 \\ &= T(n-4) + 6 \\ &= \ldots \\ &= T(n-2i) + 3i \end{aligned}$$

i به اندازه‌ای باید رشد کند که عبارت $n-2i$ به مقدار ثابت ۲ برسد. بنابراین خواهیم داشت:

$$n - 2i = 2 \ \Rightarrow \ i = \frac{n-2}{2}$$

با جایگذاری i در رابطه بالا،عبارت زیر حاصل می شود:

$$\begin{aligned} T(n) &= T(2) + \frac{3}{2}n - 3 \\ &= \frac{3}{2}n - 2 \end{aligned}$$

بنابراین $T(n) \in O(n)$.

مثال ۱۸. ۱: خروجی تابع زیر را به ازای F(۳،٦) محاسبه نمائید:

```
int  F( int  m , int  n )
{
```

```
    if (  ( m= = 1)  ||  ( n= = 0 ) ||  (m = = n)  )
         return ( 1 ) ;
    else
         return ( F (  m – 1 , n )  +  F( m – 1 ,  n – 1 ) ) ;
}
```

مراحل اجرای الگوریتم بالا را به ازای مقادیر داده شده، در شکل زیر نمایش می‌دهیم:

شکل ۱.۷ : مراحل اجرای الگوریتم F

شکل بالا مراحل محاسبه مقدار تابع بازگشتی را در دو مرحله فراخوانی و بازگشت نشان می‌دهد. و در نهایت مقدار ٤ را به عنوان خروجی نمایش می‌دهد.

٦. ١ خلاصه فصل

- الگوریتم، مجموعه‌ای از دستورات، دستورالعمل‌ها برای حل یک مسئله می باشد.
- معمولاً الگوریتم‌ها را از نظر کارایی با هم مقایسه می‌کنند.
- منظور از کارآیی الگوریتم‌ها، مقایسه زمان اجرای الگوریتم‌ها با هم می‌باشد. الگوریتمی که زمان اجراء بهتری داشته باشد معمولاً کاراتر از الگوریتم مشابه می‌باشد. (تنها معیار کارایی الگوریتم‌ها زمان اجراء نیست)
- در زمان اجراء یک الگوریتم معمولاً نوع سخت‌افزار، نوع کامپایلر و غیره تأثیرگذار هستند.
- معرفی نمادهای O، θ و Ω به صورت زیر:

$T(n) \in O(f(n)) \Leftrightarrow \exists C, n_0 > 0$ بطوریکه $\forall n \geq n_0 \;\; T(n) \leq Cf(n)$

$T(n) \in \theta(f(n)) \Leftrightarrow \exists C_1, C_2, n_0 > 0$ بطوریکه

$\forall n \geq n_0 \;\; C_1 f(n) \leq T(n) \leq C_2 f(n)$

$T(n) \in \Omega(f(n)) \Leftrightarrow \exists C, n_0 > 0$ بطوریکه $\forall n \geq n_0 \;\; Cf(n) \leq T(n)$

- برای محاسبه زمان اجراء الگوریتم‌ها باید نوع الگوریتم مشخص باشد. در کل با دو نوع الگوریتم که عبارتند از:

١)الگوریتم‌های ترتیبی

٢)الگوریتم‌های بازگشتی

سر و کار داریم.

- الگوریتمی که برای محاسبه مقدار، خود را به اندازه لازم فراخوانی کند الگوریتم بازگشتی نامیده می‌شود.
- در بسیاری از مسائل که ذاتاً بازگشتی هستند، بکارگیری الگوریتم‌های بازگشتی ضروری بنظر می‌رسد.
- الگوریتم‌های بازگشتی برای محاسبه مقدار از دو مرحله: فراخوانی و بازگشت استفاده می‌کند.
- روش تکرار با جای‌گذاری، با استفاده از جای‌گذاری های متوالی می‌تواند، جواب مناسب را تولید کند. در این روش با توجه به خاصیت رابطه بازگشتی به ازای nهای مختلف (که در نهایت به یک مقدار ثابت می‌رسد) و جای‌گذاری آنها در هم جواب مسئله حاصل می‌شود.

۱۰۷ تمرینات

۱- زمان اجرای (یعنی T(n)) را برای هر کدام از الگوریتم‌های زیر محاسبه نمائید؟

الف ـ

```
x=0
for ( i=0 ;  i< n; i++ )
    for ( j= i ; j< n ; j++ )
        x++ ;
```

ب ـ

```
S = 0;
for ( i=0 ;  i< n ;  i++ )
    for ( j=0 ; j< i  ;  j++ )
          S++ ;
```

ج ـ

```
P =0 ;
for ( i=1 ;  i< n ; i++ )
      for ( j= i + 1 ; j<= m  ; j++ )
            P++ ;
```

د ـ

```
m = 0  ;
for ( i=0 ;  j< n  ; i++ )
```

```
for ( j= i + 1  ;  j< n  ;  j++ )
     for ( k = j+ 1 ;  k< n ;  k++ )
            m++ ;
```

هـ ـ

```
L = 0 ;
for ( i=0 ;  j< n ; i++ )
   for ( j=0 ;  j< m ; j++ )
        for ( k= 0 ;  k< p  ;  k++ )
             L++ ;
```

۲- زمان اجرای الگوریتم‌های زیر را محاسبه نمائید:

الف ـ

```
i= n  ;
While (  i >=  1  ) {
  /* Some Statement requiring  θ(۱) time */
      i= i/2 ;
  }
```

ب ـ

```
i=1
While (  i  <=  n ) {
/* Some Statement requiring  θ(۱) time */
       i = i * 2 ;
}
```

ج -

```
i= n  ;
While (  i>= 1 ) {
      j= i  ;
      While ( j<= n ) {
         /* Some Statement requiring  θ(۱) time */
         j= j*2 ;
       }
       i= i/2 ;
  }
```

د-

```
/*Suppose that n>m*/

While (  n> 0  ) {
      r = n%m ;
      n = m ;
      m = r ; }
```

۳- ثابت کنید که عبارت زیر برقرار هستند:

$$۱)\ ۷n^{۲} - ۶n + ۲ \in \theta\left(n^{۲}\right)$$

$$۲)\ n^{۳} + n^{۲}\operatorname{Log} n \in \theta\left(n^{۳}\right)$$

$$۳)\ n! + ۷n^{۵} \in O\left(n^{n}\right)$$

$$۴)\ ۷n^{۲}۲^{n} + ۵n^{۲}\operatorname{Log} n \in \theta\left(n^{۲}۲^{n}\right)$$

$$۵)\ ۲n^{۲^{n}} + ۷۲^{n} \in \theta\left(n^{۲^{n}}\right)$$

$$۶)\ \sum_{i=۰}^{n} i^{۳} \in \theta\left(n^{۴}\right)$$

$$۷)\ n^{۴} + ۷۰۰n^{۲} \in \theta\left(n^{۴}\right)$$

$$۸)\ ۱۲n^{۵}/\operatorname{Log} n + ۷n^{۴} \in O\left(n^{۵}\right)$$

$$۹)\ ۲n^{۳} + ۷n^{۲} \in \Omega\left(n^{۲}\right)$$

$$۱۰)\ ۲n^{۳} + ۷n^{۲} \in \Omega\left(n^{۳}\right)$$

۴- Big-oh و Omega-oh، توابع زمانی زیر را محاسبه کنید.

الف -

$$T(n) = n^{۲} + ۱۰۰۰n$$

ب-

$$T(n) = \begin{cases} n & \text{if} \quad n \text{ is odd} \\ n^{۳} & \text{otherwise} \end{cases}$$

ج-

$$T(n) = \begin{cases} n & \text{if} \quad n \le ۱۰۰۰ \\ n^{۳} & \text{if} \quad n > ۱۰۰۰ \end{cases}$$

۵- فرض کنید $T_1(n)\in\Omega\big(f(n)\big)$ و $T_2(n)\in\Omega\big(g(n)\big)$ باشند. درستی یا نادرستی عبارات زیر را بررسی کنید.

۱) $T_1(n)+T_2(n)\in\theta\big(\max\{f(n),g(n)\}\big)$

۲) $T_1(n)*T_2(n)\in\Omega\big(\max\{f(n),g(n)\}\big)$

۳) $T_1(n)+T_2(n)\in\theta\big(\max\{f(n),g(n)\}\big)$

۴) $T_1(n)*T_2(n)\in\theta\big(f(n)\big)*\big(g(n)\big)$

۶- درستی یا نادرستی عبارات زیر را بررسی کنید.

۱) $n^5+14n^3\in\theta\left(n^3\right)$

۲) $n^5+14n^3\in\theta\left(n^3\right)$

۳) $n^5+14n^3\in\Omega\left(n^3\right)$

۴) $2^{2^n}\in\theta\left(2^n\right)$

۵) $n^6+7n^5+12\in\theta\left(n^7\right)$

۶) $7n^{107}+1000n\in\theta\left(n^2\right)$

۷) $7n^{107}+1000\in\Omega\left(n^{107}\right)$

۸) $2n^{2.81}+100n^2\in\theta\left(n^2\right)$

۹- قضیه ۱۰۱ را ثابت کنید.

۱۰- قضایای ۱۰۲و ۱۰۳ را ثابت کنید.

۱۱- یک آرایه n عنصری مرتب را در نظر گرفته توسط تابعی به نام Search، عنصر x را در آن جستجو نمائید. سپس زمان آن را تحلیل نمائید.

۱۲- در تابع زیر List را یک آرایه n عنصری در نظر بگیرید:

```
int       S( int    List [ ]  ,  int   n  )
{
          if (n = = 1)
              return ( List [1] ) ;
          else
```

```
        return ( List [n]  +  S ( List , n – 1 ) ) ;
}
```

تابع زمانی و پیچیدگی زمانی تابع بالا را محاسبه نمائید .

۱۳– تابع Func را بصورت زیر در نظر بگیرید:

```
int   Func ( int   n  )
{
     if (  n = = 1 )
        return (1) ;
    else
       return ( n + func ( n – 1 ) ) ;
}
```

تابع زمانی و پیچیدگی زمانی تابع بالا را محاسبه نمائید .

۱٤– الگوریتم بازگشتی بنویسید تا کلیه ترکیبات ارقام ۱ تا n را تولید نماید.

توجه کنید کلیه ترکیبات ارقام از ۱ تا n–۱ را داشته باشیم.

۱۵– الگوریتم بازگشتی برای یافتن بیشترین مقدار در یک آرایه از اعـداد صـحیح بنویسید. سپس مراحل محاسبه مقدار تابع بازگشتی را با ارائه مثالی بحث نمائید.

۱٦– الگوریتم بازگشتی طراحی کنید تا یک ماتریس n×n را دریافت کرده سـپس دترمینان ماتریس را به عنوان خروجی برمی‌گرداند.

۱۷–روابط بازگشتی زیر را حل کنید:

الف–

$$T(n)=\begin{cases} ۱ & \text{if} \quad n=٤ \\ T\left(\sqrt{n}\right)+c & \text{if} \quad n>٤ \end{cases}$$

ب–

$$T(n)=\begin{cases} ۱ & \text{if} \quad n=٤ \\ ۲T\left(n/٤\right)+cn & \text{if} \quad n>٤ \end{cases}$$

ج–

$$T(n)=\begin{cases} d & \text{if} \quad n=٤ \\ ۷T\left(n-٤\right)+cn & \text{if} \quad n>٤ \end{cases}$$

د–

$$T(n)=\begin{cases} d & \text{if} \quad n=۲ \\ ۵T(n/۲)+n^{۲} & \text{if} \quad n>۲ \end{cases}$$

۱۸- مسئله برج های هانوی را در نظـر گرفتـه، رابطـه بازگـشتی بـرای حـل آن بنویسید. سپس رابطه حاصل را حل نمائید.

فصل دوم

آرایه ها

اهداف

در پایان این فصل شما باید بتوانید:

- ✓ عملیاتی که روی یک ساختار خطی انجام می شود را تعریف کنید.
- ✓ آرایه را تعریف کرده و بتوانید تعداد عناصر یک آرایه داده شده را پیدا کنید.
- ✓ به تشریح نحوه ذخیره سازی آرایه یک بعدی و دوبعـدی در حافظـه کـامپیوتر بپردازید.
- ✓ جستجوی ترتیبی و جستجوی دودوئی روی آرایه داده شده را انجام دهید.
- ✓ الگوریتم های تطابق الگو روی رشته را توضیح دهید.
- ✓ آیا آرایه جوابگوی تمام نیازهای ما برای تعریف داده های مورد نیاز برنامه می باشد؟

سوالهای پیش از درس

۱- به نظر شما لزوم تعریف یک ساختار داده جدید به نام آرایه چیست؟

....

....

۲- دلیل تعریف آرایه هایی با چند بعد (یک بعدی، دوبعدی و ....) چیست؟

....

....

۳- به نظر شما چرا در زبانهـایی مثـل C و ++C رشـته را بـصورت یـک آرایـه تعریف می کنند؟

....

....

مقدمه

در این فصل یک ساختار خطی کاملاً متداولی بنام آرایه را مورد بررسی قرار می‌دهیم. از آنجایی که عملگرهایی مانند پیمایش، جستجو و مرتب کردن داده‌های آرایه عملگرهای معمول می‌باشد. لذا نیاز به ذخیره مجموعه‌ای از داده‌ها بطور نسبتاً دائمی احساس می‌شود.و غالباً آرایه‌ها چنین عملی را انجام می‌دهند.

ساختمان داده‌ها یا ساختار داده‌ها در حالت کلی به دو دسته خطی و غیرخطی تقسیم می‌شوند. ساختمان داده‌ای را خطی گویند، هرگاه عناصر آن تشکیل یک دنباله دهند. به بیان دیگر یک لیست خطی باشد. برای نمایش ساختمان داده خطی در حافظه دو روش اساسی وجود دارد. یکی از این روش‌ها عبارت است از داشتن رابطه خطی بین عناصری که بوسیله خانه‌های متوالی حافظه نمایش داده می‌شود. این ساختارهای خطی آرایه‌ها نام دارد که موضوع اصلی این فصل را تشکیل می‌دهند.

روش دیگر عبارت است از داشتن رابطه خطی بین عناصری که بوسیله اشاره‌گرها یا پیوندها نمایش داده می‌شود. این ساختارهای خطی لیستهای پیوندی نام دارد که موضوع اصلی مطالب فصل‌های بعدی را تشکیل می‌دهد.

عملیاتی که معمولاً بر روی یک ساختار خطی انجام می‌شود خواه این ساختار آرایه باشد یا یک لیست پیوندی، شامل عملیات زیر است:

عملیاتی که معمولاً بر روی یک ساختار خطی
(الف) پیمایش: رویت کردن همه عناصر داخل لیست را پیمایش گویند. **(ب) جستجوکردن:** پیداکردن مکان یک عنصر با یک مقدار داده شده یا رکورد با یک کلید معین را جستجو کردن گویند. **(ج) اضافه کردن:** افزودن یک عنصر جدید به لیست را اضافه کردن گویند. **(د) حذف کردن:** حذف یک عنصر از لیست را حذف کردن گویند. **(ه) مرتب کردن:** تجدید آرایش عناصر با یک نظم خاص را مرتب کردن گویند. **(و) ادغام کردن:** ترکیب دو لیست در یک لیست را ادغام کردن گویند.

ساختارخطی خاصی که برای یک وضعیت معین انتخاب می‌شود بستگی به تعداد دفعاتی دارد که عملیات بالا روی ساختار اجرا می‌شود.
ساختار بکارگرفته شده در کتاب بدین گونه است که ابتدا کلیه مفاهیم مربوط به موضوع را مستقل از زبان برنامه‌نویسی خاصی بررسی خواهیم کرد و سپس به بررسی چگونگی پیاده‌سازی مفاهیم طرح شده در زبان C خواهیم پرداخت و امکاناتی را که زبان در این مورد در اختیار برنامه‌نویس قرار می‌دهد را بحث خواهیم نمود.

۲,۱ آرایه‌ها

آرایه، لیستی از n عنصر یا مجموعه‌ای متناهی، از عناصر داده‌ای هم نوع می‌باشد (یعنی عناصر داده‌آی از یک نوع هستند) بطوری که:
(الف) به عناصر آرایه به ترتیب و یا مستقیم (تصادفی) و به کمک یک مجموعه از اندیسها می‌توان دسترسی پیدا کرد.
(ب) عناصر آرایه به ترتیب در خانه‌های متوالی حافظه ذخیره می‌شوند.
در تعریف بالا منظور از «متناهی» این است که تعداد عناصر آرایه مشخص است. این تعداد ممکن است کوچک یا بزرگ باشد. و منظور از عناصر همنوع این است که کلیه عناصر آرایه باید از یک نوع باشند به عنوان مثال، عناصر آرایه می‌توانند فقط از نوع صحیح و یا کاراکتری باشند نه اینکه بعضی از عناصر از نوع صحیح و بعضی دیگر از نوع کاراکتری باشند.

۲,۲ آرایه به عنوان داده انتزاعی

منظور از نوع داده انتزاعی یک مدل ریاضی است که متشکل از مجموعه عناصر و عملیاتی بر روی آن مدل تعریف شده‌اند، می‌باشد و نوع داده انتزاعی مستقل از خواص پیاده‌سازی می‌باشد.
آرایه را می‌توان بصورت یک نوع داده انتزاعی بصورت زیر درنظر گرفت:

۱- مجموعه عناصر

لیستی از مجموعه مرتب و متناهی که همه عناصر آن از یک نوع می‌باشند.

۲- عملیات اصلی

دستیابی مستقیم یا تصادفی به هر عنصر آرایه بطوریکه بتوان عمل ذخیره و بازیابی را انجام داد.

۲,۳ آرایه‌های یک بعدی

آرایه یک بعدی برای ذخیره مجموعه‌ای از عناصر همنوع بکار می‌رود عناصر آرایه یک بعدی در محل‌های متوالی حافظه ذخیره می‌شوند در این آرایه، برای دستیابی به عنصری از آریه، از اندیس استفاده می‌شود.

در زبان برنامه نویسی C و C ++ می‌توان آرایه را بصوت زیر تعریف نمود :

```
Type    Name [ Size ] ;
```

در این تعریف اندازه بیانگر تعداد مقادیری است که می‌تواند در آرایه ذخیره شود. برای مثال، دستور:

```
int Array [100] ;
```

آرایه‌ای متشکل از ۱۰۰ عدد صحیح را تعریف می‌کند. دو عمل اصلی که در مورد آرایه‌ها انجام می‌گیرد، اعمال بازیابی و ذخیره می‌باشد.

دو عمل اصلی که در مورد آرایه‌ها انجام می‌گیرد، اعمال بازیابی و ذخیره می‌باشد.

یعنی روی ساختار آرایه می توان عناصری را ذخیره کرد و تنها عملی که می توان روی آن انجام داد آن است که بتوان به عنصر ذخیره شده دستیابی پیدا کرد.

عمل بازیابی در C و ++C را با x=Array[i] نمایش می‌دهند که اندیـسی مثـل i را گرفته و عنصر i ام از آرایه را برمی‌گرداند و عمل ذخیره با دستور ا نتساب Array [i] = x نمایش داده می‌شود.
کوچکترین مقدار اندیش آرایه را حد پایین آرایه می‌نامند و با lower نشان می‌دهند که در C و ++C همواره صفر فرض می‌شود یعنی اندیش آرایه از صفر شروع می‌شـود و بزرگترین مقدار اندیس آرایه را کران بالای نام دارد و با upper نمـایش مـی‌دهنـد. در حالت کلی تعداد عناصر آرایه یک بعدی برابر است با:

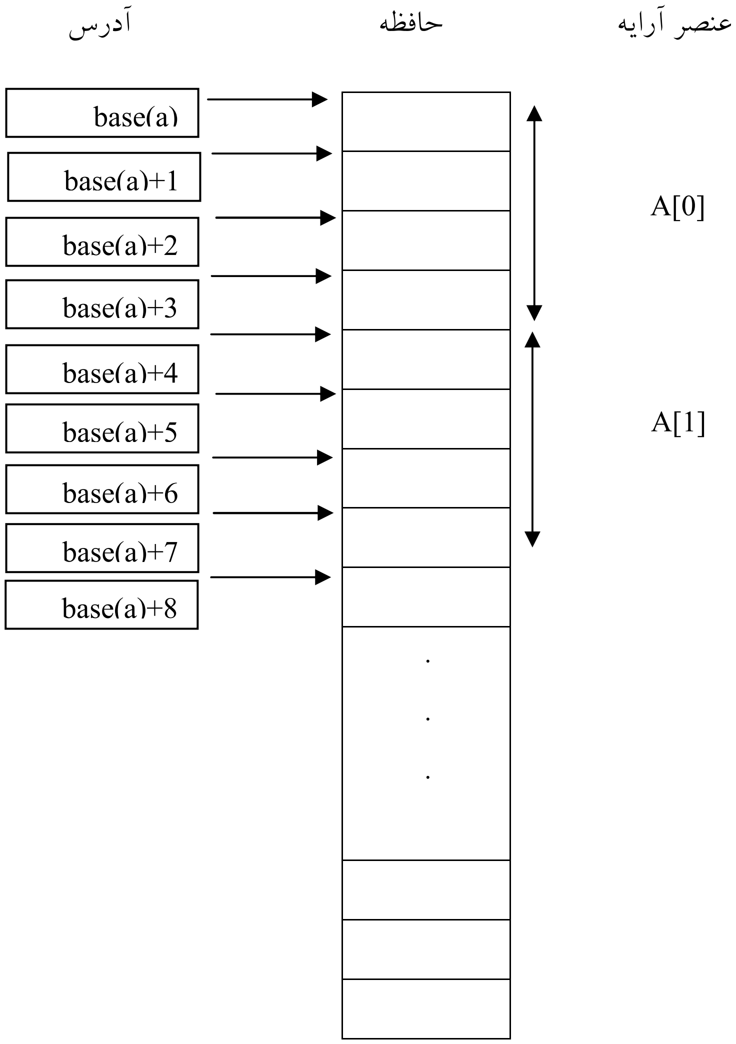
Upper – lower + 1

۲٫٤ نمایش آرایه یک بعدی

در این بخش می‌خواهیم طریقه نمایش آرایه یک بعـدی را در حافظـه نمـایش دهـیم و طریقه قرارگیری عناصر در آرایه را درک کنیم.
دستور زیر را درنظر بگیرید:

float a[10];

این دستور ۱۰ محل متوالی حافظه را تخصیص می‌دهد که در هر محـل مـی‌تـوان یـک مقدار اعشاری را ذخیره کرد. آدرس اولین محـل، آدرس پایـه نـام دارد و بـا Base(a) مشخص می‌شود با فرض اینکه هر مقدار اعشاری چهار بایت از فضای حافظه را اشغال می‌کند، در اینصورت اولین عنصر آرایـه a[0] بـا شـروع از آدرس base(a) در چهـار بایت از حافظه ذخیـره مـی‌شـود و عنـصر a[1] بـا شـروع از آدرس base(a)+4 در چهاربایت بعدی ذخیره می‌شود. شکل ۲٫۱ نحوه نمایش آرایه را نمایش می‌دهد.



شکل ۲,۱ نحوه نمایش آرایه

بطورکلی اگر هر عنصر از آرایه با نام a، به اندازه size بایت فـضا اشـغال کنـد، محـل عنصر i ام بصورت زیر محاسبه می‌شود:

$$Loc(i)=base(a)+i*size$$

مثال ۲,۱ هدف:

پیدا کردن آدرس یک خانه آرایه در حافظه

مساله : فرض کنید آرایه A بصورت float A[10] تعریف شود و base(a) برابر با ۳۰۰۰ باشد (یعنی عناصر این آرایه از خانه ۳۰۰۰ به بعد درحافظه قرار می‌گیرند) و با فرض هر عدد اعشاری چهاربایت فضا از حافظه را اشغال می‌کند آدرس خانه A[7] را محاسبه نمائید.

جواب:

$$\begin{aligned} Loc(7) &= Base\ (A) + 7 * size\ (float) \\ &= 3000 + 28 = 3028 \end{aligned}$$

۲,۵ نمونه‌ای از کاربردهای آرایه یک بعدی برای جستجو

یکی از اعمالی که در آرایه‌های یک بعدی انجام می‌گیرد، جستجوی مقداری در آرایه می‌باشد. جستجو در آرایه می‌توان به یکی از دو صورت ترتیبی یا دودویی انجام شود.

۲,۵,۱جستجوی ترتیبی در آرایه

در الگوریتم جستجو بصورت ترتیبی، ابتدا مقداری را از کاربر دریافت می کند و سپس به جستجو در آرایه برای پیداکردن مقدار موردنظر می‌پردازد. عنصر مورد جستجو نخست با اولین عنصر، دومین عنصر و ... مقایسه می‌شود که در نهایت در صورت موفق بودن عمل جستجو اندیس خانه مورد نظر ارسال می‌گردد در غیر اینصورت مقدار ۱- را بعنوان خروجی بر می‌گرداند

عنوان الگوریتم	الگوریتم جستجوی ترتیبی
ورودی	A آرایه‌ای از عناصر، n تعداد عناصر ، x عنصر مورد جستجو.
خروجی	اندیس عنصر مورد جستجو در صورت وجود.

```
int   Seq_Search ( elementtype a[  ] ,  int    n, elementtype   item )
{
      int    i
      for (  i=0 ;  i < n  ;  i++ )
          if (  a[ i ] == item )
              return ( i )
       return (-1)
}
```

- **محاسبه زمان و پیچیدگی الگوریتم جستوی خطی**

در این تابع، تعداد مقایسه‌ها به مقدار item و عدد ورودی n بستگی دارد زیرا اگر item برابر با اولین مقدار آرایه باشد آنگاه فقط یک بار مقایسه انجام می‌شود و اگر مقدار item برابر با آخرین عنصر آرایه باشد n مقایسه انجام می‌شود و اگر item برابر با هیچ کدام از عناصر آرایه نباشند آنگاه بعد از n مقایسه از حلقه for خارج خواهد شد.

پس در بهترین حالت تعداد مقایسه‌ها فقط یک بار و در بدترین حالت n مقایسه انجام می‌گیرد پس پیچیدگی الگوریتم جستجو برحسب تعداد مقایسه‌های موردنیاز T(n) برای پیداکردن item در آرایه می‌باشد. دو حالت مهم و قابل توجه که مورد بحث و بررسی قرار می‌گیرد حالت میانگین و بدترین حالت است. واضح است که بدترین حالت وقتی اتفاق می‌افتد که عملاً جستجو در تمام آرایه انجام شود و item موردنظر در آرایه پیدا نشود. در اینصورت همانطور که بحث کردیم، تعداد مقایسه‌ها برابر خواهد بود با :

$$T(n) = n+1$$

بنابراین در بدترین حالت، زمان اجرا متناسب با n است.

زمان اجرای میانگین از مفهوم امید ریاضی در احتمالات استفاده می‌کند. فرض کنید P_i احتمال آن باشد که item در a[i] ظاهر شده باشد و q احتمال آن باشد که item در آرایه ظاهر نشده باشد. در این صورت خواهیم داشت:

$$P_1+P_2+\ldots+P_n+q=1$$

هرگاه item در a[i] ظاهر شده باشد چون الگوریتم از i مقایسه استفاده می‌کند میانگین تعداد مقایسه‌ها بصورت زیر محاسبه می‌شود:

$$T(n)=1.P_1+2.P_2+\ldots+n.P_n+(n+1).q$$

(P_n یعنی احتمال اینکه داده در آرایه با اندیس n قرار داشته باشد پس به تعداد n مقایسه لازم داریم تا آن را پیدا کنیم)

همچنین فرض کنید q خیلی کوچک و item با احتمالی مساوی در هر عنصر آرایه ظاهر شده باشد بعبارت دیگر احتمال وجود عنصر مورد جستجو در همه خانه‌های آرایه یکسان باشد. آنگاه $q \approx 0$ و هر $P_i=\frac{1}{n}$ بنابراین:

$$T(n)=1\times\frac{1}{n}+2\times\frac{1}{n}+\ldots+n\times\frac{1}{n}+(n+1)\times 0=(1+2+\ldots+n)\times\frac{1}{n}$$
$$=\frac{n(n+1)}{2}\times\frac{1}{n}=\frac{n+1}{2}$$

یعنی در این حالت خاص، میانگین تعداد مقایسه‌های موردنیاز برای یافتن مکان عنصر item تقریباً برابر نصف تعداد عناصر آرایه است.

۲,۵,۲ جستجوی دودوئی در آرایه

فرض کنید a یک آرایه است که در آن داده‌های عددی بصورت مرتب ذخیره شده‌اند. آنگاه الگوریتم جستجوی بسیار کارآیی بنام جستجوی دودوئی(binary search) وجود دارد که می‌توان از آن برای پیداکردن مکان (loc) عنصر item داده شده، استفاده نمود. ایده کلی این الگوریتم را به کمک یک نمونه واقعی از مثال آشنایی شرح می‌دهیم که در زندگی روزمره با آن سروکار دارید.

جستجوی دودوئی فقط روی آرایه مرتب شده قابل انجام است.

فرض کنید بخواهید مکان یک اسم را در راهنمای دفترچه تلفن پیدا کنید. واضح است که یک جستجوی خطی روی آن انجام نمی‌دهید(یعنی دفترچه تلفن را از ابتدا به انتها نگاه نمی کنید) و به جای آن راهنمای تلفن را از وسط باز می‌کنید و دنبال آن قسمت از دفترچه راهنما می‌گردید که حدس می‌زنید اسم موردنظر شما در آن نیمه قرار دارد. آنگاه نیم اخیر را از وسط نصف کرده و در یک چهارم از راهنما، که حدس زدید اسم موردنظر شما در آن یک چهارم قرار دارد جستجو را ادامه می‌دهید و این کار را همینطور تا آخر ادامه می‌دهید. با توجه به این که خیلی سریع تعداد مکان‌های ممکن در راهنما کاهش می‌یابد در نهایت مکان اسم و اسم موردنظر را پیدا می‌کنیم.

عنوان الگوریتم	الگوریتم جستجوی دودوئی
ورودی	آرایه n عنصری مرتب شده صعودی a و item که باید جستجو شود.
خروجی	اگر عنصر item پیدا شود flag=1 و موقعیت آن را بر می‌گرداند در غیر اینصورت flag=0 و مقدار ۱- را بر می‌گرداند..

```
int   Binary_Search ( elementtype a[  ] ,  int    n, elementtype   item )
{
      int  mid , flag = 0 , first = 0 , last = n-1 ;
      while ( first < = last  &&  ! flag  )
      {
            mid = (first + last)/2 ;
```

```
                if ( item <a[mid] )
                        last = mid -1 ;
               else  if ( item > a [mid] )
                        first = mid + 1 ;
              else
               {
                        flag = 1 ;
                        return mid ;
               }
         }// End While
        retrurn -1 ;
}
```

- **پیچیدگی الگوریتم جستجوی دودوئی**

پیچیدگی الگوریتم جستجوی دودوئی بوسیله تعداد مقایسه‌های موردنیاز برای تعیین مکان item در آرایه مشخص می‌شود.از طرفی می‌دانیم که، آرایه دارای n عنصرمی‌باشد. با توجه به الگوریتم ملاحظه می‌شود هر مقایسه در الگوریتم باعث می‌شود که، اندازه ورودی نصف شود از اینرو حداکثر T(n) مقایسه لازم است تا مکان عنصر item پیدا شود بنابراین تعدادمقایسه‌ها برابر خواهد بود با:

$2^{T(n)-1} > n$ یا $T(n)=[Log_2 n]+1$

یعنی زمان اجرا در بدترین حالت برابر $O(Log_2^n)$ می‌باشد.

زمان اجرای در بدترین حالت برای جستجوی ترتیبی **O(n)** می باشد.
زمان اجرای در بدترین حالت برای جستجوی دودوئی **O(log n)** می باشد.

۲,۶ آرایه‌های دوبعدی

آرایه‌های خطی که در بخش قبل مورد بررسی قرار گرفتند آرایه‌های یک بعـدی بودنـد. و به همین دلیل به هر عنصر این نوع آرایه‌هـا مـی‌تـوان بـه کمـک تنهـا یـک انـدیس دسترسی پیدا کرد. ولی در بسیاری از مسائل آرایه‌های یک بعدی کارائی لازم را ندارنـد مثلاً برای پیاده‌سازی ماتریس‌ها بکارگیری آرایه‌های یک بعدی بسیار سـخت مـی‌باشـد. لذا برای راحتی کار استفاده از آرایه‌های دو بعـدی توصـیه مـی‌شـود. در آرایـه‌هـای دو بعدی دو اندیس برای دسترسی به عناصر آرایه تعریف می‌شـود کـه اصـطلاحاً یکـی از اندیس‌ها در سطر و دیگری در ستون حرکت می‌کند. در زیر طریقه تعریف آرایـه‌هـای دو بعدی در زبان C و C++ را ارائه می‌دهد.:

```
type    name [ row][column]  ;
```

آرایه‌های دو بعدی را در ریاضـیات ، بـرای پیـاده‌سـازی مـاتریس‌هـا و در کاربردهـای تجاری و بازرگانی برای پیاده‌سازی جدول‌هـا بکـار مـی‌برنـد. بنـابراین بـه آرایـه‌هـای دوبعدی گاهی اوقات آرایه‌های ماتریسی نیز می‌گویند.
یک روش استاندارد برای نمایش آرایه دو بعدی $m \times n$ وجود دارد کـه در آن عناصـر A تشکیل یک آرایه مستطیلی با m سطر و n ستون را می‌دهد که در آن مقـدار عنـصر A[j][k] در سطر j ام و ستون kام قرار دارد. شکل(۲,۲) نمایـشی از یـک مـاتریس A دارای ۳ سطر و ٤ ستون را ارائه می‌دهد.

$$\begin{bmatrix} A[o][o] & A[o][1] & A[o][2] & A[o][3] \\ A[1][o] & A[1][1] & A[1][2] & A[1][3] \\ A[2][o] & A[2][1] & A[2][2] & A[2][3] \end{bmatrix}$$

شکل ۲,۲ :آرایه ٤ × ۳ دوبعدی A

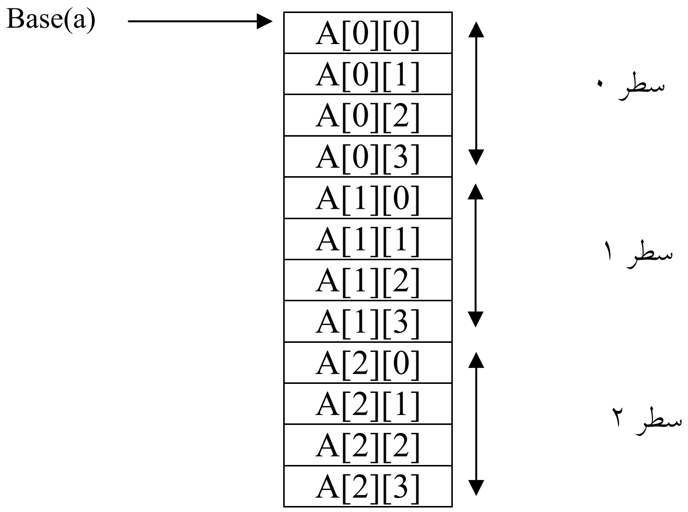
۲,٦,۱ نحوه ذخیره‌سازی آرایه‌های دوبعدی

آرایه‌های دوبعدی می‌توانند بصورت سطری یا ستونی ذخیره شوند در روش سطری ابتدا عناصر سطر اول، سپس عناصر سطر دوم و الی آخر ذخیره می‌شوند. در روش ستونی ابتدا عناصر ستون اول، سپس عناصر ستون دوم و الی آخر ذخیره می‌شوند. در زبان ++C آرایه‌ها بصورت سطری ذخیره می‌شوند.

آرایه زیر را درنظر بگیرید:

```
int A[3] [4]
```

چون عناصر آرایه درC و ++C بصورت سطری ذخیره می شوند نمایش این آرایه در حافظه مانند شکل(۲,۳) خواهد بود.



شکل ۲,۳ : نمایش سطری ماتریس

فرض کنید آرایه A با m سطر و n ستون بصورت زیر تعریف شده باشد:

```
int A[m][n]
```

در اینصورت اگر آدرس پایه این آرایه را base(A) و مقدار فضایی که هر عنصر اشغال می‌کند را size درنظر بگیریم و فرض کنیم ماتریس بصورت سطری در حافظه ذخیره می‌شود آنگاه محل عنصر A[i][j] در حافظه از رابطه زیر بدست می‌آید:

$$A[i][j] \text{ محل } = base(A)+(i*n+j) * size$$

۲,۷ ماتریس‌های اسپارس (Sparse)

بعضی از ماتریس‌ها وجود دارند که تعداد زیادی از عناصر آنها صفر است. به عنوان مثال، ممکن است در مسئله‌ای به ماتریس با ابعاد ۱۰۰۰ × ۱۰۰۰ نیاز داشته باشیم که حاوی یک میلیون عنصر است. از بین عناصر این ماتریس تنها ممکن است فقط ۱۰۰۰ عضو مخالف صفر وجود داشته باشد. به چنین ماتریسی، ماتریس اسپارس (Sparse) می‌گوییم. یعنی ماتریسی که بیشتر عناصر آن صفر باشد. اعمالی که روی ماتریس‌های اسپارس انجام می‌شود معمولاً روی عناصر غیرصفر انجام می‌شود. بنابراین بنظر می‌رسد که لازم نباشد عناصر صفر ماترس در حافظه ذخیره شوند. لذا نمایش معمولی ماتریس‌ها برای نمایش یک ماتریس اسپارس مناسب نیست، بلکه باید نمایش دیگری را درنظر گرفت. در این نمایش فقط شماره سطر، شماره ستون و خود مقدار مربوط به عنصر غیر صفر باید نگهداری شود. ماتریس اسپارس را می‌توان در یک ماتریس که دارای سه ستون است ذخیره کرد، که در آن، ستون اول حاوی شماره سطر مقدار غیر صفر، ستون دوم حاوی شماره ستون مقدار غیر صفر و ستون سوم حاوی خود مقدار غیر صفر می‌باشد. در سطر اول ماتریس، مشخصات کلی ماتریس اسپارس را می‌نویسیم. اگر تعداد عناصر غیرصفر ماتریس اصلی n باشد، آنگاه نمایش ماتریس اسپارس را برای n+1 سطر خواهد بود.

به عنوان مثال یک ماتریس اسپارس و نحوه نمایش آن، در شکل(۲,٤) نمایش داده شده است .

$$A=\begin{bmatrix} 0 & 3 & 0 & 0 & 0 \\ 6 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \end{bmatrix} \rightarrow$$

(الف)ماتریس اسپارس

تعداد مقادیر
غیر صفر

تعداد ستون

تعداد سطرها

3	5	3
0	1	2
1	0	3
2	3	2

4*3

(ب)نمایش ماتریس اسپارس

شکل ٢,٤ : ماتریس اسپارس و شکل بهبود یافته آن

اکنون مقدار فضای اشغالی توسط نمایش معمولی و نمایش اسپارس در ماتریس شکل(٢,٤) را با هم مقایسه می‌کنیم. فرض می‌کنیم عناصر آرایه از نوع float باشند و هر مقدار اعشاری ٤ بایت از حافظه را اشغال کند. در اینصمرت خوهیم داشت:

نمایش معمولی = $3\times5\times4=50$ بایت

نمایش اسپارس = $4\times3\times4=48$ بایت

متوجه می‌شویم که نمایش اسپاس موجب صرفه‌جوئی در حافظه مصرفی می‌شود. اگر تعداد عناصر صفر زیاد باشد و ماتریس نیز بزرگ باشد، این صرفه‌جوئی در حافظه چشمگیر خواهد بود.

٢,٧,١ ترانهاده ماتریس اسپارس

منظور از ترانهاده ماتریس، ماتریس دیگری است که جای سطر و ستون‌های آن عوض شده باشد، الگوریتم زیر روش پیداکردن ترانهاده ماتریس اسپارس که بصورت جدول

ایندکسی با ابعاد $3\times(N+1)$ (N تعداد عناصر غیرصفر) ذخیره شده است را پیدا می‌کند.

عنوان الگوریتم	تعیین ترانهاده ماتریس اسپارس از روی جدول ایندکسی
ورودی	نمایش ماتریس اسپارس
خروجی	ترانهاده ماتریس اسپارس
۱- با توجه به ماتریس نمایش، در سطر اول جای تعداد سطرها و ستون‌ها را عوض می‌کنیم و در ماتریس ترانهاده می‌نویسیم. ۲- در ماتریس نمایش، در ستون وسطی به دنبال اعداد گشته و در صورت پیداکردن آن را در ستون اول ماتریس ترانهاده می‌نویسیم، سپس در ماتریس نمایش، در ستون وسطی به دنبال اعداد ۱ می‌گردیم و آن را در ستون ماتریس ترانهاده می‌نویسیم.	

برای مثال ترانهاده ماتریس نمایش اسپارس شکل(۲,٤) را بصورت زیر ارائه می‌دهیم:

5	3	3
0 1 3	1 0 2	2 3 2

شکل ۲،۵ : نمایش ماتریس ترانهاده ماتریس اسپارس

۲،۸ رشته (String)

از نظر تاریخی از کامپیوتر نخست به منظور پردازش داده‌های عددی استفاده می‌شد. امروزه اغلب کامپیوتر برای پردازش داده‌های غیرعددی موسوم به داده‌های کاراکتری مورد استفاده قرار می‌گیرد.

همچنین امروزه یکی از کاربردهای اصلی کامپیوتر، در عرصه پردازش کلمات یا رشته می‌باشد. معمولاً چنین پردازش‌هایی شامل نوعی از تطبیق الگو(pattern matching) است. برای مثال بحث در مورد این که آیا یک کلمه خاص مانند S در متن داده شده T وجود دارد یا خیر. در اینجا ما قصد داریم مسأله تطبیق الگو را بطور کامل بررسی کنیم. و علاوه بر این، دو الگوریتم مختلف برای تطبیق الگو ارائه خواهیم داد.

تعریف: رشته در واقع آرایه‌ای از کاراکترهاست. ولی به نوعی آنرا در زبانهای مختلف از یک آرایه معمولی از کاراکترها متمایز می‌کنند.

در زبان C++ رشته به NULL یا '\0' ختم می‌شود.

مثال زیر نحوه ذخیره Structure در زبان C++ را نشان می‌دهد.

```
char   S[9] = "Structure"
```

S[0]	S[1]	S[2]	S[3]	S[4]	S[5]	S[6]	S[7]	S[8]	S[9]
s	t	R	u	C	t	U	r	e	'\0'

اصول کار با رشته‌ها تقریباً مشابه با آرایه‌ها می‌باشد و در زبان C++ توابع متعددی مانند کپی‌کردن، الحاق کردن، جستجوی یک رشته داخل رشته دیگر وجود دارد.

۲،۸،۱ الگوریتم های تطابق الگو (Pattern Matching)

منظور از تطبیق الگو همانطور که در بالا اشاره کردیم، مسأله‌ای است که تعیین مـی‌کنـد یک الگوی رشته داده شده P در متن رشته‌آی T وجود دارد یـا خیـر. در الگوریتهـای ارائه شده، فرض خواهیم کرد که همواره طول P کوچکتر از طول T باشد.

- **الگوریتم اول تطبیق الگو**

الگوریتم اول تطبیق الگو، الگوریتم سـاده و غیرکارآمـدی اسـت کـه در آن الگـوی داده شده P با همه زیررشته‌های Tمقایسه می‌شود. عمل مقایسه با حرکت از چپ به راسـت رشته T انجام می‌شود تا به یک تطبیق با P برسد.

فرض کنید که:

$W_K = \text{Subtring}\left(T,K,\text{Length}(P)\right)$

که در آن W_K زیررشته‌ای از T با طول P و با شروع از K امین کـاراکتر T مـی‌باشـد. ابتدا P را کاراکتر به کاراکتر با اولین زیررشته یعنـی W_1 مقایـسه مـی‌کنـیم. اگـر تمـام کاراکترها مساوی باشند در اینصورت $P=W_1$ و در نتیجه P در T ظاهر شده اسـت. از سوی دیگر اگر به این نتیجه برسیم که یک کـاراکتر P دارای متنـاظر در W_1 نیـست در اینصورت $P \neq W_1$ خواهد بود و به سراغ زیررشته بعدی یعنی W_2 می‌رویم و اعمـال بالا را ادامه می‌دهیم و الی آخر، تا عمل مقایسه متوقف شود.

این روش دارای زمان محاسباتی $O(n.m)$ می‌باشد که در آن n طـول زیررشـته P و m طول رشته T می‌باشد.

- **الگوریتم دوم تطبیق الگو**

الگوریتم دوم تطبیق الگو، الگوریتمی است که در این الگـوریتم وقتـی طـول الگـو (P) بزرگتر از تعداد کاراکترهای باقیمانده در رشته T می‌باشـد از آن خـارج مـی‌شـود، و از تست اولین و آخرین کاراکتر P و T قبل از تست بقیه کاراکترها استفاده می‌کند.

الگوریتم بدین صورت کار می‌کند که، در رشته T از ابتدا با استفاده از اندیس start بـه اندازه تعداد کاراکترهای الگو(P) انتخاب می‌شود و کـاراکتر آخـر انتخـاب شـده در T (Endmatch) با کاراکتر آخر الگو مقایسه می‌شود و در صورت مـساوی بـودن سـایر کاراکترها بررسی می‌شوند و در غیراینصورت اندیس Start یک مکان به جلـو حرکـت می‌کند و از آن مکان به اندازه کاراکترهای الگـو انتخـاب مـی‌شـود. مراحـل بـالا را تـا زمانیکه مسئله حل نشده تکرار می‌کنیم.

مثال : فرض کنید P="aab" و T="ababbaabaa" باشد. و lastt انتهای آرایه T و lastp انتهای آرایه P را نشان دهند. نخـست، T[endmatch] و P[lastp] بـا هـم مقایسه می‌شوند اگر برابر بودند الگوریتم از i و j برای حرکـت در داخـل دو رشـته تـا زمانی که یک نابرابری اتفاق بیافتد یا تا زمانی که تمام P بـا آن‌هـا برابـر باشـد اسـتفاده می‌کند.

P:

a a b

J Lastp

a b a b b a a b a a

عدم تطبیق

Start endmatch

T[endmatch] ≠ P[lastp] بنابراین Start یک خانه به جلـو حرکـت مـی‌کنـد. لـذا خواهیم داشت:

a b a b b a a b a a |

عدم تطبیق

Start endmatch

بنابراین T[endmatch]=P[lastp]. در اینصورت کلیه رشته‌ها مقایسه می‌شود و عـدم تطابق رخ می‌دهد.

a b a b b a a b a Δ |
عدم تطبیق
Start endmatch

a b a b b a a b a a |
عدم تطبیق
Start endmatch

a b a b b a a b a a |
عدم تطبیق
Start endmatch

a b Δ b b a a b a a |
عدم تطبیق
Start endmatch

سرعت پردازش در این روش بطور متوسط نسبت به روش ترتیبی بیشتر اسـت بـا ایـن حال در بدترین حالت زمان اجرا هنوز O(n.m) است.

بصورت ایده‌ال الگوریتمی مورد قبول است که در زمان :

O(strlen(string)+strlen(substring)) یعنی (طول الگو + طول رشته) O

کار کند. این زمان بهترین حالت برای این مـسأله مـی‌باشـد. حـال الگـوریتم اینکـار را بصورت کد ارائه می‌دهیم:

عنوان الگوریتم	تعیین وجود زیر رشته داخل رشته داده شده

ورودی	رشته T و زیر رشته P
خروجی	مکان اولین کاراکتری که زیر رشته با رشته تطابق دارد

```
int   nfind (  char  *T ,  char  *P  )
{
      int   i , j , Start = 0 ;
      int  lastp = strlen (T) -1 ;
      int  lastp = strlen (P) -1 ;
      int endmatch = lastp ;
      for (i=0 ; endmatch < = lastt ; endmatcht++ , star++ )
      {
          if (T[endmatch] = = P[lastp])
              for (j=0 , i=start ; j< lastp && T[i] = = P[j] ; i++, j++ )
                     if( j == lastp )
                           return Start ;  /* successful */
       }
       return   – 1 ;
        }
```

۲,۹ مسائل حل شده فصل

در این بخش قصد داریم با ارائه چندین مثال مطالب ارائه شده در این فصل بیشتر برای خواننده قابل درک باشد.

مثال: یک ماتریس n×n را درنظر بگیرید که فقط عناصر قطر اصلی آن مخالف صفر هستند (این ماتریس را ماتریس قطری می‌نامند). این ماتریس یک ماتریس اسپارس است. نمایش آن چگونه است و چقدر در فضای حافظه صرفه‌جوئی می‌شود (برحسب n).

$$A=\begin{pmatrix} a_{oo} & & 0 \\ & \ddots & \\ 0 & & a_{(n-1)(n-1)} \end{pmatrix}$$

الف: ماتریس قطری

$$AS=\begin{pmatrix} n & n & n \\ 0 & 0 & a_{oo} \\ 1 & 1 & a_{11} \\ \vdots & \vdots & \vdots \\ n-1 & n-1 & a_{(n-1)(n-1)} \end{pmatrix}$$

ب: نمایش اسپارس ماتریس A

ماتریس اولیه دارای n×n عنصر است. اگر مقدار حافظه‌ای که هر عنصر اشغال می‌کند برابر با ٤ بایت باشد میزان فضای اشغالی حافظه آن بصورت زیر محاسبه می‌شود:

A فضای حافظه = $4\times n\times n=4n^2$ بایت

اما نمایش اسپارس این ماتریس دارای n+۱ سطر (n تعداد عناصر غیرصفر) و ۳ ستون می‌باشد و در نتیجه فضای آن بصورت زیر محاسبه می‌شود:

As فضای حافظه = $3\times(n+1)\times4=12(n+1)$ بایت

در صورتی در مصرف حافظه صرفه‌جوئی می‌شود که مقدار $12(n+1)$ از $4n^2$ کمتر باشد. بنابراین با حل معادل زیر مقادیر مرزی n بدست می‌آید:

$$4n^2=12(n+1) \Rightarrow 4n^2-12(n+1)=0 \Rightarrow n^2-3n-3=0$$
$$\Rightarrow n=\frac{3+\sqrt{9+12}}{2}=\frac{3+4.5}{2} \approx 3.5$$

نتیجه می‌شود که به ازای $n \geq 4$ در حافظه صرفه‌جوئی می‌شود.

با توجه به مطلب بالا توجه کنید اگر ماتریس قطری ۳×۳ باشد. اسپارس نیست و بهتر است بصورت معمولی ذخیره شود. اما اگر $n \geq 4$ باشد، ماتریس‌های قطری اسپارس هستند و باید بصورت اسپارس نمایش داده شوند.

مثال :فرض کنید یک ماتریس پایین مثلثی مثل A را بخواهیم با یک آرایه یک بعدی مثل B نمایش بدهیم اگر هر عضو A[i][j] معادل عنصر B[L] باشد بین i و j و L چه رابطه‌ای باید برقرار باشد.

حل: فرض کنید بخواهیم آرایه مثلثی A را که در زیر ارائه شده، در حافظه کامپیوتر ذخیره کنیم. واضح است که ذخیره درایه‌های بالا قطر اصلی A کاری بیهوده است چون می‌دانیم تمام این عناصر صفر هستند از این رو تنها درایه‌های دیگر A که در شکل با پیکان مشخص شده است در آرایه خطی ذخیره می‌کنیم یعنی قرار می‌دهیم:

$B[1]=a_{11} \quad B[2]=a_{21} \quad B[3]=a_{22} \quad B[4]=a_{31} \cdots$

نخست ملاحظه می‌کنید که B شامل :

$$1+2+3+4+...+n=\frac{n(n+1)}{2}$$

عنصر است. از آنجاکه در برنامه‌های خود، مقدار a_{ij} احتیاج خواهیم داشت، از این رو فرمولی را بدست می‌آوریم که عدد صحیح L را برحسب I و J معین کند که در آن:

$B(L)=a_{iJ}$

ملاحظه می‌شود که L تعداد عناصر داخل لیست تا a_{ij} و خود آن را نمایش مـی‌دهـد.
اکنون تعداد :

$$1+2+3+...+(i-1)=\frac{i(i-1)}{2}$$

عنصر در سطر جای بالای a_{iJ} وجود دارد و عنصر در سطر i وجود دارد کـه حـداکثر تا a_{ij} و خود a_{ij} را شامل است. بنابراین :

$$L=\frac{i(i-1)}{2}+J$$

$$A=\begin{pmatrix} a_{11} & \cdots & O \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix}$$

اتحاد زیر را که در تجزیه و تحلیل انواع الگوریتم‌هـای مرتـب‌سـازی و جـستجو مـورد استفاده قرار می‌گیرد، ثابت کنید.

$$1+2+3+...+n=\frac{n(n+1)}{2}$$

یک بار عمل جمع را از عدد کوچک، بـزرگ و بـار دیگـر از عـدد بـزرگ بـه کوچـک می‌نویسیم

$$S=1+2+3+...+(n-1)+n$$

$$S=n+(n-1)+(n-4)+...+2+1$$

مجموع دو مقدار S جمع بالا بصورت زیر است:

$$2S=(n+1)+(n+1)+(n+1)...+(n+1)+(n+1)$$

$$\Rightarrow 2S=n(n+1) \Rightarrow S=\frac{n(n+1)}{2}.$$

مثال:فرض کنید A یک آرایه سه قطری n مربعی بصورتی باشد که در شکل نـشان داده شده است.

$$\begin{pmatrix} a_{11} & a_{12} & & & \\ a_{21} & a_{22} & a_{23} & & \\ & a_{32} & a_{33} & a_{34} & \\ & & \ldots\ldots\ldots\ldots\ldots\ldots\ldots & & \\ & & & a_{n,n-1} & a_{n,n} \end{pmatrix}$$

توجه کنید که روی قطر A، n عنصر و در بالا و پایین قطر ۱-n عنصر می‌باشد. از ایـن رو A حداکثر 3n-2 عنصر غیر صفر دارد. فرض کنیـد بخـواهیم A را در یـک آرایـه خطی B ذخیره کنیم. فرمولی پیدا کنید که ما را بر حسب i و J به صورتی بدست دهـد که یعنی

$A(L)=A[i][J]$

$N[1]=a_{11}$ $B[2]=a_{12}$ $B[3]=a_{41}$ $B[4]=a_{44}$

طوری که به تعداد $A[i][J]$ از طریق آرایه B دسترسی پیدا کرد.

حل: توجه کنید که بالای $A[i][J]$ تعداد 3(1-2)+2 عنصر و درست چپ A[i][J] تعداد J-i+1 عنصر وجود دارد از این رو:

$L=[3(i-2)+2]+[J-i+1]+1=2i+J-2$

فرض کنید یک آرایه ۲۰۰ عنصری مرتب شده باشد. زمان اجرای بـدترین $(F_{(n)})$ بـرای پیداکردن عنصر معلوم x در آرایه A با استفاده از جستجوی دودویی چقدر است:

$[\log_2 200]+1=7+1=8$.

تمرین ها

۱) برنامه ای بنویسید که دو عدد صحیح حداکثر ۳۰۰ رقمی را باهم جمع کند. یک روش این است که عدد صحیح ۱۹۸، ۱۷۹،٦۷۲،۵۳٤ را به صورت زیر ذخیره کند:

Block[3]=534، Block[2]=179، Block[1]=672 ، Block[0]=198

سپس دو عدد صحیح را عنصر به عنصر باهم جمع کرده و در صورت وجود رقم نقلی آن را از عنصری به عنصر دیگر منتقل کند.

۲) برنامه ای بنویسید که عدد صحیح n را خوانده و فاکتوریل آن را محاسبه کند. n می تواند هر عدد بزرگی باشد.(راهنمایی: از آرایه استفاده کنید)

۳) تمرین ۱ را برای ضرب دو عدد صحیح ۳۰۰ رقمی انجام دهید.

٤) فرض کنید آرایه های a و b آرایه های پایین مثلثی هستند. نشان دهید که چگونه یک آرایه n در n+1 مثل c می تواند حاوی کلیه عناصر غیر صفر دو آرایه باشد. کدام عناصر از آرایه c به ترتیب a[i][j] و b[i][j] را نشان می دهد؟

۵) آرایه سه قطری a یک آرایه n*n است.که در آن، اگر قدرمطلق i-j بزرگتر از یک باشد، a[i][j]==0 حداکثر تعداد عناصر غیر صفر در چنین آرایه چیست؟ چگونه این عناصر می توانند در حافظه به طور ترتیبی ذخیره شوند؟

٦) برای ذخیره سازی چندجمله ای ها و عملیانهای ویژه آنها ساختمان داده ای مناسب طراحی کنید. ADT مربوطه را بنویسید ؟

۷) فرمول کلی ذخیره سازی ستونی و سطری آرایه های چند بعدی به دست آورید؟

فصل سوم

پشته

اهداف

در پایان این فصل شما باید بتوانید:

- ✓ مفهوم پشته را بیان کرده و دلیل استفاده از آن را بیان کنید.
- ✓ پشته را طراحی و پیاده سازی کنید.
- ✓ کاربردهای پشته را بیان کنید.
- ✓ سه روش بیان عبارات محاسباتی را تشریح کنید.
- ✓ آیا پشته جوابگوی تمام نیازهای ما برای تعریف داده های مورد نیاز برنامه می باشد؟

سوالهای پیش از درس

۱- به نظر شما با توجه به آرایه لزوم تعریف یک ساختار داده جدید ضروری بنظر می رسد؟

....

....

۲- وقتی چاپگر بخواهد از چند سند به طور همزمان چاپ بگیرد به نظر شما چگونه عمل می کند؟

....

....

۳- چگونه می توان از یک آرایه به اینصورت استفاده کرد که فقط به یک طرف آن دستیابی داشته باشیم. مثلا فقط از جلوی آرایه بتوان عنصری را به آن اضافه یا حذف کرد؟

....

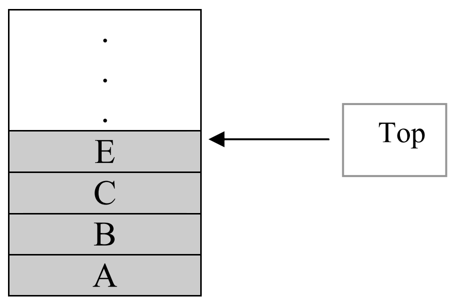
....

مقدمه

آرایه‌ها که در فصل قبل بررسی شدند، اجازه می‌دادنـد کـه عناصـر را در هـر مکـانی از لیست، ابتدای لیست، انتهای لیست یا وسط لیست حذف و یا اضافه کنیم و با توجـه بـه این که اندازه آرایـه در طـول اجـرای برنامـهٔ ثابـت مـی‌مانـد در علـم کـامپیوتر اغلـب وضعیتهایی پیش می‌آید که می‌خواهیم در عمل حذف و یا اضافه‌کردن عناصر به لیـست محدودیتهایی ایجاد کنیم بطوری که این عملیات تنها در ابتدا یا در انتهای لیـست انجـام شود و نه در هر مکانی از آن و همچنین اندازه لیست متناسب با نیازهایمـان گـسترش و یا کاهش یابد. بدین منظور در ساختمان داده پشته در علم کامپیوتر مطرح شد.

۳,۱ تعریف پشته

پشته یک لیست از عناصر است که در آن عمل اضافه‌کردن یاحذف عنـصر تنهـا از یـک طرف آن که عنصر بالا (top) نامیده می‌شود انجام می‌گیرد. شکل ۱–۳ نمونه‌ای از پشته را نشان می‌دهد.



شکل ۱–۳ : یک شکل نمونه از پشته

که پشته دارای چهار عنصر A,B,C,E می باشد و E تنها عنـصری از پـشته اسـت کـه قابل دسترس است. ساده‌ترین راه نمایش یک پشته استفاده از آرایه یک بعدی بـه طـول

n است که n بیانگر حداکثر تعداد عناصر پشته است و در کنار آرایه متغیـری بنـام top وجود دارد که به عنصر بالایی آن اشاره می‌کند.
دو اصطلاح خاص، برای دو عمل اساسی با پشته‌ها بکار می‌رود:

(الف) عمل Push که این اصطلاح برای اضافه‌کردن یک عنصر در پشته بکار می‌رود.
(ب) عمل Pop که این اصطلاح برای حذف یک عنصر از پشته بکار می‌رود.

تأکید می‌کنیم که این اصطلاحات فقط هنگام کار بـا پـشته‌هـا بکـار مـی‌رود و در هـیچ ساختمان داده دیگری مورد استفاده قرار نمی‌گیرد.
با توجه به تعریف پشته یک قلم از عنصر را می‌توان تنها از بالای پشته حـذف و یـا بـه بالای آن اضافه نمود و چون آخرین عنصر داده‌ای اضافه شـده بـه پـشته اولـین عنـصر داده‌ای است که می‌توان از آن حـذف کـرد بـه همـین دلیـل پـشته را لیـستهای آخـرین ورودی، اولین خروجی (LIFO=Last In First out) می‌نامند.
پشته را می‌توان تعدادی از بشقاب یا کتاب فرض نمود که روی هم قرار گرفتـه و بـرای برداشتن، آخرین بشقاب یا کتابی که گذاشته‌اید زودتر از بقیه بر می‌دارید.

۳,۲ نوع داده انتزاعی پشته

با توجه به عملکرد پشته که در بالا توصیف کردیم می‌توان پشته را بـصورت یـک نـوع داده انتزاعی با عناصر و عملیات اصلی زیر تعریف کردند :

۱- مجموعه عناصر
مجموعـه‌ای از عناصـر کـه فقـط از یـک طـرف مرسـوم بـه بـالای پـشته (top) قابـل دستیابی‌اند.

۲- عملیات

- Creater (ایجاد کننده پشته خالی)
- Empty عملیات تست خالی بودن پشته را انجام می‌دهد.
- Push عمل افزودن عنصری به بالا پشته
- Pop عمل حذف از بالای پشته
- Top بازیابی صفر بالای پشته

قابل ذکر است در پشته در حالت سرریزی (Orerflow) و زیرریـزی (underflow) می‌تواند اتفاق بیافتد. و هر دو حالت منجر به خطا در پشته می‌شـوند کـه بایـد از آن‌هـا اجتناب نمود. حالت سرریزی موقعی اتفاق می‌افتد که مـی‌خـواهیم عمـل اضـافه‌کـردن (push) به داخل یک پشته پر انجام دهیم و حالت زیرریزی موقعی اتفاق مـی‌افتـد کـه می‌خواهیم از پشته خالی عنصری را حذف (pop) کنیم.

مثال ۳,۱ هدف:

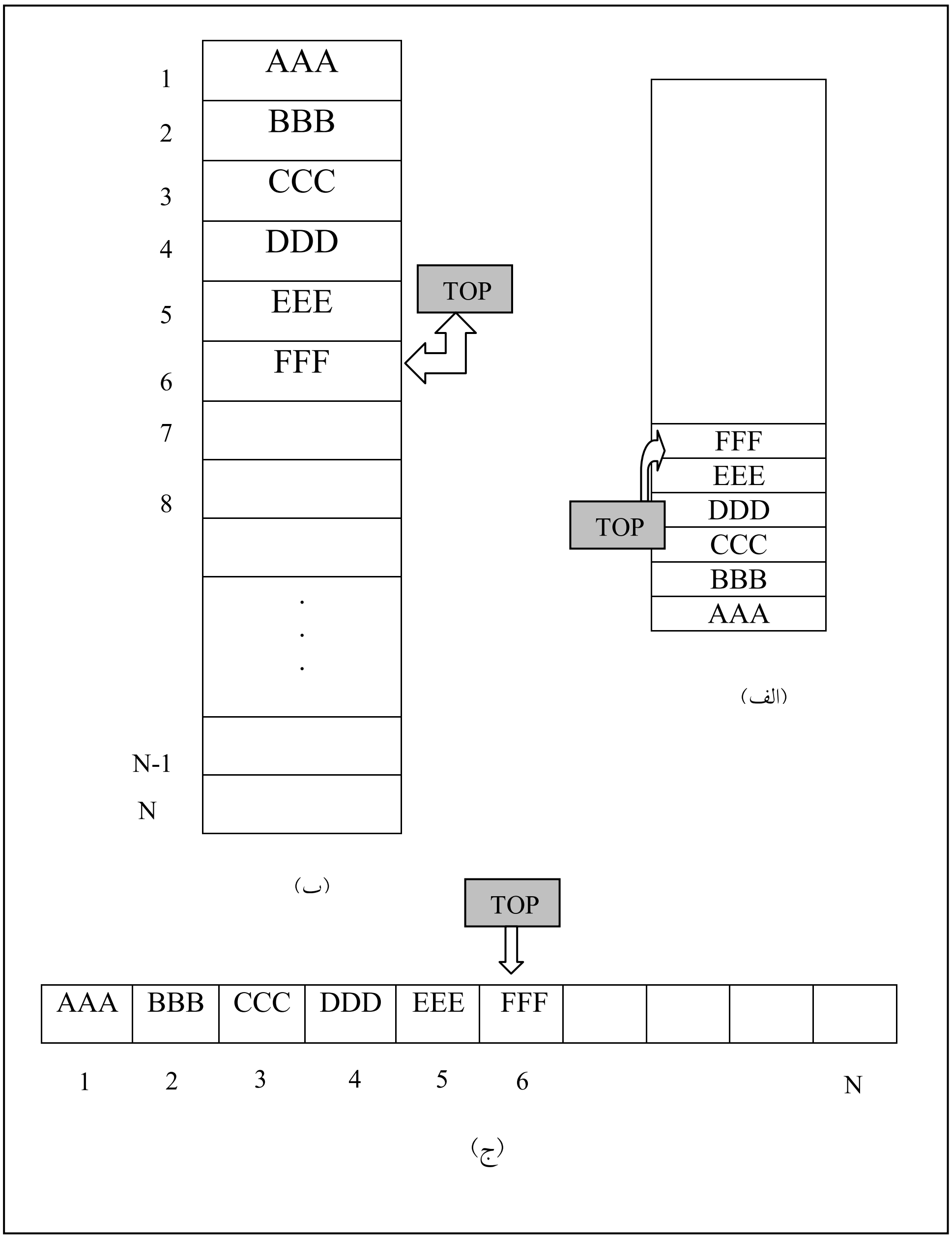
آشنایی با انواع پشته ها

مساله : فرض کنید می‌خواهیم ٦ عنصر زیر را به ترتیب از چپ به راست در یک پـشته خالی push کنیم.

AAA,BBB,CCC,DDD,EEE,FFF

جواب:

شکل ۳,۲ سه روش انجام این کار را نشان می‌دهد.



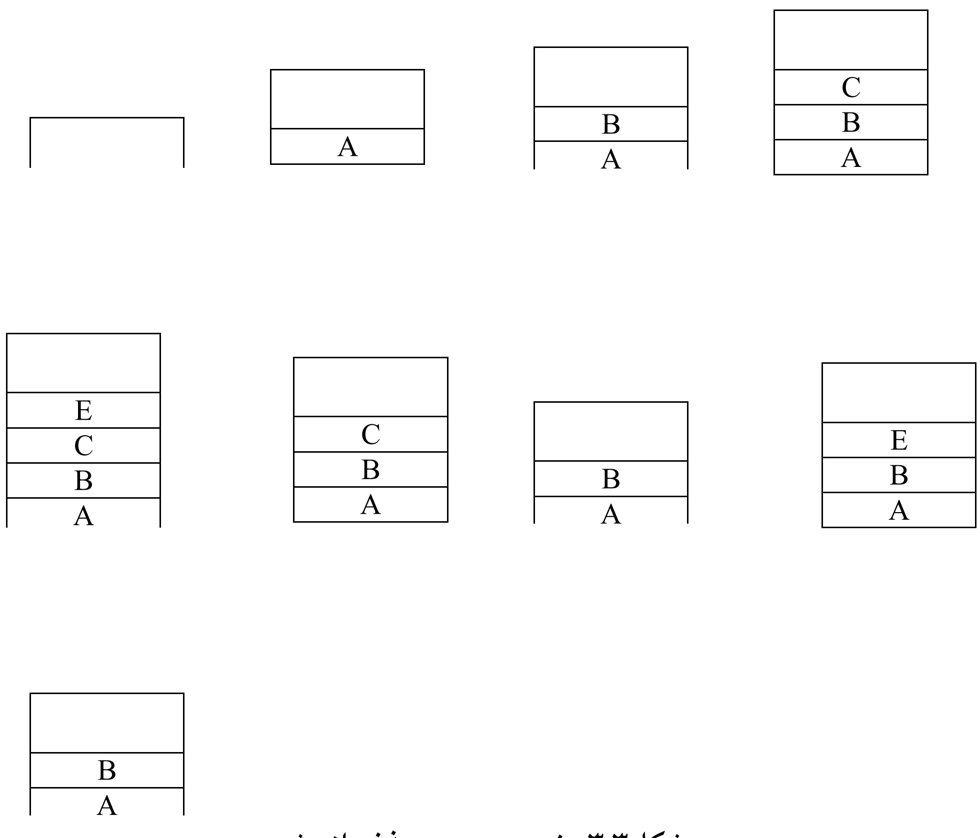
شکل ۳،۲ : روشهای نمایش پشته

در شکل (الف) و (ج) نشانگر نمایش پشته در حافظه کامپیوتر مـی‌باشـد و شـکل (ب) نمایش انتزاعی پشته مستقل از ساختار حافظه است که در این کتاب از این شـکل بـرای نشان دادن پشته استفاده خواهیم کرد.

شکل (۳،۳) برای پشته S و عنصر i عمـل Push(s,i) موجـب قـراردادن عنـصر i در بالای پشته S می‌شود، بطور مشابه عمل pop(s) عنصر بالای پشته را حـذف مـی‌کنـد. لذا دستور x=pop(s) عنصر بالای پشته را حذف و در داخل متغیـر x قـرار مـی‌دهـد.

اعمال زیر را برای بدست آوردن پشته نهایی را بکار می‌گیریم. (از چپ به راست)

Pash(S,A); Pash(S,B); Pash(S,C); Pash(S,E); Pop(S); Pop(S); Push(E); Pop(S);



شکل۳،۳ . نحوه درج و حذف از پشته

۳,۳ نمایش پشته با آرایه

طبق تعریف پشته، پشته را مجموعه ای از مرتب از اقلام داده معرفی کردیم و همچنین در تعریف آرایه، آرایه را نیز مجموعه‌ای مرتب از اقلام داده است. پس هرگاه برای حل مسأله‌ای نیاز به استفاده از پشته باشد می‌توان پشته را بصورت آرایه تعریف نمود اما باید توجه داشت آرایه و پشته کاملاً با هم متفاوتند:

تفاوت آرایه با پشته
تعداد عناصر آرایه ثابت بوده و در حین تعریف مشخص می‌شود درحالیکه پشته یک ساختمان داده پویاست که طول آن را درج یا حذف تغییر می‌کند.

پشته را بوسیله یک آرایه STACK یک متغیر اشاره‌گر TOP که حاوی مکان عنصر بالای پشته و یک متغیر MAXSTACK که بیشترین تعداد عناصر قابل نگهداری توسط پشته است، نمایش می‌دهیم. شرط Top= = -1 مبین ا ین است که پشته خالی است.

عمل اضافه کردن (Push) یک عنصر به درون پشته و عمل حذف کردن (POP) از یک پشته را به تعریف با زیربرنامه‌های Push و POP پیاده سازی می‌کنیم.

پیاده سازی زیر برنامه اضافه کردن یک عنصر به پشته

```
void    Push (int top , element item)
        {      /*  Add an item to the stack   */
              if (*top> = MAXSTACK-1)
                  {
```

```
            Stackful ( ) ;      // return an error key
            return   ;
           }
      Stack [  + + *top]=item;
       }
```

پیاده سازی زیر برنامه حذف کردن یک عنصر به پشته

```
element pop ( int * top)
     {
       //return the top element from the stack
       if (*top = = -1)
         return stack empry ( ) ;    //rerurn an error key
       return stack [(*top) - - ];
        }
```

توجه در زیر برنامه های فوق element می تواند هر نوع داده ای وابسته به عناصر موجود در پشته باشد. اگر عناصر موجود در پشته عدد باشند element ، int و اگر عناصر موجود در پشته کاراکتر باشند element ، char خواهد بود.

در اجرای زیربرنامه Push نخست باید نحقیق کنیم که آیا جا برای عنصر جدید در پشته وجود دارد یا خیر و بطور مشابه در اجرای زیربرنامه pop نخست باید تحقیق کنیم که آیا عنصری در پشته برای حذف وجود دارد یاخیر.

عمل سرریزی (Overflow) در درج و عمل زیرریزی (Underflow) در حذف اتفاق می افتد.

یک تفاوت اساسی بین زیرریزی و سرریزی در ارتباط با پشته‌ها نمایان می‌شود زیرریزی به میزان زیادی به الگوریتم داده شده و داده ورودی بستگی دارد و از این رو برنامه‌نویس هیچ کنترل مستقیمی بر آن ندارد. از طرف دیگر سرریزی بستگی به انتخاب

برنامه‌نویسی برای مقدار حافظه‌ای دارد که برای هر پشته ذخیره می‌کند. همچنین این انتخاب تعداد دفعات وقوع سرریزی را تحت‌الشعاع خود قرار می‌دهد.

در حالت کلی مقدار عناصر یک پشته با اضافه‌شدن یا کم‌شدن عناصر تغییر می‌کند. بنابراین انتخاب مقدار حافظه برای هر پشته داده مستلزم توازن بین زمان و حافظه است. بویژه این که در ابتدا ذخیره مقدار زیاد حافظه برای هر پشته تعداد دفعات وقوع سرریزی را کاهش می‌دهد. با وجود این در اکثر کارها بندرت از حافظه زیاد استفاده می‌شود. مصرف حافظه زیاد برای جلوگیری از مسأله سرریزی پرهزینه خواهد بود و زمان موردنیاز برای حل مسأله سرریزی، مثل اضافه‌کردن حافظه اضافی به پشته می‌تواند پرهزینه‌تر از حافظه اولیه باشد. روش‌های متعددی وجود دارد که نمایش آرایه‌ای پشته را به گونه‌ای اصلاح می‌کند که تعداد فضای ذخیره شده برای بیش از یک پشته را می‌تواند با کارایی بیشتری مورد استفاده قرار دهد. یک نمونه از چنین روشی در مثال زیر بیان شده است.

مثال

مثال ۳,۲ هدف: **زمان اتفاق افتادن سرریزی در دو پشته جداگانه**
مساله : فرض کنید یک الگوریتم داده شده به دو پشته A و B احتیاج دارد. برای پشته A یک آرایه STACKA با n1 عنصر و برای پشته B یک آرایه STACKB با n2 عنصر می‌توان تعریف کرد. **سرریزی وقتی اتفاق می افتد:** پشته A شامل بیش از n1 عنصر باشد یا پشته B بیش از n2 عنصر شود.

برای کاهش دادن تعداد سرریزی در دو پشته می توان از روشی بهتر استفاده کرد. این روش در مثال زیر بیان شده است:

مثال ۳,۳ هدف:

کاهش دادن تعداد سرریزی در دو پشته

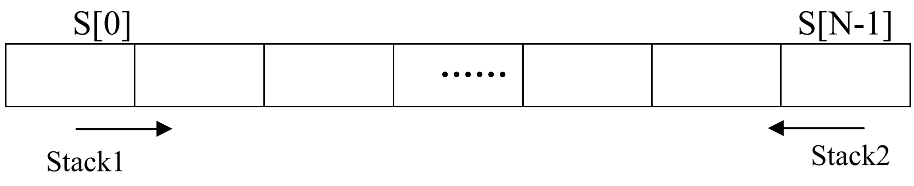
مساله : فرض کنید یک الگوریتم داده شده به دو پشته A و B احتیاج دارد. برای پشته A یک آرایه STACKA با n1 عنصر و برای پشته B یک آرایه STACKB با n2 عنصر می‌توان تعریف کرد.

نحوه تعریف پشته:

بجای کار مثال بالا، یک آرایه STACK با n=n1+n2 عنصر برای پشته‌های B,A تعریف کنیم. مانند آنچه که در شکل ۲ آمده شده است. STACK[0] را به صورت پایین پشته A تعریف می‌کنیم و به A اجازه می‌دهیم بطرف راست رشد کند و STACK[n-1] را بصورت پایین پشته B تعریف می‌کنیم و به B اجازه می‌دهیم طرف چپ رشد کند.

سرریزی تنها وقتی اتفاق می افتد:

که A و B جمعا بیش از n عنصر داشته باشند. این روش معمولاً تعداد دفعات سرریزی را کاهش می‌دهد حتی اگر ما تعداد کل فضای ذخیره شده برای دو پشته را افزایش ندهیم. بدین ترتیب از حافظه موجود بصورت بهینه استفاده می‌گردد. در استفاده از این ساختمان داده عملیات Push و POP لازم است اصلاح شوند.



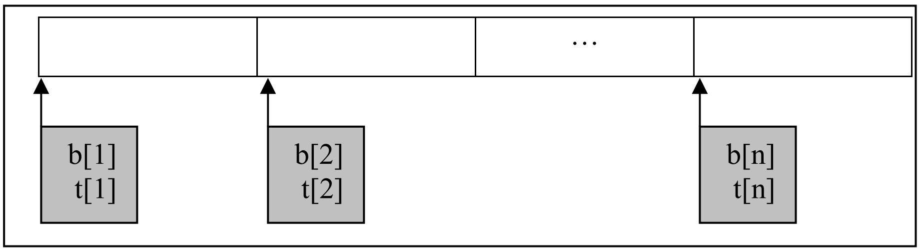
حال اگر در برنامه به بیش از دو پشته نیاز داشته باشیم می توان با الگـو گـرفتن از مثـال بالا پشته های چندگانه را تعریف کرد.

در این حال برای نمایش n پشته حافظه S[1..m] را بـه n قـسمت تقـسیم مـی‌کنـیم. بهتر است تقسیم‌بندی آرایه متناسب با نیازهایمان باشد ولی اگر از قبل نیازهای هر پشته را ندانیم بهتر است حافظه را به قسمتهای مساوی تقسیم کنیم.

فرض می‌کنیم b[i] پایین‌ترین و h[i] به بالاترین عنصر پـشته i اشـاره مـی‌کنـد و اگـر b[i]=h[i] باشد آنگاه پشته i ام خالی است و مقدار اولیه b[i] و h[i] را بصورت زیر تعریف می‌کنیم.

$$L[i] = h[i] = [m-1/n](i) \qquad ۰ \le i \le n$$

تقسیم‌بندی اولیه آرایه M به N پشته مساوی بصورت شکل زیر می‌باشد.



توابع push و pop در پشته‌های چندگانه به صورت زیر تبدیل می‌شوند.

پیاده سازی زیر برنامه Push یک عنصر در پشته‌های چندگانه

```
void push (int i , element item)
{
      if (h[i]= = b[i+1] )
            stackfulls()
      else
            h[i] = h[i]+1;
      stack [h[i]] = item ;
```

```
}
```

پیاده سازی زیر برنامه Pop **یک عنصر** در پشته‌های چندگانه

```
element pop (int  i , element * item)
{
      if (b[i] = = h[i]
           stackempty
      item else = stack [h[i]] ;
      h[i] = h[i] -1 ;
     }
```

حال مثالی از چگونگی بدست آوردن ابتدای هر پشته در ساختار پشته های چندگانه می زنیم.

مثال ۳,٤   هدف:

تعیین آدرس ابتدای هرپشته

مساله : اگر در آرایه [S[1..495 بخواهیم ٤ پشته درست کنیم، آدرس ابتدای هر پـشته را بدست آورید؟

جواب:

B[1]=1
B[2]=[495/4](2-1)+1= 123+1=124
B[3]=[495/4](3-1)+1= 123*2+1=247
B[4]=[495/4](4-1)+1= 123*3+1=370

پس پشته اول از آدرس ۱ شروع می شود و تا آدرس ۱۲۳ طول دارد و پشته ۲ از آدرس ۱۲٤ شروع می شود و تا آدرس ۲٤٦ ادامه دارد و ....

۳,٤ دو کاربرد پشته: فراخوانی تابع و ارزیابی عبارات

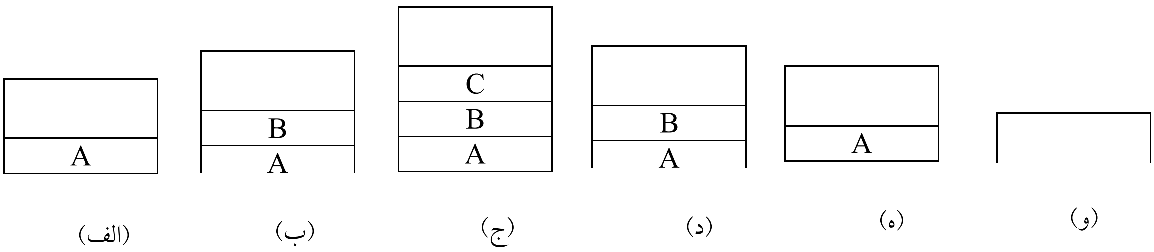
(الف) کاربرد پشته در فراخوانی تابع

پشته ها اغلب برای بیان ترتیب مراحل پردازش‌هـایی بکـار مـی‌رود کـه در آن مراحـل، یعنی پردازش باید تا برقراری و محقق شدن شرایط دیگـر بـه تعویـق بیافتنـد. بـه مثـال زیرتوجه کنید در مثال زیرفـرض مـی‌کنـیم A یـک برنامـه اصـلی و B و C و D زیـر برنامه‌هایی هستند که به ترتیب داده شده فراخوانی می‌شوند

.

فرض کنید که هنگام پردازش برنامه A نیازمند آن باشیم که روی تابعی بنام B کار کنیم که کامل شدن A مستلزم کامل شدن برنامه B می‌باشد. آنگاه پوشه‌ای که شامل داده‌های برنامه A است را در پشته قرار می‌دهیم شکل ۳,٤ (الف)
همچنین شروع به پردازش B می‌کنیم و هنگام پـردازش آن نیازمنـد کـار روی تـابع C باشیم آنگاه همانند شکل ۳,٤ (ب) Bرا در پشته بالای A قـرار مـی‌دهـیم و شـروع بـه پردازش C می‌کنیم. علاوه بر این فرض کنید هنگام پردازش C به همین ترتیـب منتهـی به پردازش D شود. آنگاه C را در پشته بالای B قرار می‌دهیم شکل ۳,٤ (ج) و شـروع به پردازش D می‌کنیم.

از طرف دیگر فرض کنید توانایی کامـل کـردن برنامـه D را داریـم. در اینـصورت تنهـا برنامه‌ای که می‌ـوانیم پردازش آن را ادامه دهیم برنامه C است که در بالای پشته اسـت. از این رو پوشهٔ برنامه C را از پشته حذف می‌کنیم پشته بوصـورتی کـه در نمـودار ۳,٤ (د) نشان داده شده باقی می‌ماند و پردازش C ادامه می‌یابـد. بـه همـین ترتیـب پـس از کامل شدن پردازش C پوشه B را از بالای پشته حذف می‌کنیم و پشته بصورتی کـه در شکل (ه) به ت صویر کشیده شده است باقی می‌مانـد و پـردازش B ادامـه مـی‌یابـد. و بالاخره پس از کامل شدن پردازش B، آخرین پوشـه، A را از پـشته حـذف مـی‌کنـیم، پشته خاص می‌ماند و پردازش برنامه اصلی ما A ادامه می‌یابد.



شکل ۳,٤ نمایش پشته فراخوانی توابع

(ب) کاربرد پشته در ارزیابی عبارات

بررسی عباراتی به شکل Postfix – infix – Perfix

یکی از کاربردهای مهم پشته ارزیابی عبارات می‌باشد. عبارات به سه شکل نوشته می‌شوند، به عنوان مثال عمل میان A و B را درنظر بگیرید این عمل بصورت A+B نوشته می‌شود که به عبارت infix معروف است. این عمل را به دو صورت زیر نیز می‌توان نوشت:

(پیشوندی)	prefix	+AB
(پسوندی)	postfix	AB+

پسوندهای pre ، post و in طریقه قرارگرفتن عملگرها را نسبت به عملوندها را نشان می‌دهند که به ترتیب معنی "قبل" "بعد" و "میان" می‌باشند. در عبارت prefix که به روش لهستانی (Polish) نیز معروف است عملگرها قبل از عملوندها در عبارت infix عملگر بین عملوندها و در عبارت postfix که به روش لهستانی معکوس (یا RPN = Reverse polish Notation) Notation نیز معروف است عملگرها بعد از عملوندها قرار می‌گیرند. عبارات prefix و postfix برخلاف ظاهرشان به سهولت مورد استفاده قرار می‌گیرند به عنوان مثال تابعی که جمع دو آرگویان A و B را برمی‌گرداند بصورت Add(A,B) فراخوانی می‌شود که عملگر Add قبل از دو عملوند A و قرار دارد و نشان دهنده عبارت Prefix می‌باشد.

برای آشنایی از کاربرد روش‌های فوق مثالی را می‌زنیم. ارزیابی عبارتی مثل A+B*C که بصورت infix نوشته شده است مستلزم اطلاع از تقدم عملگرهای + و * می‌باشد. بنابراین A+B*C را می‌توان به دو صورت (A+B)*C و یا A+(B*C) تفسیر نمود ولی با اطلاع از اینکه تقدم عملگر ضرب بیشتر از جمع است عبارت فوق بصورت A+(B*C) تفسیر می‌شود. بنابراین با تبدیل عبارت infix به prefix یا posrfix نیازی به هیچ گونه پرانتزگذاری در عبارات نداریم.

و ترتیبی که در آن عملیات انجام می‌شوند بوسیله مکان عملگرها و عملوندها بطور کامل تعیین می‌شود.
کامپیوتر معمولاً عبارت محاسباتی نوشته شده به صورت نمادگذاری میانوندی را در دو مرحله ارزیابی می‌کند. نخست عبارات مزبور را به صورت نمادگذاری پسوندی تبدیل می‌کند و آنگاه عبارت پسوندی را ارزیابی می‌کند. در هر مرحله پشته، ابزار اصلی این است که برای انجام این کار مشخص مورد استفاده قرار می‌گیرد.

تبدیل عبارات infix (میانوندی)به postfix (پسوندی)
ما برای آشنا شدن با مراحل کار ابتدا روش دستی تبدیل عبارات میانوندی به پسوندی را مطرح می‌کنیم و سپس چگونگی این تبدیل با استفاده از پشته را بررسی خواهیم کرد.
در روش دستی تبدیل عبارت میانوندی به پسوندی بصورت زیر عمل می‌کنیم.

روش دستی تبدیل عبارت میانوندی به پسوندی و پیشوندی
۱- ابتدا عبارت میانوندی را با توجه به اولویت عملگرها پرانتزگذاری می‌کنیم. ۲- هر عملگر را به سمت راست پرانتز بسته خودش انتقال می‌دهیم. ۳- تمام پرانتزها را حذف می‌کنیم. عبارت بدست آمده در فرم پسوندی خود خواهد بود یعنی عملگرها بعد از عملوند خود قرار خواهد گرفت. همچنین برای تبدیل عبارت میانوندی به پیشوندی همان سه مرحله بالا را انجام می‌دهیم فقط در مرحله (۲) هر عملگر را به سمت چپ پرانتز بازخودش منتقل می‌کنیم.

(مثال۳,۵) عبارت a*b+c-a/d را بصورت عبارت پسوندی و پیشوندی بنویسید.

با توجه به اینکه اولویت عملگر ضرب و تقسیم و جمع و تفریق بصورت زیر است

* /

\+ -

با توجه به این جدول اولویت ضرب و تقسیم نسبت به جمع و تفریق بیشتر است و اگر دو عملگر اولویت بودند (مانند ضرب و تقسیم) در عبارات از چپ بـه راسـت ارزیـابی می‌شوند. با توجه به این اولویت‌ها عبارت را بطور کامل پرانتزگذاری می‌کنیم.

(((a*b)+c)-(a/d))

در عبارت پسوندی عملگر را بعد از پرانتز بسته خودش قرار می‌دهیم و سپس پرانزها را حذف می‌کنیم.

ab*c+ad/ -

در عبارت پیشوندی عملگر را قبل از پرانتز باز خودش قرار می‌دهیم و سپس پرانتزها را حذف می‌کنیم.

\- + * abc / ad

تبدیل عبارتهای میانوندی به عبارتهای پسوندی با استفاده پشته

کامپایلرها برای محاسبه عبارت پسوندی از پشته از الگوریتم زیر استفاده می‌کند.

عنوان الگوریتم	**الگوریتم** تبدیل عبارت میانوندی به پسوندی
ورودی	عبارت میانوندی
خروجی	عبارت پسوندی
۱- پشته ای خالی برای عملگرها ایجاد کنید. ۲- عبارت میانوندی را از سمت چپ به راست بخوان وتا زمانی که به انتهـای عبـارت بعـدی اعمال زیر را انجام بده الف) نشانه بعدی (ثابت، متغیر، عملگر، پرانتز باز، پرانتز بسته) را از عبارت میانوندی دریافـت کن ب) اگر نشانه :	

o پرانتز باز است آن را در پشته قرار بده.

o عملوند است آن را در خروجی بنویس.

o عملگر است. اگر تقدم این عملگر از تقدم عملگر بالای پشته بیشتر باشد. آن را در پشته قرار دهید.

و گرنه عضو بالای پشته را حذف کنید در خروجی بنویسید.

سپس این عملگر ورودی را با عملگر جدید موجود در بالای پشته مقایسه کنید و این عمل را آن قدر ادامه دهید تا پشته خالی شود یا تقدم عملگر موجود در پشته کمتر از آن عملگر شود در این صورت آن عملگر را در پشته قرار دهید.

o پرانتز بسته است آنگاه عملگرهای بالای پشته را POP کرده و در خروجی می‌نویسیم تا هنگامی که به یک پرانتز باز برسیم آن را POP کرده ولی آن در خروجی نمی‌نویسیم.

۳- وقتی به انتهای عبارت میانوندی رسیدی، عناصر موجود در پشته را حذف کن و در خروجی بنویسیید تا پشته خالی شود.

(مثال۳,٦) شکل زیر مراحل انجام کار تبدیل عبارت a+b*(c/(d+e))*f به عبارت پسوندی را نشان می‌دهد.

ورودی‌میانوندی	[0]	[1]	[2]	[3]	[4]	[5]	خروجی
a	+						a
+							a
b							ab
*	+	*					ab
(	+	*	(				ab
c	+	*	(				abc
/	+	*	(	/			abc
(	+	*	(	/			abc
d	+	*	(	/			abcd
+	+	*	(	/			abcd
e	+	*	(	/	+		abcde
)	+	*	(	/	+		abcde+

)	+	*		abcde+/
*	+	*	(ابتدا* را pop کردیم و سپس push کردیم	abcde+/*
f	+	*		abcde+/*f*+

خانه‌های پشته

ارزیابی یک عبارت پسوندی

فرض کنید P یک عبارت محاسباتی نوشته شده که بصورت پسوندی باشد می‌توان با کمک پشته بصورت زیر عبارت موردنظر را ارزیابی کرد. عبارت پسوندی از چپ به راست خوانده می‌شود و عملوندی که خوانده می‌شود در پشته قرار داده می‌شود پس از رسیدن به یک عملگر به دو عنصر بالایی پشته را حذف نموده و این عملگر را روی آنها اثر داده و نتیجه را در پشته قرار می‌دهیم و تا زمانی که به انتهای عبارت ورودی برسیم جواب نهایی در بالای پشته قرار دارد.

مثال ۳,۷ هدف:

پیدا کردن جواب عبارتی که به صورت پسوندی نوشته شده است

مساله : عبارت محاسباتی M زیرا که بصورت پسوندی بصورت زیر نوشته شده است درنظر بگیرید:

M : 5 6 2 + * 12 4 / -

حاصل آن را با استفاده از پشته محاسبه می‌کنیم.

جواب:

بدین صورت عمل می کنیم که از چپ به راست اعداد را خوانده و در پشته قرار می دهیم و اگر به یک عملگر رسیده باشیم دو عدد بالای پشته را برداشته و عملگر مورد نظر را برروی دو عدد اعمال می کنیم و جواب را به پشته بر می گردانیم و کار را از ورودی ادامه می دهیم.

مرحله	ورودی	پشته
۱	٥	٥
۲	٦	٥,٦
۳	۲	٥,٦,۲
٤	+	٥,۸
٥	*	٤٠
٦	۱۲	٤٠,۱۲
۷	٤	٤٠,۱۲,٤
۸	/	٤٠,۳
۹	-	۳۷

در مرحله ٤ چون ورودی عملگر+ است دو عنصر بالای پشته یعنی ۲ و ٦ را از پشته برداشته و باهم جمع می کنیم و نتیجه را به پشته بر می گردانیم.

توجه نمایش پیشوندی(Perfix) را روش لهستانی یا Polish نیز می گویند. و همچنین به نمایش پسوندی (Postfix) روش لهستانی معکوس یا (RPN) می گویند.

۳,٥ زیربرنامه‌های بازگشتی

بازگشتی یک مفهوم بسیار مهم در علم کامپیوتر است. بسیاری از الگوریتم‌ها را می‌توان با استفاده از مفهوم بازگشتی بصورت کاراتری بیان نمود. اکثراً تصور می‌کنند که تابع چیزی است که توسط یک تابع دیگر فراخوانده می‌شوند. تابع کدهای خود را اجرا می‌کندو سپس کنترل را به تابع فراخوانده باز می‌گرداند. ولی توابع می‌توانند خودشان را نیز صدا بزنند (بازگشتی مستقیم) یا می‌توانند توابعی که تابع فراخواننده را صدا می‌زنند،

احضار نمایند (بازگشتی غیرمستقیم) چنین روش‌های بازگشتی نـه تنهـا قدرتمنـد بـوده بلکه می‌توان توسط آن‌ها فرایندهای پیچیده‌ای را بصورت ساده بیان کرد.

اغلب دانشجویان علم کامپیوتر معتقدند الگوریتم‌های بازگشتی، پیچیده و گـیج کننـده‌ای هستند و فقط برای بعضی مسائل کاربرد دارد. نظیر فاکتوریل یا تابع Ackermann امـا حقیقت این است که هر برنامه‌ای که بتوانیم با اسـتفاده از دسـتور انتـساب while , if else بنویسیم می‌توان آن را بصورت بازگشتی نیز نوشـت. اغلـب درک تـابع بازگـشتی راحت‌تر از نوع تکراری آن است.

هرزیر برنامه بازگشتی باید دو خاصیت زیر را داشته باشد:

(۱) باید معیار معینی وجود داشته باشد که معیار پایه یا مبنا نامیده می‌شود و با توجه بـه آن فراخوانی به پایان می‌رسد.

(۲) درباری که زیربرنامه (بطور مستقیم یا غیرمستقیم) خودش را صدا می‌زنـد. بایـد بـه معیار پایه نزدیک تر شود.

۳,۵,۱ تابع فاکتوریل

حاصلضرب اعداد صحیح مثبت از 1 تا خود n، n فاکتوریل نامیده می‌شود و معمولاً آن را با n! نمایش می‌دهند بنابراین :

$$n! = 1.2.3\ldots(n-2)(n-1)n$$

بنابه قرارداد 0!=1 تعریف می‌شود. بـدین ترتیـب تـابع فاکتورریـل بـرای تمـام اعـداد صحیح مثبت تعریف می‌شود. بنابراین داریم

$$0! = 1 \quad 1! = 1 \quad 2! = 1\times 2 = 2 \quad 3! = 1\times 2\times 3 = 6 \quad 4! = 1\times 2\times 3\times 4 = 24$$

$$5! = 1\times 2\times 3\times 4\times 5 = 120 \qquad 6! = 1\times 2\times 3\times 4\times 5\times 6 = 720$$

ملاحظه می‌کنید که

$6! = 6.5! = 6.120 = 720 \ , \quad 5! = 5.4! = 5.24 = 120$

یعنی برای هر عدد صحیح مثبت n متساوی زیر برقرار است.

$n! = n.(n-1)!$

بنابراین تابع فاکتوریل را می‌توان بصورت زیر تعریف کرد:

تعریف (تابع فاکتوریل)

(الف) اگر n=0 ، آنگاه n!=1

(ب) اگر n>0 آنگاه n!=n.(n-1)!

ملاحظه می‌کنید که این تعریف n! بازگشتی است، چون وقتی از (n-1) استفاده می‌کند به خودش مراجعه می‌کند. بنابراین با توجه به (الف) صفر مقوله پایه است و (ب) مقدار n! به ازای n دلخواه برحسب مقدار کوچکتر n تعریف می‌شود که به مقدار پایه ۰ نزدیک است.

```
long fact (int n)
{
     long nfact
     if n= 0
          nfacts=1 ,
     else
          nfact=n*fact(n-1),
     return nfact
}
```

۳,۵,۲ دنباله فیبوناچی

دنباله زیبا و معروف فیبوناچی که معمولاً با $F_\circ, F_1, F_2$ ... نمایش داده می‌شود بصورت زیر است .

$0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$

یعنی $F_\circ = \circ$ و $F_1 = 1$ هر جمله بصری مجموع دو جمله قبلی است برای مثال، دو جمله بعدی بالا بصورت زیر محاسبه می‌شود.

$21 + 34 = 55 \quad , \quad 55 + 34 = 89$

تعریف رسمی این تابع بصورت زیر است:

(الف) اگر n=0 یا n=1 آنگاه $F_n = n$

(ب) اگر n>1 آنگاه $F_n = F_{n-2} + F_{n-1}$

این مثال دیگری از یک تعریف بازگشتی است چون وقتی از F_{n-2} و F_{n-1} استفاده می‌کند این تعریف به خودش برمی‌گردد در اینجا مقادیر پایه ۰ و ۱ هستند.

می‌توان دنباله فیبوناچی را با تعریف بازگشتی زیر بیان کرد:

```
int fib (int n)
if (n<=1)  return n ;
return (fib(n-2)+ fib(n-1));
```

۳,۵,۳ ضریب اعداد طبیعی

تعریف ضریب اعداد طبیعی مثال دیگری از تعریف بازگشتی است. ضرب a*b که b*a دو عدد صحیح مثبت هستند ممکن است بصورت جمع b باز عدد a با خودش تعریف شود که یک تعریف تکراری است. تعریف بازگشتی آن بصورت زیر است:

```
a*b=a                      if b = = 1
a*b =a * (b-a) + a         if b>1
```

در این تعریف برای محاسبه ۳×٦ ابتدا باید ۲*٦ را محاسبه کرد و سپس ٦ را به حاصل ۲*٦ اضافه نمود برای محاسبه ۲*٦ باید ۱*٦ را محاسبه کرد و سپس ٦ را به آن اضافه نمود. اما ۱*٦ برابر با ٦ است. بنابراین در این الگوریتم بازگشتی حالت توقف، حالت b= =1 است

```
int product (int * , int j)
if (y = = 1)
return (x),
rerurn (x+product(x,y-1));
```

۳,۶ ارزیابی درستی پرانتزها توسط رشته

اکنون که پشته را تعریف کرده و اعمال ابتدائی مربوط به آن را بررسی کردیم، ببینیم که چگونه می‌توان از آن در حل مسایل استفاده کرد. عنوان مثال عبارت ریاضی زیر را که حاوی پرانتزهای تودرتو است درنظر بگیرید:

$$((x*((x+y)/(J-3))+y)/(4-2.5))$$

می‌خواهیم اطمینان حاصل کنیم که پرانتزها به طور صحیح به کاربرده شده است. یعنی می‌خواهیم تست کنیم که:

۱- تعداد پرانتزهای باز و بسته باهم برابرند.

۲- هر پرانتز باز با یک پرانتز بسته مطابقت می‌کند.

عبارتی مثل: A+B یا ((A+B)

شرط اول را نقض می‌کنند و عباراتی مثل:

(A+B))=(C+D) یا)A+B(-C

شرط دوم را نقض می‌کنید.

اکنون مسأله را کمی پیچیده‌تر کرده و فرض می‌کنیم در یک عبارت از ۳ جدا کرده پرانتز، براکت و آکولاد استفاده می‌شود محدوده‌ای که توسط هر کدام از آن‌ها باز می‌شود باید با جداکننده‌ای از همان نوع بسته شود. لذا رشته‌هایی مثل

(A+B],[(A+B])

در این مورد نه تنها باید مشخص شود که چند محدوده باز شده بلکه باید تعیین شود که هر محدوده توسط چه جداکننده‌ای بازشده است. تا در بستن محدوده مشکلی ایجاد شود.

پشته می‌تواند برای نگهداری انواع محدوده‌هایی که باز شده‌اند به کار رود. وقتی که یک باز کننده محدوده مشاهده شد، در پشته نگهداری شود. پس از رسیدن به یک جداکننده خاتمه دهنده محدوده، عنصر بالای پشته بررسی شود. اگر پشته خالی باشد، این خاتمه دهندهٔ محدوده با هیچ بازکننده محدوده‌ای مطابقت نشده و رشته نامعتبر است. اگر پشته خالی نباشد، عنصر را از پشته حذف کرده و چنانچه نوع آن بانوع خاتمه دهنده محدوده

یکسان باشد به پیمایش رشته ادامه داده وگرنه رشته نامعتبر است پس از رسیدن انتهـای رشته پشته باید خالی باشد، در غیراینصورت، محدوده‌ای باز شـده، ولـی بـسته نـشده و رشته معتبر نیست. الگوریتم این روند بصورت زیر است :

الگوریتم تشخیص صحت پرانتزگذاری

valid = true;
S = the empty stack
while (we have not read the entire entrire stning)
{ read the next symbel of the striny;
if (symb = = (' || symb = = [|| symb = = "{")
push (s,symb);
if (symb = = ')' || symb = = ']' || symb = = '}')
if (empty (s))
valid = false;
else
I = pop(s);
If (i is not the matching opener of symb)
Walid= false
If (! Empty (S))
valid = false;
If (valid)
Print the strin is valid
Else print the string is invalid
}

۳,۷ طراحی و ساخت کلاس پشته

ما در این بخش با توجه به نوع داده انتزاعی پشته، طراحی و **سـاخت** کـلاس پـشته در زبان C++ می‌پردازیم. ساختن کلاس پشته در دو مرحله انجـام مـی‌گیـرد: ۱- طراحـی کلاس پشته ۲- پیاده‌سازی کلاس پشته.

طراحی کلاس پشته

کلاس، شیء دنیای واقعی را مدل‌سازی می‌کند و برای طراحی کلاس لازم است عملیات دستکاری کننده شیء شناسایی شوند صرف زمان بیشتر در این مرحله، ارزشمند است، زیرا کلاس خوبی طراحی می شود که کاربرد آن ساده است.

در نوع داده انتزاعی پشته ما ۵ عمل اصلی را مشخص کردیم. بنابراین کلاس پشته حداقل باید این ۵ عملیات را داشته باشد.

پیاده‌سازی کلاس پشته

پس از طراحی کلاس، باید آن را پیاده‌سازی کرد. پیاده‌سازی کلاس شامل دو مرحله است:

۱- تعریف اعضای داده ای برای نمایش شیء پشته

۲- تعریف عملیاتی که در مرحله طراحی شناسایی شوند.

در کلاس پشته، اعضای داده‌ای، ساختار حافظه را برای عناصر پشته تدارک می‌بینند که برای پیاده‌سازی عملیات مفید هستند.

با توجه به آن چه که گفته شد، دو عضو داده‌ای برای پشته درنظر می‌گیریم:

آرایه‌ای که عناصر پشته را ذخیره می‌کند.

یک متغیر صحیح که بالای پشته را مشخص می‌کند.

توابع عضو کلاس پشته را با استفاده از عملیات تعریف شده بر روی آن می‌توان تشخیص داد این توابع عبارتند از:

Stack() : پشته خالی را ایجاد می‌کند که سازنده کلاس است.

Empty() : خالی بودن پشته را بررسی می‌کند.

Push(): عنصری را در بالای پشته اضافه می‌کند.

Pop() : عنصر بالای پشته را حذف می‌کند.

Top(): عنصر بالای پشته را بازیابی می‌کند.

Display(): محتویات پشته را نمایش می‌دهد.

با توجه به اعضای داده‌ای و توابع عضو کلاس پشته کلاس پشته را برای پشته‌ای از مقادیر صحیح می‌توان به صورت زیر نوشت:

تعریف کلاس پشته

```
#  define size 5
 class stack {
       public :
            stack ( );
            int empty ( )
           void push (int x) ;
           int pop ( ) ;
           int top ( ) ;
          void display ( ) ;
     private:
         int mytop ;
         int item[size] ;
                 }
```

در اینجا فرض کرده‌ایم عناصری که در پشته ذخیره می‌شوند. از نوع صحیح‌اند و تعداد عناصر پشته بیشتر از size نیست.

عناصر پشته می‌توانند از هر نوعی باشند. حتی ممکن است با استفاده از یونیون، پشته‌هایی با عناصر متفاوت را تعریف کرد.

پس از تعریف کلاس پشته، باید شیء از آن کلاس را تعریف و از آن استفاده کرد، به عنوان مثال به دستور زیر پشته s را از نوع کلاس stack تعریف می‌کنیم.

Stack s;

برای سهولت در ادامه بحث فرض می‌کنیم که عناصر پشته همنوع هستند و در نتیجه نیازی به یونیون نیست متغیر mytop باید از نوع صحیح باشد زیرا نشان دهنده موقعیت صفر بالای پشته در آرایه item است.

پیاده‌سازی عمل ایجاد پشته

عمل ایجاد پشته باید پشته‌هایی را ایجاد نماید متغیر نشان‌دهنده بالای پشته، mytop است که در پشته خالی برابر با ۱- است.

بنابراین عمل ایجاد پشته بصورت زیر پیاده‌سازی می‌شود.

```
Stack :: stack ()
Mytop = -1 ;
```

پیاده‌سازی عمل قسمت خالی بودن پشته

اگر s پشته موردنظر و mytop نشان دهنده عنصر بالای پشته باشد. mytop در پشته خالی برابر با ۱- است تابع empty() را می‌توان به صورت زیر پیاده‌سازی کرد:

```
int stack :: empty ()
return (myTop = = -1);
```

پیاده‌سازی عمل حذف از پشته

همان‌گونه که در درس نیز اشاره گردید، نمی‌توان عنصری را از پشته خالی حذف کرد. بنابراین در عمل حذف از پشته باید این مسئله را درنظر داشت عمل pop() سه وظیفه زیر را انجام می‌دهد.

۱- اگر پشته خالی باشد پیام انتظار را چاپ کرده اجرای برنامه را خاتمه می‌دهد.

۲- عنصر بالای پشته را حذف می‌کند.

۳- عنصر بالای پشته را برنامه فراخوان بر می‌گرداند.

پیاده‌سازی عمل حذف از پشته

int stack :: POP ()
if (empty ( ))
cout<< "stack is empty"
exit ( )
else
return items [my top..]

برای استفاده از این تابع می‌توان بصورت زیر عمل کرد:

```
Stack S;
int x;
X=s.pop ();
X=s.pop ();
```

با اجرای این دستورات آنچه که توسط تابع pop() برگردانده می‌شود در متغیر X قـرار می‌گیرد.

پیاده‌سازی عمل افزودن به پشته

این تابع را می‌توان بصورت زیر نوشت:

پیاده‌سازی عمل افزودن به پشته

void stack :: push(int x)
if (my top = = size – 1)
   cout << stack is full. Pressakey …"
   getch ();
   exit();
else
   item [++my top] = x;

پیاده‌سازی عمل بازیابی از پشته

عمل بازیابی از پشته، عنصر بالای پشته را بازیابی می‌کند ولی آن را از پشته حذف نمی‌کند. بدیهی است که این عمل باید خالی بودن پشته را بررسی کند. اگر پشته خالی باشد، امکان بازیابی عنصر وجود ندارد این تابع را می‌توان بصورت زیر نوشت:

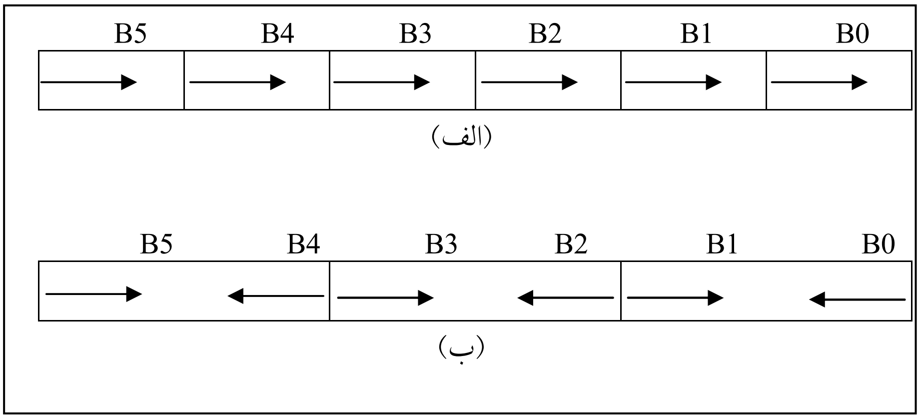
پیاده‌سازی عمل بازیابی از پشته

```
int stack : : top ()
if (empty (s))
   cout << "stack is empty. Press key .."
   getch ();
   exit ();
else
   return item [my top];
```

مثال های حل شده

۱- فرض کنید $k=6$ پشته فضای معلوم s که از N خانه همجوار حافظه تشکیل شـده، اختصاص داده شده است. روش‌های نگهداری پشته‌ها را در s توضیح دهید.

حل: فرض کنید هیچ اطلاعاتی از قبل در دست نیست تـا بیـان کنـد یـک پـشته خیلـی سریع‌تر از پشته دیگری رشد می‌کند. آنگاه می‌توان N/K خانه بـرای هـر پـشته درنظـر گرفت این عمل در شکل (الف) نشان داده شـده اسـت کـه در آن $B_1,B_2,\ldots,B_6$ بـه ترتیب عناصر پایین پشته‌ها را نمایش می‌دهد. یا می‌توان پشته‌ها را به دو قسمت تقـسیم کرد و $2N/K$ خانه حافظه برای هر جفت پشته بصورتی که در شـکل (ب) نـشان داده شده است ا ختیار کرد. روش دوم می‌تواند تعداد دفعات وقوع سرریزی را کاهش دهد.



(الف)

(ب)

۲- فرض کنید a و b نمایش در مورد صحیح مثبت باشند. فرض کنید تابع Q به شـکل زیر به صورت بازگشتی تعریف شده است:

$$Q(a,b) = \begin{cases} 0 & \text{if } a < b \\ Q(a-b,b)+1 & \text{if } b \leq a \end{cases}$$

(الف) تعداد $Q(2,3)$ و $Q(4,3)$ را پیدا کنید.

این تابع چه عملی انجام می‌دهد؟ مقدار $Q(5861,7)$ را پیدا کنید.

حل (الف) $Q(2,3) = 0$ چون $2<3$

$Q(4,3) = Q(11,3) + 1 = 4$
$Q(8,3) + 1 = 3$
$Q(5,3) + 1 = 2$
$Q(2,3) + 1 = 1$

پس جواب $Q(14,3) = 4$

(ب) هر بار که b از a کم می‌شود مقادیر Q یک واحد افزایش مـی‌یابـد از ایـن رو Q (a,b) وقتی a بر b تقسیم می‌شود خارج قسمت را پیدا می‌کند. بنابراین

$$Q(5861,7) = 837$$

۳- فرض کنید n یک عدد صحیح مثبت باشد. فرض کنیـد تـابع بازگـشتی L بـصورت زیر تعریف شده است:

$$L(n)=\begin{cases} 0 & \text{if } n=1 \\ L([n/2])+1 & \text{if } n>1 \end{cases}$$

در اینجا [K] کف عدد K را نشان می‌دهد و نشان دهنده بزرگترین عدد صحیحی است که بزرگتر از K نباشد.

(الف) مقدار $L(25)$ را بدست آورید.

(ب) این تابع چه عملی را انجام می‌دهد.

(الف)

$L(25) = L(12) + 1 \qquad = 4$
$L(6) + 1$
$L(3)+1$
$L(1)+1$

ب) هر بار که n بر ۲ تقسیم می‌شود تعداد L یک واحد افزایش مـی‌یابـد از ایـن رو L بزرگترین عدد صحیحی است که

$n \le 2L$

بنابراین L[log2n] را بدست می‌دهد.

٤- یک پشته خالی با اعداد از 1 تا 6 در ورودی داده شده است. اعمال زیر بر روی پشته قابل انجام هستند:

push : کوچک‌ترین عدد ورودی را برداشته و وارد پشته می‌کنیم.

Pop : عنصر بالای پشته را در خروجی نوشته و سپس آن را حذف می‌کنیم.

موارد زیر را بررسی کنید و بگویید کدام ترتیب را نمی‌توان با هیچ عملی از push و pop در خروجی چاپ نمود. (اعداد را از چپ به راست بخوانید)

الف) 2 1 5 3 4 6   ب) 1 2 3 5 6 4

ج) 4 3 2 1 6 5   د) 3 2 4 6 5 1

حل. با توجه به عملکرد پشته هنگامی که به یک عدد بزرگ‌تر از پشته خارج می‌شود کلیه اعداد کمتر از آن باید به ترتیب نزولی خارج شوند (چون به ترتیب صعودی در پشته قرار گرفته‌اند) این عملکرد را به صورت قضیه زیر بیان می‌کنیم:

قضیه: ورودی n , ... , 3 , 2 , 1 یا A , B , C ,..., Z (از چپ به راست) را در نظر بگیرید که بر روی پشته با استفاده از push یا pop اعمال می‌شوند. دنباله p_1 , p_2 , p_3,...,p_n در خروجی قابل تولید است اگر و تنها اگر هیچ اندیسی همانند $i < j < k$ وجود نداشته باشد که $pj<pk<pi$ باشد.

حال با توجه به قضیه فوق موارد سؤال را بررسی می‌کنیم.

الف) اندیس‌ها $\rightarrow$ 1 2 3 4 5 6

در این دنباله اندیس 3<4<5 باید $p3 < p4 < p5$ باشد که بدین صورت نیست، بلکه $p4<p5<p6$ پس با هیچ ترتیبی نمی‌توان آن را از پشته خارج نمود.

سایر گزینه‌ها را می‌توان در خروجی چاپ نمود.

٥- اگر دنباله اعداد1,3,4,5,7 به ترتیب از سمت چپ به راست وارد پشته کنیم، کـدام یک از خروجی‌های زیر از پشته امکان‌پذیر نیست؟

الف)1 3 4 5 7 ب)4 5 7 3 1

ج)4 5 3 7 1 د)5 7 3 4 1

در گزینه ج اندیس 4<3<2 ولی p2< p4< p3 پس آن را با هیچ ترتیبی نمی‌تـوان در خروجی چاپ کرد.

٦- عبارت پسوندی (postfix) پیشوندی معادل عبارت ریاضـی a/b-c+d*e-a*c/d را بدست آورید.

حل: ابتدا عبارت مورد نظر را بطور کامل پرانتزگذاری می‌کنیم.

((((a/b)-c)+(d*e))-((a*c)/d))

برای بدست آوردن عبارت پسوندی عملگرها را بـه بعـد از پرانتـز مربـوط بـه خـودش انتقال می‌دهیم و سپس پرانتزها حذف می‌کنیم.

ab/c-de*+ac*d/-

و برای بدست آوردن عبارت پیشوندی عملگرها را به قبل از پرانتز مربـوط بـه خـودش انتقال می‌دهیم و سپس پرانتزها را حذف می‌کنیم.

- + -/abc*de/*ac d

٧- معادل پیشوندی و پسوندی عبارت میانوندی زیر را پیدا کنید.

$$\left((A+B)*(C-D)\right)$$

معادل پیشوندی. هر عملگر را به قبل از پرانتز مربوط به خود انتقال می‌دهیم.

$$\left((A+B)*(C-D)\right)=*+AB-CD$$

معادل پسوندی. هر عملگر را به بعد از پرانتز مربوط به خود انتقال می‌دهیم.

$$\left((A+B)*(C-D)\right)=AB+CD-*$$

٨- عبارت پیشوندی زیر را به عبارت پسوندی معادل تبدیل کنید.

$$/-*+ABC-DE+FG$$

ابتدا عبارت را به میانوندی تبدیل می‌کنیم و از آن به پسوندی تبدیل می‌کنیم. هر عملگر مربوط به نزدیک‌ترین دو عملوند است.

$$\big((((A+B)*C-(D-E)\big)/(F+G)\big)$$
$$= AB+C*DE--FG+/$$

۹- عبارت ریاضی زیر را به روشهای مختلف پارانتز گزاری کنید.

X = A/B – C + D * E – A * C

برخی از روشهای پارانتزگزاری آن بصورت زیر می باشد

A/(B-C) + D *E – A * C
(A/B) – (C + D) * (E-A) * c
A/(B-C+D*E) – (A*C)

حال عبارت اصلی را با توجه به جدول تقدم زیر پرانتزگزاری کنید.

Priority	**Operator**
1	Unary -, !
2	*,/,%
3	+,-
4	<,<=,>,>=
5	==,!=
6	&&
7	||

با توجه به جدول فوق پارانتزگزاری X به صورت زیر می شود.

X = A/B – C + D * E – A * C
  = ((((A/B) – C) + (D*E)) – (A*C))

۱۰- اگر نمایش میانوندی یک عبارت بـه صـورت زیـر باشـد، نمـایش پـسوندی آن را بدست آورید.

Infix: A / B – C + D * E – A * C

جواب:

Postfix: A B / C - D E * + A C * -

تمرین های فصل

۱) زبان فرضی در نظر بگیرید که در آن، آرایه به عنـوان نـوعی داده نیـست بلکـه پشته به عنوان نوعی داده است. یعنی تعریف زیر ممکن است:

Stack s;

همچنین فـرض کنیـد اعمـال push، pop، test emptyو top در ایـن زبـان تعریف شده اند. نشان دهید که یک آرایه یک بعدی چگونه می تواند بـا اسـتفاده از این اعمال بر روی دو پشته پیاده سازی شود.

۲) هر یک از عبارات زیر را به عبارات perfix و postfix تبدیل کنید.

a. A+B-C
b. (A+B)*(C-D)$E*F
c. A+B/C+D
d. A-(B-(C(D-E)))
e. (A+B)/C+D
f. (A-B)*(C-(D+E))
g. (A + B) * (C + D) – E
h. A + B * (C + D) – E / F * G + H
i. ((A + B) / (C – D) + E) * F – G

۳) عبارتهای پسوندی زیر را به عبارت میانوندی تبدیل کنید:

a. a b + c d - *
b. a b c + - d *
c. a b c d / / /
d. a b + c – d e * /
e. a b / c / d /

٤) عبارات پسوندی زیر را به ازای a=7، b=4، c=3 و d=2 ارزیابی کنید:

a. a b c +/ d *
b. a b c - - d -
c. a b c d - - -
d. a b c + + d +

e. a b + c / d *

۵) برنامه ای بنویسید که عبارتی محاسباتی را به صورت رشته خوانده و آن را از نظر درستی پرانتزگذاری بررسی کند و در صورتی که تعداد پارانتزهای باز و بسته یکسان نباشد پیغام خطا دهد.

٦) برنامه ای بنویسید که عناصری را خوانده و در پشته ذخیره کند و سپس با حداقل حافظه کمکی عناصر پشته را معکوس کند.

۷) برنامه ای بنویسید که perfix را به postfix و برعکس تبدیل کند.

۸) برنامه ای بنویسید که یک عبارت میانوندی را خوانده و تمام پارانتزهای اضافی را حذف کند.

۹) اگر کاراکترهای A، B، C و D به ترتیب وارد پشته شوند. چه خروجی هایی از این پشته امکان پذیر خواهد بود.

۱۰) پنج مثال برای کاربرد واقعی پشته نام ببرید.

۱۱) در مسئله برجهای هانوی، n حلقه که دارای شعاعهای ۱ تا n هستند به ترتیب نزولی روی یک میله قرا دارند. دو میله خالی نیز وجود دارند. هدف مسئله انتقال حلقه ها به میله سوم است به طوری که ترتیب حلقه ها تغییر نکند. در ضمن در هر حرکت مجاز به انتقال یک حلقه به میله های دیگر هستید به طوریکه هیچگاه یک حلقه بزرگتر روی یک حلقه کوچکتر قرار نگیرد. فرض کنید که شما مجاز به استفاده از یک پشته آرایه ای با اندازه محدود (m) هستید. در این حالت می توانید از هر کدام از میله ها یک حلقه به پشته اضافه کنید و برعکس. توضیح دهید که چگونه می توان با استفاده از این پشته مسئله برجهای هانوی را برای n های بزرگتر از m حل کرد.

۱۲) توضیح دهید که چگونه با پشته می توان بزرگترین مقسوم علیه مشترک دو عدد دلخواه را پیدا کرد.

۱۳) برنامه ای بنویسید که مراحل زیر را انجام دهد:

a. یک پشته ایجاد کنید.

b. تابعی بنویسید که یک رشته را از کاربر بگیرد و مشخص کند که آیا کلمه دوطرفه هست یا نه؟ برای این کار از پشته قسمت a استفاده کنید.

فصل چهارم

صف ها

اهداف

در پایان این فصل شما باید بتوانید:

- ✓ صف را تعریف کرده و برخی از کاربردهای آن را نام ببرید.
- ✓ اعمال درج و حذف ار صف را پیاده سازی کنید.
- ✓ مشکلات صف را عنوان کرده و چگونگی حل آن را بیان کنید .
- ✓ چگونگی پیاده سازی صف حلقوی را توضیح دهید.
- ✓ مشکلات پیاده سازی صف با استفاده از آرایه را توضیح دهید.
- ✓ آیا صف جوابگوی تمام نیازهای ما برای تعریف داده های مورد نیاز برنامه می باشد؟

سوالهای پیش از درس

۱- به نظر شما با توجه به پشته لزوم تعریف یک ساختار داده جدید ضروری بنظر می رسد؟

....

....

۲- مثالهایی از صف را در دنیای واقعی نام ببرید.

....

....

۳- با توجه به معنی صف در دنیای واقعی ، آن را با پشته مقایسه کنید؟

....

....

مقدمه

یک صف- لیست خطی از عناصر است که در آن عمل حذف عناصر تنها می‌تواند از یک انتهای آن موسوم به سر صف یا ابتدای صف front و عمل اضافه شدن تنها می‌تواند از انتهای دیگر آن موسوم به ته صف یا انتهای آن rear صورت گیرد.

قابل ذکر است اصطلاح ابتدای صف front و انتهای صف rear در توصیف یک لیست خطی تنها وقتی مورد استفاده قرار می‌گیرد که به عنوان یک صف پیاده‌سازی شود.که front نشان دهنده ابتدای صف و rear نشان دهنده انتهای صف می باشد.

صف‌ها را لیست‌های اولین ورودی اولین خروجی یا (First Input First output)FIFO می‌نامند. چون اولین عنصری که وارد صف می‌شود اولین عنصری است که از آن خارج می‌شود. صف ها در مقابل پشته‌ها قرار دارند که لیست‌های آخرین ورودی اولین خروجی (LIFO) هستند.

صف‌ها در زندگی روزمره ما به وفور دیده می‌شوند. به عنوان مثال صفی که مردم برای گرفتن نان در جلوی نانوایی تشکیل می‌دهند یا صفی از کارها در سیستم کامپیوتری که منتظرند از یک دستگاه خروجی مثل چاپگر استفاده کنند و مثالی دیگر از صف در علم کامپیوتر، در سیستم اشتراک زمانی اتفاق می‌افتد که در آن برنامه‌هایی که دارای اولویت یکسان هستند تشکیل یک صف می‌دهند و در حال انتظار برای اجرا بسر می‌برند.

٤,١ نوع داده انتزاعی صف

با توجه به تعریف و عملکرد صف که از آن کردیم می‌توان آن را به صورت یک نوع داده انتزاعی به صورت زیر تعریف نمود:

عناصر داده:

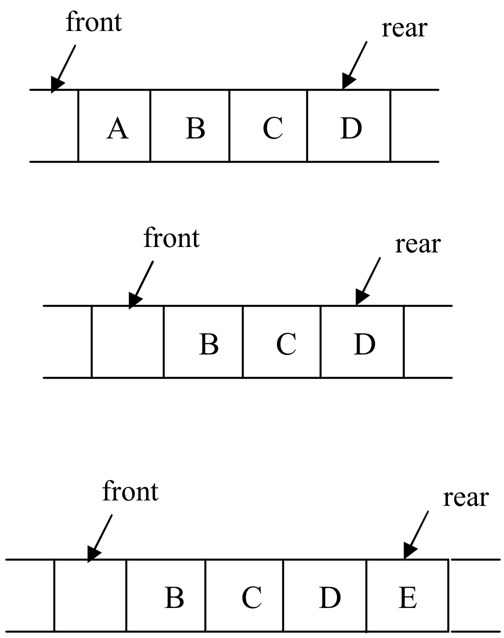
مجموعه‌ای مرتب از عناصر که در آن، عناصر از یک طرف موسوم به جلوی صف (Front) حذف و از طرف دیگر موسوم به ته یا آخر صف (rear) اضافه می‌گردند.

عملیات اصلی:

- Create محل ایجاد یک صف خالی
- Empty عمل تست خالی بودن صف
- Add افزودن عنصری به آخر صف
- Delete حذف عنصری از ابتدای صف
- Process بازیابی عنصری از جلوی صف

صرفنظر از نوع پیاده‌سازی صف، در آن از دو متغیر اشاره‌گری به نام front که همیشه به عنصر قبل از عنصر ابتدایی اشاره می‌کند و دیگری rear که همیشه به آخرین عنصر اشاره می‌کند، استفاده می‌کنیم. شکل ۱-٤ صفی را نشان می‌دهد که حاوی چهار عنصر A , B , C , D می‌باشد. A در جلوی صف و D در انتهای صف قرار دارد. همانطور که مشاهده می کنید front به عنصر قبل از عنصر ابتدایی اشاره می‌کند و rear به آخرین عنصر اشاره می‌کند در شکل (۱-٤ب) عنصری از صف حذف شده است و چون عناصر فقط از جلوی صف حذف می‌شوند عنصر A حذف می‌شود و B در جلوی صف قرار می‌گیرد. نحوه حذف کردن بدین صورت می تواند انجام شود که front را یک خانه به جلو حرکت دهیم و خانه ای که front حال به آن اشاره می کند را آزاد کنیم. در شکل (۱-٤ج) عنصر E به صف اضافه شده است. این عنصر به آخر صف اضافه شده است. و نحوه اضافه کردن عنصری به صف بدین صورت می

تواند انجام شود که rear را یک خانه به جلو حرکت دهیم تا به خانه خالی اشـاره کنـد و در خانه خالی E را قرار دهیم.



(الف)

(ب)

(ج)

شکل ۱-٤

٤,۲ نمایش صف‌ها

صف‌هـا را مـی‌تـوان در کـامپیوتر بـه صـورت‌هـای متفـاوتی نمـایش داد. یـک روش پیاده‌سازی صف این است که از یک آرایه برای ذخیره کردن عناصر صـف و دو متغیـر front و rear به ترتیب برای نمایش ابتدا و انتهای صـف اسـتفاده گـردد. هـر یـک از صف‌های داخل کتاب توسط یک آرایه خطـی queue و دو متغیـر اشـاره‌گـر front و rear پیاده‌سازی می‌گردد. نمونه‌ای از یک صف را که شامل اعداد صحیح است ممکـن است به صورت زیر تعریف شود:

تعریف صف برای ذخیره اعداد صحیح

```
#define  maxqueue  100
struct  queue  {
        int  items[maxqueue];
        int  front, rear;
                }q;
```

که در پیاده‌سازی صف شرایط اولیه زیر را در نظر می‌گیریم.

مقداردهی اولیه   rear = = front = = -1

پر بودن صف   rear = = maxqueue –1

خالی بودن صف   rear = = front

زیر برنامه اضافه کردن به صف addqueue (q , x) به صورت زیر می‌باشد:

پیاده سازی زیر برنامه اضافه کردن یک عنصر به صف

```
void addqueue (int rear , int  item)
{
   if (rear = = maxqueue –1)
           queuefull( );           //پیغام پر بودن صف داده می‌شود
   else
          queue [+ + rear] = item;
}
```

در زیر برنامه فوق ابتدا پر بودن صف کنترل می‌شود، که در صورت پر بودن آن پیغام «صف پر است» داده می‌شود و در غیر این صورت ابتدا یک واحد به rear اضافه می‌شود (چون rear به خانه آخرین عنصر آرایه اشاره می‌کند، یک واحد به آن اضافه می‌شود تا به خانه‌ای خالی اشاره کند) و سپس عنصر مورد نظر به این خانه اضافه می‌شود.

و زیر برنامه حذف اولین عنصر از صف delequeue (q) به صورت زیر است.

پیاده سازی زیر برنامه حذف کردن یک عنصر از صف

```
int  delequeue (int  front , int  rear)
        {
          if (front = = rear)
            queue empty ( )    // پیغام پر بودن صف داده می‌شود
        else
            return queue [+ + front];
        }
```

اولین زیر برنامه نیز ابتدا خالی بودن صف کنترل می‌شود. چون از صف خالی نمی‌توان عنصری حذف کرد. سپس چون همیشه front به یک خانه جلوتر از اولین عنصر اشاره می‌کند، ابتدا به front یک واحد اضافه می‌کنیم تا به اولین عنصر اشاره کند و بعداً این عنصر را به برنامه فراخواننده برمی‌گردانیم.

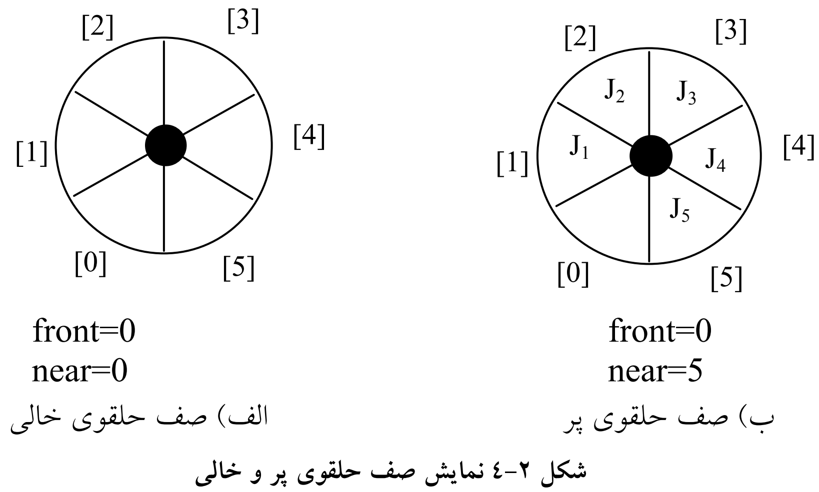
وقتی اندیس انتها (rear) برابر با maxqueue-1 می‌شود و در نتیجه به نظر می‌رسد که صف پر می‌باشد، در حالی که امکان دارد به دلیل حذف عنصری از صف اوایل صف خالی باشد- پس مشکل اصلی صف معمولی این است که فقط یک بار قابل استفاده است.

مشکل اصلی صف معمولی این است که فقط یک بار قابل استفاده است.

یک روش برای حل این مشکل این است که تمام عناصر به ابتـدای صـف شـیفت داده شوند که تغییر مکان عناصر در یک آرایه بسیار وقت‌گیر می‌باشد، مخصوصاً اگـر آرایـه دارای عناصر زیادی باشد. در واقع در بدترین حالت O(maxqueue) می‌باشد. بـرای رفع این مشکل از صف حلقوی استفاده می‌کنیم.

٤,٣ صف حلقوی

صف حلقوی نمایش مؤثرتری برای پیاده‌سازی عملکرد صف مـی‌باشـد. در ایـن صـف اندیس ابتدا (front) همیشه به یک موقعیت عقب‌تر از اولین عنـصر موجـود در صـف اشاره می‌کند و اندیس انتها (rear) به انتهای فعلی صف اشاره می‌کند. اگر front = = rear باشد، صف خالی خواهد بود. اگر در صف حلقوی فقط دارای یک مکـان خـالی باشد، اضافه کردن یک عنصر موجب می‌شود که front = = rear شود که همان شرط خالی بودن صف است در حالی که صف خالی نیست. یعنی نمی‌توانیم یک صف پـر و خالی را از هم تشخیص دهیم. به همین دلیل در یک صف حلقوی به انـدازه n در هـر لحظه حداکثر n-1 عنصر وجود دارد بدین ترتیب می‌توان بین حالت پر و خـالی تمـایز قایل شد. شکل ٢-٤ نمایشی از صف‌های حلقوی تهی و غیرتهی می‌باشد.



شکل ٢-٤ نمایش صف حلقوی پر و خالی

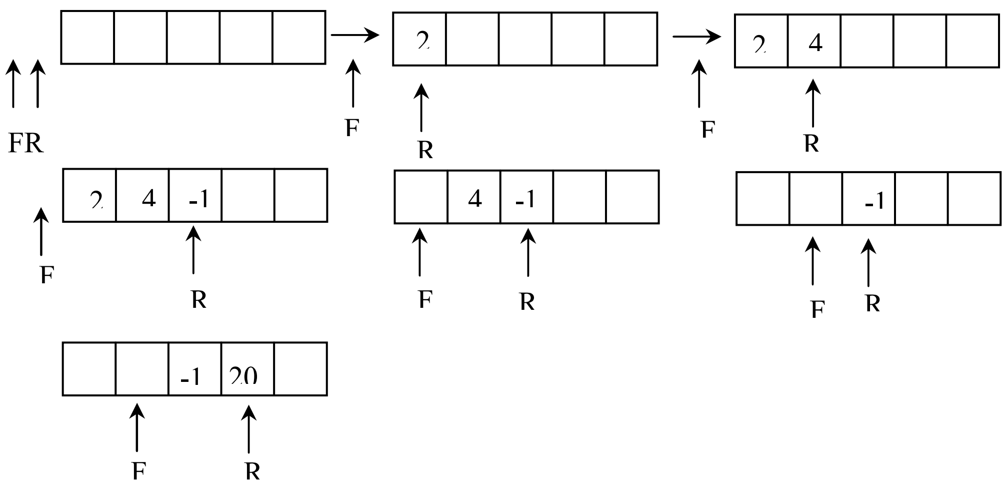
مثال ٤,١: صف امروزه در بسیاری از مسائل کامپیوتر کاربرد دارد و شـاید متـداول‌تـرین مثال، ایجاد یک صف از برنامه‌ها به وسیله سیستم عامل باشد. اگر سیستم عامـل مـسأله تقدم را در نظر نگیرد، برنامـه‌هـا بـه همـان ترتیبـی کـه وارد سیـستم مـی‌شـوند، اجـرا می‌گردند. شکل (٣-٤) نشان می‌دهد که چگونه یک سیستم عامل ممکن است برنامه‌هـا را در صورت نمایش ترتیبی صف اجرا کند.

Front	Rear	Q[0]	Q[1]	Q[2]	Q[3]	توضیح
-1	-1					Queue is empty
-1	0	J_1				Queue 1 is empty
-1	1	J_1	J_2			Queue 2 is empty
-1	2	J_1	J_2	J_3		Queue 3 is empty
0	2		J_2	J_3		Queue 1 is empty
1	2			J_3		Queue 2 is empty

شکل ٣-٤: جایگذاری و حذف از یک صف ترتیبی

مثال ٤,٢ : عملیات زیر را به ترتیب از چپ به راست روی صف اعمال می‌کنیم.

Add q(2) , add q(4) , add q(-1) , dele q(A) , dele q(B) , add(20)



برنامه حذف یک عنصر از صف حلقوی به صورت زیر است:

پیاده سازی زیر برنامه حذف کردن یک عنصر از صف حلقوی

```
int delete q  (int * front , int rear)
        {
           dement  item;
           /* remove front element from the queue and put it in item */
             if (* front = = rear)
                  queue  empty ( );
              else
                 * front = (* front + 1) % max-Queue-size;
                  return queue [* front];
        }
```

همان گونه که مشاهده می‌کنید که تست پر بودن یک صف حلقوی در addq و خالی بودن صف حلقوی در deleteq یکسان می‌باشند.

توابع ( ) queue-empty و ( ) queue–full بدون توضیح ارائه شده‌اند. پیاده‌سازی آنها بسته به کاربردهای خاص می‌باشد و یا فقط می‌توانند یک پیغام خط برگردانند.

پیاده‌سازی addq و deleteq برای یک صف حلقوی کمی مشکل‌تر می‌باشد، زیرا باید مطمئن شویم که یک جابجایی و چرخش حلقوی انجام می‌گیرد. این با استفاده از یک عملگر پیمانه‌ای (modular) بدست می‌آید. چرخش حلقوی rear به صورت زیر انجام می‌گیرد.که n بیانگر حداکثر اندازه حلقوی می‌باشد

rear = (rear+1) % n

در زیر **پیاده سازی اضافه کردن یک عنصر به صف حلقوی** نشان داده شده است.

پیاده سازی زیر برنامه اضافه کردن یک عنصر به صف حلقوی

```
void addq (int front , int * rear , int item)
        {
            /* add an item to the queue */
            * rear = (* rear+1) % max Queue ;
                if (front = = rear)
                    {
                        Queue-full ( )
                    }
                else
                   Queue [* rear] = item ;
        }
```

به همین ترتیب در front , delete q را به وسیله عبارت زیر چرخش می‌دهیم.

front = (front+1) % n

٤,٤ صف اولویت

صف و پشته ساختمان داده‌هایی هستند که ترتیب عناصر آنها، همان ترتیب ورود به آنها است. عمل pop آخرین عنصری را که در پشته قرار گرفته است حذف می‌کند و عمل delete q اولین عنصری را که در صف وجود دارد حذف می‌کند. اگر بین عناصر یک ترتیب طبیعی موجود داشته باشد (مثل ترتیب عددی یا الفبایی)، در اعمال مربوط به صف و پشته نادیده گرفته می‌شوند.

صف اولویت (Priority queue) ساختمان داده‌ای است که در آن ترتیب طبیعی عناصر، نتایج حاصل از عملیات آن را مشخص می‌کند. در این نوع صف، عمل اضافه

کردن عنصر جدید به هر ترتیبی امکان‌پذیر است ولی حذف یک عنصر از آن به صورت مرتب انجام می‌شود. صف اولویت بر دو نوع است: صف اولویت صعودی و صف اولویت نزولی. صف اولویت صعودی صفی است که درج عناصر در آن به هر صورتی امکان‌پذیر است ولی در موقع حذف کوچک‌ترین عنصر حذف می‌شود.
صف اولویت نزولی همانند صف صف اولویت صعودی است با این تفاوت که در عمل حذف بزرگ‌ترین عنصر صف حذف می‌شود.

٤,٥ طراحی و ساخت کلاس صف

ما در این بخش با توجه به نوع داده انتزاعی صف، طراحی و **ساخت** کلاس صف در زبان C++ می‌پردازیم. ساختن کلاس پشته در دو مرحله انجام می‌گیرد: ۱- طراحی کلاس صف ۲- پیاده‌سازی کلاس صف.

طراحی کلاس صف

کلاس، شیء دنیای واقعی را مدل‌سازی می‌کند و برای طراحی کلاس لازم است عملیات دستکاری کننده شیء شناسایی شوند صرف زمان بیشتر در این مرحله، ارزشمند است، زیرا کلاس خوبی طراحی می شود که کاربرد آن ساده است.
در نوع داده انتزاعی صف ما ۵ عمل اصلی را مشخص کردیم. بنابراین کلاس صف حداقل باید این ۵ عملیات را داشته باشد.

پیاده‌سازی کلاس صف

پس از طراحی کلاس، باید آن را پیاده‌سازی کرد. پیاده‌سازی کلاس شامل دو مرحله است:

۱- تعریف اعضای داده ای برای نمایش شیء پشته

۲- تعریف عملیاتی که در مرحله طراحی شناسایی شوند.

در کلاس صف، اعضای داده‌ای، ساختار حافظه را برای عناصر پشته تدارک می‌بینند که برای پیاده‌سازی عملیات مفید هستند.

با توجه به آن چه که گفته شد، سه عضو داده‌ای برای صف درنظر می‌گیریم:

آرایه‌ای که عناصر صف را ذخیره می‌کند.

به دو متغیر صحیح که ابتدا و انتهای صف را مشخص می‌کند.

توابع عضو کلاس صف را با استفاده از عملیات تعریف شده بر روی آن می‌توان تشخیص داد این توابع عبارتند از:

تابع queue() : سازنده ای است که صف خالی را ایجاد می کند.

تابع empty(): خالی بودن صف را تست می کند. اگر صف خالی باشد مقدار ۱ و گرنه صفر را برمیگرداند.

تابع addq() : عنصری را به آخر صف اضافه می کند.

تابع deleteq() : عنصری را از جلوی صف حذف می کند.

تابع process() : عنصر جلوی صف را بازیابی می کند.

با توجه به اعضای داده‌ای و توابع عضو کلاس صف کلاس صف را برای صفی از مقادیر صحیح می‌توان به صورت زیر نوشت:

تعریف کلاس صف

```
#  define size 5
 class queue {
       public :
          queue ( );
          int empty ( )
          void addq (int &, int & ) ;
          void process (int &, int & ) ;
          void deleteq (int &, int & ) ;
      private:
         int item[size] ;
         int front ;
```

```
         int rear ;
              }
```

پیاده سازی عمل ایجاد صف

```
queue :: queue ()
  {
     front=0;
     rear=-1;
  }
```

پس در ابتدا front برابر با صفر و rear **برابر با ۱- می باشد. بنابراین اگر**

front < rear صف خالی می باشد. و تعداد عناصر در هر لحظه برابر است با:

front-raer+1

پیاده سازی عمل تست خالی بودن صف

```
int queue :: empty ()
  {
     if (rear <  front )
     return 1;
    return 0;
  }
```

پیاده سازی افزودن عنصر به آخر صف

```
int queue :: addq (int &x , int &overflow)
  {
     if (rear == size-1 )
     overflow = 1;
     else
        {
        overflow =0;
     item[++rear]=x;
    }
  }
```

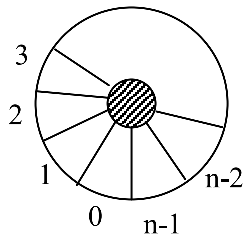
پیاده سازی عمل حذف از جلوی صف

```
int queue :: deleteq (int &x , int &underflow)
```

```
{
   if (empty() )
   underflow = 1;
   else
     {
       underflow =0;
   x= item[front++]=x;
   }
 }
```

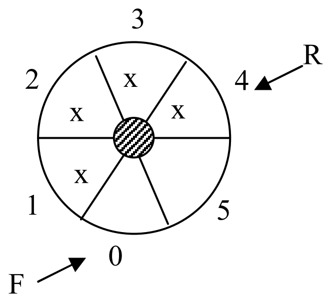
مسائل حل شده در صف‌ها

۱– با توجه به صف حلقوی شکل فرض کنید N تعـداد اقـلام در یـک صـف دایـره‌ای باشد. متغیر F به خانه‌ای که بلافاصله قبل از جلوی صـف قـرار دارد اشـاره مـی‌کنـد و متغیر R به عقب صف اشاره می‌کند. فرمولی که تعداد اقلام در یـک صـف دایـره‌ای را محاسبه می‌کند، محاسبه کنید.



حل: مسئله را برای دو حالت $R > F$ و $R < F$ حل می‌کنیم

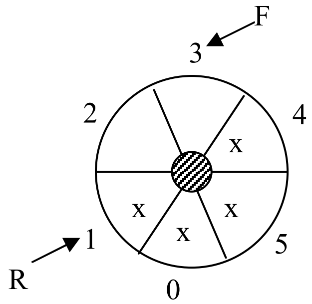
برای $R > F$ شکل فرضی زیر را رسم می‌کنیم.



متوجه می‌شویم که تعداد اقلام برابر است با:

$R - F = 4 - 0 = 4$

برای حالت $R < F$ شکل فرضی زیر را رسم می‌کنیم.



متوجه می‌شویم که تعداد اقلام برابر است با

$$N-(F-R)=6-(3-1)=4$$

پس در حالت کلی داریم

$$N=\begin{cases} n-(F-R) & \text{if } F>R \\ R-F & \text{if } R>F \end{cases}$$

۲- برای یک ساختار با صف حلقوی با $n = 7$، چه حالتی بیان کننده خالی و یا پر بودن صف می‌باشد؟

حل: شرط خالی بودن صف حلقوی آن است که Front = Rear (یعنی حالت‌هایی که با هم برابرند مثل

$$F=۲\ ,R=۲$$
$$F=۳\ ,R=۳$$
$$F=s\ ,R=s$$

در کلیه این حالت‌ها، صف خالی است.

و شرط پر بودن صف

$$(Rear+1)\bmod 7=F \quad \leftarrow (Rear+1)\bmod n=F$$

به عنوان مثال اگر Rear به خانه ٦ اشاره کند و Front به خانه ٥ اشاره کند، داریم

$$(6+1)\bmod 7=0$$

پس صف پر می‌باشد.

۳- در نمایش صف حلقوی به کمک آرایه، چرا از یک خانه استفاده نمی‌شود؟

حل: اگر صف حلقوی دارای ۸ خانه باشد حداکثر از n-1 خانه آن برای ذخیره داده‌ها می‌توان استفاده کرد. اگر از تمام خانه‌ها استفاده شود هنگامی که rear = front شود نمی‌توانیم تشخیص دهیم یک صف پر است یا خالی.

تمرین های فصل

۱) نشان دهید که چگونه می تـوان صـفی از اعـداد صـحیح بـا اسـتفاده از آرایـه queue[100] که queue[0] برای نشان دادن ابتـدای صـف و queue[1] برای نشان دادن انتهـای صـف و queue[2] تـا queue[99] بـرای نمـایش عناصر صف به کار می روند پیاده سازی کرد. نشان دهید چگونـه آرایـه ای را به عنوان صف خالی ارزش دهی کرد؟ روالهای صف (درج ، حذف و تـست خالی) را برای این پیاده سازی بنویسید؟

۲) نشان دهید که چگونه می توان صفی را که عناصر آن متشکل از تعداد متغیـری از اعداد صحیح است پیاده سازی کرد.

۳) یک ADT(نوع داده انتزاعی) برای صف اولویت بنویسید.

٤) چگونه می توان چندین صف را داخل یک آرایه پیـاده سـازی کـرد. روالهـای مربوطه را بنویسید.

٥) چگونه می توان n صف متـوالی حلقـوی را در آرایـه ای بـه طـول q[size] نمایش داد.(روالهای addq ، deleq و empty و full را بنویسید)

٦) برنامه ای بنویسید که کوچکترین عنصر صف را حذف کرده و برگرداند. ضمن اینکه ترتیب بقیه عناصر تغییر نکند. عناصر صف اعداد طبیعی هستند. شما در مورد نحوه پیاده سازی صف نباید هیچ فرضی بکنید. فقط می توانید از متدهای استاندارد صف و یا یک صف دیگر استفاده کنید. پیچیدگی زمانی برنامه شما چقدر است؟

۷) برنامه ای بنویسید که یک رشته را از کاربر بگیرد و ابتدا تمام حروف بزرگ را به ترتیبی که در رشته آمده اند چاپ کند. سپس، تمام حروف کوچک را به ترتیبی که در رشته آمده اند چاپ کند و در نهایت تمام ارقام موجود در رشته را به همان ترتیبی که در رشته آمده اند چاپ کند. برنامه شما باید از سه

صف و از توابع isdigit و islower و isupper که در ctype.h موجود هستند استفاده کند. پیچیدگی زمانی برنامه شما چقدر است؟

۸) توضیح دهید که چگونه می توان عناصر یک پشته را طوری در یک صف قرار داد که عنصرهایی که زودتر وارد پشته شده بودند زودتر از صف خارج شوند. یعنی اولین عنصر صف آخرین عنصر پشته باشد و .... پیچیدگی زمانی راه حل شما چقدر است ؟

۹) توضیح دهید که چگونه با صف می توان کوچکترین مضرب مشترک دو عدد دلخواه را پیدا کرد.

۱۰) توضیح دهید که چگونه می توان توسط دو صف ، یک پشته درست کرد.

۱۱) الگوریتمی ارائه کنید که یک صف را به یک صف دیگر اضافه کرده و صف اولی را تغییرندهد. از متدهای استاندارد صف برای این کار استفاده کنید.

۱۲) توضیح دهید که چگونه می توان توسط دو پشته، یک صف درست کرد.

۱۳) چگونه می توان یک پشته و صف را داخل یک آرایه نمایش داد. توابع حذف و اضافه را بنویسید. آیا این شکل نمایش مناسب است.

۱٤) توضیح دهید چگونه می توان عناصر یک پشته را وارد یک پشته دیگر نمود به نحوی که ترتیب عناصر پشته دوم و اول یکسان باشند. می توانید از یک صف کمکی برای حل مساله استفاده کنید.

۱۵) چگونه می توان با استفاده از متدهای استاندارد صف و بدون استفاده از پشته، ترتیب عناصر یک صف را معکوس کرد. برای حل مساله می توانید از چندین صف استفاده کنید.

۱٦) برنامه ای بنویسید که رشته ای از کاراکترها را از ورودی خواند، هر کاراکتر را هنگام خواندن در یک پشته و و یک صف قرار دهد. وقتی به انتهای رشته رسید، برنامه باید با استفاده از عملیات اصلی پشته و صف تعیین کند آیا رشته

متقارن است یا خیر. (رشته ای متقارن است که وقتی ترتیب آن عوض شـود ، تغییر نمی کند. مثل madam ، 532235)

فصل پنجم

لیست پیوندی

اهداف

در پایان این فصل شما باید بتوانید:

- ✓ لیست پیوندی را تعریف کرده و برخی از کاربردهای آن را نام ببرید.
- ✓ اعمال درج ، حذف و پیمایش در لیست پیوندی را پیاده سازی کنید.
- ✓ لیست پیوندی دو طرفه و حلقوی را تعریف کرده و اعمال درج ، حذف و پیمایش در آنها را پیاده سازی کنید
- ✓ پشته و صف را توسط لیست پیوندی پیاده سازی کنید..
- ✓ آرایه را با لیست پیوندی مقایسه کنید.

سوالهای پیش از درس

۱- به نظر شما با توجه به آرایه ، صف و پشته آیا لزوم تعریف یک ساختار داده جدید ضروری بنظر می رسد؟

....

....

۲- واگن های یک قطار که به هم وصل هستند مثالی از لیست می باشد. مثالهایی از لیست ها را در دنیای واقعی نام ببرید.

....

....

۳- در مثال واگن های یک قطار، هر واگن راهنمایی کننده واگن بعدی می باشد.آیا می توانید ساختار دادهای طراحی کنید که یک همچنین خصوصیتی داشته باشد؟

....

....

مقدمه

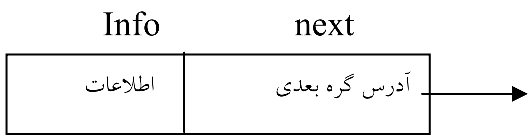
استفاده از اصطلاح «لیست» در زندگی روزمره به یک مجموعه خطی از اقلام داده‌ای مربوط می‌شود. لیست دارای عنصر اول، عنصر دوم و... و عضو آخر می‌باشد. اغلب از ما خواسته می‌شود یک عنصر را به لیست اضافه کنیم یا آن را از لیست حذف کنیم.

داده‌پردازی شامل ذخیره و پردازش داده‌هایی است که در لیست‌ها سازمان‌دهی شده‌اند. استفاده از آرایه یک روش ذخیره چنین داده‌هایی است که در فصل ۲ مورد بحث و بررسی قرار گرفت. آرایه‌ها دارای معایبی بودند. به عنوان مثال اضافه کردن و حذف عناصر در آرایه نسبتاً پرهزینه است. علاوه بر این از آنجایی که هر آرایه معمولاً یک بلاک از فضای حافظه را اشغال می‌کند از این رو هنگام نیاز به حافظه اضافی به راحتی نمی‌توان اندازه یک آرایه را دو برابر یا سه برابر نمود. به همین دلیل به آرایه‌ها، لیست‌های فشرده یا متراکم می‌گویند. علاوه بر این به آرایه‌ها ساختمان داده ایستا نیز گفته می‌شود.

راه دیگر ذخیره یک لیست در حافظه آن است که هر عنصر را در یک لیست، که شامل یک فیلد اطلاعات و آدرس عنصر بعدی در لیست است و این عنصر آدرس بعدی، پیوند یا اشاره‌گر نامیده می‌شود، قرار دهیم. بدین ترتیب لازم نیست عناصر متوالی داخل لیست فضای مجاور در حافظه را اشغال کنند. این کار باعث می‌شود اضافه کردن و حذف عناصر لیست به راحتی انجام شود. این ساختمان داده لیست پیوندی نام دارد.

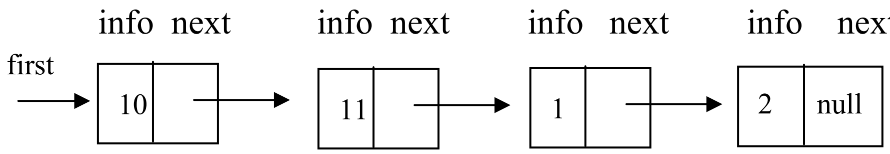
۵٫۱ لیست‌های پیوندی

لیست پیوندی همانند پشته و صف از مجموعه‌ای از عناصر تشکیل شده است. به هر یک از عناصر لیست یک گره (node) گفته می‌شود. هر گره شامل دو فیلد است: فیلد اطلاعات و فیلد آدرس گره بعدی.



یک گره لیست پیوندی

فیلد اطلاعات داده‌ها را ذخیره می‌کند و فیلد آدرس، حاوی آدرس گره بعدی است. چون هر گره لیست پیوندی آدرس گره بعدی را دارد، لازم نیست عناصر لیست در حافظه در کنار هم قرار گیرند. چون هر گره عنصر بعدی خود را مشخص می کند. فیلد آدرس را اشاره‌گر نیز می‌گویند. زیرا به گره بعدی اشاره می‌کند. برای دسترسی به عناصر لیست پیوندی، از یک اشاره‌گر خارجی مانند first استفاده می‌شود که به اولین گره لیست اشاره می‌کند. این اشاره‌گر حاوی آدرس اولیه گره لیست است. برای تشخیص انتهای لیست، فیلد آدرس آخرین گره لیست برابر با تهی (null) می‌گیرند.



نمونه‌ای از لیست پیوندی خطی

لیست فاقد گره را لیست خالی یا لیست تهی می‌گویند. مقدار اشاره‌گر خارجی که به چنین لیستی اشاره می‌کند یک اشاره‌گر تهی است. اگر اشاره‌گر خارجی ما first باشد برای بدست آوردن یک لیست خالی کافی است از عمل first = null استفاده گردد.

لیست پیوندی یک ساختار داده پویاست. تعداد گره‌های لیست دائماً با درج و حذف عناصر تغییر می‌کند. طبیعت پویای لیست با طبیعت ایستای آرایه که طول آن ثابت باقی می‌ماند، مغایرت دارد.

فرق آرایه با لیست پیوندی
۱- لیست پیوندی یک ساختار داده پویاست، تعداد گره‌های لیست دائماً با درج و حذف عناصر تغییر می‌کند. اما طول آرایه همیشه ثابت باقی می ماند.

۲- طول آرایه ابتدای برنامه تعریف می شود و بر اساس تعریف یک تعداد از خانه های حافظه به طور پیوسته به آن تخصیص می یابد. اما طول لیست پیوندی بر اساس نیاز می تواند کم یا زیاد شود.

۵,۲ پیاده‌سازی لیست پیوندی

در زبان C برای پیاده‌سازی لیست پیوندی، از اشاره‌گرها استفاده می‌شود. برای پیاده‌سازی لیست پیوندی، به ابزارهای زیر نیاز داریم:

ابزارهای مورد نیاز برای پیاده سازی لیست پیوندی
۱- ابزارهایی برای تقسیم کردن حافظه به گره‌هایی که شامل فیلد آدرس و فیلد اطلاعات می‌باشند. ۲- عملیاتی برای دستیابی به مقادیر ذخیره شده در هر گره ۳- ابزارهایی برای آزادسازی و نگهداری گره‌هایی که از لیست حذف می‌شوند.

تعریف یک گره

گره لیست پیوندی را می‌توان یک struct به صورت زیر تعریف کرد که دارای یک فیلد داده و فیلد آدرس باشد.

تعریف یک گره لیست پیوندی

```
Struct  node
          {
           int  info;
           Node * next;
          };
```

تعریف اشاره‌گر خارجی

برای تعریف اشاره‌گر خارجی، اشاره‌گرهایی از نوع گره لیست تعریف می‌کنیم

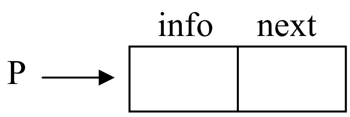
```
node  *p ;
```

بدست آوردن یک گره جدید و خالی از حافظه

در لیست پیوندی، گره‌های لیست در زمان اجرا می‌تواننـد ایجـاد شـوند. بـرای بدسـت آوردن یک گره جدید از تابع getnode ( ) بـه صـورت p = getnode ( ) اسـتفاده می‌کنیم. که این عمل یک گره خالی را ایجاد مـی‌کنـد و آدرس آن را در متغیـر p قـرار می‌دهد. اکنون p به گره جدید اشاره می‌کنـد. خـود تـابع getnode ( ) مـی‌توانـد بـه صورت زیر پیاده‌سازی گردد.

پیاده سازی تابع getnode ( ) : بدست آوردن یک گره جدید از حافظه
node * first; first = (node *) motloc (Size of (struct node));

دستور اول اشاره‌گر p را از نوع node تعریف می‌کند و دستور دوم حافظه‌ای به انـدازه ساختمان node از سیستم می‌گیرد و آدرس آن را در p قرار می‌دهد.



مراجعه به گره‌های لیست

اگر first یک اشاره‌گر خارجی به گره‌ای از لیست باشد، برای مراجعه بـه فیلـد آدرس گره از $first \rightarrow next$ و برای مراجعه به فیلد اطلاعـات از $first \rightarrow info$ اسـتفاده می‌کنیم.

شروع لیست

فرض می‌کنیم اشاره‌گری به نام first به ابتدای لیست اشاره می‌کند. اگر p نیـز بخواهـد به گره اول لیست اشاره کند با عمل p = first ، p نیز به گره اول اشاره خواهد کرد.

برگرداندن گره به مخزن حافظه

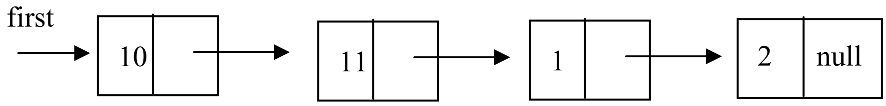
اگر به گره‌ای از لیست پیوندی نیاز نداشته باشیم، آن را به مخزن حافظت برمی‌گـردانیم. برای این منظور از تابع free () استفاده می‌شود. به عنوان مثال، دستور زیر حافظه‌ای را که p به آن اشاره می‌کند، به مخزن حافظه برمی‌گرداند.

Free (p)

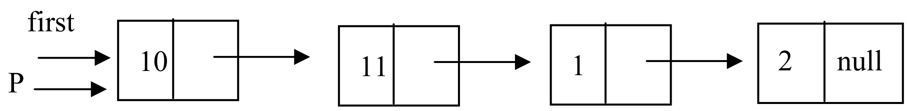
پیمایش لیست

منظور از پیمایش لیست این است که به تمام عناصر لیست دستیابی داشـته باشـیم و در صورت لزوم بتوانیم آنها را پردازش کنیم. برای عمـل پیمـایش لیـست بایـد بـه غیـر از اشاره‌گر first که به ابتدای لیست اشاره می‌کند باید اشاره‌گر دیگری مانند p را با عمـل p = first تعریف کنیم تا آن نیز به اول لیست اشاره کند.اگر ایـن کـار را نکنـیم در آن صورت با حرکت first ابتدای لیست پیوندی را از دست خواهیم داد.

فرض کنید می‌خواهیم لیست زیر را پیمایش کنیم:



با دستور p = first ، اشاره‌گر p نیز به ابتدای لیست اشاره خواهد کرد. با اشـاره‌گـر p می‌توانیم به تمام گره‌ها دستیابی داشته باشیم.

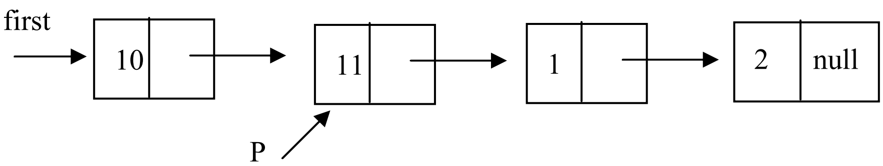


یعنی گره محتوی (11)

برای دستیابی به گره‌ای با محتوای 11 باید پیوندها را دنبـال کنـیم. P بایـد بـه گـره‌ای اشاره کند که آدرس آن در فیلد آدرس گره با محتویات 10 قرار داد. بنابراین

$p = p \to next$

چون p در ابتدا به گرهی با محتویات 10 اشاره می‌کند بـا $p \to next$ ، p بـه گـره بـا محتوای 11 اشاره خواهد کرد.



برای پیمایش گرهی با محتویات 1 باید روند قبلی را تکرار کنیم یعنی

$p = p \to next$

اکنون برای پردازش محتوای خانه‌ای که p به آن اشـاره مـی‌کنـد از عمـل $p \to info$ استفاده می‌کنیم.

با توجه به آنچه گفته شد پیمایش لیست می‌تواند به صورت زیر پیاده‌سازی شود:

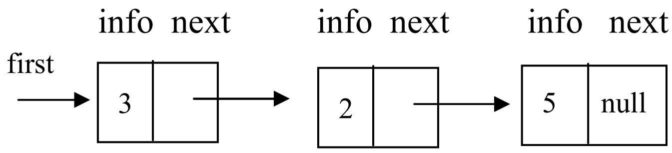
پیاده سازی پیمایش لیست پیوندی

```
p =first
while (p) = null)
    {
        process (p→info)   //   پردازش گره
        p = p →next   //   گره بعدی
      }
```

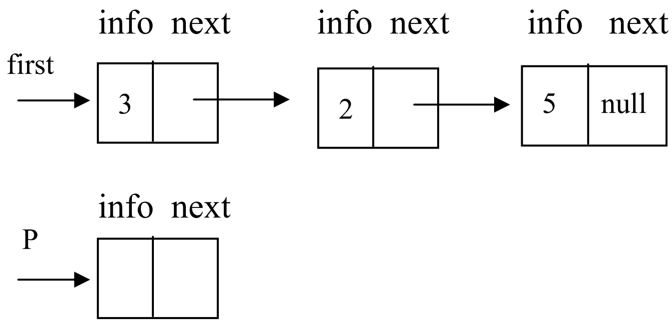
۵٫۳ درج و حذف گره‌ها از لیست پیوندی

الف)درج گره به ابتدای لیست

فرض می‌کنیم لیست پیوندی اولیه زیر را داشته باشیم.



ابتدا با استفاده از عمل ( ) getnode یک گره خالی را بدست می‌آوریم و آدرس آن را در متغیر p قرار می‌دهیم یعنی شکل ۵٫۱ (ب) وضعیت لیست را پس از بدست آوردن گره جدید نشان می‌دهد.

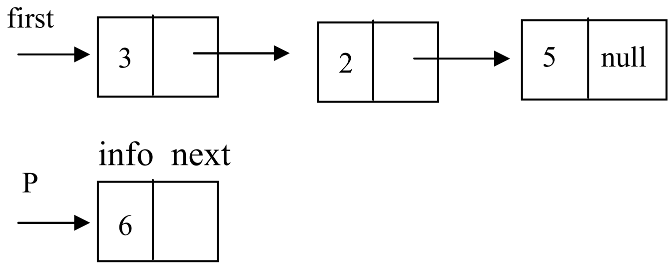


شکل ۵٫۱ (ب) بدست آوردن گره جدید

در مرحله بعدی مقدار 6 در قسمت اطلاعات گره جدید درج می‌شود. ایـن مرحلـه بـا عمل

$$p \to \inf o = 6$$

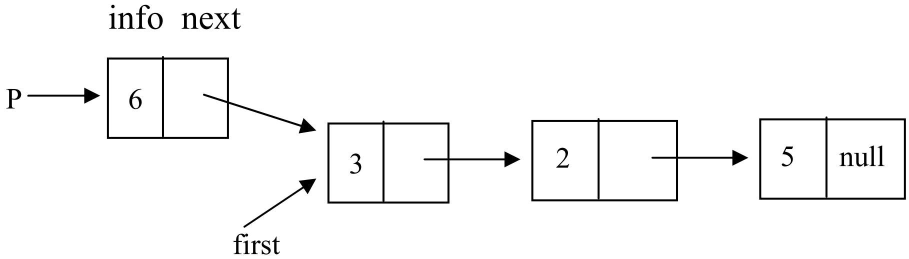
انجام می‌گیرد. شکل ۵,۱(ج)



شکل ۵,۱ (ج) درج اطلاعات در گره جدید

پس از مقدار گرفتن قسمت اطلاعات گره اکنون قـسمت آدرس گـره جدیـد نیـز بایـد مقدار بگیرد، چون گره جدید باید به ابتدای لیست اضافه شود. عنـصری کـه در ابتـدای لیست قرار دارد عمصر بعد از عنصر جدید خواهد بود.

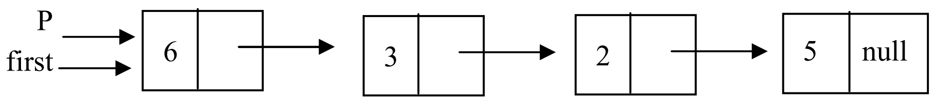
با اجرای عمل $p \to next = first$ ، مقـدار first (کـه بـه آدرس اولـین گـره لیـست می‌باشد) در قسمت آدرس گره جدید قرار می‌دهد. (شکل د)



شکل ۵,۱ (ج) اتصال گره جدید به لیست پیوندی

چون اشاره‌گر first باید به ابتدای لیست اشاره کند، مقدار آن باید طوری تغییر کند که دوباره به ابتدای لیست حاصل اشاره کند. این کار با اجرای عمل زیر انجام می‌گیرد.

$$first = p$$



بنابراین الگوریتم افزودن عدد 6 به ابتدای لیست به صورت زیر خلاصه می‌شود:

$$p = getnode\,(\,);$$
$$p \rightarrow next = 6;$$
$$p \rightarrow next = first;$$
$$first = p;$$

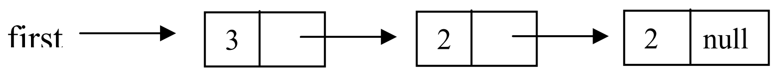
ب)حذف اولین گره از لیست

شکل ۵،۲ مراحل حذف اولین گره از لیست غیرتهی و قرار دادن مقدار آن در متغیر x را نشان می‌دهد. عمل حذف گره دقیقاً عکس عمل افزودن یک گره به ابتدای لیست پیوندی می‌باشد.

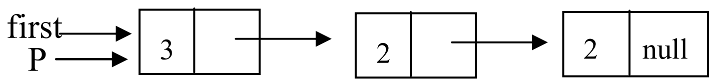
(ب) $p = first$

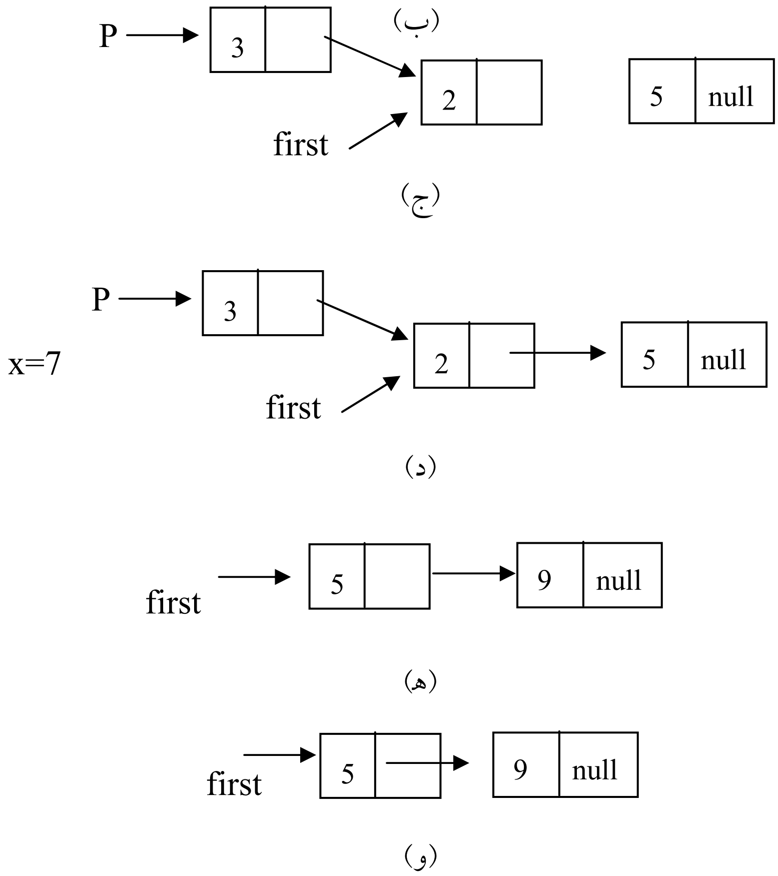
(ج) $first = p \rightarrow next$

(د) $x = p \rightarrow info$



(الف)



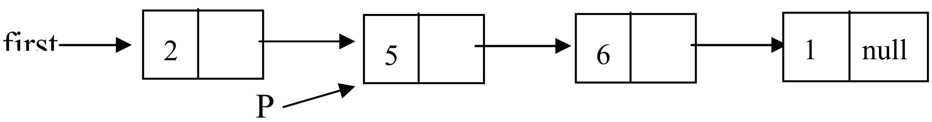


شکل ۵,۲

در (ه) جایی که p به آن اشاره می‌کند آدرس آن در فیلد آدرس هیچ گرهی وجود ندارد. بنابراین دسترسی به آن گره غیرممکن خواهد بود. در این صورت با این که حافظه‌ای در اختیار این گره است ولی عملاً بلااستفاده می‌ماند. بنابراین باید مکانیزمی وجود داشته باشد که این خانه بلااستفاده را برای کاربردهای بعد آزاد نماید. برای این منظور از عمل freenode (p) استفاده می‌شود.

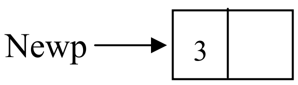
پس ( ) getnode گره جدیدی را ایجاد می‌کند و ( ) freenode آن را از بین می‌برد.
با این دید می‌توان گفت که گره‌ها مورد استفاده مجدد قرار نمی‌گیرند، بلکه ایجاد شـده و از بین می‌روند.

مثال ۵٫۱ : فرض کنید می‌خواهیم گره‌ای با محتویات 3 را بعد از گره‌ای بـا محتویـات 5 به لیست پیوندی اضافه کنیم. برای این کار ابتدا فرض می‌کنیم اشاره‌گری به نام p را آن قدر به طرف جلو حرکت دهیم تا به گرهی که محتویات آن 5 است اشاره کند. بعـد از انجام این کار مراحل زیر را انجام می‌دهیم.



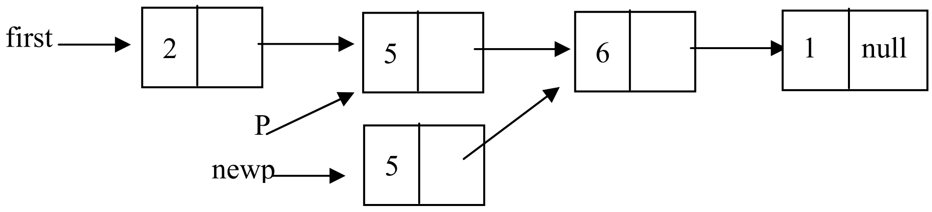
۱–ابتدا گره جدیدی را ایجاد می‌کنیم و آدرس آن را در New p قرار مـی‌دهـیم و نیـز بخش داده آن را برابر 3 قرار می‌دهیم.

$$\text{New p} = \text{getnode ( )}$$
$$\text{New p} \rightarrow \text{inf o} = 3$$



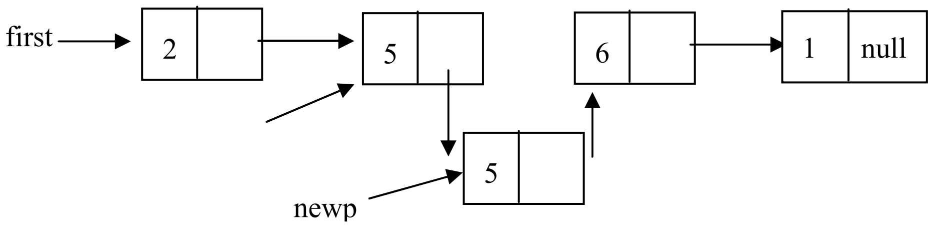
۲–فیلد آدرس گره‌ای را که New p به آن اشاره می‌کند، برابر با بخـش آدرس گـره‌ای قرار دهید که p به آن اشاره می‌کند. یعنی:

$$\text{New p} \rightarrow \text{next} = \text{p} \rightarrow \text{next};$$



۳–فیلد آدرس گره‌ای را که p به آن اشاره می‌کند برابر با New p قرار دهید. یعنی

$p \rightarrow next = new\,p;$



(د)عمل حذف گره از لیست پیوندی

در عمل حذف باید دو حالت را در نظر گرفت:

حذف گره از ابتدای لیست (که قبلاً به آن اشاره شد)

حذف گره‌ای که قبل از آن گره دیگری وجود دارد.

(ج)عمل درج هر لیست پیوندی

برای درج مقدار جدیدی در لیست پیوندی، ابتدا باید گره جدیدی ایجاد کنیم و سپس آن مقدار را در فیلد اطلاعات آن ذخیره کنیم. با استفاده از دستور getnode () گره جدیدی را ایجاد می‌کنیم و آدرس آن را در اشاره‌گری به نام New p قرار می‌دهیم:

New p = getnode ()

پس از این که گره جدید ایجاد شد، باید آن را در لیست درج کنیم. درج گره در لیست دو حالت دارد که باید از هم تفکیک شود.

درج در ابتدای لیست (قبلاً به آن اشاره شد)

درج گره جدید بعدی از گره‌ای در لیست.

فرض کنید می‌خواهیم گره جدید با محتویات x را بعد از گرهی با محتویات y به لیست پیوندی اضافه کنیم. برای این کار ابتدا اشاره‌گری به نام p باید به خانه‌ای که محتویات آن y است اشاره کند و برای درج گره جدید با محتویات x مراحل زیر را انجام می‌دهیم.

۱–گره جدید را ایجاد کـرده، آدرس آن را در Newp قـرار دهیـد و بخـش داده آن را برابر با x قرار دهید؛

$$\text{New}\,p = \text{getnode}()$$

$$\text{New}\,p \to \inf o = x$$

۲–فیلد آدرس گره‌ای را که New p به آن اشاره می‌کند برابر بـا بخـش آدرس گرهـی قرار دهید که p به آن اشاره می‌کند. یعنی

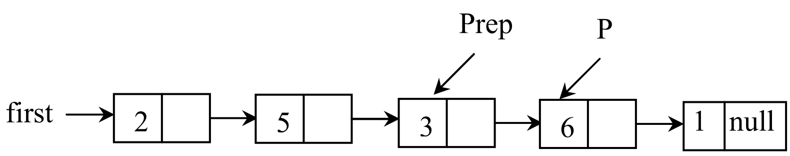
$$\text{New}\,p \to \text{next} = p \to \text{vext}$$

۳–فیلد آدرس گره‌ای را که P به آن اشاره می‌کند برابر با New p قرار دهید.

$$p \to \text{next} = \text{New}\,p;$$

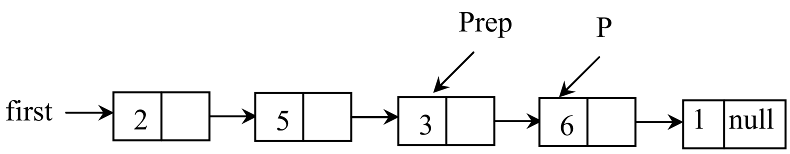
حالت حذف گره‌ای که قبل از آن گره دیگری را با استفاده از مثالی توضیح می‌دهیم.

در لیست پیوندی زیرفرض کنید می‌خواهیم گرهی با محتویات 6 را از لیـست پیونـدی حذف کنیم. فرض می‌کنیم اشاره‌گر p به گره‌ای با محتویات 6 (گرهی که بایـد حـذف شود) و pre P به گره‌ای با محتویات ۳ (گره قبل از گره‌ای که باید حذف شود) اشاره می‌کند. برای حذف هر گرهی با محتویات ٦ مراحل زیر را انجام دهید.



۱–فیلد آدرس per P را برابر با فیلد آدرس گره p قرار دهید:

$$\text{pre}\,P \to \text{next} = p \to \text{next}$$



۲–گره‌ای را که p به آن اشاره می‌کند به مخزن حافظه برگردانید.

```
freenode (p);
```

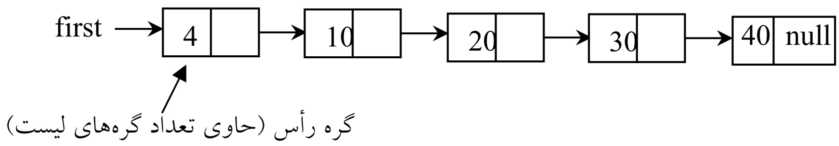
گره رأس حاوی تعداد کارآکترهای لیست و اشاره‌گر به لیست node است.

۵,٤ ساختارهای دیگری از لیست پیوندی

لیست پیوندی که تا کنون بررسی شد، دارای این ویژگی بود که هر گره آن حاوی فیلد داده و فیلد آدرس بود. شکل‌های دیگری از لیست پیوندی وجود دارد که در این بخش مورد بحث و بررسی قرار می‌گیرد. این لیست‌ها عبارت‌اند از: لیست‌هایی با گره رأس و گره انتهایی، لیست‌های حلقوی و لیست‌های پیوندی.

۵,٤,۱ لیست‌هایی با گره رأس

گاهی ممکن است یک گره اضافی که عضوی از لیست پیوندی محسوب نمی‌شود در ابتدای لیست قرار گیرد. این گره را گره رأس می‌نامند. فیلد اطلاعات این گره معمولاً برای نگهداری اطلاعات کلی در مورد لیست بکار می‌رود. شکل یک لیست پیوندی با گره رأس را نشان می‌دهد که محتویات گره رأس برابر تعداد کل گره‌های لیست می‌باشد. در چنین ساختمانی عمل درج و حذف مستلزم کار بیشتری است، زیرا اطلاعات موجود در گره رأس باید تغییر کند. اما تعداد گره‌های لیست را می‌توان از رأس بدست آورد و نیازی به پیمایش نیست.



پیاده‌سازی گره رأس

گره رأس می‌تواند حاوی اطلاعات عمومی در مورد لیست باشد، مثل طول لیست، اشاره‌گر به گره فعلی یا اشاره‌گر به گره آخر لیست.

پیاده‌سازی گره رأس

```
struct   node    {
                 char  info;
                 node  next;
                  };
Struct  char str  {
                 int   length;
                 node  *first char;
                           };
char    str   S1,S2;
```

گره رأس حاوی تعداد کارآکترهای لیست و اشاره‌گر به لیست node است.

۵,٤,۲ مزایای لیست با گره رأس و انتهایی

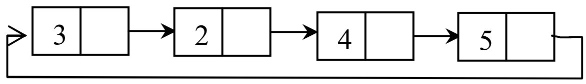
همان گونه که در درج و حذف عناصر به لیست پیوندی مشاهده کردید، باید دو حالت در نظر گرفته شود. یک حالت برای درج و حذف از ابتدای لیست و حالت دیگر برای درج گره‌ای بعد از یک گره و حذف گره‌ای که بعد از گره دیگری قرار دارد. گره رأس موجب می‌شود که هر گره لیست دارای یک گره قبلی باشد و در نتیجه برای حذف و درج، لازم نیست دو حالت در نظر گرفته شود.

گره انتهایی گره‌ای است که در آخر هر لیست پیوندی قرار می‌گیرد که هر گره‌ای از لیست دارای یک گره بعدی باشد.

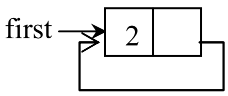
۵,۵ لیست‌های پیوندی حلقوی

اگرچه لیست پیوندی خطی ساختار داده مفیدی است ولی چندین عیب دارد. در لیست پیوندی خطی اگر اشاره‌گر p به گرهی از لیست اشاره نماید نمی‌توان به گره‌های قبلی

آن دسترسی داشت. باید اشاره‌گر خارجی لیست ذخیره شود تا بتوان مجدداً به لیست مراجعه کرد. لیست پیوندی حلقوی مشابه لیست یک طرفه می‌باشد با این تفاوت که فیلد آدرس آخرین گره به جای آن که null باشد به اولی لیست (سر لیست) اشاره می‌کند. در لیست یک طرفه همواره باید برای پیمایش لیست آدرس اولین گره یا سر لیست را داشته باشیم. ولی در لیست پیوندی حلقوی با داشتن آدرس هر گره دلخواه می‌توان به تمام گره‌ها دسترسی داشت. نمونه‌ای از یک لیست پیوندی حلقوی در شکل نمایش داده شده است.

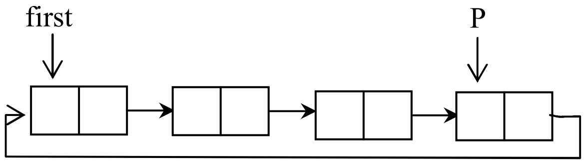


عمل درج در لیست خالی، یک حالت خاص است زیرا در این حالت در لیست یک گره باید به خودش اشاره کند.



الگوریتم درج در لیست پیوندی پس از گره خاصی همانند لیست پیوندی یک طرفه است. در هنگام ساخت لیست پیوندی خطی، علاوه بر اشاره‌گر ابتدای لیست، اشاره‌گر دیگری به انتهای لیست اشاره می‌کند تا گره‌ها پس از اشاره‌گر انتهای لیست اضافه شوند. البته این ابتدا و انتهای طبیعی نیست، بلکه باید ابتدا و انتهای آن را با قرارداد مشخص نمود. ساده‌ترین قرارداد این است که اشاره‌گر خارجی لیست حلقوی به آخرین گره اشاره می‌کند و گره بعد از آن، اولین گره لیست باشد (شکل۵,۳). اگر p یک اشاره‌گر خارجی به لیست حلقوی باشد و $p \rightarrow \inf o$ به آخرین گره لیست و

$p \to next$ به اولین گره لیست مراجعه می‌کند. این قرارداد موجب می‌شـود تـا عمـل حذف و اضافه به ابتدا یا انتهای لیست به سهولت انجام گیرد.



شکل ۵,۳

اگر فرض کنیم end p به انتهای لیست حلقوی اشاره می‌کند اضافه کردن گره در ابتـدا یا انتهای لیست حلقوی به صورت زیر است:

اضافه کردن گره در ابتدا یا انتهای لیست حلقوی

```
New p = getnode ()
New p →info = item;
  if (end p = = null) / * لیست خالی است */
        {
          end p = New p;
          New p →next = New p /* گره به خودش اشاره می‌کند */
        }
else
        {
           New p →next = end p →next;
           end p →=new p;
        }
```

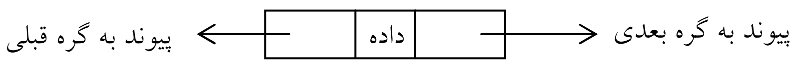
first باید همواره به ابتدای لیست اشاره کند و چون گره جدید New p به اول لیست اضافه شده است

first = new p ;

آن را در first قرار می‌دهیم تا دوباره first به اول لیست اشاره کند.

۵,۶ لیست پیوندی دوطرفه

در لیست‌هایی که تا کنون بررسی کردیم می‌توانستیم از گره‌ای به گره بعدی برویم. به این لیست‌ها یک پیوندی نام نهادیم. در بسیاری از کاربردها لازم است به گره قبلیِ گره‌ای دستیابی داشته باشیم. این کار در لیست‌های یک پیوندی مستلزم جست‌وجو از ابتدای لیست است. شکل دیگری از لیست پیوندی به نام لیست دوپیوندی وجود دارد که هر گره آن دو پیوند دارد. یکی از پیوندها به گره بعدی و پیوند دیگر به گره قبلی اشاره می‌کند. شکل زیر نمایشگر لیست دوپیوندی می‌باشد:



در این لیست به کمک اشاره‌گرهای سمت راست (RLink) و سمت چپ (LLink) می‌توان در هر دو طرف لیست حرکت کرد. بنابراین با داشتن آدرس یک گره، کلیه گره‌ها قابل دستیابی هستند.

ساختار گره لیست دوپیوندی

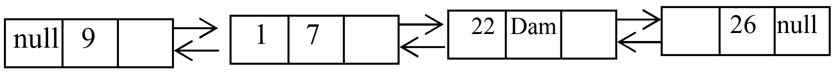
گره لیست دوپیوندی در نمایش پیوندی به صورت زیر خواهد بود:

ساختار گره لیست دوپیوندی

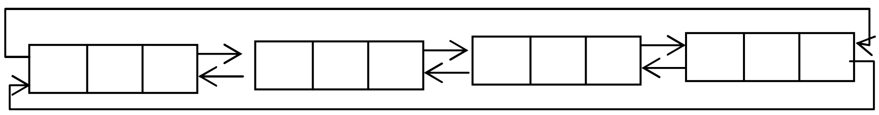
```
struct   node
         {
           node * left;
```

```
int  info;
node  *  right;
};
```

لیست‌های دوپیوندی می‌توانند به شکل‌های گوناگونی باشند. لیست دوپیونـدی عـادی، دوپیوندی حلقوی (شکل ٥,٤)



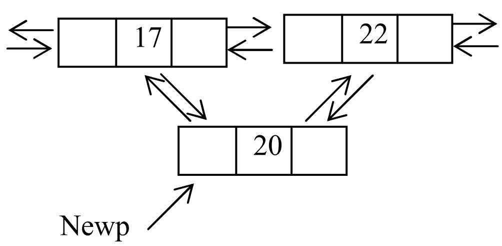
شکل ٥,٤ : (الف) لیست دوپیوندی عادی



شکل ٥,٤ : (ب) لیست دوپیوندی حلقوی

الگوریتم‌های عملیات اصلی در لیست دوپیونـدی، همـان‌هـایی هـستند کـه در لیـست پیوندی مطرح شدند و تفاوت آنها در تنظیم پیوندهای بیشتر است. به عنوان مثـال، درج گره جدید در لیست دوپیوندی شامل تنظیم پیوندهای رو به جلو و رو بـه عقـب اسـت تا به گره‌های قبل و بعد اشاره کنند. سپس باید پیوند رو به جلوی گره قبلی و پیوند رو به عقب گره بعدی را طوری تنظیم کرد که به گره جدید اشاره کنند.

با توجه به لیست شکل ٥,٥(الـف) درج گـره‌ای بـا محتویـات 20 پـس از گـره‌ای بـا محتویات 17 به صورت زیر خواهد بود.



شکل ۵,۵ درج گره جدید در لیست پیوندی دوطرفه

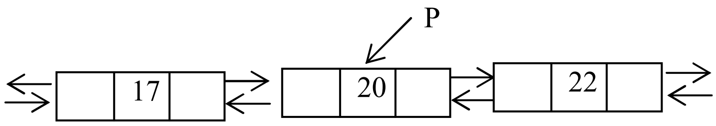
$\text{New p} \rightarrow \text{left} = \text{pre P};$

$\text{New p} \rightarrow \text{right} = \text{pre P} \rightarrow \text{right};$

$\text{per P} \rightarrow \text{right} \rightarrow \text{left} = \text{new p};$

$\text{per P} \rightarrow \text{right} = \text{new p};$

برای حذف گره از لیست دوپیوندی، پیوند رو به جلوی گره قبل و پیونـد رو بـه عقـب بعد از آن باید طوری تنظیم شوند که به این گره اشـاره نکننـد. بـه عنـوان مثـال، بـرای حذف گره‌ای با محتویات 20 به صورت زیر عمل می‌شود:



$p \rightarrow \text{right} \rightarrow \text{left} = p \rightarrow \text{left};$

$p \rightarrow \text{left} \rightarrow \text{right} = p \rightarrow \text{right};$

$\text{free (p)};$

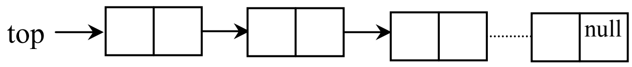
۵,۷ پیاده‌سازی پشته با لیست پیوندی

یکی از معایب پیاده‌سازی پشته و صف با استفاده از آرایه ایـن اسـت کـه انـدازه ثابـت آرایه، اندازه پشته و صف را محدود می‌کند. استفاده از لیـست پیونـدی بـه جـای آرایـه جهت نمایش پشته یا صف، بدون محدودیت رشد می‌کند و بدون به هدر دادن حافظـه کوچک می‌شود.

یکی از معایب پیاده‌سازی پشته و صف با استفاده از آرایه این است که اندازه ثابت آرایه، اندازه پشته و صف را محدود می‌کند.

همان طور که اشاره گردید پشته لیستی از عناصر است که فقـط از یـک طـرف بـه نـام بالای پشته قابل دستیابی‌اند. بنابراین عمل افزودن یک عنصر به ابتدای لیست پیوندی به

مانند این است که عنصری به بالای پشته افزوده شده است. در هر مورد، عنـصر جدیـد طوری اضافه می‌شود که تنها عنصری باشد که فوراً قابل دسترسی اسـت. پـشته فقـط از طریق عنصر بالای آن (top) و لیست پیوندی فقط از طریق اشاره‌گـری کـه بـه ابتـدای لیست اشاره می‌کند، قابل دسترسی است. به طور مشابه عمل حذف اولین عنصر لیـست پیوندی همانند حذف عنصر بالای پشته است. در هر دو مورد، فقط عنـصری کـه فـوراً قابل دسترسی است از مجموعه حذف می شود. شکل ٥,٦ یک پشته پیوندی را نمـایش می‌دهد.



شکل ٥,٦ پشته پیوندی

ساختار گره پشته پیوندی همانند گره لیست پیوندی است. برای دستیابی به عناصر پشته پیوندی فقط به یک اشاره‌گر نیاز داریم که به ابتدای لیست پیوندی اشاره نماید.

پیاده سازی پشته با استفاده از لیست پیوندی

ساختار گره

```
struct node {
        int info;
        struct node * next;
        }s;
node * top;
```

پس از تعریف گره، اشاره‌گر top را برابر با تهی قرار می‌دهند.

```
top = null;
```

.

عمل تست خالی بودن پشته

تست می‌کند که آیا اشاره‌گر top تهی است یا خیر. به عبارت دیگر تست می‌کند که آیا عضو برای حذف کردن وجود دارد یا خیر.

```
if (top = = null)
```

```
printf(" stack is empty");
```

عمل push به پشته

ابتدا با استفاده از دستورات زیر گره‌ای به لیست پیوندی اضافه می‌شود

```
ptr = (struct node *) maltoc (size of (struct node));
ptr→info = item;
ptr→next = top;
top = ptr ;
```

عمل PoP.

اولین گره لیست (گره‌ای که top به آن اشاره می‌کند) را حذف می‌کند.

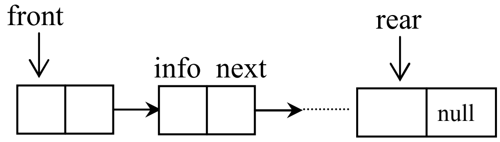
```
ptr = top;
top = top→next;
free (ptr);
```

پشته را می‌توان با استفاده از لیست پیونـدی حلقـوی نیـز پیـاده‌سـازی نمـود. کـه ایـن پیده‌سازی به عنوان تمرین به عهده دانشجو می‌باشد.

۵٫۸ پیاده‌سازی صف با لیست پیوندی

پیاده سازی صف با لیست پیوندی مـشابه پیـاده‌سـازی پیونـدی پـشته اسـت. در صـف پیوندی، اولین گره را جلوی صف front در نظر می‌گیریم. به این ترتیب محـل حـذف گره‌ای از صف پیوندی مشابه حذف گره‌ای از پشته پیوندی پیاده‌سازی مـی‌شـود. ولـی قرار دادن گره‌ای در صف پیوندی مستلزم پیمایش لیست و یافتن آخرین گره آن اسـت. برای جلوگیری از پیمایش لیست جهت یافتن آخرین عنصر آن از یک اشاره‌گر دیگـری به نام rear استفاده می‌کنیم که به انتهای صف پیوندی اشاره می‌کند.

شکل ۵٫۷ نمایش پیوندی صف می‌باشد



شکل ۵,۷ نمایش پیوندی صف

ساختار گره صف پیوندی مانند لیت پیوندی است. به دو اشاره‌گر خارجی بنام های front برای نشان دادن ابتدای صف پیوندی و دیگری بنام rear برای نشان دادن انتهای صف نیاز داریم:

پیاده سازی صف با استفاده از لیست پیوندی

ساختار گره

```
struct queues {
                 int info;
                queue *next;
                    };
          queue front, *rear;
```

پس از تعریف گره، اشاره‌گر های زیر را برابر با تهی قرار می‌دهند.

```
front= rear= null;
```

.

عمل تست خالی بودن صف

تست می کند که آیا اشاره‌گر front تهی است یا خیر

```
if (front = null)
          printf ( "queue is empty");
```

عمل افزودن عنصری به صف

گره‌ای را به انتهای لیست پیوندی اضافه می‌کند. اگر rear به آخرین گره لیست اشاره کند، دستورات زیرگره ptr را به آخر صف پیو.ندی اضافه می‌کند:

```
ptr= (struct queue *) malloc (size of (struct queue));
ptr → info = item;
ptr → next= null;
rear → next = ptr;
rear= ptr;
```

عمل حذف گره از صف

اولین گره صف را حذف می‌کنیم. دستورات زیر این کار را انجام می‌دهد:

```
ptr = front;
front = front → next;
free (ptr);
```

۵,۹ معایب پیاده‌سازی صف و پشته از طریق لیستهای پیوندی عبارتند از:

۱–یک گره در لیست پیوندی نسبت به عنصر متناظر خود در آرایه حافظه بیشتر را اشغال می‌کند، چون در لیست پیوندی علاوه بر قسمت اطلاعات به قسمت آدرس نیز نیاز است اما باید توجه داشت که فضایی که گره اشغال می‌کند معمولاً در برابر فضایی که توسط عنصر آرایه اشغال می‌شود نیست.

۲–مدیریت لیست آماده مستلزم صرف وقت است. هر عمل افزودن و حذف یک عنصر از پشته یا صف مستلزم حذف و اضافه به لیست آماده است.

مزیت نمایش پشته و صف به صورت لیست پیوندی این است که همگی از گره‌های موجود در یک لیست آماده استفاده می‌کنند گره‌ای که توسط پشته استفاده نشده باشد توسط پشته دیگری قابل استفاده است. به طوری که تعداد گره‌های در حال استفاده در یک زمان خاص از حداکثر تعداد گره‌های قابل استفاده بیشتر نیستند.

مثالهای حل شده

۱- با استفاده از لیستهای پیوندی شبه برنامه ای به زبان سی بنویسید که نام و شماره ده نفر را در یک لیست ذخیره کرده و سپس دوباره همه آنها را روی مونیتور نمایش داده و در آخر نامی گرفته و شماره معادل آن را نمایش دهد.

```
main()
{
   x,y,temp: NODE;
   int  i,n;
   char s [20];
   scanf("%d",&n);
   new(n);
   scanf("%s %d", x->name,x->num);
   x->link=nil; temp=x;
   for( i:=2 ; i<=n ; i++)
       {
        new(j);
        scanf("%s %d", x->name,x->num);
        j->link=nil;
        temp->link=j;
        temp=temp->link;
       }
    temp=x;
    while (temp!=nil )
         {
          printf("%s %d" , temp->name,temp->num);
          temp=temp->link;
         }
     scanf("%s ", s);
     temp=x;
```

```
while (temp!=nil)
      {
         if (temp->name==s )
           {
             printf("%s " , temp->name);

           }
           temp=temp->link;
      }
   printf("NOT FOUND");
}
```

۲–برنامه ای بنویسید که یک گره را به لیست پیوندی اضافه کند.

```
new(j);
j->data:=item;
if (head=nil)
   {
     head=j;
     j->link=nil;
    }
else
   {
     j->link=x->link;
    x->link=j;
   }
```

۳–برنامه ای بنویسید که اشاره گر و لیست پیوندی را بگیرد و تعداد گره های لیست را برگرداند.

```
int Node count(Node PTR L)
{Node ptr  g ;
```

```
int count;
if(l==NULL)
   return 0;
else
   g=l;
   while(g)
{
      g= g -> Next;
   count ++;
}
return count;
}
```

٤- برنامه ای بنویسید که با استفاده از لیست پیوندی داده گره نام را برگرداند.

```
int node data(node ptr l, int i, float &x)
{
int j=1, h=0;
h=node count (l);
if((i>0)&&i<=h))
{
    for (p=l;j!=i;j++,p=p->next)
    x=p->data;
    return 1;
}
Return 0;
}
```

٥- برنامه ای بنویسید که داده و لیست پیوندی را از آخر به اول چاپ کند.

```
void print_reverse(NODE PTR L)
{
```

```
If (L!=NULL)
{
   print_reverse(L->next);
   printf("%f",L->data);
}
```

٦- تابعی به نام intersect بنویسید که اشاره گر لیست پیوندی را به عنوان پارامتر گرفته و از داده های مشترک آن دو لیست، لیست سومی ساخته و اشاره گر اول لیست سوم را برگرداند.

```
NODE PTR intersect(NODE PTR L1, NODE PTR L2)
{
NODE PTR T1,T2,L3;
for(T1=L1;T1!=NULL;T1=T1->next)
for(T2=L2;T2!=NULL;T2=T2->next)
if(T1->data=T2->data)
{
   Add_node(L3,T1->data);
   Break;
}
return L3;
}
```

٨- تابعی بنویسید که محتویات لیست یک طرفه را بروش بازگشتی نشان دهد.

```
void travers(list pointer x)
{
if(x!=NULL)
{
```

```
printf("%f",x->data);
travers(x->link);
}
}
```

۱۲- نحوه اضافه کردن گره به لیست پیوندی دوطرفه.

```
new(x);
Llink(R link(j)):=x;
Rlink(x):=Rlink(j)
Rlink(j):=x;
Llink(x):=j;
```

تمرین

۱) توابعی بنویسید که در گره mام و nام یک لیست پیوندی را با هم عوض کند. برنامه‌ای بنویسید که از آن استفاده کند.

۲) الگوریتمهایی برای اعمال زیر بنویسید:

الف) **افزودن عنصر دو انتهای لیست پیوند**

ب) الحاق دو لیست

ج) معکوس کردن لیست به طوری که عنصر آن به عنصر اول... تبدیل شود.

د) ترکیب دو لیست مرتب در یک لیست مرتب دیگر

ه) مجموع عناصر لیست صحیح

و) محاسبه تعداد عناصر لیست

ی) کپی از لیست

۳- تعداد توسط گره‌هایی که در جستجو برای یک عنصر خاص در یک لیست نامرتب، لیست مرتب و یک آرایه نامرتب و یک آرایه مرتب دستیابی می‌شوند چقدر است؟

٤- فرض کنید List یک لیست پیوندی در حافظه و شامل مقادیر عددی باشد.

برای هر یک از حالتهای زیر یک زیربرنامه بنویسید که:

الف) ماکزیمم مقادیر لیست را پیدا کنید.

ب) مینیمم مقادیر لیست را پیدا کنید.

ج) میانگین مقادیر لیست را پیدا کنید.

د) حاصلضرب عناصر لیست را پیدا کنید.

۵- تابعی بنویسید که بدون هیچ تغییری در مقدارهای info لیست را مرتب کند.

٦- پشته ای را به کمک لیست پیوندی پیاده سازی کرده و سپس به کمک آن یک عبارت postfix را محاسبه و چاپ کنید.

۷- برنامه ای بنویسید که دو چندجمله ای یک متغیره را در لیست پیوندی ذخیره کـرده و سپس جمع آنها را محاسبه کرده و چاپ کند.

۸- نشان دهید چگونه می توان کار بر روی هم ارزی ها از لیست پیوندی استفاده کرد.

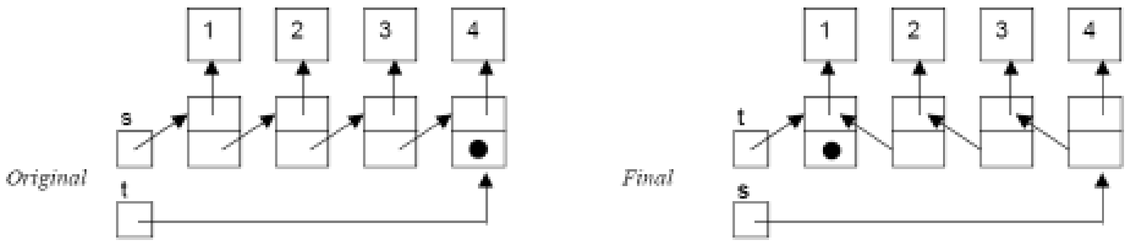
۹- برنامه ای بنویسید که دو لیست حلقوی را به هم الحاق کند.

۱۰- برنامه ای بنویسید که دو لیست حلقوی دو طرفه را به هم الحاق کند.

۱۱- تابعی بنویسید که مقادیر ماکزیمم، مینیمم و متوسط یـک لیـست پیونـدی را پیـدا کند.

۱۲- برنامه ای بنویسید که ۱۰۰ عدد تصادفی را خوانده و در لیست قرار دهد. و سـپس با استفاده از تابع تمرین ۱۱ مقادیر ماکزیمم، مینیمم و متوسط آن را پیدا کند.

۱۳- با توجه به شکل سمت چپ ، مجموعه ای ازدستورات ارائه کنید تا به شکل سمت راست برسیم. در این شکل s و t از نوع اشاره گر به نود هستند.



۱۴-تابعی بنویسید که دو لیست پیوندی مرتب را دریافت کرده آن هـا را طـوری ادغـام کند که حاصل نیز مرتب باشد.

۱۵- برنامه ای بنویسید که اعداد صحیح بزرگ را به صورت رشته ای خوانده و در یک لیست پیوندی قرار دهد . سپس حاصل جمع دو عدد بزرگ را به دست آورد.

فصل ششم

درخت

در پایان این فصل شما باید بتوانید:

- ✓ درخت را تعریف کرده و تمام مفاهیم و تعاریف موجود در درخت را بیان کنید.
- ✓ داده های خود را در قالب درخت نمایش دهید.
- ✓ درخت را در قالب های متفاوت ذخیره کرده و معایب و امتیازات هر یک را بیان کنید.
- ✓ درخت دودوئی را با درخت مقایسه کنید.
- ✓ اطلاعات موجود در درخت را به روش های گوناگون پردازش کنید.
- ✓ انواع کاربردهای درخت را تشریح کنید؟

سوالهای پیش از درس

۱- به نظر شما با توجه به پشته، صف و لیست پیوندی لزوم تعریف یک ساختار داده جدید ضروری بنظر می رسد؟

....

....

۲- به نظر شما وقتی بخواهیم با یک حرکت نصف داده های موجود را کنار بگذاریم چه نوع ساختاری می توانیم تعریف کنیم.

....

....

۳- در حافظه های جانبی کامپیوتر اطلاعات را معمولا به چه صورتی ذخیره می کنند؟

....

....

مقدمه

تا اینجا، انواع مختلف ساختمان داده‌های خطی از قبیل رشته‌ها، آرایه‌ها، لیستها، پشته‌ها و صف‌ها مورد مطالعه و بررسی کامل قرار گرفته است. این فصل یک ساختار داده غیرخطی موسوم به درخت را تعریف می‌کند. این ساختمان اساساً برای نمایش داده‌هایی که شامل رابطه سلسله مراتبی بین عناصر مانند رکوردها، درختهای خانوادگی و جدول فهرست مطالب کتاب است به کار می‌رود.

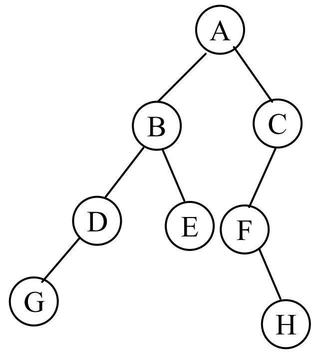
تعریف: درخت مجموعه محدودی از یک یا چند گره به صورت زیر می‌باشد:

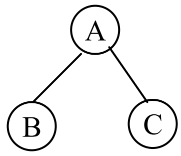
دارای گره خاصی بنام ریشه است.

بقیه گره‌ها به $n \geq 0$ مجموعه مجزا $T_1,...,T_n$ تقسیم شده که هر یک از این مجموعه‌ها خود یک درخت هستند. $T_1,...,T_n$زیردرخت‌های ریشه نامیده می‌شوند.

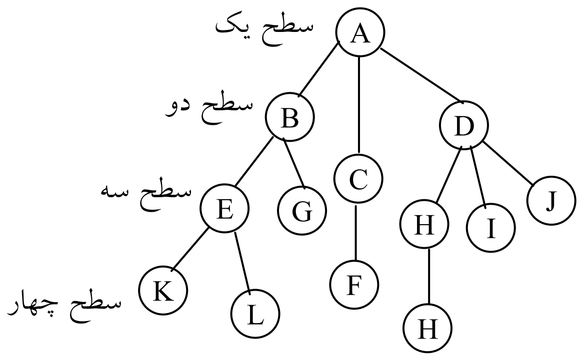
چند درخت کامپیوتری را در شکل ٦,١ مشاهده می‌کنید برخلاف درختان طبیعی که ریشه‌های آن‌ها در پایین و برگ‌ها در بالا قرار دارند. در درختهای کامپیوتری، ریشه در بالا و برگ‌ها در پایین قرار دارند.

درختها به طور کلی بر دو دسته تقسیم می‌شوند. درختهای عمومی و درخت‌های دودوئی. درخت دودوئی (binary tree) درختی است که از هر گره آن حداکثر دو پیوند خارج می‌شود. درختی که دودوئی باشد، درختی عمومی است. در شکل ٦,١(الف) و (ب) درختی دودوئی هستند و (ج) درختی عمومی می‌باشد.





سطح درخت



(ج): درخت عمومی

شکل ٦,١ انواع درختها

اصطلاحات زیادی در ارتباط با مطالعه درخت‌ها به کار برده می‌شود که بایستی آنها را ابتدا تعریف کرد.

٦,١ اصطلاحات مربوط به درخت‌ها

گره: به عناصر موجود در درخت گره گویند. درخت شکل ٦,١ (ج) را در نظر بگیرید. این درخت سیزده گره دارد که داده موجود در هر گره برای سهولت یکی از حروف الفبا در نظر گرفته شده است. A ریشه درخت است معمولاً در بالا قرار می‌گیرد.

درجه گره: درجه گره برابر با تعداد فرزندان است. یا تعداد زیردرختهای یک گره درجه آن گره نامیده می شود. در شکل ٦,١ (ج) درجه A برابر با ٣، درجه C برابر با ١ و درجه F صفر است.

برگ: گره‌هایی که درجه صفر دارند، برگ یا گره‌های پایانی نامیده می‌شوند. برای مثال G,F,L,K و J,I,M,G مجموعه‌ای از گره‌های برگ هستند. سایر گره‌ها عناصر غیرپایانی هستند.

درجه درخت: درجه یک درخت حداکثر درجه گره‌های آن درخت می‌باشد. برای مثال در درخت (ج) حداکثر درجه گره ۲ می‌باشد پس درجه درخت ۳ است.

گره‌های همزاد یا هم‌نیا: گره‌ای که دارای زیردرختانی است، والد ریشه‌های درختان و ریشه‌های زیردرختان، فرزندان آن گره می‌باشند. برای مثال، گروه B والد گره‌های F , E بوده و برعکس F , E فرزندان B می باشند. فرزندان یک گره، گره‌های همزاد یا هم نیا نامیده می‌شوند. برای مثال، گره‌های H , I , J همزادند. می‌توان این نامگذاری را به پدربزرگ و نوه تعمیم داد. مثلاً می‌توانیم بگوئیم که D پدربزرگ H است و A نیز پدربزرگ گره‌های J , I , H , G, F, E می‌باشد. اجداد یک گره، گره‌هایی هستند که در مسیر طی شده از ریشه تا آن گره وجود دارند. برای مثال اجداد M گره‌های H , D, A هستند. برعکس نسل‌های یک گره شامل تمام گره‌هایی است که زیردرخت آن گره قرار دارند. برای مثال، گره‌های L , K , F, E همگی از نسل گره B می‌باشند.

سطح درخت: هر گره موجود در خت دارای سطحی است، سطح گره ریشه، یک در نظر گرفته می‌شود (بعضی از کتاب‌ها سطح گره ریشه را صفر فرض می‌کنند). سطوح بقیه گره‌ها یک واحد بیشتر از گره بالایی است. سطوح درخت شکل ٦,١ (ج) در کنار آن نوشته شده است.

عمق یا ارتفاع درخت: بزرگ‌ترین سطح برگ‌های درخت (طول بزرگ‌ترین مسیر از ریشه به برگها است) را عمق درخت گویند. در شکل ٦,١ (ج) عمق درخت برابر با ٤ است.

طول مسیر درخت: طول مسیر درخت، مجموع طول‌های تمام مسیرها از ریشه به آن است.

۲٤= ۳ × ۳ + ۲ × ٦ + ۱ × ۳

طول مسیر تعداد طول مسیر تعداد طول مسیر تعداد(B,C,D)

(K, L, H) (E,F,G,H,I,J)

<u>**درخت kتائی:**</u> درختی که تعداد فرزندان هر گره در آن حداکثر k باشد.

<u>**درخت متوازن:**</u> درختی که اختلاف سطح برگ‌های آن حداکثر یک باشد و اگر این اختلاف صفر باشد، آنگاه درخت را کاملاً متوازن می‌گویند.

فرض کنید T یک درخت باشد. علاوه بر نمایش ارائه شده در شکل ٦,۱روشهای مختلفی برای رسم یک درخت وجود دارد یکی از این راههای جالب استفاده از لیست یا فرم پرانتزی درخت T می‌باشد. بدین مفهوم که شکل ٦,۱(ج) را می توان به صورت زیر هم نمایش داد. به نحوی که هر زیر درخت خود یک لیست می‌باشد.

(A(B(E(K,L),F),C(G),D(H(M),I,J)))

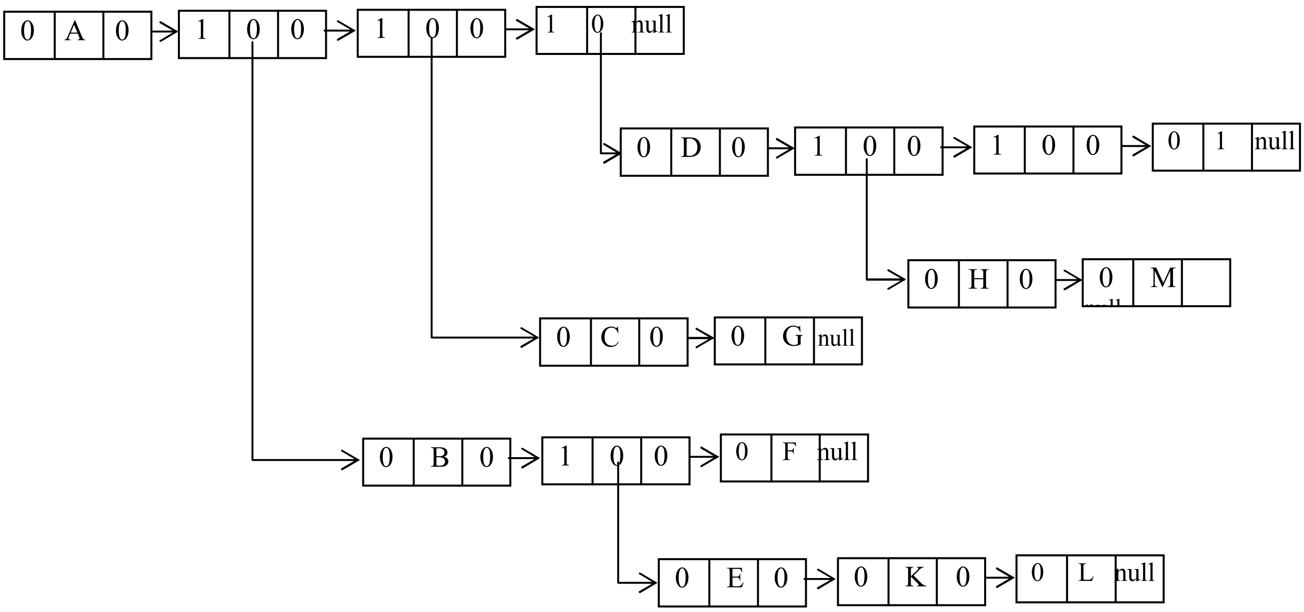
توجه داشته باشید که در این فرم ابتدا اطلاعات ریشه و سپس در داخل پرانتزها اطلاعات فرزندان آن گره به ترتیب از چپ به راست نوشته می‌شود.

روش دیگر نمایش درخت برای رسم یک درخت استفاده از یک لیست پیوندی می‌باشد. در این صورت بایستی یک گره و تعدادی متغیر فیلد، بسته به تعداد انشعاب آن را داشته باشد. و در این گره هر فرزندی که برگ نباشد با اضافه کردن یک سطح به لیست به صورت یک زیرلیست نمایش داده می‌شود.

برای نمایش شکل ٦,۱ (ج) به صورت یک لیست پیوندی داریم:

استفاده از لیست یا فرم پرانتزی ، لیست پیوندی و شکل درختی سه روش نمایش درخت ها می باشند.

(یعنی A، به سه گره وصل است)



شکل٦,٢ نمایش پیوندی شکل ٦,١ (ج)

٦,٢ درخت دودوئی (binary tree)

یک درخت دودوئی T به صورت مجموعه‌ای متناهی از عناصر بنام گره‌ها تعریف می‌شود. به طوری که:

الف)T خالی است که به آن درخت پوچ یا تهی می‌گویند یا

ب)حاوی مجموعه‌ای محدود ازگره‌ها یک ریشه و در زیردرخت T_2,T_1 می‌باشد که به ترتیب به آن آنها زیردرختهای چپ و راست گفته می‌شود.

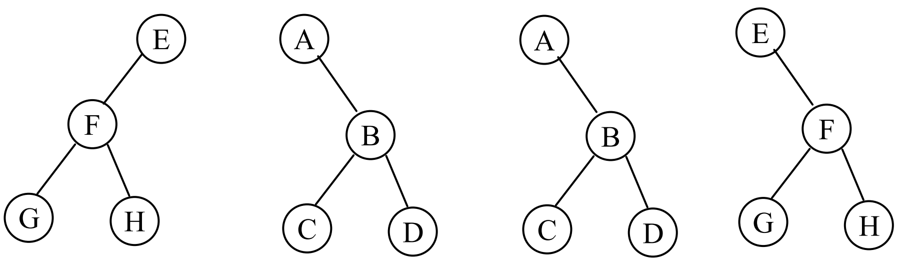
با توجه به تعریف- درخت دودوئی متوجه می‌شویم که درخت دودوئی هر گره حداکثر دو فرزند دارد.

تفاوت میان یک درخت دودوئی و یک درخت عادی جای بحث و توجه کامل دارد. اول از همه درخت عادی تهی یا صفر گره وجود ندارد. اما درخت دودوئی تهی وجود دارد. ثانیاً در یک درخت دودوئی ترتیب فرزندان دارای اهمیت است ولی در درخت عادی ترتیب فرزندان مهم نیست.

درخت عادی تهی یا صفر گره وجود ندارد. اما درخت دودوئی تهی وجود دارد.

درختهای دودوئی T_2, T_1 را مشابه گویند هرگاه دارای یک ساختار باشند؛ به عبارت دیگر این درختها دارای یک شکل باشند، درختها را کپی هم گویند اگر این درختها مشابه بوده و محتوای گره‌های آن یکسان باشد.

(مثال ٦,١) چهار درخت دودوئی شکل ٦,٣ را در نظر بگیرید. سه درخت (الف)، (ب) و (ج) مشابه هم هستند و درختهای (الف) و (ب) کپی هم هستند. درخت (د) نه مشابه (ج) است و نه کپی آن، زیرما در یک درخت دودوئی بین یک گره بعدی چپ و یک گره بعدی راست حتی وقتی تنها یک گره بعدی وجود داشته باشد تفاوت و تمایز قائل می‌شویم. همانگونه که گفتیم دو درخت (ج) و (د) دو درخت دودوئی متفاوتی هستند ولی اگر این دو درخت، عادی فرض شوند، یکسان هستند. چون در درختهای عادی ترتیب زیردرختان مهم نیست.



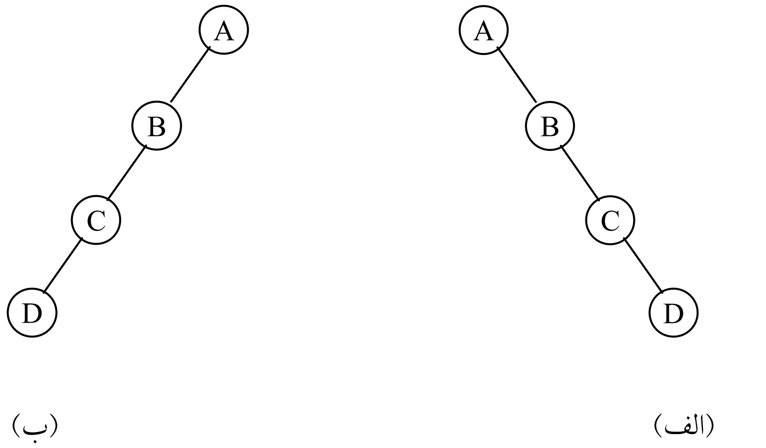
(د) (الف) (ب) (ج)

شکل ٦,٣ چهار درخت دودوئی

٦,٣ انواع درختهای دودوئی

بر این بخش به تعریف انواع درختهای دودوئی می‌پردازیم.

درخت مورب: یک درخت مورب به چپ می‌باشد هر گاه، هر گـره فرزنـد چـپ پـدر خود باشد و یک درخت مورب به راست می‌باشد، هرگاه، هرگـره، فرزنـد راسـت پـدر خود باشد. شکل ٦,٤ (الف) و (ج) دو درخـت مـورب بـه راسـت و چـپ را نمـایش می‌دهند.

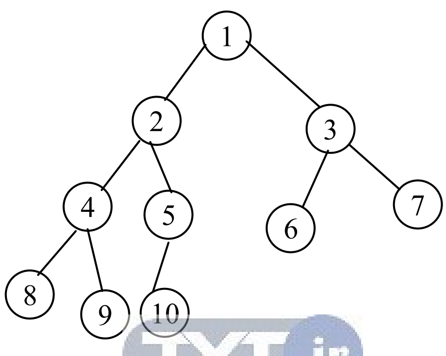


شکل ٦,٤ دو درخت مورب به چپ و راست

درخت دودوئی کامل

درخت دودوئی دلخواه T را در نظر بگیرید. هر گره حداکثر مـی‌توانـد دو بچـه داشـته باشد.

بنابراین می توان نشان داد سطح i از درخت T حـداکثر مـی‌توانـد 2^{i-1} $(i \geq 1)$ گـره داشته باشد. درخت T را کامل گویند اگر تمام سطح‌های آن بـه جـز احتمـالاً آخـرین سطح، حداکثر تعداد گره ممکن را داشته باشد و همچنین تمام گره‌هـای آخـرین سـطح در سمت چپ و در دورترین مکان آن باشد. درخت کامل T_{10} با ١٠ گره در شـکل٦,٥ رسم شده است.



شکل ٦,٥ درخت کامل با ده گره

اگر یک درخت دودوئی کامل با n گره به صورت بالا تعریف شده باشد، آنگاه عمق d_n درخت کامل از رابطه $D_n = [\log_2 n + 1]$ به دست می‌آید. و برای هر گره با اندیس i و $1 \le i \le n$ داریم:

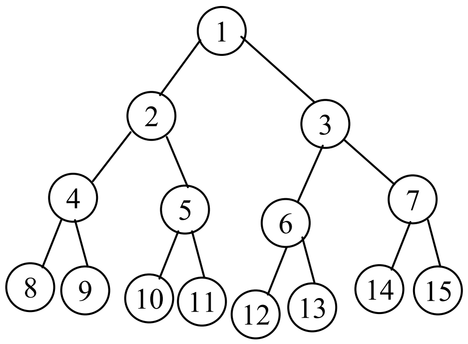
اگر $i \ne 1$ آنگاه پدر i در [i/2] است. اگر i=1 ، i ریشه است و پدری نخواهد داشت.

اگر $2i \le n$ آنگاه فرزند چپ i در 2i است. اگر $2i > n$ آنگاه i فرزند چپ ندارد.

اگر $2i+1 \le n$ آنگاه فرزند راست i در $2i+1$ است. اگر $2i+1 > n$ آنگاه i فرزند راست ندارد.

درخت پر

درختی دودوئی T را درخت پر گویند هرگاه همه گره‌های آن به جز گره‌های سطح آخر دقیقاً دو فرزند داشته باشند. شکل ٦,٦ یک درخت پر دودوئی با ٤ سطح را نمایش می‌دهد.



شکل ٦,٦ درخت پر با چهار سطح

٦,٤ خواص درختهای دودوئی

قبل از اینکه به چگونگی نمایش درختهای دودوئی بپردازیم در این بخش به ارائه برخی از خواص درخت دودوئی می‌پردازیم. مانند اینکه در یک درخت دودوئی با عمق h، حداکثر تعداد گره‌ها چقدر است، همچنین چه ارتباطی بین تعداد گره‌های برگ

و تعداد گره‌های درجه دو در یک درخت دودوئی وجود دارد. ما هر دو موضـوع فـوق را به صورت چند اصل موضوعی ارائه می‌کنیم.

اصل موضوعی (۱) [حداکثر تعداد گره‌ها]

حداکثر تعداد گره‌ها در سطح i ام یک درخت دودوئی برابر با $2^{i-1}(i \geq 1)$ است.

حداکثر تعداد گره‌ها در یک درخت دودوئی به عمق k ، برابر است با:

$$\sum_{i=1}^{k} (\text{حداکثر تعداد گره‌ها در سطح } i) = \sum_{i=1}^{k} 2^{i-1} = 2^k - 1$$

اصل موضوعی (۲) [رابطه بین تعداد گره‌های بزرگ و گره‌های درجه ۲]

برای هر درخت دودوئی غیرتهی مانند T، اگر n_o تعداد گره‌هـای پایـانی و n_2 تعـداد گره‌های درجه ۲ باشد، آنگاه خواهیم داشت:

$$n_o = n_2 + 1 \qquad (۱)$$

(یعنی تعداد برگ‌ها همواره در درخت دودوئی یکی بیشتر از تعداد گـره‌هـای درجـه ۲ می‌باشد).

اثبات: اگر n_1 را تعداد گره‌های درجه ۱ و n را تعداد کل گره‌های درخت فرض کنیم، چون همه گره‌ها در T درجه‌ای کمتر یا مساوی ۲ دارند پس خواهیم داشت:

$$n = n_o + n_1 + n_2$$

و اگر B نشانگر تعداد انشعاب‌های یک درخت دودوئی باشد آنگـاه (۳) $n = B + 1$ خواهد بود و می‌دانیم همه انشعاب‌ها یا از یک گره با درجه یک یا از یک گره با درجـه دو به وجود آمده‌اند. بنابراین:

$$B = n_1 + 2n_2 \qquad (٤)$$

آنگاه با جایگذاری رابطه (٤) در رابطه (۳) خواهیم داشت:

$$n = 1 + n_1 + 2n_2 \qquad (۵)$$

با کم کردن عبارت (۵) از (۲) و ساده نمودن آن خواهیم داشت:

$n_o = n_2 + 1$

۶,۵ نمایش درخت‌های دودوئی در حافظه

فرض کنید T یک درخت دودوئی باشد. این بخش دو روش نمایش T را در حافظه مورد بحث و بررسی قرار می‌دهد. روش اول و معمول به نمایش پیوندی درخت T است و مشابه روش لیست‌های پیوندی است. روش دوم که تنها از یک آرایه استفاده می‌کند نمایش ترتیبی درخت T است. اصلی‌ترین موضوع در هر نمایش درخت T، آن است که به ریشه R درخت T دسترسی مستقیم داشته باشیم و با معلوم بودن هر گره N از T باید بتوان به هر بچه N دسترسی پیدا کرد.

۶,۵,۱ نمایش ترتیبی درختهای دودوئی

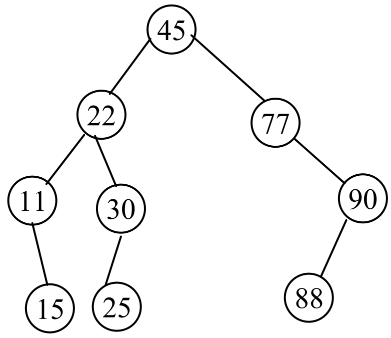
فرض کنید T یک درخت دودوئی باشد که کامل یا تقریباً کامل است. آنگاه روش کاراتری برای نگهداری T در حافظه وجود دارد. که نمایش ترتیبی T نام دارد. این روش نمایش تنها از یک آرایه یک بعدی به نام Tree به صورت زیر استفاده می‌کند:

(از موقعیت صفر آرایه استفاده نمی‌شود)

ابتدا گره‌های درخت دودوئی را از یک شماره‌گذاری می‌کنیم.

محتوای هر گره درخت دودوئی با شماره خاص در خانه‌ای از آرایه با همان شماره ذخیره می‌شود.

نمایش ترتیبی درخت دودوئی T شکل ۶,۷ در ۶,۸ نشان داده شده است.



شکل ٦,٧

١	٢	٣	٤	٥	٦	٧	٨	٩	١٠	١١	١٢	١٣	١٤	١٥
٤٥	٢٢	١٧	١١	٣٠٠		٩٠		١٥	٢٥				٨٨	

شکل ٦,٨ نمایش ترتیبی درخت شکل ٦,٧

استفاده از فرم آرایه برای درختهای کامل یا تقریباً کامل مناسب است چرا که حافظه کمتری به هدر می‌رود و استفاده از آن به دلیل هدر دادن حافظه برای دیگر درختها توصیه نمی‌شود.

امتیازات نمایش درخت دودوئی با آرایه
١- هرگره‌ای از طریق گره دیگری به راحتی و از طریق محاسبه اندیس قابل دستیابی است. ٢- فقط داده‌ها ذخیره می‌شوند و نیازی به ذخیره اشاره‌گرهای زیردرخت چپ و راست نمی‌باشد. ٣- در زبان برنامه‌سازی که فاقد تخصیص حافظه پویا هستند (مثل بیسیک و فرترن)، نمایش آرایه تنها راه ذخیره درخت است.

ولی با وجود امتیازات فوق و به طور کلی، نمایش ترتیبی یک درخت به عمق h به $2^h - 1$ خانه آرایه نیاز دارد که بر طبق آن به این نمایش ترتیبی معمولاً از کارایی لازم برخوردار نیست بخصوص برای درخت های مورب فقط h خانه مورد استفاده قرار می‌گیرد و بقیه خانه‌ها بدون استفاده می‌ماند.

. ٦,٥,٢ نمایش پیوندی درختهای دودوئی

همانطور که در بخش قبل مشاهده کردید، آرایه برای نمایش درخت‌های دودوئی کامل مناسب است، ولی برای نمایش سایر درختهای دودوئی موجب اتلاف حافظه می شود. علاوه بر این، نمایش درخت‌ها به صورت آرایه، مشکلات مربوط به آرایه از جمله، درج و حذف گره‌ها مستلزم جابجایی عناصر است، را دارد. این مشکلات با استفاده از پیاده‌سازی درخت‌ها از طریق لیست پیوندی (استفاده از اشاره‌گرها) را می‌توان برطرف کرد.

ساختار گره درخت دودوئی در نمایش پیوندی

درخت دودوئی T را در نظر بگیرید. درخت T در حافظه به وسیله یک نمایش پیوندی نگهداری می‌شود و هر گره این لیست از سه فیلد Lchild , Rchild , data به صورت زیر تشکیل یافته است:

Lchild	Data	Rchild

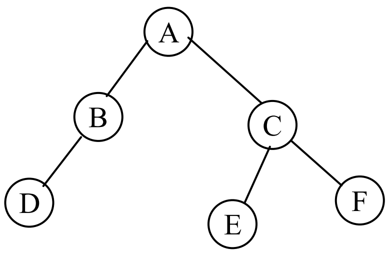
به گونه‌ای که:

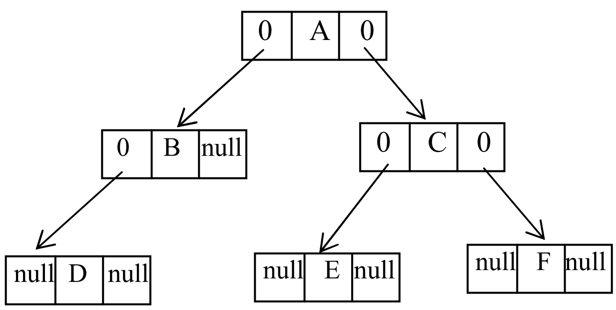
data حاوی داده گره N است.

Rchild حاوی مکان یا اشاره‌گری به فرزند راست گره N است.

Lchild حاوی مکان یا اشاره‌گری به فرزند چپ گره N است.

مثال: نمایش پیوندی درخت شکل ٦,٩(الف) به صورت شکل٦,٩ (ب) در حافظه می‌باشد.





شکل ۶٫۹

همانگونه که گفتیم تعیین پدر یک اصل اساسی می‌باشـد کـه در روش فـوق تعیـین آن مشکل می‌باشد.برای حل این مشکل می‌تواند فیلد چهارمی به نام parent ماننـد شـکل زیر به هر گره اضافه نمود که به پدرش اشاره کند. با استفاده از این فیلـد مـی‌توانـد بـه پدر گره‌ها دستیابی پیدا کرد.

Lchild	Parent	Data	Rehild

مانند سایر ساختارها که در فصل قبل که برای پیاده‌سازی آنها از struct استفاده کردیم برای پیاده‌سازی گره‌های درخت از struct به صورت زیر استفاده می‌کنیم.

پیاده سازی گره های درخت

```
Struct node type
        {
         Node type *left;
         int info
         Node type * right;
         };
         node type *node;
```

پس از توصیف ساختار گره، باید گره‌ای را ایجاد کنیم و به درخت اضـافه کـنیم. بـرای انجام این کار از تابع ( )malloc موجود در زمان C به صورت زیر استفاده می‌کنیم:

```
node = (node type *)malloc (size of (struct node type))
```

این تابع حافظه‌ای به اندازه ساختمان node Type اختصاص می‌دهد و آدرس را در اشاره‌گر node قرار می‌دهد.

معایب نمایش درخت دودویی با آرایه
۱- در نمایش درختهای دودوئی با استفاده از آرایه به غیر از درختهای دودوئی کامل و پر، مکانهای زیادی از آرایه خالی می‌ماند. ۲- با افزایش گره‌های درخت، طول آرایه قابل افزایش نیست. ۳- اعمال درج یا حذف گره از درخت کارآمد نمی‌باشد، زیرا نیاز به جابجایی عناصر آرایه می‌باشد.

٦,٦ پیمایش درخت‌های دودوئی

اعمال زیادی وجود دارد که می‌توان روی درخت‌های دودوئی انجام داد. مانند پیدا کردن پدر یا برادر یک گره. عملکردی که معمولاً بیشتر روی درختهای دودوئی صورت می‌گیرد، ایده پیمایش درخت یا دستیابی به هر گره آن می‌باشد. پیمایش کامل درخت، یک لیست یا ترتیب خطی از اطلاعات موجود در آن درخت را ایجاد می‌کند. پیمایش‌های مختلف، لیستهای متفاوتی را ایجاد می‌کند.

اگر R,V,L به ترتیب حرکت به چپ (زیر درخت چپ)، ملاقات کردن یک گره (برای مثال چاپ اطلاعات موجود در گره) حرکت به راست (زیر درخت راست) باشد، آنگاه شش ترکیبی ممکن برای پمایش یک درخت خواهیم داشت:

RLV , RVL , VRL , VLR, LRV, LVR

اگر تنها حالتی را انتخاب کنیم که ابتدا به سمت چپ و سپس به سمت راست حرکت کنیم، تنها سه ترکیب VLR , LRV , LVR را خواهیم داشت که این سه ترکیب، سه روش استاندارد برای پیمایش یک درخت دودوئی T با ریشه R می‌باشد. این سه ترکیب با توجه به موقعیت V (visit) نسبت به L (left) و R (Right) به ترتیب inorder (میانوندی)، Postorder (پسوندی) و preorder (پیشوندی) می‌نامند.

هر یک از این سه روش پیمایش را می توان به دو صورت بازگشتی و غیربازگشتی نوشت. اما پیاده‌سازی بازگشتی این الگوریتم‌ها ساده‌تر از پیاده‌سازی غیربازگشتی آنها است. بنابراین ابتدا به بیان روش بازگشتی این روشها می‌پردازیم و الگوریتم غیربازگشتی یکی از این روشها را مورد بررسی قرار خواهیم داد.

۱,۶,۶ روش پیمایش Preorder

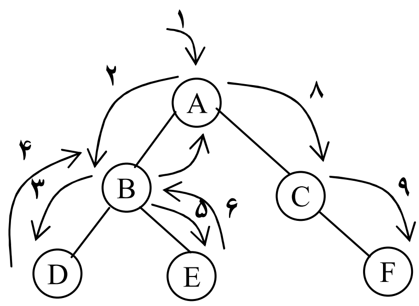
در روش پیمایش Preorder یا VLR یک درخت دودوئی غیرخالی به صورت زیر پیمایش می‌شود:

روش پیمایش Preorder
۱- ریشه را ملاقات کنید. ۲- زیردرخت چپ را به روش Preorder پیمایش کنید. ۳-زیردرخت راست را به روش Preorder پیمایش کنید.

پیمایش Preorder با توجه به مراحل فوق بدین صورت است که: از ریشه شروع می‌کنیم و آن را ملاقات می‌کنیم. سپس به سمت چپ حرکت کرده و اطلاعات گره‌های موجود را تا رسیدن به آخرین گره سمت چپ در مسیر حرکت، می‌نویسیم. پس از رسیدن به آخرین گره سمت چپ و ملاقات آن به سمت راست حرکت می‌کنیم

(چنانچه حرکت به سمت راست ممکن نباشد به گره بالاتر می‌رویم) و زیر درخت سمت راست آن را ملاقات می‌کنیم و این روند را برای تمام گره‌های درخت ادامه می‌دهیم.

مثال ۶,۲: درخت دودوئی شکل ۶,۱۰ را در نظر بگیرید.



شکل ۶,۱۰

ملاحظه می‌کنید که A ریشه این درخت است و زیردرخت چپ آن شامل گره‌های E , D, B و زیردرخت راست آن شامل گره‌های F , C می‌باشد. حال پیمایش Preorder را بر روی این درخت انجام می‌دهیم. برای حل به صورت شماره‌هایی روی خط‌چین‌ها نوشته شده است.

در مرحله (۱) گره ریشه را ملاقات می‌کنیم و آن را در خروجی می‌نویسیم. سپس به سمت گره چپ ریشه یعنی B حرکت می‌کنیم و آن را در خروجی می‌نویسیم (مرحله ۲). بعد به طرف چپ گره B یعنی D حرکت می‌کنیم و آن را در خروجی می‌نویسیم (مرحله ۳).

چون گره D دارای فرزند سمت چپ نیست به طرف سمت راست گره D حرکت می‌کنیم و چون این گره دارای فرزند راست نیز نیست به گره بالاتر یعنی گره B برمی‌گردیم (مرحله ٤). حال به طرف فرزند راست گره B حرکت می‌کنیم و آن را ملاقات کرده و در خروجی می‌نویسیم (مرحله ٥). چون گره E دارای فرزند راست و

چپی ندارد، به بالا می‌گردیم (مرحله ٦) و چون فرزنـدان راسـت و چـپ گـره B قـبلاً ملاقات شده‌اند باز یک مرحله دیگر هم به بـالا برمـی‌گـردیم (مرحلـه ۷) حـال فرزنـد راست گره A را ملاقات می‌کنیم و در خروجی می‌نویـسیم (مرحلـه ۸). چـون گـره C دارای فرزند چپی نمی‌باشد بـه طـرف فرزنـد راسـت آن حرکـت مـی‌کنـیم و آن را در خروجی می‌نویسیم (مرحله ۹).

خروجی حاصل از پیمایش این درخت به صورت زیر خواهد بود.

ABDECF

پیاده‌سازی پیمایش Preorder به صورت بازگشتی

تابع Preorder نشان دهنده برنامـه همـایش درخـت دودوئـی بـه صـورت پیـشوندی می‌باشد. همانگونه که اشاره گردید در این روش پیمایش ابتدا ریشه ملاقـات مـی‌شـود سپس فرزندان چپ و بعد از آن فرزندان است:

زیربرنامه پیمایش Preorder به صورت بازگشتی

```
void Preorder (node type *node)
        {
            if (node)
                {
                        Pintf(”%d”, node → info);
                        Preorder (node→ Lchild);
                        Preorder (node→ Rchild);
                }
        }
```

٦,٦,٢ روش پیمایش inorder

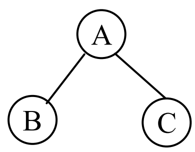
در روش پیمایش inorder یا LVR یک درخت دودوئی غیرخالی به صورت زیر پیمایش می‌شود:

روش پیمایش inorder
۱- زیردرخت چپ را به روش inorder پیمایش کنید. ۲- ریشه را پردازش کنید. ۳- زیردرخت راست را به روش inorder پیمایش کنید.

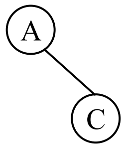
پیمایش inorder با توجه به مراحل فوق بدین صورت است که: از ریشه شروع کرده تا جائی که ممکن است به سمت چپ حرکت می‌کنیم. با رسیدن به آخرین گره سمت چپ، محتویان آن گره را چاپ می‌کنیم و سپس به سمت راست حرکت می‌کنیم و با آن مثل گره ریشه برخورد می‌کنیم و به منتهی‌الیه سمت چپ می‌رویم و آن گره را ملاقات می‌کنیم. اگر در گره‌ای حرکت به سمت راست ممکن نباشد، یک گره به سمت بالا برمی‌گردیم آن را ملاقات می‌کنیم و سپس به سمت راست حرکت می‌کنیم. این روند تا ملاقات کردن تخلیه گره‌های درخت ادامه می‌دهیم.

مثال٦,٣: گره درخت ساده شکل زیر را در نظر بگیرید. در پیمایش میانوندی این درخت ابتدا فرزند چپ آن یعنی B چاپ می‌شود و سپس خود ریشه A و بعد از آن فرزند راست ریشه یعنی C چاپ می‌شود.

Inorder= BAC



و در درخت ساده شکل چون زیردرخت چپ خالی است پیمایش به صورت AC خواهد بود.



ما در بررسی پیمایش‌های درخت، سعی خواهیم کرد درخت‌ها را به زیردرخت‌های ساده‌ای مثل درختهای فوق تبدیل کنیم تا پیمایش درخت به سهولت انجام گیرد.

پیاده‌سازی پیمایش inorder به صورت بازگشتی

تابع inorder نشان دهنده برنامه پیمایش درخت دودوئی به صورت میانوندی می‌باشد. همانگونه که در مثال فوق دیدید در این روش پیمایش ابتدا زیردرخت چپ و سپس ریشه و بعد زیردرخت راست پیمایش می‌شود.

زیربرنامه پیمایش inorder به صورت بازگشتی

```
void inorder (node typoe *node)
   {
        if(node)
          {
                inorder(node→Lchild);
                printf("%d", node→ info);
                inorder(node→ Rchild);
          }
   }
```

٦,٦,٣ روش پیمایش Postorder

در روش پیمایش Postorder یا LRV یک درخت دودوئی غیرخاص به صورت زیر پیمایش می‌شود:

روش پیمایش Postorder

۱- زیردرخت چپ را به روش Postorder پیمایش کنید.

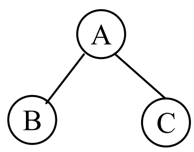
۲- زیردرخت راست را به روش Postorder پیمایش کنید.

۳- ریشه را پردازش کنید.

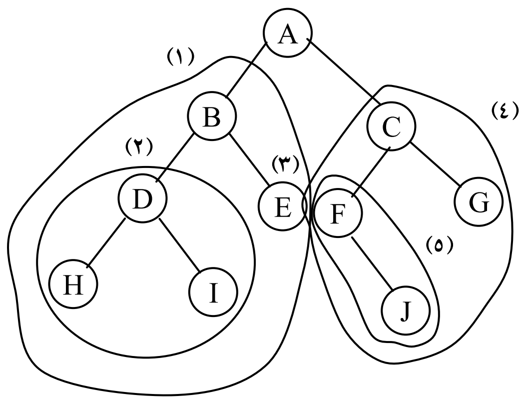
پیمایش Postorder بدین صورت است که: از ریشه شروع می‌کنیم و به طرف چپ حرکت می‌کنیم تا به آخرین گره برسیم و از این گره شروع می‌کنیم تا جایی که ممکن است به سمت راست حرکت می‌کنیم. چنانچه حرکت به راست ممکن نباشد، محتویات این گره را می‌نویسیم و به گره بالایی برمی‌گردیم.

مثال۶,۴: اگر درخت ساده شکل زیررا در نر بگیرید در پیمایش پسوندی این درخت ابتدا فرزند چپ ریشه، سپس فرزند راست ریشه و بعد خود ریشه ملاقات می‌شود.

Postorder = BCA



مثال۶,۵: درخت شکل زیر را در نظر بگیرید همایش inorder درخت را به دست آورید.



در مرحله (۱) به سراغ زیردرخت چپ ریشه یعنی A می‌رویم. حال با این زیردرخت مثل مرحله (۱) عمل می‌کنیم. یعنی به سراغ زیردرخت چپ آن می‌رویم یعنی گره B را ریشه فرض می‌کنیم. (مرحله ۲) مشاهده می‌کنید که این زیردرخت نشان دهنده یک درخت ساده است که می‌توان به راحتی عمل پیمایش را روی آن انجام داد. پیمایش میانوندی این زیردرخت ساده به صورت HDI می‌باشد. حال پس از پیمایش زیردرخت مرحله (۲) گره ریشه یعنی B ملاقات می‌کنیم و به سراغ زیر درخت راست گره B می‌رویم و آن را پیمایش می‌کنیم (مرحله ۳). بعد از اتمام کلیه گره‌های زیر درخت چپ گره A حال خود ریشه یعنی A را ملاقات می‌کنیم و پس از آن به سشراغ زیردرخت راست A می‌رویم (مرحله ٤). حال به سراغ زیردرخت چپ C می‌رویم (مرحله ۵). زیردرخت مرحله (۵) یک درخت ساده می‌باشد که به راحتی می‌توان آن را همایش کرد (F , G). بعد از پمایش زیر درخت مرحله (۵) ریشه C را ملاقات می‌کنیم و به سراغ زیر فرزند است آن یعنی G می‌رویم و آن پیمایش می‌کنیم.

خروجی حاصل از پیمایش این درخت به صورت زیر خواهد بود:

DFIBEAJCG

پیاده سازی پیمایش postroder به صورت بازگشتی

تابع Postorder نشان دهنده برنامه پیمایش درخت دودوئی به صورت پسوندی می‌باشد. در این روش پیمایش ابتدا زیر درخت چپ سپس زیر درخت راست و بعد ریشه پیمایش می‌شود.

زیربرنامه پیمایش postroder به صورت بازگشتی

```
void Postorder (node Type *node)
   {
        if (node)
          {
                Postorder (node→Lchild);
```

```
                Postorder(node→Rchild);
                Printf("%d",node→info);
                }
        }
```

٦,٧ پیمایش غیربازگشتی درخت دودوئی

همانگونه که اشاره گردید، الگوریتم‌های پیمایش درخت‌هـای دودوئـی را مـی‌تـوان بـه صورت بازگشتی نوشت. حال در این قسمت الگوریتم غیربازگشتی پیمـایش میانونـدی را بررسی می‌کنیم. در محل‌های پیمایش، گره‌ها باید در پشته قرار گیرند. و در صـورت لزوم از آن خارج شوند. در حالت بازگشتی عمل قرار دادن گـره‌هـا در پـشته و حـذف آن‌ها از پشته توسط سیستم انجام می‌شود. در حالی که در روش غیربازگشتی، این عمل باید توسط برنامه صورت گیرد. تابع ( )inorder2 نشان دهنده پیاده‌سازی میانوندی به صورت غیربازگشتی می‌باشد

زیربرنامه پیمایش inorder و preorder به صورت غیربازگشتی

inorder

```
#define M 100
void inorder 2 (node Typoe *node)
  {
        struct stack
          {
                int top'
                node, Type item [M]
          }s;
           node Type *p;
          s.top=-1;
          p=tree;
                do{
                        while (p!=null)
```

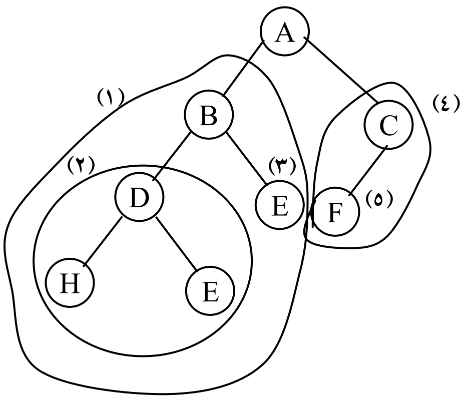
```
            {
             push (s,p);
              p=p→Lchild;
           }
           if (!empty(s))
             {
                    p=pop(s)
                    print("%d",P→info);
                    p=P→Rchild;
              }
           }while (!empty(s) || P!=null);
}
```

preorder

و برای نوشتن پیمایش preorder درخت به صورت غیربازگشتی کافی است در برنامه فوق دستور printf("%d",P→info) را بـه قبـل از دسـتور Push (s,p) انتقـال دهید.

.

مثال٦,٦: درخت شکل زیر را در نظر بگیرید و پیمایش Postorder این درخـت را بـه دست آورید:



ابتدا به سمت زیر درخت چپ ریشه یعنی A می‌رویم (مرحله ۱) حال B را بـه عنـوان ریشه جدید در نظر می‌گیریم و به طرف زیر درخـت چـپ آن مـی‌رویـم (مرحلـه ۲). زیردرخت مرحله (۲) یک درخت دودوئی ساده می‌باشد که به صورت HID پیمـایش می‌شود. حال به سراغ زیردرخت راست B می‌رویم و آن را پیمایش مـی‌کنـیم (مرحلـه ۳). و چون زیردرخت راست و چپ B پیمایش شد حال خود B ملاقات می‌شود.

حال به ریشه اصلی یعنی A برمی‌گردیم و به سراغ زیـر درخـت راسـت آن مـی‌رویـم (مرحله ٤) در این مرحله گره C را به عنوان ریشه جدید در نظر می‌گیریم و بـه سـراغ زیر درخت چپ آن می‌رویم یعنی F (مرحله ٥) و آن را پیمایش مـی‌کنـیم و چـون C دارای زیر درخت راست نمی‌باشد خودش را پیمـایش مـی‌کنـیم. چـون دو زیردرخـت چپ و راست ریشه اصلی یعنی A پیمایش شد نه خود A نیز پیمایش می‌شود.

خروجی حاصل از پیمایش این درخت به صورت زیر می‌باشد:

HIDEBJFGCA

٦,٨ ساخت درخت دودوئی با استفاده از پیمایش آن

ما تاکنون با استفاده از درخت دودوئی داده شده مبادرت به پیمایش آن می‌کردیم. اکنون می‌خواهیم با استفاده از پیمایش داده شده یک درخت دودوئی، اقدام بـه سـاخت خـود درخت کنیم. قابل ذکر است اگر یک نوع پیمایش از درخت موجـود باشـد، نمـی‌تـوان درخت دودوئی منحصر به فردی را ایجاد کنیم.

اگر یک نوع پیمایش از درخت موجود باشد، نمی‌توان درخت دودوئی منحصر به فردی را ایجاد کنیم.

به عنوان مثال اگر فقط پیمایش inorder درخت در دست باشـد و بـا اسـتفاده از ایـن پیمایش بخواهیم درخت را سازیم، نمی‌توانیم درخت اولیه را بسازیم، بلکه چند درخت به دست می آید که ممکن است یکی از آنها درخت اولیه بوده باشد. ولی اگـر پیمـایش

inorder درخت و یکی از دو پیمایش Postorder و یا preorder درخـت موجـود باشد، می‌توان درخت منحصر به فردی را ساخت.

اگر پیمایش میانوندی و یکی از پیمایش های پسوندی یا پیشوندی یک درخت دودوئی را داشته باشیم ، می توانیم آن درخت را به صورت یکتا ترسیم کنیم.

قاعده اصلی برای ایجاد درخت با استفاده از پیمایش آن به صورت زیر است:

ساخت درخت دودوئی به صورت یکتا با داشتن پیمایش میانوندی و یکی از پیمایش های پسوندی یا پیشوندی
اگر پیمایش Preorder مشخص باشد، اولین گره آن به ریشه است. اگر نمایش Postorder مشخص باشد آخرین گره، ریشه است. وقتی گره ریشه مشخص شد، تمام گره‌های زیردرخـت چـپ و زیردرخـت راسـت را می‌توان با استفاده از نمایش inorder پیدا کرد.

با توجه به سه روش پیمایش بررسی شده ملاحظه مـی کنـیم کـه هـر الگـوریتم دارای همین سه مرحله است و زیردرخـت چـپ ریـشه همـواره قبـل از زیردرخـت راسـت پیمایش می شود. تفاوت این سه الگوریتم در زمـانی اسـت کـه در آن ریـشه پـردازش می‌شود. به طور مشخص در الگوریتم ”pre“، ریشه قبـل از پیمـایش زیـر درخـت‌هـا پردازش می‌شود. در الگوریتم دارای ”In“ ریشه مـابین پیمـایش زیردرختهـا پـردازش می‌شود و الگوریتم دارای ”post“ ریشه بعد از پیمایش زیردرختها پردازش می‌شود.

نکته: اگر پیمایش های پسوندی یا پیشوندی یک درخت دودوئی را داشته باشیم ، ممکن است نتوانیم آن درخت را به صورت یکتا ترسیم کنیم.

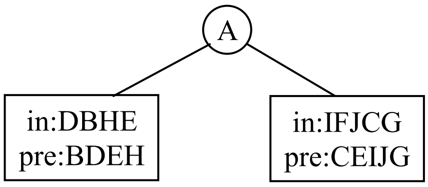
مثال٦،٧: فرض کنید پیمایش‌های inorder و preorder یک درخت دودوئی به صورت زیر باشد، درخت دودوئی موردنظر را رسم کنید.

D B H E A I F J C G :inroder

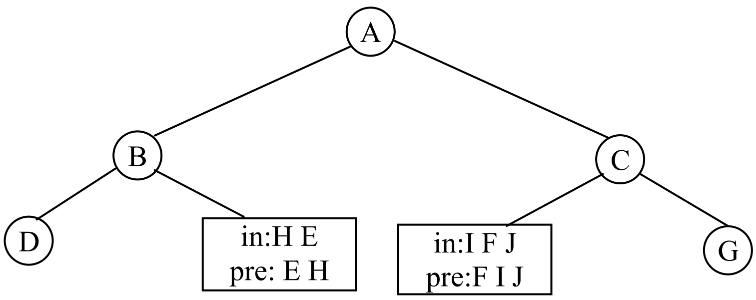
A B D E H C F I J G :Preorder

با توجه به پیمایش preorder متوجه می‌شویم که A ریشه درخت است.

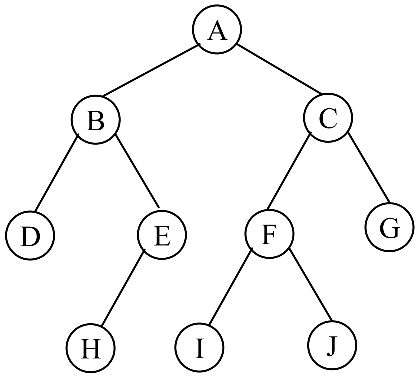
در پیمایش inorder، تمام گره‌های موجود در سمت چپ A متعلق به زیردرخت چپ و تمام گره‌های موجود در سمت راست A، متعلق به زیردرخت راست است.



حال مراحل بالا را برای زیردرخت چپ و راست A تکرار می‌کنیم. در زیردرخت چپ با توجه به پیمایش preorder متوجه می‌شویم که B ریشه است. پس در inorder تمام گره‌ها سمت چپ B را به عنوان زیردرخت و تمام گره‌های راست آن را به عنوان زیردرخت راست در نظر می‌گیریم و همین کار را برای زیردرخت راست ریشه اصلی یعنی A نیز انجام می‌دهیم.



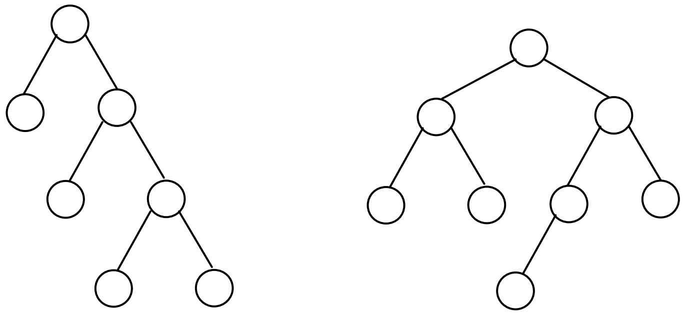
حال در بسته‌ای با شماره (۱) مشخص می‌شود که E ریشه می‌باشد و H فرزند چپ آن و در بسته‌ای باشماره (۲) F ریشه می‌باشد و I فرزند چپ آن J فرزندراست آن خواهد بود.



۶٫۹ نمایش عبارات محاسباتی با درخت دودوئی

کاربرد دیگری از درختهای دودوئـی، روش نمـایش یـک عبـارت حـاوی عملونـدها و عملگرها توسط درخت دودوئی محض است درخت دودوئی محض درختـی دودوئـی است که در آن تمام گره‌ها از درجه صفر یا از درجه ۲ باشند.

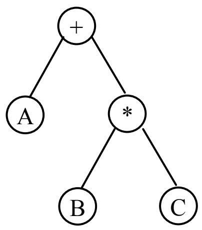
به عنوان مثال درخت شکل ۶٫۱۱(الف) نشان دهنده یک درخت دودوئـی محـض مـی باشد و در حالی که درخت شکل۶٫۱۱ (ب) درخت دودوئی محض است:



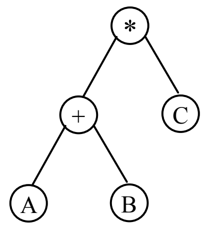
شکل۶٫۱۱ (ب) درخت دودوئی که محض نیست (الف) درخت دودوئی محض

برای نمایش عبارت توسط درخت دودوئی محض تعریف شده در بـالا بـدین صـورت عمل می‌کنیم که ریشه درخت دودوئی محض حاوی عملگری است کـه بایـد بـر روی نتایج ارزیابی زیر درختهای چپ و راست عمل کند. همچنین در این درخت عملگرهـا در گره‌های غیربرگ و عملونـدها در گـره‌هـای بـرگ درخـت دودوئـی محـض قـرار می‌گیرند.

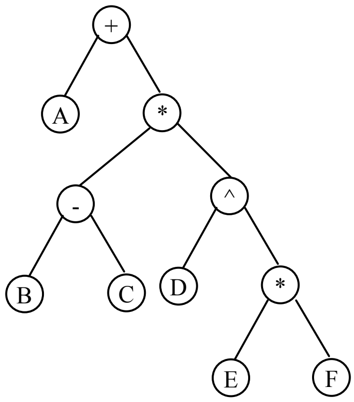
شکل ٦,١٢ چند عبارت و نمایش آنها را نشان می‌دهد. (کاراکتر n برای نمایش توان بـه کار رفته است).



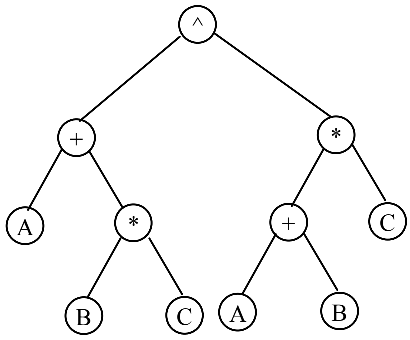
(a) $A + B \times C$



(b) $(A + B) \times C$



(c) $A + (B - C) \times D^\wedge (E \times F)$



(d) $(A+B\times C)^\wedge((A+B)\times C)$

شکل ۶٫۱۲ چند عبارت و نمایش درختی آنها

پیمایش preorder درخت‌های فوق عبارت prefix و پیمایش postorder درخت‌های فوق عبارت postfix معادل را تولید می‌کند. اما اگر درخت شکل (a) به روش inorder پیمایش گردد، عبارت A+B*C که یک عبارت infix است حاصل می گردد. اما چون ترتیب انجام اعمال، از ساختار درخت نتیجه می‌شود، درخت دودوئی فاقد هرگونه پارامتر است. لذا یک عبارت infix که مستلزم استفاده از پرانتزها جهت تعویض تقدم عادی عملگرها است را نمی‌توان با پیمایش inorder ساده به دست آورد.

۶٫۱۰ پیمایش ترتیب سطحی

پیمایش های پیشوندی، میانوندی و پسوندی که در بخش قبلی مورد بحث و بررسی قرار دادیم در واقع پیمایش‌های عمقی هستند که برای بکارگیری آنها نیاز به استفاده از پشته است. پیمایش ترتیب سطحی روش دیگری از پیمایش درخت دودوئی است که بجای پشته از صف استفاده می‌کند. عملکرد این پیمایش بدین صورت است که در ابتدا ریشه بازیابی می‌شود سپس فرزند چپ ریشه و به دنبال آن فرزند راست ریشه بازیابی می‌گردد. این روش بازیابی را برای تمام سطوح درخت اعمال می‌کنیم.
الگوریتم این پیمایش به صورت زیر می‌باشد:

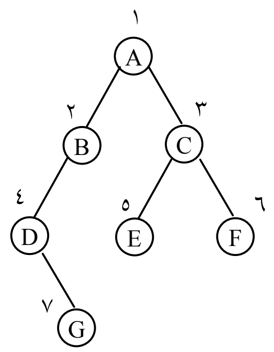
الگوریتم پیمایش ترتیب سطحی درخت دودوئی

```
void level order (node typoe node)
        {
            front = = near= = -1;
            while (node)
                  {
                    printf("%d", node→ info);
```

```
                    if (node→Lchild!=null)
                    addq (node→Lchile);
                    if (node→Rchild!=null)
                    addq (node→ Rchild);
                   delet(node);
          }
    }
```

مثال: درخت دودوئی شکل٦,١٣ را در نظر بگیرید. پیمـایش ترتیـب سـطحی آن بـدین صورت است که ابتدا ریشه را بازیابی می‌کنیم.س پس گره‌های سحط بعـدی (فرزنـدان ریشه) را از چپ به راست بازیابی می‌کنـیم و بعـد از آن بـه سـراغ سـطح‌هـای بعـدی می‌رویم تا زمانی که کل درخت را پیمایش کرده باشیم.

$\Rightarrow$ A B C D E F G



شکل ٦,١٣

یا به بیان دیگر کلیه گره‌ها را از بالا به پایین و از چپ به راست شماره‌گذاری می‌کنیم و به ترتیب شماره در خروجی می‌نویسیم.

٦,١١ درختان نخی دودوئی

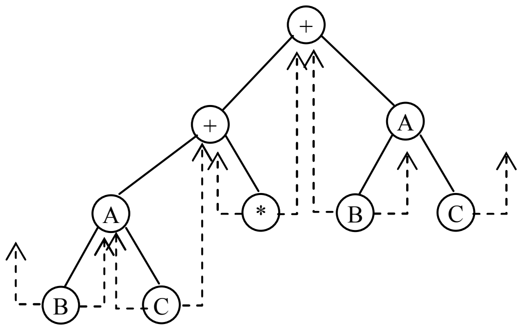
اگر نمایش پیوندی درخت دودوئی T را در نظر بگیرید، ملاحظـه مـی‌شـود کـه تعـداد اتصالات تهی در ورودیهای فیلدهای Lchild و Rchild بیشتر از تعـداد اتـصال‌هـای غیرتهی است. در یک درخت دودوئی تعداد از تعـداد کـل اتـصالات آن یعنـی 2n،بـه تعداد n+1 اتصال تهی است. این فضا با قرار دادن نوع دیگـری از اطلاعـات بـه جـای

ورودیهای پوچ می‌تواند به شکل کاراتری مورد استفاده قرار گیرد. به طور مشخص ما اشاره‌گرهای خاصی را جانشین ورودیهای تهی می‌کنیم که به گره‌های بالاتر درخت اشاره می‌کند. این اشاره‌گرهای خاص را نخ‌کشی ها و درخت دودوئی با این اشاره‌گرها را درختهای نخ‌کشی شده می‌گویند.

در یک درخت دودوئی تعداد از تعداد کل اتصالات آن یعنی 2n، به تعداد n+1 اتصال تهی است(استفاده نشده است) و n-1 اتصال استفاده شده است.

نخ‌کشی ها در یک درخت نخ کشی شده باید از اشاره‌گرهای معمولی تمیز داده شوند. در نمودار یک درخت نخ‌کشی شده، نخ‌کش‌ها را معمولاً با خط‌چین نمایش می‌دهند.
برای نخ‌کشی یک درخت دودوئی راههای متعددی وجود دارد اما هر نخ‌کشی متناظر با یک پیمایش خاص T است. نخ‌کشی ما متناظر با پیمایش inorder درخت T است.
برای ایجاد اتصالات نخی می‌توان از قوانین زیر استفاده نمود (فرض کنید که ptr نشان دهنده یک گره می‌باشد).
اگر $ptr \rightarrow Lchild$ تهی باشد آن را طوری تغییر می‌دهیم که به گره‌ای که در پیمایش inorder قبل از ptr قرار دارد، اشاره کند.
اگر $ptr \rightarrow Rchild$ تهی باشد، آن را طوری تغییر می‌دهیم که به گره‌ای که در پیمایش inorder بعد از ptr قرار دارد، اشاره کند.
شکل٦,١٤ نمونه‌ای از درخت نخی را نشان می‌دهد که در آن اتصالات نخی به صورت نقطه‌چین مشخص شده است. این درخت دارای ۹ گره و ۱۰ اتصال تهی که با اتصالات نخی تعریف شده‌اند. اگر درخت را به روش inorder پیمایش کنیم، گره‌های حاصل به صورت H D I B E A F C G خواهند بود. برای مشاهده اینکه چگونه اتصالات نخی ایجاد می‌شوند، گره E را به عنوان نمونه انتخاب می‌کنیم. چون فرزند چپ E یک اتصال تهی است آن را به گونه‌ای تغییر می‌دهیم که به گره‌ای که قبل از E

قرار دارد، یعنی B اشاره کند. به طور مشابهی، چون فرزند راست E نیز یک اتصال تهی است، آن را با اشاره‌گر به گره‌ای که بعد از E قرار می‌گیرد یعنی A تعویض می‌کنیم. بقیه اتصالات به طور مشابهی ایجاد می‌گردند.



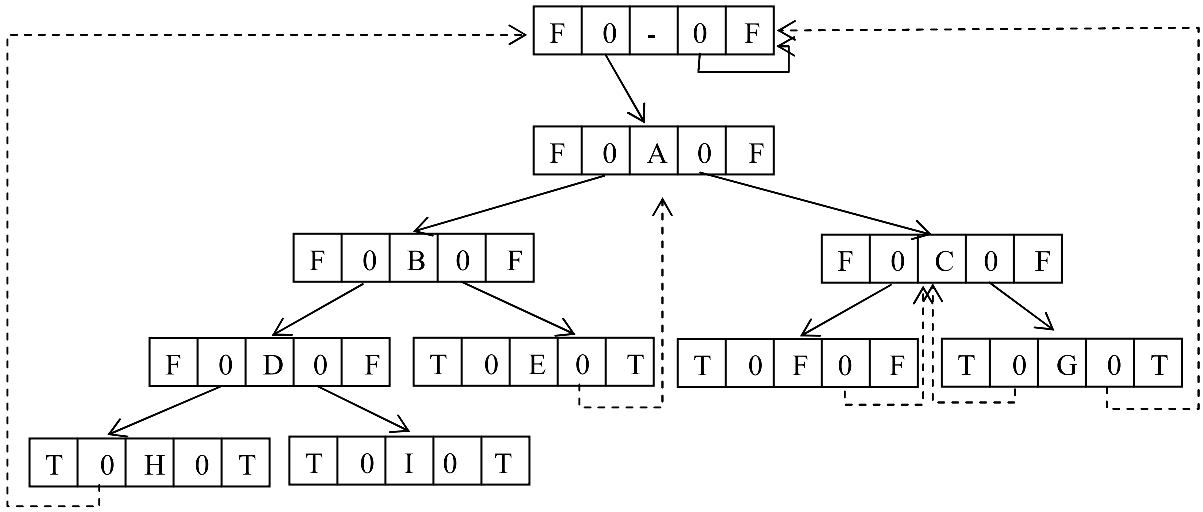
شکل ٦,١٤ نمونه‌ای از درخت نخی

در شکل فوق اشاره‌گر سمت راست یعنی G و اشاره‌گر سمت چپ یعنی H به هیچ جائی اشاره نمی‌کند و در حالی که هدف از ارائه درخت های نخی این بود هیچ اشاره‌گر تهی نداشته باشیم پس یک گره Head برای هر درخت دودوئی نخی در نظر می‌گیریم. یعنی همواره درخت نخی تهی دارای یک گره بنام گره Head مطابق شکل زیر داریم:

left-thread	Lchild	Data	Rchild	right-thread
TRUE	0	-	0	FALSE

Left – child به شروع گره اول درخت واقعی اشاره می‌کند توجه داشته باشید اشاره‌گرهای تهی رها شده (loose threads) به گره head اشاره می‌کند.

نمایش حافظه‌ای کامل درخت شکل   در شکل   ارائه شده است. متغیر root به گره Head درخت اشاره می‌کند و $root \rightarrow left-child$ به شروع گره اول درخت واقعی اشاره می‌کند.



۶٫۱۲ درختهای جستجوی دودوئی (Binary Search Tree- BST)

این بخش یکی از مهمترین ساختمان داده علم کامپیوتر یعنی درخت جستجوی دودوئی را مورد بحث و بررسی قرار می‌دهد. این ساختار به ما امکان می‌دهد تا یک عنصر را جستجو کنیم و آن را با زمان اجرای میانگین $T(n) = O(\log_2 n)$ پیدا کنیم. علاوه بر این به سادگی می‌توان عنصر را در این ساختار داده اضافه کرد یا از آن حذف کرد.

این ساختار داده در مقابل ساختارهای زیر قرار دارد:

الف)آرایه مرتب شده. در این ساختار می‌توان یک عنصر را جستجو کرد و آن را با زمان اجرای میانگین $a\log_2(n)$ پیدا کرد. اما اضافه کردن و حذف کردن پرهزینه است.

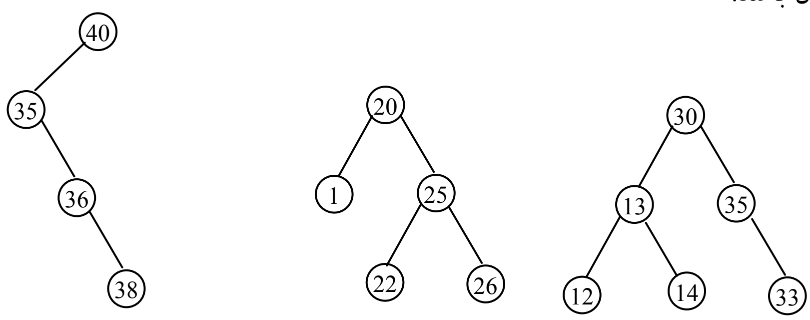
ب)لیست پیوندی. در اینجا به سادگی می‌توان عناصر را اضافه یا حذف کرد. اما در این روش جستجوی عنصر و پیدا کردن آن پرهزینه است. چون باید از جستجوی خطی با زمان اجرای $o(n)$ استفاده کرد.

تعریف: یک درخت جستجوی یک درخت دودوئی است که ممکن است تهی باشد. اگر درخت تهی نباشد دارای خصوصیات زیر می باشد:

هر عنصر دارای یک کلید است و دو عنصر نباید دارای کلید یکسان باشند، در واقع کلیدها منحصر به فرد هستند.

مقدار هر گره بزرگتر از هر مقدار در زیر درخت چپ و کوچکتر از هر مقدار در زیر درخت راست آن است.

چند نمونه از درختان دودوئی در شکل ٦,١٥ ارائه شده است. درخت شکل (الف و ب) درختهای جستجوی دودوئی هستند. اما درخت شکل (ج) یک درخت جستجوی دودوئی نیست زیرا در این درخت، زیردرخت راست گرهی با کلیه ۳۵ باید بزرگتر از آن باشد.



(الف) (ب) (ج)

شکل ٦,١٥ نمونه ای از درختهای جستجوی دودوئی

از آنجایی که درخت جستجوی دودوئی شکل خاصی از یک درخت دودوئی است، لذا برنامه ها برای یک درخت جستجوی دودوئی تفاوتی با برنامه‌هایی که قبلاً برای یک درخت دودوئی به کار گرفتیم، ندارد. تمام عملکردهای درختهای دودوئی که قبلاً مورد بحث قرار گرفت نیز برای درختهای جستجو نیز اعمال می‌شود. به عنوان مثال می‌توانیم از پیمایش‌های postroder, preorder, inorder بدون هیچگونه تغییری استفاده کنیم. به این اعمال می‌توان، درج، حذف و جستجو را نیز اضافه نمود.

٦,١٢,١ جستجوی یک عنصر در درخت جستجوی دودوئی

از آنجائی که تعریف درخت جستجوی دودوئی را به صورت بازگشتی انجام دادیم، لذا بیان و ارائه یک روش جستجوی بازگشتی نیز ساده می‌باشیم. اگر بخواهیم در درخت به جستجوی عنصری با کلید key بگردیم، برای جستجو ابتدا از ریشه شروع می‌کنیم. اگر ریشه تهی باشد جستجو ناموفق خواهد بود و اگر غیرتهی باشد در این صورت key را با ریشه مقایسه می‌کنیم اگر مقدار key کمتر از مقدار ریشه باشد به سراغ زیردرخت چپ می‌رویم و اگر مقدار key بزرگتر از ریشه باشد، به سراغ زیردرخت راست می‌رویم. تابع search درخت را به صورت بازگشتی جستجو می‌کند.

زیربرنامه جستجوی یک عنصر در درخت جستجوی دودوئی

```
int search (tree root, int key)
{
        /*return a pointer to the node that contains key, if there no
        such node, return Null.*/
        if (!root) return Null;
        if (key = = root → data) return root;
        if (key < root → data)
                return search (root → Lchild, key)
        return search  (root → Rchild, key);
```

در تابع فوق tree struct را می توان به صورت زیر نوشت:

نحوه تعریف درخت

```
struct tree {
        tree *Lchild;
        int data;
        tree * Rchild;
            }
```

پیچیدگی الگوریتم جستجوی عنصر

فرض کنید در یک درخت جستجوی دودوئی T می‌خواهیم یک عنصر اطلاعاتی را جستجو کنیم. تعداد مقایسه‌ها محدود به عمق درخت است. این موضوع از این واقعیت ناشی می‌شود که ما از یک مسیر درخت به طرف پایین پیش می‌رویم. بنابراین زمان اجرای جستجو متناسب با عمق درخت است.

فرض کنید n عنصر اطلاعاتی $A_N, ..., A_2, A_1$ داده شده است و می خواهیم در یک درخت جستجوی دودوئی اضافه شوند. برای n عنصر تعداد !n جایگشت وجود دارد. هر یک از چنین جایگشتی باعث به وجود آمدن درخت مربوط به خود می‌شود. می‌توان نشان داد که عمق میانگین !n درخت تقریباً برابر با $c\log_2 n$ است که در آن C=1.4 می‌باشد. بنابراین زمان اجرای میانگین جستجو یک عنصر در درخت دودوئی T با n عنصر متناسب با $\log_2 n$ است یعنی $f(n) = O(\log_2 n)$.

تحلیل الگوریتم insert:

زمان لازم برای جستجوی key برابر با n (ارتفاع درخت) می‌باشد و بعـد از جـستجو، عمل درج نیاز به زمان ؟ دارد. بنابراین زمـان کـل مـورد نیـاز insert برابـر بـا O(h) می‌باشد.

٦،١٢،٢ درج عنصری به داخل درخت جستجوی دودوئی

فرض کنید T یک درخت جستجوی دودوئی باشد، می‌خواهیم عنصری بـا مقـدار key را در درخت وارد کنیم. در واقع، جستجو و وارد کردن یک عنصر، تنها با یک الگوریتم جستجو و وارد کردن انجام می‌شود.

فـرض کنیـد عنـصر key داده شـده اسـت. الگـوریتم زیـر مکـان key را در درخـت جستجوی دودوئی جستجو کنـد. اگـر جـستجو نـاموفق باشـد، key را در محلـی کـه جستجو خاتمه پیدا نموده است، درج می‌کنیم.

الف)key را با N ریشه درخت مقایسه کنید.

i)اگر key<N به طرف بچه چپ N پیش بروید.

ii)اگر key>N به طرف بچه راست N پیش بروید.

ب)مرحله (الف) را تکرار کنید تا یکی از حالتهای زیر اتفاق بیفتد.

i)گره N را وقتی key=N است ملاقات کنید. در این حالت جستجو موفق است.

ii)یک زیردرخت خالی را ملاقات کنید که بیان می‌کند جستجو موفق نیست و key را به جای زیردرخت خالی اضافه کنید.

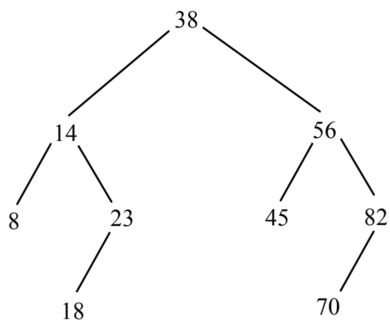
بنابراین الگوریتم اضافه کردن شبیه الگـوریتم جـستجو اسـت و فقـط بایـد بـه انتهـای الگوریتم تابع زیر را اضافه کنید.

تابع درج عنصری به داخل درخت جستجوی دودوئی

```
void insert (tree *node,int key)
{
  tree ptr;
        {
```

```
ptr=(tree) malloc (size of (node));
          ptr → data = key
          ptr → Lchild = Null;
          ptr → Rchild= wall;
                    if (*node →data >key)
                              *node → lchild = ptr;
                    else if (*node → data<key)
                              *node → Rchild=Ptr;
}
```

مثال۶,۸: درخت دودوئی T شکل۶,۱۶را در نظر بگیرید.



T یک درخت جستجوی دودوئی است. یعنی هر گره N در T از هر عـدد زیردرخـت چپ آن بزرگتر و از هر عدد زیردرخت راست آن کوچکتر است.
فرض کنید عدد ۳۵ جای‌گزین عدد ۲۳ شود آنگاه T همچنان یـک درخـت جـستجوی دودوئی باقی خواهد ماند ولی اگر عدد ۴۰ را جایگزین عدد ۲۳ کنـیم T یـک درخـت جستجوی دودوئی نخواهد بود، چون در زیردرخت عدد ۳۸ هیچ عددی نباید بزرگتـر از آن باشد.

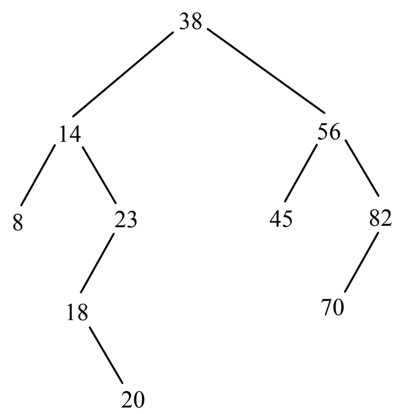
حال فرض کنید می‌خواهیم عدد ۲۰ را در درخت درج کنیم عملیات موردنظر به صورت زیر خواهد بود:

key=20 را با ریشه یعنی ۳۸ مقایسه می‌کنیم چون از آن کوچکتر است به طرف فرزند چپ ۳۸ یعنی ۱٤ می‌رویم.

Key=20 را با ۱٤ مقایسه می‌کنیم. چون از آن بزرگتر است به طرف بچه راست ۱٤ یعنی ۲۳ می‌رویم.

Key=20 را با ۲۳ مقایسه می‌کنیم. چون از آن کوچکتر است به طرف بچه چپ ۲۳ یعنی ۱۸ می‌رویم.

Key=20 را با ۱۸ مقایسه می‌کنیم چون از آن بزرگتر است به طرف بچه راست ۱۸ می‌رویم و چون ۱۸ بچه راست ندارد، ۲۰ را به عنوان بچه راست به درخت اضافه می‌کنیم.



٦,۱۲,۳ حذف یک عنصر از درخت جستجوی دودوئی

فرض کنید T یک درخت جستجوی دودوئی است. می‌خواهیم عنصر اطلاعاتی key را از درخت T حذف کنیم. در این بخش الگوریتمی برای انجام این کار ارائه خواهد شد. این الگوریتم ابتدا با استفاده از الگوریتم جستجو مکان گره N که حاوی عنصر

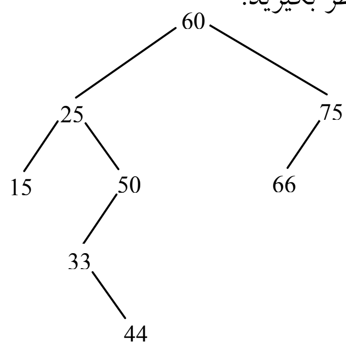
key است و همچنین مکان پدر آن P(N) را پیدا می‌کند. روشی که با آن N از درخت حذف می‌شود بستگی به تعداد بچه‌های N دارد که سه حالت وجود دارد:

حالت ۱) N بچه‌ای ندارد. در این صورت با جایگزین شدن مکان N در گره پدر P(N) به وسیله اشاره‌گر Null، گره N از درخت حذف می‌شود.

حالت ۲) N دقیقاً یک بچه دارد. در این صورت با جایگزین شدن مکان N در P(N) به وسیله مکان تنها بچه N، گره N از درخت حذف می‌شود. (یعنی بچه گره N جایگزین خود گره N می‌شود.)

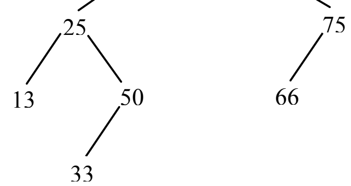
حالت ۳) N دو بچه دارد. فرض کنید S(N) نمایش گره بعدی پیمایش Inorder در گره N باشد. دانشجو می‌تواند تحقیق کند که S(N) بچه چپ ندارد. آنگاه با حذف S(N) از T (با استفاده از حالت ۱ یا ۲) و سپس با جانشین کردن گره S(N) به جای گره N در درخت T ، گره N از T حذف می‌شود.

مثال٦,٩: درخت جستجوی شکل٦,١٧را در نظر بگیرید.



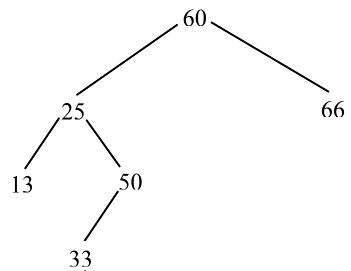
الف)فرض کنید می‌خواهیم گره ٤٤ را از درخت حذف کنیم. با توجه به اینکه گره ٤٤ فرزندی ندارد کافی است اشاره‌گر پدر آن (یعنی ٣٣) را Null کنیم. شکل (الف) این درخت را پس از حذف ٤٤ نشان می‌دهد.

شکل ٦,١٧ (الف)



ب)فرض کنید می‌خواهیم به جای گره ٤٤، گره ۷۵ را از درخت حذف کنیم. با توجه به اینکه گره ۷۵ تنها یک فرزند دارد، گره ۷۵ حذف و فرزند آن ٦٦ جایگزین آن می‌شود.

شکل (ب) این درخت را پس از حذف ۷۵ نمایش می دهد.

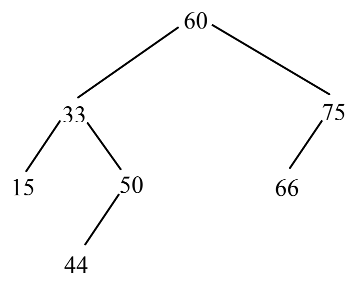


(ج) فرض کنید می خواهیم به جای ٤٤ یا ۷۵ گره ۲۵ را از درخت حذف کنیم. توجه کنید که گره ۲۵ دو بچه دارد. ابتدا پیمایش میانوندی درخت را بدست می آوریم.

15 25 33 44 50 60 66 75

ملاحظه می کنید که گره ۳۳ گره بعدی ۲۵ در پیمایش میانوندی است. بنابراین باید گره ۳۳ را جانشین گره ۲۵ کنیم.

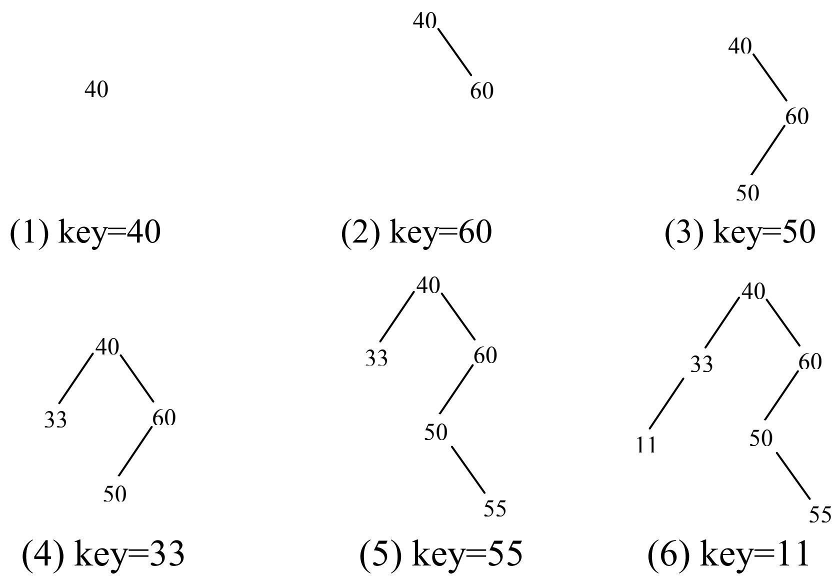
یعنی ابتدا گره ۳۳ را بنا به حالت ۲ حذف کرده و سپس آن را به جای ۲۵ قرار می دهیم. تاکید می کنیم که جانشینی گره ۳۳ به جای گره ۲۵ در حافظه تنها با تغییر اشاره گرها انجام می گیرد نه با جابجایی محتوای یک گره از یک مکان به مکان دیگر. شکل (ج) درخت را پس از حذف گره ۲۵ نمایش می دهد.



مثال٦,١٠ فرض کنید شش عدد زیر به ترتیب در یک درخت جستجوی دودوئی خالی اضافه شده است:

40 , 60 , 50 , 33 , 55, 11

شکل ٦,١٨ شش مرحله از درخت را نشان می‌دهد. تاکید می‌کنیم کـه اگـر شـش عـدد داده شده با ترتیب مختلف داده شده باشد، آنگاه ممکن است
درخت‌های حاصل نیز با هم فرق کنند و عمق مختلفی داشته باشند.



شکل ٦,١٨ درج تعدای عدد در درخت جستجوی خالی

٦,١٢,٤ حذف عناصر تکراری: کاربردی از درخت جستجوی دودوئی

مجموعه‌ای از n عنصر اطلاعاتی $A_1, A_2, ..., A_N$ را در نظر بگیرید. فرض کنید بخواهیم تمام عناصر تکراری را که در این مجموعه وجود دارند را پیدا کرده و آنها را حذف کنیم. یک راه ساده برای این منظور به شرح زیر است:

الگوریتم الف)عناصر را از A_1 تا A_N یعنی از چپ به راست بخوانید.

(i)برای هر عنصر A_k, A_k را با $A_1, A_2, ...A_{k1}$ مقایسه کنید. (یعنی A_k را با عناصری که قبل از A_k هستند مقایسه کنید)

(ii)اگر A_k در بین $A_1, A_2, ..., A_{k1}$ وجود داشت،آنگاه A_k را حذف کنید.

پس از آن که تمام عنصر خوانده شد و مورد بررسی قرار گرفت آنگاه در این مجموعه عناصر تکراری نخواهد داشت.

پیچیدگی زمانی الگوریتم (الف)

پیچیدگی زمانی الگوریتم A به وسیله تعداد مقایسه‌ها تعیین می‌شود. هر مرحله شامل A_k به طور تقریبی به $k-1$ مقایسه احتیاج دارد. چون A_k با عنصرهای $A_1, A_2, ..., A_{k1}$ مقایسه می‌شود، بنابراین $T(n)$ تعداد مقایسه‌های مورد نیاز در الگوریتم A تقریباً برابر است با:

$$1+2+3+...+(n-2)+(n-1)=\frac{n(n-1)}{2}=O(n^2)$$

برای مثال برای n=1000 عنصر، الگوریتم (الف) تقریباً به 500000 مقایسه احتیاج دارد.

با استفاده از یک درخت جستجوی دودوئی می‌توان الگوریتم دیگر نوشت که عناصر تکراری را از یک مجموعه n عنصری $A_1, A_2, ..., A_N$ حذف می‌کند.

الگوریتم ب)با استفاده از عناصر $A_1, A_2, ..., A_N$ یک درخت جستجوی دودوئی بسازید. هنگام ساختن درخت، در صورتی که مقدار A_k قبلاً در درخت ظاهر شده باشد A_k را از لیست حذف کنید.

مزیت اصلی الگوریتم (ب) آن است که هر عنصر A_k تنها با عنصرهای یک شاخه درخت مقایسه می‌شود. می‌توان نشان داد که طول میانگین چنین شاخه‌ای تقریباً برابر با $C\log_2 k$ است که در آن $C=1.4$ می‌باشد. بنابراین $T(n)$ تعداد کل مقایسه‌های موردنیاز در الگوریتم (ب) تقریباً به $n\log_2 n$ مقایسه نیاز دارد برای مثال برای $n=1000$ عنصر، الگوریتم (ب) مستلزم تقریباً ۱۰۰۰۰ مقایسه است که در الگوریتم الف)تعداد مقایسه‌ها برابر ۵۰۰۰۰۰ است قابل توجه است که در بدترین حالت (حالتی که درخت جستجوی دودوئی به صورت مورب باشد) تعداد مقایسه‌های الگوریتم (الف) و (ب) با هم برابر هستند.

مثال٦,١١: فرض کنید الگوریتم (الف) و (ب) بر لیست ۱۵ عددی زیر به کار گرفته شده است:

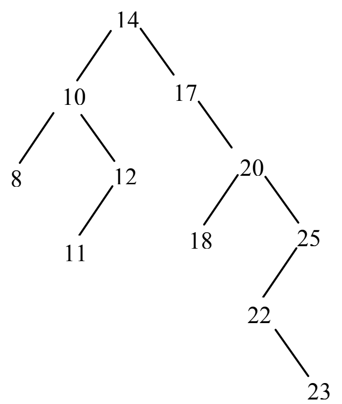
14 , 10 , 17, 12, 10 , 11, 20 , 12, 18 , 25, 20 , 8, 22, 11, 23

تعداد مقایسه‌ها را برای حذف عناصر تکراری با استفاده از دو الگوریتم الف و ب را به دست آورید.

تعداد دقیق مقایسه‌ها در الگوریتم (الف)

$$0+1+2+3+2+4+5+4+6+7+6+8+9+5+10= 72$$

با اعمال الگوریتم (ب) بر این لیست عددی، درخت شکل٦,١٩ به دست می‌آید.



تعداد دقیق مقایسه‌ها برابر است با:

$$0+1+1+2+2+3+2+3+3+3+3+2+4+4+5= 38$$

۶,۱۳ هرم‌ها (HEAPS)

در بخش قبل درخت دودوئی کامل تعریف شد. در این بخش شکل خاصی از درخت دودوئی کامل که در اکثر کاربردها مخصوصاً مرتب سازی اطلاعات مورد استفاده قرار می‌گیرد، معرفی می گردد.

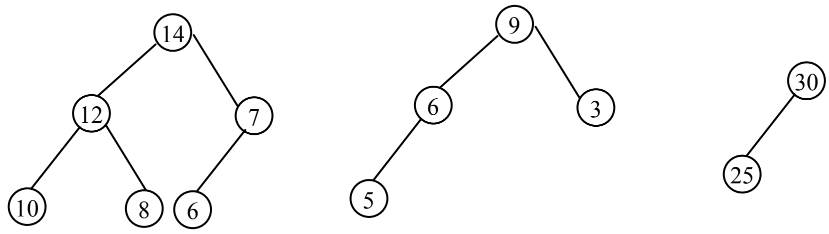
درخت HEAP (هرم)

تعریف: max tree درختی است که مقدار کلید هر گره آن کمتر از مقادیر کلید فرزندانش (اگر وجود داشته باشد) نباشد. (مساوی یا بیشتر باشد) max heap یک درخت دودوئی کامل است که یک max tree نیز می‌باشد.

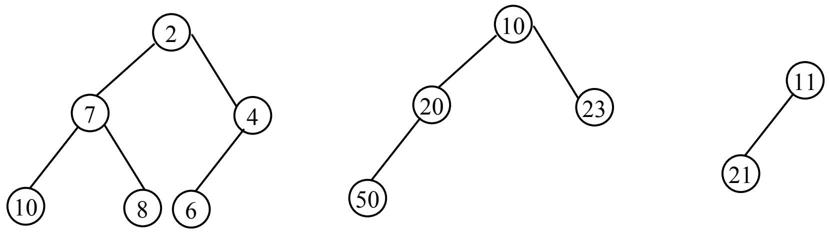
تعریف: min tree درختی است که مقدار کلید هر گره آن بیشتر از مقادیر کلیدهای فرزندانش (اگر وجود داشته باشند) نباشد (کوچکتر یا مساوی باشد). Min heap یک درخت دودوئی کامل است که در واقع یک min tree باشد.

توجه داشته باشید که کامل بودن درخت یک شرط لازم برای heap بودن می‌باشد. و ریشه min heap حاوی کوچکترین کلید موجود در درخت و ریشه max heap حاوی بزرگترین کلید موجود در درخت می‌باشد.

شکل ۶,۲۰ چند مثال max heap و شکل نیز چند نمونه از min heap را ارائه می‌کند.



شکل ۶,۲۰ چند نمونه از max heap



شکل ٦,٢٠ چند نمونه از Min heap

ما هرم (heap) را با استفاد از یک آرایه، البته بدون استفاده از موقعیت 0 آن، پیاده‌سازی می‌شود.

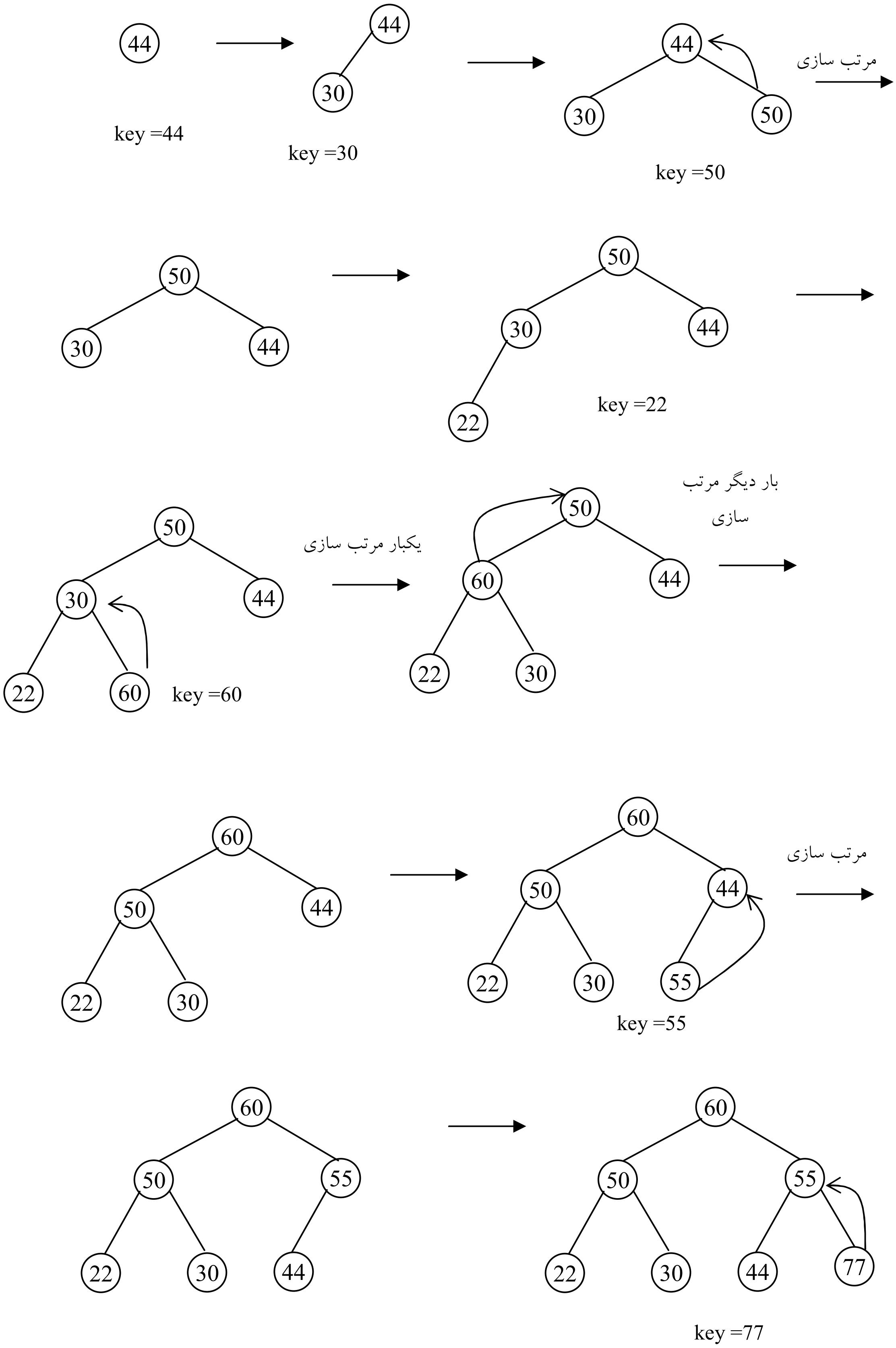
٦,١٣,١ اضافه کردن یک عنصر در max heap

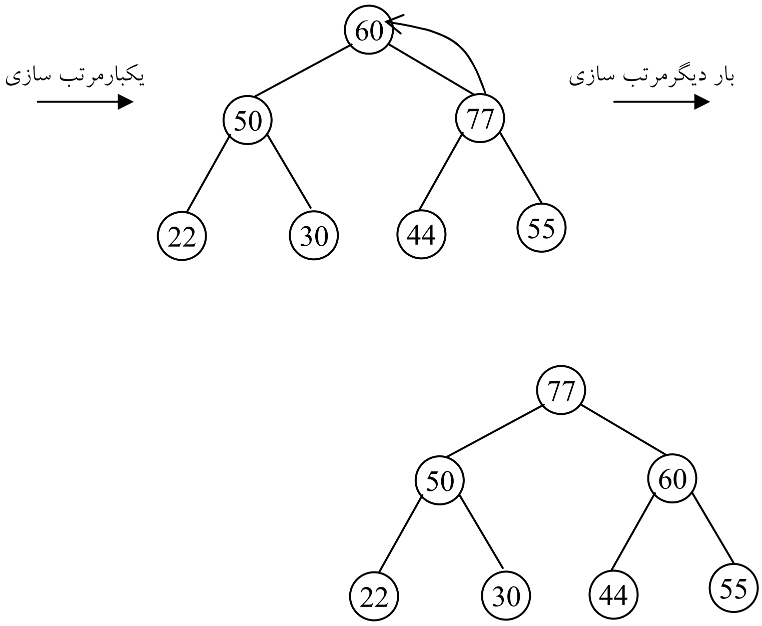
در استفاده از درخت هرم، اعمال اضافه کردن یک عنصر در درخت و حذف عنصری از آن به صورتی انجام می‌گیرد که درخت به صورت هرم (یعنی درخت کامل) باقی بماند. بنابراین پس از عملیات اضافه کردن و یا حذف کردن کارهای اضافی برای تنظیم درخت لازم است. عمل اضافه کردن به این صورت است که: همواره درخت از چپ به راست در هر سطح آخر پر شده و سپس پر کردن از سطح بعدی انجام می‌گیرد. پس از اضافه کردن عنصر عمل مرتب‌سازی انجام می‌شود. در این عمل یک گره در پایین‌ترین سطح تا جایی که لازم باشد با گره پدرش جابجا می‌شود.

مثال٦,١٢: فرض کنید بخواهیم یک Max Heap از لیست عددی زیر بسازیم:

44 , 30 , 50 , 22 , 60 , 55 , 77

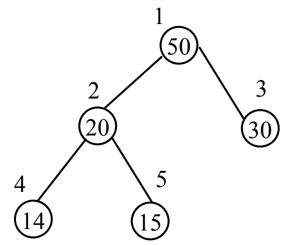
این کار را می‌توان با اضافه کردن هشت عدد یکی پس از دیگری در درخت خالی انجام داد. شکل ٦,٢١ (الف) تا (ح) تصویرهای مربوطه Heap را پس از اضافه شدن هر یک از هشت عنصر نشان می‌دهد.


key =44
key =30
key =50
مرتب سازی
key =22
key =60
یکبار مرتب سازی
بار دیگر مرتب
سازی
key =55
مرتب سازی
key =77



شکل ۶,۲۱ درج تعدادی عدد در هرم

همانگونه که اشاره گردید معمولاً درخت Heap به صورت آرایه پیاده‌سازی می‌شود. به این صورت که مکان هر گره در درخت دقیقاً متناظر با اندیس در یک خانه آرایه می‌باشد.



1	2	3	4	5
50	20	30	14	15

و شماره فرزند چپ گره i برابر 2i و شماره فرزند راست گره i برابر 2i+1 می‌باشد. همانطور که در درخت مشاهده می‌کنید فرزند راست گره 20 (با شماره i=2) به گره 15 با شماره i=5 می‌باشد.

تابع insert – max- heap عمل اضافه کردن عنصری به max heap حاوی n عنصر را انجام می‌دهد.

پیاده سازی زیر برنامه اضافه کردن عنصری به max heap

```
void insert- max- heap (element item, int * n)
 {
        int i;
        i=++(*n);
        while ((i!=1) && (item.key)>heap [i/2].key))
                {   heap [i] =heap [i/2];
                        i=/2;
                }
                heap [i]=item;
        }
```

تحلیل پیچیدگی زمان عمل اضافه کردن به heap

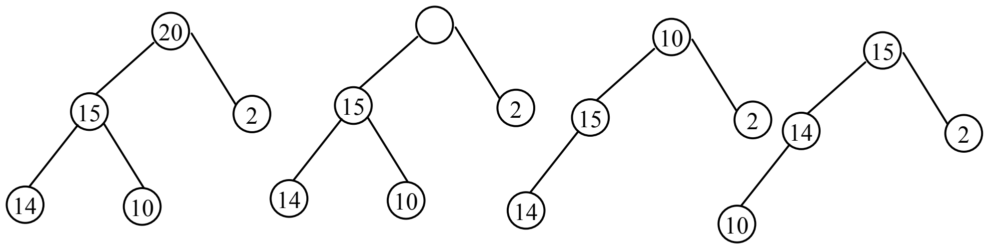
از آنجا که heap یک درخت کامل با n عنصر می‌باشد. دارای ارتفاع $[\log_2 (n+1)]$ می‌باشد. این بدین معنی است که حلقه while به میزان $O(\log_2 n)$ تکرار می‌شود. بنابراین پیچیدگی تابع اضافه کردن به heap برابر با $O(\log_2 n)$ می‌باشند.

٦,١٣,٢ حذف عنصری از Max heap

در عمل حذف همواره ریشه heap حذف می‌شود و سمت راست‌ترین عنصر موجود در پایین ترین سطح در ریشه قرار می‌گیرد و درخت مجدداً تنظیم می‌شود. برای مثال از

heap شکل ٦,٢٢ (الف) می‌خواهیم ٢٠ را حذف کنیم، از آنجایی که heap حاصل دارای تنها چهار عنصر می‌باشد، باید درخت را به گونه‌ای سازماندهی کنیم که متناظر با درخت دودوئی کامل با چهار عنصر گردد.

ساختار مطلوب در شکل ٦,٢٢ (ب) آورده شده است. بعد از حذف گره 20، عنصر 10 را در گره ریشه قرار می‌دهیم. شکل ٦,٢٢ (ج) در حال حاضر ساختار درخت صحیح است ولی درخت حاصل max heap نمی‌باشد. برای برقراری مجدد heap، به سمت پایین حرکت نموده و گره پدر را با فرزندان آن مقایسه و عناصر خارج از ترتیب را تا برقراری مجدد heap تعریف می‌کنیم. شکل (د) درخت heap نهایی را نشان می‌دهد. تابع detelte- max- heap این مراحل را نشان می‌دهد.



(الف) (ب) (ج) (د)

شکل ٦,٢٢

پیاده سازی زیر برنامه حذف کردن عنصری به max heap

```
element delete –max- heap (int *n)
  {
        int parent , child;
        element item, temp;
        if (heap – Empty (*n)]
          {
                fprintf(stderr, “The heap is empty\n”);
                exit(1);
          }
```

```
        item = heap [1];
        temp = heap [(*n)--];
        parent =1;
        parent =2;
        child =2;
        while (child <=*n)
          {
                if (child<*n) && (heap
[child].key<heap[child+1].key)
                  child++;
                if (temp.key)=heap[child].key)
                   break;
                heap [parent]= heap [child];
                child * = 2;
           }
                heap [parent]=temp;
                return item;
   }
```

تحلیل تابع delete- max- heap

این تابع با حرکت به سمت پایین درخت heap به مقایسه و تعویض گره‌های پدر و فرزند را تا هنگامی که تعریف heap دوباره برقرار شود، انجام می‌دهد. از آنجایی که ارتفاع یک درخت heap با n عنصر برابر با $\lceil \log_2(n+1) \rceil$ می‌باشد. حلقه while، به میزان $O(\log_2 n)$ مرتبه تکرار می‌گردد. بنابراین پیچیدگی حذف برابر با $O(\log_2 n)$ می‌باشد.

٦,١٤ صف اولویت

غالباً هرم‌ها برای پیاده سازی صف اولویتها استفاده می‌شوند. برخلاف صفهایی که در فصل ٤ بررسی کردیم، در صف اولویت عنصری که دارای بالاترین (یا پایین ترین) اولویت هست، حذف می‌شود. در هر لحظه می‌توانیم عنصری را با اولویت اختیاری به

داخل صف اولویت اضافه کنیم. آرایه ساده ترین نمایش برای یک صف اولویت می‌باشد. فرض کنید که این آرایه دارای n عنصر باشد، در آرایه به سادگی می‌توانیم با قرار دادن یک عنصر جدید در انتهای آن، عمل درج به صف اولویت را انجام دهیم. بنابراین عمل درج دارای پیچیدگی زمانی $\theta(n)$ است. برای حذف، ابتدا باید بزرگتر (یا کوچکترین) کلید را جستجو و سپس آن را حذف کنیم. زمان جستجو برابر است با $\theta(n)$ می‌باشد. استفاده از لیست پیوندی غیرمرتب زمان اجرای برنامه را تا حدودی بهبود می‌بخشد. می‌توانیم عنصری را به ابتدای لیست در $\theta(n)$ اضافه نمائیم و باید لیست را برای پیدا نمودن عنصری با بزرگترین کلید جستجو کنیم، زمان حذف برابر با $\theta(n)$ است. با این نمایش، تنها زمان مورد نیاز جهت تغییر مکان عناصر حذف می‌گردد. نمایش صف اولویت به صورت هرم (heap) این امکان را فراهم می‌سازد که هم درج و هم حذف را در زمان $O(og_2 n)$ انجام دهیم و همین امر از آن نمایش مطلوبی می‌سازد.

ساختار داده	درج	حذف
آرایه غیرمرتب	$\theta(1)$	$\theta(n)$
لیست پیوندی غیرمرتب	$\theta(1)$	$\theta(n)$
آرایه مرتب	$O(n)$	$\theta(1)$
لیست پیوندی مرتب	$O(n)$	$\theta(1)$
هرم (max heap)	$O(\log_2 n)$	$og_2 n)$

انواع نمایش مختلف صف اولویت

۶,۱۵ کاربرد heap در مرتب کردن اطلاعات

فرض کنید آرایه A با N عنصر داده شده است. الگوریتم Heap sort که A را مرتب می‌کند از دو مرحله زیر تشکیل یافته است:

مرحله A: یک heap از عناصر آرایه A بسازید.

مرحله B: عنصر ریشه heap را به طور مکرر حذف کنید.

از آنجا که ریشه heap همواره بزرگترین گره درخت است، مرحله B، عنصرهای آرایه A را به ترتیب نزولی حذف کنید.

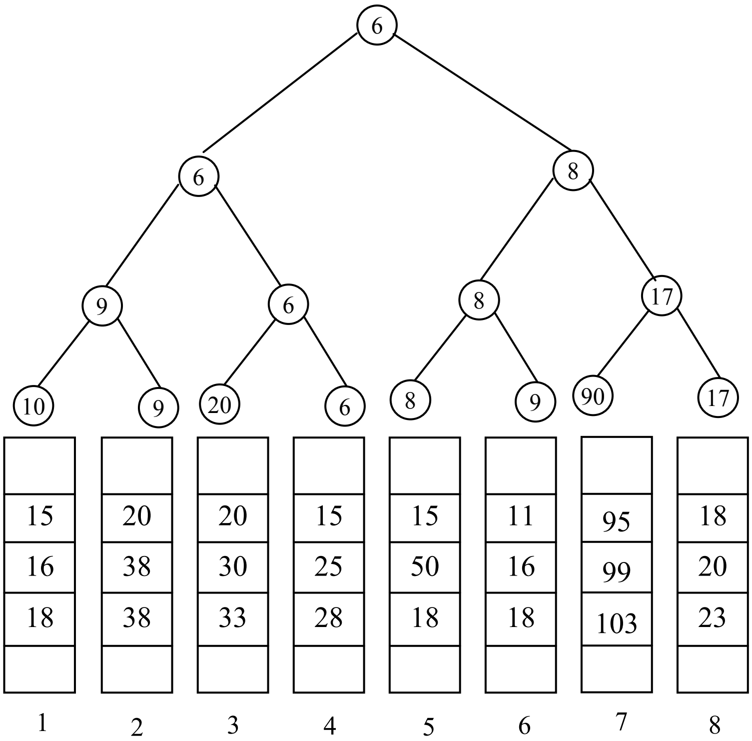
۶,۱۶ درختهای انتخابی

فرض کنید k آرایه مرتب شده غیرنزولی از عناصر داریم و می خواهیم این k آرایه را در یک آرایه واحد ادغام کنیم. n مجموع کل خانه های k آرایه می‌باشد. ساده ترین روش برای ادغام k آرایه مرتب آن است که سطر اول تمام آرایه‌ها را با هم مقایسه کرده و کوچکترین آنها را به دست آوریم. این عمل به k-1 مقایسه نیاز دارد. کوچکترین عنصر را در خروجی چاپ می‌کنیم و چون عنصر خانه اول آرایه به دلیل کوچکترین عنصر بودن حذف شود پس سایر عناصر همان آرایه را به طرف خانه اول انتقال می‌دهیم. چون در هر بار k-1 مقایسه نیاز دارد.و چون در کل n خانه داریم، مرتبه اجرایی آن $O(n,k)$ می‌باشد. با استفاده از ایده درخت انتخابی، تعداد مقایسه‌های لازم را کاهش می‌دهیم.

تعریف: یک درخت انتخابی، یک درخت دودوئی است که هرگاه آن کوچکتر از دو فرزند خود می‌باشد. بنابراین، گره ریشه نشان دهنده کوچکترین گره در درخت می‌باشد.

شکل ۶,۲۳ برای حالت k=8 یک درخت انتخابی را نشان می‌دهد.

در این درخت هر گره غیربرگ نشان دهنده گره کوچکتر است و گره ریشه کوچکترین کلید را نشان می‌دهد و به خروجی هدایت می‌شود.



شکل ٦,٢٣ درخت انتخابی

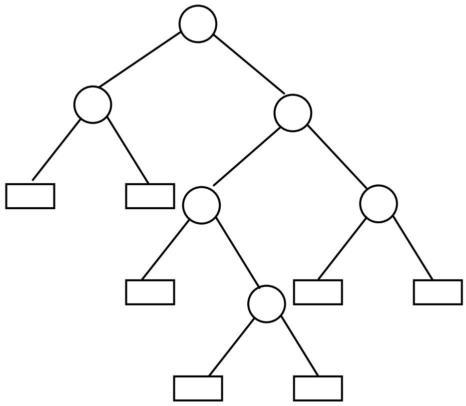
همانگونه که مشاهده می‌کنید ابتدا 10 با 9 مقایسه شده و کوچکتر یعنی 9 به بالا انتقال داده می‌شود. 20 با 6 مقایسه می‌شود و کوچکتر یعنی 6 بالا انتقال داده می‌شـود و الـی آخر. سپس در سطح بعدی 9 با 6 مقایسه شده و کوچکتر به بالا انتقال داده شود و الـی آخر و در نهایت 6 کوچکترین گره خواهد شد. چون کوچکترین گره از آرایـه شـماره 4 می‌باشد در مرحله بعدی خانه دوم از آرایه شماره چهار یعنـی 15 را بـه جـای 6 در نظر می‌گیریم و مراحل بالا را تکرار می‌کنیم.

با توجه به مثال فوق زمان تجدید ساختار درخت برابر با $O(\log_2 k)$ (k تعداد آرایه‌ها می‌باشد). بنابراین زمان لازم جهت ادغام کل n خانه برابر $O(n \log k)$ می‌باشد.

٦,١٧ الگوریتم هافمن: کاربرد درختهای دودوئی

طول مسیر

تعریف: یک درخت دودوئی گسترش یافته یا ۲- درخت ، یک درخت دودوئی است که در آن هر گره 0 یا 2 فرزند دارند. گرههایی که 0 فرزند دارند گرههای خارجی نام دارند و همچنین گرههایی که 2 فرزند دارند گرههای داخلی نام دارند. شکل یک ۲- درخت را نشان میدهد که در آن گرههای داخلی با دایره و گرههای خارجی با مربع نشان داده شده است. در هر ۲- درخت تعداد گرههای خارجی N_E یک واحد بیشتر از تعداد گرههای داخلی N_I میباشد یعنی:



$$N_E = N_I + 1$$

در شکل فوق داریم: $N_I = 6$, $N_E = 7$

تعریف: طول مسیر خارجی L_E یک ۲-درخت T به صورت مجموع طولهای تمام مسیرهایی تعریف میشود که حاصل جمع طول تمام مسیرها از ریشه درخت تا یک گره خارجی است. طول مسیر داخلی L_I به طور مشابه تعریف میشود که در آن به جای گرههای خارجی – از گرههای داخلی استفاده میشود. برای درخت شکل داریم:

$L_I=0+1+1+2+2+3=9$, $L_E=2+2+3+4+4+3+3=21$

ملاحظه می کنید که

$L_I+2n=9+2*6=21=L_E$

که در آن $n=6$ تعداد گره های داخلی است. در واقع، فرمول

$$L_E=L_I+2n$$

برای هر ۲- درخت با n گره داخلی برقرار است.

فرض کنید T یک ۲- درخت با n گره خارجی است و به هر گره خارجی یک وزن(غیر منفی) نسبت داد شده است. طول مسیر وزن داده شده (خارجی) درخت T بنا به تعریف مجموع طول مسیر وزن داده شده است یعنی:

$$P=W_1L_1 + W_2L_2 + \dots + W_nL_n$$

که در آن W_i و L_i به ترتیب وزن و طول مسیر یک گره خارجی N_i است

فرض کنید یک لیست با n وزن داده شده است:

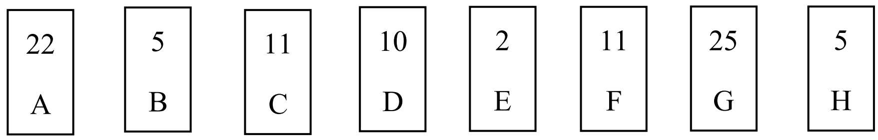
$$w_1, w_2, \dots, w_n$$

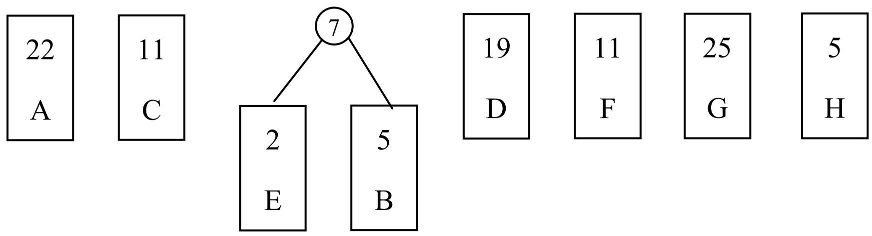
در میان تمام ۲- درخت دارای n گره خارجی و n وزن داده شده، مطلوب است تعیین یک درختT با حداقل طول مسیر وزن داده شده. هافمن الگوریتمی برای پیدا کردن چنین درخت T ای ارائه داد که ما اکنون آن را بیان می‌کنیم.

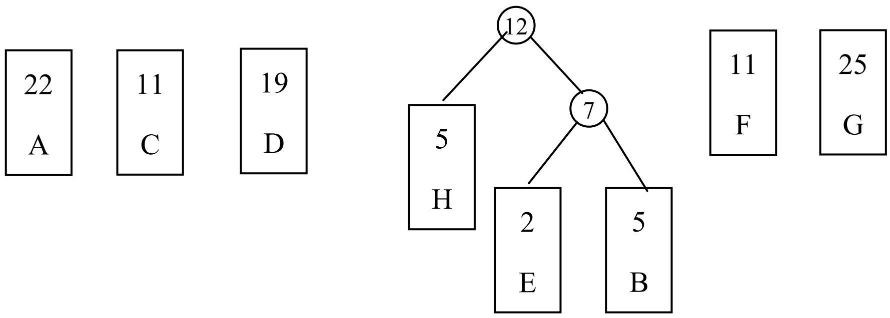
مثال٦,١٣: فرض کنید A, B, C, D, E, F, G , H ، 8 عنصر اطلاعاتی هستند. همچنین فرض کنید به این عناصر وزن‌های زیر نسبت داده شده است:

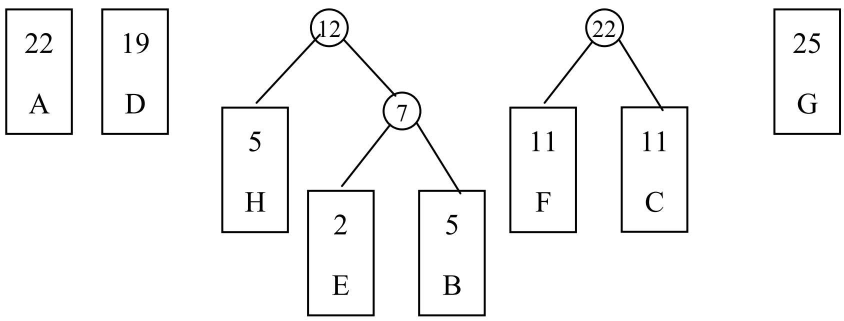
عنصر:	A	B	C	D	E	F	G	H
وزن:	22	5	11	19	2	11	25	5

شکل ٦,٢٤ (الف) تا (ج) چگونگی ساخته شدن درخت T را با حداقل طول مسیر وزن داده شده با استفاده از اطلاعات بالا و الگوریتم هافمن نشان می‌دهد در هر مرحله دو درختی که ریشه کمتری دارند با هم ترکیب می‌کنیم. (وزن آنها را با هم جمع می‌کنیم.)

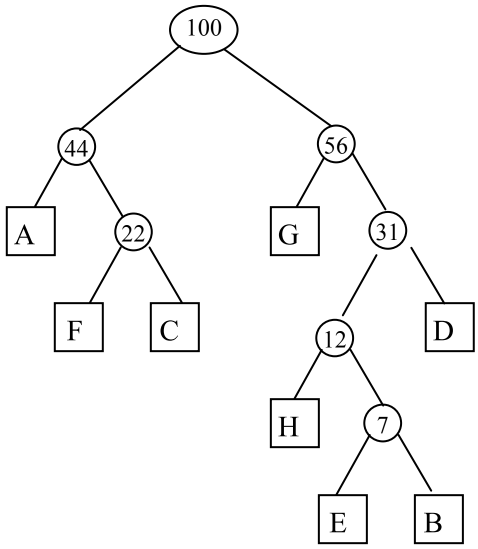

22 A
5 B
11 C
10 D
2 E
11 F
25 G
5 H


22 A
11 C
7
2 E
5 B
19 D
11 F
25 G
5 H


22 A
11 C
19 D
12
5 H
7
2 E
5 B
11 F
25 G


22 A
19 D
12
5 H
7
2 E
5 B
22
11 F
11 C
25 G

اگر همینطور ادامه دهیم شکل نهایی بصورت زیر می شود:



شکل ٦,٢٤

٦,١٨ رمزگذاری هافمن

فرض کنید الفبایی با n عنصر داریم و یک پیام طولانی متشکل از عنصرهای این الفبا موجود است. می خواهیم پیام را بصورت رشته ای از بیت ها کدگذاری کنیم. برای رمزگذاری به هر عنصر یک کد نسبت داده می شود. قابل ذکر است که هر عنصر را می توان بوسیله r بیت کدگذاری کرد که در آن

$$2^{r-1} < n <= 2^r$$

که n نشان دهنده تعداد کاراکترها می باشد.

به عنوان مثال فرض کنید الفبایی حاوی ٤ عنصر با کاراکترهای A,B,C,D است.($2^1<4<=2^2$ پس r=2 یعنی با ٢ بیت می توان این ٤ کاراکتر را کدگذاری کرد.) که کدهای آن به شرح زیر است:

عنصر	A	B	C	D
کد	00	01	10	11

در اینصورت پیام ABACCDA بـه شـکل 00010010101100 رمزگـذاری مـی شود که طول آن ۱٤ بیت است. می خواهیم طول پیام رمزی را به حداقل برسانیم.
این پیام را دوباره بررسی می کنیم. هر یک از حروف B,D فقـط یکبـار در پیـام ظـاهر شده اند، درحالیکه حرف A سه بار تکرار شده است. اگر کدی انتخاب شـود کـه طـول A کمتر از طول B و D باشد طول پیام رمزگذاری شده کمتر خواهد بود. این کـد مـی تواند بصورت زیر باشد:

عنصر	A	B	C	D
کد	0	110	10	111

بـا اسـتفاده از ایـن جـدول پیـام ، ABACCDA بـصورت 0110010101110 رمزگذاری می شود که طول آن ۱۳ بیت است.
در پیام های خیلی طولانی که عناصر زیاد تکرار می شوند، هزینـه صـرفه جـویی قابـل توجه است. درخت های دودوئی برای یافتن کدهای با حداقل طـول بکـار مـی رونـد. برای این منظور از نوعی درخت دودوئی بنام رمزگذاری بنام درخت رمزگذاری هـافمن استفاده می شود. ابتدا هر عنصر یا کاراکتر و فراوانی یا تکـرار آن عنـصر در رشـته داده می شود و با استفاده از درخت هافمن می توان کدی با حداقل طول برای رشـته مـورد نظر تولید کرد.
ساخت درخت هافمن را در بخش قبلی توضیح دادیم فقط در این الگـوریتم بـه شـاخه های سمت چپ درخت هافمن درست شده بیت 0 و به شاخه های راسـت آن بیـت 1 را نسبت می دهیم.
حال با استفاده از مثال ساده ای ساخت درخت رمزگذاری هافمن را توضیح می دهیم.

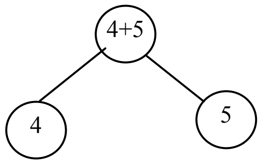
مثال: فرض کنید ۵ عنصر E,T,N,I,S وجود دارند که فراوانی (تکرار) آن ها در رشته مانند جدول زیر است. دقت کنید که فراوانی یا تکرار عنصرها را وزن عنصرها نیز می نامند.

عنصر	E	T	N	I	S
وزن	29	10	9	5	4

ابتدا لیستی از درخت‌های دودوئی یک گره‌ای که هر گره حاوی وزن‌های مربوط به عنصر هستند، ایجاد می‌شوند.

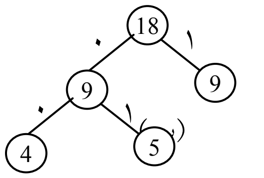
4 5 9 10 20

دو مقدار کوچکتر، یعنی 4 , 5 را انتخاب کرده، درختی دودوئی ایجاد می‌کنیم که ریشه آن مجموع این دو مقدار و این مقادیر برگ‌های آن باشند و به پیوند سمت چپ مقدار 0 و به پیوند سمت راست مقدار 1 را نسبت می‌دهیم. (می‌توان 0 را به پیوند چپ و 1 را به پیوند راست نیز نسبت داد.)

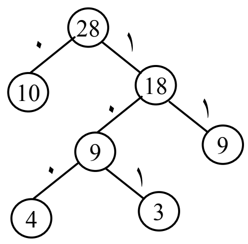


(الف)

اکنون در دنباله (29 , 10 , 9 ,5 , 4) به جای 4 , 5، مجموع آنها یعنی 9 را قرار می‌دهیم تا دنباله (9,9,10,29) به دست آید. اکنون دو مقدار کوچکتر یعنی 9,9 را از این لیست انتخاب کرده و یک درخت دودوئی مثل قبل می‌سازیم.

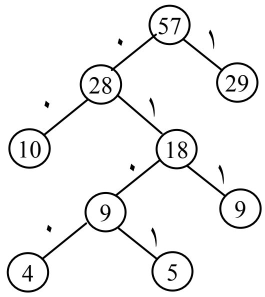


اکنون به جای مقادیر 9,9 در دنباله مجموع آنها یعنی 18 را قرار می‌دهیم تا دنباله (29 , 18 ,10) به دست آید. اکنون با استفاده از دو مقدار 10 , 18 یک درخت دودویی همانند قبل ایجاد می‌کنیم.

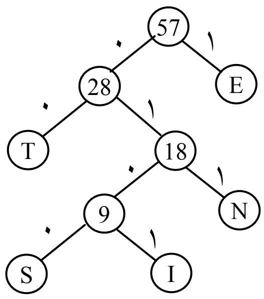


(ج)

اکنون در دنباله (29 , 18 , 10) به جای دو مقدار 10, 18 مجموع آنها یعنی 28 را قرار می‌دهیم تا دنباله (28,29) به دست آید. یک درخت دودوئی از این دو مقدار می‌سازیم.



اکنون این درخت را به درخت رمزگذاری هافمن تبدیل می‌کنیم. برای این کار به جای برگ‌ها نمادهای متناظر آن را در جدول قرار می‌دهیم تا شکل به دست آید. این درخت برای رمزگذاری و رمزگشای رشته‌هایی به کار می‌رود که از نمادهای (E , T, N , I , S) تشکیل می‌شوند که هر عنصر، با ردیابی مسیری از ریشه به آن نماد به دست می‌آید.

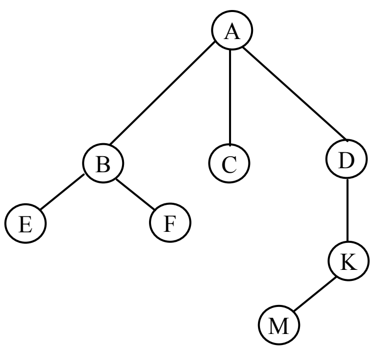


فراوانی	کد	عنصر
29	1	E
19	00	T
9	001	N
5	0101	I
4	0100	S

رمز	پیام
0100101100	SENT
0111010000	NEST
0100010100	SIT

۶,۱۹ درخت عمومی (general tree)

درخت عمومی یک درخت kتایی است که در آن فقط یک گره به نام ریشه با درجه ورودی صفر وجود دارد. و سایر گره‌ها دارای درجه ورودی یک هستند. شکل یک درخت عمومی ۳ تایی با ۸ گره نشان می‌دهد.



ریشه درخت فوق A و فرض می‌کنیم بچه‌های یک گره از چپ به راست مرتب هستند مگر آنکه خلاف آن بیان شود.

تفاوت عمده بین درخت دودوئی و درخت عمومی (یا درخت) این است که در درخت‌ها هر گره می‌تواند بیش از دو فرزند داشته باشد. در حالی که در درخت دودوئی، هر گره حداکثر دو فرزند دارد. به عبارت دیگر، درخت ساختار داده‌ای است که قادر است رابطه سلسله مراتبی بین یک گره والد و چند گره فرزند را نمایش دهد.

به این ترتیب می‌توان گفت درخت مجموعه‌ای متناهی از گره‌هاست که:

گره خاصی به نام ریشه وجود دارد.

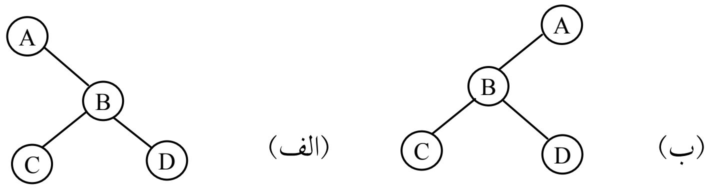
بقیه گره‌ها به n $(n \geq 0)$ مجموعه مجزا به نام‌های $T_1, T_2, ..., T_n$ تقسیم می‌شوند که در آن هر T_i به ازای $i = 1,2,...,n$ یک درخت است. $T_1, T_2, ..., T_n$ زیردرخت‌های ریشه نامیده می‌شوند.

با توجه به تعریف فوق می‌توان این نکته را فهمید که یک درخت دودوئی T' حالت خاصی از درخت عمومی T نیست و این دو در دو دسته مختلف قرار دارند. اختلاف اساسی آنها عبارتند از:

الف)یک درخت دودوئی می‌تواند خالی باشد ولی یک درخت عمومی نمی‌تواند خالی (یا تهی) باشد.

ب)فرض کنید درخت تنها یک فرزند دارد آنگاه این فرزند در یک درخت دودوئی با عنوان فرزند راست یا چپ از هم متمایز می‌شوند، اما در یک درخت عمومی هیچگونه تمایزی بین آنها وجود ندارد.

مثال۶,۱۴: در شکل زیر را در نظر بگیرید:



شکل‌های الف و ب به عنوان درختهای دودوئی دو درخت متمایز هستند ولی به عنوان درختهای عمومی با هم هیچگونه تفاوتی ندارند.

۶,۱۹,۱ نمایش درخت عمومی

چون در درخت عمومی، هر گره ممکن است هر تعداد فرزندی داشته باشد، پیاده‌سازی درخت عمومی پیچیده‌تر از درخت‌های دودوئی است. سه روش را برای نمایش درختها بررسی می‌کنیم:

نمایش درخت عمومی با آرایه

نمایش پیوندی (نمایش درخت عمومی با لیست پیوندی)

نمایش درخت عمومی با استفاده از درخت دودوئی

نمایش درخت عمومی با آرایه

نمایش درخت عمومی با آرایه ساده می‌باشد. برای این کار به سه آرایه نیاز است:

آرایه data برای ذخیره محتویات گره درخت

آرایه Lchild برای نگهداری چپ‌ترین فرزند گره

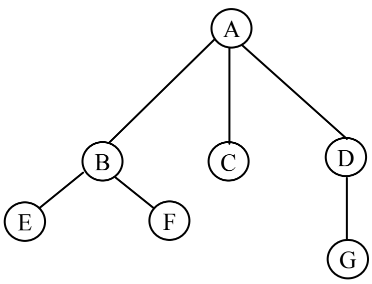
Sibiling همزاد گره را ذخیره می‌کند.

برای جلوگیری از این اتلاف حافظه، می‌توان اجازه داد که تعداد فرزندان هر گره متغیر باشد. در این صورت اندازه هر گره، بر اساس تعداد فرزندان آن تعیین می‌شود. در این حالت ساختار گره درخت را می‌توان به صورت زیر تعریف کرد که در آن، فرزندان هر گره در یک لیست پیوندی قرار می‌گیرند.

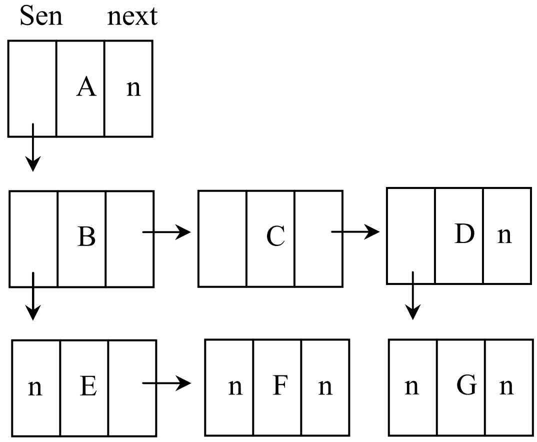
تعریف ساختار درخت

```
Struct tree Node {
        int info;
        Tree Node *son;
        Tree Node *next;
};
```

مثال۶,۱۵: شکل ۶,۲۵ (الف) را در نظر بگیرید. نمایش پیوندی این درخت عمومی در شکل ۶,۲۵ (ب) نمایش داده شده است.



الف



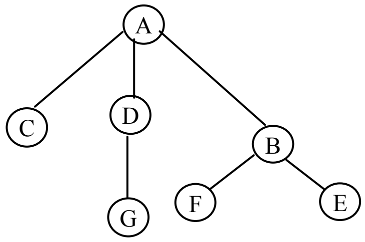
٦,١٩,٢ نمایش درخت عمومی به صورت درخت دودوئی

نمایش درخت عمومی با استفاده از درخت دودوئی، کارآمد و عملی است. هر درخت را می‌توان به صورت یک درخت دودوئی منحصر به فرد نمایش داد. با الگوریتم زیر می‌توان یک درخت عمومی را به درخت دودوئی معادل و منحصر به فردش تبدیل کرد:

در هر سطح کلیه گره‌های کنار هم که فرزند یک پدر هستند را به یکدیگر وصل کنید.
ارتباط کلیه گره‌ها به پدر را به جز اتصال سمت چپ‌ترین فرزند قطع کنید.
گره‌های متصل به هم در هر سطح افقی را ٤٥ درجه در جهت حرکت عقربه‌های ساعت بچرخانید.

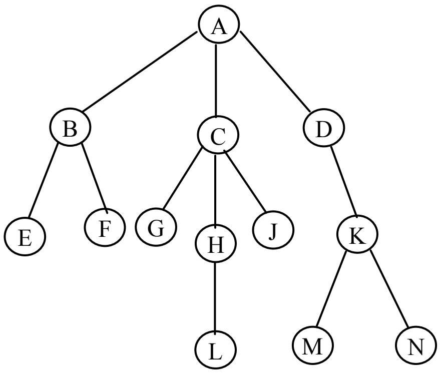
مثال٦,١٦ : درخت عمومی شکل (الف) را در نظر بگیرید. نمایش این درخت با استفاده از آرایه به صورت شکل (ب) می‌باشد.

	Data	Lchild	Sibling
1	A	2	0
2	B	0	3
3	C	0	4
4	D	6	0
5	G	0	6
6	F	0	7
7	E	0	0



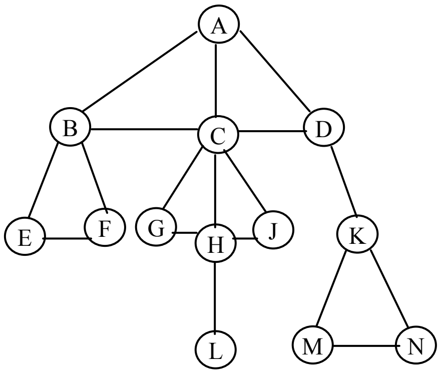
(الف)

مثال۶٫۱۷: درخت شکل زیر را در نظر بگیرید.

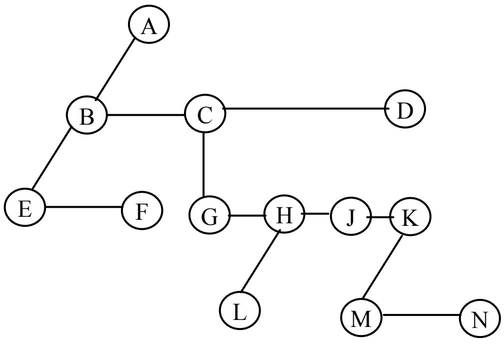


این درخت را به صورت دودوئی درمی‌آوریم.

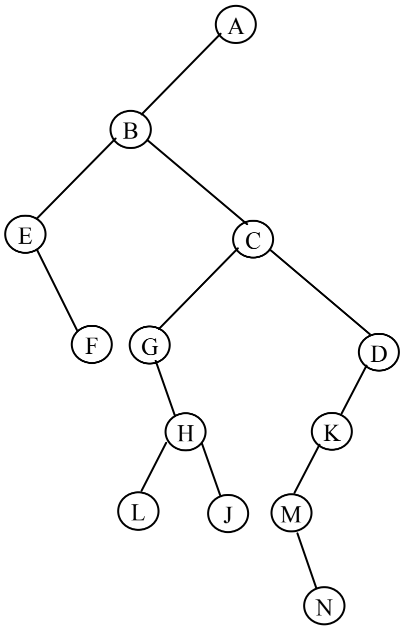
۱–ابتدا در هر سطح کلیه گره‌های کنار هم را به یکدیگر وصل می‌کنیم. (توجه کنید باید متعلق به یک پدر باشند)



۲–ارتباط کلیه گره‌ها به پدر خودش به جز سمت چپ‌ترین فرزند قطع می‌کنیم.



۳-گره‌هایی متصل به هم در سطح افقی را ٤٥ درجه در عقب عقربه‌های ساعت می‌چرخانیم.



٦,١٩,٣ نمایش پیوندی درخت

یکی از نکات مهم در درخت‌ها، گره است. هر گره درخت دودوئی شامل یک فیلد اشاره‌گر است. که به فرزندان چپ و راست اشاره می‌کند. اما هر گره در در درخت می‌تواند چندین اشاره‌گر داشته باشد. تعداد فرزندان هر گره درخت متفاوت است و می‌تواند خیلی زیاد یا خیلی کم باشد. یک روش این است که تعداد فرزندان یک گره را محدود کنیم به عنوان مثال می‌توانیم m در نظر بگیریم. در این صورت ساختار گره را به صورت زیر نمایش داد:

پیوند m	....	پیوند ۲	پیوند ۱	داده

چنین ساختاری را در C می‌توان به صورت زیر پیاده‌سازی کرد:

```
# define M  20
   struct tree Node{
        int info;
        tree Node *sons [M];
             };
```

اگر تعداد فرزندان هر گره ۲ یا حتی صفر باشد در این صورت فضای زیادی از حافظه به هدر می‌رود. فرض کنید، می‌خواهیم یک درخت mتایی (درختی با درجه m) را که حاوی n گره است نمایش دهیم. لم زیر نشان می‌دهد که به چقدر فضای حافظه اصراف می‌شود.

لم:

اگر T درخت mتایی با n گره باشد آنگاه $n(m-1)+1$ تعداد از $n*m$ پیوند تهی خواهند بود $(n \geq 1)$. بر اساس این لم، در یک دئرخت سه‌تایی، بیش از $\frac{2}{3}$ پیوندها تهی‌اند. میزان فضایی که به هدر می‌رود با افزایش درجه سوخت افزایش می‌یابد.

٦,١٩,٤ پیمایش درخت‌ها

پیمایش درخت‌ها به سه روش انجام می‌شود که عبارتند از:

روش میانوندی (inorder)

روش پسوندی (postrorder)

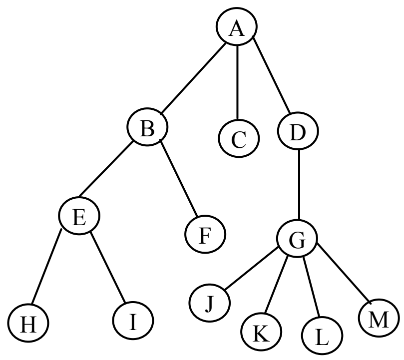
روش پیشوندی (preorder)

روش پیمایش inorder را می‌توان به صورت تابع زیر نوشت:

روش پیمایش inorder درخت

```
void inorder (tree * P)
        {
                if (P!=Null)
                  {
                 inorder (P→Son);
                  Printf("%d", P→ info);
                  Inorder (P→next);
                  }
        }
```

توابع مشابهی را می‌توان برای سایر پیمایش‌ها نوشت. درخت شکل پیمایش‌های مختلف آن را نشان می‌دهد.



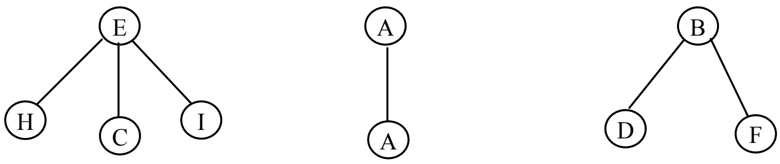
inorder :HIEFBCJKLMGDA
preorder: ABEHIFCDGJKLM
postrorder: HIEFBCJKLMGDA

٦,٢٠ جنگل‌ها

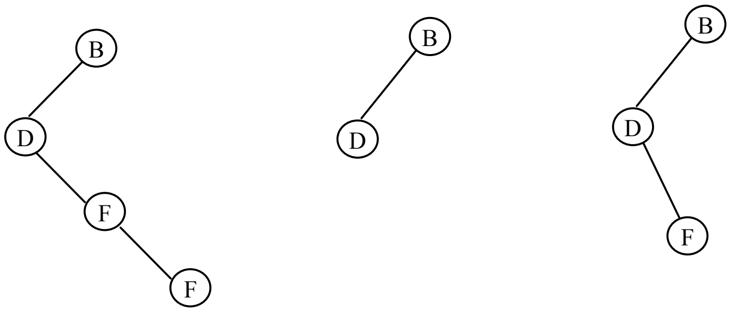
یک جنگل مجموعه‌ای مرتب از صفر یا چند درخت متمایز است. به عبارتی دیگر جنگل مجموعه‌ای $N \geq 0$ درخت مجهز است. مفهوم جنگل خیلی نزدیک به مفهوم درخت است. زیرا اگر ریشه درخت را حذف کنیم، جنگل به وجود می‌آید.

تبدیل جنگل به درخت دودوئی

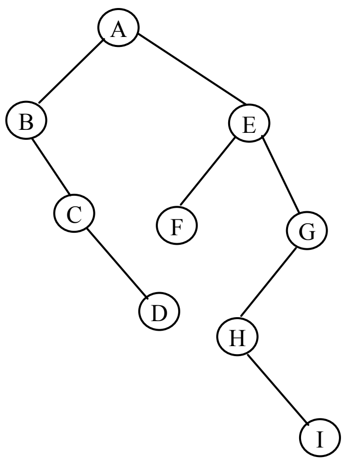
با تبدیل یک جنگل به درخت دودوئی را با استفاده از مثالی توضیح می‌دهیم. فرض کنید جنگلی از سه درخت شکل زیر تشکیل یافته باشد.



ابتدا این سه درخت را به درخت دودوئی خود تبدیل می کنیم.



سپس از چپ به راست ریشه هر درخت را به عنوان فرزند راست درخت سمت چپی در نظر می‌گیریم. یعنی:

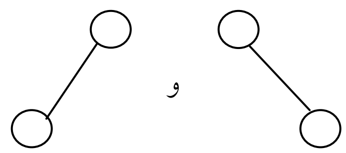


٦,٢١ شمارش درختهای دودوئی

به عنوان نتیجه‌گیری از این فصل، سه مسأله جدا از هم را که به طـور جـالبی دارای راه حل یکسانی هستند، بررسی می‌کنیم. می‌خواهیم تعداد درختهای متمایز با n گره، تعداد جایگشتهای مجزا اعداد ١ تا n توسط پشته و در نهایت تعداد ضرب‌هـای متمـایز n+1 ماتریس را مشخص کنیم.

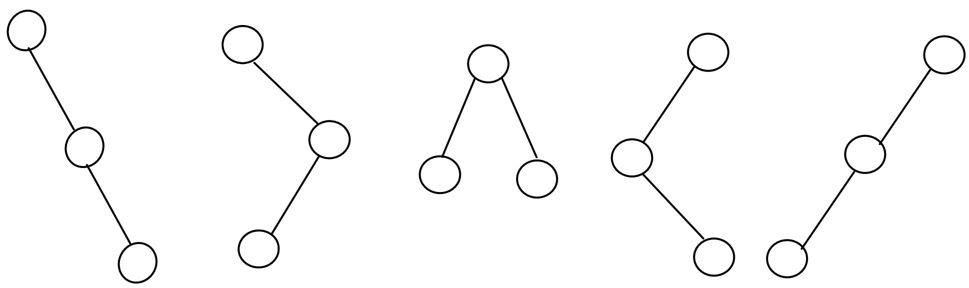
درختهای دودوئی متمایز

می‌دانیم اگر n=0 یا n=1 باشد، آنگاه فقط یک درخت دودوئی داریـم. امـا اگـر n=2 باشد، می‌توانیم دو درخت دودوئی ( ) را داشته باشیم.



درختهای دودویی مجزا با n=2

و اگر n=3 باشد می‌توان 5 درخت داشت.



درختهای دودوئی متمایز با n=3

به طور کلی با n گره چند درخت دودوئی را می‌توان سـاخت؟ قبـل از پاسـخ بـه ایـن سوال، دو مساله که معادل با این سوال هستند را بیان می‌کنیم.

جایگشت پشته

فرض کنید اعداد۱، ۲ و ۳ (n=3) به ترتیب از راست به چپ وارد پشته‌ای می‌شـوند ۵ خروجی زیر امکان‌پذیر است (از چپ به راست)

(1,2,3) , (1,2,3) , (2,1,3) , (2,3,1) , (3,2,1)

به عنوان مثال اگر ترتیب (1,2 , 3) امکانپذیر نیـست. (توجـه: مـا ایـن مـساله را طبـق قضیه‌ای در مسائل حل شده فصل پشته بیان کردیم.)

حال سوال اینجاست که با n عدد ورودی که به ترتیب وارد یک پـشته مـی‌شـوند چنـد حالت خروجی می‌توان داشت؟

ضرب ماتریس

موضوع دیگری که به طور جالبی به دو مسأله قبل مربوط است به صورت زیـر مطـرح می‌شود:

فرض کنید می‌خواهیم حاصلضرب چند ماتریس را به دست آوریم:

$$M_1 * M_2 * ... * M_n$$

با توجه به اینکه ضرب ماتریس‌ها شـرکت‌پـذیر اسـت، مـی‌تـوان عمـل ضـرب را بـا ترتیب‌های گوناگونی انجام داد. سوال اینجاست، چند راه برای انجام ایـن حاصلـضرب وجود دارد؟ برای مثال اگر n=3 باشد، دو حالت ممکن است:

$$(M_1 * M_2) * M_3$$
$$M_1 * (M_2 * M_3)$$

و اگر n=4 باشد، ۵ حالت مختلف وجود خواهد داشت:

$$((M_1 * M_3) * M_3) * M_4$$
$$(M_1 * (M_2 * M_3)) * M_4$$
$$M_1 * ((M_2 * M_3) * M_4)$$
$$(M_1 * (M_2 * (M_3 * M_4)))$$
$$((M_1 * M_2) * (M_3 * M_4))$$

می‌توان ثابت کرد که تعداد حالات فوق با هم مساویند و از فرمول زیر محاسبه می‌شوند:

$$b_n = \binom{1/2}{n+1}(-1)^n 2^{2n+1} = \frac{1}{n+1}\binom{2n}{n}$$

که در نتیجه داریم:

$$b_n = O(4^n / n^{3/2})$$

تمرین های فصل

۱) درخت های جستجوی دودوئی حاصل از درج کاراکترهای زیر را رسم کنید:

a. R , A , C , E , S

b. S , C , A , R , E

c. C , O , R , N , F , L , A , K , E , S

۲) درخت های جستجوی دودوئی حاصل از درج اعداد زیر را رسم کنید:

a. 1, 2 , 3 , 4 , 5

b. 5 , 4 , 1 , 7 , 8

c. 6 , 5 , 3 , 1

d. 4 , 1 , 5 , 3 , 6 , 2

۳) برای عبارات ریاضی زیر، درخت دودوئی رسم کنید. سپس با پیمایش درخت ، عبارات پیشوندی و پسوندی آن ها را چاپ کنید :

a. (A – B) – C

b. A – (B – C)

c. A / (B – (C – (D – (E – F))))

d. (A * C + (B – C) / D) * (E – F % G)

e. (A * B + D / (C – K))

٤) پیمایش های میانوندی و پسوندی درختی به صورت زیـر هـستند، درخـت را رسم کنید:

Inorder: GFHKDLAWRQPZ

Postorder: FGHDALPQRZWK

٥) پیمایش های میانوندی و پیشوندی درختی به صورت زیر هـستند، درخـت را رسم کنید:

Inorder: GFHKDLAWRQPZ

Perorder: ADFGHKLPQRWZ

٦) با مثالی نشان دهید که اگر نتایج پیمایش هـای پیـشوندی و پـسوندی درختـی معلوم باشد، نمی توان درخت منحصربه فردی را رسم کرد.

٧) الگوریتم بازگشتی و غیربازگشتی برای تعیین موارد زیر بنویسید:

الف) تعداد گره های در یک درخت دودوئی.

ب) مجموع محتویات کلیه گره ها در یک درخت دودوئی.

ج) عمق درخت دودوئی.

٨) الگوریتم بنویسید که تعیین کند آیا یک درخت دودوئی:

الف) دودوئی محض است.

ب) کامل است.

ج) تقریبا کامل است.

۹) دو درخت دودوئی وقتی شبیه به هم هستند که هر دو خالی باشند یا اگر غیر خالی هستند زیردرخت چپ آنها با هم مشابه و زیردرخت راست آنها نیز مشابه هم باشند. الگوریتمی بنویسید که مشخص کند دو درخت دودوئی مشابه هستند یا خیر.

۱۰) دو درخت دودوئی وقتی کپی هم هستند که هر دو خالی باشند یا اگر غیر خالی هستند زیردرخت چپ آنها با هم کپی و زیردرخت راست آنها نیز کپی هم باشند. الگوریتمی بنویسید که مشخص کند دو درخت دودوئی کپی هستند یا خیر.

۱۱) الگوریتمی بنویسید که اشاره گر به یک درخت دودوئی را پذیرفته و کوچکترین عنصر درخت را حذف کند.

۱۲) تابعی بنویسید که اشاره گر به یک درخت دودوئی و اشاره گر به یک گره دلخواهی از آن را پذیرفته و مشخص کند سطح آن گره در درخت چیست؟

۱۳) تابعی بنویسید که یک درخت دودوئی را با استفاده از پیمایش میانوندی و پسوندی ایجاد کند.

۱٤) درخت دودوئی فیبوناچی مرتبه n را به صورت زیر تعریف کنید: اگر n=0 یا n=1 درخت فقط حاوی یک گره است. اگر n>1 باشد درخت متشکل از یک ریشه، با درخت فیبوناچی مرتبه n-1 به عنوان زیردرخت چپ و درخت فیبوناچی مرتبه n-2 به عنوان زیردرخت راست است.

الف) تابعی بنویسید که اشاره گر به یک درخت فیبوناچی را برگرداند.

ب)آیا چنین درختی، دودوئی محض است.

ج)تعداد برگ های درخت فیبوناچی مرتبه n چیست؟

د) عمق درخت فیبوناچی مرتبه n چیست؟

۱۵) تابعی بنویسید که تعداد کل گره ها، تعداد برگها، تعداد گره های تک فرزندی، تعداد گره های دو فرزندی و تعداد شاخه های یک درخت را محاسبه و برگرداند.

۱٦) تابعی بازگشتی و غیر بازگشتی بنویسید که زیردرختهای چپ و راست یک درخت را جابجا کند.

۱۷) برنامه ای بنویسید که یک درخت عمومی را از وردی خوانده تبدیل به درخت دودوئی معادل کرده و پیمایش های این درخت حاصل را چاپ کند.

۱۸) برنامه ای بنویسید که جنگلی را از وردی خوانده تبدیل به درخت دودوئی معادل کرده و پیمایش های این درخت حاصل را چاپ کند.

۱۹) برنامه ای بنویسید که فرم پرانتزی یک درخت دودوئی را به صورت رشته از ورودی خوانده و آن را به فرم لیست پیوندی در حافظه پیاده سازی کند و سپس پیمایش های مختلف آن را چاپ کند.

۲۰) چند درخت با n گره وجود دارد؟

۲۱) چند درخت با گره n گره و حداکثر سطح m وجود دارند؟

۲۲) ثابت کنید که به سمت چپ ترین گره سطح n در یک درخت دودوئی محض تقریبا کامل، عدد 2^n نسبت داده می شود.

۲۳) ثابت کنید که اگر m فیلد اشاره گر در هر گره درخت عمومی برای اشاره به حداکثر m پسر وجود داشته باشد و تعداد گره های درخت برابر با n باشد، تعداد فیلدهای اشاره گر پسر که برابر null هستند برابر با n*(m-1)+1 است.

۲٤) چگونه می توان یک درخت عمومی را به درخت دودوئی محض تبدیل کرد. الگوریتم به زبان فارسی برای انجام این کار را بنویسید.

۲۵) درخت Heap حاصل از درج اعداد زیر را مرحله به مرحله رسم کنید:

a. 2 , 4 , 7 , 3 , 1 , 8
b. 1 , 2 , 3 , 5 , 6 , 9

c. 9 , 6 , 5 , 3 , 2 , 1
d. 3 , 5 , 6 , 4 , 2

۲۶) برنامه ای بنویسید که اعدادی را به ترتیب از ورودی خوانده و در یک درخت Heap (به صورت آرایه) ذخیره کند. سپس این آرایه را چاپ کند.

۲۷) الگوریتم های درج، حذف و جستجو در درخت AVL را بنویسید.

۲۸) برنامه ای بنویسید که تعدادی داده با درصد احتمال بروز هـر یـک را گرفتـه و کد هافمن معادل را چاپ کند.

۲۹) حروف a, d, b, c, e با جدول فراوانی زیر داده شده است. درخـت هـافمن این مساله را رسم کنید.

حروف	a	b	c	d	e
فراوانی	0.05	0.1	0.25	0.28	0.32

فصل هفتم

گرافها

اهداف

در پایان این فصل شما باید بتوانید:

- ✓ گراف را تعریف کرده و برخی از کاربردهای آن را نام ببرید.
- ✓ انواع گراف را بیان کرده و خصوصیات آنها را بیان کنید.
- ✓ پیمایش عمقی و عرضی یک گراف را بدست آورید.
- ✓ درخت پوشا را تعریف کرده و الگوریتم وارشال و پریم را برای بدست آوردن درخت پوشا به کار ببرید.

سوالهای پیش از درس

۱- به نظر شما با آیا گراف را می توان یک نوع داده جدید معرفی کرد؟

....

....

۲- چه مسائلی را می توان با گراف مدلسازی نمود.

....

....

۳- شما معمولا برای پیدا کردن کوتاهترین مسیر از شهری به شهر دیگر چگونه عمل می کنید؟

....

....

مقدمه

در این فصل یکی دیگر از ساختمان داده غیرخطی، موسوم به گراف را مورد بحث و بررسی قرار می‌دهیم. گراف یک ساختار کلی است که درخت حالت خاصی از آن است. گراف‌ها برای مدلسازی شبکه‌های کامپیوتری و سایر شبکه‌هایی مفید است که درآنها سیگنالها، پالس های الکتریکی و مانند اینها در مسیرهای گوناگونی از یک گره به گره دیگر جریان می‌یابند.

۷٫۱ چند اصطلاح نظریه گراف

تعریف گراف:

یک گراف G از دو مجموعه زیر تشکیل شده است:

- V مجموعه‌ای از عناصر که گره‌ها یا رأس‌ها نامیده می‌شود.
- E مجموعه‌ای از یالها طوری که هر یال e در E به وسیله یک جفت منحصر به فرد نامرتب (u,v) از گره‌ها در v مشخص می‌شود و آن را با e(u,v) نشان می‌دهند.

گره‌های همجوار: دو گروه x , y را در صورتی همجوار گویند که یالی از x به y وجود داشته باشد.

تعریف مسیر: یک مسیر P به طول n از یک گره u به گره v به صورت دنباله‌ای از n+1 گره تعریف می‌شود.

$$P = (v_0, v_1, v_2, \ldots, v_n)$$

به طوری که $u = v_o$ و $u_n = v$ است و v_{i-1} به ازای $i = 1,2,\ldots,n$ مجاور v_i می‌باشد.

مسیر P را بسته گویند اگر $v_o = v_n$، مسیر P را ساده گویند اگر تمام گره‌های آن متمایز باشد. یک دور یا حلقه مسیر ساده بسته‌ای به طول ۳ یا بیشتر است. گراف G را همبند گویند اگر بین هر دو گره آن مسیری وجود داشته باشد.

گراف G را کامل گویند اگر هر گره u در G مجاور هر گره v در G باشد. واضح است چنین گرافی همبند است.

گراف کامل با n گره، $\frac{n(n-1)}{2}$ یال دارد. گراف G را برچسب‌دار گویند هرگاه اطلاعاتی به یال‌های آن نسبت داده شود. به ویژه، گراف G را وزن‌دار گویند اگر به هر یال e در G یک مقدار عددی غیرمنفی w که وزن e نامیده می‌شود نسبت داده شود. در گراف G یال e را حلقه گویند اگر نقاط پایانی یکسانی داشته باشد..

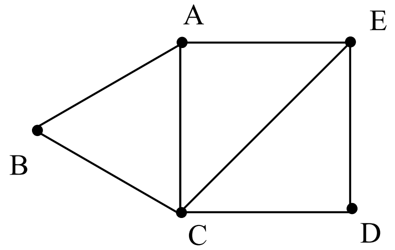
(مثال) شکل (الف) نمودار یک گراف همبند با 5 گره A , B ,C ,D , E و 7 یال

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

می‌باشد.

از B به E دو مسیر ساده به طول 2 وجود دارد. (B,C,E) , (B,A,E)

در یک گراف deg(A)=3 چون A به 3 یال تعلق دارد و به طور مشابه deg(C)=4 و deg(D)=2



گراف جهت‌دار

در گرافی که یالها جهت را نشان دهند گراف جهت‌دار گویند. درجه خروجی گره u در G که به صورت Outdeg(u) نمایش داده می‌شود تعداد یالهایی است که با u شروع می‌شوند یا از u خارج می شوند. به همین ترتیب درجه ورودی u که به صورت indeg(u) نمایش داده می‌شود نشانگر تعداد یالهایی است که در u به پایان می‌رسند. گره u یک گره منبع نامیده می‌شود اگر درجه خروجی مثبت داشته باشد و درجه ورودی‌اش صفر باشد. به همین ترتیب گره u یک چاه نامیده می‌شود اگر درجه خروجی صفر و درجه ورودی مثبت باشد.

گره u از گره v قابلی دسترس است اگر u به v یک مسیر جهت‌دار وجود داشته باشد.

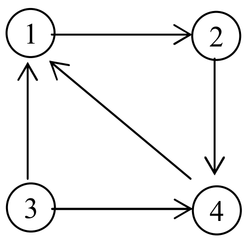
گراف جهت دار G را همبند یا همبند قوی گویند اگر برای هر زوج u , v از گره‌ها در G هم یک مسیر از u به v و هم یک مسیر از v به u وجود داشته باشد. از طرف دیگر، گراف G را همبند یک طرفه گویند اگر برای هر زوج u، v از گره‌ها در G یک مسیر از u به v یا یک مسیر از v به u وجود داشته باشد.

۷,۲ نحوه نمایش گراف‌ها در حافظه

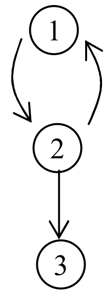
دو روش متداول و استاندارد برای نمایش گرافها وجود دارد. یک روش که نمایش ترتیبی G نامیده می‌شود به وسیله ماتریس مجاورت آن انجام می‌شود. روش دیگر نمایش پیوندی آن است.

ماتریس مجاورتی

گراف G=(V , E) را با n گره در نظر بگیرید. ماتریس همجواری این گراف یک آرایه دوبعدی n×n است که نام آن را T انتخاب می‌کنیم. اگر (v_i, v_J) یالی در گراف باشد (دقت کنید که در گراف جهت‌دار، این یال را به صورت $< v_i, v_J >$ نمایش می‌دهیم) آنگاه $T[i][J]=1$ خواهد بود. اگر چنین یالی در گراف موجود نباشد، آنگاه $T[i][J]=0$ خواهد بود. در شکل ۷,۱ چند گراف و ماتریس همجواری آنها نمایش داده شده است.



G1



G2

$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & & & & \\ 2 & & & & \\ 3 & & & & \\ 4 & & & & \end{array}\quad \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

$$\begin{array}{c|ccc} & 1 & 2 & 3 \\ \hline 1 & & & \\ 2 & & & \\ 3 & & & \end{array}\quad \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

شکل ۷,۱ چند گراف و ماتریس مجاورتی آنها

همانطور که در شکل ۷,۱ مشاهده می کنید، ماتریس همجواری مربوط به گراف بدون جهت ، متقارن است. یعنی به ازای هر $J \le n$ و $J \le i$ داریم $T[i][J] = T[J][i]$. علتش این است که اگر (v_i, v_J) یالی در گراف باشد آنگاه (v_J, v_i) نیز یالی در گراف است. بنابراین، اگر تعداد گره‌های گراف بدون جهت زیاد باشد، در نتیجه ماتریس همجواری آن بزرگ خواهد شد. می‌توانیم برای صرفه‌جویی در حافظه فقط ماتریس بالا مثلثی ماتریس همجواری را ذخیره کنیم.

با استفاده از ماتریس همجواری به سادگی و از مرتبه $o(1)$ می توان تعیین نمود که آیا بین دو گروه یالی وجود دارد یا خیر برای گراف بدون جهت، درجه هر گره مثل i برابر با مجموع عناصر سطر iام ماتریس همجواری است.

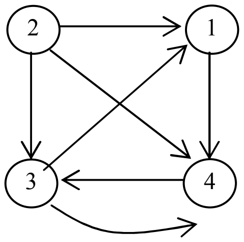
درجه گره i در گراف بدون جهت $= \sum_{J=1}^{n} T[i][J]$

در گراف جهت‌دار، برای به دست آوردن درجه خروجی گره‌ای مثل i عناصر سطر i را با هم جمع می‌کنیم و برای محاسبه درجه ، آن عناصر ستون i را با هم جمع می‌کنیم.

درجه خروجی گره i گراف جهت‌دار $= \sum_{J=1}^{n} T[i][J]$

درجه ورودی i در گراف جهت‌دار $= \sum_{J=1}^{n} T[J][i]$

مثال۷,۱ : گراف G شکل ۷,۲ را در نظر بگیرید.



شکل ۷,۲

A ماتریس مجاورتی گراف G چنین است:

$$A = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

توجه کنید تعداد 1ها در A برابر تعداد یال‌های گراف G است.

توانهای $A^1, A^2, A^3, ...$ از ماتریس مجاورتی A گراف G را در نظر بگیرید. فرض کنید:

$a_k(i,J)$ درایه iJام ماتریس A^k

ملاحظه می‌کنید که $a_1(i,J) = a_{iJ}$ تعداد مسیرهای به طول ۱ از گره v_i به v_j را به دست می‌دهد. می‌توان نشان داد که $a_2(i,J)$ تعداد مسیرهای به طول 2 از v_i به v_J است. در واقع مثال نتیجه کلی زیر را ثابت می‌کنیم.

قضیه: فرض کنید A ماتریس مجاورتی گراف G باشد. آنگاه $a_k(i,J)$ درایه I Jام ماتریس A^k تعداد مسیرهایی از v_i به v_J را به دست می‌دهد که طول k دارند.

بار دیگر ماتریس مجاورتی گراف شکل ۷,۲ را در نظر بگیرید. ماتریسهای $A^2,, A^3, A^k$ ماتریس A به شرح زیر است:

$$A^2 = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 2 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \end{pmatrix} \qquad A^3 = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 1 & 0 & 2 & 2 \\ 0 & 0 & 1 & 1 \end{pmatrix} \qquad A^4 = \begin{pmatrix} 0 & 0 & 1 & 1 \\ 2 & 0 & 2 & 3 \\ 1 & 0 & 1 & 2 \\ 1 & 0 & 1 & 11 \end{pmatrix}$$

یک مسیر از v4 به v1 به طول ۲ وجود دارد. در مسیر از v2 به v3 با طول 3 وجود دارد و سه مسیر از v2 به v4 با طول 4وجود دارد. فرض کنیم اکنون ماتریس B_r را به صورت زیر تعریف کرده‌ایم:

$$B_r = A + A^2 + A^3 + ... + A^r$$

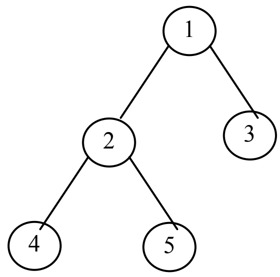
آنگاه درایه jام ماتریس B_r تعداد مسیرهای به طول r یا کمتر از r را از گره v_i به v_j محاسبه می کند.

نمایش گراف با استفاده از لیست پیوندی

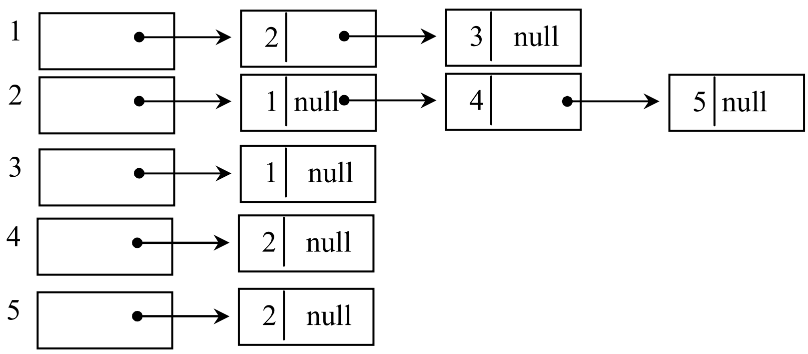
فرض کنید G یک گراف با m گره باشد. نمایش ترتیبی G در حافظه یعنی نمایش G به کمک ماتریس مجاورتی A دارای چند اشکال عمده است. قبل از هر چیز اضافه و حذف گره‌ها با این نمایش در G مشکل است. چون به علت قابل تغییر بودن اندازه A، گره‌ها را الزاماً باید از نو مرتب کرد، از این رو در ماتریس A تغییرات بسیار زیادی انجام می‌شود. علاوه بر این اگر تعداد یال‌ها o(m) یا o(mlogm) باشد، آنگاه ماتریس A، خلوت خواهد چون دارای صفرهای بسیار زیادی خواهد بود. از این رو مقدار زیادی از حافظه به هدر می‌رود. بنابراین G را معمولاً در حافظه به صورت پیوندی نمایش می‌دهند.

با این نمایش برای هر رأس از گراف G یک لیست وجود خواهد داشت. در هر لیست مشخصی مانند i گره‌های لیست حاوی رئوس مجاور از رأس i می‌باشد. هر لیست یک گره Head دارد که به ترتیب شماره‌گذاری شده‌اند و این امر دستیابی سریع به لیستهای مجاورتی برای رأس خاصی را به آسانی امکانپذیر می‌سازد.

مثال۷,۳ : گراف G شکل زیر را در نظر بگیرید.



لیست مجاورتی گراف به صورت زیر ترسیم می شود.



با توجه به شکل فوق درجه هررأس یک گراف بدون جهت را می‌توان به سادگی با شمارش تعداد گره‌های آن در لیست مجاورتی مشخص نمود و همچنین اگر تعداد رئوس گراف G برابر n باشد تعداد کل لبه‌ها در زمان o(n+e) تعیین می‌شود.

۷,۳ عملیات بر روی گراف‌ها

بعضی از عملیات مربوط به گراف را در این بخش به طور کامل مورد بحث و بررسی قرار خواهیم داد. این عملیات عبارتند از:

۱–پیمایش گراف‌ها　　　　۲–جستجوی در گراف

۳–اضافه کردن گره‌ای به گراف　　　　٤–حذف گره‌ای از گراف

پیمایش گراف‌ها

جستجو و پیمایش در گراف، ارتباط تنگاتنگی با یکدیگر دارند. به طوری که عمل جستجو می‌تواند با عملیات پیمایش انجام شود. با مفهوم پیمایش در درخت‌ها آشنا شدید و سه روش پیمایش را مورد بررسی قرار دادیم. در پیمایش درخت‌ها، چنانچه از ریشه درخت شروع کنیم، پیمایش کل درخت امکان‌پذیر خواهد بود، زیرا از ریشه درخت می‌توان به هر گره‌ای رسید. اما در یک گراف ممکن است نتوان از هر گره‌ای به گره دیگر رسید. لذا ممکن است از گره‌ای که به عنوان شروع استفاده می‌کنیم، به تمام گره‌های گراف نرسیم. تعریف پیمایشی که به ساختار گراف مربوط می‌شود، به سه دلیل پیچیده‌تر از پیمیاش درخت است:

۱– به طور کلی در گراف گرهی به عنوان اولیه گره وجود ندارد که عمل پیمایش از آن شروع می‌شود. در حالی که در درخت چنین گرهی موجود است. علاوه بر این، وقتی گره شروع مشخص شد و کلیه گره‌هایی که از آن طریق قابل دسترسی‌اند ملاقات شدند، ممکن است گره‌هایی در گراف باشند که پیمایش نشده باشند، زیرا با شروع از این گره قابل دسترسی نیستند. در درختها این حالت اتفاق نمی‌افتد.

۲– بین جانشین‌های (successor) یک گره، ترتیب خاصی وجود ندارد. بنابراین هیچ ترتیبی وجود ندارد که گره‌های جانشین یک گره، بر اساس آن پیمایش شوند.

۳– برخلاف گره‌های درخت، هرگره گراف ممکن است بیش از یک گره پیشین داشته باشد. اگر xها جانشین (فرزند) دو گره y و z باشد، x ممکن است بعد از y و قبل از z ملاقات شود. بنابراین ممکن است گره‌ای، قبل از یکی از گره‌های پیشین خود (پدر خود) ملاقات شود.

با توجه به این سه مورد تفاوت که بیان شد، الگوریتم‌های پیمایش گراف، سه ویژگی مشترک دارند:

۱- الگوریتم ممکن است طوری تهیه شود که پیمایش را از گره خاصی شروع کنیم یا گره‌ها را به طور تصادفی انتخاب نماید و پیمایش را از آن گره شروع کند. چنین الگوریتمی، بر اساس این که از کدام گره شروع به پیمایش می‌کند، ترتیب گوناگونی از گره‌ها را به خروجی می‌برد.

۲– به طور کلی پیاده سازی گراف، ترتیب ملاقات گره‌های جانشین یک گره را مشخص می‌کند. به عنوان مثال، اگر از ماتریس همجواری برای پیاده‌سازی گراف استفاده شود، شماره گذاری گره‌ها از 0 تا n-1، این ترتیب را مشخص می‌کند. اگر از پیاده‌سازی لیست همجواری استفاده شود، ترتیب یالها در لیست همجواری، ترتیب ملاقات گره‌های جانشین را تعیین می‌کند.

۳– اگر گرهی بیش از یک گره پیشین داشته باشد، ممکن است بیش از یک بار در پیمایش ظاهر شود. الگوریتم پیمایش باید بررسی کند که آیا گره قبلاً ملاقات شده است یا خیر. یک روش برای این کار بدین صورت است که:

برای هر گره یک نشانگر (Flag) در نظر گرفته می‌شود. این نشانگر می‌تواند مقادیر مختلفی را بپذیرد تا نشان دهنده وضعیت گره باشد. گره می‌تو.اند یکی از شرایط زیر را داشته باشد:

- حالت آماده (گره آماده است تا دستیابی شود) STATUS=1
- حالت انتظار (در صف یا پشته قرار دارد تا ملاقات شود) STATUS=2
- ملاقات شده STATUS = 3

ما حالت آماده را با 1، حالت انتظار با 2 و حالت ملاقات شده را با 3 مشخص می‌کنیم.

اگر G=(V,E) یک گراف و x گره‌ای در این گراف باشد، در پیمایش گراف G لازم است مشخص می‌کنیم چه گره‌هایی از طریق گره x قابل دستیابی‌اند. برای این کار از دو روش استاندارد استفاده می‌شود:

- جستجوی عمقی یا پیمایش عمقی (depth – first search = dfs)
- جستجوی عرضی با پیمایش عرضی (breadth – first search = bfs)

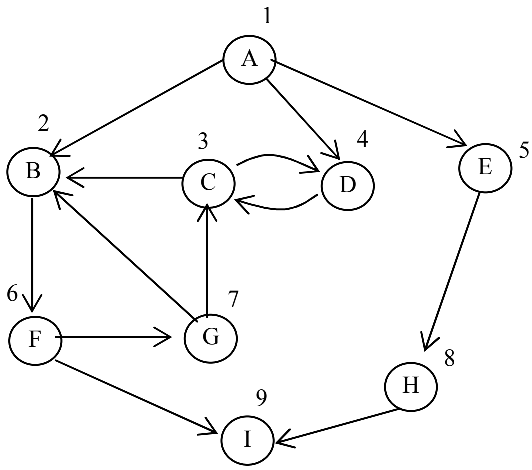
۷,٤ جستجوهای عرضی

ایده کلی جستجوی عرضی بدین صورت است که کار را با گره آغاز به شرح زیر شروع می کنیم. نخست گره آغازین A را ملاقات می‌کنیم. آنگاه تمام همسایه‌ها یا گره‌های مجاور A را ملاقات می‌کنیم. سپس تمام همسایه‌های گره‌های مجاور A را ملاقات می‌کنیم و الی آخر. لازم است اطمینان داشته باشیم که هیچ گره‌ای بیشتر از یک بار پردازش نشود. این کار با استفاده از یک صف جهت نگه داشتن گره‌هایی که در انتظار پردازش بسر می‌برند و با استفاده از فیلد STATUS که وضعیت جاری هر گره را به ما اطلاع می‌دهد انجام می‌شود.

الگوریتم به شرح زیر است:

الگوریتم جستجوی عرضی
الگوریتم: این الگوریتم جستجوی عرضی را با شروع از گره آغازین A روی یک گراف G اجرا می‌کند. ۱- تمام گره‌هایی که در حالت آماده (STATUS=1) هستند مقدار اولیه می‌دهد. ۲- گره آغازین A را در صف (QEUEUE) قرار دهید و وضعیت آن را به حالت انتظار (STATUS=2) تغییر دهید. ۳- مرحله‌های ٤ و ٥ را تا وقتی صف خالی نشده تکرار کنید. ٤- گره N را از ابتدای صف حذف کنید. N را پردازش کنید. و وضعیت آن را به حالت پردازش شده (STATUS=3) تغییر دهید. ٥- تمام همسایه‌های N را به انتهای صف اضافه کنید که در حالت آماده هستند (STATUS=1) و وضعیت آنها را به حالت انتظار (STATUS=2) تغییر دهید.

مثال۷,٤: گراف G شکل ۷,۳ (الف) را در نظر بگیرید و لیست مجاورتی گره ها در شکل (ب) نشان داده شده است.



اکنون الگوریتم جستجوی عرضی را بر روی گراف شکل (ب) اجرا می‌کنیم.

توضیح: فیلد وضعیت تمام گره‌ها در ابتدا ۱ است که به معنای حالت آماده است.

۱- A را در صف قرار دهید و فیلد وضعیت آن را به ۲ تغییر دهید.

۲- A را از صف حذف کنید و آن را در خروجی بنویسید و فیلد وضعیت آن را به ۳ تغییر دهید.

۳- گره‌های همجوار A، یعنی E , D, B را در صف قرار دهید و فیلد وضعیت آن را به ۲ تغییر دهید.

٤- B را از صف حذف کنید، در خروجی بنویسید و فیلد وضعیت آن را به ۳ تغییر دهید.

۵- گره‌های همجوار B را که در حالت آماده‌اند، یعنی F را در صف قرار دهید و فیلد وضعیت آن را بـه ۲ تغییر دهید.

٦- D را از صف حذف کنید، در خروجی بنویسید و فیلد وضعیت آن را به ۳ تغییر دهید.

۷- گره‌های همجوار D را که در حالت آماده‌اند، یعنی C , H را در صف قرار دهید و فیلد وضعیت آن را به ۲ تغییر دهید.

۸- گرهE را از صف خارج کرده و در خروجی بنویسید و فیلد وضعیت آن را به ۳ تغییر دهید.

۹- گره‌های همجوار E را که در حالت آماده قرار دارند در صف قرار دهید. گره همجوار E گره H اسـت که قبلاً در صف قرار گرفته است.

۱۰.F را از صف خارج کرده و در خروجی بنویسید و فیلد وضعیت آن را به ۳ تغییر دهید.

۱۱.گره‌های همجوار F را که در حالت آماده قرار دارند در صف قرار دهید و حالت آن را به ۲ تغییر دهیـد این گره‌ها G, I هستند.

۱۲.C را از صف خارج کرده و در خروجی بنویسید و فیلد وضعیت آن را به ۳ تغییر دهید.

۱۳.گره‌های همجوار C گره های Dو G هستند که قبلاً بررسی شده‌اند.

۱٤.گره H را از صف خارج کرده و در خروجی بنویسید و فیلد وضعیت آن را به ۳ تغییر دهید.

۱۵.گره H همجوار ندارد که در صف قرار گیرد.

۱٦.گره G را از صف خارج کرده و در خروجی بنویسید و وضعیت آن را به ۳ تغییر دهید.

۱۷.گره‌های همجوار G قبلاً بررسی شده‌اند.

۱۸.I را از صف خارج کرده و در خروجی بنویسید و وضعیت آن را به ۳ تغییر دهید.

۱۹.چون صف خالی شد، الگوریتم به پایان می‌رسد و پیمایش عرضی به صورت زیر خواهد بود:

ABDEFCHGI

الگوریتم زیر پیاده‌سازی bfs را به صورت غیربازگشتی و با استفاده از صف q نشان می‌دهد.

الگوریتم پیمایش عرضی

```
void bfs (int v)
  {
        visited [v]=true;
        addq(q,v);
        while not empty  euque (q)
           {
               delq (q,v);
               for all node W adjacent to V do
                        {
                          addq (q,w);
                          visited [w]= true;
```

}
}
}

در الگوریتم فوق V نشان دهنده گره آغازین می‌باشد.

تحلیل bfs

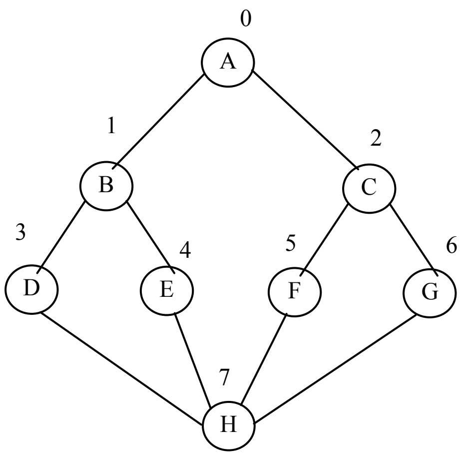
اگر گراف G توسط لیست مجاورتی ارائه شود، می‌توانیم رئوس مجاور با رأس v را با دنبال کردن زنجیری از اتصالات مشخص کنیم. از آنجا که در الگوریتم bfs هر گره در سیستمهای مجاورتی یک بار ملاقات می‌شود، کل زمان جستجو o(e) (e تعداد لبه‌ها) است اگر G توسط ماتریس مجاورتی نمایش داده شده، زمان لازم برای تعیین همه رئوس مجاور به v، o(n) است از آنجا که حداکثر n رأس وجود دارد کل زمان $O(n^2)$ خواهد شد.

۷٫۵ جستجوی عمقی

ایده کلی الگوریتم جستجوی عمقی بدین صورت است که کار را با گره آغازین A شروع می‌کنیم، آنگاه یک همسایه A را پردازش می‌کنیم سپس همسایه همسایه A و الی آخر را پردازش می‌کنیم و یا به بیان دقیق‌تر:

در آغاز رأس A را ملاقات می‌کنیم. بعد رأسی مانند v را که قبلاً ملاقات نشده است و مجاور A است را انتخاب کرده و روش جستجوی عمقی را با آن ادامه می‌دهیم. و در نهایت به رأسی مانند w می‌رسیم که فاقد هرگونه رأس غیرملاقات شده در لیست مجاورتی می‌باشد در این مرحله رأسی از پشته انتخاب شده و فرایند فوق تکرار می‌گردد.

با استفاده از مثال ساده‌ای مراحل کار را به صورت صوری و نه الگوریتمیک بیان می‌کنیم. فرض کنید گراف ساده زیر نشان داده شده است:



ابتدا به گره‌ها از بالا به پایین و از چپ به راست شماره می‌دهیم. سپس گره با شماره صفر را ملاقات کرده و آن را در خروجی می‌نویسیم. بعد به سراغ همسایه آن گره (گره مجاور) با عدد کوچکتر می‌رویم (گره با شماره ۱) آن را ملاقات کرده و در خروجی می‌نویسیم. سپس به سراغ همسایه ۱ با عدد کوچکتر (گره با شماره ۳) می‌رویم (متذکر می‌شویم که گره با شماره صفر نیز همسایه ۱ است ولی قبلاً آن را ملاقات کرده‌ایم). بعد به سراغ گره با شماره ۷، بعد به سراغ همسایه گره با شماره ۷ با عدد کوچکتر که قبلاً ملاقات نکرده‌ایم (گره با شماره ٤) می‌رویم و آن را نیز در خروجی چاپ می‌کنیم. هنگامی که بر روی گره با شماره ٤ هستیم مشاهده می‌کنیم که همه همسایه‌های آن قبلاً پیمایش شده است. لذا یک مرحله به عقب برمی‌گردیم و روی گره با شماره ۷ می‌ایستیم. حال به سراغ گره ۵ بعد ۲ و بعد ٦ می‌رویم.

Dfs=0,1,3,7,4,5,2,6

اکنون به بیان الگوریتم جستجوی عمقی می‌پردازیم. الگوریتم شبیه جستجوی عرضی است با این تفاوت که به جای صف از یک پشته استفاده می‌کنیم. مجدداً از فیلد STATUS استفاده می‌کنیم که به ما وضعیت جاری یک گره را اعلام می‌کند. الگوریتم به صورت زیر است:

الگوریتم: این الگوریتم جستجوی عمقی را با شروع از گره آغازین A روی یک گراف G اجرا می‌کند.

الگوریتم جستجوی عمقی
۱- تمام گره‌هایی را که در حالت آماده STATUS=1 هستند مقدار اولیه می‌دهد. ۲- گره آغازین A را در پشته push کنیم و وضعیت آن را به حالت انتظار STATUS=2 تغییر دهید. ۳- مرحله‌های ٤ و ۵ را تا وقتی پشته خالی نشده است تکرار کنید. ٤- گروه N بالای پشته را pop کنید و آن را پردازش کنید و وضعیت آن را به STATUS=3 تغییر دهید. ۵- تمام همسایه گره N را به داخل پشته Push کنید که همچنان در حالت آماده STATUS=1 هستند و وضعیت آنها به حالت انتظار (STATUS=2) تغییر دهید.

الگوریتم زیر این روش پیمایش را نشان می‌دهد.

الگوریتم پیمایش عمقی

```
void dfs (int v)
  {
        printf (Data (v));
```

```
        visited [v]=ture;
        for (each vertex wadjacent to v) do
                if (not visited [w])
                        dfs(w)
         }
```

تحلیل dfs

دراین روش نیز مانند روش عرضی، مرتبه اجرائی با استفاده از لیست همجواری برابر با O(e) و با استفاده از ماتریس همجواری برابر با $O(n^2)$ خواهد بود.

۷,۶ گراف‌های متصل

با استفاده از دو روش جستجوی مطرح شده، می‌توان عملکردهای قابل توجه دیگری بر روی گراف اعمال نمود. برای بیان این هدف، می‌توانیم به مساله تعیین اینکه آیا یک گراف بدون جهت متصل است یـا خیـر، می‌توانیم مساله فوق را با فراخوانی dfs(0) یا bfs(0) در تعیین اینکه آیا رأسی بدون اینکه ملاقـات شـود باقی مانده است یا خیر، حل نماییم.

۷,۷ درخت‌های پوشا و درخت پوشای کمینه

فرض کنید می خواهیم چند شهر معین را با جاده به هم وصل کنیم، به طوری که مردم بتوانند از هر شـهر، به شهر دیگر بروند. اگر محدودیت‌های بودجه‌ای در کار باشد، ممکن است طراح بخواهـد ایـن کـار را بـا حداقل جاده‌کشی انجام دهد. در این بخش می‌خواهیم الگوریتمی ارائه دهیم که ایـن مـسئله و مـسئله‌هـای مشابه را حل کنند. برای این منظور باید دو موضوع را یادآوری کنیم:

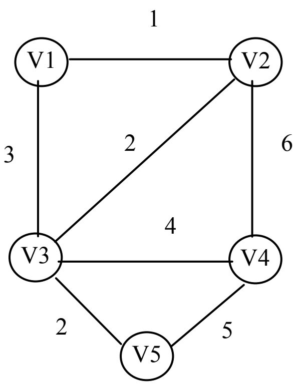
۱- گراف‌های چرخه‌دار و بدون چرخه

۲- تفاوت گراف و درخت

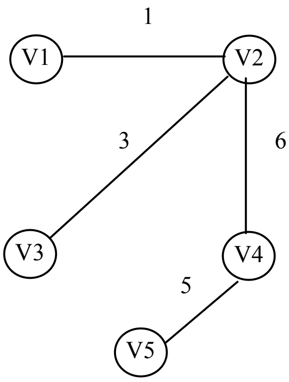
اگر در گرافی (بدون جهت و جهت‌دار)، مسیری گره‌ای را به خـودش وصـل کنـد، مـی‌گـوییم آن گـراف چرخه‌دار است و اگر چنین مسیری وجود نداشته باشد، گوییم این گراف بدون چرخه است. درخـت یـک گراف بدون جهت متصل و بی‌چرخه است.

می‌توانیم از گراف بدون جهت و وزن‌دار G، یالهایی را حذف کنیم به طوری که زیر گراف به دسـت آمـده متصل باقی بماند و حاصل جمع وزن‌های یال‌های باقیمانده را کمینه کند. این مـسئله کاربردهـای متعـددی دارد. به عنوان مثال در ارتباطات راه دور می‌خواهیم حداقل طول کابل و در لوله‌کشی مـی‌خـواهیم حـداقل مقدار لوله مصرف شود. یک زیرگراف با حداقل وزن باید درخت باشد.

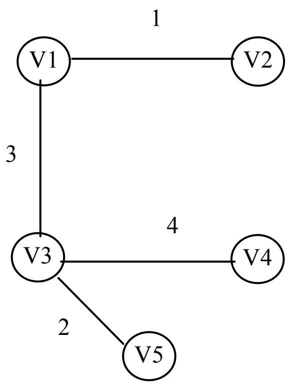
درخت پوشای گراف G، زیرگراف متصلی است که حاوی تمام گـره‌هـای G بـوده، درخـت باشـد (فاقـد چرخه باشد). درختهای شکل (ب) و (ج) درختهای پوشا برای گراف شکل (الف) هستند.



(الف) گراف متصل، موزون و بدون جهت G



(ب) درخت پوشا برای G



(ج) درخت پوشای کمینه برای G

درخت پوشای کمینه، درخت پوشایی است که حداقل وزن را داشته باشـد. بـه عنـوان مثـال، وزن درخـت پوشای شکل (ج) برابر با ۱۰ و وزن درخت‌پوشای شکل (ب) برابر با ۱۵ است. درخت شکل (ج) یـک درخت‌پوشای کمینه برای گراف است. توجه داشته باشید که یک گراف ممکن است بیش از یـک درخـت پوشای کمینه داشته باشد.

در بخش بعدی برای به دست آوردن درخت پوشای کمینه یک گراف – الگوریتم‌هـای وارشـال و پـریم را بررسی خواهیم کرد.

روش ما برای تعیین درخت پوشا با حداقل وزن و هزینه باید سه شرط زیر را داشته باشد:

- ✓ فقط باید از لبه‌های داخل گراف استفاده کند.
- ✓ باید دقیقاً از n-1 لبه استفاده کند (n تعداد رأسها)
- ✓ نباید از لبه‌هایی که ایجاد حلقه می‌کنند استفاده کنیم.

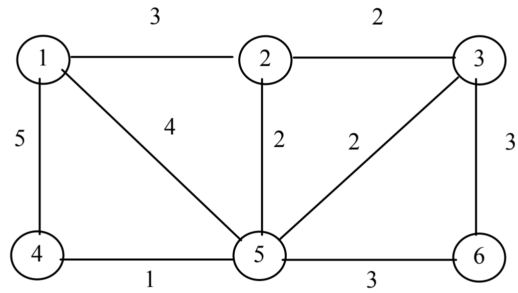
۷,۸ الگوریتم وارشال برای ساخت درخت پوشای کنید.

در این الگوریتم درخت پوشای کمینه، لبه به لبه ساخته می‌شود. لبه‌های گراف بر حسب وزن به ترتیب صعودی مرتب می‌شوند. لبه جدید وقتی به درخت T اضافه می‌شود که با لبه های موجود در T چرخه‌ای ایجاد نکند. این الگوریتم را می‌توان به صورت زیر نوشت:

الگوریتم وارشال برای تهیه درخت پوشای کمینه

الگوریتم وارشال برای تهیه درخت پوشای کمینه
۱– لبه‌ها را به ترتیب صعودی، از کمترین وزن به بیشترین وزن مرتب کنید. ۲– لبه‌های مرتب شده را به ترتیب به درخت T اضافه کنید. اگر با افزودن این لبه، چرخه ایجاد شود، از آن لبه صرفنظر کنید و لبه بعدی را بررسی نمائید. ۳– مرحله ۲ را آنقدر تکرار کنید تا به انتهای لیست یال‌های مرتب شده برسید.

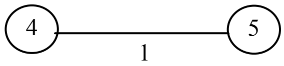
مثال۷,۵ : گراف شکل (الف) را در نظر بگیرید. می‌خواهیم با اعمال الگوریتم وارشال بر روی گراف، درخت پوشای کمینه آن را به دست آوریم.



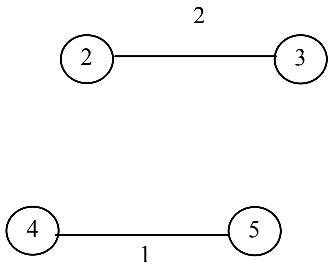
در مرحله اول لبه‌ها را به صورت صعودی مرتب می‌کنیم:

لبه هایی با وزن ۱	(4,5)
لبه‌هایی با وزن ۲	(5,3) (2,5) (2,3)
لبه‌هایی با وزن ۳	(5,6) (3,6) (1,2)
لبه هایی با وزن ٤	(1,5)
لبه‌هایی با وزن ۵	(1,4)

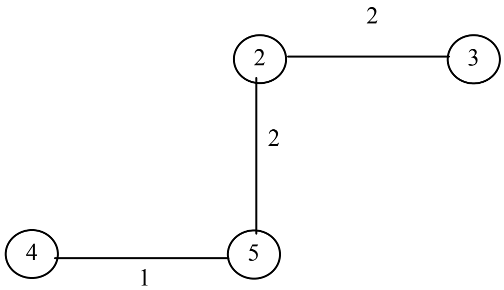
۱–ابتدا لبه‌ی با وزن ۱ را به درخت اضافه می‌کنیم.



۲–یکی از لبه‌ای با وزن ۲ را به درخت اضافه می‌کنیم.

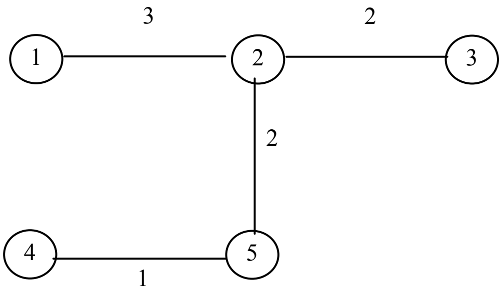


۳–یکی دیگر از لبه‌هایی با و.زن ۲ را به درخت اضافه می‌کنیم.

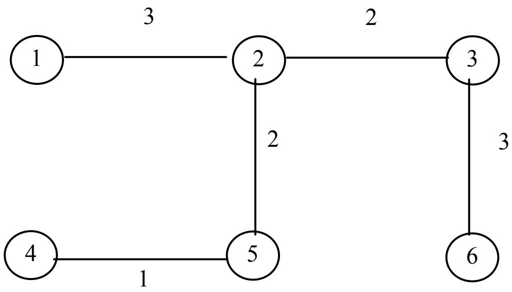


٤–لبه $(5,3)$ را که وزن ۲ دارد به درخت اضافه نمی‌کنیم چرا که در این صورت یک چرخه درست خواهد شد.

٥–لبه‌ای با وزن ۳ $(1,2)$ را به درخت اضافه می‌کنیم.

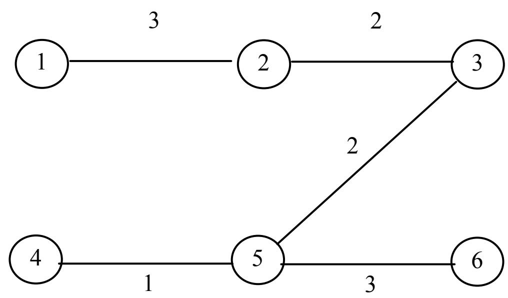


٦–لبه $(3,6)$ را که دارای وزن ۳ به درخت اضافه می‌کنیم.



۷–هیچ کدام از لبه‌های (5,6) , (1,5) , (1, 4) را نمی‌توان به سرعت اضافه نمود چون تشکیل چرخه می‌دهند.

بنابراین وزن درخت پوشانی کمینه برابر با ۱۱ خواهد شد. قابل توجه است که این درخت پوشا به دلیل داشتن لبه‌هایی با وزن یکسان منحصر به فرد نخواهد بود و یکی دیگر از درخت پوشای کمینه با وزن ۱۱ برای گراف شکل (الف) به صورت زیر خواهد بود.



۷,۹ الگوریتم پریم برای تعیین درخت پوشای کمینه

پریم نام مخترع این الگوریتم است. در این الگوریتم V را مجموعه‌ای از رأس‌های گراف در نظر می‌گیریم. این الگوریتم با مجموعه‌ای تهی از یال‌ها به نام F و زیرمجموعه‌ای از رئوس به نام T آغاز می‌شود. زیرمجموعه T حاوی یک رأس دلخواه است . اگر به عنوان مثال T حاوی یک رأس برابر با $\{v_1\}$ باشد نزدیکترین رأس به T، رأسی در V-T است که توسط یالی با وزن کمینه در T متصل است. این رأس به T و یال $\{v_1, v_2\}$ به F اضافه می‌شود. این روند آنقدر ادامه می‌یابد تا $T = V$ شود. الگوریتم پریم نیز همانند الگوریتم وارشال در هر زمان یک لبه از درخت پوشای کمینه را می‌سازد ولی در الگوریتم پریم در هر مرحله، مجموعه لبه‌های انتخاب شده تشکیل یک درخت می‌دهند و در الگوریتم وارشال، مجموعه لبه‌های انتخاب شده در هر مرحله تشکیل یک جنگل می‌دهند.

تعیین درخت پوشای کمینه

۱–ابتدا رأس ۱ به طور دلخواه انتخاب می‌شود.

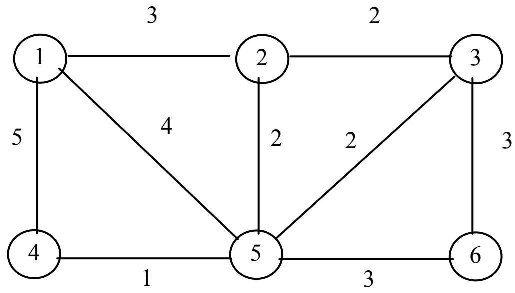
۲– رأس 2 انتخاب می‌شود زیرا نزدیکترین رأس به مجموعه {1} است.

۳– رأس 5 انتخاب می‌شود زیرا نزدیکترین رأس به {1,2} است.

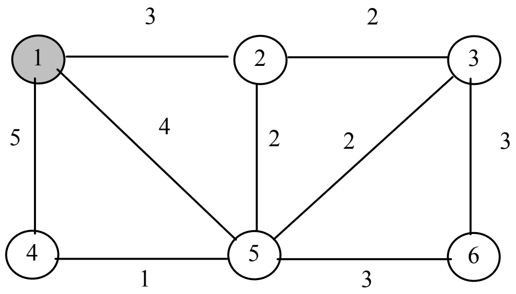
٤– رأس 4 انتخاب می‌شود. زیرا نزدیکترین رأس به {1,2,5} است.

۵– رأس 3 انتخاب می‌شود زیرا نزدیکترین رأس به مجموعه {1,2,5,4} است.

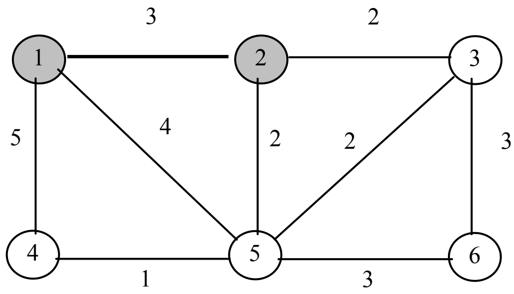
٦– رأس 6 انتخاب می‌شود زیرا نزدیکترین رأس به مجموعه {1,2,5,4,3} است.


3
2
1
2
3
5
4
2
2
3
4
5
6
1
3

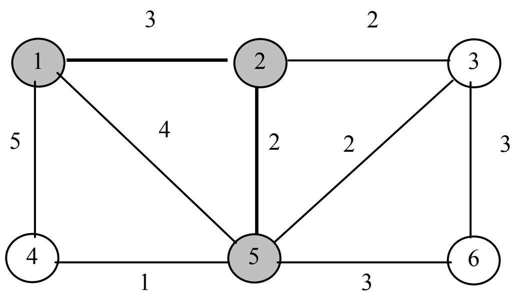
(۱)


3
2
1
2
3
5
4
2
2
3
4
5
6
1
3

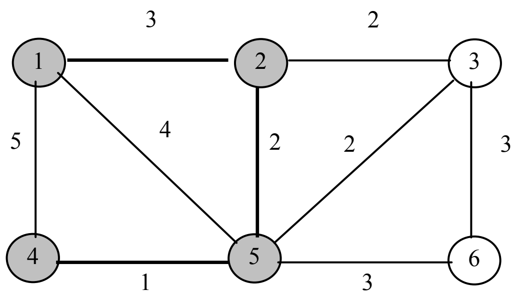
(۲)


3
2
1
2
3
5
4
2
2
3
4
5
6
1
3

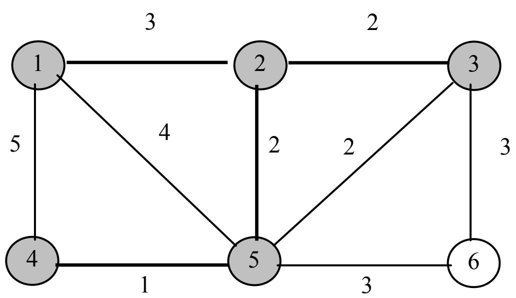
(۳)


3
2
1
2
3
5
4
2
2
3
4
5
6
1
3

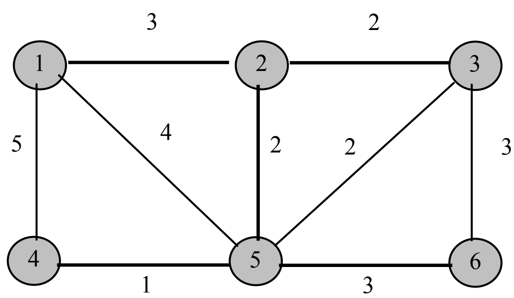
(٤)


3
2
1
2
3
5
4
2
2
3
4
5
6
1
3

(٥)


3
2
1
2
3
5
4
2
2
3
4
5
6
1
3

(٦)


3
2
1
2
3
5
4
2
2
3
4
5
6
1
3

تمرین های فصل

۱) روالهایی بنویسید که با توجه به یک ماتریس همجواری و دو گره از گراف، موارد زیر را محاسبه کند:

الف) تعداد مسیرهای با طول معین بین آنها

ب) تعداد کل مسیرهای موجود بین آنها

۲) برنامه ای بنویسید که گرافی را دریافت کرده و مشخص کند آن گراف متصل است یا خیر.

۳) برنامه ای بنویسید که الگوریتم پریم را برای ساخت درخت پوشا پیاده سازی کند.

٤) برنامه ای بنویسید که تعداد گره ها و اینکه هر گره مجاور کدام گره هاست را از کاربر گرفته و ماتریس مجاورتی آن را چاپ کند. ورودی برنامه به صورت مجموعه های V و E باشد. برنامه باید برای گراف های جهت دار و بدون جهت کار کند.

٥) برنامه ای بنویسید که گرافی را خوانده و جستجوی عمقی آن را چاپ کند. گراف یکبار به صورت لیست مجاورتی و بار دیگر به صورت ماتریس مجاورتی باشد.

٦) برنامه ای بنویسید که گرافی را خوانده و جستجوی ردیفی آن را چاپ کند. گراف یکبار به صورت لیست مجاورتی و بار دیگر به صورت ماتریس مجاورتی باشد.

۷) برنامه ای بنویسید که گرافی وزن دار را خوانده و با استفاده ازالگوریتم پریم درخت پوشای مینممم آن را چاپ کند.

۸) الگوریتمی بنویسید که کوتاهترین مسیرهای راس 0 تا تمام رئوس دیگر گراف را به ترتیب غیر نزولی پیدا نماید.

۹) با توجه به گراف کامل با n راس، نشان دهید که حداکثر تعداد مسیرهای بین رئوس برابر $O((n-1)!)$ می باشد.

۱۰) نشان دهید که اگر T یک درخت پوشا برای گراف بدون جهت G باشد، آنگاه اضافه کردن یک لبه مانند e موجب ایجاد یک حلقه منحصر بفرد می گردد.

۱۱) با توجه به گراف کامل با n راس، نشان دهید که تعداد درختهای پوشای حداقل، برابر با $2^{n-1}-1$ می باشد.

۱۲) برای گراف بدون جهت G با n راس، ثابت کنید که موارد زیر یکسان و معادل هستند:

الف) G یک درخت می باشد.

ب) G متصل می باشد، اما اگر هر یک از لبه های آن حذف شود گراف حاصل متصل نمی باشد.

ج) G فاقد حلقه بوده و دارای n-1 لبه می باشد.

د) برای هر راس مجزا تنها یک مسیر ساده از u به v وجود دارد.

فصل هشتم

مرتب سازی

اهداف

در پایان این فصل شما باید بتوانید:

- ✓ مفهوم مرتب سازی را بیان کرده و دلیل استفاده از آن را بیان کنید.
- ✓ مرتب سازی را با جستجو مقایسه کرده و تحلیل کنید چه موقعی از کدامیک استفاده می شود.
- ✓ کاربردهای مرتب سازی را بیان کنید.
- ✓ روشهای مرتب سازی را تشریح کرده و در مورد مرتبه زمانی آنها بحث کنید.
- ✓ روشهای مرتب سازی را با توجه به شرایط مساله با یکدیگر مقایسه کنید؟

سوالهای پیش از درس

۱- اگر بخواهید یک شماره تلفن از دفترچه شماره تلفن پیدا کنید چه کارهایی را انجام می دهید؟

....

....

۲- اگر بخواهید دفتر شماره تلفن خود را مرتب کنید چه کارهایی را انجام می دهید؟

....

....

۳- آیا برای اینکه به اطلاعات به صورت مرتب دسترسی پیدا کرد، مرتب سازی تنها راه حل آن است؟

....

....

مقدمه

مرتب کردن و جستجوی اطلاعات از عملیات اساسی و اصلی در علم کامپیوتر است. مرتب کـردن عبـارت است از عمل تجدید آرایش داده‌ها با یک ترتیب مشخص، مثلاً برای داده‌های عددی به ترتیب صعودی یـا نزولی اعداد یا برای داده‌های کاراکتری ترتیب الفبایی آنها. جستجو کردن عبارت است از عمل پیـدا کـردن مکان یک عنصر داده شده در بین مجموعه‌ای از عناصر.

مرتب کردن و جستجوی اطلاعات، اغلب روی یک فایل از رکوردها به کار می‌رود، از ایـن رو لازم اسـت چند اصطلاح استاندارد را یادآوری کنیم. هر رکورد در یک فایل می‌تواند چند فیلد داشـته باشـد امـا فیلـد ویژه‌ای وجود دارد که مقدارهای آن به طور منحصر به فردی، رکوردهای داخل فایل را معین می‌کند. چنین فیلدی یک کلیه اولیه یا اصلی نامیده می‌شود، مرتب کردن فایل معمولاً به مرتب کردن نسبت به کلید اولیـه خاصی گفته می‌شود و جستجوی اطلاعات در فایل به جستجوی رکـورد بـا مقـدار کلیـدهای معـین گفتـه می‌شود.

۸،۱ مرتب کردن

فرض کنید A یک لیست عنصری از $A_1, A_2, \ldots, A_n$ در حافظه باشد، منظور از مرتب کردن A عمـل تجدید آرایش محتوای A است به طوری که با ترتیب صعودی (عددی یا فرهنگ لغتی) یـا نزولـی باشـند، یعنی به طوری که:

$$A_1 \le A_2 \le A_3 \ldots \le A_n$$

تعریف: اگر عمل مرتب‌سازی بر روی رکوردهای موجود در حافظه انجام شود، مرتب‌سازی داخلـی و اگـر بر روی رکوردهای موجود در حافظه جانبی (مانند دیسک ها)صورت گیـرد، مرتـب‌سـازی خـارجی نـام دارد. در این کتاب مرتب‌سازی‌های داخلی مورد بحث وبررسی قرار خواهد گرفت.

تعریف: ممکن است دو یا چند رکورد دارای کلید یکسانی باشند . فرض کنید به ازای هر رکورد i و j اگـر $i<j$ و در لیست ورودی $k_i=k_j$ (یعنی دو کلید برابر باشند) آنگاه اگر در لیست مرتب شده R_i قبل از R_j واقع شود روش مرتب‌سازی را پایدار (Stable) می‌گویند. یعنی یک روش مرتب‌سازی پایدار رکوردهـای با کلیه‌های مساوی را به همان ترتیب قبل از عمل مرتب‌سازی نگهداری می‌کنیم.

به عنوان مثال فرض کنید اگر رشته ورودی به صورت $4,2^{(1)},5,1,2^{(2)}$ باشـد اگـر رشـته مرتـب شـده بـه صورت $1,2^{(1)},2^{(2)},4$ باشد الگوریتم مرتب‌سازی پایدار می‌باشـد و اگـر رشـته مرتـب شـده بـه صـورت $1,2^{(2)},2^{(1)},4,5$ باشد الگوریتم مرتب‌سازی پایدار نمی‌باشد. توجه کنیـد شـماره‌هـای بـالای عـدد۲ نـشان دهنده ترتیب ورود آنها است.

تعریف: اگر الگوریتم مرتب‌سازی از فضای ممکن به طول ثابت، مستقل از تعداد عناصر ورودی برای مرتب‌سازی استفاده کند، روش مرتب‌سازی را درجا (inplace) و در غیر این صورت برون‌جا (outplace) می‌نامند.

۸,۲ مرتب سازی با آدرس

عمل مرتب سازی می‌تواند بر روی خود رکوردها یا بر روی جدولی از اشاره‌گرها صورت گیرد. به عنوان مثال شکل (الف) را که در آن فضای فایلی با ۵ رکورد نشان داده شده در نظر بگیرید. اگر فایل بر حسب شماره کلید به طور صعودی مرتب گردد، نتیجه آن در شکل (ب) مشاهده می گردد.

	کلید	سایر فیلدها
رکورد ۱	4	DDD
رکورد ۲	2	BBB
رکورد ۳	1	AAA
رکورد ٤	5	EEE
رکورد ۵	3	CCC

فایل

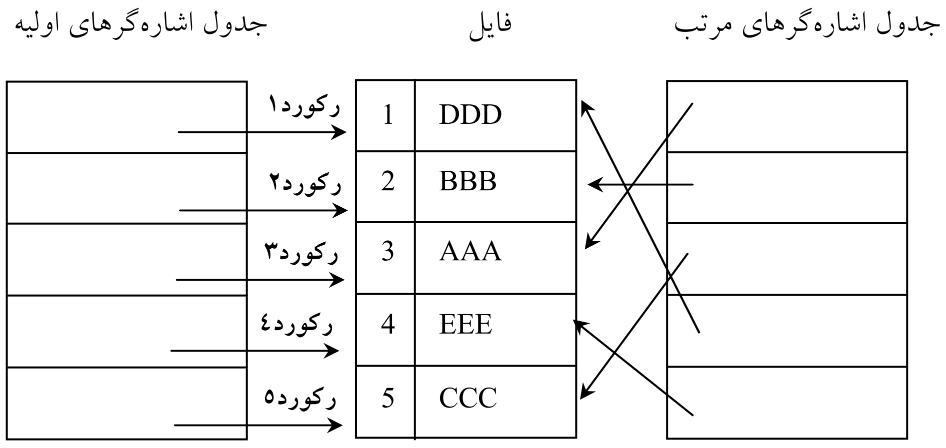
(الف) فایل اصلی

کلید	سایر فیلدها
1	AAA
2	BBB
3	CCC
4	DDD
5	EEE

فایل

(ب) فایل مرتب

اما فرض کنید مقدار داده‌های هر رکورد فایل شکل (الف) بسیار زیاد باشد. در این صورت جابجایی واقعی داده‌ها مستلزم هزینه زیادی است. در این حالت ممکن است جدولی از اشاره‌گرها مورد استفاده قرار گیرد که به جای جابجایی واقعی رکوردها، اشاره‌گرها جابجا می‌گردند. (شکل ۸,۱)



شکل ۸,۱ مرتب‌سازی به کمک جدولی از اشاره‌گرها

این روش مرتب‌سازی را مرتب‌سازی با آدرس می‌گوییم. توجه داشته باشید که هیچ کدام از رکوردهای فایل جابجا نشده‌اند.

در بیشتر برنامه‌های این فصل تکنیک‌هایی را برای مرتب‌سازی واقعی رکوردها توضیح می‌دهیم. توسعه این تکنیک‌ها برای مرتب‌سازی با آدرس ساده بود، و به عهده دانشجو واگذار می‌شود.

۸,۳ مرتب‌سازی یا جستجو

به دلیل ارتباط تنگاتنگ بین مرتب‌سازی و جستجو، معمولاً در هر کاربردی این سوال مطرح می‌شود که آیا فایل مرتب شود سپس عمل جستجو درآن انجام گیرد یا خیر. گاهی عمل جستجو در یک فایل نسبت به مرتب‌سازی فایل و سپس جستجوی یک عنصر خاصی به کار کمتری نیاز دارد. به عبارت دیگر، اگر فایلی مکرراً برای دستیابی عناصر خاصی مورد استفاده قرار گیرد بهتر است، مرتب گردد. در این صورت کارایی آن بیشتر خواهد بود. علتش این است که هزینه جستجوهای متوالی ممکن است خیلی بیشتر از هزینه یکبار مرتب‌سازی و جستجوهای متوالی از فایل مرتب باشد. بنابراین نمی‌توان گفت که برای کارایی بیشتر فایل باید مرتب گردد یا خیر. برنامه‌نویس باید بر اساس شرایط و نوع مساله تصمیم بگیرد. وقتی تصمیم به عمل مرتب‌سازی گرفته شد، باید تصمیماتی راجع‌به روشهای مرتب‌سازی و چیزهای که باید مرتب گردند، اخذ شود. هیچ روش مرتب‌سازی موجود ندارد که از هر جهت از سایر روش‌ها ممتاز باشد. برنامه‌نویس باید مساله را به دقت بررسی کرده و با توجه به نتایجی که انتظار می‌رود، روش مناسبی را انتخاب کنیم.

۸,٤ ملاحظات کارایی

همانطور که در این فصل مشاهده خواهید کرد روشهای متعددی برای مرتب‌سازی در یک فایل وجود دارند. برنامه‌نویس باید با ملاحظات کارایی الگوریتم‌ها آشنا بوده و انتخاب هوشمندانه‌ای را در تعیین روش مرتب‌سازی مناسب برای یک مسئله خاص داشته باشد. سه موضوع مهمی را که در این رابطه باید در نظر گرفت عبارتند از:

- ✓ مدت زمانی که برنامه‌نویس باید برای نوشتن برنامه مرتب‌سازی صرف نماید.
- ✓ مدت زمانی از وقت ماشین که به اجرای این برنامه مرتب‌سازی اختصاص می‌یابد.
- ✓ حافظه مورد نیاز برنامه

اگر فایل کوچک باشد کارایی تکنیک‌های پیچیده‌ای که برای کاستن میزان فضا و زمان طراحی می‌شوند بدتر یا کمی بهتر از کارایی الگوریتمهای ساده است، اگر یک برنامه مرتب سازی فقط یک بار اجرا شود و زمان فضای کافی برای آن وجود دارد، جالب نیست که برنامه‌نویس روزها وقت صرف کند تا بهترین روش مرتب‌سازی را جهت به دست آوردن حداکثر کارایی پیدا کند.

اغلب به کارایی زمان یک روش مرتب‌سازی را با تعداد واحدهای زمانی مورد نیاز اندازه‌گیری نمی‌کنیم. بلکه با تعداد اعمال بحرانی که باید صورت گیرد می‌سنجیم. مثالهایی از این اعمال کلیدی عبارتند از: مقایسه کلیدها، انتقال رکوردها یا جابجایی دو رکورد.

آن دسته از اعمال بحرانی انتخاب می‌گردند که بیشترین وقت را به خود اختصاص می‌دهند. برای مثال در عمل مقایسه کلیدها، اگر کلیدها طولانی باشند یک عمل بحرانی است. بنابراین زمان لازم برای مقایسه کلیدها خیلی بیشتر از زمان لازم برای افزایش یک واحد به اندیس حلقه تکرار for است. همچنین تعداد اعمال ساده مورد نیاز معمولاً متناسب با مقدار مقایسه کلیدهاست. به همین دلیل تعداد مقایسه کلیدها کمیت خوبی برای سنجش کارایی زمان مرتب‌سازی است.

۸,۵ مقایسه روش‌های مرتب‌سازی

با توجه به مفهوم مرتبه یک روش مرتبه‌سازی، می‌توان تکنیک‌های مرتب‌سازی مختلف را با هم مقایسه کرد و آنها را به دو دسته خوب یا بد تقسیم نمود. ممکن است فردی آرزوی کشف مرتب‌سازی بهینه‌ای از مرتبه $O(n)$ را صرفنظر از محتویات یا درجه ورودی داشته باشد. اما متاسفانه می‌توان نشان داد که چنین روش مرتب‌سازی وجود ندارد. زمانهای اغلب روشهای مرتب‌سازی کلاسیک که در این جا بررسی می‌شوند در حدود $O(n\log n)$ تا $O(n^2)$ هستند. سرعت رشد n^2 نسبت به $n\log\ n$ بسیار زیاد است اما همین که مرتبه یک روش مرتب سازی $O(n\log n)$ است، دلیلی برای انتخاب این روش مرتب‌سازی نیست! ارتباط بین طول فایل و سایر عبارات تشکیل دهنده زمان مرتب‌سازی باید مشخص گردد.

در بسیاری از موارد، زمان لازم برای مرتب‌سازی به ترتیب اولیه داده‌ها بستگی دارد. برای بعضی از روشهای مرتب‌سازی، اگر داده‌ها تقریباً مرتب باشند در زمان $O(n)$ به طور کامل مرتب می‌گردند در حالی که اگر داده‌ها به ترتیب معکوس مرتب باشند. زمان لازم برای مرتب سازی برابر با $O(n^2)$ خواهد شد. در بعضی دیگر از روش‌های مرتب‌سازی، صرفنظر از ترتیب اولیه داده‌ها، زمان لازم برابر با $O(n\log n)$ است. بنابراین اگر از ترتیب اولیه داده‌ها باخبر باشیم می‌توانیم تصمیم هوشمندانه‌تری در انتخاب روش مرتب‌سازی داشته باشیم. از طرف دیگر، اگر چنین اطلاعاتی نداشته باشیم ممکن است الگوریتمی را بر اساس بدترین حالت یا حالت متوسط انتخاب کنیم. به طور کلی می‌توان گفت که بهترین روش مرتب‌سازی که در همه موارد قابل استفاده باشد وجود ندارد.

> به طور کلی می‌توان گفت که بهترین روش مرتب‌سازی که در همه موارد قابل استفاده باشد وجود ندارد.

وقتی که یک روش مرتب‌سازی خاص انتخاب شد، برنامه‌نویس باید مبادرت به نوشتن برنامه‌ای کند که حداکثر کارایی را داشته باشد این امر ممکن است از خوانایی برنامه بکاهد. یکی از علل این است که ممکن است عمل مرتب‌سازی قسمت اصلی و مهم برنامه باشد و هرگونه بهبود در سرعت عمل مرتب‌سازی، کارایی برنامه را بالا ببرد. علت بعدی این است که اغلب مرتب سازی مکرراً مورد استفاده قرار می‌گیرند، لذا بهبودی هرچند ناچیز در روش مرتب‌سازی موجب صرفه‌جویی زیادی در وقت کامپیوتر می‌گردد.

ملاحظات حافظه نسبت به ملاحظات زمان از اهمیت کمتری برخوردارند. یکی از دلایل این است که در بیشتر برنامه‌های مرتب‌سازی میزان حافظه مورد نیاز به O(n) نزدیک‌تر است تا $O(n^2)$ علت دیگر این است که در صورت نیاز به حافظه بیشتر، همواره می توان آن را با حافظه‌های جانبی تأمین کرد. یک روش مرتب‌سازی ایده‌آل مرتب سازی درجا است. در این مرتب‌سازی فضای اضافی مورد نیاز O(n) است. یعنی مرتب‌سازی درجا عمل مرتب سازی را در آرایه یا لیستی که حاوی این عناصر است انجام می‌دهد. فضای اضافی مورد نیاز، صرفنظر از اندازه مجموعه‌ای که باید مرتب گردد، به صورت تعداد ثابتی از محل‌ها (مانند متغیرهای تعریف شده یک برنامه) می‌باشد.

معمولاً ارتباط بین زمان و حافظه موردنیاز یک روش مرتب‌سازی به این صورت است: الگوریتم‌هایی که به زمان کمتری نیاز دارند، حافظه بیشتری را بخود اختصاص می‌دهند و برعکس. اما الگوریتم‌های باهوشی وجود دارند که از حداقل زمان و حافظه استفاده می‌کنند. این الگوریتم ها همان الگوریتم‌های درجا هستند که از درجه $O(n\log n)$ می‌باشند.

۸,۶ روش‌های مرتب‌سازی

۸,۶,۱ مرتب‌سازی حبابی (Bubble Sort)

مرتب‌سازی حبابی از نوع مرتب‌سازی‌های تعویضی می‌باشد. در مرتب‌سازی تعویضی، جفت‌هایی از عناصر با هم مقایسه می‌شوند و در صورتی که به ترتیب مناسبی نباشند، جای آنها تعویض می‌گردد تا فایل مرتب شود.

توجه در هر یک از مثالها، آرایه‌ای از اعداد صحیح مانند A را در نظر می‌گیریم که باید مرتب گردند.

در مرتب‌سازی حبابی باید چندین بار در طول آرایه حرکت کنیم و هر بار عنصری را با عنصر بعدی خودش مقایسه می‌شود و در صورتی که عنصر اول از عنصر دوم بزرگتر باشد (در مرتب‌سازی صعودی) جای آنها عوض می‌شود. به عنوان مثال، فایل زیر را در نظر بگیرید:

25 57 48 37 12 92 86 33

در گذر اول، مقایسه‌های زیر باید انجام گیرد:

A[0] با A[1]	(25 با 57)	جابجایی انجام نمی‌گیرد.
A[1] با A[2]	(57 با 48)	جابجایی انجام می‌گیرد.
A[2] با A[3]	(57 با 37)	جابجایی انجام نمی‌گیرد.
A[3] با A[4]	(57 با 12)	جابجایی انجام می‌گیرد.
A[4] با A[5]	(57 با 92)	جابجایی انجام نمی‌گیرد.
A[5] با A[6]	(92 با 86)	جابجایی انجام می‌گیرد.
A[6] با A[7]	(92 با 33)	جابجایی انجام می‌گیرد.

بنابراین پس از گذر (pass) اول، محتویات فایل به صورت زیر خواهد بود:

25 48 37 12 57 86 33 92

توجه داشته باشید که پس از گذر اول، بزرگ‌ترین عنصر (در مرتب‌سازی صعودی) در موقعیت مناسب خود در آرایه قرار می‌گیرد.

به طور کلی پس از تکرار iام، x[n-i] در موقعیت مناسب خود قرار می‌گیرد. پس از گذر دوم، محتویات فایل به صورت زیر خواهد بود:

25 37 12 48 57 33 86 92

توجه کنید که عدد 86 دومین محل از انتهای آرایه را اشغال کرده است و این محل جای مناسب آن می‌باشد. فایلی به طول n حداکثر در n-1 تکرار، مرتب می‌گردد.

تکرارهای مختلف منجر به مرتب شدن فایل موردنظر می‌گردد عبارتند از:

فایل اولیه

25	57	48	37	12	92	86	33
25	48	37	12	57	86	33	92
25	37	12	48	57	33	86	92
25	12	37	48	33	57	86	92
12	25	37	33	48	57	86	92
12	25	33	37	48	57	86	92
12	25	33	37	48	57	86	92
12	25	33	37	48	57	86	92

با توجه به این توضیحات، می‌توان برنامه مرتب‌سازی حبابی را نوشت. اما بهبودهایی را می‌توان در روش بیان شده اعمال کرد.

۱–چون پس از تکرار iام، کلیه عناصر موجود در موقعیت‌های بزرگتر از n-i در محل مناسب خود قرار دارند، نیازی به بررسی آنها در تکرار بعدی نیست. بنابراین در گذر اول n-1 مقایسه در گذر دوم n-2 مقایسه و در گذر (n-1) فقط یک مقایسه (میان A[0] , A[1]) صورت می‌گیرد.

۲–نشان دادیم که برای مرتب‌سازی فایلی به طول n، حداکثر n-1 تکرار لازم است، اما در مثال قبلی که تعداد عناصر فایل ۸ بود، فایل در ۵ تکرار مرتب می‌شود و نیازی به دو تکرار آخر نیست. برای حذف گذرهای زاید، باید قادر به تشخیص این کار باشیم که آیا در طی یک گذر جابجایی صورت گرفته است یا نه، به همین دلیل از متغیر منطقی flag در الگوریتم استفاده می‌کنیم. اگر پس از هر گذر flag=0 آنگاه لیست از قبل مرتب شده است و هیچ نیازی به ادامه کار نیست. این کار باعث کم شدن تعداد گذرها می‌شود.

با استفاده از این بهبودها، تابعی به نام bubble را می‌نویسیم. این تابع در متغیر A و n را می‌پذیرد که A آرایه‌ای از اعداد و n تعداد اعدادی است که باید مرتب گردند (n ممکن است کمتر از تعداد عناصر آرایه باشد).

الگوریتم مرتب سازی حبابی

```
void bubble (int A [ ] , int n)
  {
        int i,j,nt;
        int flag=1;
        for (i=n-1;i>0   &&  flat; i--)
         {
              flag=0;
              for (j=0; j<i ; j++)
                if (A[j]>A[j+1])
                   {
                      flag =1;
                      t=A[j];
                      a[j]=a[j+1];
                      a[j+1]=t;
                      }
              }
        }
```

پیچیدگی الگوریتم مرتب کردن حبابی

اگر بهبودهای مطرح شده، به الگوریتم اعمال نشوند، تحلیل آن ساده است. برای مرتب سازی فایلی به طول n، حداکثر به n-1 گذر لازم است. در هر گذر n-1 مقایسه انجام می‌گیرد. بنابراین تعداد کل مقایسه‌ها عبارت است از:

$$(n-1)*(n-1) = n^2 - 2n + 1$$

که از مرتبه $O(n^2)$ می‌باشد.

اکنون ببینیم که بهبودهای مطرح شده چه تاثیری بر روی سرعت اجرای الگوریتم دارند. تعداد مقایسه‌ها در تکرار iام برابر با n-i است. لذا اگر k تعداد تکرار باشد، تعداد تکرار مقایسه‌ها برابر است با:

$$(n-1)+(n-2)+(n-3)+...+(n-k) = \frac{(2nk - k^2 - k)}{2}$$

می توان نشان داد که تعداد متوسط تکرارها(k) از مرتبه O(n) است. ولی فرمول کلی از مرتبه $O(n^2)$ است. البته ضریب ثابت از حالت قبلی کوچکتر است. اما در این روش کارهای اضافی دیگری از قبیل

تست و تعداددهی اولیه به متغیر ($flag$) در هر گذر و قرار دادن مقدار ۱ در این متغیر (یک بار برای هـر جابجایی) باید صورت گیرد.

مرتب سازی حبابی در صورتی که فایل به طور کامل (یا تقریباً کامل) مرتب باشـد از مرتبـه O(n) اسـت. بنابراین مرتب‌سازی حبابی در بردار مرتب بهترین عملکرد و در بردار نامرتب بدترین عملکرد را دارد. ایـن الگوریتم پایدار (stable) می‌باشد.

ویژگی های مرتب سازی حبابی
✓ مرتب سازی حبابی در صورتی که فایل به طور کامل (یا تقریباً کامل) مرتب باشـد از مرتبه O(n) است. ✓ مرتب‌سازی حبـابی در بـردار مرتـب بهتـرین عملکـرد و در بـردار نامرتـب بـدترین عملکرد را دارد. ✓ این الگوریتم پایدار (stable) می‌باشد.

۸,٦,۲ مرتب‌سازی انتخابی (selection sort)

فرض کنید آرایه A بـا n عنـصر A[1] , A[2] , ... , A[n] در حافظـه اسـت. الگـوریتم مرتـب کـردن انتخابی برای مرتب کردن آرایه A به صورت زیر عمل می‌کند. نخست کوچکترین عنصر داخـل لیـست را پیدا می‌کند و آن را در مکان اول لیست قرار می‌دهد (جای آن را با عنصر اول لیست عوض می‌کند) آنگـاه کوچکترین عنصر دوم داخل لیست را پیدا می‌کنید و آن را در مکان دوم لیست قرار می‌دهد و الی آخر.

به عنوان مثال فرض کنید لیست زیر باید به طور صعودی مرتب شود:

77 , 33 , 44 , 11 , 88 , 22 , 66 , 55

ابتدا لیست را برای پیدا کردن کوچکترین عنصر پیمایش می‌کنیم و آن را در موقعیـت 4 مـی‌یـابیم و ایـن عنصر را با عنصر اول تعویض می‌کنیم و در نتیجه کوچکترین عنصر لیست در ابتدای لیست قرار می‌گیرد.

11 , 33 , 44 , 77 , 88 , 33 , 66 , 55

اکنون از موقعیت 2 تا انتهای لیست، کوچکترین عنصر را پیدا می‌کنیم و آن را در موقعیت 6 می‌یابیم و این عنصر را با عنصر دوم لیست تعویض می‌کنیم و این عنصر نیز در موقعیت مناسب خود قرار می‌گیرد.

11 , 22 , 44 , 77 , 88 , 33 , 65 , 55

مراحل فوق را n-1 بار تکرار می‌کنیم تا همه عناصر در جای مناسب خود قرار گیرند.

الگوریتم مرتب سازی انتخابی

```
void selection (int A[ ] , int n)
 {
        int i,j , min post , t;
  for (i=0 , i<n-1;i++)
        {
          minpos =i;
           for (j=i+1,j<n;j++)
             if (A[j]<A[minpos])
                   minpos=j;
             t=A[minpos];
             A[minpos]=A[i];
             A[i]=t;
        }
  }
```

پیچیدگی الگوریتم مرتب‌سازی انتخابی

در اولین تکرار حلقه خارجی i، حلقه تکرار داخلی j به تعداد n-1 بار اجـرا مـی‌شـود. در مرحلـه دوم بـه تعداد n-2 بار تکرار می‌شود.

بنابراین:

$$(n-1)(n-2)+...+1=\frac{n(n-1)}{2}=O(n^2)$$

این الگوریتم پایدار (stable) نیست و ممکن است ترتیب عناصر مساوی را در آرایه حفظ نکند.

ویژگی های مرتب سازی انتخابی

- ✓ مرتب سازی انتخابی در همه موارد دارای مرتبه زمانی $O(n^2)$ می باشد.
- ✓ این الگوریتم پایدار (stable) نیست و ممکن است ترتیب عناصـر مـساوی را در آرایه حفظ نکند.

۸,٦,۳ مرتب سازی سریع (Quick sort)

مرتب سازی سریع الگوریتمی از نوع تقسیم و غلبه است که دارای میانگین زمانی بسیار مناسبی می‌باشد. روش مرتب سازی سریع ارایه شده توسط C.A.R Hoare در بین مرتب‌سازی‌های مورد مطالعه دارای بهترین متوسط زمانی می‌باشد.

راهبرد تقسیم و غلبه یک روش بازگشتی است که در آن، مسئله‌ای که باید حل شود به مسئله‌های کوچکتر تقسیم می‌گردد که هر کدام مستقلاً حل می‌شوند.

در مرتب‌سازی سریع عنصری به نام محور (pivot) انتخاب می‌گردد و سپس دنباله‌ای از تعویض ها صورت می‌گیرد تا عناصری که کوچکتر از این محور هستند در سمت چپ محور و بقیه در سمت راست آن قرار گیرند. بدین ترتیب محور درجای مناسب خود قرار می‌گیرد و لیست را به دو بخش کوچکتر تقسیم می‌کند که هر کدام از این بخشها به طور مستقل و به همین روش مرتب می‌شوند.

برای آشنایی با این الگوریتم لیست زیر را در نظر بگیرید.

75 , 70 , 65, 84 , 98 , 78 , 100 , 93 , 55, 81 , 68

برای سهولت اولین عنصر لیست یعنی 75 را به عنوان محور می‌گیریم. کاری که ما باید انجام دهیم بدین صورت خواهد بود که کلیه عناصر کوچکتر از 75 را به سمت چپ آن و کلیه عناصر بزرگتر از 75 را به سمت راست آن انتقال دهیم. و سپس این زیرلیستها را نیز به طور بازگشتی عمل فوق را روی آن انجام دهیم.

برای عمل فوق بدین ترتیب عمل می‌کنیم: از انتهای راست لیست اولین کوچکترین عنصر از محور (عدد 68) را پیدا می‌کنیم و ابتدای چپ لیست اولین بزرگترین عنصر از محول (عدد 84) را پیدا می‌کنیم.

75 , 70 , 65, 84 , 98 , 78, 100 , 93 , 55 , 61, 81 , 68

سپس جای این عناصر را تعویض می کنیم.

75 , 70 , 65, 68 , 98 , 78 , 100 , 93 , 55 , 61 , 81 , 84

جستجو را از سمت راست ادامه می‌دهیم تا عنصر دیگری که کوچکتر از 75 (عدد 61) پیدا شود و سمت چپ ادامه می‌دهیم تا عنصر دیگر بزرگتر از 75 (عدد 98) پیدا شود.

75 , 70 , 65 , 68 , 98 , 78 , 100 , 93 , 55 , 61 , 81 , 84

جای این عناصر را تعویض می‌کنیم.

75 , 70 , 65 , 68 , 61 , 78, 100 , 93 , 55 , 98 , 81 , 84

در جستجوی مرحله بعد مقادیر 78 , 55 پیدا می‌شوند.

75 , 70 , 65 , 68 , 78 , 100 , 93 , 55 , 98 , 81 , 84

جای این عناصر را تعویض می‌کنیم:

75 , 70 , 65 , 68 , 55 , 100 , 93 , 78, 98 , 81 , 84

اکنون که جستجو را از سمت راست از سر می‌گیریم عنصر 55 را پیدا می‌کنیم کـه در جـستجوی قبلـی از چپ پیدا شده بود.

75 , 70 , 65, 68 , 61 , 55 , 100 , 93 , 78 , 98, 81 , 84

در این جا اشاره‌گرهای مربوط به جستجوهای چپ و راست با هم برخورد می‌کند و بیانگر این اسـت کـه جستجو خاتمه یافته است. اکنون 55 را با محور 75 عوض می‌کنیم.

55 , 70 , 65 , 68 , 61 , 75 , 100 , 93 , 78, 98, 81, 84

توجه داشته باشید که تمام عناصر سمت چپ 75 ازآن کـوچکتر و تمـام عناصـر راسـت آن از آن بزرگتـر هستند. و در نتیجه 75 در جای مناسبی ذخیره شده است.

لیست سمت چپ عبارت است از:

55 , 70 , 65 , 68 , 61

و لیست سمت راست عبارت است از:

100 , 93 , 78, 98, 81 , 84

هر کدام از این لیست‌ها، با انتخاب عنصر محوری در هر کدام، روند قبلی را تکرار کنید.

پیاده‌سازی الگوریتم مرتب‌سازی سریع

الگوریتم مرتب‌سازی سریع

تابع Split( ) برای تقسیم کردن آرایه به کار می‌رود.

```
void quicksort (int A[ ] , int first, int last)
  {
          int pos;                /*final position of pivot */
             if (fitst <last)
               {
                    /*split int two sublists*/
                    quicksort (a,fisrt , pos-1);
                    quicksort (a,pos+1, last);
          }

//*********************************************
void split (int A[ ], int first , int last , int * pos)
   {
          int left = first , right = last , pivot = A[first], t;
          while (left <right)
                {
                    while (A[right] >pivot)
                       right - -;
                   while (left < right && A[left] <= pivot)
                       left++;
                    if (left <right)
```

```
        {
            t=A[left];
            A[left]=A[right];
            A[right]=t
          }
/*end of searches, place pivot in correct position*/
            *pos=right;
            A[first]=A[*pos]
            A[*pos]=pivot;
}
```

پیچیدگی الگوریتم (Quick sort)

زمان اجرای یک الگوریتم مرتب کردن معمولاً با تعداد دفعات مقایسه مورد نیاز ((f(n)) برای مرتب کردن n عنصر اندازه‌گیری می‌شود. الگوریتم Quick sort که دارای تعداد زیادی مقایسه است به شدت مورد مطالعه و بررسی قرار گرفته است. در حالت کلی این الگوریتم در بدترین حالت زمان اجرائی از مرتبه $O(n^2)$ دارد اما زمان اجرای حالت میانگین آن از مرتبه O(nlogn) است. دلیل آن در زیر ارائه شده است:

بدترین حالت وقتی اتفاق می‌افتد که لیست از قبل مرتب شده باشد آنگاه نخستین عنصر به n مقایسه احتیاج دارد. تا معلوم شود در مکان اول قرار گیرد. علاوه بر این، لیست کوچک شده اول خالی خواهد بود اما لیست کوچک شده دوم n-1 عنصر دارد. بنابراین عنصر دوم به n-1 مقایسه احتیاج دارد تا معلوم شود در مکان دوم قرار می گیرد و الی آخر. در نتیجه، مجموعاً تعداد:

$$f(n) = n + (n-1) + \ldots + 2 + 1 = \frac{n(n+1)}{2} = O(n^2)$$

مقایسه انجام شود ملاحظه می‌کنید که این عدد برابر پیچیدگی الگوریتم مرتب کردن حبابی است.

پیچیدگی O(nlogn) حالت میانگین از این واقعیت ناشی می شود که به طور متوسط، هر مرحله ساده‌سازی در الگوریتم دو لیست کوچکتر تولید می‌کند. بنابراین:

با ساده شدن لیست اول، ۱ عنصر در جای خود قرار می گیرد و دو لیست کوچکتر تولید می‌شود.

با مساوی شدن دو لیست، ۲ عنصر در جای خود قرار می‌گیرد و چهار لیست کوچکتر تولید می‌شود.

با ساده شدن چهار لیست، ٤ عنصر در جای خود قرار می‌گیرد و هشت لیست کوچک تولید می‌شود و الی آخر.

ملاحظه می‌کنید که مرحله ساده شدن در kامین سطح مکان عنصر 2^{k-1} ام را پیدا می کند و از این دو تقریباً $\log n$ سطح ساده سازی وجود دارد. علاوه بر این هر سطح حداکثر از n مقایسه استفاده می کند. بنابراین $f(n) = O(n \log n)$

در واقع تحلیل ریاضی و ملاحظات تجربی هر دو نشان می دهند که:

$$f(n) \cong 1.4[n \log n]$$

تعداد انتظاری مقاسیه‌ها برای الگوریتم quich sort می‌باشد.

قابل تذکر است که این الگوریتم پایدار نمی‌باشد. و پیچیدگی این الگوریتم در حالت کلی به صورت زیر است:؛

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

ویژگی های مرتب سازی سریع
✓ در این الگوریتم انتخاب عنصر محوری (pivot) تاثیر مهمی در سرعت اجرای آن دارد. ✓ در مرتب سازی سریع بدترین حالت زمانی رخ می دهدکه عنصر محوری کوچکترین یا بزرگترین عنصر آرایه باشد. ✓ در مرتب سازی سریع اگر آرایه از قبل مرتب باشد، بدترین حالت رخ می دهد که در اینصورت مرتبه زمانی برابر $O(n^2)$ خواهد شد. ✓ اگر عنصر محوری آرایه را به دو زیر آرایه تقریبا یکسان تبدیل کند در اینصورت بهترین حالت رخ داده و مرتبه زمانی برابر O(nlogn) می باشد. ✓ اگر مرتب سازی سریع در بهترین حالت خود باشد در آنصورت سریع ترین روش مرتب سازی خواهد بود. ✓ این الگوریتم پایدار(متعادل) نیست. ✓ در حالت کلی در آرایه های مرتب دارای بدترین عملکرد و در آرایه های نامرتب دارای بهترین عملکرد می باشد.

۸,٦,٤ مرتب‌سازی درجی (Insertion sort)

در مرتب‌سازی به روش درج، ابتدا دو عنصر اول لیست مرتب می‌شوند. سپس عنصر سوم نسبت به عنصر اول و دوم درجای مناسبی قرار می‌گیرد. سپس عنصر چهارم نسبت به عناصر اول و دوم و سوم در جای مناسبی قرار می‌گیرد و این روند تا مرتب شدن کامل لیست ادامه می‌یابد. یا به صورت دقیق‌تر:

مرحله ۱: A[1] خودش به طور بدیهی مرتب است.

مرحله ۲: A[2] را یا قبل از یا بعد از A[1] درج می‌کنیم طوری که A[1] و A[2] مرتب شوند.

مرحله ۳: A[3] را در مکان صحیح در A[1] و A[2] درج می‌کنیم به گونه‌ای که A[1]، A[2] و A[3] مرتب شده باشند.

.

.

.

مرحلهn: A[n] را در مکان صحیح خود در A[1]، A[2] ، ...، A[n-1] به گونه‌ای درجه می‌کنیم که کل آرایه مرتب باشد.

برنامه زیر روش فوق را پیاده‌سازی می‌کند.

الگوریتم مرتب سازی درجی

```
void Insertion sort (int A[ ] , int n)
      {
         int i,3,t;
         for (i=1; i<n , i++)
           {
              t=A[i];
              for (j=i; j>0 && A[j-1] >t;j-- )
                A[j] = A[j-1];
              A[j]=t;
      }
}
```

پیچیدگی مرتب‌سازی درجی

f(n) تعداد مقایسه‌های الگوریتم مرتب‌سازی درجی را می‌توان به سادگی محاسبه کرد. قبل از همه خاطرنشان می‌کنیم که بدترین حالت وقتی اتفاق می‌افتد که آرایه A به ترتیب عکس مرتب باشد و حلقه خارجی بخواهد از حداکثر تعداد مقایسه استفاده کند. از این رو

$$f(n) = 1 + 2 + ... + (n-1) = \frac{n(n-1)}{2} = O(n^2)$$

اگر آرایه‌ای که در اختیار الگوریتم مرتب‌سازی درجی قرار می‌گیرد تقریباً مرتب باشد آنگها الگوریتم از مرتبه O(n) خواهد بود.

$$1 + 1 + ... + 1 + 1 = n - 1 = O(n)$$

این الگوریتم جزء الگوریتمهای پایدار می‌باشد و همانگونه که اشاره شد در یک بردار مرتب بهترین حالت وبرای یک بردار مرتب شده معکوس بدترین حالت را دارد. این الگوریتم برای n های کوچک روش بسیار مناسبی است و ثابت شده است که برای $n \le 20$ سریعترین روش مرتب‌سازی است.

ویژگی های مرتب سازی درجی
✓ این الگوریتم متعادل بوده و در یک آرایه کاملا مرتب بهترین حالت و برای یک آرایه مرتب شده معکوس بدترین حالت را دارد. ✓ برای n کوچک این روش بهترین روش مرتب سازی می باشد. .

۸,۶,۵ مرتب‌سازی هرمی

قبلاً در فصل درخت، درختهای Heap و مرتب‌سازی هرمی را به طور کامل بررسی کردیم. هرم تقریباً مرتب است، زیرا هر مسیری از ریشه به برگ، مرتب است. به این ترتیب، الگوریتم کارآمدی به نام مرتب‌سازی هرمی را می‌توان با استفاده از آن به دست آورد. این مرتب‌سازی همانند سایر مرتب‌سازیها بر روی یک آرایه صورت می‌گیرد. این روش مرتب‌سازی همانند مرتب‌سازی سریع از یک تابع کمکی استفاده می‌کند. پیچیدگی آن همواره O(nlogn) است و برخلاف مرتب‌سازی سریع به صورت بازگشتی نیست.

ویژگی های مرتب سازی هرمی
✓ کلیه اعمال در مرتب سازی هرمی از مرتبه O(nlogn) است ✓ در این روش درخت heap روی آرایه ساخته می شود. ✓ مرتب‌سازی هرمی از نوع درجا می باشد. ✓ این الگوریتم پایدار (stable) نمی‌باشد.

۸,۶,۶ مرتب سازی ادغام (Merge sort)

ادغام کردن دو لیست مرتب

فرض کنید A یک لیست مرتب شده با r عنصر و B یک لیست مرتب شده با s عنصر باشد. عمل ترکیب عناصر آرایه B , A در یک لیست مرتب شده C با n=r+s عنصر، ادغام کردن نام دارد. یک راه ساده برای ادغام دو آرایه فوق بدین صورت است که عناصر B را پس از عناصر A قرار دهیم و آنگاه از یک الگوریتم مرتب سازی برای مرتب کردن لیست استفاده کنیم. این روش از این واقعیت که B , A به صورت انفرادی مرتب شده هستند استفاده نمی‌کند.

در این بخش الگوریتمی با کارایی به مراتب بیشتری را بیان می‌کنیم. فرض کنید دو آرایه A , B هر یک دارای s,r عنصر به صورت مرتب شده داشته باشیم. در هر مرحله دو عنصر جلویی با هم مقایسه می‌شوند و عنصر کوچکتر در آرایه به نام C قرار داده می‌شود. این کار را تا زمانی که یکی از آرایه‌ها خارج می‌شود

ادامه می‌دهیم. هرگاه یکی از آرایه‌ها خالی شد تمام عناصر آرایــه باقیمانــده را بــه انتهــای آرایــه C اضــافه می‌کنیم.

اکنون بحث بالا را به یک برنامه رسمی تبدیل می‌کنیم.

الگوریتم ادغام دو آرایه مرتب

```
void merge (int A[ ] , int n1 , int B[ ] , int n2 , int C[ ] , int n3)
        {
        int i=0 , j=0, k;
          if ((n1 +n2)! =n3)
                {
                    printf(" size of n3 incorrect");
                    getch ( );
                    exit(0);
                }
        for (k=0; i<n1 && j<n2; k++)
          if (A[i]< B[j])
                C[k]=A[i++];
          else
                C[k]=B[j++];
           while (i<n1)
                C[k++]=A[i++];
  whild (j<n2)
                C[k++]=B[j++];
        }
```

پیچیدگی الگوریتم ادغام کردن

ورودی الگوریتم ادغام کردن را تعداد کل عنصرهای B , A یعنــی n=r+s عنـصر تـشکیل مـی‌دهـد. هـر مقایسه یک عنصر را در آرایه قرار می‌دهد. که در نهایـت n عنـصر دارد. بنـابراین تعـداد مقایـسه‌هـا f(n) نمی‌تواند بیشتر از n باشد:

$$f(n) \leq n = O(n)$$

به بیان دیگر الگوریتم ادغام کردن دو آرایه مرتب زمان اجرای خطی دارد.

جستجوی دودوئی و الگوریتم درج کردن

فرض کنید r تعداد عناصر آرایه مرتب شده A خیلی کوچکتر از S تعداد عناصر آرایه مرتب شده B است. عمل ادغام آرایه A و B را می‌توان به صورت زیر انجام داد. برای هر عنصر A[k] از آرایه A برای پیدا کردن مکان صحیح جهت درج کردن A[k] در B، از یک جستجوی دودوئی روی B استفاده می‌کند. هر جستجوی به حداکثر logs مقایسه احتیاج دارد. از این رو این جستجوی دودوئی و الگوریتم درج کردن جهت ادغام دو آرایه A و B به حداکثر logs مقایسه احتیاج دارد. تأکید می‌کنیم که تنها وقتی که r<<s یعنی وقتی r خیلی کوچکتر از s باشد، این الگوریتم بسیار کاراتر از الگوریتم ادغام کردن متداول که در بخش قبلی بحث کردیم، می‌باشد.

مثال. فرض کنید A دارای S عنصر و B دارای 100 عنصر است. آنگاه ادغام A , B توسط الگوریتم متداول ادغام تقریباً از 100 مقایسه استفاده می‌کند. از طرف دیگر، تنها تقریباً به log100=7 مقایسه احتیاج است تا با استفاده از جستجوی دودویی مکان صحیح یک عنصر A جهت درج شدن در B تعیین شود. از این رو تنها تقریباً به 5.7=35 مقایسه احتیاج است تا عمل ادغام A , B با استفاده از جستجوی دودوئی و الگوریتم درج کردن صورت گیرد.

حال مرتب‌سازی ادغام را توضیح می‌دهیم. در این نوع مرتب‌سازی ابتدا آرایه n عنصری را به صورت n آرایه مرتب شده به طول یک در نظر می‌گیرد. سپس این آرایه‌های تک خانه‌ای دو به دو با هم ادغام می‌شوند تا n/2 آرایه به اندازه ۲ به دست آید (اگر n فرد باشد یک آرایه به طول یک خواهیم داشت). سپس این n/2 آرایه را دو به دو با هم ادغام می‌کنیم و این عملیات را آنقدر تکرار می‌کنیم تا نهایتاً به یک لیست مرتب شده برسیم.

شکل زیر مراحل مرتب‌سازی ادغام را نشان می‌دهد.

آرایه اولیه [20] , [40] , [22] , [31], [35] , [18], [17] , [50] , [67] , [80] , [15] , [25] , [33] , [32]

مرحله ۱ 20 , 40 , 22 , 31, 18 , 35, 17 , 50 , 67, 80 , 15 , 25, 32, 33

مرحله ۲ 20 , 22 , 31, 40 , 17 , 18 , 35, 50 , 15, 25, 67, 80 , 32, 33

مرحله ۳ 17 , 18 , 20 , 22 , 31 , 35, 40 , 50 , 15, 25, 32, 33, 67, 80

مرحله ٤ 15 , 17 , 18 , 20 , 22 , 25, 31, 32, 33, 35, 40 , 50, 67, 80

در مرتب‌سازی ادغامی بر روی آرایه فوق، ابتدا آرایه اولیه به آرایه‌های یک عنصری تبدیل می‌کند. در مرحله (۱) دو به دو آرایه تک عنصری با هم ادغام می‌شوند. در مرحله ۲، دو به دو آرایه‌های دو عنصری با هم اذغام می‌شوند و در مرحله ۳، دو به دو آرایه‌های ٤ عنصری با هم ادغام می‌شوند. و در مرحله ٤ به لیست مرتب شده می‌نویسیم.

در مثال فوق پس از k مرحله، آرایه A به چند زیرآرایه مرتب شده تجزیه می‌شود که در آن هر زیرآرایه به جز احتمالاً آخری، حاوی دقیقاً 2^k عنصر است. از این رو این الگوریتم حداکثر Logn مرحله جهت مرتب کردن آرایه n عنصری احتیاج دارد.

حال الگوریتم مرتب‌سازی ادغام را توسط دو زیر تابع و به صورت تکراری (غیربازگشتی) پیاده سازی می‌کنیم. تابع mergepass عمل ادغام را در هرمرحله انجام می‌دهد و تابع mergesort با فراخوانی مکرر mergepass آرایه را مرتب می‌کند.

الگوریتم تابع mergepass

```
void mergepass (int list [ ], int sorted [ ], int n, int length)
        {
                int i,j;
                   i=1;
                   while (i<(n-2*length +1))
                         {
                         merge (list, sorted, i,i+length , i+2 * length;
                           i=i+2*L;
                         }
                         // merge remaining list of length (2*L)
                if (i+length –1<n)
                         merge (list ,  sorted , i,i+length –1,n);
                else
                    for (j=1,i<n;j++)
                         sorted [j]=list [j];
        }
```

الگوریتم فوق روی آرایه list با n عنصر عمل می‌کند که آرایه list را از زیر آرایه‌های مرتب شده تشکیل شده است. هر زیر آرایه از length عنصر تشکیل شده است. به استثنای زیر آرایه آخری که ممکن است از length عنصر داشته باشد. نتیجه دو آرایه sorted ریخته می‌شود و حلقه while زیرآرایه‌ها را جفت به جفت با هم ترکیب می‌کند.

الگوریتم mergeresort به کمک الگوریتم فوق به صورت زیر می‌باشد.

الگوریتم مرتب سازی ادغامی

```
void mergesort (int list [], int n)
        {
            int length;
            int sorted [ ]
                {
```

```
		length =1;
		while (length <n)
			mergepass (list, sorted , n,length);
			length =2 * length ;
			mergepass (sorted , list, n,  length);
		}
	}
```

در الگوریتم فوق n تعدادعناصر آرایه list و length طول هر زیر آرایه در هر مرحله است. از آرایه sorted به عنوان آرایه واسطه‌ای استفاده کرده‌ایم. یک بار list را ادغام کرده و در sorted می‌ریزیم و دوباره sorted را ادغام کرده و در List می‌ریزیم ولی در آخر خروجی در آرایه list قرار می‌گیرد.

تحلیل پیچیدگی مرتب‌سازی ادغام

یک مرتب سازی ادغام متشکل از چندین گذر یا عبور بر روی داده‌های ورودی است. اولین گذر، لیست‌هایی با اندازه ۱ را ادغام می‌کند. در دومین گذر لیست هایی با اندازه ۲ ادغام می‌شوند و در i امین گذر لیست‌هایی با اندازه 2^{i-1} ادغام خواهند شد. بنابراین تعداد کل گذرها $[\log_2 n]$ خواهد بود. همانطور که در تابع merge نمایش داده شد می توانیم در لیست مرتب شده را در یک زمان خطی ادغام کنیم. بدین مفهوم که هرگذر را در زمانی برابر با O(n) انجام دهیم و از آنجایی که $[\log_2 n]$ گذر وجود دارد، زمان مورد نیاز O(nlogn) می‌باشد.

مرتب‌سازی ادغام معمولاً برای مرتب سازی فایلها مورد استفاده قرار می‌گیرد. هرچند که می‌توان آن را برای بردارهای موجود در حافظه اصلی نیز به کار برد. پس از این روش به صورت داخلی و خارجی قابل پیاده‌سازی است. اشکال عمده این روش نیازمندی به یک آرایه کمکی به طول n برای مرتب‌سازی می باشد. الگوریتم‌هایی که قبلاً بررسی کردیم تنها به محدودی خانه اضافی مستقل از تعداد عناصر نیاز داشتند. پس این روش مرتب‌سازی درجا نمی‌باشد و در عین حال این روش پایدار می‌باشد.

ویژگی های مرتب سازی ادغامی
✓ مرتبه اجرایی این الگوریتم همواره O(nlogn) است ✓ مرتب سازی ادغام معمولا برای مرتب کردن فایلها استفاده می شود. ✓ مشکل اصلی این روش این است که برای مرتب کردن به یک آرایه کمکی با n عنصر نیازمند است. پس این روش درجا نیست. ✓ این الگوریتم پایدار (stable) می‌باشد .

۸,۶,۷ مرتب سازی درخت دودوئی

در این روش از درختهای جستجوی دودوئی (BST) برای مرتب‌سازی استفاده می‌شود. اگر درخت BST به صورت inorder پیمایش شود، دنباله به دست آمده به صورت صعودی خواهد بود.

کارایی نسبی این روش به ترتیب اولیه داده‌ها بستگی دارد. آرایه ورودی کاملاً مرتب باشد، درخت جستجوی دودوئی به صورت درخت مورب خواهد بود. در این مورد برای اولین گره یک مقایسه انجام می‌گیرد. گره دوم به دو مقایسه،گره سوم به سه مقایسه نیاز دارد.

بنابراین تعداد کل مقایسه‌ها عبارت است از:

$$1+2+3+\ldots+n=\frac{n*(n+1)}{2}$$

تعداد مقایسه‌ها از $o(n^2)$ است.

از طرف دیگر، اگر داده‌ها طوری سازماندهی شده باشند. برای عدد خاصی مانند oc، نصفی از اعداد کوچکتر و نصف دیگر بزرگتر از a باشند، درخت متعادل ایجاد می‌گردد. در چنین موردی عمق درخت دودوئی حاصل $\log_2(n)$ خواهد بود.

تعداد گره‌های هر سطح مثل L برابر با 2^{L-1} و تعداد مقایسه‌های لازم جهت قرار دادن یک گره در سطح L برابر با L است. بنابراین تعداد کل مقایسه‌ها بین دو تعداد زیر می‌باشد:

$\sum_{l=1}^{d}2^{L-1}*(L)$, $d+\sum_{L=1}^{d-1}2^{L-1}*(L)$ که d عمق درخت می‌باشد.

می توان از طریق ریاضی نشان داد که نتیجه مجموع از o(nlogn) است.

خوشبختانه می‌توان نشان داد که اگر احتمال هر ترتیب ممکنی از ورودی یکسان در نظر گرفته شود، احتمال متعادل بودن درخت نتیجه، از متعادل نبودن آن بیشتر است. اگرچه ثابت تناسب در حالت متوسط از بهترین حالت بیشتر است، ولی زمان متوسط مرتب‌سازی درخت دودوئی از O(nlogn) است. اما در بدترین حالت (ورودی مرتب) مرتب‌سازی درخت دودوئی از $O(n^2)$ است.

توجه کنید روش مرتب‌سازی درخت دودوئی همانند مرتب سازی ادغام به فضای کمکی به طول آرایه ورودی نیاز دارد که به صورت درخت BST جلوه می‌کند. پس این روش مرتب سازی درجا نمی‌باشد. مساله پایدار بودن در این روش به علت طرفی عدم وجود عناصر با کلیدهای یکسان مطرح نمی‌باشد.

ویژگی های مرتب سازی درختی
✓ مرتبه اجرایی این الگوریتم در بهترین حالت و حالت متوسط O(nlogn) و در بدترین حالت $O(n^2)$ است ✓ مشکل اصلی این روش این است که برای مرتب کردن به یک آرایه کمکی با n عنصر نیازمند است. پس این روش درجا نیست. ✓ مساله پایدار بودن به دلیل فرض عدم وجود عناصر با کلیدهای یکسان مطرح نیست..

۸،٦،۸ مرنب کردن مبنایی (Radio sort)

مرتب کردن مبنایی روشی است که افراد بسیاری به طور شهودی از آن استفاده می‌کنند یا هنگامی که لیست بزرگی از اسامی را به صورت الفبا مرتب می‌کنیم از آن استفاده می‌کنیم. در اینجا مبنا ٢٦ است، علت آن ٢٦ حرف الفبای انگلیسی است. به طور مشخص لیست اسامی نخست بر اساس حرف اول هر اسم مرتب می‌شود. به بیان دیگر اسامی به ٢٦ دسته مرتب می‌شوند که در آن دسته اول، از اسامی‌ای تشکیل می‌شود که حرف اول آنها با B شروع می‌شود و الی آخر. در طی مرحله دوم، هر دسته بر اساس حرف دوم اسم به صورت الفبایی مرتب می‌شود و الی آخر. اگر هیچ اسمی برای مثال، بیشتر از ١٢ حرف نداشته باشد، اسامی حداکثر در ١٢ مرحله به صورت الفبایی مرتب می‌شوند.

حال این روش را برای مرتب‌سازی تعدادی عدد شرح می‌دهیم.

برای هر رقم با شروع از کم ارزش‌ترین به باارزش‌ترین رقم این اعمال را انجام می‌دهیم.

هر عدد را به ترتیبی که در آرایه قرار دارد خوانده و بر اساس ارزش رقمی که در حال پردازش است آن را در یکی از ١٠ صفر قرار می‌دهیم. سپس هر صف را با شروع از صفی که با رقم صفر شماره‌گذاری شده تا صفی که با رقم ٩ شماره‌گذاری شده است. در بردار اولیه می‌نویسیم. وقتی این عمل برای دو رقم انجام گرفت (با شروع از رقم سمت راست به سمت رقم سمت چپ) آرایه مرتب خواهد شد. توجه کنید که در این روش مرتب‌سازی ابتدا بر اساس ارقام کم ارزش صورت می‌گیرد.

مثال: شکل زیر مراحل مرتب سازی آرایه را با روش مرتب‌سازی مبنا نشان می‌دهد:

25 , 57, 48, 37, 12, 92, 86, 33

گذر اول: فقط رقم یکان اعداد را نگاه کرده و هر یک را در صف مربوطه می‌نویسیم.

صفها	Front	Rear
q[0]		
q[1]		
q[2]	12	92
q[3]	33	
q[4]		
q[5]	25	
q[6]	86	
q[7]	57	37
q[8]	48	
q[9]		

آرایه بعد از گذر اول : 12,92,33,25,86,57,37,48

گذر دوم: اعداد آرایه به دست آمده را بر اساس رقم دوم در یکی از صفها قرار می‌دهیم.

صفها	Front	Rear
q[0]		
q[1]	12	
q[2]	25	
q[3]	33	37
q[4]	48	
q[5]	57	
q[6]		
q[7]		
q[8]	86	
q[9]	92	

آرایه بعد از گذر دوم : 12,25,33,37,48,57,86,92

چون اعداد ۲ رقمی بودند، در ۲ گذر آرایه مرتب شد. اگر اعداد ۵ رقمی بودنـد، بـه 5 گـذر نیـاز داریـم. الگوریتم این مرتب‌سازی به صورت زیر است:

الگوریتم مرتب سازی عددی(مبنایی)

```
for (i=1,i<=5;i++)
  {
        for(j=0;j<n;j++)
          {
            k=ith digit of x[j];
            place x[j] at rear of q[k];
        }
        for (j=0; j<10;j++)
              place element of q[j] in next sequential position of x;
        }
```

پیچیدگی مرتب کردن مبنایی

فرض کنید A لیست A عنصری $A_1, A_2, ..., A_n$ داده شده است.فرض کنید d نمایش مبنا باشد. مثلاً برای ارقام دهدهی d=10، برای حروف‌ها d=26 و برای بیتها d=2 است، همچنین فرض کنید هر عنصر A_i با s رقم زیر نمایش داده می‌شود.

$$A_i = d_{i1} d_{i2} \dots d_{is}$$

الگوریتم مرتب کردن مبنایی نیازمند s مرحله، یعنی تعداد ارقام هـر عنـصر اسـت. در مرحلـه k هـر رقـم d_{ik} با هر یک از d رقم مقایسه می‌شود. از این رو C(n) تعداد مقایسه‌ها برای الگوریتم بـه صـورت زیـر است:

$$C(n) \le d * s * n$$

اگرچه d مستقل از n است اما s به n بستگی دارد. بدترین حالت، s=n، از این رو $c(n)=o(n^2)$. در بهترین حالت $s=\log_d^n$، از این رو $c(n)=o(n\log n)$ به بیان دیگر، مرتب کردن مبنایی تنها وقتی خوب اجرا می‌شود که s تعداد ارقام در این نمایش A_i ها کوچک باشد.
عیب دیگر مرتب سازی مبنایی این است که ممکن است به $d*n$ خانه حافظه احتیاج داشته باشد. این عیب را می توان با استفاده از لیست‌های پیوندی به جای آرایه، به حداقل رساند. با وجود این همچنان به 2*n خانه حافظه نیازمندیم.

ویژگی های مرتب سازی مبنایی
✓ مرتبه اجرایی این الگوریتم در بهترین حالت و حالت متوسط O(nlogn) و در بدترین حالت $O(n^2)$ است ✓ اگر اعداد یا حروف s رقمی باشند الگوریتم به s گذر نیاز دارد تا ورودی را مرتب کند.

۸,۷ مقایسه روش های مرتب سازی
از چندین روش مرتب‌سازی ارایه شده هیچکدام روش مناسب و خوبی نیستند. برخی از روشها برای مقادیر کوچک n و برخی دیگر برای مقادیر بزرگ n مناسب هستند. مرتب‌سازی در جی زمانی که لیست به صورت جزئی مرتب شده باشد، خوب کار می‌کند و از آنجا که این روش حداقل سرباری را دارد برای مقادیر کوچک n مناسب است. مرتب‌سازی ادغام بهترین روش برای بدترین حالت می‌باشد. اما آن بیشتر از heapsort به حافظه نیاز دارد و سربازی آن بیشتر از مرتب‌سازی سریع می‌باشد. مرتب سازی سریع بهترین میانگین را دارد، اما در بدترین حالت، زمان آن از $o(n^2)$ خواهد شد. عملکرد مرتب‌سازی مبنا بستگی به کلید و انتخاب مبنا دارد.
آزمایش‌هایی که روی روشهای مرتب انجام شده است، نشان می‌دهد که به ازای $n \le 20$ مرتب‌سازی درجی سریعتری روش است. برای مقادیر یعنی 20 و تا 45 مرتب‌سازی سریع بهترین و سریعترین می‌باشد. برای مقادیر بزرگتر از A، مرتب‌سازی ادغام سریعترین می‌باشد. در عمل مناسب است که سه مرتب‌سازی فوق را با هم ترکیب کنیم به طوری که مرتب سازی ادغام برای زیرلیست‌های کمتر از ۴۵ از مرتب‌سازی سریع استفاده کند و مرتب‌سازی سریع نیز زمانی که طول زیرلیست‌ها کمتر از ۲۰ باشد از مرتب‌سازی درجی استفاده می‌کند.
حال این مقایسه را از دیدگاه دیگری نیز بیان می‌کنیم.
کارایی مرتب سازی درجی از مرتب‌سازی حبابی بیشتر است. مرتب‌سازی انتخابی نسبت به مرتب‌سازی درجی به انتسابهای کمتر و مقایسه‌های بیشتر نیاز دارد. لذا در فایل‌های کوچک که رکوردها بزرگ و کلیه‌ها ساده هستند، مرتب‌سازی انتخابی پیشنهاد می‌گردد. علتش این است که در فایلی با این خصوصیات

عمل انتساب رکوردها گران نبوده و عمنل مقایسه کلیدها ارزان تمام می شـود. اگـر عکـس ایـن وضـعیت برقرار باشد، مرتب‌سازی درجی پیشنهاد می‌گردد. اگر ورودی در لیست پیوندی باشد، حتی اگـر رکوردهـا بزرگ باشند ، مرتب‌سازی درجی پیشنهاد می‌گردد. زیرا نیاز به جابجایی داده‌ها نیست. البته کـارایی مرتـب heapsort و quicksort برای مقادیر بزرگ n از کارایی مرتب‌سازی‌های درجی و انتخابی بیشتر است.

در حال متوسط heapsort کارایی quicksort را ندارد. تجربه‌ها نشان می‌دهند کـه در ورودی تـصادفی، زمان لازم در heapsort دو برابـر زمـان لازم در quicksort اسـت. امـا در بـدترین حالـت heapsort مناسب‌تر از quicksort است. Heapsort همچنین برای مقادیر کوچک n مفید نیست. علتش این اسـت که ایجاد اولیه heap و محاسبه محل پدر و پسرهای گره‌ها مستلزم وقت قابل ملاحظه‌ای است.

در جدول زیر مرتبه زمانی الگوریتم‌های مرتب‌سازی بیان شده است.

نام الگوریتم	بهترین حالت	حالت متوسط	بدترین حالت	ویژگی
مرتب‌سازی حبابی	$O(n)$	$O(n^2)$	$O(n^2)$	پایداراست و درجا
مرتب سازی درجی	$O(n)$	$O(n^2)$	$O(n^2)$	پایدار است و درجا
مرتب‌سازی انتخابی	$O(n^2)$	$O(n^2)$	$O(n^2)$	پایدارنیست و درجا
مرتب سازی سریع	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	پایدارنیست و درجا
مرتب سازی ادغام	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	پایداراست و غیردرجا
مرتب سازی هرمی	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	پایدار نیست و درجا
مرتب سازی درختی	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	غیردرجا

تمرین های فصل

۱) عمل اصلی در الگوریتم مرتب سازی انتخابی، پیمایش لیست $x_1,x_2,...,x_n$ است تا کوچکترین عنصر پیدا شود و در ابتدای لیست قرار گیرد. روش دیگر انجام این کار این است که کوچکترین و بزرگترین عنصر پیدا شوند، کوچکترین عنصر در ابتدا و بزرگترین عنصر در انتهای لیست قرار گیرد. در مرحله بعد، این کار برای لیست $x_2,...x_{n-1}$ انجام می شود و غیره

الف)تابعی برای پیاده سازی این روش مرتب سازی بنویسید.

ب)زمان تابع را محاسبه کنید

ج)با استفاده از یک آرایه فرضی این روش مرتب سازی را نشان دهید.

۲) یک تابع بازگشتی برای مرتب سازی انتخابی بنویسید.

۳) یک تابع بازگشتی برای مرتب سازی درجی بنویسید.

٤) یک الگوریتم حبابی بنویسید که برای لیست پیوندی مناسب باشد.

۵) برای اعداد زیر، مرتب سازی درجی را دنبال کنید:

a. 13, 57, 85, 70, 22, 64, 48
b. 13, 57, 12, 39, 40, 54, 55, 2, 68
c. 70, 57, 99, 34, 56, 89

٦) برای اعداد زیر، مرتب سازی حبابی را دنبال کنید:

a. 13, 57, 85, 70, 22, 64, 48
b. 13, 57, 12, 39, 40, 54, 55, 2, 68
c. 70, 57, 99, 34, 56, 89

۷) برای اعداد زیر، مرتب سازی انتخابی را دنبال کنید:

a. 13, 57, 85, 70, 22, 64, 48
b. 13, 57, 12, 39, 40, 54, 55, 2, 68
c. 70, 57, 99, 34, 56, 89

۸) برای اعداد زیر، مرتب سازی سریع را دنبال کنید:

a. 13, 57, 85, 70, 22, 64, 48
b. 13, 57, 12, 39, 40, 54, 55, 2, 68
c. 70, 57, 99, 34, 56, 89

۹) برای اعداد زیر، مرتب سازی هرمی را دنبال کنید:

a. 13, 57, 85, 70, 22, 64, 48
b. 13, 57, 12, 39, 40, 54, 55, 2, 68
c. 70, 57, 99, 34, 56, 89

۱۰) برای اعداد زیر، مرتب سازی درختی را دنبال کنید:

a. 13, 57, 85, 70, 22, 64, 48
b. 13, 57, 12, 39, 40, 54, 55, 2, 68
c. 70, 57, 99, 34, 56, 89

۱۱) برای اعداد زیر، مرتب سازی ادغام را دنبال کنید:

a. 13, 57, 85, 70, 22, 64, 48
b. 13, 57, 12, 39, 40, 54, 55, 2, 68
c. 70, 57, 99, 34, 56, 89

۱۲) برای اعداد زیر، مرتب سازی مبنایی را دنبال کنید:(یکبار به صورت صعودی و بار دیگر به صورت نزولی)

a. 13, 57, 85, 70, 22, 64, 48
b. 13, 57, 12, 39, 40, 54, 55, 2, 68
c. 70, 57, 99, 34, 56, 89

۱۳) تابعی بنویسید که ادغام سه طرفه را پیاده سازی کند. یعنی سه فایل مرتب را در یک فایل دیگر به طور مرتب ادغام کند.

۱٤) اثبات کنید که بهترین حالت ممکن برای هر الگوریتمی که n عنصر را مرتب می کند(O(nlogn می باشد.

۱۵) نشان دهید که هر فرایندی که یک فایل را مرتب می کند می تواند برای پیدا کردن موارد تکراری در فایل توسعه داده شود.

۱٦) نشان دهید که اگر k، کوچکترین مقدار صحیح بزرگتر یا مساوی n+logn-2 باشد، برای بزرگترین عنصر و دومین عنصر از نظر بزرگی در مجموعه n عنصری لازم و کافی است که k مقایسه انجام گیرد.

۱۷) ثابت کنید تعداد گذرهای لازم در مرتب سازی حبابی، قبل از این که فایل مرتب شود(غیر از آخرین گذر، که کشف می کند فایل مرتب شده است) برابر با بزرگترین فاصله ای است که یک عنصر از یک اندیس بزرگتر به اندیس کوچکتر منتقل شود.

۱۸) مرتب سازی با شمارش به این صورت انجام می گیرد: آرایه ای به نام count تعریف کنید و count[i] را برابر با تعداد عناصری که کوچکتر از x[i] هستند قرار دهید. سپس x[i] را در موقعیت count[i] یک آرایه خروجی قرار دهید.(اما، مواظب تساوی عناصر باشد) یک تابع بنویسید که یک آرایه x به اندازه n را به این روش مرتب نماید.

۱۹) مرتب سازی جابجایی <<فرد – زوج>> به این صورت انجام می گیرد: در سراسر فایل چند بار گذر کنید. در گذر اول x[i] را با x[i+1] برای کلیه مقادیر فرد i مقایسه کنید. در گذر دوم، x[i] را با x[i+1] برای کلیه مقادیر زوج i مقایسه کنید. هر وقت x[i]>x[i+1] جای آنها را باهم عوض کنید. این فرایند را تا مرتب شدن فایل ادامه دهید.

الف) شرط پایان روش مرتب سازی چیست؟

ب) این روش مرتب سازی را توضیح دهید؟

ج) کارایی این روش در حالت متوسط چگونه است؟

۲۰) روش پیدا کردن عنصر pivot در مرتب سازی سریع به این صورت تغییر دهید که مقدار میانی عنصر اول، عنصر میانی و عنصر آخر را به کار ببرد. در چه مواردی استفاده از این روش مرتب

سازی نسبت به مرتب سازی مطرح شده در متن کارآمادتر است؟ در چه مواردی کـارایی کمتـری دارد؟