ساختمان داده‌ها و الگوریتم‌ها

(رشتهٔ کامپیوتر)

مؤلفان:

مهندس جعفر تنها

مهندس ناصر آیت

ناشر: دانشگاه پیام‌نور

۱۳۸۷

فهرست

پیشگفتار

خداوند منان را شکر می‌گوییم که با اعطای نعمت حیات، سلامتی، دانش و عنایت خاص او، توفیق آن را یافتیم تا کتابی را تحت عنوان ساختمان داده‌ها و الگوریتم‌ها به دانش‌پژوهان و دانشجویان تقدیم کنیم. با توجه به نیاز مبرم دانشجویان رشته علوم و مهندسی کامپیوتر، مهندسی فناوری اطلاعات و ریاضی کاربردی به داشتن منبعی برای درس ساختمان داده‌ها، بر آن شدیم که تجربه چندین ساله خود در زمینه موضوع مذکور را در قالب کتابی در اختیار دانش‌پژوهان و دانشجویان عزیز قرار دهیم. وجه تمایز این مجموعه با منابع موجود، علاوه از خودآموز بودن آن، تکمیل بودن سرفصل درسی و تحت پوشش قرار دادن کامل واحد درس ساختمان داده‌ها و الگوریتم‌ها می‌باشد. در این مجموعه سعی شده نخست ADT یا نوع داده مجرد هر ساختار داده شده و سپس روش پیاده‌سازی و پیچیدگی زمانی هر کدام از آنها بررسی شده است. به خصوص پیچیدگی زمانی در حالات مختلف مسئله بیشتر مورد بحث واقع شده است. در نهایت در پایان هر فصل تعدادی مثال یادگیری بیشتر ارائه گردیده است.

آنچه در پیش روی شماست، مجموعه‌ای با هشت فصل می‌باشد که بطور جامع و کامل در اختیار دانشجویان عزیز قرار داده شده است. در فصل اول سعی شده خواننده با مطالعه آن، بتواند راه‌حل‌های عملی لازم را برای محاسبه تابع و پیچیدگی زمانی الگوریتم فرا بگیرد و بخصوص بتواند مهارت‌های لازم جهت محاسبه و حل

روابط بازگشتی را کسب نماید. در این فصل راه‌حل‌های، حل روابط(بازگشتی و غیربازگشتی) بحث شده است و از مهمترین فصل کتاب محسوب می‌شود. در فصل دوم اولین و تقریباً ساده‌ترین ADT را ارائه دادیم و در ادامه مسائل زیادی را با این ADT، بحث کردیم. در فصل‌های سوم و چهارم نیز ADT های جدیدی را ارائه کردیم و در کل خروجی روش‌های مختلف را با هم مقایسه کردیم. در فصل پنجم، لیست‌های پیوندی را بعنوان یک ساختار داده مورد بررسی قرار دادیم.

فصل ششم را با یک ADT جدیدی بنام درخت شروع کردیم و انواع ساختار درخت‌ها را بررسی کردیم. در نهایت آخرین ADT را در فصل هفتم ارائه دادیم که مربوط به گراف و کاربردهای آن می‌باشد.

فصل هشتم یکی از مهمترین فصل‌های کتاب می‌باشد، برای اینکه انواع الگوریتم‌های مرتب‌سازی را مورد ارزیابی قرار داده است. بخصوص اینکه انواع الگوریتم‌ها را از نظر پیچیدگی زمانی با هم مقایسه می‌کند.

در انتهای هر فصل علاوه از تعدادی تمرین برای تکمیل سناریو آموزشی، یک یا چند پروژه برنامه‌نویسی برای یادگیری کامل مفاهیم گنجانده شده است.

امید است دانش‌پژوهان و دانشجویان عزیز پس از مطالعه این مجموعه مهارتهای لازم را برای درس ساختمان داده‌ها و الگوریتم‌ها کسب نمایند.

در پایان، مؤلفین وظیفه خود می‌دانند که از تمامی عزیزان، که در تألیف و نگارش این مجموعه زحمات زیادی را متحمل شده‌اند، تشکر و قدردانی نمایند، همچنین نهایت تشکر و قدردانی خود را از آقایان مهندس محمدجواد رستمی، غلامرضا حلمی‌پسند و حمیدرضا شریفی که زحمت ویراستاری ادبی این مجموعه را عهده دار شده‌اند، اعلام داریم.

سخن را با تشکر از تمامی دانش‌پژوهان و دانشجویان عزیزی که با ارسال یادداشت، نظرات خود را به منظور بهتر و پربار کردن این کتاب برای مولفین ارسال می‌کنند تا در چاپ‌های دیگر مورد استفاده قرار گیرد، به پایان رسانده و این مجموعه را تقدیم به پدران بزرگوار و مادران عزیزی می‌کنیم که در همه حال در فکر تحصیل و سعادت فرزندان خود بوده و چراغ راه زندگی فرزندانشان می‌باشند و از خداوند منان

مسئلت داریم، این خدمت خالصانه را در راه رضای خود تلقـی فرمایـد و مـا را یـاری نموده و نیرویی مضاعف مرحمت فرمایـد تـا بتـوانیم در خـدمت بـه دانـش‌پژوهـان و دانشجویان به پاره‌ای از آنچه آرزو داریم، تحقق بخشیم.

مؤلفین
مهندس جعفرتنها tanha@pnu.ac.ir
مهندس ناصر آیت ayat@pnu.ac.ir

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.

سؤال‌های پیش از درس

۱. به نظر شما چگونه می‌توان فهمید یک برنامه نوشته شده از برنامه مشابه دیگر بهتر عمل می‌کند؟

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 می‌شود؟

مقدمه

سؤالی که در مورد یک الگوریتم یا الگوریتم‌های یک مسئله مطرح می‌شود اینست که کدام الگوریتم برای حل یک مسئله خاص بهتر عمل می‌کند؟ پاسخ دادن به این سؤال به‌راحتی امکان‌پذیر نیست. مشخصه‌های زیادی از جمله سادگی، وضوح ، زمان اجرا ، میزان حافظه مصرفی و غیره برای یک الگوریتم خوب می‌باشند. در این میان، زمان اجرا و میزان حافظه مصرفی نقش بسیار مهمی ایفا می‌کنند (در این کتاب بیشتر زمان اجرا مورد بحث قرار می‌گیرد)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 می‌باشدو 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 می‌بایست دارا باشد:

- 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 می‌کنیم.

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 می‌شود، 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 باشدالبته 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 به‌صورت زیر محاسبه کرد:

سطر	زمان	تعداد
1	$C_۱$	۱
2	$C_۲$	$n+۱$
3	$C_۳$	n

با توجه به جدول، $T(n)$ برابر است با:

$$T(n) = C_۱ + C_۲(n+۱) + C_۳ n$$

حال C را بیشترین مقدار $C_۱$، $C_۲$، $C_۳$ در نظر می‌گیریم بنابراین:

$$T(n) = C(۲n+۲)$$

حال قطعه کد زیر را در نظر 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 به‌صورت 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 بالا، تأثیرگذاری کمتری دارند. بنابراین هدف ما از محاسبه مرتبه یک الگوریتم به‌دست آوردن زمان، در تعداد تکرارهای بزرگ یا خیلی بزرگ می‌باشد. در حالت کلی ضرایب، تأثیر چندانی در زمان اجرا ندارندبه همین دلیل غالبا ً از آنها در محاسبات صرف‌نظر می‌کنند.

مثال ۱-۱: 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 به‌صورت زیر محاسبه می‌شود:

سطر	هزینه	تعداد
2	C_1	۱
3	C_2	n
4	C_3	$n-1$
5	C_4	۱

بنابراین $T(n)$ برابر است با:

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

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

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

مثال ۲-۱: تابع زیر مربوط به حاصل‌جمع دو ماتریس می‌باشد (a,b,c سه ماتریس n*m می‌باشند):

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

تابع زمانی، الگوریتم بالا به صورت 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.

۱-۲-۱ نماد Big-oh

برای بررسی میزان رشد توابع زمانی الگوریتم‌ها، نماد Big-oh را به‌کار 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 $n_\circ$ وجود داشته باشند که برای همه مقادیر $n \geq n_\circ$، داشته باشیم $T(n) \leq Cf(n)$ (رابطه $T(n) \in O(f(n))$ را بخوانید $T(n)$ متعلق به اوی بزرگ $f(n)$).

تعریف بالا به‌صورت زیر نیز بیان می‌شود:

$T(n) \in O\big(f(n)\big) \Leftrightarrow \exists C, n_\circ > \circ$ به‌طوری‌که $\forall n \geq n_\circ \quad T(n) \leq Cf(n)$

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

در حالت کلی $f(n)$ مرتبه زمانی اجرای الگوریتم نامیده می‌شود (اصطلاحاً 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:

$T(n) = C(\text{۲}n + \text{۲})$

C زمان اجرای عملیات، یک مقدار ثابت است با فرض $C = \text{۱}$ خواهیم داشت (قبلاً اشاره شد که C به نوع سخت‌افزار، زبان برنامه‌نویسی و غیره بستگی دارد):

$$\begin{aligned} T(n) &= \text{۲}n + \text{۲} \\ &\leq \text{۳}n \Rightarrow T(n) \in O(n) \end{aligned}$$

که در آن $n_\circ = \text{۲}$ و $C = \text{۳}$ می‌باشد. بنابراین بازای n_0 و C مشخص $T(n) \in O(n)$ خواهد بود.

مثال ۳-۱: زمان اجرای $T(n)$ مربوط به تعدادی الگوریتم موجود است مرتبه یا پیچیدگی زمانی این الگوریتم‌ها را محاسبه نمایید:

i) $T_\text{۱}(n) = \text{۲}n^\text{۲} + \text{٤}n$

ii) $T_\text{۲}(n) = \text{۳}n^\text{۳} + \text{۳}n$

iii) $T_\text{۳}(n) = \text{٤}n + \text{٥}n\,\mathrm{Log}n + \text{۲}$

حل:

i) $T_1(n) = ۲n^۲ + ٤n \le ۳n^۲$

که در آن اگر $C = ۳$ و $n_\circ = ٤$ باشد آنگاه $T_1(n) \in O(n^۲)$ می‌باشد.

ii) $T_۲(n) = ۳n^۳ + ۳n$

$\le ٤n^۳$

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

iii) $T_۳(n) = ٤n + ٥n \operatorname{Log} n + ۲ \le ٤n \operatorname{Log} n + ٥n \operatorname{Log} n + ۲n \operatorname{Log} n$

$= ۱۱n \times \operatorname{Log} n$

که در آن اگر $C = ۱۱$ و $n_\circ = ۱$ باشد، آنگاه $T_۳(n) \in O(n \operatorname{Log} n)$ خواهد بود.

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

نکته: وقتی رابطه $T(n) \in O(F(n))$ برقرار باشد،گوییم $F(n)$ یک کران بالا برای $T(n)$ می‌باشد.

مثال ٤-۱: زمان اجرای $T(n)$ مربوط به مثال **۱-۱** و مثال **۲-۱ موجود** است مرتبه یا پیچیدگی زمانی این الگوریتم‌ها را محاسبه نمایید:

i) $T(n) = C(۲n + ۱)$

ii) $T(n) = C(۲nm + ۲n + ۱)$

حل:

i) $T(n) = C(۲n + ۱)$

C زمان اجرای عملیات، یک مقدار ثابت است با فرض $C = ۱$ خواهیم داشت:

$T(n) = C(۲n + ۱) = ۲n + ۱$

$\le ۳n$

که در آن اگر $C = ۳$ و $n_\circ = ۱$ باشد آنگاه $T(n) \in O(n)$ می‌باشد.

ii) $T(n) = C(۲nm + ۲n + ۱)$

C زمان اجرای عملیات، یک مقدار ثابت است با فرض $C = ۱$ و $m<n$ خواهیم داشت:

$T(n) \le (۲n^۲ + ۲n + ۱)$

$\le ۳n^۲$

که در آن اگر $C=3$ و $n_0=3$ باشد آنگاه $T(n)\in O(n^2)$ خواهد بود.

مثال ۵-۱: 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:

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

$\le 2n^2$

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.

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

$\le 6n^2$

همانطور که ملاحظه می‌کنید نمی توان C و n_0معرفی کرد که رابطه

$5n^2+n+1\le Cn$

به ازای $n\ge n_0$, C همواره برقرار باشد به عبارت دیگر T(n)به هیچ وجه در حالت کلی نمی‌تواند کمتر از Cn باشد. در نتیجه رابطه بالا یک رابطه نادرست می‌باشد.

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

$\le 5*2^n$

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 ۱-۱: اگر $T(n)=a_m n^m + a_{m-1} n^{m-1} + \ldots + a_1 n + a_0$ 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 ۶-۱: 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) $T(n) = ۳n^۳ + ۲n^۲ \le ۵n^۳$

با توجه به تعریف به ازای $n_\circ = ۱$ و $C = ۵$ رابطه برقرار است.

iii) $T(n) = ۵^n \notin O\left(۲^n\right)$

فرض کنید C و $n_\circ$ موجود است به‌طوری‌که به ازای هر $n \ge n_\circ$ داشته باشیم:

$$۵^n \le C۲^n$$

آنگاه عبارت $C \ge \left(\frac{۵}{۲}\right)^n$ 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.

۲-۲-۱ نماد Big-Omega

تعریف: گوئیم $T(n) \in \Omega(f(n))$ اگر و فقط اگر ثابت صحیح C و n_0 وجود داشته باشد که به ازای همه مقادیر $n \ge n_\circ$ داشته باشیم $T(n) \ge Cf(n)$ (رابطـه $T(n) \in \Omega(f(n))$ را بخوانید $T(n)$ امگای بزرگ $f(n)$).

تعریف بالا را به‌صورت زیر نیز ارائه می‌دهند:

$$T(n) \in \Omega(f(n)) \Leftrightarrow \exists C, n_\circ > ۰ \quad \forall n \ge n_\circ \qquad Cf(n) \le T(n)$$

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.

مثال ۷-۱: زمان اجرای $T(n)$ الگوریتمی محاسبه شده، $\Omega(f(n))$ آن‌را به‌دست آورید.

$$T(n) = an^۲ + bn + c \quad \text{and} \quad a, b, c > \circ$$

حل:

$$an^۲ + bn + c > an^۲ + b + c$$
$$> an^۲$$

بنابراین اگر $n_\circ = ۱$ و $C = a$ باشد، آنگاه $T(n) \in \Omega(n^۲)$ خواهد بود.

مثال ۸-۱: زمان اجرای $T(n)$ الگوریتمی محاسبه شده، $\Omega(f(n))$ آنرا به‌دست آورید.

$$T(n) = n^٤ + ٥n^۲$$

حل:

$$T(n) = n^٤ + ٥n^۲ \geq n^٤$$

بنابراین اگر $n_\circ = ۱$ و $C = ۱$ باشد، آنگاه $T(n) \in \Omega(n^٤)$ خواهد بود.

مثال-۹: 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) $T(n) = ۳n + ۲ \notin \Omega(n^۲)$

iii) $T(n) = ٥^n + n^۲ \in \Omega(۲^n)$

حل:

i) $T(n) = ٦n + ٤ \geq ٦n$

بنابراین اگر $n_\circ = ۱$ و $C = ٦$ باشد، آنگاه $T(n) \in \Omega(n)$ خواهد بود.

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

فرض کنید C و $n_\circ$ موجود است به‌طوری‌که به ازای هر $n \geq n_\circ$ داشته باشیم:

$$T(n) = ۳n + ۲ \geq Cn^۲$$

$$\Rightarrow Cn^۲ - ۳n - ۲ \leq \circ$$

همان‌طور که ملاحظه می‌کنید در نامعادله بالا C و $n_\circ$ با مقدار معین وجود ندارد به‌طوریکه بازای هر $n \geq n_\circ$ رابطه بالا برقرار باشد. بنابراین $۳n + ۲ \notin \Omega(n^۲)$ خواهد بود.

iii) $T(n) = ٥^n + n^۲ \geq ٥^n \geq ۲^n$

بنابراین اگر $n_{\circ} = ۱$ و $C = ۱$ باشد، آنگاه $T(n) \in \Omega(۲^n)$ خواهد بود.
حال در حالت کلی قضیه زیر را ارائه می‌دهیم.

قضیه-۲: اگر $T(n) = a_m n^m + a_{m-۱} n^{m-۱} + \ldots + a_۱ n + a_{\circ}$ زمان اجرای یک الگوریتم بوده و $a_m > 0$ باشد آنگاه $T(n) \in \Omega(n^m)$.
اثبات: به‌عنوان تمرین به خواننده واگذار می‌شود.

۳-۲-۱ نماد θ

تا حالا یک کران پایین و یک کران بالا برای تابع زمانی یک الگوریتم توسط نمادهای Ω و O ارائه دادیم. حال نماد θ را به‌صورت زیر تعریف می‌کنیم:

تعریف: گوئیم $T(n) \in \theta(f(n))$ اگر و فقط اگر ثابتهای $C_۱$، $C_۲$ و ثابت صحیح $n_{\circ}$ وجود داشته باشد به‌طوری‌که برای همه مقادیر $n \geq n_{\circ}$:

$$C_۱ f(n) \leq T(n) \leq C_۲ f(n)$$

تعریف بالا به‌صورت زیر نیز ارائه می‌شود:

$$\exists C_۱, C_۲, n_{\circ} > \circ \text{ به‌طوری‌که } \forall n \geq n_{\circ} \quad C_۱ f(n) \leq T(n) \leq C_۲ f(n)$$

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

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

مثال ۱۰-۱: اگر $T(n) = \frac{۱}{۲} n^۲ - ۳n$ باشد $\theta(f(n))$ را محاسبه کنید.

حل:

نشان می‌دهیم که $T(n) \in \theta(n^۲)$.

طبق تعریف: $T(n) \in \theta(n^۲) \quad \Leftrightarrow$

$$\exists C_۱ و C_۲ و n_{\circ} \text{ به‌طوری‌که } \forall n \geq n_{\circ} \qquad C_۱ n^۲ \leq \frac{۱}{۲} n^۲ - ۳n \leq C_۲ n^۲ \quad (۱)$$

حال رابطه (۱) را به $n^۲$ تقسیم می‌کنیم: $C_۱ \leq \frac{۱}{۲} - \frac{۳}{n} \leq C_۲$

با توجه به عبارت بالا، طرف راست، به ازای $C_2 \geq \frac{1}{2}$ و $n \geq 1$ برقرار است.
به‌همین ترتیب برای طرف چپ عبارت بالا به ازای $n \geq 7$، $C_1 \leq \frac{1}{4}$ حاصل می‌شود.
بنابراین به ازای $C_1 = \frac{1}{4}$ و $C_2 = \frac{1}{2}$ و $n_0 \geq 7$ عبارت $T(n) \in \theta(n^2)$ خواهد بود.

مثال ۱۱-۱: فرض کنید $T(n) = 7n^3$ باشد. آیا $T(n) \in \theta(n^2)$ می‌تواند باشد.

حل: طبق تعریف θ باید داشته باشیم:

$\exists C_1, C_2, n_0 > 0$ به‌طوری‌که $\forall n \geq n_0 \quad C_1 n^2 \leq 7n^3 \leq C_2 n^2$

طرف چپ رابطه بالا همیشه برقرار است اما طرف راست رابطه بالا در صورتی برقرار است که $n \leq \frac{C_2}{7}$ باشد 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.

مثال ۱۲-۱: زمان اجرای $T(n)$ مربوط به تعدادی الگوریتم موجود است. درستی عبارات زیر را ثابت کنید.

i) $T(n) = 6n + 4 \in \theta(n)$

ii) $T(n) = 3n + 6 \notin \theta(n^2)$

حل: طبق تعریف θ باید داشته باشیم:

$\exists C_1, C_2, n_0 > 0$ به‌طوری‌که $\forall n \geq n_0 \quad C_1 n \leq 6n + 4 \leq C_2 n$

با توجه به رابطه بالا، طرف راست بازای مقادیر $C_2 = 7$ و $n_0 = 2$ برقرار می‌باشد و طرف چپ رابطه بالا، بازای $C_1 = 6$ و $n_0 = 1$ برقرار خواهد بود.
بنابراین به ازای $C_1 = 6$ و $C_2 = 7$ و $n_0 \geq 2$ عبارت $T(n) \in \theta(n)$ خواهد بود.

ii) $T(n) = 3n + 6 \notin \theta(n^2)$

طبق تعریف θ باید داشته باشیم:

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

طرف راست رابطه بالا همیشه برقرار است اما طرف چپ رابطه بالا بازای هر n برقرار نیست و این با تعریف θ که در آن رابطه برای هر n بزرگ‌تر از n_0 برقرار است

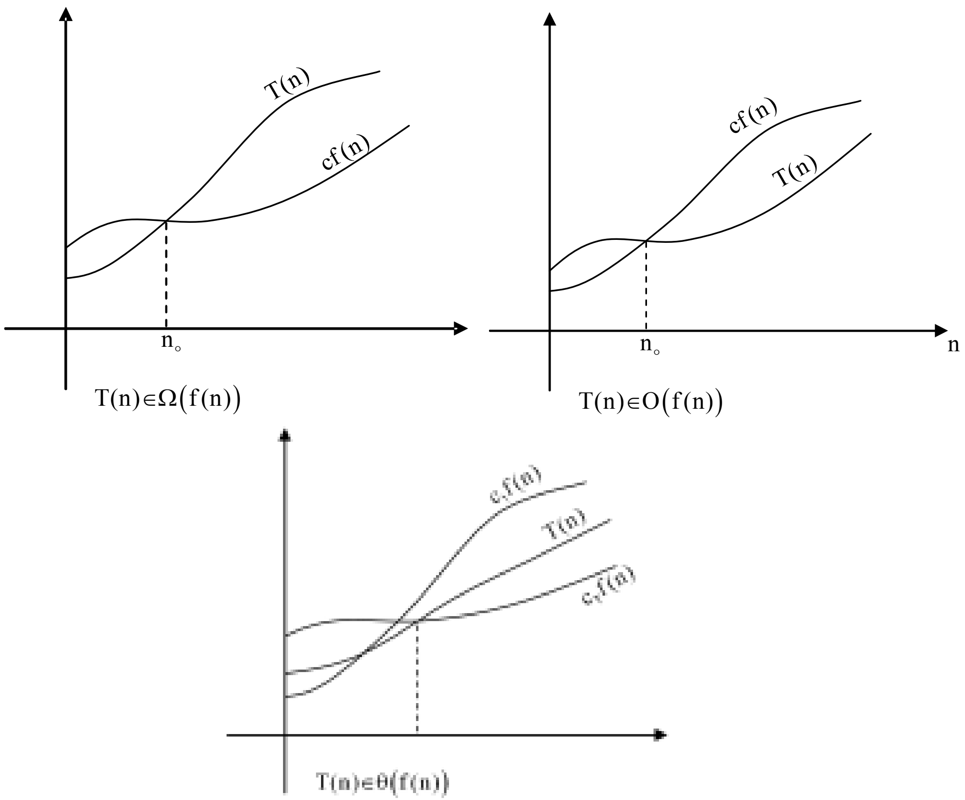
منافات دارد، لذا $T(n)$ نمی‌تواند متعلق به $\theta(n^۲)$ باشد.

قضیه **۳-۱:** اگر $T(n) = a_m n^m + a_{m-۱} n^{m-۱} + \ldots + a_۱ n + a_۰$ زمان اجرای یک الگوریتم بوده و $a_m > 0$ باشد آنگاه $T(n) \in \theta(n^m)$.

اثبات: به‌عنوان تمرین به خواننده واگذار می‌شود.

شکل زیر نمادهای Big-oh، Big-Omega و θ را با استفاده از نمودار نمایش دهید.

با توجه به تعاریف نمادهای بالا نمودار را ترسیم می‌کنیم در این نمودارها $T(n)$ تابع زمانی و $F(n)$ پیچیدگی زمانی تابع $T(n)$ می‌باشد.



شکل ۱-۱ نمودار نمادهای θ, Ω, O

۴-۲-۱ مرتبه رشد

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

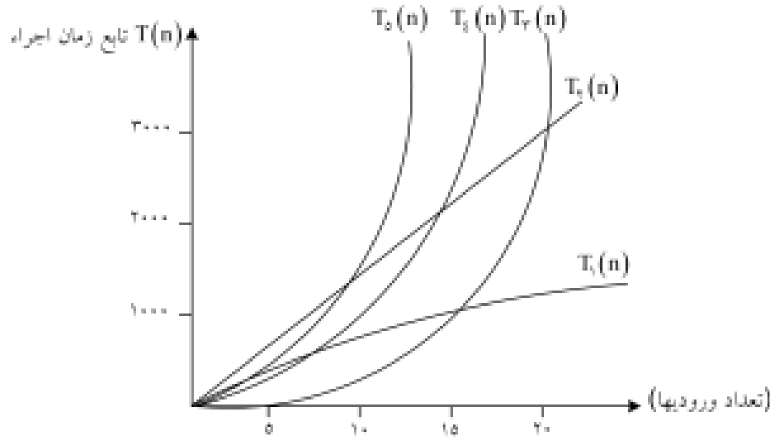
فرض کنید که توابع زمان اجرای برنامه‌ها به صورت زیر باشند:

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

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

$T_5(n) = 2^n$

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



شکل ۲-۱ 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 به‌صورت لگاریتمی می‌باشد، نسبت به بقیه خیلی بهتر عمل می‌کنند

در جدول ۱-۱ زمان اجرای چهار برنامه با پیچیدگی زمانی متفاوت نشان داده

می‌شود. ماشین و کامپایلری که برای اجرای این برنامه‌ها به‌کار برده شده، خاص بوده و فرض می‌کنیم که معیار اجراء براساس ثانیه بوده و در مرحله اول اجراء با تعداد ورودی، ۱۰۰۰ ثانیه زمان مصرف می‌شود.

بیشترین اندازه مسئله	بیشترین اندازه مسئله برای $۱۰^٤$ ثانیه	بیشترین اندازه مسئله برای $۱۰^۳$ ثانیه	زمان اجراء $T(n)$
۱۰۰۰	۱۰۰	۱۰	$۱۰۰n$
۳۲۰	٤۵	۱٤	$۵n^۲$
۲۳۰	۲۷	۱۲	$n^۳/۲$
۱۳۰	۱۳	۱۰	۲^n

جدول ۱-۱: 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 به‌صورت نمایی می‌باشد.

۱-۳ روش‌های تحلیل الگوریتم‌ها

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

قبلاً اشاره کردیم که الگوریتم‌ها را براساس زمان اجراء و میزان حافظه مصرفی با هم مقایسه می‌کنند. (در این کتاب الگوریتم‌ها را براساس زمان اجرا 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 به‌کار می‌بریم به دو دسته اصلی تقسیم می‌شوند:

۱. الگوریتم‌های ترتیبی (غیر بازگشتی)
۲. الگوریتم‌های بازگشتی

۱-۳-۱ الگوریتم‌های ترتیبی (غیر بازگشتی)

برای به‌دست 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 بودکه معمولاً این قسمت از برنامه‌ها توسط حلقه‌ها نمایش داده می‌شوند.

۳. اگر برنامه شامل ساختار if و else باشد که هر کدام ن‌های 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 زمای تکه برنامه‌ها می‌باشد.

به‌طورشهودی، 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 ۱-۱ مرتب‌سازی حبابی**

مسئله: 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 می‌شود. همچنین چون زمان اجرای دستور exchange و شرط if ثابت است لذا زمان کل حلقه داخلی شرط 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 به‌صورت 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 گرفتن C=d و $n_0=2$ (چون n تعداد ورودی‌ها است بنابراین $n_0=1$ یا $n_0=0$ بی‌مفهوم می‌باشد) خواهیم داشت:

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

همچنین می‌توان $\Omega(f(n))$ را به‌صورت 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 $n_0=2$ خواهیم داشت:

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

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

$T(n) \in \theta(n^۲)$

در مثال‌های بعدی برای سادگی محاسبه مقدار ثابت زمان اجراء را برابر یک در نظر خواهیم گرفت.

• **الگوریتم ۲-۱ جستجوی ترتیبی**

مسئله: 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 عنصر مورد جستجو با اولین 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 خانه اول، دوم،... و یا nام آرایه باشد یکسان می‌باشد و مقدار آن برای هر یک از خانه‌ها برابر $\frac{۱}{n}$ می‌باشد. از طرف دیگر اگر عنصر مورد جستجو در خانه اول، دوم،... و یا nام باشد تعداد مقایسه‌ها به‌ترتیب برابر ۱، ۲،... یا n خواهد بود بنابراین زمان متوسط اجراء برابر خواهد بود با:

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

$$T(n)=\frac{1}{n}\sum_{i=1}^{n} i=\frac{1}{n}\,\frac{n(n+1)}{2}=\frac{n+1}{2}$$

بنابراین:

$$T(n)=\frac{n+1}{2}\le\frac{1}{2}(n+n)=n$$

لذا به ازای $C=1$ و $n_0=1$ 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 ۵-۱ یافتن بیشترین مقدار

مسئله: 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 $n-1$، 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:

$$T(n)=\sum_{i=1}^{n-1} d=(n-1)d$$

بنابراین می‌توان نوشت:

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

$\leq d(2n) = 2d \times n$

با در نظر گرفتن $C = 2d$ و $n_0 = 2$ 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 بودند، حال می‌خواهیم الگوریتم‌های نوع دوم که به الگوریتم‌های بازگشتی معروف‌اند را بررسی کنیم.

۲-۳-۱ الگوریتم‌های بازگشتی (recursive algorithm)

معمو لاًدر الگوریتم‌های بازگشتی، مسئله را به دو یا چند زیرمسئله کوچک‌تر تقسیم می‌کنیم. عمل تقسیم مسئله به زیرمسئله‌ها را تا زمانی که اندازه زیرمسئله‌ها به اندازه کافی کوچک شوند ادامه می‌دهیم. بعد از تقسیم به اندازه کافی، برای حل زیرمسئله‌ها از خود الگوریتم استفاده می‌کنیم. حا صل زیرمسئله‌ها را با هم ترکیب می‌کنیم تا راه‌حل مسئله بزرگ‌تر حاصل شود. اعمال ترکیب حاصل زیرمسئله‌ها 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 معمو لاً مقدار ثابتی است)
۲. زمان لازم برای شکستن مسئله به زیرمسئله‌ها
۳. زمان لازم برای ادغام جوابهای زیرمسئله‌ها.

اگر مجموع سه زمان بالا را محاسبه کنیم، زمان اجرای الگوریتم به‌دست خواهد آمد.

۳-۳-۱ محاسبه الگوریتم‌های بازگشتی (recursive algorithm)

همانطورکه 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 اشاره کرد.

در بسیاری از موارد با توجه به خصوصیات الگوریتم‌های بازگشتی ممکن است برای به‌کارگیری 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 دهندلذا 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 به‌کارگیری توابع بازگشتی دو مرحله بالا بترتیب انجام می‌گیرد. در مرحله فراخوانی اعمال زیر انجام می‌شود:

تغیرهای 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 مراحله بازگشت عکس عملیات فوق، به‌صورت زیر انجام می‌شود:

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

۲. آدرس بازگشت از بالای پشته به‌دست می‌آید.

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

٤. کنترل برنامه از آدرس بازگشت بند۲ یابلمه می

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 به‌نام‌های pop و push وجود دارد که بترتیب اولی برای حذف از بالای پشته و دومی برای اضافه کردن به بالای پشته‌می‌روند.

همانطور که اشاره کردیم با به‌کارگیری الگوریتم‌های بازگشتی اعمال فوق بترتیب انجام می‌شود. و همانطور که ملاحظه می‌کنید در بعضی از مواقع امکان استفاده از الگوریتم‌های بازگشتی به‌دلیل 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.

٤-۳-۱ محاسبه مقادیر الگوریتم بازگشتی

همانطور که در بالا اشاره کردیم 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.

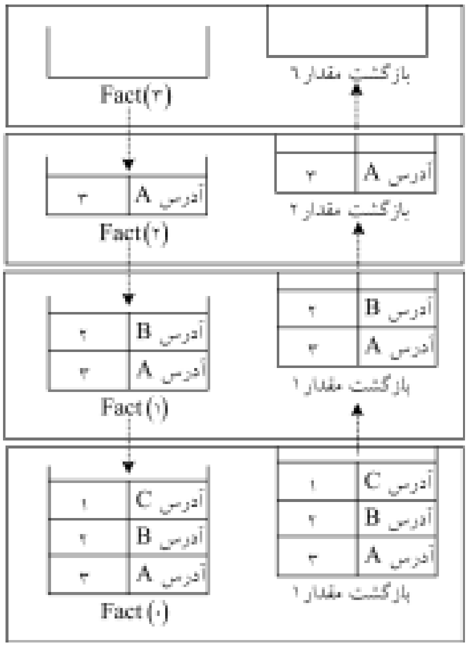
- **روش بازگشتی محاسبه فاکتوریل**

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

$$n! = \begin{cases} 1 & \text{if} \quad n = 0 \\ n \times (n-1)! & \text{if} \quad n > 0 \end{cases}$$

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 ۳-۱ 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 به‌صورت غیربازگشتی نیز ارائه داد. ولی ذاتا به‌صورت بازگشتی است. همچنین ارائه آن به‌صورت بازگشتی به نظر ساده می‌رسد.

به‌صورت زیر می‌توان رابطه بازگشتی سری را نمایش داد:

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

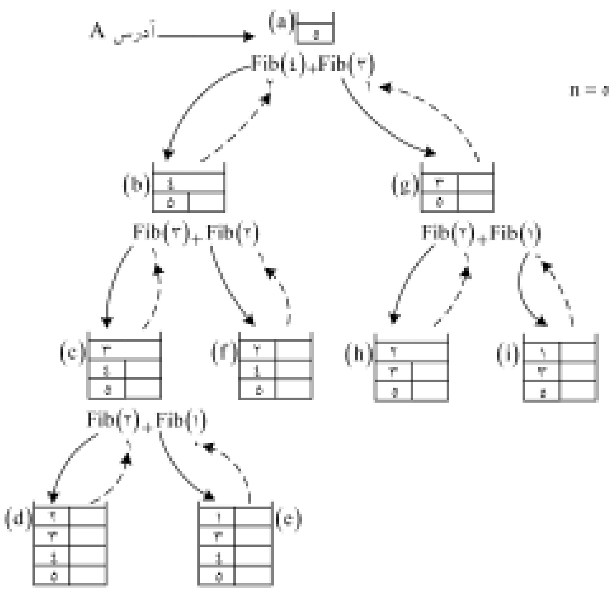
در حالت کلی جملات سری عبارتند از:

۰ ۱ ۱ ۲ ۳ ۵ ۸ ۱۳ ...

حال، تابع بازگشتی زیر را برای تولید جملات سری فیبوناچی به‌کار 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 ١-٤ نمایش داده شده است.



شکل ١-٤ مراحل محاسبه الگوریتم‌های بازگشتی برای سری فیبوناچی

در مرحله اول اجراء، فراخوانی تابع شروع می‌شود که با فلش تو پردر شکل نشان داده شده است، بعد از انجام هر مرحله کامل از فراخوانی مرحله بازگشت شروع می‌شود که با فلش‌های منقطع در شکل مشخص شده است.ترتیب فراخوانی ها با حروف Aتا H در شکل مشخص می‌باشد.

در نهایت تابع مقدار ۳ را به‌عنوان خروجی برمی‌گرداند.

- **وحلسبا زرگجتها نوی**

یکی دیگر از مسائل کلاسیک که حل آن به روش بازگشتی قدرت این روش را نشان می‌دهد. مسئله‌ای بنام برج هانوی است. در این مسئله سه محور ثابت (میله) به نام‌های A، B و C داریم که در ابتدای کار هشت دیسک (Disk) با اندازه‌های متفاوت و از بزرگ به کوچک حول محور A رویهم انباشته شده‌اند (به شکل توجه کنید).

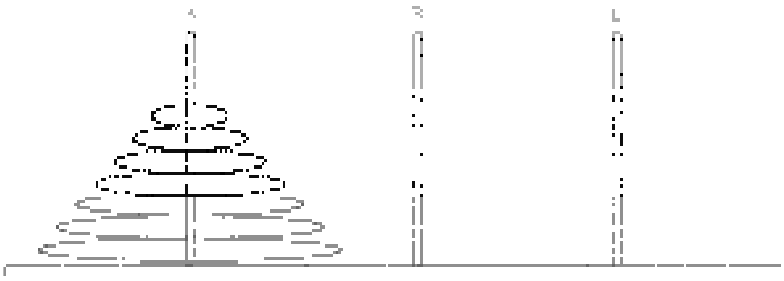
در این مسئله هدف انتقال تمام دیسک‌های روی میله Aبه میله دیگر مثلاً C می‌باشد، به‌طوری‌که قواعد زیر رعایت شود:

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 نظر گرفت. چنانچه n را برابر 2 بگیریم و معما را حل کنیم شناخت بهتری راجع به مسئله پیدا خواهیم کرد. برای حل مسئله در این حالت ابتدا دیسک بالا را از محور A به محور B منتقل می‌کنیم، در مرحله بعد دیسک دیگر از محور A به محور C منتقل می‌گردد و در نهایت دیسک محور B را به محور C منتقل می‌کنیم.



شکل ۵-۱ وضعیت اولیه مسئله برج هانوی

حال مسئله 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 به‌صورت زیر می‌باشد:

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

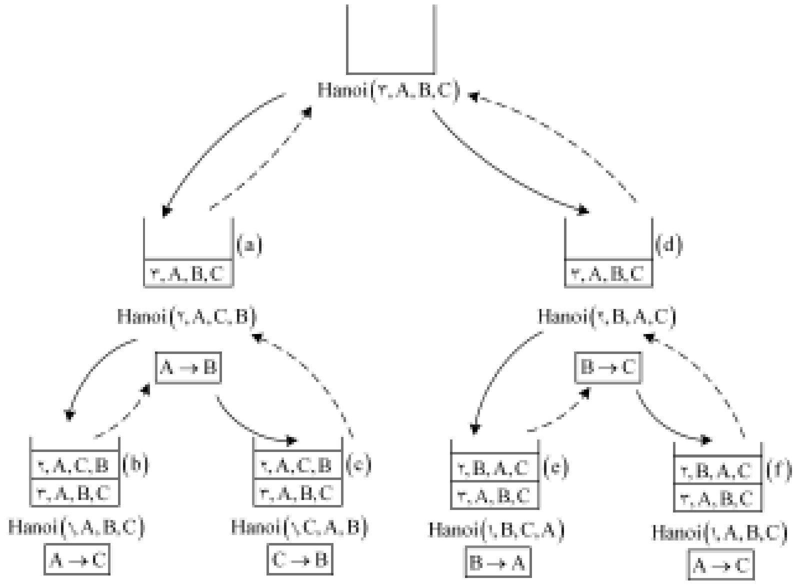
ب) آخرین دیسک محور A به محور C منتقل شود.

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

این روند را می‌توان ادامه داد و مسئله برج هانوی برای n دیسک را حل کرد. در واقع شاهکار روش بازگشتی در این است که حل یک مسئله بزرگ‌تر را منوط به حل مسئله کوچک‌تر می‌کند و مسئله کوچک‌تر را به مسائل کوچک‌تر، تا بالاخره مسئله بسیار کوچک به روش ساده حل شود و از حل آن به‌ترتیب عکس مسائل بزرگ‌تر حل می‌گردد. در زیر الگوریتم Hanoi نمونه‌ای از هنر طراحی الگوریتم‌های بازگشتی را به نمایش می‌گذارد. مقدار n (تعداد دیسک‌ها) و محورهای A، B و C به‌عنوان 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 شکل ۶-۱ نمایش می‌دهیم.



شکل ٦-۱: مراحل اجرای الگوریتم بازگشتی برج هانوی

در شکل بالا فلش‌های توپر مرحله فراخوانی تابع و فلش‌های با خطوط منقطع مرحله بازگشت را نمایش می‌دهند.

۵-۳-۱ محاسبه تابع زمانی الگوریتم‌های بازگشتی

در اینجا قصد داریم طریقه محاسبه تابع زمانی الگوریتم‌های بازگشتی را بحث کنیم. برای روشن شدن مطلب از یک مثال استفاده می‌کنیم.

- **الگوریتم ٦-۱ محاسبه فاکتوریل**

مسئله: الگوریتم بازگشتی برای محاسبه فاکتوریل یک عدد نوشته و زمان 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:

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

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

شکل (۲) روش بازگشتی مسئله محاسبه فاکتوریل یک عدد را نشان می‌دهد. 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:

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

T(n) را به صورت زیر نیز می‌توان نوشت:

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

بنابراین توانستیم تابع زمانی، الگوریتم بازگشتی fact را محاسبه کنیم. 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 مورد حل روابط بازگشتی در درس طراحی الگوریتم ارائه می‌شود.)

۱-٤ 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 می‌کنیم.روابط بازگشتی معمو لاً با توجه به اندازهٔ ورودی به یک معادله یا نامعادله تبدیل می‌شوند.

در این اینجا قصد داریم روش‌هایی را برای حل روابط بازگشتی ارائه دهیم.یکی از این روش‌ها، روش تکرار با جایگذاری می‌باشد. در این روش با توجه به خاصیت روابط بازگشتی به ازای nهای مختلف و جایگذاری آنها در هم، جواب مسئله حاصل می‌شود.

۱-٤-۱ روش تکرار با جای‌گذاری

این روش 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.

مثال ۱۳-۱: رابطه بازگشتی زیر را در نظر بگیرید:

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

رابطه بالا را به روش تکرار با جایگذاری حل کنید.

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

$$\begin{aligned} T(n) &= T(n-۲)+d \\ &= T(n-٤)+۲d \end{aligned}$$

$$= \ldots$$
$$= T(n-2i) + i \times d$$

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:

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

بنابراین $T(n) \in O(n)$.

مثال ۱۴-۱: رابطه بازگشتی زیر را در نظر بگیرید:

$$T(n) = 2T\left(\left\lfloor {n}/{2} \right\rfloor\right) + d \qquad \textbf{(1-1)}$$

که در آن d یک ثابت زمانی می‌باشد. روش تکرار با جای‌گذاری را برای رابطه بازگشتی (۱-۱) به‌صورت زیر به‌کار می‌بریم:

$$T(n) = 2T\left(\left\lfloor {n}/{2} \right\rfloor\right) + d$$
$$= 4T\left(\frac{\left\lfloor {n}/{2} \right\rfloor}{2}\right) + 3d$$
$$\le 4T\left({n}/{4}\right) + 3d$$
$$= \ldots$$
$$\le 2^i T\left({n}/{2^i}\right) + \left(2^i - 1\right)d \qquad \textbf{(1-2)}$$

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

$$\frac{n}{2^i} = 1 \Rightarrow i = \mathrm{Log}_2^n$$

حال i را در رابطه (۱-۲) 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:

$$T(n) \le \begin{cases} C_1 & \text{if} \quad n = 1 \\ 2T\left(n/2\right) + C_2 n & \text{if} \quad n > 1 \end{cases} \qquad \textbf{(1-3)}$$

روش تکرار با جای‌گذاری را برای حل رابطه بالا به‌کار

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

$$T\left(n/2\right) \le 2T\left(n/4\right) + C_2\, n/2 \qquad \textbf{(1-4)}$$

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

$$\begin{aligned} T(n) &\le 4T\left(n/4\right) + 2C_2 n \\ &= \dots \\ &\le 2^i T\left(n/2^i\right) + iC_2 n \qquad \textbf{(1-5)} \end{aligned}$$

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

(با فرض اینکه n توانی از ۲ می‌باشد)

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

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

$$\begin{aligned} T(n) &\le 2^{Log\ n} T(1) + C_2 n\ Log\ n \\ &= C_1 n + C_2 n\ Log\ n \end{aligned}$$

بنابراین $T(n) \in O(n\ Log\ n)$.

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.

۱-۵ ارائه چند مثال

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

مثال ۱-۱: فرض کنید $T_۱(n)$ و $T_۲(n)$ زمان اجرای دو قطعه برنامه $P_۱$ و $P_۲$ باشـد و داریم:

$$T_۱(n) \in O(F(n))$$
$$T_۲(n) \in O(g(n))$$

مقدار $T_۱(n)+T_۲(n)$، زمانی که قطعه برنامه $P_۲$ در راستای قطعه برنامه $P_۱$ اجرا می‌شود را محاسبه نمایید.

حل: می‌دانیم که $T_۱(n) \in O(F(n))$ بنابراین $C_۱$ و $n_۱$ وجود دارد که برای:

$$\forall n \geq n_۱ \qquad T_۱(n) \leq C_۱F(n)$$

و همچنین $T_۲(n) \in O(g(n))$. بنابراین $C_۲$ و $n_۲$ وجود دارد که برای:

$$\forall n \geq n_۱ \qquad T_۲(n) \leq C_۲g(n)$$
$$\Rightarrow T_۱(n)+T_۲(n) \leq C_۱F(n)+C_۲g(n)$$
$$\leq (C_۱+C_۲)\max\{F(n),g(n)\}$$

که در آن با انتخاب $n_۰ = \max\{n_۱,n_۲\}$ و $C = C_۱+C_۲$ خواهیم داشت:

$$T_۱(n)+T_۲(n) \in O(\max\{F(n),g(n)\}$$

مثال ۲-۱: الگوریتمی دارای تابع زمانی زیر می باشد:

$$T(n)=\begin{cases} ۰ & n=۱ \\ ۱ & n=۲ \\ T(n-۲)+۳ & n>۲ \end{cases}$$

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

حل:

$$
\begin{aligned}
T(n) &= T(n-۲)+۳ \\
&= T(n-٤)+٦ \\
&= \dots \\
&= T(n-۲i)+۳i
\end{aligned}
$$

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

$$n-۲i=۲ \;\Rightarrow\; i=\frac{n-۲}{۲}$$

با جایگذاری i در رابطه بالا،عبارت زیر حاصل می‌شود:

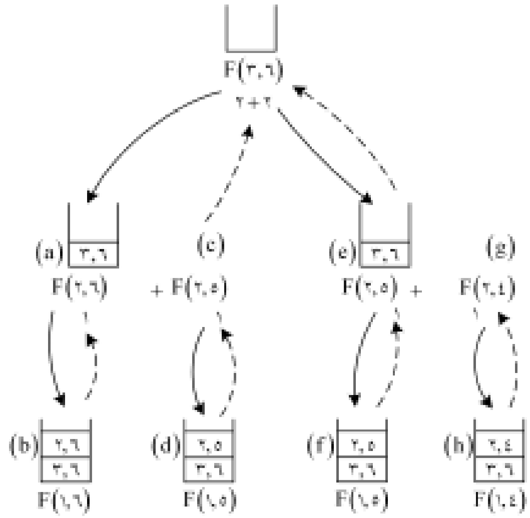
$$
\begin{aligned}
T(n) &= T(۲)+\frac{۳}{۲}n-۳ \\
&= \frac{۳}{۲}n-۲
\end{aligned}
$$

بنابراین $T(n)\in O(n)$.

مثال ۳-۱: 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 ۷-۱ 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 به‌عنوان خروجی نمایش می‌دهد.

مثال-٤: ضرب اعداد طبیعی؛ تعریف ضرب اعداد طبیعی مثال دیگری از تعریف بازگشتی است. ضرب a*b که b*a دو عدد صحیح مثبت هستند ممکن است به‌صورت جمع b بار عدد a با خودش تعریف شود که یک تعریف تکراری است. تعریف بازگشتی آن به‌صورت زیر است:

$$a * b = \begin{cases} a & \text{if} \quad b = ١ \\ a * (b - ١) + a & \text{if} \quad b > ١ \end{cases}$$

در این تعریف برای محاسبه ٦×٣ ابتدا باید ٦×٢ را محاسبه کرد و سپس ٦ را به حاصل ٦×٢ اضافه نمود برای محاسبه ٦×٢ باید ٦×١ را محاسبه کرد و سپس ٦ را به آن اضافه نمود. اما ٦×١ برابر با ٦ است. بنابراین در این الگوریتم بازگشتی حالت توقف، حالت b= =1 است.

```
int   product (int  x  , int   y)
{
     if (y = = 1)
```

```
        return (x),
     rerurn (x + product(x , y – 1)) ;
}
```

مثال ۵-۱: فرض کنید a و b نمایش دو عدد صحیح مثبت باشند و تابع Q به شکل زیر به صورت بازگشتی تعریف شده است:

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

الف) تعداد Q(2,3) و Q(14,3) را پیدا کنید.

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

حل:

الف)

Q(2,3) = 0   2<3 چون

Q (14,3) = 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 می‌یابد از این رو Q (a,b) وقتی a بر b تقسیم می‌شود خارج قسمت را پیدا می‌کند. بنابراین:

Q (5861,7) = 837

مثال ۶-۱: فرض کنید n یک عدد صحیح مثبت باشد. فرض کنید تابع بازگشتی L به‌صورت زیر تعریف شده است:

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

در اینجا $\lfloor K\rfloor$ کف عدد K را نشان می‌دهد و نشان‌دهنده بزرگ‌ترین عدد صحیحی است که بزرگ‌تر از K نباشد.

الف) مقدار L(25) را به‌دست آورید.

ب) این تابع چه عملی را انجام می‌دهد.

حل:

الف)

L(25) = L(12) + 1 = 4
L(12)=L(6) + 1
L(6)=L(3)+1
L(3)=L(1)+1

ب) هر بار که n بر ۲ تقسیم می‌شود تعداد L یک واحد افزایش می‌یابد از این رو L بزرگ‌ترین عدد صحیحی است که: $n \le 2^L$

بنابراین $L=[\log_2 n]$ را به‌دست می‌دهد.

۱-۶ خلاصه فصل

- الگوریتم، مجموعه‌ای از دستورات، دستورالعمل‌ها برای حل یک مسئله می‌باشد.
- معمولاً الگوریتم‌ها را از نظر کارایی با هم مقایسه می‌کنند
- منظور از کارآیی الگوریتم‌ها، مقایسه زمان اجرای الگوریتم‌ها با هم می‌باشد. الگریتمی 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:

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

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

$\forall n \ge n_0 \quad C_1 f(n) \le T(n) \le C_2 f(n)$

$T(n) \in \Omega(f(n)) \Leftrightarrow \exists C, n_0 > 0$ به‌طوری‌که $\forall n \ge n_0 \quad Cf(n) \le T(n)$

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

۱. الگوریتم‌های ترتیبی

۲.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، به‌کارگیری 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 جای‌گذاری‌های 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.

۷-۱ تمرین‌های فصل

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

الف)

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

ب)

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

ج)

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

د)

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

هـ)

```
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)

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

ب)

```
i=1
While (i  <=  n) {
 */θ(١)/* Some Statement requiring time
       i = i * 2 ;
}
```

ج)

```
i= n  ;
While (i>= 1) {
       j= i  ;
      While (j<= n) {
  /* Some Statement requiring  C times */
           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:

١) $7n^2 - 6n + 2 \in \theta(n^2)$

٢) $n^3 + n^2 \operatorname{Log} n \in \theta(n^3)$

٣) $n! + 7n^5 \in O(n^n)$

٤) $7n^2 2^n + 5n^2 \operatorname{Log} n \in \theta(n^2 2^n)$

٥) $2n^{2^n} + 72^n \in \theta(n^{2^n})$

٦) $\sum_{i=0}^{n} i^3 \in \theta(n^4)$

٧) $n^4 + 700n^2 \in \theta(n^4)$

٨) $12n^5 / \operatorname{Log} n + 7n^4 \in O(n^5)$

٩) $2n^3 + 7n^2 \in \Omega(n^2)$

۱۰) $2n^3 + 7n^2 \in \Omega(n^3)$

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

الف)

$$T(n) = n^2 + 1000n$$

ب)

$$T(n) = \begin{cases} n & \text{if} \quad n \text{ is odd} \\ n^3 & \text{otherwise} \end{cases}$$

ج)

$$T(n) = \begin{cases} n & \text{if} \quad n \le 1000 \\ n^3 & \text{if} \quad n > 1000 \end{cases}$$

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

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

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

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

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

۶. درستی یا نادرستی عبارات زیر را بررسی کنید.

۱) $n^5 + 14n^3 \in \theta(n^3)$

۲) $n^5 + 14n^3 \in \theta(n^3)$

۳) $n^5 + 14n^3 \in \Omega(n^3)$

۴) $2^{2^n} \in \theta(2^n)$

۵) $n^6 + 7n^5 + 12 \in \theta(n^7)$

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 ۱-۱ را ثابت کنید.

۱۰. قضایای ۲-۱ و ۳-۱ 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 به‌صورت 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.

۱۴. 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 اعداد صحیح بنویسید. سپس مراحل محاسبه مقدار تابع بازگشتی را با ارائه مثالی بحث نمائید.

۱۶. الگوریتم بازگشتی طراحی کنید تا یک ماتریس n×n را دریافت کرده سپس

دترمینان ماتریس را به‌عنوان خروجی برمی‌گرداند.

۱۷. روابط بازگشتی زیر را حل کنید:

الف)

$$T(n)=\begin{cases} 1 & \text{if} \quad n=4 \\ T\left(\sqrt{n}\right)+c & \text{if} \quad n>4 \end{cases}$$

ب)

$$T(n)=\begin{cases} 1 & \text{if} \quad n=4 \\ 2T\left(n/4\right)+cn & \text{if} \quad n>4 \end{cases}$$

ج)

$$T(n)=\begin{cases} d & \text{if} \quad n=4 \\ 7T\left(n-4\right)+cn & \text{if} \quad n>4 \end{cases}$$

د)

$$T(n)=\begin{cases} d & \text{if} \quad n=2 \\ 5T\left(n/2\right)+n^2 & \text{if} \quad n>2 \end{cases}$$

۱۸. مسئله برج های هانوی را در نظر گرفته، رابطه بازگشتی برای حل آن بنویسید. سپس رابطه حاصل را حل نمائید.

فصل دوم

آرایه‌ها

اهداف

در پایان این فصل شما باید بتوانید:

✓ عملیاتی که روی یک ساختار خطی انجام می‌شود 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 دوبعدی در حافظه کامپیوتر 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 می‌باشد؟

سؤال‌های 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 زبانهایی مثل C و ++C رشته را به‌صورت یک آرایه تعریف می‌کنند؟

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 به‌طورنسبتاً 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 به‌وسیله 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 به‌وسیله 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 (که بعدا بحث خواهد شد)، شامل عملیات زیر است:

عملیات معمول 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 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 به‌کارگرفته شده در کتاب بدین گونه است که ابتدا کلیه مفاهیم مربوط به موضوع و نوع داده مجرد (ADT) مربوطه 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 می را بحث خواهیم نمود.

۱-۲ مفهوم نوع داده مجرد (Abstract Data Type)

با توجه به اینکه نوع داده مجرد در تمامی فصول این کتاب به‌کار برده شده، بنابراین در اینجا یک تعریف و بحث کلی در مورد نوع داده مجرد (ADT) انجام می‌دهیم.

غالباً در هر زبان برای نوشتن برنامه، زبان مربوطه انواع داده‌هایی را در اختیار برنامه‌نویس قرار می‌دهد تا بتوان برنامه نوشت. غالباً داده‌هایی از نوع صحیح، کاراکتری، منطقی و غیره انواع داده‌های اصلی، اکثر زبان‌ها می‌باشند. علاوه از اینها غالباً مکانیزم‌هایی از قبیل آرایه، ساختار، کلاس و غیره نیز در زبان‌های برنامه‌نویسی وجود دارد. به‌عنوان نمونه آرایه‌ها، مجموعه‌ای از عناصر از یک نوع داده می‌باشند و به‌صورت ضمنی تعریف می‌شوند.

تعریف نوع داده: نوع داده مجموعه‌ای از انواع داده مقصد (object) و عملگرهایی است که بر روی این نوع داده‌ها عمل می‌کنند.

همانطور که از تعریف برمی‌آید هر نوع داده از دو بخش تشکیل می‌شود. نخست مقصد نوع داده و سپس عملگرها روی نوع داده می‌باشد.

برای مثال نوع داده صحیح شامل داده‌های مقصود بین (Max int.....۱+,۰,۱−,...) بوده و عملگرهای حسابی +,۰,*,/ و غیره روی آنها عمل می‌کنند.

علاوه از شناخت عملگرهای نوع داده، نحوه نمایش نوع داده نیز می‌تواند مهم باشد. به‌طور مثال دانستن تعداد بایت‌های موردنیاز برای یک عدد صحیح یا اعشاری می‌تواند بسیار مهم باشد. حال نوع داده مجرد را به‌صورت زیر تعریف می‌کنیم

تعریف نوع داده مجرد (ADT): نوع داده مجرد، یک نوع داده است که در ساختار آن نیاز به دانستن مشخصات داده‌های مقصود و مشخصات اعمالی (عملگرها) که بر روی آنها انجام می‌شود، می‌باشد و در آن قسمت نمایش مقصودها و پیاده‌سازی عملگر از یکدیگر متمایز می‌باشند.

غالباً نوع داده مجرد مستقل از نحوه پیاده‌سازی می‌باشد. در این کتاب ADT را با مفهوم بالا به‌کار برده‌ایم. نحوه پیاده‌سازی و نمایش مستقل از عملگرها و داده‌های مقصود ارائه داده‌ایم.

۲-۲ آرایه‌ها

آرایه، لیستی از n عنصر یا مجموعه‌ای متناهی، از عناصر داده‌ای هم نوع می‌باشد (یعنی عناصر داده‌ای از یک نوع هستند) به‌طوری 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 به‌عنوان 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.

۲-۳ آرایه به‌عنوان داده انتزاعی (Abstract Data Type)

منظور از نوع داده انتزاعی یک مدل ریاضی است که متشکل از مجموعه عناصر و عملیاتی بر روی آن مدل تعریف شده‌اند، می‌باشد. توجه کنید که، نوع داده انتزاعی مستقل از خواص پیاده‌سازی می

حال، آرایه را بعنوان یک نوع داده مجرد در نظر می‌گیریم. بنابراین داده‌های مقصود و عملگرهای آن را بصورت زیر ارائه می‌دهیم:

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 به‌طوریکه بتوان عمل ذخیره و بازیابی را انجام داد.

٤-٢ آرایه‌های یک بعدی

آرایه یک بعدی برای ذخیره مجموعه‌ای از عناصر همنوع به‌کار 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 می‌توان عناصری را ذخیره کرد و تنها عملی که می‌توان 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 اندیسی مثل i را در نظر گرفته و عنصرi 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 می‌شود و بزرگ‌ترین مقدار اندیس آرایه را کران بالای آرایه نام دارد و با upper نمایش می‌دهند. در حالت کلی تعداد عناصر آرایه یک بعدی برابر است با:

Upper − lower + 1

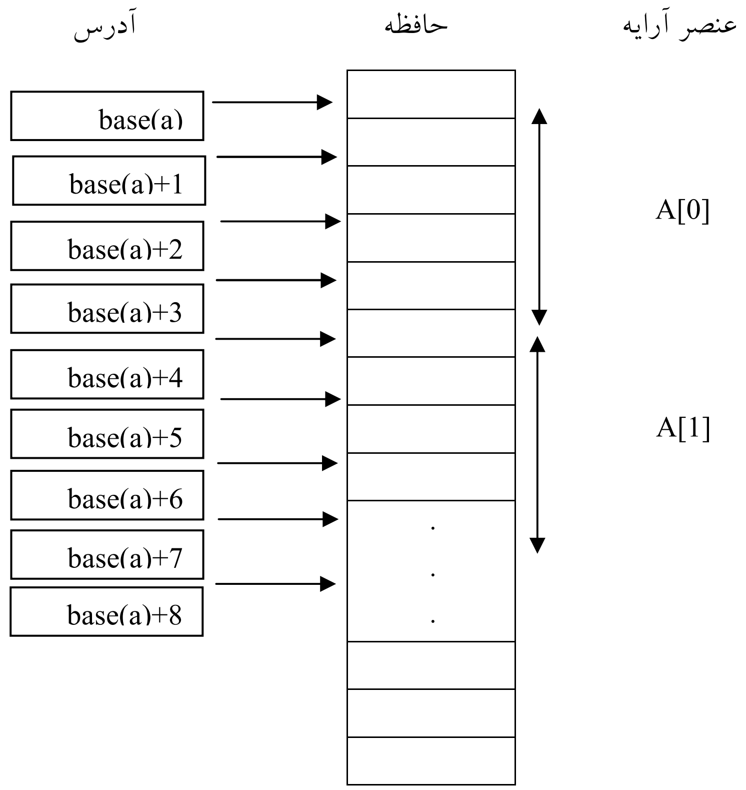
۵-۲ نمایش آرایه یک بعدی

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 محل مـی‌تـوان یک مقدار اعشاری را ذخیره کرد. آدرس اولین محل، آدرس پایه نام 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 آرایه a[0] با شروع از آدرس base(a) در چهار بایت از حافظه ذخیره می‌شود و عنـصر a[1] بـا شـروع از آدرس base(a)+4 در چهاربایـت بعدی ذخیره می‌شود. شکل ۱-۲ نحوه نمایش آرایه را نمایش می‌دهد.



شکل ۱-۲ نحوه نمایش آرایه

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

$$\boxed{Loc(i)=base(a)+i*size}$$

مثال ۱-۲: فرض کنید آرایه A به‌صورت float A[10] تعریف شود و base(a) برابر بـا ۳۰۰۰ باشد (یعنی عناصر این آرایه از خانه ۳۰۰۰ به بعد درحافظه قرار می‌گیرنـد) و بـا فرض هر عدد اعشاری چهاربایت فضا از حافظه را اشغال می‌کند آدرس خانـه A[7] را محاسبه نمائید.

حل:

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

۲-٦ نمونه‌ای از کاربردهای آرایه یک بعدی برای جستجو

یکی از اعمالی که در آرایه‌های یک بعدی انجام می‌گیرد، جـستجوی مقـداری در آرایـه می‌باشد. جستجو در آرایه می‌تواند به یکی از دو صورت:

- ترتیبی(خطی)
- دودویی (Binary Search)

انجام شود.

۲-٦-۱ جستجوی ترتیبی در آرایه

الگوریتم جستجو به‌صورت ترتیبی، ابتدا مقداری را از کاربر دریافت می‌کند و سپس بـه جستجو در آرایه برای پیداکردن مقدار موردنظر می‌پردازد. عنصر مورد جستجو نخـست با اولین عنصر، دومین عنصر و... مقایسه می‌شود که در نهایت در صورت موفـق بـودن عمل جستجو اندیس خانه مورد نظر ارسال می‌گردد در غیـر ایـن‌صـورت مقـدار ۱- را به‌عنوان خروجی برمی‌گرداند.

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 بنابراین 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:

$$P_1+P_2+...+P_n+q=1$$

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

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

(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 $q\approx ۰$ و هر $P_i=\frac{1}{n}$ بنابراین:

$$T(n)=1\times\frac{1}{n}+2\times\frac{1}{n}+...+n\times\frac{1}{n}+(n+1)\times ۰=(1+2+...+n)\times\frac{1}{n}$$
$$=\frac{n(n+1)}{2}\times\frac{1}{n}=\frac{n+1}{2}$$

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

۲-۶-۲ جستجوی دودوئی در آرایه

فرض کنید a یک آرایه است که در آن داده‌های عددی به‌صورت مرتب ذخیره شده‌اند. آنگاه الگوریتم جستجوی بسیار کارآیی به‌نام 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.

نکته: 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 نمی‌کنید) 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 به‌وسیله تعداد مقایسه‌های موردنیاز برای تعیین مکان item در آرایه مشخص می‌شود. از طرفی می‌دانیم که، آرایه دارای n عنصر

می‌باشد. با توجه به الگوریتم ملاحظه می‌شود هر مقایسه در الگوریتم باعث می‌شود که، اندازه ورودی نصف شود ازاین‌رو حداکثر T(n) مقایسه لازم است تا مکان عنصر item پیدا شود، بنابراین تعداد مقایسه‌ها برابر خواهد بود با:

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

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

نکته: زمان اجرای در بدترین حالت برای جستجوی ترتیبی O(n) می‌باشد و زمان اجرای در بدترین حالت برای جستجوی دودوئی O(log n) می‌باشد.

۲-۷ آرایه‌های دوبعدی

آرایه‌های خطی که در بخش قبل مورد بررسی قرار گرفتند آرایه‌های یک بعدی بودند. و به همین دلیل به هر عنصر این نوع آرایه‌ها می‌توان به کمک تنها یک اندیس دسترسی پیدا کرد. ولی در بسیاری از مسائل آرایه‌های یک بعدی کارائی لازم را ندارند مثلاً برای پیاده‌سازی ماتریس‌ها به‌کارگیری آرایه‌های یک بعدی بسیار سخت می‌باشد. لذا برای راحتی کار استفاده از آرایه‌های دو بعدی توصیه می‌شود. در آرایه‌های دو بعدی دو اندیس برای دسترسی به عناصر آرایه تعریف می‌شود که اصطلاحاً یکی از اندیس‌ها در سطر و دیگری در ستون حرکت می‌کند. در زیر طریقه تعریف آرایه‌های دو بعدی در زبان C و ++C را ارائه می‌دهد:

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

آرایه‌های دوبعدی را در ریاضیات، برای پیاده‌سازی ماتریس‌ها و در کاربردهای تجاری و بازرگانی برای پیاده‌سازی جدول‌ها به کار می‌برند. بنابراین به آرایه‌های دوبعدی گاهی اوقات آرایه‌های ماتریسی نیز می‌گویند.

یک روش استاندارد برای نمایش آرایه دو بعدی m×n وجود دارد که در آن عناصر A تشکیل یک آرایه مستطیلی با m سطر و n ستون را می‌دهد که در آن مقدار عنصر A[j][k] در سطر j ام و ستون k ام قرار دارد. شکل (۲-۲) نمایشی از یک ماتریس 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

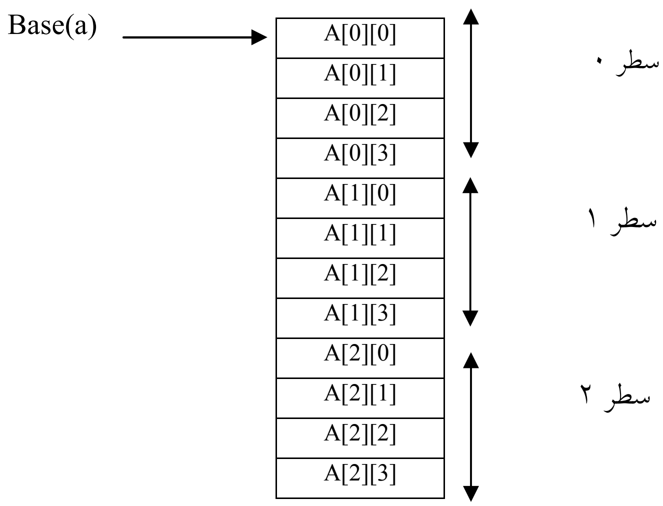
۱-۷-۲ نحوه ذخیره‌سازی آرایه‌های دوبعدی

آرایه‌های دوبعدی می‌توانند به‌صورت 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 به‌صورت سطری ذخیره می‌شوند.

آرایه زیر را در نظر بگیرید:

int A[3] [4]

چون عناصر آرایه درC و C++ به‌صورت سطری ذخیره می‌شوند نمایش این آرایه در حافظه مانند شکل(۳-۲) خواهد بود.



شکل ۳-۲: نمایش سطری ماتریس

فرض کنید آرایه A با m سطر و n ستون به‌صورت 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 در نظر بگیریم و فرض کنیم ماتریس به‌صورت سطری در حافظه ذخیره می‌شود آنگاه محل عنصر A[i][j] در حافظه از رابطه زیر به‌دست می‌آید:

محل A[i][j] =base(A)+(i*n+j) * size

و اگر ماتریس به‌صورت ستونی در حافظه ذخیره شود آنگاه محل عنصر A[i][j] در حافظه از رابطه زیر به‌دست می‌آید:

محل A[i][j] =base(A)+(j*m+i) * size

۲-۸ ماتریس‌های اسپارس (Sparse)

بعضی از ماتریس‌ها وجود دارند که تعداد زیادی از عناصر آنها صفر است. به‌عنوان مثال، ممکن است در مسئله‌ای به ماتریس با ابعاد 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 می‌شودمعمو لاً روی عناصر غیرصفر انجام می‌گیرد.

بنابراین به‌نظر 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.

به‌عنوان مثال یک ماتریس اسپارس و نحوه نمایش آن، در شکل (۲-۴) نمایش داده شده است.

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

(الف) ماتریس اسپارس

تعداد مقادیر غیر صفر

تعداد ستون

تعداد سطرها

3	5	3
0	1	3
1	0	6
2	3	2

(ب) نمایش ماتریس اسپارس

4*3

شکل ٤-۲ ماتریس اسپارس و شکل بهبود یافته آن

اکنون مقدار فضای اشغالی توسط نمایش معمولی و نمایش اسپارس ماتریس شکل(٤-۲) 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 این‌صورت 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.

۲-۸-۱ 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 به‌صورت 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 (۴-۲) را به‌صورت زیر ارائه می‌دهیم:

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

شکل ۵-۲: نمایش ماتریس ترانهاده ماتریس اسپارس

حال تابع ماتریس ترانهاده ماتریس اسپارس را بصورت زیر پیاده‌سازی می‌کنیم
ساختار داده مسئله بصورت :

```
Typedef    struct  {
            int   col , row ;
            elementtype  value ;
            } sparse;
Sparse   mat[Max-Size];
```

می‌باشد. بر اساس این ساختار داده، تابع ترانهاده را ارائه می‌دهیم:

```
void    transpose_sparse ( sparse  mat[ ] , sparse  Tmat[ ] )
{
        int  n , i , j , current_Tm ;
        n = mat[0].value ;
        Tmat[0].row = mat[0].col ;
```

```
        Tmat[0].col = mat[0].row ;
        Tmat[0].value = n ;
        If  ( n > 0 )
        {
              current_Tm = 1 ;
              for ( i = 0 ; i < mat[0].col  ;  i++ )
                   for( j = 1 ; j <= n ; j ++  )
                      if  (  mat[j].col == i  )
                      {
                           Tmat[current_Tm].row = mat[j].col ;
                           Tmat[current_Tm].col  = mat[j].row ;
                           Tmat[current_Tm].value= mat[j].value ;
                       }
          }
}
```

در این تابع زمان اجرا مربوط به، دو حلقه for می‌باشد که در آن حلقه اول به تعداد col بار و حلقه دوم به تعداد عناصر ماتریس یعنی value بار تکرار می‌شوند. بنابراین پیچیدگی زمانی تابع بالا برابر خواهد بود با :

$T(n) \in O(col \times value)$

۹-۲ رشته (String)

از نظر تاریخی، 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 می‌باشدمعمو لاً 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 مسئله تطبیق الگو را به‌طور کامل بررسی کنیم. و علاوه بر این، سه الگوریتم مختلف برای تطبیق الگو ارائه خواهیم داد.

تعریف: رشته در واقع آرایه‌ای از کاراکترهاست. ولی به نوعی آن را 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 رشته 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 کپی کردن، الحاق کردن، جستجوی یک رشته داخل رشته دیگر وجود دارد.

۱-۹-۲ الگوریتم های تطابق الگو (Pattern Matching)

منظور از تطبیق الگو همانطور که در بالا اشاره کردیم، مسئله‌ای است که تعیین می‌کند یک الگوی رشته داده شده P در متن رشته‌ای T وجود دارد یا خیر. در الگوریتهای ارائه شده، فرض خواهیم کرد که همواره طول P کوچک‌تر از طول T باشد.

- **الگوریتم اول تطبیق الگو**

الگوریتم اول تطبیق الگو، الگوریتم ساده و غیرکارآمدی است که در آن الگوی داده شده P با همه زیررشته‌های T مقایسه می‌شود. عمل مقایسه با حرکت از چپ به راست 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 کاراکتر T می‌باشد. ابتدا P را کاراکتر به کاراکتر با اولین زیررشته یعنی W_1 مقایسه می‌کنیم. اگر تمام کاراکترها مساوی باشند، در اینصورت $P=W_1$ و در نتیجه P در T ظاهر شده است. از سوی دیگر اگر به این نتیجه برسیم که یک کاراکتر P دارای متناظر در W_1 نیست در این‌صورت $P \neq W_1$ خواهد بود و به سراغ زیررشته بعدی یعنی W_2 می‌رویم اعمال بالا را ادامه می‌دهیم تا زمانیکه، عمل مقایسه متوقف شود.

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

به‌عنوان تمرین تابع الگوریتم بالا را بنویسید.

- **الگوریتم دوم تطبیق الگو**

الگوریتم دوم تطبیق الگو، الگوریتمی است که در این الگوریتم وقتی طول الگو (P) بزرگ‌تر از تعداد کاراکترهای باقیمانده در رشته T می‌باشد از آن خارج می‌شود، 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 به اندازه تعداد کاراکترهای الگو(P) انتخاب می‌شود و کاراکتر آخر انتخاب شده در T (Endmatch) با کاراکتر آخر الگو مقایسه می‌شود و در صورت مساوی بودن، سایر کاراکترها بررسی می‌شوند و در غیراینصورت اندیس Start یک مکان به جلو حرکت می‌کند و از آن مکان به اندازه کاراکترهای الگو انتخاب می‌شود. مراحل بالا را تا 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. نخست، [T[endmatch و [P[lastp با هم مقایسه می‌شوند اگر برابر بودند الگوریتم از i و j برای حرکت در داخل دو رشته تا زمانی که یک نابرابری اتفاق بیافتد یا تا زمانی که تمام P با آن‌ها برابر باشد ادامه می‌یابد.

P:

A a B
J Lastp

a b a b b a a b a a

عدم تطبیق

Start endmatch

$T[endmatch] \neq P[lastp]$ بنابراین Start یک خانه به جلو حرکت می‌کند.

لذا 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 a |
                      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 a b b a a b a A |
                      عدم تطبیق
        Start  endmatch
```

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

حال تابع الگوریتم فوق بنام nfind بصورت زیر 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  lastt = strlen (T) -1 ;
     int  lastp = strlen (P) -1 ;
     int endmatch = lastp ;
     for (i=0 ; endmatch < = lastt ; endmatch++ , 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 ;
}
```

- **الگوریتم سوم تطابق الگو**

به‌صورت 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 مسئله می‌باشد.

در بدترین حالت باید حداقل یکبار تمام کاراکترهای T و P را تست نمود. می‌خواهیم رشته‌ای را برای یافتن یک الگو جستجو کنیم، بدون اینکه به عقب برگردیم. بنابراین اگر یک عدم تطابق رخ دهد، می‌خواهیم که از اطلاعات خود در مورد کاراکترها و موقعیت آن در الگو یعنی محلی که عدم تطبیق روی داده است، برای تعیین جایی که باید جستجو را ادامه دهیم، استفاده کنیم.

حال فرض کنید:

$P = aaba$

و $T = T_1T_2...T_m$ که در آن T_i کاراکتر iام T را نمایش می‌دهد و فرض کنید کاراکتر اول T با دو کاراکتر اول P انطباق دارد. یعنی فرض کنید $T = aa...$. آنگاه T دارای یکی از سه صورت زیر است:

$1) T = aab... \qquad 2) T = aaa... \qquad 3) T = aax$

که در آن x می‌تواند هر کاراکتر دلخواهی به غیر از a یا b باشد. فرض کنید T_3 را خواندیم و دریافتیم که $T_3 = b$. در این صورت، به دنبال آن T_4 را می‌خوانیم تا ببینیم آیا $T_4 = a$ است یا خیر. P با W_1 تطابق خواهد داشت. از طرف دیگر، فرض کنید $T_3 = a$. در این صورت می‌دانیم که $P \neq W_1$ همچنین می‌دانیم که $W_2 = aa...$، یعنی دو کاراکتر اول زیررشته W_2 با دو کاراکتر اول P تطابق دارد. از این رو بعد از T_4 را می‌خوانیم تا ببینیم آیا $T_4 = b$ است؟ بالاخره فرض کنید $T_3 = x$. در این

صورت می‌دانیم $P \neq W_1$، اما همچنین می‌دانیم که $P \neq W_2$ و $P \neq W_3$، چون x در P ظاهر نمی‌شود، از این رو بعد از آن، T_4 را می‌خوانیم تا ببینیم آیا $P_4 = a$ یا خیر؟ یعنی باید ببینیم که آیا کاراکتر اول W_4 با کاراکتر اول P تطابق دارد یا خیر؟

دو نکته قابل توجه و مهم در روش بالا وجود دارد: نخست، هنگام خواندن T_3، تنها لازم است T_3 با آن تعداد از کاراکترها که در P وجود دارد مقایسه شود. اگر هیچ یکی از این کاراکترها مطابقت نداشت، در این صورت ما در حالت آخری هستیم که کاراکتری مانند x در P وجود ندارد. دوم، پس از خواندن و بررسی T_3 و T_4 را می‌خوانیم‌لازم نیست مجدداً به متن T برگردیم.

۱۰-۲ 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 ۱-۲: یک ماتریس n×n را درنظر بگیرید که فقط عناصر قطر اصلی آن مخالف صفر می باشد (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).

$$A=\begin{pmatrix} a_{oo} & & 0 \\ & \ddots & \\ 0 & & a_{(n-1)(n-1)} \end{pmatrix} \qquad AS=\begin{pmatrix} n & n & n \\ 0 & 0 & a_{00} \\ 1 & 1 & a_{11} \\ \vdots & \vdots & \vdots \\ n-1 & n-1 & a_{(n-1)(n-1)} \end{pmatrix}$$

الف: ماتریس قطری **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 به‌صورت زیر محاسبه

می‌شود:

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

اما نمایش اسپارس این ماتریس دارای n+1 سطر (n تعداد عناصر غیرصفر) و ۳ ستون می‌باشد و در نتیجه فضای آن به‌صورت زیر محاسبه می‌شود:

As فضای حافظه $= 3\times(n+1)\times 4=12(n+1)$ بایت

با توجه به روابط بالا، 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 به‌صورت معمولی ذخیره شود. اما اگر $n\geq 4$ باشد، ماتریس‌های قطری اسپارس هستند و باید به‌صورت اسپارس نمایش داده شوند.

مثال ۲-۲: 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.

$$A=\begin{pmatrix} a_{11} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix}$$

حل: فرض کنید بخواهیم آرایه مثلثی A را که در زیر ارائه شده، در حافظه کامپیوتر ذخیره کنیم. واضح است که ذخیره درآیه‌های بالای قطر اصلی A کاری بیهوده است چون می‌دانیم تمام این عناصر صفر هستند از این رو تنها درایه‌های دیگر A را 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} را احتیاج خواهیم داشت، از این رو فرمولی را به‌دست 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

می‌دهد. اکنون تعداد:

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

عنصر در سطر بالای a_{ij} وجود دارد و عنصر در سطر i وجود دارد که حداکثر تا a_{ij} و خود a_{ij} را شامل است. بنابراین:

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

مثال ۳-۲: فرض کنید A یک آرایه سه قطری n مربعی به‌صورتی باشد که در شکل نشان داده شده است.

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

توجه کنید که روی قطر A، n عنصر و در بالا و پایین قطر n-۱ عنصر می‌باشد. از این رو در ماتریس A حداکثر 3n-2 عنصر غیر صفر وجود دارد. فرض کنید بخواهیم A را در یک آرایه خطی B ذخیره کنیم. فرمولی پیدا کنید که ماتریس را بر حسب i و j به صورتی نمایش دهد که:

B[L]=A[i][j]

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

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

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

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

مثال ۴-۲: فرض کنید یک آرایه ۲۰۰ عنصری مرتب شده باشد. زمان اجرای بدترین $(F_{(n)})$ برای پیداکردن عنصر معلوم x در آرایه A با استفاده از جستجوی دودویی چقدر است:

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

۲-۱۱ تمرین‌های فصل

۱. الگوریتمی ارائه دهید که، دو عدد صحیح حداکثر ۳۰۰ رقمی را باهم جمع کند. یک روش این است که عدد صحیح ۱۹۸، ۱۷۹، ۶۷۲،۵۳۴ 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. الگوریتمی ارائه دهید که، عدد صحیح n را خوانده و فاکتوریل آن را محاسبه کند. n می‌تواند 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 باشند. نشان دهید که چگونه یک آرایه n در n+1 مثل c می‌تواند 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 خواهد بود. حداکثر تعداد عناصر غیر صفر در چنین آرایه چیست؟ چگونه این عناصر می‌توانند در حافظه به طور ترتیبی ذخیره شوند؟

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

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

۸. فرض کنید لیستی از عناصر در آرایه‌ای به طول Max size قرار گرفته‌اند توسط تابعی لیست را مرتب کنید تعداد عناصر لیست را n در نظر بگیرید.

۹. لیست حاصل از مسئله ۸ را در نظر گرفته سپس توسط تابعی به نام Insert عنصر جدیدی را در لیست طوری اضافه کنید که ترتیب عناصر به هم نخورد.

۱۰. الگوریتم ارائه شده در مسئله ۹ برای تابع Insert را تحلیل زمانی نمایید.

۱۱. لیست حاصل از مسئله ۸ را در نظر گرفته سپس با دریافت عنصر جدید از ورودی توسط تابعی به نام delete عنصر موردنظر از لیست حذف نمایید.

۱۲. الگوریتم ارائه شده در مسئله ۱۱ را در نظر گرفته، آن را تحلیل زمانی کنید.

۱۳. دو چند جمله P(x) از درجه n و Q(x) از درجه m را در نظر گرفته تابعی به نام

Add بنویسید که مجموع دو چند جمله‌ای را محاسبه نمایید. پیچیدگی زمانی تابع Add را محاسبه نمایید.

۱٤. دو ماتریس اسپارس B, A را در نظر بگیرید. تابعی برای محاسبه حاصلضرب دو ماتریس ارائه دهید و زمان آن را محاسبه نمایید.

۱۵. دو لیست مرتب L(n) و L(m) را در نظر بگیرید. تابعی برای ادغام دو لیست مرتب ارائه دهید طوری که لیست حاصل مرتب باشد.

۱٦. دو ماتریس اسپارس B, A را در نظر بگیرید. تابعی برای محاسبه مجموع دو ماتریس ارائه دهید و زمان آن را محاسبه نمایید.

۱۷. دو چند جمله P(x) از درجه n و Q(x) از درجه m را در نظر گرفته تابعی به نام Mult بنویسید که حاصلضرب دو چند جمله‌ای را محاسبه نمایید. پیچیدگی زمانی تابع را محاسبه نمایید.

۱۸. الگوریتم سوم تطابق الگو را با ارائه یک مثال و بیان حالت شکست آن و تابع مربوطه بنویسید.

۲-۱۲ پروژه‌های برنامه‌نویسی

۱. جدول جادوئی یا مربع جادوئی، یک ماتریس $n \times n$ از اعداد صحیح ۱ تا n^2 می‌باشد به گونه‌ای که مجموع عناصر هر سطر، ستون و دو قطر اصلی باید یکسان باشند. در این مربع، عدد یک را در وسط سطر اول قرار دهید سپس به بالا و به سمت چپ رفته و عدد بعدی را به ترتیب نزولی در خانه خالی قرار دهید. اگر با این اعمال از مربع خارج شدید. مربع را حلقوی در نظر بگیرید و خانه بعد را پیدا و همین فرایند را ادامه دهید اگر خانه پیدا شده پر باشد عدد مورد نظر را در زیر عدد قبلی قرار دهید.

برنامه‌ای بنویسید که مربع بالا را بسازد.

۲. شطرنج، سرگرمیهای بسیار جالبی که کاملاً مجزا از خود بازی است، فراهم می‌سازد. خلی از این سرگرمی‌ها براساس حرکت عجیب «L مانند» اسب است. یک مثال متداول، مسأله گردش یا تور اسب می‌باشد که توجه ریاضیدانان و افراد علاقه‌مند به معما را از اوایل قرن ۱۸ به خود جلب نموده است. به‌طور خلاصه،

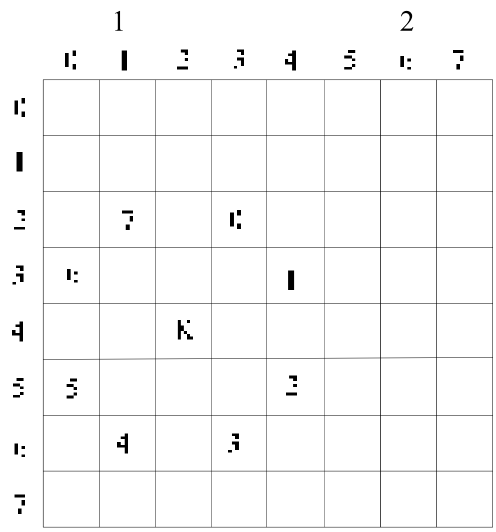
مسأله بدین صورت می‌باشد که اسب را از هر متر مربع صفحه شطرنج حرکت می‌دهند به نحوی که به‌طور موفقیت‌آمیزی تمامی ٦٤ مربع شطرنج را طی نماید (به شرطی که هر مربع را فقط و فقط یکبار طی نماید.) مناسب است که راه‌حلی با قرار دادن اعداد ٦٣,...,١,٠ در مربع‌های شطرنج ارائه شود. این شماره‌ها نشان‌دهنده ترتیبی هستند که مربع‌ها طی شده‌اند. توجه داشته باشید که نیازی نیست که اسب با یک حرکت اضافی به موقعیت شروع خود برسد. اگر این اتفاق بیافتد گردش اسب «تازه داوطلب» نامیده می‌شود. یکی از روش‌های زیرکانه برای حل مسأله گردش اسب توسط وارنسدورف در سال ۱۸۲۳ ارائه شده است. پیشنهاد او بدین صورت است که اسب همیشه باید به یکی از مربع‌هایی که دارای خروجی‌های کمتری نسبت به مربعی که قبلاً طی شده باشد، حرکت نماید.

هدف این تمرین نوشتن برنامه‌ای است که قاعده وارنسدورف را پیاده و اجرا نماید و به‌طور یقین دنبال کردن این روش بسیار ساده خواهد بود. هر چند توصیه می‌شود که شما ابتدا سعی نمایید یک راه‌حل خاص برای مسأله با دست پیدا نموده سپس به بقیه مطالب بپردازید.

مهمترین نکته‌ای که در حل چنین مسأله‌ای مهم به نظر می‌رسد، این می‌باشد که داده‌ها در کامپیوتر چگونه ذخیره می‌شوند. شاید طبیعی‌ترین روش برای نمایش صفحه شطرنج، آرایه ٨×٨ به نام BOARD است که در شکل آورده شده است.

همچنین هشت حرکت ممکن است در مربع (٤,٢) در شکل نشان داده شده است. به‌طور کلی اسب در موقعیت (i,j) ممکن است به یکی از مربع‌های زیر حرکت نماید: (i−٢,j+١)، (i−١,j+٢)، (i+١,j+٢)، (i+٢,j+١)، (i+١,j−٢)، (i−١,j−٢) و (i−٢,j−١). هر چند توجه داشته باشید که اگر (i,j) در نزدیکی یکی از لبه‌های صفحه قرار داشته باشد، بعضی از این هشت حرکت موجب می‌شود که اسب از صفحه خارج شود و البته این امر مجاز نمی‌باشد. هشت حرکت ممکن اسب را می‌توان به‌طور مناسبی توسط دو آرایه به‌نام‌های ktmove 1 و ktmove 2 به شرح زیر ارائه کرد.

ktmove 2	ktmove 2
1	-2
2	-1



شکل ۱-۱ حرکات مجاز برای اسب

ktmove 2	ktmove 2
2	1
2	-1
1	-2
-1	-2
-2	-1

سپس اسب در موقعیت (i, j) ممکن است به محل $(i+ktmove\ [k], j+ktmove\ 2[k])$ حرکت کند، به نحوی که k مقداری بین 0 و 7 خواهد داشت. در زیر توصیف و شرح الگوریتمی برای حل مسأله گردش اسب با استفاده از قاعده وارنسدورف آورده شده است. نمایش داده‌ها مانند بخش قبلی فرض شده است:

۱-۲ مقداردهی اولیه صفحه شطرنج. به ازای $0 \leq i$، board[iIj] را برابر صفر قرار دهید.

۲-۲ موقعیت شروع. (i و j) را خوانده و چاپ نمایید. سپس board[iIj] را برابر 1 قرار دهید.

۳-۲ حلقه. برای $1 \leq m \leq 63$ مراحل ٤-۲ تا ۷-۲ را انجام دهید.

٤-۲ مجموعه خانه‌های ممکن بعدی را تشکیل دهید. هر یک از هشت خانه مربوط

به حرکت اسب از محل (i, j) را تست نموده و لیست امکان حرکت خانه‌های بعدی $(nexti[l], nextj[l])$ را تشکیل دهید. اجازه دهید npos تعداد حالات حرکت باشد. (این بدین مفهوم است که پس از اجرای این مرحله، به ازای مقدار خاصی از k بین 0 تا 7 خواهیم داشت: $(nexti[l]=i+ktmove\ 1[k]$ و $next\ j[l]=j+ktmove\ 2[k])$. بعضی از خانه‌های $(i=ktmove\ 1[k], j+ktmove\ 2[k])$ برای حرکت بعدی غیرممکن هستند. این بدین خاطر است که یا آنها خارج از صفحه شطرنج قرار می‌گیرند و یا اینکه آنها قبلاً توسط مهره اسب اشغال شده‌اند یا به‌عبارت دیگر آنها حاوی مقدار مخالف صفر هستند. در هر حالت خواهیم داشت $O \le npos \le 8$.

۲-۵ تست حالات خاص. چنانچه $npos = 0$ باشد، گردش اسب نابهنگام به اتمام رسیده است. این شکست را گزارش نموده و به مرحله ۸-۲ بروید. اگر npos=1 باشد تنها یک امکان برای حرکت بعدی وجود دارد، min را برابر 1 قرار دهید و به مرحله ۷-۲ بروید.

۲-٦ خانه بعدی را با حداقل تعداد خروجی پیدا نمایید. به ازای $1 \le l \le npos$، $exits[l]$ را برابر با تعداد خروجی‌ها از خانه $nexti[l], nextj[l]$ قرار دهید. این بدین مفهوم است که برای هر مقدار l هر خانه بعدی $(nexti[l] + ktmove1[k], nextj[l] + ktmove2[k])$ را برای اینکه مشخص شود آیا خروجی از $nexti[l], nextj[l]$ وجود دارد، تست می‌کند و همچنین تعداد چنین خروجی‌ها را از k در $exits[l]$ می‌شمارد. (یادآوری می‌شود که یک خانه به‌عنوان خروجی محسوب می‌گردد، اگر آن خانه در صفحه شطرنج قرار داشته باشد و همچنین قبلاً توسط اسب عبور نشده باشد). سرانجام *min* را برابر موقعیت مقدار حداقل exits قرار دهید. (بیش از یک رویداد مقدار حداقل exits وجود داشته باشد. اگر این مورد اتفاق بیافتد، مناسب است. min به اولین رویداد دلالت می‌کند، هرچند این مسأله مهم است که با چنین روشی پیدا نمودن یک راه‌حل تضمین نمی‌گردد، گرچه با استفاده از این روش شانس پیدا نمودن گردش اسب به‌طور کامل فراهم می‌شود و همین امر برای اهداف این مسأله کافی می‌باشد).

۲-۷ حرکت اسب. اعمال فوق را انجام دهید $i = nexti[\min]$، $j = nextj[\min]$ و $board[i][j] = m$ (بنابراین (i, j) نشان‌دهنده موقعیت جدید اسب می‌باشد و

$board[i][j]$ حرکت در روش و ترتیب صحیح را نگهداری می‌کند).

۸-۲ چاپ اطلاعات. محتویات board را که نشان‌دهنده راه‌حل مسأله گردش اسب می‌باشد را چاپ و سپس الگوریتم را خاتمه دهید.

برنامه‌ای بنویسید که براساس این الگوریتم عمل نماید. این تمرین به کمک لجن‌هاسن و ریمن طراحی شده است.

فصل سوم

پشته (Stack)

اهداف

در پایان این فصل شما باید بتوانید:

✓ مفهوم پشته را بیان کرده و دلیل استفاده از آن را ارائه دهید.

✓ پشته را طراحی و پیاده‌سازی کنید.

✓ کاربردهای پشته را بیان کنید.

✓ سه روش بیان عبارات محاسباتی را تشریح کنید.

✓ آیا پشته جوابگوی تمام نیازهای ما برای تعریف داده‌های مورد نیاز برنامه می‌باشد؟

سؤال‌های پیش از درس

۱. به نظر شما با توجه به آرایه لزوم تعریف یک ساختار داده جدید ضروری بنظر می‌رسد؟

۲. وقتی چاپگر بخواهد چند سند را به طور همزمان چاپ کند، به نظر شما چگونه عمل می‌کند؟

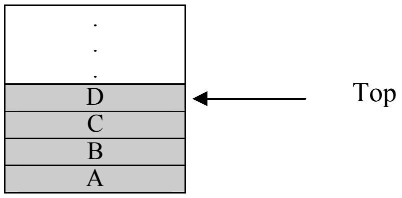
۳. چگونه می‌توان از یک آرایه به اینصورت استفاده کرد که فقط به یک طرف آن دستیابی داشته باشیم. مثلا فقط از انتهای آرایه بتوان عنصری را به آن اضافه یا حذف کرد؟

مقدمه

آرایه‌ها که در فصل قبل بررسی شدند، اجازه می‌دادند که عناصر را در هر مکانی از آرایه (ابتدای آرایه، انتهای آرایه یا وسط آرایه) حذف و یا اضافه کنیم. در علم کامپیوتر اغلب وضعیتهایی پیش می‌آید که می‌خواهیم در عمل حذف و یا اضافه عناصر محدودیتهایی ایجاد کنیم بطوری که این عملیات تنها در ابتدا یا در انتهای لیست عناصر انجام شود و نه در هر مکانی از آن. بدین منظور پشته مطرح گردید. این ساختار داده می‌تواند در بسیاری از مسائل به‌کار برده شود.

۱-۳ تعریف پشته

پشته، ساختار داده‌ای است که در آن عمل اضافه‌کردن یا حذف عنصر تنها از یک 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

پشته دارای چهار عنصر A,B,C,D می‌باشد. در این پشته D تنها عنصری از پشته است که در وضعیت فعلی، قابل دسترس می‌باشد. ساده‌ترین راه نمایش یک پشته استفاده از آرایه یک بعدی به طول n است که n بیانگر حداکثر تعداد عناصر پشته است و در کنار آرایه متغیری بنام top وجود دارد که به عنصر بالایی آن اشاره می‌کند.

دو عملگر خاص، برای دو عمل اساسی در پشته‌ها به‌کار می‌رود:

(الف) عملگر Push: برای اضافه‌کردن یک عنصر در پشته به‌کار می‌رود.
(ب) عملگر POP: برای حذف یک عنصر از پشته به‌کار می‌رود.

با توجه به تعریف پشته یک عنصر را می‌توان تنها از بالای پشته حذف و یا به بالای آن اضافه نمود و چون آخرین عنصر داده‌ای اضافه شده به پشته، اولین عنصر داده‌ای است که می‌توان از آن حذف کرد، به همین دلیل پشته را لیستهای آخرین ورودی، اولین خروجی (LIFO=Last Input First Output) می‌نامند.

نحوه قرارگیری عناصر در پشته را می‌توان شبیه تعدادی، بشقاب یا کتاب فرض نمود که روی هم قرار گرفته‌اند به وضوح برداشتن، آخرین بشقاب یا کتابی که گذاشته‌اید زودتر از بقیه امکانپذیر است.

۲-۳ نوع داده انتزاعی پشته

با توجه به عملکرد پشته که در بالا توصیف کردیم می‌توان پشته را به‌صورت یک نوع داده انتزاعی با عناصر و عملیات اصلی زیر تعریف کرد:

۱. مجموعه اقلام

مجموعه‌ای از عناصر که فقط از یک طرف موسوم به بالای پشته (top) قابل دستیابی‌اند.

۲. عملیات

- Create: یک پشته خالی ایجاد می‌کند
- Empty: عملیات تست خالی بودن پشته را انجام می‌دهد.
- Push: عمل افزودن عنصری به بالای پشته را انجام می‌دهد.
- POP عمل حذف از بالای پشته را انجام می‌دهد.
- Top به عنصر بالای پشته اشاره می‌کند.

قابل ذکر است در پشته دو حالت سرریزی (Overflow) و زیرریزی (underflow) می‌تواند اتفاق بیافتد. و هر دو حالت منجر به خطا در پشته می‌شوند که باید از آن‌ها اجتناب نمود. حالت سرریزی موقعی اتفاق می‌افتد که می‌خواهیم عمل اضافه‌کردن (Push) به داخل یک پشته پر انجام دهیم و حالت زیرریزی موقعی اتفاق می‌افتد که

می‌خواهیم از پشته خالی عنصری را حذف (POP) کنیم.

حال می‌خواهیم ADT بالا را پیاده‌سازی کنیم. ساده‌ترین روش پیاده‌سازی، استفاده از یک آرایه یک بعدی بنام stack با ساختار زیر می‌باشد:

```
typedef   struct {
            int   key ;
            /*  other fields  */
            } elementtype ;
elementtype   stack[MAXSTACK];
```

حال شرط خالی بودن و پر بودن پشته را بررسی می‌کنیم. برای اینکار همراه پشته یک متغیر به نام top تعریف می‌کنیم، که به عنصر بالائی پشته اشاره می‌کند. مقدار پشته وضعیت آن را در هر لحظه بیان می‌کند. برای اولین بار مقدار آن بصورت زیر می‌باشد:

```
int   top =  -1 ;
```

بنابراین شرط خالی بودن:

```
if (  top   == -1  )
     Cout << " Stack  is  Empty "
```

و شرط پر بودن:

```
if (  top   == MAXSTACK  )
     Cout << " Stack  is  Full "
```

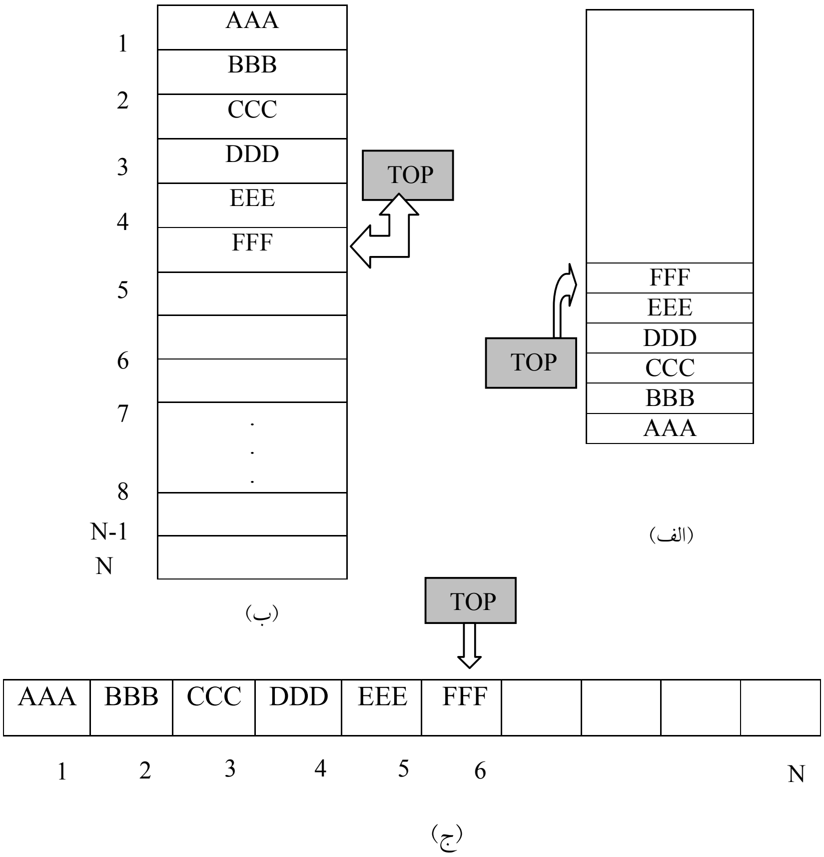
می‌باشد.

مثال ۱-۳: فرض کنید می‌خواهیم ٦ عنصر زیر را به ترتیب از چپ به راست در یک پشته خالی Push کنیم.

AAA,BBB,CCC,DDD,EEE,FFF

حل:

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



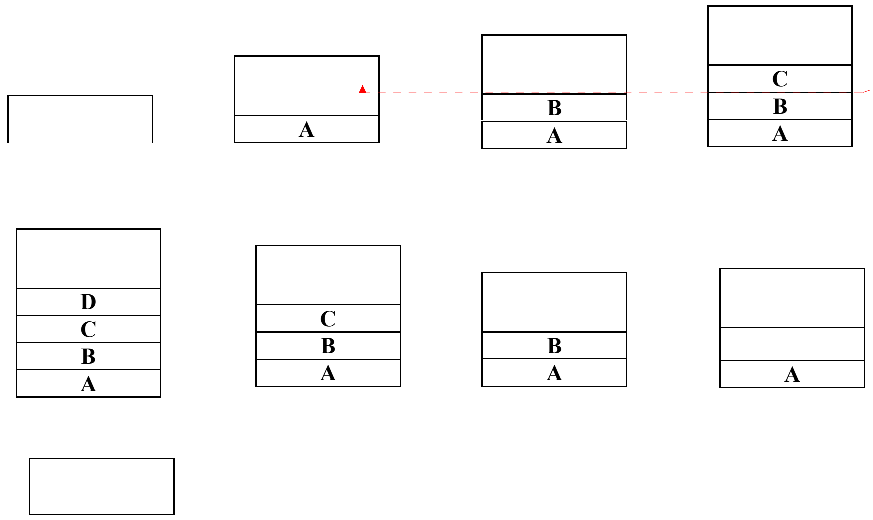
شکل ۲–۳ روش‌های نمایش پشته

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

شکل (۳–۳) برای پشته S و عنصر i عمل Push(s,i) موجب قراردادن عنصر i در بالای پشته S می‌شود، به‌طور مشابه عمل POP(s) عنصر بالای پشته را حذف می‌کند.

لذا دستور x=POP(s) عنصر بالای پشته را حذف و در داخل متغیر x قرار می‌دهد.
اعمال زیر را برای به‌دست آوردن پشته نهایی به‌کار می‌گیریم. (از چپ به راست)

Pash(S,A); Pash(S,B); Pash(S,C); Pash(S,D); POP(S); POP(S);
Push(S,E); POP(S);



Formatted: Font: Bold, Complex Script Font: Bold

شکل ۳-۳ نحوه درج و حذف از پشته

۳-۳ پیاده‌سازی عملگرهای پشته

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

پشته را به‌وسیله یک آرایه به نام 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 1- = =top مبین این است که پشته خالی است.

عمل اضافه کردن (Push) یک عنصر به درون پشته و عمل حذف کردن (POP)

از یک پشته را با توابع Push و POP پیاده سازی می‌کنیم.

پیاده‌سازی تابع اضافه کردن یک عنصر به پشته

```
void    Push (int *top , elementtype  item)
        {       /*  Add an item to the stack   */
              if (*top > = (MAXSTACK – 1))
                   {
                     Stackful () ;      // return an error key
                     return   ;
                   }
           stack [  + + *top]=item;
               }
```

پیاده‌سازی تابع حذف کردن یک عنصر به پشته

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

توجه: در زیر توابع فوق elementtype می‌تواند هر نوع داده‌ای وابسته به عناصر موجود در پشته باشد. اگر عناصر موجود در پشته عدد باشند elementtype، int و اگر عناصر موجود در پشته کاراکتر باشند elementtype، char خواهد بود.

در اجرای **تابع** Push نخست باید تحقیق کنیم که آیا جا برای عنصر جدید در پشته وجود دارد یا خیر؟ و به‌طور مشابه در اجرای تابع POP نخست باید تحقیق کنیم که آیا عنصری در پشته برای حذف وجود دارد یا خیر؟

نکته: سرریزی (Overflow) در درج و عمـل زیرریـزی (Underflow) در حـذف اتفاق می‌افتد.

یک تفاوت اساسی بین زیرریزی و سرریزی در ارتباط با پشته‌ها نمایان مـی‌شـود زیرریزی به میزان زیادی به الگوریتم داده شده و داده ورودی بستگی دارد و از ایـن رو برنامه‌نویس هیچ کنترل مستقیمی بر آن ندارد. اما سرریزی بستگی بـه مقـدار حافظـه دارد که برای هر پشته ذخیره مـی شـود. همچنـین ایـن انتخـاب تعـداد دفعـات وقـوع سرریزی را تحت‌الشعاع خود قرار می‌دهد.

در حالت کلی تعداد عناصر یک پشته با اضافه‌شـدن یـا کـم‌شـدن عناصـر تغییـر می‌کند. بنابراین انتخاب مقدار حافظه برای هر پـشته داده مـستلزم تـوازن بـین زمـان و حافظه است. به‌ویژه این که در ابتدا ذخیره مقدار زیـاد حافظـه بـرای هـر پـشته تعـداد دفعات وقوع سرریزی را کاهش می‌دهد. با وجود این در اکثر کارها بنـدرت از حافظـه زیاد استفاده می‌شود. مصرف حافظه زیاد برای جلـوگیری از مـسئله سـرریزی پرهزینـه خواهد بود و زمان موردنیاز برای حل مسئله سرریزی، مثل اضافه‌کـردن حافظـه اضـافی به پشته می‌تواند پرهزینه‌تر از حافظه اولیه باشـد. روش‌هـای متعـددی وجـود دارد کـه 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. یـک نمونـه از چنـین روشی در مثال زیر بیان شده است.

۳-۳-۱ تحلیل پیچیدگی زمانی

همانطور که ملاحظه کردید تـابع Push بـرای پـشته از تعـدادی عملیـات ثابـت (نظیـر جایگزینی، جمع و غیره) تشکیل شده است. بنابراین پیچیدگی زمانی آن ثابت بوده و از مرتبه $O(1)$ خواهد بود.

تابع POP برای پشته نیز مثل تابع Push بوده و از تعدادی دستور بـا زمـان ثابـت تشکیل شده است. لذا مرتبه زمانی آن نیز $O(1)$ خواهد بود.

۳-۳-۲ پشته‌های چندگانه

توصیف پشته‌های چندگانه را با ارائه چند مثال شروع می‌کنیم.

مثال ۲-۳: فرض کنید یک الگوریتم داده شده بـه دو پـشته A و B احتیـاج دارد. بـرای پشته A یک آرایه STACKA با n1 عنصر و برای پشته B یک آرایـه STACKB بـا n2 عنصر می‌توان تعریف کرد. سرریزی چگونه رخ می‌دهد؟

حل: سرریزی وقتی اتفاق می افتد:

پشته A شامل بیش از n1 عنصر باشد یا پشته B بیش از n2 عنصر داشته باشد.

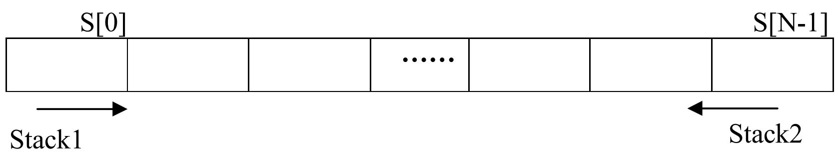
مثال ۳-۳: فرض کنید یک الگوریتم به دو پشته A و B احتیاج دارد. برای پشته A یـک آرایه STACKA با n1 عنصر و برای پشته B یک آرایه STACKB با n2 عنصر می‌توان تعریف کرد.

حل:

به جای تعریف دو ارایه جداگانه برای دو پشته می توان یک آرایه به نـام STACK را با اندازه n=n1+n2 برای پشته‌های B,A تعریف کرد، مانند آنچه که در شکل زیـر آمـده است. STACK[0] را به ابتدای پشته A تعریف می‌کنیم و به A اجازه می‌دهـیم بطـرف راست رشد کند و STACK[n-1] را ابتدای پـشته B تعریـف مـی‌کنـیم و بـه B اجـازه می‌دهیم به‌طرف چپ رشد کند. سرریزی تنها وقتی اتفاق می‌افتد:

که A و B جمعا بیش از n عنصر داشته باشنداین روش معمـو لاً تعـداد دفعـات سرریزی را کاهش می‌دهد حتی اگر ما تعداد کل فضای ذخیره شـده بـرای دو پـشته را افزایش ندهیم. بدین ترتیب از حافظه موجود به‌صورت بهینه استفاده می‌گردد.

هنگام استفاده از این ساختمان داده، عملگرهای Push و POP لازم است اصـلاح شوند.



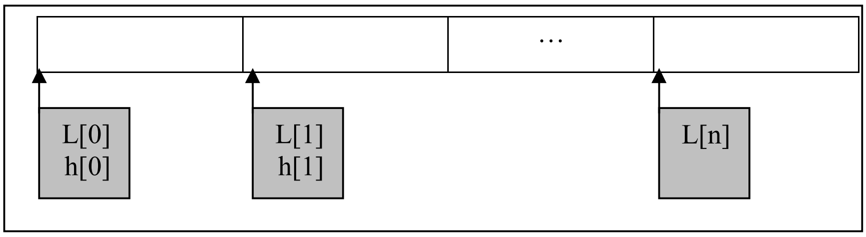
حال اگر در برنامه به بیش از دو پشته نیاز داشته باشیم می‌توان با الگـو گـرفتن از مثال بالا پشته‌های چندگانه را تعریف کرد.

برای نمایش n پشته، حافظه S[m] را به n قسمت تقسیم می‌کنیم. 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 L[i] پایین‌ترین و h[i] به بالاترین عنصر پشته i اشاره می‌کند و اگر L[i]=h[i] باشد آنگاه پشته i ام خالی است و مقدار اولیه L[i] و h[i] را به‌صورت زیر تعریف می‌کنیم:

$$L[0] = h[0] = -1 \qquad i = 0$$
$$L[i] = h[i] = (m/n) \times i \qquad 1 \le i < n$$
$$L[n] = m - 1 \qquad i = n$$

تقسیم‌بندی اولیه آرایه بطول M به N پشته مساوی به‌صورت شکل زیر می‌باشد:



حال توابع Push و Pop این نوع پشته را پیاده‌سازی می‌کنیم:

پیاده‌سازی تابع Push یک عنصر در پشته‌های چندگانه

```
void Push (int i , elementtype  item)
{
     if ( h[i] = = L[ i + 1] )
          stackfull();
     else
      {
          h[i] = h[i] +1 ;
          stack [h[i]] = item ;
       }
}
```

پیاده‌سازی تابع POP یک عنصر در پشته‌های چندگانه

```
void  POP (int  i , elementtype  * item)
{
     if (L[i] = = h[i])
          stackempty() ;
     else
      {
          *item = stack [h[i]] ;
          h[i] = h[i] -1 ;
       }
}
```

حال مثالی از چگونگی به‌دست آوردن ابتدای هر پشته در ساختار پـشته‌هـای چندگانـه ارائه می‌دهیم.

مثال ٤-٣: اگر در آرایه S[495] بخواهیم ٤ پشته درست کنیم، آدرس ابتدای هر پشته را به‌دست آورید؟

حل:

L[0]=-1
L[1]=[495/4]*1= 123
L[2]=[495/4]*2= 123*2=246
L[3]=[495/4]*3= 123*3=369

پس پشته اول از آدرس 0 شروع می‌شود و تا آدرس 212 طول دارد و پشته 2 از آدرس 123 شروع می‌شود و تا آدرس 245 ادامه دارد و....

٤-٣ دو کاربرد از پشته‌ها

در اینجا قصد داریم کاربردهایی از پشته‌ها را در حل بسیاری از مسائل ارائه دهیم:

الف) کاربرد پشته در فراخوانی تابع

پشته‌ها اغلب برای بیان ترتیب مراحل پردازش‌ها، به‌کار می‌رود که در آن مراحل، یعنـی

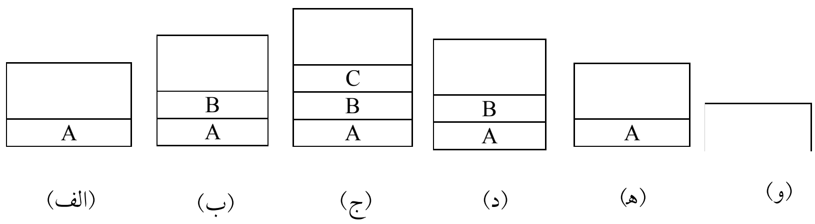
پردازش باید تا برقراری و محقق شدن شرایط دیگر به تعویق بیافتند. به مثال زیرتوجه کنید.

فرض می‌کنیم A یک برنامه اصلی و B و C و D زیر برنامه‌هایی هستند که به ترتیب داده شده فراخوانی می‌شوند.

فرض کنید که هنگام پردازش برنامه A نیازمند آن باشیم که روی تابعی به‌نام 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 ٤–٣ (الف)

سپس، شروع به پردازش B می‌کنیم و هنگام پردازش آن نیازمند کار روی تابع C می‌باشیم آنگاه همانند شکل ٤–٣ (ب) B را در پشته، بالای A قرار می‌دهیم و شروع به پردازش C می‌کنیم. علاوه بر این فرض کنید هنگام پردازش C به ترتیبی که گفته شد، پردازش D شروع می‌شود. آنگاه C را در پشته، بالای B قرار می‌دهیم شکل ٤–٣ (ج) و شروع به پردازش D می‌کنیم.

از طرف دیگر فرض کنید توانایی کامل کردن تابع D را داریم. در اینصورت تنها برنامه‌ای که می‌توانیم پردازش آن را ادامه دهیم برنامه C است که در بالای پشته است. از این رو پوشهٔ برنامه C را از پشته حذف می‌کنیم پشته به‌صورتی که در شکل ٤–٣ (د) نشان داده شده باقی می‌ماند و پردازش C ادامه می‌یابد. به همین ترتیب پس از کامل شدن پردازش C پوشه B را از بالای پشته حذف می‌کنیم و پشته به‌صورتی که در شکل (هـ) به تصویر کشیده شده است باقی می‌ماند و پردازش B ادامه می‌یابد. و بالاخره پس از کامل شدن پردازش B، آخرین پوشه، A را از پشته حذف می‌کنیم، پشته خالی می‌شود و پردازش برنامه اصلی ما A ادامه می‌یابد.



شکل ٤–٣ نمایش پشته فراخوانی توابع

ب) کاربرد پشته در ارزیابی عبارات

در این بخش با استفاده از پشته‌ها به ارزیابی عبارات ریاضی می‌پردازیم و قصد داریم عباراتی به شکل Prefix، infix و Postfix را مورد ارزیابی قرار می‌دهیم.

یکی از کاربردهای مهم پشته، ارزیابی عبارات می‌باشد. عبارات به سه شکل نوشته می‌شوند، هرگاه عملگر میان دو عملوند A و B قرار گیرد این عمل به‌صورت A+B نوشته می‌شود که به عبارت infix معروف است. اگر عملگر قبل از دو عملوند A و B قرار بگیرید به آن، عبارت prefix می‌گویند. و اگر عملگر بعد از دو عملوند A و B قرار بگیرید به آن، عبارت postfix می‌گویند:

+AB prefix (پیشوندی)

AB+ postfix (پسوندی)

پسوندهای pre، post و in طریقه قرارگرفتن عملگرها را نسبت به عملوندها 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) نیز معروف است عملگرها بعد از عملوندها قرار می‌گیرند. عبارات prefix و postfix برخلاف ظاهرشان به سهولت مورد استفاده قرار می‌گیرند

برای آشنایی از کاربرد روش‌های فوق مثالی را ارائه می‌دهیم. ارزیابی عبارتی مثل A+B*C که به‌صورت 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 به‌صورت A+(B*C) تفسیر می‌شود. برای حل این مسئله، با تبدیل عبارت infix به prefix یا postfix نیازی به هیچ گونه پرانتزگذاری در عبارات نداریم. و ترتیبی که در آن عملیات انجام می‌شوند به‌وسیله مکان عملگرها و عملوندها به‌طور کامل تعیین می‌شود

کامپیوتر معمو لاً عبارت محاسباتی نوشته شده به‌صورت نمادگذاری میانوندی را، در دو مرحله ارزیابی می‌کند. نخست عبارات مزبور را به‌صورت نمادگذاری پسوندی

تبدیل می‌کند سپس، عبارات پسوندی را ارزیابی می‌کند. پشته ابزار اصلی برای این تبدیل است.

حال می‌خواهیم با تبدیل عبارات با فرم‌های مختلف به هم، درک صحیح و دقیقی از نحوه ارزیابی عبارات داشته باشیم.

۱. تبدیل عبارات infix (میانوندی)به postfix (پسوندی) و prefix (پیشوندی)

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 به‌صورت 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 به‌دست 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

مثال ۵-۳: عبارت a*b+c-a/d را به‌صورت عبارت پسوندی و پیشوندی بنویسید.

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

* /
+ -

با توجه به این جدول، اولویت ضرب و تقسیم نسبت به جمع و تفریق بیشتر است. همچنین اگر دو عملگر دارای اولویت یکسان باشند (مانند ضرب و تقسیم) عبارات از چپ به راست ارزیابی می‌شوند. با توجه به این اولویت‌ها عبارت را به‌طور کامل

پرانتزگذاری می‌کنیم:

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

در عبارات پسوندی همانطور که گفتیم عملگر را بعد از پرانتز بسته خودش قرار می‌دهیم و سپس پرانتزها را حذف می‌کنیم.

ab*c+ad/ -

همچنین، همانطور که گفتیم در عبارات پیشوندی عملگر را قبل از پرانتز باز خودش قرار می‌دهیم و سپس پرانتزها را حذف می‌کنیم.

- + * abc / ad

۲. تبدیل عبارتهای میانوندی به عبارتهای پسوندی با استفاده از پشته

کامپایلرها برای محاسبه عبارت پسوندی با استفاده از پشته، از الگوریتم زیر استفاده می‌کنند.

عنوان الگوریتم	الگوریتم تبدیل عبارت میانوندی به پسوندی
ورودی	عبارت میانوندی
خروجی	عبارت پسوندی

۱. پشته‌ای خالی برای عملگرها ایجاد کنید.

۲. عبارت میانوندی را از سمت چپ به راست بخوان وتا زمانی که به انتهای عبارت بعدی نرسیدی، اعمال زیر را انجام بده:

الف) نشانه بعدی (ثابت، متغیر، عملگر، پرانتز باز، پرانتز بسته) را از عبارت میانوندی دریافت کن

ب) اگر نشانه:

- پرانتز باز است آن را در پشته قرار بده.
- عملوند است آن را در خروجی بنویس.
- 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 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.

○ 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 آنرا در خروجی نمی‌نویسیم.

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

مثال ٦-۳: شکل زیر مراحل تبدیل عبارت a+b*(c/(d+e))*f به عبارت پسوندی را نشان می‌دهد.

جدول ۱-۳: خانه‌های پشته

ورودی میانوندی	[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*+

۳. ارزیابی یک عبارت پسوندی

فرض کنید P یک عبارت محاسباتی است که به‌صورت پسوندی نوشته شده باشد می‌توان با کمک پشته به‌صورت زیر عبارت موردنظر را ارزیابی کرد. عبارت پسوندی از چپ به راست خوانده می‌شود و به محض مشاهده عملوند، آن‌را در پشته Push می‌کنیم، سپس با مشاهده یک عملگر، 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 ۷–۳: عبارت محاسباتی M زیرا که به‌صورت پسوندی نوشته شده است درنظر بگیرید:

M: 5 6 2 + * 12 4 / -

حاصل آن را با استفاده از پشته محاسبه می‌کنید.

حل: بدین صورت عمل می‌کنیم 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 می‌کنیم و جواب را به پشته بر می‌گردانیم و کار را از ورودی ادامه می‌دهیم.

مرحله	ورودی	پشته
۱	۵	۵
۲	٦	۵,٦
۳	۲	۵,٦,۲
٤	+	۵,۸
۵	*	٤۰
٦	۱۲	٤۰,۱۲
۷	٤	٤۰,۱۲,٤
۸	/	٤۰,۳
۹	−	۳۷

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

۶-۳ ارزیابی درستی پرانتزها توسط پشته

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. به‌عنوان 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.

حال کار را با یک مثال شروع می‌کنیم.

نخست عبارتی مثل: A+B یا (A+B)) را در نظر بگیرید. این عبارت شرط اول را نقض می‌کند و عباراتی مثل:

(A+B))=(C+D) یا (A+B)-C

شرط دوم را نقض می‌کند.

اکنون مسئله را کمی پیچیده‌تر کرده و فرض می‌کنیم در یک عبارت از ۳ جدا کننده، پرانتز، براکت و آکولاد استفاده شود محدوده‌ای که توسط هر کدام از آن‌ها باز می‌شود باید با جداکننده‌ای از همان نوع بسته شود. برای مثال عبارت ([A+B],[(A+B]) را در نظر بگیرید. در این عبارت 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 نشود.

پشته می‌تواند برای نگهداری انواع محدوده‌هایی که باز شده‌اند به‌کار رود. وقتی که یک بازکننده محدوده مشاهده شود، در پشته نگهداری می‌شود. پس از رسیدن به یک جداکننده پایانی محدوده، عنصر بالای پشته بررسی می‌شود. 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 با نوع خاتمه‌دهنده محدوده یکسان باشد به پیمایش رشته ادامه داده وگرنه رشته نامعتبر است، پس از رسیدن به انتهای رشته، پشته باید خالی باشد، در غیر این‌صورت، محدوده‌ای باز

شده، ولی بسته نشده و رشته معتبر نیست.

الگوریتم این روند به‌صورت زیر است:

الگوریتم تشخیص صحت پرانتزگذاری

```
valid = true;
S = the empty stack  ;
while (we have not read the entire string)
{
        read the next symbol of the string;
        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 operator of symb)
            valid= false ;
        if (! empty (S))
           valid = false;
        if (valid)
             Print the string is valid ;
        else print the string is invalid
}
```

۳-۷ مزایا و معایب پشته

همانطور که ملاحظه کردید ساختار داده پشته دارای عملگرهای Push و POP بود. که این عملگرها از پیچیدگی زمانی خوبی برخوردارند. بنابراین از نظر پیچیدگی این ساختار داده خوب عمل می‌کند. به‌طوری‌که زمان دو عملگر بالا O(1) می‌باشد. از معایب این ساختار داده می‌توان به عدم وجود عملگر جستجو، درج در جای مناسب و حذف دلخواه در این ساختار داده می‌توان اشاره کرد.

۸-۳ طراحی و ساخت کلاس پشته

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

۱. طراحی کلاس پشته ۲. پیاده‌سازی کلاس پشته.

۱. 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.

۲. پیاده‌سازی کلاس پشته

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

۱. تعریف اعضای داده‌ای 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، به‌عنوان 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 نشان‌دهنده موقعیت صفر بالای پشته در آرایه item است.

۳. 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 به‌صورت زیر پیاده‌سازی می‌شود.

```
Stack:: stack ()
{
      MyTop = -1 ;
}
```

٤. پیاده‌سازی عمل تست خالی بودن پشته

اگر s پشته موردنظر و myTop نشان‌دهنده عنصر بالای پشته باشد. myTop در پشته خالی برابر با ۱- است تابع empty() را می‌توان به‌صورت زیر پیاده‌سازی کرد:

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

۵. پیاده‌سازی عمل حذف از پشته

همان‌گونه که در بخش‌های قبلی 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 به برنامه فراخواننده بر می‌گرداند.

پیاده‌سازی عمل حذف از پشته

```
int stack:: POP ()
{
     if (empty ()){
        cout<< "stack is empty" ;
        exit () ;
      }
     else
          return items [myTop-- ] ;
}
```

برای استفاده از این تابع می‌توان به‌صورت 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 قرار می‌گیرد.

٦. پیاده‌سازی عمل افزودن به پشته

این تابع را می‌توان به‌صورت زیر نوشت:

پیاده‌سازی عمل افزودن به پشته

```
void   stack:: Push(int x)
{
       if (my top = = size – 1){
         cout << stack is full. Press any key …" ;
         getch ();
         exit();
        }
       else
           item [++ myTop] = x ;
}
```

۷. 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 به‌صورت زیر نوشت:

پیاده‌سازی عمل بازیابی از پشته

```
int stack:: top ()
{
    if (empty (s)){
       cout << "stack is empty. Press key.." ;
       getch ();
      exit ();
```

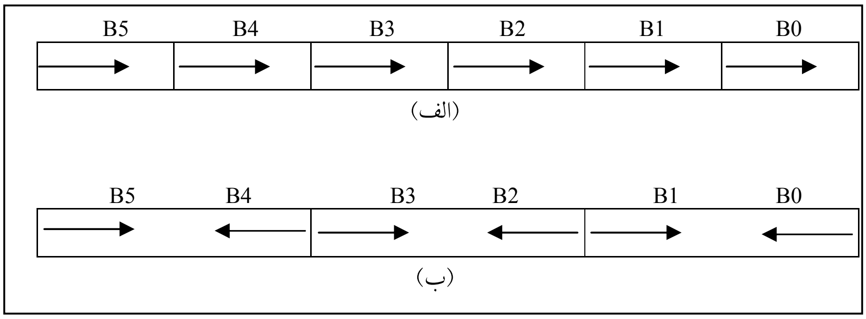
```
    }
    else
        return item [my top] ;
}
```

۹-۳ مثال‌های حل شده

در این بخش قصد داریم با استفاده از چند مسئله و پشته‌ها و کاربردهای آنرا مورد بررسی دقیق‌تر، قرار دهیم.

مثال ۱-۳: فرض کنید k=6 پشته فضای معلوم s که از N خانه همجوار حافظه تشکیل شده، اختصاص داده شده است. روش‌های نگهداری پشته‌ها را در s توضیح دهید.

حل: فرض کنید هیچ اطلاعاتی از قبل در دست نیست تا بیان کند یک پشته خیلی سریع‌تر از پشته دیگری رشد می‌کند. آنگاه می‌توان N/K خانه برای هر پشته درنظر گرفت این عمل در شکل (الف) نشان داده شده است که در آن $B_1, B_2, \ldots, B_6$ به ترتیب عناصر پایین پشته‌ها را نمایش می‌دهد. یا می‌توان پشته‌ها را به دو قسمت تقسیم کرد و 2 N/K خانه حافظه برای هر جفت پشته به‌صورتی که در شکل (ب) نشان داده شده است اختیار کرد. روش دوم می‌تواند تعداد دفعات وقوع سرریزی را کاهش دهد.



مثال ۲-۳: 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 به‌صورت قضیه زیر بیان می‌کنیم

قضیه ۱-۳: 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 مانند $i < j < k$ وجود نداشته باشد که $p_j<p_k<p_i$ باشد.

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

الف) 1 2 3 4 5 6 ← اندیس‌ها

در این دنباله اندیس 3<4<5 باید $p_3 < p_4 < p_5$ باشد که بدین صورت نیست، بلکه $p_4<p_5<p_3$ پس با هیچ ترتیبی نمی‌توان آن را از پشته خارج نمود.

سایر گزینه‌ها را می‌توان در خروجی چاپ نمود.

مثال ۳-۳: اگر دنباله اعداد 1,3,4,5,7 به ترتیب از سمت چپ به راست وارد پشته کنیم، کدام یک از خروجی‌های زیر از پشته امکان‌پذیر نیست؟

الف) 7 5 4 3 1   ب) 1 3 7 5 4

ج) 1 7 3 5 4   د) 1 4 3 7 5

حل: در قسمت ج اندیس 2<3<4 ولی $p_3 <p_4 < p_2$ پس آن را با هیچ ترتیبی نمی‌توان در خروجی چاپ کرد.

مثال ٤-۳: عبارت پسوندی (postfix) و پیشوندی معادل عبارت ریاضی *a/b-c+d*e-a*c/d* را به‌دست آورید.

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

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

برای به‌دست آوردن عبارت پسوندی عملگرها را به بعد از پرانتز مربوط به خودش انتقال می‌دهیم و سپس پرانتزها حذف می‌کنیم.

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

و برای به‌دست آوردن عبارت پیشوندی عملگرها را به قبل از پرانتز مربوط به خودش انتقال می‌دهیم و سپس پرانتزها را حذف می‌کنیم

*- + -/abc*de/*ac d*

مثال ۵-۳: معادل پیشوندی و پسوندی عبارت میانوندی زیر را پیدا کنید.

$((A+B)*(C-D))$

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

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

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

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

مثال ٦-۳: 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.عملگر مربوط به نزدیک‌ترین دو عملوند است.

$$\begin{aligned}&((((A+B)*C-(D-E))/(F+G))\\&\quad = AB+C*DE--FG+/\end{aligned}$$

مثال ۷-۳: عبارت ریاضی زیر را به روشهای مختلف پارانتز گزاری کنید.

$X = A/B - C + D * E - A * C$

برخی از روشهای پارانتزگزاری آن به‌صورت 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 عبارت X به‌صورت زیر می‌باشد.

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

۳-۱۰ تمرین‌های فصل

۱. زبان فرضی در نظر بگیرید که در آن، آرایه به‌عنوان نوعی داده نیست بلکه پشته به‌عنوان نوعی داده است. یعنی تعریف زیر ممکن است:

Stack s ;

همچنین فرض کنید اعمال Push، POP، test empty، و top در این زبان تعریف شده‌اند. نشان دهید که یک آرایه یک بعدی چگونه می‌تواند با استفاده از این اعمال بر روی دو پشته پیاده‌سازی شود.

۲. هر یک از عبارات زیر را به عبارات prefix 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 به‌عبارات 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. تابعی بنویسید که عبارتی محاسباتی را به‌صورت 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.

۶. الگوریتمی ارائه دهید که عناصری را خوانده و در پشته ذخیره کند و سپس با حداقل حافظه کمکی عناصر پشته را معکوس کند.

۷. الگوریتمی ارائه دهید که prefix را به postfix و برعکس تبدیل کند.

۸. الگوریتمی ارائه دهید که یک عبارت میانوندی را خوانده و تمام پارانتزهای اضافی را حذف کند.

۹. اگر کاراکترهایA، B، C و D به ترتیب وارد پشته شوند. چه خروجی‌هایی از این پشته امکان پذیر خواهد بود.

۱۰.پنج مثال برای کاربرد واقعی پشته نام ببرید.

۱۱.توضیح دهید که چگونه با پشته می‌توان بزرگ‌ترین مقسوم علیه مشترک دو عدد دلخواه را پیدا کرد.

۱۲.برنامه‌ای 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.(یعنی رشته متقارن است یا نه؟)

۱۳. ساده‌ترین راه برای پیاده‌سازی سه پشته در یک آرایه را ارائه دهید.

۱٤. آیا در حالت کلی می‌توان تعداد حالت‌ها و خروجی‌های مختلف با مقادیر A تا Z را در پشته S به‌دست آورد؟

۱۵. آیا در حالت کلی می‌توان تعداد حالت‌ها و خروجی‌های مختلف با مقادیر ۱ تا n را در پشته S بدست آورد؟

۱۱-۳ پروژه‌های برنامه‌نویسی

۱. یک ماشین حساب مهندسی را در نظر بگیرید که قادر است عباراتی را که شامل عملگرهای:

+,-,*,/,log,sin,cos, …

می‌باشند را محاسبه نماید. همچنین این دستگاه می‌تواند با علامت ( ) اولویت بین عملگرها قائل شود.
حال با توجه به توصیف بالا، برنامه‌ای بنویسید که کار ماشین حساب بالا را شبیه‌سازی نماید و با دریافت هر عبارت ریاضی مقدار آن را محاسبه کند.

۲. ماشینی را در نظر بگیرید که فقط دارای یک ثبات و شش دستورالعمل به‌صورت زیر باشد:

عملوند A را در ثبات قرار می‌دهد.	LD A
محتویات ثبات را در متغیر A قرار می‌دهد.	ST A
محتویات A را با ثبات جمع می‌کند.	AD A
محتویات A را از ثبات کم می‌کند.	SB A
محتویات A را در ثبات ضرب می‌کند.	ML A
محتویات ثبات را بر A تقسیم می‌کند.	DV A

برنامه‌ای بنویسید که یک عبارت Postfix حاوی عملگرهای یک کاراکتری و عملگرهای +,-,*,/ را پذیرفته و دنباله‌ای از دستورات را چاپ کند که عبارت را ارزیابی نماید و نتیجه را در ثبات قرار دهد. متغیرهای موقت را به‌صورت $TempP_n$ انتخاب کنید.

فصل چهارم

صف (Queue)

اهداف

در پایان این فصل شما باید بتوانید:

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

سؤال‌های پیش از درس

۱. به نظر شما با توجه به پشته لزوم تعریف یک ساختار داده جدید ضروری بنظر می‌رسد؟

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

همانطور که متوجه شدید پشته محدودیت‌های خاصی داشت بنابراین ساختار جدیدی به‌نام صف را مطرح می‌کنیم. صف، ساختار داده‌ای است که شامل لیست خطی از عناصر است که در آن عمل حذف عناصر تنها می‌تواند از یک طرف 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 به‌عنوان یک صف پیاده‌سازی شود که front نشان‌دهنده ابتدای صف و rear نشان‌دهنده انتهای صف می‌باشد.

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. صف‌ها در مقابل پشته‌ها قرار دارند که لیست‌های آخرین ورودی اولین خروجی (LIFO) هستند.

صف‌ها در زندگی روزمره ما به وفور دیده می‌شوند. به‌عنوان 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 به‌سر می‌برند.

۴-۱ نوع داده انتزاعی صف

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

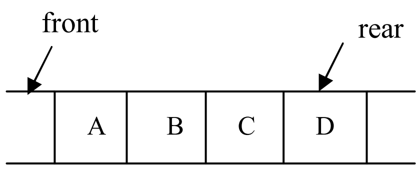
عناصر داده:

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

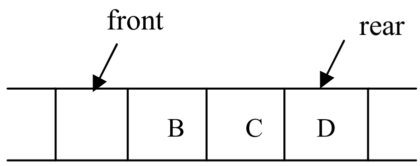
عملیات اصلی:

- Create برای ایجاد یک صف خالی
- queueempty عمل تست خالی بودن صف
- Addqueue افزودن عنصری به آخر صف
- Deletequeue حذف عنصری از ابتدای صف
- Process بازیابی عنصری از جلوی صف

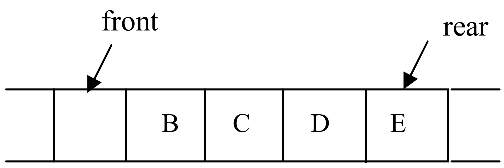
در صف (صرفنظر از نوع پیاده‌سازی) از دو متغیر اشاره‌گر به نام front که به عنصر قبل از عنصر ابتدایی اشاره می‌کند و دیگری rear که همیشه به آخرین عنصر اشاره می‌کند، استفاده می‌کنیم. شکل ۱-۴ صفی را نشان می‌دهد که حاوی چهار عنصر D, C, B, A می‌باشد. A در جلوی صف و D در انتهای صف قرار دارد. همانطور که مشاهده می‌کنید front به عنصر قبل از عنصر ابتدایی اشاره می‌کند و rear به آخرین عنصر اشاره می‌کند در شکل (۱-۴ ب) 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. برای حذف کردن ابتدا front را یک خانه به جلو حرکت می‌دهیم و سپس عنصر خانه‌ای که front به آن اشاره می‌کند را حذف می‌کنیم. شکل (۱-۴ج) عنصر E به صف به آخر صف اضافه شده است. برای اضافه کردن یک عنصر ابتدا rear را یک خانه به جلو حرکت می‌دهیم تا به خانه خالی اشاره کند و سپس در خانه خالی E را قرار دهیم.



(الف)



(ب)



(ج)

شکل ۱-٤

٤-۲ پیاده‌سازی عملگرهای صف

صف‌ها را می‌توان به صورت‌های متفاوتی نمایش داد. یک روش پیاده‌سازی صف این است که از یک آرایه برای ذخیره کردن عناصر صف و دو متغیر front و rear به ترتیب برای نمایش ابتدا و انتهای صف استفاده گردد. هر یک از صف‌های داخل کتاب توسط یک آرایه خطی queue و دو متغیر اشاوه‌گر front rear پیاده‌سازی می‌گردد. حال در حالت کلی ساختار داده صف را به صورت زیر تعریف می‌کنیم:

تعریف ساختار داده صف

```
struct  q{
          elementtype  items[maxqueue];
          int  front, rear;
          };
Struct  q  queue ;
```

قبل از پیاده‌سازی عملگرهای صف شرایط اولیه زیر را در نظر می‌گیریم.

مقداردهی اولیه $\quad$ queue .rear = = queue .front = = -1

پر بودن صف $\quad$ queue .rear = = maxqueue –1

خالی بودن صف queue .rear = = queue .front

تابع اضافه کردن به صف addqueue (queue, x) به صورت زیر می‌باشد:

پیاده‌سازی تابع اضافه کردن یک عنصر به صف

```
void   Addqueue  (struct q  *queue , elementtype  item)
{
  if (*queue.rear = = maxqueue –1)
    queuefull();              //پیغام پر بودن صف داده می‌شود
  else
        *queue.items [+ + queue->rear] = item ;
}
```

در **تابع** 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. **تابع** حذف اولین عنصر از صف Deletequeue (queue) به صورت زیر است:

پیاده‌سازی تابع حذف کردن یک عنصر از صف

```
elementtype  Deletequeue (struct  q *queue)
         {
          if (*queue.front = = *queue.rear)
            queueempty () ;   // پیغام پر بودن صف داده می‌شود
         else
            return queue->items [+ + queue->front];
         }
```

تابع نخست، خالی بودن صف را کنترل می‌کند. چون از صف خالی نمی‌توان عنصری حذف کرد. سپس با توجه به اینکه، 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، به نظر می‌رسد که صف پر می‌باشد، در حالی که امکان دارد به دلیل حذف عنصری از صف، اوایل صف خالی باشد، بنابراین مشکل اصلی صف معمولی این است که فقط یک بار قابل استفاده است.

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

یک روش برای حل این مشکل، این است که، بازای هر حذف، تمام عناصر به ابتدای صف شیفت داده شوند. اما، تغییر مکان عناصر در یک آرایه بسیار وقت‌گیر می‌باشد، مخصوصاً اگر آرایه دارای عناصر زیادی باشد. در واقع در بدترین حالت O(maxqueue) می‌باشد. برای رفع این مشکل از صف حلقوی استفاده می‌کنیم

۴-۲-۱ تحلیل پیچیدگی زمانی

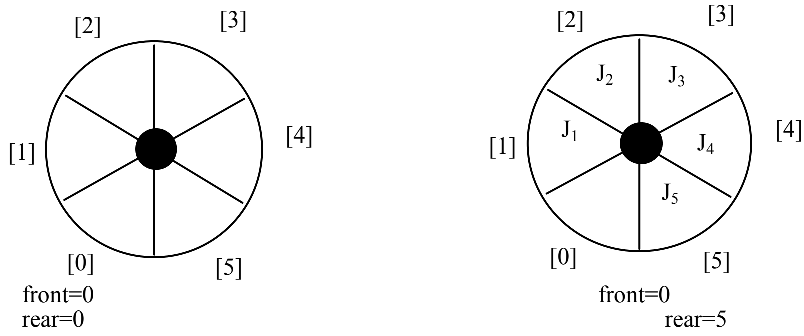
تابع اضافه کردن به صف همان‌طور که ملاحظه کردید از تعدادی عمل ثابت (جایگذاری، اضافه کردن، غیره) تشکیل شده است. بنابراین پیچیدگی زمانی تابع فوق O(1) خواهد بود.

تابع حذف در صف نیز مثل تابع اضافه کردن به صف از تعدادی عمل ثابت تشکیل می‌شود. بنابراین این تابع نیز پیچیدگی زمانی O(1) خواهد داشت.

۴-۳ صف حلقوی

صف حلقوی نمایش مؤثرتری برای پیاده‌سازی عملکرد صف می‌باشد. در این صف اندیس سر صف (front) همیشه به یک موقعیت عقب‌تر از اولین عنصر موجود در صف اشاره می‌کند و اندیس انتها (rear) به انتهای فعلی صف اشاره می‌کند. در این صف، انتهای صف با سر صف به نوعی در ارتباطند. اگر front = = rear باشد، صف خالی خواهد بود. اگر صف حلقوی فقط دارای یک مکان خالی 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 پر و خالی را از هم تشخیص دهیم. به همین دلیل در یک صف حلقوی به اندازه n در هر لحظه حداکثر n-1

وجود دارد بدین ترتیب می‌توان بین حالت پر و خالی تمایز قایل شد. شکل ۲-٤ نمایشی از صف‌های حلقوی تهی و پر می‌باشد.



الف) صف حلقوی خالی

ب) صف حلقوی پر

شکل ۲-٤ نمایش صف حلقوی پر و خالی

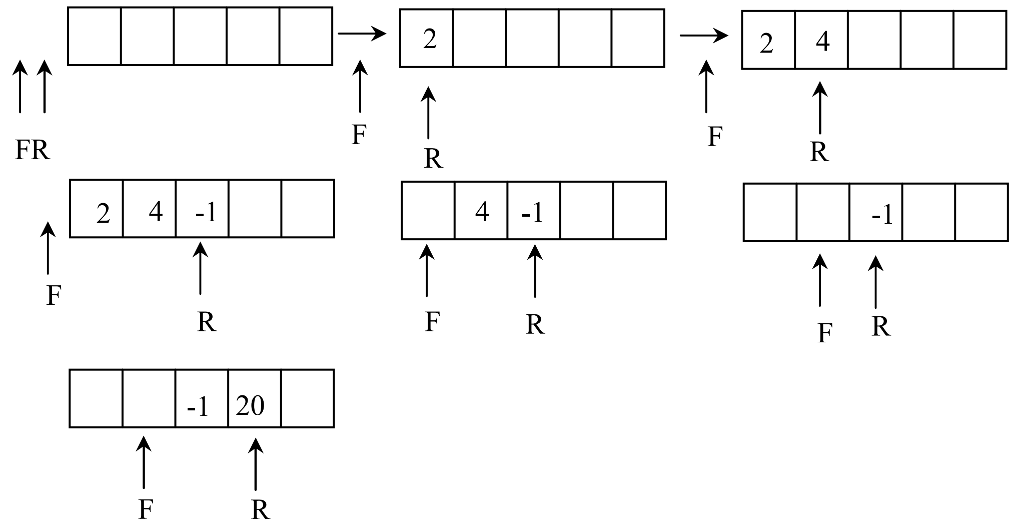
مثال ۱-٤: صف امروزه در بسیاری از مسائل کامپیوتر کاربرد دارد و شاید متداول‌ترین مثال، ایجاد یک صف از برنامه‌ها به وسیله سیستم عامل باشد. شکل (۳-٤) نشان می‌دهد که چگونه یک سیستم عامل ممکن است برنامه‌ها را به صورت نمایش ترتیبی صف اجرا کند.

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

شکل ۳-٤ جایگذاری و حذف از یک صف ترتیبی

مثال ۲-٤: عملیات زیر را به ترتیب از چپ به راست روی صف اعمال می‌کنیم

addq(2) , addq(4) , addq(-1) , deleq() , deleq() , addq(20)



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

rear = (rear+1) % n

به همین ترتیب در front, deleteq را به وسیله عبارت زیر چرخش می‌دهیم:

front = (front+1) % n

حال تابع حذف یک عنصر از صف حلقوی را به صورت زیر ارائه می‌دهیم:

پیاده‌سازی تابع حذف کردن یک عنصر از صف حلقوی

```
elementtype  deleteq  (struct q *queue)
{
                /* remove front element from the queue and put it in item */
               if (queue->front = = queue-> rear)
                      queueempty ();
               else
                     queue-> front = (queue->front + 1) % maxqueue;
                return queue->items [queue->front] ;
}
```

در زیر **پیاده‌سازی اضافه کردن یک عنصر به صف حلقوی** نشان داده شده است:

پیاده‌سازی تابع اضافه کردن یک عنصر به صف حلقوی

```
void   addq (struct q  * queue , elementtype  item)
        {
            /* add an item to the queue */
            *queue.rear = (*queue.rear+1) % maxqueue ;
                  if (*queue.front = = *queue.rear)
                      queuefull ();
                  else
                     *queue.items [*queue.rear] = item ;
        }
```

توابع () queueempty و () queuefull بدون توضیح ارائه شده‌اند. پیاده‌سازی آنها بسته به کاربردهای خاص می‌باشد و یا فقط می‌توانند یک پیغام خطا برگردانند.

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

۴-۴ صف اولویت (Priority queue)

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 deleteq اولین عنصری را که در صف وجود دارد حذف می‌کند. در این ساختارها هیچ ترتیبی در خروجی دیده نمی‌شود. برای حل این مشکل از صف اولویت استفاده می‌کنیم.

صف اولویت، ساختمان داده‌ای است که در آن ترتیب طبیعی عناصر (مرتب شده به‌صورت صعودی)، نتایج حاصل از عملیات روی این ساختار می‌باشد. به‌عبارت دیگر، 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 عنصر با کمترین

اولویت حذف می‌شود (حذف کوچک‌ترین عنصر). و صف اولویت نزولی همانند صف اولویت صعودی است با این تفاوت که در عمل حذف بزرگ‌ترین عنصر صف، حذف می‌شود.

ساختار بالا کاربردهای فراوانی دارد. یکی از مهمترین کاربردهای آن در سیستم عامل‌ها می‌باشد.

٤-٥ مزایا و معایب صف

همان‌طور که ملاحظه کردید ساختار داده صف دارای عملگرهای Addqueue و Deletequeue بود. که این عملگرها از پیچیدگی زمانی خوبی برخوردارند. بنابراین از نظر پیچیدگی این ساختار داده خوب عمل می‌کند. بطوریکه زمان دو عملگر بالا O(1) می‌باشد. از معایب این ساختار داده می‌توان به عدم وجود عملگر جستجو، درج در جای مناسب و حذف دلخواه در این ساختار داده می‌توان اشاره کرد.

٤-٦ طراحی و ساخت کلاس صف

ما در این بخش با توجه به نوع داده انتزاعی صف، به 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

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 داده‌ای برای نمایش شیء صف
۲. تعریف عملیاتی که در مرحله طراحی شناسایی می‌شوند.

در کلاس صف، اعضای داده‌ای، ساختار حافظه را بـرای عناصـر صـف تـدارک 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.
- دو متغیر صحیح که ابتدا و انتهای صف را مشخص می‌کند.
- توابع عضو کلاس صف

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 می‌کند.

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

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

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

تابع ()process: عنصر جلوی ف را بازیابی می‌کند.

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

تعریف کلاس صف

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

پیاده‌سازی عمل ایجاد صف

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

پس در ابتدا front برابر با 1- و rear **نیز برابر با ۱- می باشد. بنابراین اگر** rear = front صف خالی می‌باشد.

پیاده‌سازی عمل تست خالی بودن صف

```
int queue:: empty ()
  {
     if (rear ==  front)
         return 1;
    return 0;
   }
```

پیاده‌سازی افزودن عنصر به آخر صف:

```
int queue:: addq (int &x , int &overflow)
  {
     if (rear == Maxsize – 1)
          overflow = 1;
     else
        {
          overflow =0;
          item[++rear]=x;
        }
   }
```

پیاده‌سازی عمل حذف از جلوی صف:

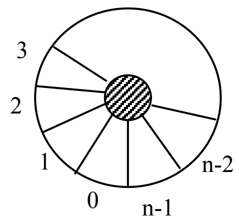
```
int queue:: deleteq (int &x , int &underflow)
  {
     if (empty())
         underflow = 1;
     else
       {
         underflow =0;
         x= item[front++];
       }
   }
```

توجه کنید که، در کلاس بالا دو متغیر underflow و overflow بترتیب برای خالی و پر بودن صف بکار برده می‌شوند. می‌توانستیم از شرط‌های خالی و پر بودن بجای آنها استفاده کرد، که به‌عنوان تمرین به خواننده واگذار می‌شود.

٤-٧ مسائل حل شده در صف‌ها

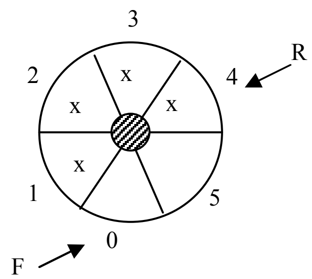
برای روشن شدن مفاهیم صف مثال‌هائی را در زیر ارائه می‌دهیم:

مثال ١-٤. با توجه به صف حلقوی شکل زیر، فرض کنید N تعداد اقلام در یک صف دایره‌ای باشد. متغیر F به خانه‌ای که بلافاصله قبل از جلوی صف قرار دارد اشاره می‌کند و متغیر R به عقب صف اشاره می‌کند. فرمولی که تعداد اقلام در یک صف دایره‌ای را محاسبه می‌کند بدست آورید.



حل: مسئله را برای دو حالت R > F و R < F حل می‌کنیم.

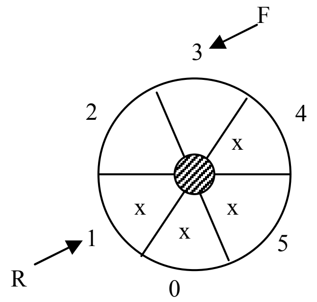
برای R > F شکل فرضی زیر را رسم می‌کنیم:



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:

$$M = \begin{cases} n-(F-R) & \text{if } F > R \\ R-F & \text{if } R > F \end{cases}$$

که در آن n تعداد خانه‌های صف می شد

مثال ۲-٤. 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:

$$F = 2\ , R = 2$$
$$F = 3\ , R = 3$$
$$F = 5\ , R = 5$$

در کلیه این حالت‌ها، صف خالی است.

و شرط پر بودن صف عبارت است از:

$$(Rear + 1) \bmod n = F \quad \Rightarrow \quad (Rear + 1) \bmod 7 = F$$

به‌عنوان مثال اگر Rear به خانه ٦ اشاره کند و Front به خانه ۰ اشاره کند، داریم:

$(6+1) \bmod 7 = 0$

پس صف پر می‌باشد.

مثال ۳-٤. 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 صف پر است یا خالی.

مثال-٤ عناصر صف‌های Q_2, Q_1 از چپ به راست به صورت زیر می‌باشند:

$$Q_1 = 10, 25, 17, 41, 44, 26, 75$$
$$Q_2 = 1, 5, 7, 4, 9, 6$$

اگر x و y عناصر صف باشند پس از اجرای قطعه کد زیر محتوای صف Q_3 چه مقداری خواهد بود؟

```
i = 0;
while ( !Empty(Q1) && !Empty(Q2) ){
        i++;
        x = Delete(Q1);
        y = Delete(Q2);
        if (y == i)
              Add(Q3, x);
}
```

حل: مراحل اجراء را در جدول زیر نمایش می‌دهیم:

i	1	2	3	4	5	6
X	10	25	17	41	19	26
Y	1	5	7	4	9	6
	AddQ			AddQ		AddQ

بنابراین خروجی برابر خواهد بود با:

۱۰ ٤۱ ۲٦

۸-٤ تمرین‌های فصل

۱. نشان دهید که چگونه می‌توان صفی از اعداد صحیح با استفاده از آرایه queue[100] که queue[0] برای نشان دادن ابتدای صف و queue[1] برای نشان دادن انتهای صف و queue[2] تا queue[99] برای نمایش عناصر صف به کار می‌روند پیاده‌سازی کرد. نشان دهید که چگونه می‌توان آرایه‌ای را به‌عنوان صف خالی ارزش‌دهی کرد؟ عملگرهای صف (درج، حذف و تست خالی) را برای این پیاده‌سازی بنویسید؟

۲. نشان دهید که چگونه می‌توان صفی را که عناصر آن متشکل از تعدادی متغیر از اعداد صحیح استی پیکلده‌ساز

۳. یک ADT (نوع داده انتزاعی) برای صف اولویت بنویسید.

٤. چگونه می‌توان چندین صف را داخل یک آرایی پیکلده‌سلزملگرهای مربوطه را بنویسید.

۵. چگونه می‌توان n صف متوالی حلقوی را در آرایه‌ای به طول q[size] نمایش داد. (عملگرهای addq، deleq و empty و full را بنویسید)

٦. الگوریتمکی بنویتید که چک ین عنصر صف را حذف کند. ضمن اینکه ترتیب بقیه عناصر تغییر نکند. عناصر صف اعداد طبیعی هستند. شما در مورد نحوه پیاده-سازی صف نباید هیچ فرضی بکنید. فقط می‌توانید از متدهای استاندارد صف و یا یک صف دیگر استفاده کنید. پیچیدگی زمانی برنامه شما چقدر است؟

۷. الگوریتمی بنویسید که یک رشته را از کاربر بگیرد و ابتدا تمام حروف بزرگ را به ترتیبی که در رشته آمده‌اند چاپ کند. سپس، تمام حروف کوچک را به ترتیبی که در رشته آمده‌اند چاپ کند و در نهایت تمام ارقام موجود در رشته را به همان ترتیبی که در رشته ظاهر شده‌اند، 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 می‌توان 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 پیچیدگی زمانی راه‌حل شما چقدر است ؟

۹. توضیح دهید که چگونه با صف می‌توان کوچک‌ترین مضرب مشترک دو عدد دلخواه را پیدا کرد.

۱۰. توضیح دهید که چگونه می‌توان توسط دو صف، یک پشته درست کرد.

۱۱. الگوریتمی ارائه نمائید 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 می‌توان توسط دو پشته، یک صف درست کرد.

۱۳. چگونه می‌توان یک پشته و صف را داخل یک آرایه نمایش داد. توابع حذف و اضافه را بنویسید.

۱٤. توضیح دهید چگونه می‌توان عناصر یک پشته را وارد یک پشته دیگر نمود به نحوی که ترتیب عناصر پشته دوم و اول یکسان باشند. می‌توانید از یک صف کمکی برای حل مسئله استفاده کنید.

۱۵. چگونه می‌توان با استفاده از متدهای استاندارد صف و بدون استفاده از پشته، ترتیب عناصر یک صف را معکوس کرد. برای حل مسئله می‌توانید از چندین صف استفاده کنید.

۱٦. الگوریتمی بنویسید که رشته‌ای از کاراکترها را از ورودی خوانده، هر کاراکتر را هنگام خواندن در یک پشته و در 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. (رشته‌ای متقارن است که وقتی ترتیب آن عوض شود، تغییر نمی‌کند. مثل madam، 532235)

۱۷. یک صف دو سویه (Double-Ended Queue) یک لیست خطی است که عناصر را می‌توان در آن از هر دو سو حذف یا اضافه کرد اما حذف یا اضافه کردن عنصر از وسط امکان‌پذیر نیست برای این ساختار داده تابع حذف را اضافه بنویسید.

۱۸. ساختار داده صف را در نظر بگیرید آیا به نظر شما در این ساختار داده می‌توان جستجو را انجام داد؟ دلیل خود را در هر صورت بیان کنید.

٤-۹ پروژه‌های برنامه‌نویسی

۱. برنامه‌ای برای شبیه‌سازی یک سیستم چندکاربره ساده بنویسید. سیستم به‌صورت زیر

کار می‌کند:

هر کار مرکب Id منحصر به فرد دارد و می‌خواهد تراکنش‌هایی را انجام دهد، اما در هر لحظه فقط یک تراکنش می‌تواند توسط کامپیوتر پردازش شود. هر خط ورودی یک کاربر را نشان می‌دهد و حاوی Id کاربر زمان شروع و تعدادی از اعداد صحیح است که نشان‌دهندهٔ مدت هر کدام از تراکنش‌های آن است. ورودی‌ها برحسب زمان شروع و به ترتیب صعودی مرتب می‌باشند و کلیه زمان‌ها و مدت‌ها برحسب ثانیه می‌باشند. فرض کنید یک کاربر تا زمانی که تراکنش قبلی آن پاسخ داده نشده، تراکنش دیگری را درخواست نمی‌کند و کامپیوتر به اولین تقاضا اول پاسخ می‌دهد برنامه باید سیستم را شبیه‌سازی کند و پیامی که حاوی Id کاربر زمان شروع و پایان تراکنش است را چاپ کند. در پایان شبیه‌سازی، برنامه باید متوسط زمان انتظار برای یک تراکنش را چاپ کند.

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 یک پیاده‌ساز
- ✓ لیست پیوندی دو طرفه و حلقوی را تعریف کرده و اعمال درج، حذف و پیمایش در آنها را پیاده‌سازی کنید و پیچیدگی زمانی اعمال فوق را تحلیل نمائید.
- ✓ پشته و صف را توسط لیست پیوندی پیاده‌ساز
- ✓ لیست پیوندی را با سایر ساختار داده‌ها مقایسه کنید.

سؤال‌های پیش از درس

۱. به نظر شما با توجه به آرایه، صف و پشته آیا لزوم تعریف یک ساختار داده جدید ضروری بنظر می‌رسد؟

۲. واگن‌های یک قطار که به هم وصل هستند مثالی از لیست می‌باشد. مثالهایی از لیست‌ها را در دنیای واقعی نام ببرید.

۳. در مثال واگن‌های یک قطار، هر واگن راهنمایی‌کننده واگن بعدی می‌باشد. آیا می‌توانید ساختار داده‌ای طراحی کنید که همچنین خصوصیتی داشته باشد؟

مقدمه

استفاده از اصطلاح «لیست» در زندگی روزمره به یک مجموعه خطی از اقلام داده‌ای گفته می‌شود. لیست دارای عنصر اول، عنصر دوم و... و عضرآخر می‌باشد. اغلب از ما خواسته می‌شود یک عنصر را به لیست اضافه کنیم یا آن را از لیست حذف کنیم.

داده‌پردازی که شامل ذخیره، بازیابی و پردازش داده‌ها است در لیست‌ها جزء اعمال رایج می‌باشد.

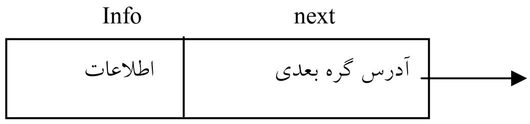
استفاده از آرایه‌ها، یک روش ذخیره چنین داده‌هایی است که در فصل ۲ مورد بحث و بررسی قرار گرفت. آرایه‌ها دارای معایبی بودند. به‌عنوان مثال اضافه کردن و حذف عناصر در آرایه نبتاً پرهزینه است‌علاوه بر این از آنجایی که هر آرایه معمو لاً یک بلاک از فضای حافظه را اشغال می‌کند از این رو تعداد عناصر قابل ذخیره در یک آرایه محدود به اندازه آرایه است و هنگام نیازبه ذخیره تعداد عناصر بیشتر از اندازه آرایه نمی‌توان اندازه آرایه را افزایش داد. 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 گره (node)، که شامل فیلدهای اطلاعات و آدرس گره بعدی در لیست است، 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.

نکته مهم: *در این فصل لیست‌های پیوندی را به‌صورت یک ساختمان داده در نظر می‌گیریم (یعنی روش پیاده‌سازی) و نه به عنوان نوع داده (یعنی ساختمان منطقی با اعمال ابتدائی تعریف شده). بنابراین در اینجا مشخصات ADT برای لیست پیوندی ارائه نمی‌دهیم).*

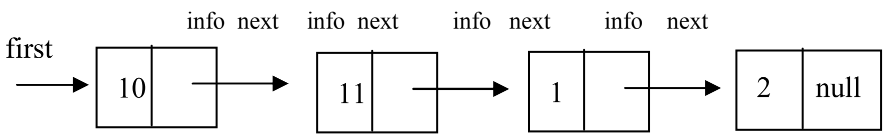
۱–۵ لیست‌های پیوندی خطی (یکطرفه)

لیست پیوندی خطی (اصطلاحاً لیست پیوندی) همانند پ شته 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 می‌کند. 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 گره اول، لیست است. برای تشخیص انتهای لیست، فیلد آدرس آخرین گره لیست به تهی (NULL) اشاره می‌کند.



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 به‌دست آوردن یک لیست خالی کافی است از عمل first = NULL استفاده گردد.

لیست پیوندی یک ساختار داده پویاست. تعداد گرهای لیست دائماً با درج و حذف عناصر تغییر می‌کند. طبیعت پویای لیست با طبیعت ایستای آرایه که طول آن ثابت باقی می‌ماند، مغایرت دارد.

مقایسه آرایه با لیست پیوندی
۱. لیست پیوندی یک ساختار داده پویاست، تعداد گرهای لیست دائماً با درج و حذف عناصر تغییر می‌کند. اما طول آرایه همیشه ثابت باقی می‌اند.
۲. طول آرایه ابتدای برنامه تعریف می‌شود و براساس تعریف یک تعداد از خانه های حافظه به‌طور پیوسته به آن تخصیص می‌یابد. اما طول لیست پیوندی براساس نیاز می‌تواند کم یا زیاد شود.
۳. هزینه‌های درج و حذف در آرایه‌ها بسیار پرهزینه می‌باشند.

۲-۵ پیاده‌سازی لیست پیوندی

در زبان C و ++C برای پیاده‌سازی لیست پیوندی، از اشاره‌گرها استفاده می‌شود. برای پیاده‌سازی لیست پیوندی، به ابزارهای زیر نیاز داریم:

ابزارهای مورد نیاز برای پیاده‌سازی 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

هر 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
        {
            elementtype info;
            Node * next;
         };
```

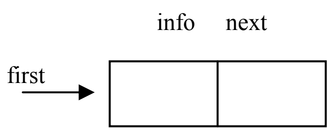
• به‌دست آوردن یک گره جدید و خالی از حافظه

در لیست پیوندی، گره‌های لیست در زمان اجرا می‌توانند ایجاد شوند. برای به‌دست آوردن یک گره جدید از تابع () getnode به صورت () p = getnode استفاده می‌کنیم. که این عمل یک گره خالی را ایجاد می‌کند و آدرس آن را در متغیر p قرار می‌دهد. گره اکنون p به ... ه می‌کند. خود تابع () getnode می‌تواند به صورت زیر پیاده‌سازی گردد:

پیاده‌سازی تابع () getnode: به‌دست آوردن یک گره جدید از حافظه

```
Node    *getnode( )
{
                                                        در C به‌صورت زیر:
    Node  *  first;
    first = (Node *) malloc ( sizeof (struct Node) );
                                                        در ++C به‌صورت زیر:
    Node  *  first;
    first =  new  struct Node ;
    return first ;
}
```

حال می‌خواهیم یک گره از لیست را تشکیل دهیم با یک دستور اشاره‌گر p را از نوع Node تعریف می‌کنیم و دستور دوم حافظه‌ای به اندازه ساختمان Node از سیستم می‌گیریم آدرس آن را در first قرار می‌دهیم. بنابراین خواهیم داشت:



• مراجعه به گره‌های لیست

اگر first یک اشاره‌گر خارجی به گره‌ای از لیست باشد، برای مراجعه به فیلد آدرس گره بعدی از $first \rightarrow next$ و برای مراجعه به فیلد اطلاعات از $first \rightarrow \inf o$

استفاده می‌کنیم.

• شروع لیست

فرض می‌کنیم اشاره‌گری به نام first (head) به ابتدای لیست اشاره می‌کند. اگر p نیز بخواهد به گره اول لیست اشاره کند با عمل p = first، p نیز به گره اول اشاره خواهد کرد.

• آزاد کردن حافظه

اگر به گره‌ای از لیست پیوندی نیاز نداشته باشیم، آن را به مخزن حافظه برمی‌گردانیم (حافظه ان را آزاد می کنیم). برای این منظور از تابع () free در C و از تابع delete در C++ استفاده می‌شود. به‌عنوان مثال، دستور زیر حافظه‌ای را که p به آن اشاره می‌کند، به مخزن حافظه برمی‌گرداند:

در C از تابع free (p)

و در C++ از تابع delete p

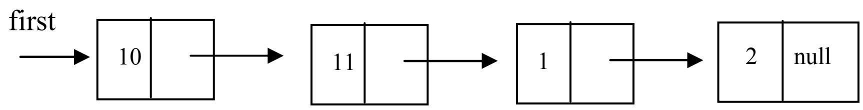
استفاده می‌شود.

• پیمایش لیست

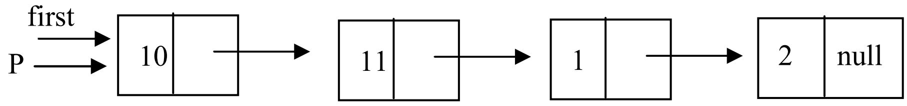
منظور از پیمایش لیست، این است که به تمام عناصر لیست دستیابی داشته باشیم و در صورت لزوم بتوانیم آنها را پردازش کنیم.

نکته: برای عمل پیمایش لیست باید به غیر از اشاره‌گر first که به ابتدای لیست اشاره می‌کند باید اشاره‌گر دیگری مانند p را با عمل p = first تعریف کنیم تا آن نیز به اول لیست اشاره کند.اگر این کار را نکنیم در آن صورت با حرکت first ابتدای لیست پیوندی را از دست خواهیم داد.

فرض کنید می‌خواهیم لیست زیر را پیمایش کنیم:



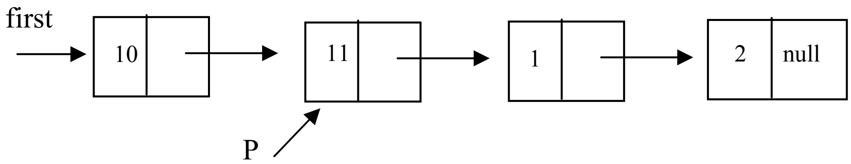
با دستور $p = first$، اشاره‌گر p نیز به ابتدای لیست اشاره خواهد کرد. با اشاره‌گر p می‌توانیم به تمام گره‌ها دستیابی داشته باشیم.



برای دستیابی به گره‌ای با محتوای 11 باید پیوندها را دنبال کنیم. P باید به گره‌ای اشاره کند که آدرس آن در فیلد آدرس گره با محتویات 10 قرار داد. بنابراین

$p = p \rightarrow next$

چون p در ابتدا به گرهی با محتویات 10 اشاره می‌کند با $p \rightarrow next$ ، p به گره با محتوای 11 اشاره خواهد کرد.



برای پیمایش گرهی با محتویات 1 باید روند قبلی را تکرار کنیم یعنی:

$p = p \rightarrow next$

اکنون برای پردازش محتوای خانه‌ای که p به آن اشاره می‌کند از عمل $p \rightarrow info$ استفاده می‌کنیم.

با توجه به آنچه گفته شد پیمایش لیست می‌تواند به صورت زیر پیاده‌سازی شود:

پیاده‌سازی پیمایش لیست پیوندی

```
p = first  ;
while ( p != NULL )
 {
        process ( p -> info ) ;  //   پردازش گره
        p =   p -> next   ; //   گره بعدی
 }
```

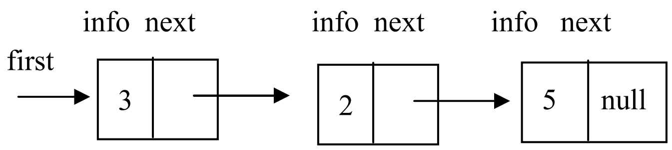
۳-۵ درج و حذف گره‌ها از لیست پیوندی

در این اینجا قصد داریم عملگرهای لیست‌های پیوندی را بررسی نمائیم. همانطور که

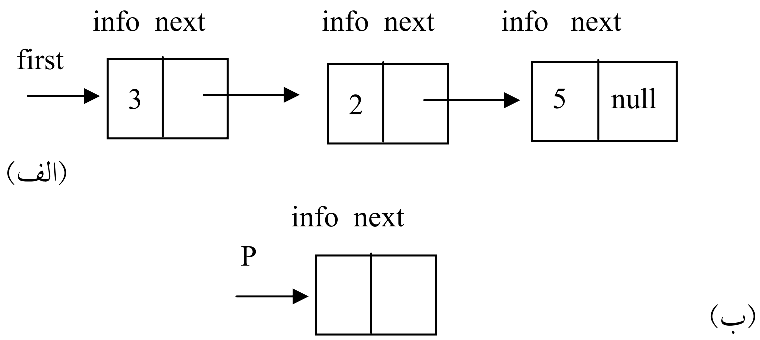
می‌دانید، عملگرهای درج و حذف در ساختار داده‌ها از اهمیت خاصی برخوردارند، لـذا در اینجا عملگرهای درج و حذف را بررسی می‌کنیم.

اضافه کردن گره به ابتدای لیست

فرض کنید لیست پیوندی اولیه زیر را داریم:



ابتدا با استفاده از عملگر getnode() یک گره خالی را ایجاد می‌کنـیم و آدرس آن را در متغیر p قرار می‌دهیم. شکل ۱–۵ ب وضعیت لیـست را پـس از بـه‌دسـت آوردن گـره جدید نشان می‌دهد:

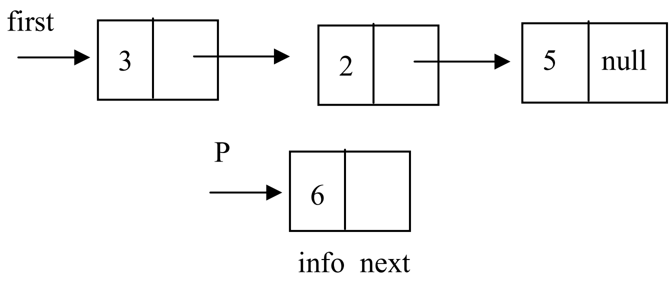


شکل ۱–۵: ایجاد گره جدید

در مرحله بعدی مقدار 6 در قسمت اطلاعات گره جدید درج می‌شـود. ایـن مرحلـه بـا عمل:

$p \to \inf o = 6$

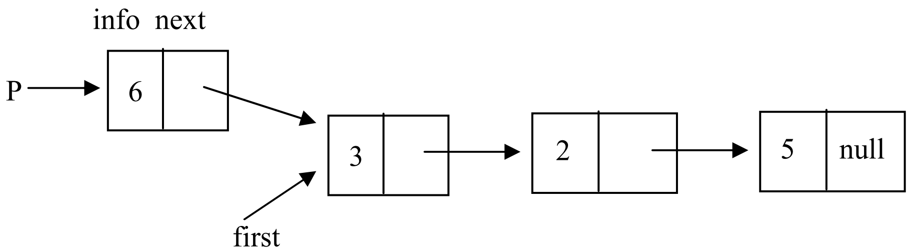
انجام می‌گیرد. (شکل ۲–۵)



شکل ۲-۵ درج اطلاعات در گره جدید

پس از مقداردهی قسمت اطلاعات گره قسمت آدرس گره جدید نیز باید مقدار بگیرد، چون گره جدید باید به ابتدای لیست اضافه شود. عنصری که در ابتدای لیست قرار دارد عمصر بعد از عنصر جدید خواهد بود.

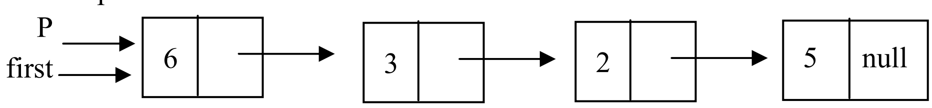
با اجرای عمل $p \rightarrow next = first$ ، مقدار first (که به آدرس اولین گره لیست اشاره می‌کند) در قسمت آدرس گره جدید قرار می‌دهد. (شکل ۳-۵)



شکل ۳-۵ 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 دستور زیر انجام می‌گیرد:

first = p



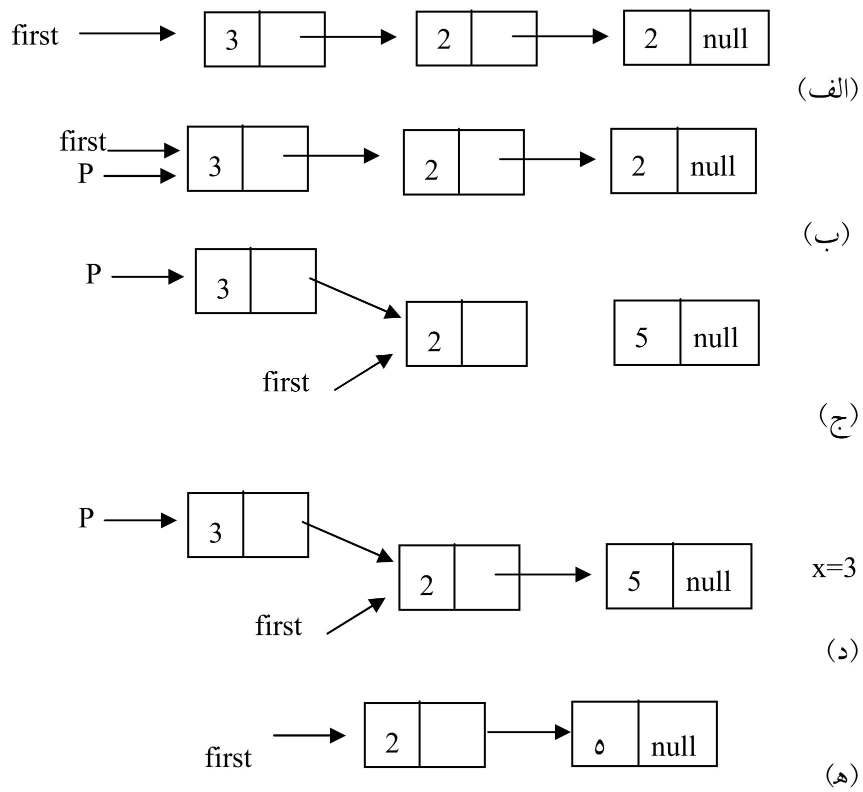
بنابراین الگوریتم افزودن عدد 6 به ابتدای لیست به صورت زیر خلاصه می‌شود:

$p = getnode();$
$p \rightarrow next = 6;$
$p \rightarrow next = first;$
$first = p;$

- **حذف اولین گره از لیست**

شکل ٤–٥ مراحل حذف اولین گره از لیست غیرتهی و قرار دادن مقدار آن در متغیر x را نشان می‌دهدعمل حذف گره دقیقاً عکس عمل افزودن یک گره به ابتدای لیست پیوندی می‌باشد.

$p = \quad first$
$first = p \rightarrow next$
$x = p \rightarrow \inf o$



شکل ٤–٥ طریقه حذف از لیست

همانطور که در شکل ٤–٥ (هـ) نشان داده شده است هیچ گره ای به گره‌ای که p بـه آن اشاره می‌کند دسترسی ندارد (اشاره نمی کند). بنابراین دسترسی به ایـن گـره از طریـق گره‌های لیست غیر ممکن است. در این صورت حافظه‌ای که در اختیار این گـره اسـت بلااستفاده می‌ماند. بنابراین باید مکانیزمی وجود داشته باشد که این خانـه بلااسـتفاده را برای کاربردهای بعد آزاد نماید. برای این منظور از عمل freenode (p) استفاده می‌شود.

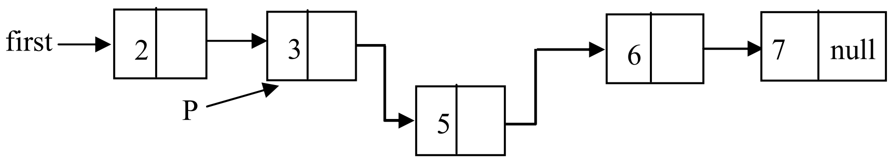
پـس getnode () گـره جدیـدی را ایجـاد مـی‌کنـد و freenode () آن را از بـین می‌برد. با این دید می‌توان گفت که گره‌ها مورد استفاده مجـدد قـرار نمـی‌گیرنـد، بلکـه ایجاد شده و از بین می‌روند.

- **درج یک عنصر در لیست مرتب**

در اینجا قصد داریم الگوریتم درج یک عنصر در لیست مرتب بطوریکه ترتیـب عناصـر بهم نخورد را بررسی کنیم. در زیر الگوریتم این مسئله را ارائه می‌دهیم:

فرض کنید می‌خواهیم گره‌ای با محتویـات 5 را در لیـست پیونـدی زیـر طـوری اضافه کنیم که ترتیب عناصر لیست بهم نخورد.

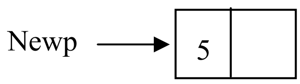
برای این کار ابتدا فرض می‌کنیم اشاره‌گری به نـام p را آن قـدر بـه طـرف جلـو حرکت می‌دهیم تا به گرهی که محتویـات آن ۳ اسـت اشـاره کنـد (چـون گـره جدیـد می‌بایست بعد از گره مذکور اضافه شود). بعد از انجام ایـن کـار مراحـل زیـر را انجـام می‌دهیم:



۱. ابتدا گره جدیدی را ایجاد می‌کنیم و آدرس آن را در Newp قرار می‌دهـیم و بخـش داده آن را برابر ۵ قرار می‌دهیم.

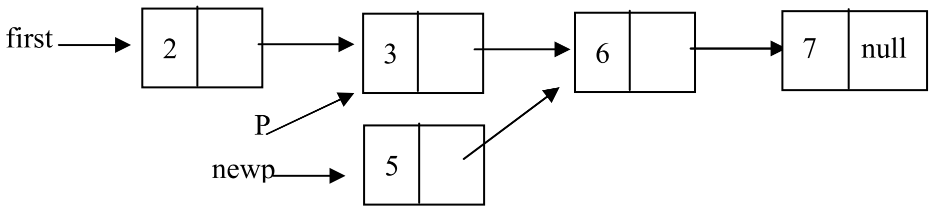
$Newp = getnode();$

$Newp \to info = 5;$



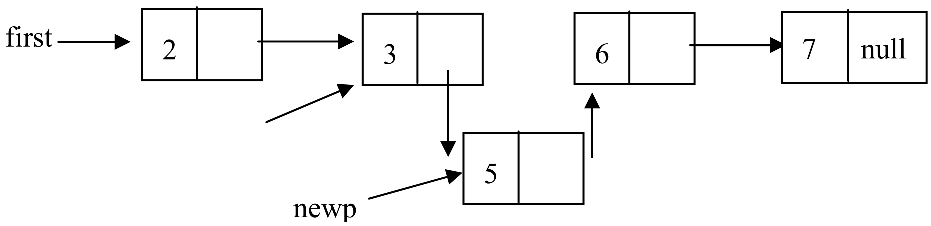
۲. فیلد آدرس گره‌ای را که Newp به آن اشاره می‌کند، برابر با بخش آدرس گره‌ای قرار دهید که p به آن اشاره می‌کند. یعنی:

$Newp \rightarrow next = p \rightarrow next;$



۳. فیلد آدرس گره‌ای را که p به آن اشاره می‌کند برابر با New p قرار دهید. یعنی

$p \rightarrow \text{next} = \text{new p};$



همان‌طور که ملاحظه کردید عنصر جدید در محل مربوط به خود قرار گرفت.

در حالت کلی، برای درج مقدار جدیدی در لیست پیوندی، به‌صورت زیر عمل می‌کنیم:

ابتدا باید گره جدیدی ایجاد کنیم و سپس مقدار را در فیلد اطلاعات گره جدید ذخیره می‌کنیم. درج گره در لیست دو حالت دارد که باید از هم تفکیک شود:

- ✓ اضافه کردن در ابتدای لیست(قبلاً به آن اشاره شد)
- ✓ درج گره جدید بعد از گره‌ای در لیست.

فرض کنید می‌خواهیم گره جدید با محتویات x را بعد از گرهی با محتویات y به لیست پیوندی(مرتب) 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 از تابع زیر استفاده می‌کنیم:

```
void        Insert(Node  *P, elementtype  x )
{
      Newp = getnode();
      if  (  IS_FULL(Newp) ) {
            cout<<" The  memory  is  full "
            exit(1);
       }
      Newp → inf o = x;
      Newp → next = p → next
      p → next = Newp;
}
```

- **عمل حذف گره از لیست پیوندی**

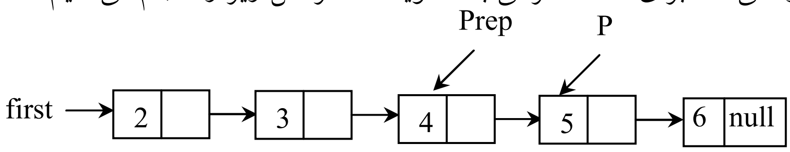
در عمل حذف نیز باید دو حالت را در نظر گرفت:

✓ حذف گره از ابتدای لیست(که قبلاً به آن اشاره شد)

✓ حذف گره‌ای که قبل از آن گره دیگری وجود دارد.

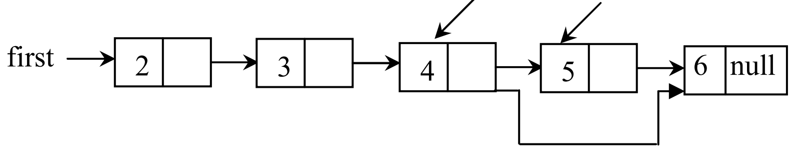
حال، حالت حذف گره‌ای که قبل از آن گره دیگری وجود دارد را توضیح می‌دهیم:

در لیست پیوندی زیرفرض کنید می‌خواهیم گرهی با محتویات ۵ را از لیست پیوندی حذف کنیم. فرض می‌کنیم اشاره‌گر p به گره‌ای با محتویات ۵ (گرهی که باید حذف شود) و Prep به گره‌ای با محتویات 4 (گره قبل از گره‌ای که باید حذف شود) اشاره می‌کند. برای حذف گرهی با محتویات 5 مراحل زیر را انجام می‌دهیم:



۱. فیلد آدرس Perp را برابر با فیلد آدرس گره p قرار دهید:

$$Prep \rightarrow next = p \rightarrow next$$



۲. گره‌ای را که p به آن اشاره می‌کند به مخزن حافظه برگردانید. (آزادسازی حافظه)
freenode (p);

حال می‌خواهیم تابع حذف از یک لیست (حذف از وسط لیست) را در حالت کلی با فرض اینکه آدرس عنصر قبل از عنصر مورد نظر(عنصری که قرار است حذف شود) در دسترس باشد ارائه دهیم.

فرض کنید p گرهی است که می‌خواهیم از لیست حذف کنیم و Pper آدرس گره قبل از p باشد. بنابراین تابع حذف به‌صورت زیر خواهد بود:

```
void    delete (  Node  *Pper , Node  *p )
{
        if  ( p == NULL ){
             cout<<" List is empty ";
             exit(1);
        }
        Prep → next = p → next
        freenode (p);
}
```

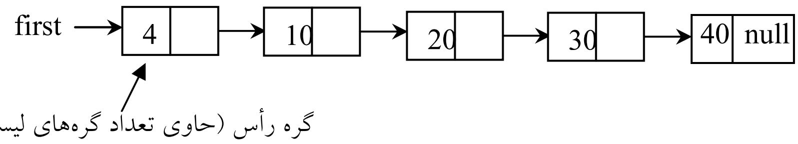
٤-٥ ساختارهای دیگری از لیست پیوندی

لیست پیوندی که، تاکنون 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 دوپیوندی.

١-٤-٥ 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 می‌نامندفیلد اطلاعات این گره معمو لاً برای نگهداری اطلاعات کلی در مورد لیست به‌کار می‌رود. شکل٥-٥ یک لیست پیوندی با گره رأس را نشان می‌دهد که محتویات گره رأس برابر تعداد کل گره‌های

لیست می‌باشد. در چنین ساختاری، عمل درج و حذف مستلزم کار بیشتری است، زیرا اطلاعات موجود در گره رأس باید تغییر کند. اما تعداد گره‌های لیست را می‌توان از گره رأس به‌دست آورد و نیازی به پیمایش لیست نیست.



شکل۵-۵ لیست پیوندی با گره رأس

• 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. در زیر ساختار داده هر گره را ارائه می‌دهیم:

پیاده‌سازی گره رأس

```
struct   Node    {
                   elementtype  info;
                   Node  *next;
                     };
struct  charstr  {
                   int   length;
                   Node  *firstchar;
                  };
charstr   S1,S2;
```

گره رأس حاوی تعداد گره های لیست و اشاره‌گر به ابتدای لیست است.

۵-۴-۲ مزایای لیست با گره رأس و انتهایی

همان‌گونه 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. گـره رأس موجب می‌شود که هر گره لیست دارای یک گره قبلی باشد و در نتیجه بـرای حـذف و درج، لازم نیست دو حالت در نظر گرفته شود.

گره انتهایی گره‌ای است که در آخر لیست پیوندی قرار دارد. هر گره‌ای از لیست دارای یک گره بعدی باشد.

۳-٤-۵ پیچیدگی زمانی عملگرهای لیست پیوندی

عملگرهای مشهور در لیست‌های پیوندی خطی، عملگرهای حـذف و درج مـی‌باشـند، به‌خصوص اگر بخواهیم عناصر خاصی را به لیست مرتب اضافه یا از آن حذف نمـاییم. همانطور که مشاهده کردید عملگر درج در یک لیست فقط شامل چند دستور بـا زمـان ثابت می‌باشند. بنابراین پیچیدگی زمانی تابع از درجه ثابت می‌باشد یعنی از مرتبه O(1) می‌باشد. عملگر حذف نیز مثل عملگر درج از تعدادی دستور با زمان ثابت تشکیل شده است. لذا مرتبه زمانی تابع حذف نیز O(1) خواهد بود.

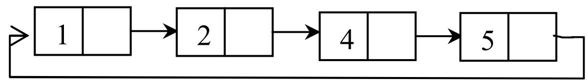
در لیست همانطور که اشاره گردید، عملگر جستجو هم وجـود دارد و همـانطور که ملاحظه کردید جستجو موجود در ایـن سـاختار داده، جـستجوی خطـی مـی‌باشـد. بنابراین زمان آن از مرتبه O(n) خواهد بود.

علاوه از مزایای زیـادی کـه لیـست‌هـای پیونـدی دارنـد معـایبی نیـز دارنـد. در لیست‌های پیوندی یکطرفه وقتی از گرهی به گره دیگر حرکت می‌کنیم امکان بازگـشت به گره قبلی وجود ندارد. بنابراین ایـن یـک مـشکل در لیـست‌هـای پیونـدی یکطرفـه می‌باشدبرای حل این مشکل غالب اً از لیست پیوندی چرخشی استفاده می‌کنند.

٤-٤-۵ لیست‌های پیوندی حلقوی (چرخشی)

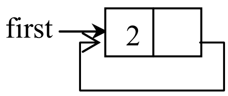
اگرچه لیست پیوندی خطی ساختار داده مفیدی است ولی چندین عیب دارد. در لیـست پیوندی خطی اگر اشاره‌گر p به گرهی از لیست اشاره نماید نمی‌توان به گره‌هـای قبلـی آن دسترسی داشت. باید اشارگر خارجی لیست ذخیره شود تا بتوان مجـدد اً بـه لیـست مراجعه کرد. لیست پیوندی حلقوی مشابه لیست یک طرفه می‌باشد با ایـن تفـاوت کـه فیلد آدرس آخرین گره به جای آن که به NULL اشاره کند، بـه گـره اول لیـست (سـر

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 شکل٦–٥ نمایش داده شده است.

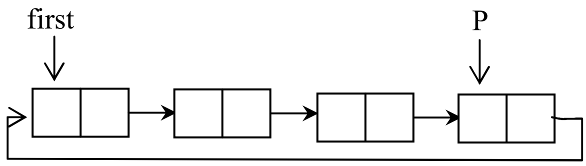


شکل ٦–٥ نمونه‌ای از لیست پیوندی چرخشی

عمل درج در لیست خالی، یک حالت خاص است زیرا در این حالت در لیست یک گره باید به خودش اشاره کند.



الگوریتم درج پس از گره خاصی در لیست پیوندی حلقوی 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. ابتدا و انتهای لیست به صورت قراردادی تعیین می شود ساده‌ترین قرارداد این است که اشاره‌گر خارجی لیست حلقوی به آخرین گره اشاره می‌کند و گره بعد از آن، اولین گره لیست باشد (شکل٧–٥). اگر p یک اشاره‌گر خارجی به لیست حلقوی باشد و $p \rightarrow \inf o$ اطلاعات آخرین گره لیست و $p \rightarrow next$ به اولین گره لیست اشاره می‌کند. این قرارداد موجب می‌شود تا عمل حذف و اضافه به ابتدا یا انتهای لیست به سهولت انجام گیرد.



شکل ٧–٥ نمایش لیست حلقوی

اگر فرض کنیم endp به انتهای لیست حلقوی اشاره کند اضافه کردن گره در ابتدا یا انتهای لیست حلقوی به صورت زیر است:

```
void   AddNode( Node  *endp , elementtype  item )
{
  Newp = getnode () ;
  Newp → info = item ;
  if (endp = = NULL)
   {
          endp = Newp;
          Newp → next = Newp
    }
 else
  {
      Newp → next = endp → next;
      endp → next = Newp;
   }
}
```

first باید همواره به ابتدای لیست اشاره کند و چون گره جدید Newp به اول لیست اضافه شده است:

first = newp ;

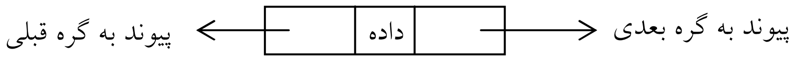
آن را در first قرار می‌دهیم تا دوباره first به اول لیست اشاره کند.

اگرچه مشکل لیست پیوندی یک‌طرفه توسط لیست چرخشی حل شد، ولی راه‌حل ارائه شده مرتبه زمانی بالائی دارد. بنابراین استفاده از لیست پیوندی دوطرفه را مورد بررسی قرار می‌دهیم.

۵-۵ لیست‌های پیوندی دوطرفه (لیست‌های دوپیوندی)

در لیست‌هایی که تاکنون بررسی کردیم می‌توانستیم از گره‌ای به گره بعدی برویم. به این لیست‌ها یک پیوندی نام نهادیم. در بسیاری از کاربردهالازم است به گره قبلیِ گره‌ای دسترسی 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 یک گره لیست دوپیوندی می‌باشد:



در این لیست به کمک اشاره‌گرهای سمت راست (right) و سمت چپ (left) می‌توان در هر دو طرف لیست حرکت کرد. بنابراین با داشتن آدرس یک گره، کلیه گره‌ها قابل دستیابی هستند.

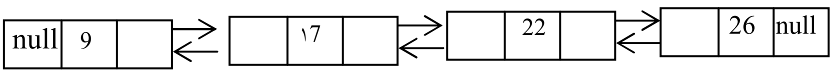
- **ساختار گره‌ها در لیست دوپیوندی**

حال ساختار هر گره در لیست دوپیوندی را ارائه می‌دهیم. هر گره لیست دوپیوندی در نمایش پیوندی به صورت زیر خواهد بود:

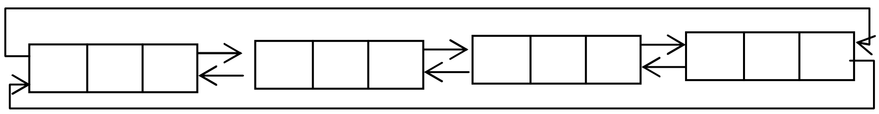
ساختار گره در لیست دوپیوندی

```
struct  Node
        {
          Node * left;
          elementtype  info;
          Node  *  right;
        };
```

لیست‌های دوپیوندی می‌توانند به شکل‌های گوناگونی ارائه شوند. لیست دوپیوندی عادی و دوپیوندی حلقوی نمونه‌هائی از آنها می‌باشد.(شکل ۸–۵)



الف) لیست دوپیوندی عادی



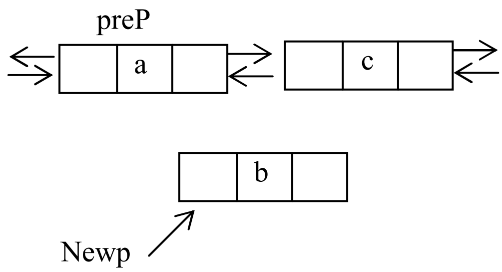
ب) لیست دوپیوندی حلقوی

شکل ۸–۵ انواع لیست‌های دوپیوندی

الگوریتم‌های عملیات اصلی در لیست دوپیوندی، همان‌هایی هستند که در لیست پیوندی مطرح شدند و تفاوت آنها بیشتر در تنظیم پیوندها می‌باشد. به‌عنوان مثال، درج گره جدید در لیست دوپیوندی شامل تنظیم پیوندهای رو به جلو و رو به عقب است 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.

همان‌طور که از لیست پیوندی چرخشی می‌دانید لیست دوپیوندی چرخشی، لیستی است که، انتهای لیست به ابتدای لیست و ابتدای لیست به انتهای لیست اشاره می‌کند.

با استفاده از لیست شکل۹-۵ الگوریتم درج گره‌ای پس از گره‌ای با محتویات a به‌صورت زیر خواهد بود:



شکل ۹-۵ درج گره جدید در لیست پیوندی دوطرفه

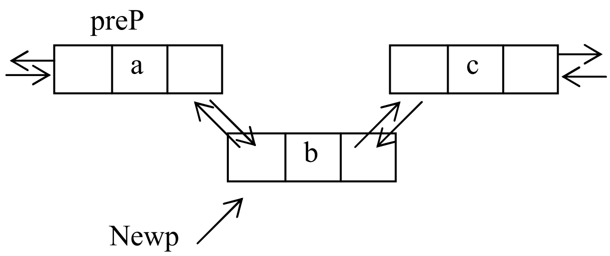
الگوریتم به‌صورت زیر خواهد بود:

۱. با استفاده از تابع getnode() یک گره جدید ایجاد می‌کنیم.
۲. مقدار جدید را در قسمت info گره جدید قرار می‌دهیم.
۳. آدرس گره a را در فیلد چپ گره جدید جایگزاری می‌کنیم.
٤. آدرس فیلد راست گره a را در فیلد راست گره جدید قرار می‌دهیم.
٥. آدرس گره جدید b را در فیلد چپ گره بعد از a جایگزاری می‌کنیم.
٦. آدرس گره جدید b را را در فیلد راست گره a قرار می‌دهیم.

در این‌صورت گره جدید در لیست درج می‌شود. در زیر قطعه کد مربوط به عملگر insert ارائه می‌شود:

```
void  Insert( Node  *perP , elementtype b )
{
   Node    *Newp ;
   Newp = getnode() ;
   if ( IS_FULL(Newp) ){
         cout<<" The memory is full"; exit(1);
   }
   Newp → info = b;
   Newp → left = preP;
   Newp → right = preP → right;
   perP → right → left = Newp;
   perP → right = Newp;
}
```

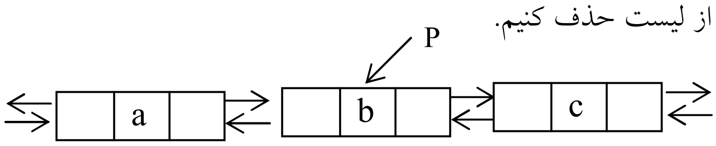
شکل ۱۰–۵ نتیجه درج عنصر جدید b در لیست می‌باشد:



شکل ۱۰–۵ لیست پیوندی دوطرفه حاصل

حال عملگر حذف از لیست دو پیوندی را مورد بررسی قرار می‌دهیم:

برای حذف گره از لیست دوپیوندی، پیوند رو به جلوی گره قبل و پیوند رو به عقب بعد از آن باید طوری تنظیم شوند که به این گره اشاره نکنند. همانطور که در شکل ۱۱–۵ مشاهده می‌کنید می‌خواهیم گره با مقدار b که اشاره‌گر p به آن اشاره می‌کند را از لیست حذف کنیم.



شکل ۱۱–۵ لیست پیوندی دوطرفه

الگوریتم حذف در حالت کلی به صورت زیر می‌باشد :

۱. در فیلد چپ گره بعد از گره b آدرس گره چپ گره b را قرار می‌دهیم.

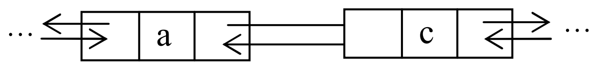
۲. در فیلد راست گره قبل از گره b آدرس گره راست گره b را قرار می‌دهیم.

۳. مقدار حافظه اختصاص داده شده به p را آزاد می‌کنیم.

حال قطعه کد مربوط به حذف از لیست دوپیوندی را به‌صورت زیر ارائه می‌دهیم:

```
void    Delete(Node *p )
{
   p → right → left = p → left;
   p → left → right = p → right;
   freenode (p);
}
```

بعد از حذف گره لیست زیر حاصل می‌شود:



شکل ۱۲-۵ لیست پیوندی دوطرفه حاصل

۱-۵-۵ پیچیدگی زمانی عملگرهای لیست دوپیوندی

عملگرهای مشهور در این ساختار داده، همان عملگر حذف و درج می‌باشد. برای درج یک عنصر در محل مخصوص به خود همانطور که مشاهده کردید تعداد محدودی دستور که هر کدام در یک زمان ثابت اجرا می‌شوند، تشکیل شده است. بنابراین مرتبه زمانی آن O(1) می‌باشد.

تابع حذف نیز همانطور که ملاحظه کردید مشابه تابع درج از تعدادی دستور با زمان ثابت تشکیل شده است. لذا پیچیدگی زمانی آن O(1) خواهد بود. عملگر جستجو در لیست دو پیوندی به‌صورت خطی عمل می‌کند. بنابراین مرتبه زمانی آن O(n) می‌باشد.

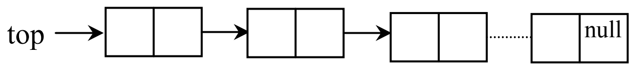
٦-٥ پیاده‌سازی پشته با لیست پیوندی

در اینجا قصد داریم پشته را با استفاده از لیست‌های پیوندی پیاده‌سازی کنیم. با توجه

اینکه، یکی از معایب پیاده‌سازی پشته و صف با استفاده از آرایه این است که اندازه ثابت آرایه، اندازه پشته و صف را محدود می‌کند. استفاده از لیست پیوندی به جای آرایه جهت پیاده‌سازی پشته یا صف، محدودیت آرایه‌ها برای پیاده‌سازی را از بین می‌برد. و باعث می‌شود پیاده‌سازی پشته و صف بدون محدودیت انجام گیرد.

نکته: یکی از معایب پیاده‌سازی پشته و صف با استفاده از آرایه این است که اندازه ثابت آرایه، اندازه پشته و صف را محدود می‌کند.

همان‌طور که اشاره کردیم، پشته لیستی از عناصر است که فقط از یک طرف به عناصر آن می‌توان دسترسی پیدا کرد(از بالای پشته). بنابراین عمل افزودن یک عنصر به ابتدای لیست پیوندی به مانند این است که عنصری به بالای پشته افزوده شده است. در هر مورد، عنصر جدید طوری اضافه می‌شود که تنها عنصری باشد که بالاترین اولویت را برای pop کردن را دارا می‌باشد. پشته فقط از طریق عنصر بالای آن (top) و لیست پیوندی فقط از طریق اشاره‌گری که به ابتدای لیست اشاره می‌کند، قابل دسترسی است. به‌طور مشابه عمل حذف اولین عنصر لیست پیوندی همانند حذف عنصر بالای پشته است. در هر دو مورد، فقط عنصی که فور اً قابل دسترسی است از مجموعه حذف می شود. شکل ۱۳–۵ یک پشته پیوندی را نمایش می‌دهد.



شکل ۱۳–۵ 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.

پیاده‌سازی پشته با استفاده از لیست پیوندی

ساختار گره:
`struct Node {`
`        elementtype info;`
`        struct Node * next;`
`        }s;`
`Node * top;`

پس از تعریف گره، اشاره‌گر top را برابر با تهی قرار می‌دهیم:

```
top = NULL;
```

عمل تست خالی بودن پشته

تست می‌کند که آیا اشاره‌گر top تهی است یا خیر. به عبارت دیگر تست می‌کند که آیا عضوی برای حذف کردن وجود دارد یا خیر.

```
if (top = = NULL)
        cout<<" stack is empty";// printf(" stack is empty");
```

عمل push به پشته

با استفاده از دستورات زیر گره‌ای به لیست پیوندی اضافه می‌شود. این لیست در حقیقت پشته را پیاده‌سازی می‌کند

```
void    push(Node *top  , elementtype  item )
{
    ptr = getnode();
    if ( IS_FULL(ptr) ) {
         cout<<" The memory is full " ;
         exit(1) ;
    }
    ptr → info = item;
    ptr → next = top;
    top = ptr ;
}
```

عمل PoP

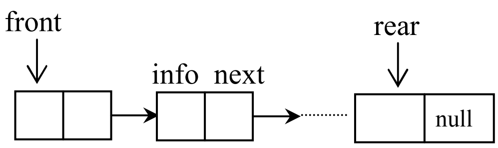
طبق تعریف اولین گره لیست (گره‌ای که top به آن اشاره می‌کند) را باید حذف کنیم. برای اینکار به‌صورت زیر عمل می‌کنیم:

```
void    pop(Node *top )
{
  Node  *ptr;
  ptr = top;
  top = top → next;
  freenode (ptr);
}
```

پشته را می‌توان با استفاده از لیست پیوندی حلقوی نیز پیاده‌سازی نمود
پیاده‌سازی به‌عنوان تمرین به عهده دانشجو گذارده می‌شود.

۷-۵ پیاده‌سازی صف با لیست پیوندی

پیاده سازی صف با لیست پیوندی مشابه پیاده‌سازی پیوندی پشته است. در صف پیوندی، اولین گره را جلوی صف(سر صف) front در نظر می‌گیریم. به این ترتیب محل حذف گره‌ای از صف پیوندی مشابه حذف گره‌ای از پشته پیوندی پیاده‌سازی می‌شود. ولی قرار دادن گره‌ای در صف پیوندی مستلزم پیمایش لیست و یافتن آخرین گره آن است. برای جلوگیری از پیمایش لیست جهت یافتن آخرین عنصر، از یک اشاره‌گر دیگری به نام rear استفاده می‌کنیم که به انتهای صف پیوندی اشاره می‌کند.
شکل ۱۴-۵ نمایش پیوندی صف را نمایش می‌دهد.



شکل ۱۴-۵ نمایش پیوندی صف

ساختار گره صف پیوندی مانند لیست پیوندی است. به دو اشاره‌گر خارجی بنام‌های front برای نشان دادن ابتدای صف پیوندی و دیگری به‌نام rear برای نشان دادن انتهای صف نیاز داریم:

پیاده‌سازی صف با استفاده از لیست پیوندی

```
struct queue {
                    elementtype info;
                    queue *next;
                    };
queue  *front, *rear;
```

پس از تعریف گره، اشاره‌گرهای زیر را برابر با تهی قرار می‌دهیم:

```
front= rear= NULL;
```

عمل تست خالی بودن صف

تست می‌کند که آیا اشاره‌گر front تهی است یا خیر؟

```
if (front == NULL)
          cout<<"queue is empty";// 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 پیوندی اضافه می‌کند:

```
void    Add(Node *rear , elementtype item )
{
    ptr = getnode();
    if ( IS_FULL(ptr) ) {
         cout<<" The memory is full " ;
         exit(1) ;
    }
    ptr → info = item;
    ptr → next= NULL;
    rear → next = ptr;
    rear= ptr;
}
```

عمل حذف گره از صف پیوندی

همانطور که از قبل می‌دانید حذف از صف، از سر صف امکانپذیر می‌باشد. اولـین گـره صف را (سر صف) حذف می‌کنیم. دستورات زیر این کار را انجام می‌دهد:

```
void    Delete(Node *front )
{
  Node  *ptr;
  ptr = front;
  front = front → next;
  freenode (ptr);
}
```

۸-۵ معایب پیاده‌سازی صف و پشته از طریق لیست‌های پیوندی

هرچند استفاده از لیست پیوندی برای پیاده‌سازی صـف و پـشته محـدودیت‌هـایی را از بین می‌برد ولی معایبی نیز دارد. در زیر به آنها اشاره می‌کنیم:

۱. یک گره در لیست پیوندی نسبت به عنصر متناظر خود در آرایه حافظه بیـشتر را اشغال می‌کند، چون در لیست پیوندی علاوه بر قسمت اطلاعـات بـه قـسمت آدرس نیز نیاز است.

۲. مدیریت لیست پیوندی مستلزم صرف وقت است. هر عمل افـزودن و حـذف یک عنصر از پشته یا صف مستلزم حذف و اضافه به لیست پیوندی است.

مزیت نمایش پشته و صف به‌صورت لیـست پیونـدی ایـن اسـت کـه همگـی از گره‌های موجود در یک لیست آماده استفاده می‌کنند گـره‌ای کـه توسـط پـشته اسـتفاده نشده باشد توسط پشته دیگری قابل استفاده است. به‌طوری‌که تعداد گـره‌هـای در حـال استفاده در یک زمان خاص از حداکثر تعداد گره‌های قابل استفاده بیشتر نیستند.

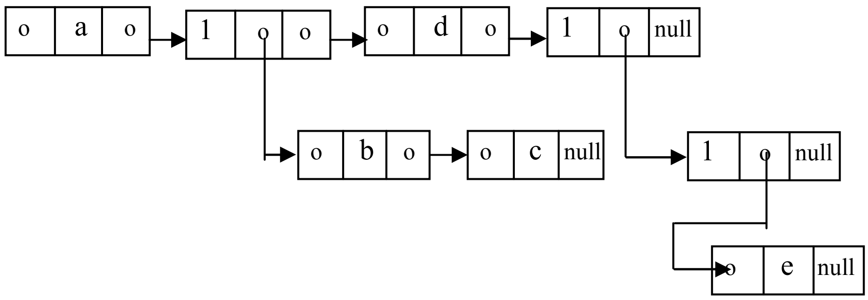
۹-۵ لیست‌های عمومی

لیست عمومی، لیست پیوندی است که هر فیلد گـره‌هـای آن مـی‌توانـد برحـسب نیـاز اشاره‌گر باشد. ساختار هر گره این نوع لیست به صورت زیر است:

tag	info/Dlink	Link

که در آن اگر tag برابر صفر یا false باشد نشان‌دهنده این است که فیلـه Info دارای ارزش می‌باشد. یعنی در فیلد وسط فقط داده قرار می‌گیرد و اگـر tag برابـر بـا یـک یـا True باشد نشان‌دهندهٔ این است که فیلد وسط DLink دارای ارزش اسـت بـه عبـارت دیگر فیلد وسط خود اشاره‌گر است و به یک زیر لیست دیگر اشاره می‌کند.

شکل ۱۵–۵ نمایشی از یک لیست عمومی را ارائه می‌دهد.



شکل ۱۵–۵ نمایش لیست عمومی

لیست شکل ۱۵–۵ را می‌توان به فرم پرانتزی نیز نمایش داد:

GList= (a, (b,c), d, ((e)))

غالباً لیست‌های عمومی برای نمایش درخت‌ها (که در فصل بعـد بحـث خواهـد شد) به‌کار برده می‌شوند.

برای پیمایش لیست پیوندی می‌توان به‌صورت ذیل عمل کرد:

```
void     traversal (Node    *GList)
{
        if (GList != NULL)
        {
                if (GList → tag== 0)
                        cout << GList → Info;
                else  traversal (GList → Info);
                traversal (GList → Link);
        }
}
```

۱۰–۵ نمایش چند جمله‌ای‌ها به‌صورت لیست‌های پیوندی

می‌خواهیم مسائلی را مورد بررسی قرار دهیم که استفاده از لیست‌های پیوندی برای پیاده‌سازی آنها امری منطقی باشد. برای نمونه مثالی از قبیل چندجمله‌ای‌ها، ماتریس‌های اسپارس و غیره مسائلی هستند که می‌توان آنها را با لیست‌های پیوندی نمایش داد و عملگرهای مختلف از جمله جمع، ضرب و غیره را روی آنها اعمال کرد.

حال نمایش چند جمله‌ای‌ها را با لیست‌های پیوندی مورد بررسی قرار می‌دهیم.
در حالت کلی چندجمله‌ای زیر را در نظر بگیرید:

$$P(x) = a_{n-1}x^{e_{n-1}} + \ldots + a_0 x^{e_0}$$

که در آن فرض می‌کنیم $e_{n-1} > e_{n-2} > \ldots > e_0 \geq 0$ باشد. برای نمایش این چندجمله‌ای با لیست پیوندی هر جمله این چندجمله‌ای را با یک گره لیست پیوندی نمایش می‌دهیم. هر جمله از یک ضریب و یک توان تشکیل شده است. بنابراین هر گره لیست پیوندی دارای ساختار زیر خواهد بود:

Coef	exp	next

حال در حالت کلی ساختار هر گره لیست پیوندی را به‌صورت زیر پیاده‌سازی می‌کنیم:

پیاده‌سازی هر گره لیست

```
struct Node {
        int coef;
        int exp;
        Node * next;
    };
```

برای مثال p(x) را به‌صورت زیر در نظر بگیرید:

$p(x) = 5x^{10} + 3x^{6} + 12$

لیست پیوندی حاصل برای پیاده‌سازی چندجمله‌ای بالا به‌صورت ذیل می‌باشد:

5	10	→	3	6	→	12	0	Null

حال با توجه به نمایش چندجمله‌ای به‌صورت لیست پیوندی می‌توان به راحتی مجموع و حاصلضرب دو چند جمله‌ای را محاسبه کرد (به‌عنوان تمرین به خواننده واگذار شود).

نمایش بالا را می‌توان برای پیاده‌سازی ماتریس‌های اسپارس نیز به‌کار برد.

۵-۱۱ مثال‌های حل شده

در اینجا قصد داریم با حل تعدادی مثال مفاهیم لیست‌های پیوندی به‌صورت کامل ارائه شود:

مثال ۵-۱: قطعه کدی بنویسید که یک گره را به لیست پیوندی اضافه کند.

```
j= getnode();
j → inf o = item;
if ( head == NULL)
  {
     head=j;
     j → next = NULL;
}
else
  {
  j → next = list → next;
  list → next = j;
}
```

مثال ۵-۲: تابعی بنویسید که اشاره‌گر و لیست پیوندی را بگیرد و تعداد گره‌های لیست را برگرداند.

```
int count (Node  *ptr )
{
   Node   *list ;
   int  c=0;
   list = ptr;
   if( list ==NULL)
       return 0;
   while(list)
```

```
   {
      list = list → next;
      c ++;
   }
   return c;
}
```

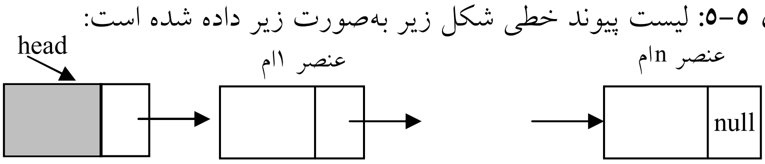
مثال ۳–۵: تابعی بنویسید که داده و لیست پیوندی را از آخر به اول چاپ کند.

```
void   print_reversal( Node  *list )
{
      if (list!= NULL)
       {
          print_reversal( list → next ) ;
          cout<< list → info ; // printf("%f", list → info );
       }
}
```

مثال ٤–۵: تابعی به نام Intersect بنویسید که دو لیست پیوندی را به‌عنوان پارامتر دریافت کرده و از داده‌های مشترک آن دو لیست، لیست سومی ساخته و اشاره‌گر اول لیست سوم را برگرداند.

```
Node  *Intersect( Node  *L1 , Node  * L2 )
{
   Node  *T1,*T2,*L3 ;
   for(T1=L1 ; T1!=NULL ;  T1 = T1 → next )
      for(T2=L2 ; T2!=NULL ;  T2 = T2 → next )
         if(  T1 → info == T2 → info  )
         {
            Add_Node(L3 ,  T1 → info );
            Break;
         }
    return  L3 ;
}
```

مثال ۵–۵: لیست پیوند خطی شکل زیر به‌صورت زیر داده شده است:



بر روی این لیست قطعه کد زیر اجرا می‌شود (head به سر لیست اشاره می‌کند):

```
List=head→ next;

While (List!=NULL)
{
	P=List;
	While (P!=NULL)
	{
		P=P→next;
		cout <<" Data " ;
	}
	List=List→next;
}
```

اگر لیست n عنصر داشته باشد کلمه Data چند بار در خروجی چاپ می‌شود.

حل: حلقه دوم در مرحله اول از عنصر اول لیست با استفاده از اشاره‌گر p کل لیست را پیمایش می‌کند. در مرحله دوم از عنصر دوم به بعد این عمل را انجام می‌دهد. و در مرحله آخر یک عنصر لیست که آخرین عنصر می‌باشد پیمایش می‌شود.

بنابراین تعداد اجراهای دستور چاپ برابر خواهد بود با:

$$\text{تعداد اجراها} = n + (n-1) + \cdots + 1 = \frac{n(n+1)}{2}$$

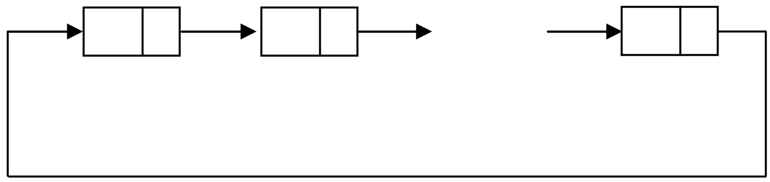
مثال ٦-٥: تابعی بنویسید که لیست L را دریافت کرده، معکوس لیست پیوندی خطی را به‌دست آورد.

حل: تابع را به‌صورت زیر می‌نویسیم:

```
Node	*reverse ( Node *L )
{
	Node	*m,*t;
	m=NULL;
	While (L){
		t=m; m=L;
		L=L→next;
		m→next=t;
	}
	return m ;
}
```

مثال ۷-۵: می‌خواهیم تغییراتی در یک لیست پیوندی اعمال کنیم که عمل افزودن ابتـدا یا انتهای لیست با عملیاتی از مرتبه O(1) قابل انجام باشد. به نظر شما از چه نوع لیستی برای این کار استفاده کنیم.

حل: برای حل این مسئله می‌بایست از یک لیست حلقوی استفاده کنیم:



آدرس آخرین گره را می‌بایست ذخیره کنیم. با داشتن آدرس آخرین گره می‌توان اعمال خواسته شده را در زمانی با مرتبه O(1) انجام داد.

۱۲-۵ تمرین‌های فصل

۱. توابعی بنویسید که دو گره mام و nام یک لیست پیوندی را با هم عوض کند.

۲. الگوریتم‌هایی برای اعمال زیر بنویسید:

الف) افزودن عنصر در انتهای لیست پیوند خطی.

ب) الحاق دو لیست پیوندی.

ج) ترکیب دو لیست **پیوندی** مرتب در یک لیست **پیوندی** مرتب دیگر.

د) مجموع عناصر لیست **پیوندی** صحیح.

ه) محاسبه تعداد عناصر لیست **پیوندی.**

و) کپی از لیست **پیوندی.**

۳. تعداد گره‌هایی که در جستجو برای یک عنصر خاص در یک لیست نامرتب، لیست مرتب و یک آرایه نامرتب و یک آرایه مرتب مورد بررسی قرار می‌گیرند، چقدر است؟

٤. فرض کنید List یک لیست پیوندی در حافظه و شامل مقادیر عددی باشد. برای هر یک از حالت‌های زیر یک تابع بنویسید که:

الف) ماکزیمم مقادیر لیست **پیوندی** را پیدا کنید.

ب) مینیمم مقادیر لیست **پیوندی** را پیدا کنید.

ج) میانگین مقادیر لیست **پیوندی** را پیدا کنید.

د) حاصلضرب عناصر لیست **پیوندی** را پیدا کنید.

۵. تابعی بنویسید که بدون هیچ تغییری در مقدارهای info لیست **پیوندی** را مرتب کند.

٦. پشته‌ای را به کمک لیست پیوندی پیاده‌سازی کرده و سپس به کمک آن یک عبارت postfix را محاسبه و چاپ کنید.

۷. تابعی بنویسید که دو چندجمله‌ای یک متغیره را در لیست پیوندی ذخیره کرده و سپس جمع آنها را محاسبه کرده و چاپ کند.

۸. نشان دهید چگونه می‌توان از لیست پیوندی برای هم ارزی‌ها استفاده کرد.

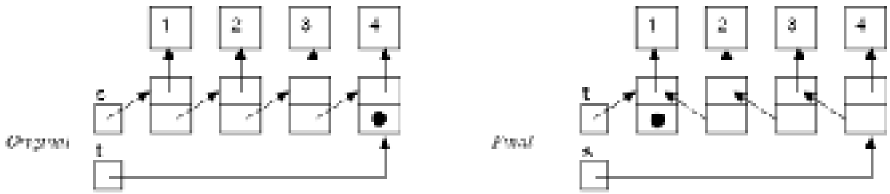
۹. تابعی بنویسید که دو لیست حلقوی را به هم الحاق کند.

۱۰. تابعی بنویسید که دو لیست حلقوی دو طرفه را به هم الحاق کند.

۱۱. تابعی بنویسید که مقادیر ماکزیمم، مینیمم و متوسط یک لیست پیوندی را پیدا کند.

۱۲. تابعی بنویسید که ۱۰۰ عدد تصادفی را خوانده و در لیست قرار دهد. و سپس با استفاده از تابع تمرین ۱۱ مقادیر ماکزیمم، مینیمم و متوسط آن را پیدا کند.

۱۳. با توجه به شکل سمت چپ، مجموعه‌ای از دستورات ارائه کنید تا به شکل سمت راست برسیم. در این شکل s و t از نوع اشاره‌گر به نود هستند.



۱٤. تابعی بنویسید که دو لیست پیوندی مرتب را دریافت کرده آن ها را طوری ادغام کند که حاصل نیز مرتب باشد.

۱۵. تابعی بنویسید که اعداد صحیح بزرگ را به صورت رشته‌ای خوانده و در یک لیست پیوندی قرار دهد. سپس حاصل جمع دو عدد بزرگ را به دست آورد.

۱٦. دو ماتریس اسپارس B, A را در نظر بگیرید. تابعی بنویسید که دو ماتریس را به وسیله لیست‌های پیوندی پیاده‌سازی نماید

۱۷. با در نظر گرفتن خروجی تمرین ۱٦ تابعی بنویسید که جمع دو ماتریس اسپارس را محاسبه نماید.

۱۸. تابعی بنویسید که مشخصات دانشجویان را از ورودی دریافت کرده و در یک لیست حلقوی به طرز مرتب براساس شماره دانشجویی قرار دهد.

۱۹. تابعی بنویسید که لیست پیوندی را دریافت کرده و سپس دو گره پشت سر هم را حذف نماید.

۱۳-۵ پروژه‌های برنامه‌نویسی

۱. اطلاعات زیر مربوط به یک دانشجو را در نظر بگیرید:

Id	شماره دانشجویی
Name	نام
Family	نام خانوادگی
Major	رشته

Year سال ورود

Semester نیمسال

سپس اطلاعات مربوط به هر نیمسال را به‌صورت زیر در نظر بگیرید:

Couse Name اسم درس

Code کد درس

Test date تاریخ امتحان

Date تاریخ تشکیل کلاس

حال با استفاده از لیست‌ها، لیست چندگانه‌ای طراحی کنید به‌طوری‌که بتوان به‌راحتی اعمال زیر را انجام داد:

- ثبت نام دانشجو
- انتخاب واحد دانشجو
- حذف و اضافه یکدرس به دانشجو
- حذف ترم دانشجو
- حذف یک دانشجو از لیست
- و غیره

۲. می‌خواهیم که یک لیست پیوندی کامل برای انجام اعمال حسابی بر روی ماتریس‌های اسپارس با استفاده از نمایش لیست پیوندی ارائه دهیم. سپس اعمال زیر را روی این لیست پیاده‌سازی کنیم:

۱. تشکیل گره‌های لیست با استفاده از ماتریس اسپارس

۲. دو ماتریس اسپارس را با هم جمع کنید.

۳. تفاضل دو ماتریس اسپارس را محاسبه کنید.

٤. دو ماتریس اسپارس را در هم ضرب کنید.

فصل ششم

درختان (trees)

در پایان این فصل شما باید بتوانید:

- ✓ درخت را تعریف کرده و تمام مفاهیم و تعاریف موجود در درخت را بیان کند.
- ✓ داده‌های خود را در قالب درخت نمایش دهید.
- ✓ درخت را در قالب‌های متفاوت ذخیره کرده و معایب و امتیازات هر یک را بیان کنید.
- ✓ انواع درخت‌های دودوئی را باهم مقایسه کنید.
- ✓ اطلاعات موجود در درخت را به روش‌های گوناگون پردازش کنید.
- ✓ انواع درخت‌ها و کاربردهای آنها را تشریح کنید؟

سؤال‌های 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 داده‌های موجود را کنار بگذاریم چه نوع ساختاری می‌توانیم تعریف کنیم.

۳. در حافظه‌های جانبی کامپیوتر اطلاعات را معمولا به چه صورتی ذخیره مي‌کنند

مقدمه

تا اینجا، انواع مختلف ساختمان داده‌های خطی از قبیل رشته‌ها، آرایه‌هـا، لیـستها، پـشته‌هـا و صف‌ها مورد مطالعه و بررسی کامل قرار گرفته است. این فصل یک ساختار داده غیرخطی موسوم به درخت را تعریف می‌کنداین ساختمان اساساً برای نمایش داده‌هـایی کـه شـامل رابطه سلسله مراتبی بین عناصر آنهـا وجـود دارد، بـه‌کـار مـی‌رود. رکوردهـا، درخـت‌هـای خانوادگی و جدول فهرست مطالب کتاب نمونه‌هائی از رابطه سلسله مراتبی می‌باشد.

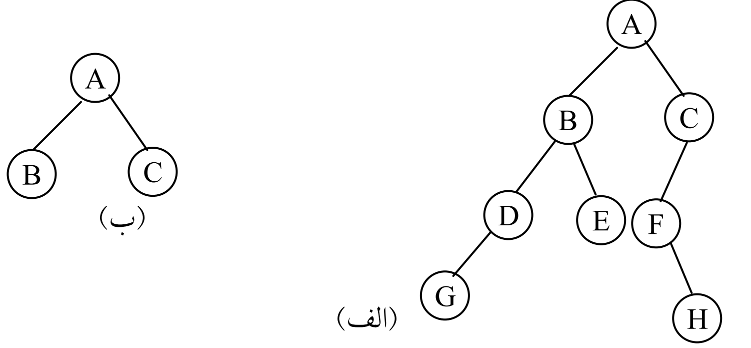
تعریف: درخت مجموعه محدودی از یک یا چند گره به‌صورت زیر می‌باشد:

- دارای گره خاصی بنام ریشه(Root) است.
- بقیه گره‌ها به مجموعه‌های مجزای $T_1,...,T_n$ تقسیم شده که هـر یـک از ایـن مجموعه‌ها خود یک درخت هستند. $T_1,...,T_n$زیردرخت‌های ریشه نامیده می‌شوند.

لازم به ذکر است که بدانید، درخت فاقد دور است. بعبارت دیگر، در درخت بین هر دو گره فقط یک میسر وجود دارد.

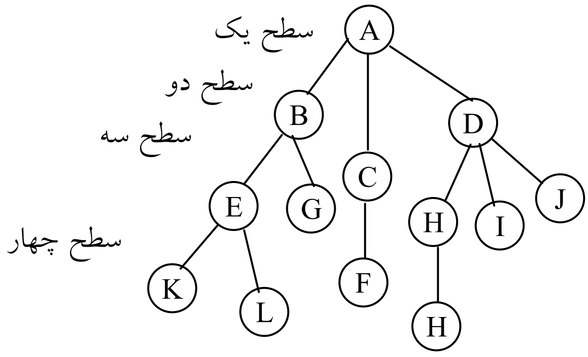
حال برای آشنائی چند درخت که اصطلاحاً درخت کامپیوتری نامیـده مـی‌شـود، ارائه می‌دهیم. شکل ۱−٦ را مشاهده کنید. برخلاف درختان طبیعی که ریشه‌هـای آن‌هـا در پایین و برگ‌ها در بالا قرار دارند. در درخت‌های کامپیوتری، ریشه در بالا و برگ‌هـا در پایین قرار دارند.

به‌طور کلی درخت‌هـا بـر دو دسـته تقـسیم مـی‌شـوند. درخـت‌هـای عمـومی و درخت‌های دودوئی. درخـت دودوئـی (binary tree) درختـی اسـت کـه هـر گـره آن حداکثر دو پیوند(فرزند) داشته باشد. درختی که دودوئی نباشد، درختی عمـومی اسـت. درخت‌های (الف) و (ب) در شکل ٦,۱ درخت‌های دودوئی هـستند و درخـت (ج) در شکل مذکور درخت عمومی می‌باشد.



(الف) (ب)

سطح درخت



(ج): درخت عمومی

شکل ۱–۶ انواع درخت‌ها

اصطلاحات زیادی در ارتباط با درخت‌ها به کار برده می‌شود که بایستی آنها را تعریف کرد.

۱–۶ اصطلاحات مربوط به درخت‌ها

در اینجا قصد داریم مفاهیم مربوط به درخت مورد بحث و بررسی قرار دهیم تا بتوانیم درک صحیحی از درخت داشته باشیم.

• **گره(node):** به عناصر موجود در درخت گره گویند. درخت شکل ۶,۱ (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.

• **درجه گره(degree):** درجه گره برابر با تعداد فرزندان آن گره است. یا تعداد زیردرخت‌های یک گره درجه آن گره نامیده می شود. در شکل ۶,۱ (ج) درجه گره A برابر با ۳،، درجه گره C برابر با ۱ و درجه گره F برابر صفر است.

• **برگ(leaf):** 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 یا غیربرگ هستند.

• **درجه درخت:** درجه یک درخت حداکثر درجه گره‌های آن درخت می‌باشد. برای مثال در درخت (ج) حداکثر درجه گره ۳ می‌باشد پس درجه درخت ۳ است.

• **گره‌های همزاد یا هم‌نیا(sibling):** گره‌هایی که دارای پدر(والد) مشترک دارند. **گره‌های همزاد** نامیده می‌شوند. برای مثال، گره B والد گره‌های F , E بوده و برعکس F , E فرزندان B می‌باشند. فرزندان یک گره، گره‌های همزاد یا هم نیا نامیده می‌شوند. همچنین، گره‌های H , I , J همزادند.

• **سطح درخت(level):** هر گره موجود در درخت، 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 ۱–٦ (ج) در کنار آن نوشته شده است.

• **عمق یا ارتفاع درخت(depth):** بزرگ‌ترین سطح برگ‌های درخت را عمق درخت گویند. در شکل ۱–٦ (ج) عمق درخت برابر با ٤ است.

• **تعریف یال و مسیر:** خطی که از گره N به یک گره بعدی رسم می‌شود یک یال و دنباله‌ای از یال‌های متوالی یک مسیر نامیده می‌شود.

• **طول مسیر درخت:** طول مسیر درخت، مجموع طول‌های تمام مسیرها از ریشه به تمام گره های درخت است. بطور مثال برای درخت (ج) شکل ۱–٦ داریم:

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)

• **درخت kتائی:** درختی که تعداد فرزندان هر گره در آن حداکثر k باشد.

• **درخت متوازن:** درختی که اختلاف سطح برگ‌های آن حداکثر یک باشد درخت را متوازن می‌نامند واگر این اختلاف صفر باشد، آنگاه درخت را کاملاً متوازن می‌گویند.

فرض کنید T یک درخت باشد. علاوه بر نمایش ارائه شده در شکل ۱–٦ روشهای 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 ۱-۶ (ج) را می‌توان به‌صورت زیر هم نمایش داد. به نحوی که هر زیر درخت خود یک لیست می‌باشد.

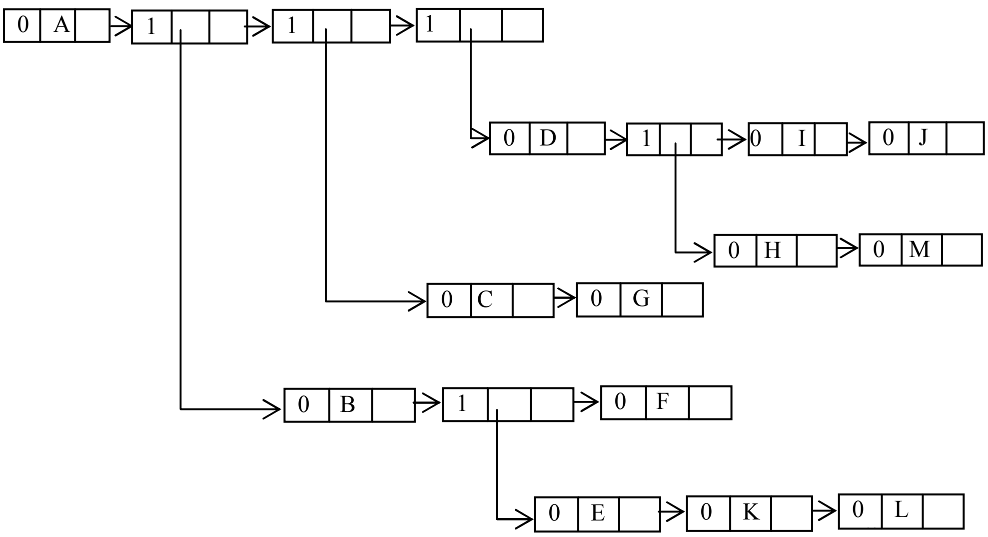
(A(B(E(K,L),F),C(G),D(H(M),I,J)))

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، استفاده از یک لیست پیوندی می‌باشد. در این صورت بایستی یک گره و تعدادی متغیر فیلد، بسته به تعداد انشعاب‌ها تعریف کنیم. در این گره هر فرزندی که برگ نباشد با اضافه کردن یک سطح به لیست به‌صورت یک زیرلیست نمایش داده می‌شود.

برای نمایش درخت شکل ۱-۶ (ج) به‌صورت یک لیست پیوندی داریم:

استفاده از لیست پیوندی، فرم پرانتزی و شکل درختی، سه روش نمایش درخت‌ها می‌باشند.



شکل ۲-۶ نمایش پیوندی درخت شکل ۱-۶ (ج)

۲-۶ درخت دودوئی (binary tree)

یک درخت دودوئی T به‌صورت مجموعه‌ای متناهی از عناصر بنام گره‌ها تعریف

می‌شود. به‌طوری‌که:

الف) اگر T خالی باشد، به آن درخت پوچ یا تهی می‌گویند یا

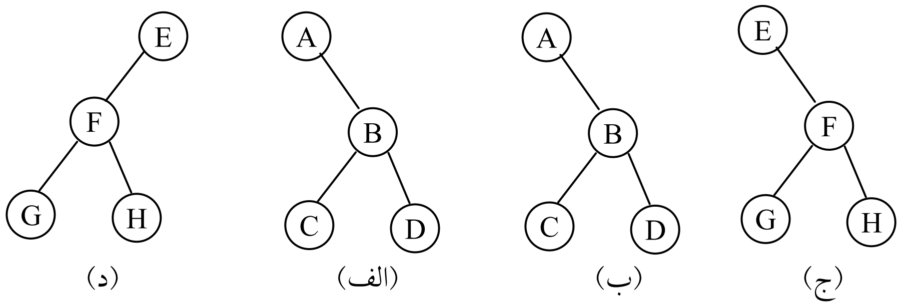
ب)حاوی مجموعه‌ای محدود از گره‌ها، یک ریشه و دو زیردرخت T_2, T_1 می‌باشد که به ترتیب به آنها زیردرخت‌های چپ و راست گفته می‌شود.

با توجه به تعریف درخت دودوئی متوجه می‌شویم که درخت دودوئی، درختی است که، هر گره آن حداکثر دو فرزند 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 آنها یکسان باشد.

مثال ١-٦: چهار درخت دودوئی شکل ٣-٦ 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،

همانگونه که گفتیم دو درخت (ج) و (د) دو درخت دودوئی متفاوتی هستند ولی اگر این دو درخت، عمومی فرض شوند، یکسان هستند. چون در درخت‌های عمومی ترتیب زیردرختان مهم نیست.



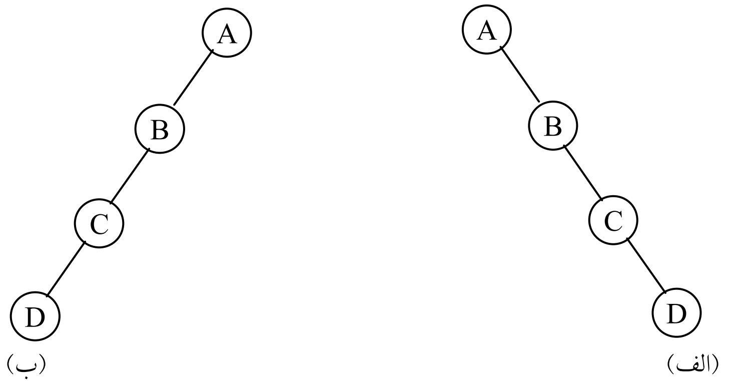
شکل ٣-٦ چهار درخت دودوئی

٣-٦ انواع درخت‌های دودوئی

در این بخش به تعریف انواع درخت‌های دودوئی می‌پردازیم.

• **درخت مورب:** یک درخت مورب به چپ می‌باشد هرگاه، هر گره، فرزند چپ پدر خود باشد و یک درخت مورب به راست می‌باشد، هرگاه، هرگره، فرزند

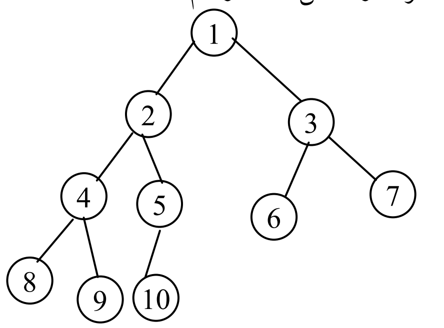
راست پدر خود باشد. شکل ٤-٦ (الف) و (ب) دو درخت مورب به راست و چپ را نمایش می‌دهند.



شکل ٤-٦ دو درخت مورب به چپ و راست

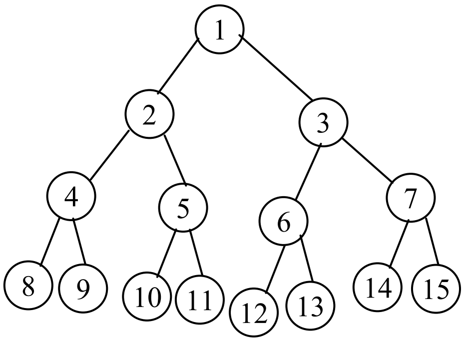
• **درخت دودوئی کامل:** درخت دودوئی است که هر گره می‌تواند دو فرزند داشته باشد و حداکثر اختلاف بین سطح گره‌های آن یک باشد. همچنین درخت از چپ به راست پر شود.

درخت کامل T_{10} با ۱۰ گره در شکل ٥-٦ رسم شده است.



شکل ٥-٦ درخت کامل با ده گره

• **درخت پر**: درختی دودوئی T را درخت پر گویند هرگاه همه گره‌های آن به جزء گره‌های سطح آخر دقیقاً دو فرزند داشته باشند. شکل ٦-٦ یک درخت دودوئی پر با ٤ سطح را نمایش می‌دهد.



شکل ٦-٦ درخت پر با چهار سطح

٤-٦ خواص درخت‌های دودوئی

قبل از اینکه به چگونگی نمایش درخت‌های دودوئی بپردازیم در این بخش به ارائه برخی از خواص درخت دودوئی می‌پردازیم.از قبیل 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 موضوع فوق را به‌صورت چند اصل موضوعی ارائه می‌کنیم

• اصل موضوعی (۱)

اگر T یک درخت دودوئی کامل با n گره باشد، بطوریکه گره‌های آن با اندیس I و $1 < i \leq n$ اندیس‌گذاری شده است (از چپ به راست) آنگاه:

۱. اگر $i \neq 1$ باشد آنگاه پدر i در [i/2] است. اگر i=1، i ریشه است و پدری نخواهد داشت.

۲. اگر $2i > n$ باشد آنگاه فرزند چپ i در 2i است. اگر $2i > n$ باشد آنگاه i فرزند چپ ندارد.

۳. اگر $2i+1 \leq n$ باشد آنگاه فرزند راست i در $2i+1$ است. اگر $2i+1 > n$ باشد آنگاه i فرزند راست ندارد.

• اصل موضوعی (2) [حداکثر تعداد گره‌ها]

اگر T یک درخت دودوئی کامل با n گره باشد، بطوریکه گره‌های آن با اندیس i و

$1 < i \le n$ اندیس‌گذاری شده است(از چپ به راست) آنگاه:

۱. حداکثر تعداد گره‌ها در سطح i ام یک درخت دودوئی برابر با $2^{i-1} (i \ge 1)$ است.

۲. 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 (۳) [رابطه بین تعداد گره‌های پایانی و گره‌های درجه ۲]**

برای هر درخت دودوئی غیرتهی مانند T، اگر n_o تعداد گره‌های پایانی و n_2 تعداد گره‌های درجه ۲ باشد، آنگاه خواهیم داشت:

$$n_o = n_۲ + ۱ \qquad (۱)$$

(یعنی تعداد برگ‌ها همواره در درخت دودوئی یکی بیشتر از تعداد گره‌های درجه 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:

$$n = n_o + n_1 + n_2 \ (۲)$$

و اگر B نشانگر تعداد انشعاب‌های یک درخت دودوئی باشد آنگاه

$$n = B + 1 \ (۳)$$

خواهد بود و می‌دانیم همه انشعاب‌ها یا از یک گره با درجه یک یا از یک گره با درجه دو به‌وجود آمده‌اند. بنابراین:

$$B = n_1 + 2n_2 \qquad (٤)$$

آنگاه با جایگذاری رابطه (٤) در رابطه (۳) خواهیم داشت:

$$n = 1 + n_1 + 2n_2 \qquad (٥)$$

با کم کردن عبارت (٥) از (۲) و ساده نمودن آن خواهیم داشت:

$$n_o = n_2 + 1$$

٥-٦ نمایش درخت‌های دودوئی

فرض کنید T یک درخت دودوئی باشد، می شهای نمایش درخت دودوئی T

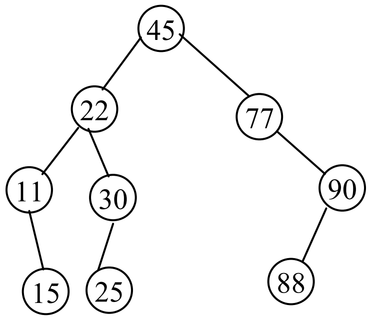
را بررسی کنیم. در این بخش دو روش نمایش T را در مورد بحث و بررسی قرار می‌دهیم. روش اول و معمول، روش نمایش پیوندی درخت T است و مشابه روش لیست‌های پیوندی است. روش دوم که تنها از یک آرایه استفاده می‌کند نمایش ترتیبی درخت T است. اصلی‌ترین موضوع در نمایش درخت T، اینست که به ریشه R درخت T دسترسی مستقیم داشته باشیم و با معلوم بودن هر گره N از T باید بتوان به هر فرزند N دسترسی پیدا کرد.

۱-۵-٦ نمایش ترتیبی درخت‌های دودوئی

فرض کنید T یک درخت دودوئی باشد که کامل یا تقریباً کامل است. روش کاراتری برای نگهداری T در حافظه وجود دارد. که نمایش ترتیبی T [illegible] ش تنها از یک آرایه یک بعدی به نام Tree به‌صورت زیر استفاده می‌کند: (از موقعیت صفر آرایه استفاده نمی‌شود)

- ابتدا گره‌های درخت دودوئی را از یک شماره‌گذاری می‌کنیم (از چپ به راست).
- محتوای هر گره درخت دودوئی با شماره خاص در خانه‌ای از آرایه با همان شماره ذخیره می‌شود.

نمایش ترتیبی درخت دودوئی T زیر در شکل ۷-٦ در شکل زیر نشان داده شده است.



شکل زیر نمایش ترتیبی درخت بالا را نمایش می‌دهد:

۱	۲	۳	٤	٥	٦	۷	۸	۹	۱۰	۱۱	۱۲	۱۳	۱٤	۱٥
٤٥	۲۲	۷۷	۱۱	۳۰		۹۰		۱٥	۲٥				۸۸	

شکل ۸-٦ نمایش ترتیبی درخت

استفاده از فرم آرایه برای درخت‌های کامل یا تقریباً کامل مناسب است چرا که حافظه کمتری به هدر می‌رود. علاوه از آن، استفاده از آرایه برای پیاده‌سازی درخت‌های کامل همان‌طور که می‌دانید، ساده‌تر می‌باشد.

برای درخت‌های غیرکامل استفاده از آرایه به دلیل هدر دادن حافظه زیاد توصیه نمی‌شود.

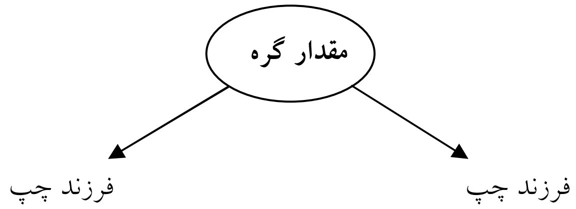
مزایای نمایش درخت دودوئی کامل با آرایه
۱. هر گره‌ای از طریق گره دیگری به راحتی و از طریق محاسبه اندیس قابل دستیابی است (با استفاده از اصول موضوعی بخش قبل). ۲. فقط داده‌ها ذخیره می‌شوند و نیازی به ذخیره اشاره‌گرهای زیردرخت چپ و راست نمی‌باشد. ۳. در زبان برنامه‌سازی که فاقد تخصیص حافظه پویا هستند (مثل بیسیک و فرترن)، نمایش آرایه تنها راه ذخیره درخت است. ٤. پیاده‌سازی با آرایه ساده‌تر است.

به‌طورکلی و با توجه به امتیازات فوق، نمایش ترتیبی یک درخت به عمق h به $2^h - 1$ خانه آرایه نیاز دارد بنابراین این روش برای درختان غیر پر از کارایی لازم برخوردار نیست به‌خصوص برای درخت‌های مورب که فقط h خانه آن مورد استفاده قرار می‌گیرد و بقیه خانه‌ها بدون استفاده می‌ماند.

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.

۲-۵-٦ 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 می‌شود. علاوه بر این، نمایش درخت‌ها با آرایه، مشکل نیاز به جابجایی عناصر برای انجام درج و حذف گره‌ها را دارد. این مشکلات با استفاده از پیاده‌سازی از طریق لیست پیوندی (استفاده از اشاره‌گرها) را می‌توان برطرف کرد.



- **ساختار هر گره درخت دودوئی در نمایش پیوندی**

درخت دودوئی T را در نظر بگیرید. درخت T در حافظه به‌وسیله یک نمایش پیوندی نگهداری می‌شود و هر گره این لیست از سه فیلد left , right , info به‌صورت زیر تشکیل یافته است:

left	info	Right

به‌طوری‌که:

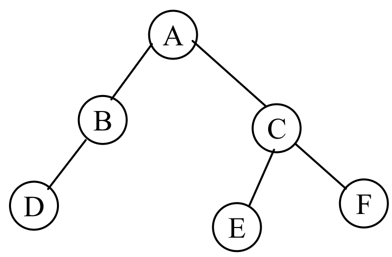
info حاوی مقدار هر گره است.

right حاوی مکان یا اشاره‌گری به فرزند راست گره N است.

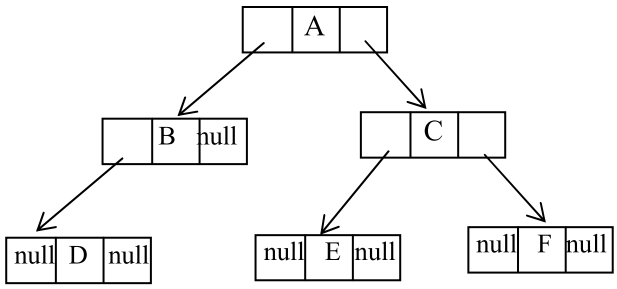
left حاوی مکان یا اشاره‌گری به فرزند چپ گره N است.

گره‌های با ساختار بالا ساخته شده و با به هم وصل شدن، تشکیل یک درخت دودوئی می‌دهند که با لیست پیوندی پیاده‌سازی شده است.

مثال ۲-٦: درخت دودوئی زیر را در نظر بگیرید:



نمایش پیوندی درخت دودوئی بالا به‌صورت زیر خواهد بود:



همان‌گونه که گفتیم تعیین پدر، یک اصل اساسی می‌باشد که در روش فوق تعیین آن مشکل می‌باشد. برای حل این مشکل می‌توان فیلد چهارمی به نام parent مانند شکل زیر به ساختار هر گره اضافه نمود که به پدرش اشاره کند. با استفاده از این فیلد می‌توان به پدر گره‌ها دستیابی پیدا کرد.

left	Parent	info	right

حال ساختار گره‌ها در درخت را به‌صورت زیر ارائه می‌دهیم:

پیاده‌سازی گره‌های درخت دودوئی

```
struct  node {
            node  *left ;
            elementtype  info ;
            node   *right ;
        };
```

پس از توصیف ساختار گره، باید گره‌ای را ایجاد کنیم و به درخت اضافه کنیم. برای انجام این کار از تابع getnode() با ساختار گره جدید باید استفاده کنیم.

این تابع حافظه‌ای به اندازه ساختمان node اختصاص می‌دهد و آدرس را در اشاره‌گر تعریف شده، قرار می‌دهد.

٦-٦ پیمایش درخت‌های دودوئی

اعمال زیادی وجود دارد که می‌توان روی درخت‌های دودوئی انجام داد. مانند پیدا کردن گره اضافه کردن گره جدید و غیره. اما عملگی که معمولاً بیشتر روی درخت‌های دودوئی صورت می‌گیرد، ایده پیمایش درخت یا دستیابی به همه گره‌های درخت 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 حرزکیت‌دریخچپچ(پ)، ملاقات کردن یک گره (برای مثال چاپ اطلاعات موجود در گره) حرکت به راست (زیر درخت راست) باشد، آنگاه شش ترکیب ممکن برای پمایش یک درخت خواهیم داشت:

RLV , RVL , VRL , VLR, LRV, LVR

اگر تنها حالت‌هائی را انتخاب کنیم که ابتدا گره سمت چپ و سپس گره سمت راست ملاقات کنیم، تنها سه ترکیب VLR , LRV , LVR را خواهیم داشت که این سه ترکیب، سه روش استاندارد برای پیمایش 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 پیاده‌سازی کرد. 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.

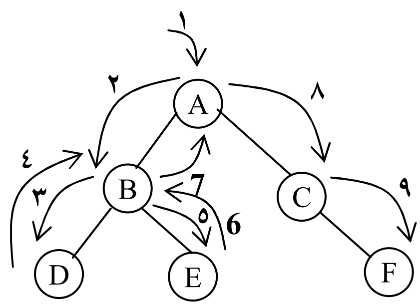
١-٦-٦ روش پیمایش پیشوندی (Preorder)

در روش پیمایش Preorder یا VLR یک درخت دودوئی غیرخالی به‌صورت زیر پیمایش می‌شود:

روش پیمایش پیشوندی (Preorder)
ریشه را ملاقا. ت کن.
۲. زیردرخت چپ را به روش Preorder پیمایش کن.
۳. زیردرخت راست را به روش Preorder پیمایش کن.

پیمایش Preorder با توجه به مراحل فوق بدین صورت است: ع
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 ۳-۶: درخت دودوئی زیر را در نظر گرفته، درخت را به روش preorder پیمایش کنید.



شکل ۱۰-٦ درخت دودوئی

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. راه‌حل به‌صورت 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 فرزند راست و چپی ندارد، به بالا می‌گردیم (مرحله ٦) و چون فرزندان راست و چپ گره B قبلاً ملاقات شده‌اند باز یک مرحله دیگر هم به بالا برمی‌گردیم (مرحله ۷) حال فرزند راست گره A را ملاقات می‌کنیم و در خروجی می‌نویسیم (مرحله ۸). چون گره C دارای فرزند چپی نمی‌باشد به طرف فرزند راست آن حرکت می‌کنیم و آن را در خروجی می‌نویسیم (مرحله ۹).خروجی حاصل از پیمایش این درخت به‌صورت زیر خواهد بود:

نتیجه پیمایش: ABDECF

- **پیاده‌سازی پیمایش Preorder به‌صورت بازگشتی**

هدف تابع Preorder پیمایش درخت دودوئی به‌صورت پیشوندی می‌باشد. همانگونه که اشاره گردید در این روش پیمایش ابتدا ریشه ملاقات می‌شود سپس فرزندان چپ و بعد از آن فرزندان است:

زیربرنامه پیمایش Preorder به‌صورت بازگشتی

```
void    Preorder (node   *tree)
{
        if (tree)
        {
              cout<<tree -> info ; //Printf("%d", tree -> info );
              Preorder (tree -> left );
              Preorder (tree -> right );
        }
}
```

۲-٦-٦ روش پیمایش میانوندی (inorder)

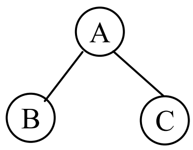
در روش پیمایش inorder یا LVR یک درخت دودوئی غیرخالی به‌صورت زیر پیمایش می‌شود:

روش پیمایش inorder
۱. زیردرخت چپ را به روش inorder پیمایش کن. ۲. ریشه را ملاقات کن. ۳. زیردرخت راست را به روش inorder پیمایش کن.

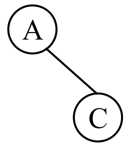
پیمایش inorder با توجه به مراحل فوق بدین صورت است که: از ریشه شروع کرده تا جائی که ممکن است به سمت چپ حرکت می‌کنیم با رسیدن به آخرین گره سمت چپ، محتویات آن گره را ملاقات می‌کنیم و سپس به سمت راست حرکت می‌کنیم و با آن مثل گره ریشه برخورد می‌کنیم و به منتهی‌الیه سمت چپ می‌رویم و آن گره را ملاقات می‌کنیم در گره ای 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 کلیه گره‌های درخت ادامه می‌دهیم.

مثال ٤-٦: درخت ساده شکل زیر را در نظر بگیرید. در پیمایش میانوندی این درخت ابتدا فرزند چپ آن یعنی B ملاقات می‌شود و سپس خود ریشه A و بعد از آن فرزند راست ریشه یعنی C ملاقات می‌شود.

Inorder= BAC

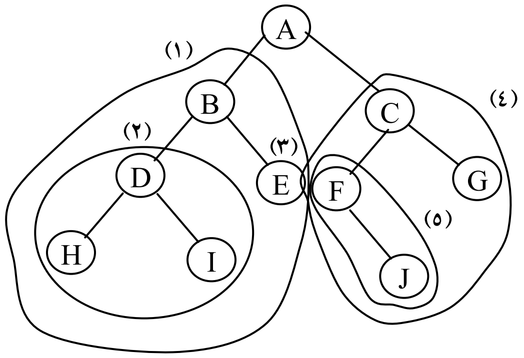


درخت ساده شکل زیر را در نظر بگیرید، چون زیردرخت چپ خالی است پیمایش به‌صورت AC خواهد بود:



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.

مثال ۵–٦: درخت شکل زیر را در نظر بگیرید پیمایش inorder درخت را به‌دست آورید.



در مرحله (۱) به سراغ زیردرخت چپ ریشه 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 نشان‌دهنده یک درخت ساده است که می‌توان به راحتی عمل پیمایش را روی آن انجام داد. پیمایش میانوندی این زیردرخت ساده به‌صورت HDI می‌باشد. حال پس از پیمایش زیردرخت مرحله (۲) گره ریشه یعنی B را 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 سراغ زیردرخت راست گره 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 پیمایش کرد (F , G). بعد از پمایش زیر درخت مرحله (۵) ریشه C را ملاقات می‌کنیم و به سراغ زیر درخت راست آن یعنی G رفته و آن را پیمایش می‌کنیم.

خروجی حاصل از پیمایش این درخت به‌صورت زیر خواهد بود:

HDIBEAFJCG

• **پیاده‌سازی پیمایش inorder به‌صورت بازگشتی**

هدف تابع inorder، پیمایش درخت دودوئی به‌صورت میانوندی می‌باشد. همان‌گونه که در مثال فوق دیدید در این روش پیمایش ابتدا زیردرخت چپ و سپس ریشه و بعد زیردرخت راست پیمایش می‌شود.

تابع پیمایش inorder به‌صورت بازگشتی

```
void   inorder (node  *tree)
 {
          if(tree)
            {
                  inorder (tree → left );
                cout<<tree → info ; //Printf("%d", tree → info );
                  inorder ( tree → right );
            }
}
```

۳-۶-۶ روش پیمایش پسوندی (Postorder)

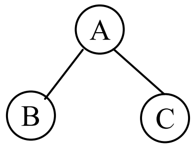
در روش پیمایش Postorder یا LRV یک درخت دودوئی غیرخالی به‌صورت زیر پیمایش می‌شود:

روش پیمایش Postorder
۱. زیردرخت چپ را به روش Postorder پیمایش کن.
۲. زیردرخت راست را به روش Postorder پیمایش کن.
۳. ریشه را ملاقات کن.

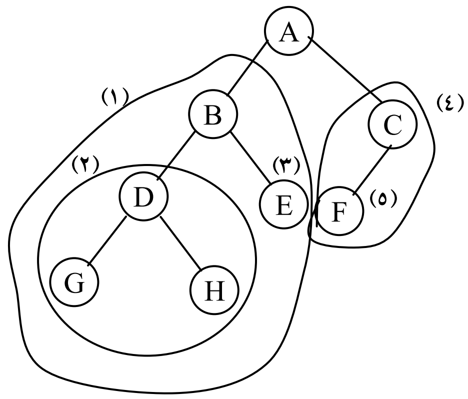
پیمایش Postorder بدین صورت است که: از ریشه شروع می‌کنیم و به طرف چپ حرکت می‌کنیم تا به آخرین گره برسیم و از این گره شروع می‌کنیم تا جایی که ممکن است به سمت راست حرکت می‌کنیم. چنانچه حرکت به راست ممکن نباشد، محتویات این گره را ملاقات و به گره بالایی برمی‌گردیم.

مثال ٦-٦: درخت ساده شکل زیر را در نظر بگیرید. در پیمایش پسوندی این درخت ابتدا فرزند چپ ریشه، سپس فرزند راست ریشه و بعد خود ریشه ملاقات می‌شود:

Postorder = BCA



مثال ٦-٧: درخت شکل زیر را در نظر بگیرید و پیمایش Postorder این درخت را به‌دست آورید:



ابتدا به سمت زیر درخت چپ ریشه می‌رویم (مرحله ۱) حال B را به‌عنوان ریشه جدید در نظر می‌گیریم و به طرف زیر درخت چپ آن می‌رویم (مرحله ۲).
زیردرخت مرحله (۲) یک درخت دودوئی ساده می‌باشد که به‌صورت GHD پیمایش می‌شود. حال به سراغ زیردرخت راست B می‌رویم و آن را پیمایش می‌کنیم (مرحله 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 زیر درخت راست آن

می‌رویم (مرحله ٤) در این مرحله گره C را به‌عنوان ریشه جدید در نظر می‌گیریم و به سراغ زیر درخت چپ آن می‌رویم یعنی F (مرحله ۵) و آن را پیمایش می‌کنیم و چون C دارای زیر درخت راست نمی‌باشد خودش را پیمایش می‌کنیم چون دو زیردرخت چپ و راست ریشه اصلی یعنی A پیمایش شد، بنابراین خود A نیز پیمایش می‌شود خروجی حاصل از پیمایش این درخت به‌صورت زیر می‌باشد:

GHDEBFCA

• **پیاده‌سازی پیمایش postroder به‌صورت بازگشتی**

هدف تابع Postorder، پیمایش درخت دودوئی به‌صورت پسوندی می‌باشد. در این روش پیمایش ابتدا زیر درخت چپ سپس زیر درخت راست و بعد ریشه پیمایش می‌شود.

تابع پیمایش postroder به‌صورت بازگشتی

```
void Postorder (node   *tree)
{
        if ( tree)
          {
                Postorder (tree -> left );
                Postorder(tree -> right );
                cout<<tree -> info ;//Printf("%d",node -> info);
          }
}
```

٤-٦-٦ پیمایش غیربازگشتی درخت دودوئی

همان‌گونه که اشاره گردید، الگوریتم‌های پیمایش درخت‌های دودوئی را می‌توان به‌صورت بازگشتی نوشت. حال در این قسمت الگوریتم غیربازگشتی پیمایش میانوندی را بررسی می‌کنیم. در این پیمایش، گره‌ها باید در پشته قرار گیرند و در صورت لزوم از آن خارج شوند. در حالت بازگشتی عمل قرار دادن گره‌ها در پشته و حذف آن‌ها از پشته توسط سیستم انجام می‌شود. در حالی که در روش غیربازگشتی، این عمل باید

توسط برنامه صورت گیرد. تابع ()inorder2 نشان‌دهنده پیاده‌سازی میانوندی به‌صورت غیربازگشتی می‌باشد.

تابع پیمایش inorder به‌صورت غیربازگشتی

inorder

```
#define M 100
void inorder2 (node  *tree)
  {
        struct stack
          {
                int top ;
                node  item [M] ;
          }s;
           node  *p ;
          s.top=-1;
          p=tree;
          do{
                while ( p!= NULL)
                  {
                    push (s,p);
                     p = p → left ;
                }
                if (!empty(s))
                {
                    p=pop(s) ;
                  cout<< p → inf o ;//printf("%d",P → info);
                    p = p → right ;
                }
         }while (!empty(s) ||  P!= NULL) ;
 }
```

برای نوشتن پیمایش preorder درخت به‌صورت غیربازگشتی کافی است در برنامه فوق دستورات را به قبل از دستور Push (s,p) انتقال دهید.

۷-۶ کاربردهای پیمایش درخت دودوئی

در این بخش می‌خواهیم اشاره‌ای به کاربردهای پیمایش درخت دودوئی بپردازیم تا

اهمیت موضوع روشن شود.

۱-۷-٦ ساخت درخت دودوئی با استفاده از پیمایش آن

ما تاکنون با استفاده از درخت دودوئی داده شده، مبادرت به پیمایش آن می‌کردیم. اکنون می‌خواهیم برعکس اینکار را انجام دهیم یعنی، با استفاده از پیمایش داده شده یک درخت دودوئی، اقدام به ساخت خود درخت کنیم. سؤال اینست که آیا این‌کار امکان‌پذیر است یا نه؟

قابل ذکر است اگر یک نوع پیمایش از درخت موجود باشد، نمی‌توان 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.

به‌عنوان مثال اگر فقط پیمایش inorder درخت در دست باشد و با استفاده از این پیمایش بخواهیم درخت را بسازیم، نمی‌توانیم درخت اولیه را بسازیم، بلکه چند درخت به‌دست می‌آید که ممکن است یکی از آنها درخت اولیه بوده باشد. ولی اگر پیمایش inorder درخت و یکی از دو پیمایش Postorder و یا preorder درخت موجود باشد، می‌توان درخت منحصر به فردی را ساخت.

اگر پیمایش میانوندی و یکی از پیمایش‌های پسوندی یا پیشوندی یک درخت دودوئی را داشته باشیم، می‌توانیم آن درخت را به‌صورت یکتا ترسیم کنیم.

قاعده اصلی برای ایجاد درخت با استفاده از پیمایش آن به‌صورت زیر است:

ساخت درخت دودوئی به‌صورت یکتا با داشتن پیمایش میانوندی و یکی از پیمایش‌های پسوندی یا پیشوندی
اگر پیمایش Preorder مشخص باشد، اولین گره آن، ریشه است. اگر نمایش Postorder مشخص باشد آخرین گره، ریشه است. وقتی گره ریشه مشخص شد، تمام گره‌های زیردرخت چپ و زیردرخت راست را می‌توان با استفاده از نمایش inorder پیدا کرد.

با توجه به سه روش پیمایش بررسی شده، ملاحظه می‌کنیم که هر الگوریتم دارای همین سه مرحله است و زیردرخت چپ ریشه همواره قبل از زیردرخت راست پیمایش می‌شود. تفاوت این سه الگوریتم در زمان ملاقات ریشه می‌باشد. به‌طور مشخص در الگوریتم "pre"، ریشه قبل از پیمایش زیر درخت‌ها ملاقات می‌شود. در الگوریتم دارای "In" ریشه مابین پیمایش زیردرخت‌ها پردازش می‌شود و الگوریتم دارای "post" ریشه بعد از پیمایش زیردرخت‌ها ملاقات می‌شود.

نکته: اگر پیمایش های پسوندی یا پیشوندی یک درخت دودوئی را داشته باشیم، ممکن است نتوانیم آن درخت را به‌صورت یکتا ترسیم کنیم.

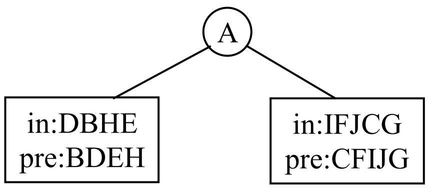
مثال ۸–٦: فرض کنید پیمایش‌های inorder و preorder یک درخت دودوئی به‌صورت زیر باشد، درخت دودوئی موردنظر را رسم کنید.

inroder: D B H E A I F J C G

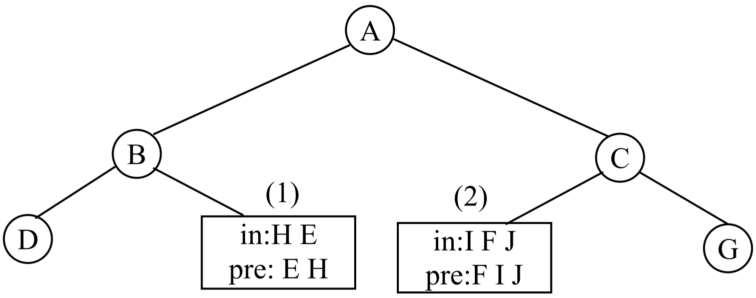
Preorder: A B D E H C F I J G

با توجه به پیمایش preorder متوجه می‌شویم که A ریشه درخت است.

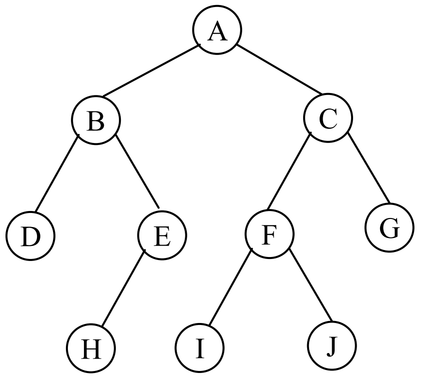
در پیمایش inorder، تمام گره‌های موجود در سمت چپ A متعلق به زیردرخت چپ و تمام گره‌های موجود در سمت راست A، متعلق به زیردرخت راست است.



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 گره‌های سمت چپ B را به‌عنوان زیردرخت و تمام گره‌های راست آن را به‌عنوان زیردرخت راست در نظر می‌گیریم و همین کار را برای زیردرخت راست ریشه اصلی یعنی A نیز انجام می‌دهیم.



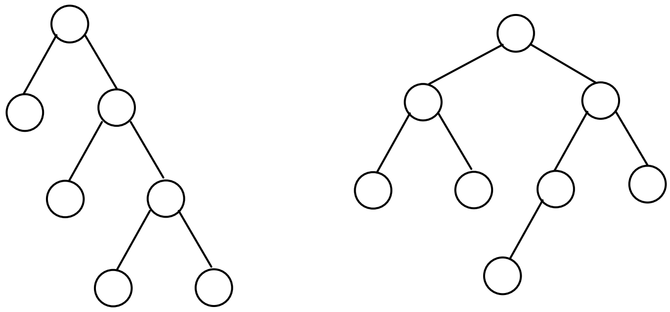
حال در بسته‌ای با شماره (۱) مشخص می‌شود که E ریشه می‌باشـد و H فرزنـد چپ آن و در بسته‌ای با شماره (۲)، F ریشه مـی‌باشـد و I فرزنـد چـپ آن و J فرزنـد راست آن خواهد بود.



۲-۷-٦ نمایش عبارات محاسباتی با درخت دودوئی

یکی از کاربردهای درختهای دودوئی، نمایش عبارات محاسباتی توسط درخت دودویی محض است. درخت دودوئی محض یک درخت دودوئی است که در آن تمام گره‌ها از درجه صفر یا از درجه ۲ باشند. به‌عبارت دیگر، هر گره غیربرگ فقط دو فرزند دارد.

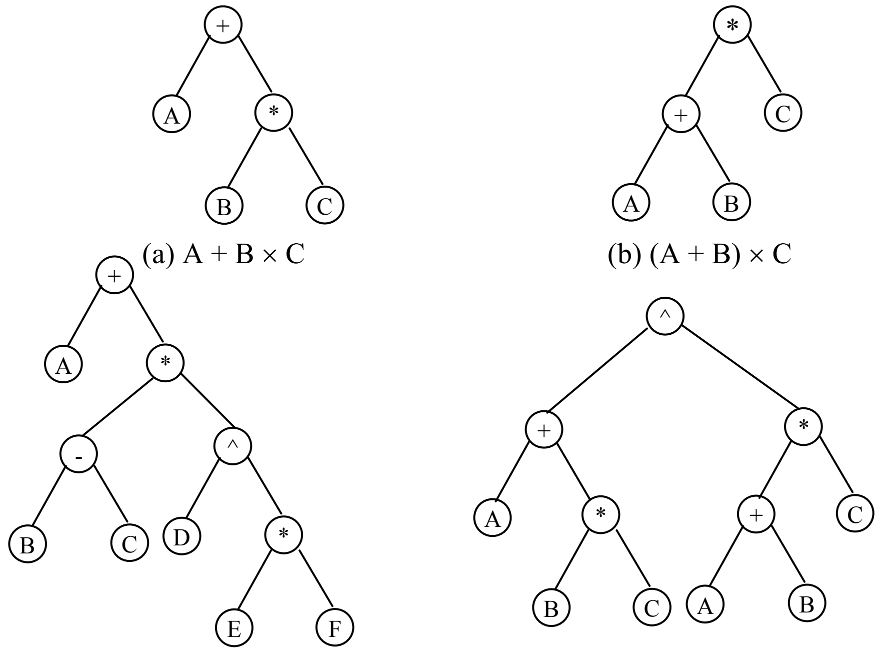
به‌عنوان مثال درخت شکل ۱۱-٦ (الف) نشان‌دهنده یک درخت دودوئی محـض می‌باشد و در حالی که درخت شکل ۱۱-٦ (ب) درخت دودوئی محض نیست:



شکل ۱۱-٦ (ب) درخت دودوئی که محض نیست (الف) درخت دودوئی محض

برای نمایش عبارت محاسباتی توسط درخت دودوئی محض هر عملگری را در یک گره و عملوندهای اول و دوم آن را در زیر درخت‌های چپ و راست گره نشان می‌دهیم. در این درخت عملگرها در گره‌های داخلی (غیربرگ) و عملوندها در گره‌های برگ درخت دودوئی محض قرار می‌گیرند.

شکل ۱۲-٦ چند عبارت و نمایش آنها را به‌صورت درخت نشان می‌دهد.



(a) A + B × C (b) (A + B) × C

(c) A + (B – C) × D^ (E × F) (d) (A+B×C)^((A+B)×C)

شکل ۱۲-٦ چند عبارت و نمایش درختی آنها

پیمـایش preorder درخـتهـای فـوق عبـارت prefix و پیمـایش postorder درخت‌های فوق عبارت postfix معادل را تولید می‌کند. اما اگر درخـت شـکل (a)، بـه روش inorder پیمایش گردد، عبارت A+B*C کـه یـک عبـارت infix اسـت حاصـل می‌گردد. اما چون ترتیب انجام اعمال، از ساختار درخت نتیجه می‌شود، درخت دودوئی فاقد هرگونـه پـارامتر اسـت. لـذا یـک عبـارت infix کـه مـستلزم اسـتفاده از پرانتزهـا جهت تعویض تقدم عادی عملگرها است را نمی‌توان با پیمایش inorder ساده به‌دسـت آورد.

۳-۷-٦ پیمایش ترتیب سطحی

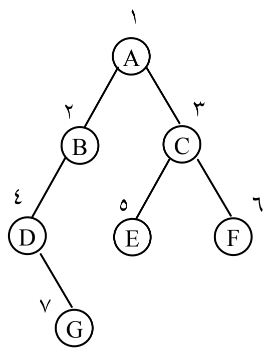
پیمایش‌های پیشوندی، میانوندی و پسوندی که در بخش قبلـی مـورد بحـث و بررسـی قرار دادیم در واقع پیمایش‌های عمقی هستند که برای به‌کارگیری آنها نیاز به اسـتفاده از پشته است. پیمایش ترتیب سطحی روش دیگری از پیمایش درخت دودوئی اسـت کـه بجای پشته از صف استفاده می‌کند. عملکرد این پیمایش بدین صورت است که در ابتدا ریشه بازیابی می‌شود سپس فرزند چپ ریشه و به دنبال آن فرزند راست ریشه بازیـابی می‌گردد. این روش بازیابی را برای تمام سطوح درخت اعمال می‌کنیم

الگوریتم این پیمایش به‌صورت زیر می‌باشد:

الگوریتم پیمایش ترتیب سطحی درخت دودوئی

```
void   level_order (node   *tree)
{
        front = rear= -1;
        while (tree)
        {
                cout<< tree → inf o ;//printf("%d", node → info);
                if (tree → left )
                        addq ( tree → left );
                    if ( tree → right )
                        addq ( tree → right ) ;
                    delete(tree) ;
        }
}
```

درخت دودوئی شکل ۱۳–٦ را در نظر بگیرید. در پیمایش ترتیب سطحی ابتدا ریشه را بازیابی می‌کنیم، گره‌های سطح بعدی (فرزندان ریشه) را از چپ به راست بازیابی می‌کنیم و بعد از آن به سراغ سطح‌های بعدی می‌رویم تا زمانی که کل درخت را پیمایش کرده باشیم.



$\Rightarrow$ A B C D E F G

شکل ۱۳–٦ درخت دودوئی

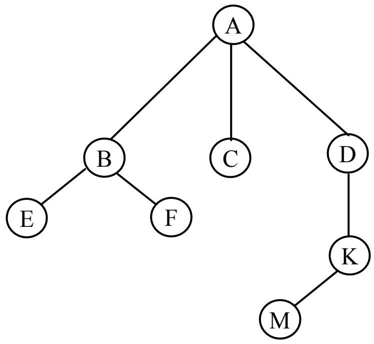
یا به بیان دیگر، می‌توان چنین گفت که، کلیه گره‌ها را از بالا به پایین و از چپ به راست شماره‌گذاری می‌کنیم و به ترتیب شماره در خروجی می‌نویسیم.

۸–٦ بررسی انواع درخت‌ها

در این بخش می‌خواهیم انواع درخت‌ها را مورد بررسی قرار دهیم، تا بتوانیم در حل مسائل از آنها استفاده کنیم.

۱–۸–٦ 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 ۱٤–٦ یک درخت عمومی با ۸ گره نشان می‌دهد.



شکل-۱۵ درخت عمومی

ریشه درخت فوق A و فرض می‌کنیم فرزندهای 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 مجموعه مجزا به نام‌های $T_n,...,T_2,T_1$ تقسیم می‌شوند که در آن هر T_i به ازای $i=1,2,...,n$ یک درخت است. $T_n,...,T_2,T_1$ زیردرخت‌های ریشه نامیده می‌شوند.

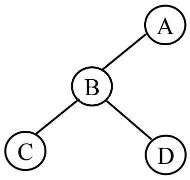
با توجه به تعریف فوق می‌توان این نکته را فهمید که یک درخت دودوئی T' حالت خاصی از درخت عمومی T نیست و این دو در دو دسته مختلف قرار دارند. اختلاف اساسی آنها عبارتند از:

الف) یک درخت دودوئی می‌تواند خالی باشد ولی یک درخت عمومی نمی‌تواند خالی (یا تهی) باشد.

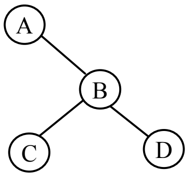
ب) فرض کنید درخت تنها یک فرزند دارد آنگاه این فرزند در یک درخت دودوئی با عنوان فرزند راست یا چپ از هم متمایز می‌شوند، اما در یک درخت عمومی

هیچگونه تمایزی بین آنها وجود ندارد.

مثال ۹-٦: در شکل زیر را در نظر بگیرید:



(ب)



(الف)

شکل‌های الف و ب به‌عنوان درخت‌های دودوئی دو درخت متمایز هستند ولی به‌عنوان درخت‌های عمومی با هم هیچگونه تفاوتی ندارند.

• 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:

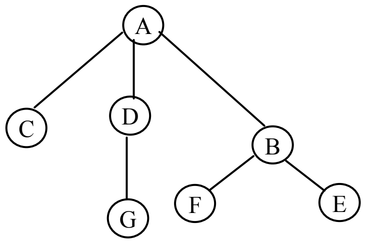
۱. نمایش درخت عمومی با آرایه
۲. نمایش پیوندی (نمایش درخت عمومی با لیست پیوندی)
۳. نمایش درخت عمومی با استفاده از درخت دودوئی

۱. نمایش درخت عمومی با آرایه

نمایش درخت عمومی با آرایه ساده می‌باشد. برای این کار به سه آرایه نیاز است:

- آرایه info برای ذخیره محتویات گره درخت
- آرایه son برای نگهداری چپ‌ترین فرزند گره
- Sibiling همزاد گره را ذخیره می‌کند.

مثال ۱۰-٦: درخت عمومی شکل (الف) را در نظر بگیرید. نمایش این درخت با استفاده از آرایه به‌صورت شکل (ب) می‌باشد.



(الف)

	Info	son	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

(ب)

۲. نمایش پیوندی درخت

یکی از نکات مهم در درخت‌ها، گره است. هر گـره درخـت دودوئـی شـامل دو فیلـد اشاره‌گر است. که به فرزندان چپ و راست اشـاره مـی‌کنـد. امـا هـر گـره در درخـت عمومی می‌تواند چندین اشاره‌گر داشته باشد. تعداد فرزندان هـر گـره درخـت متفـاوت است و می‌تواند خیلی زیاد یا خیلی کم باشد. یک روش برای جلوگیری از اینکار، ایـن است که تعداد فرزندان یک گره را محدود کنیم به‌عنوان مثال می‌توانیم حداکثر فرزنـدان هر گره را m در نظر بگیریم. در این صورت ساختار هر گره را به‌صورت زیر مـی‌تـوان نمایش داد:

داده	پیوند ۱	پیوند ۲	....	پیوند m

چنین ساختاری را می‌توان به‌صورت زیر پیاده‌سازی کرد:

```
# define M  20
   struct   tree {
          elementtype  info;
          tree   *sons [M];
     };
```

اگر تعداد فرزندان هر گره ۲ یا حتی صفر باشد در این صورت فـضای زیـادی از حافظه به هدر می‌رود. فرض کنید، می‌خواهیم یک درخت m تایی (درختی با درجه m) را که حاوی n گره است نمایش دهیم. لم زیر نشان می‌دهد کـه چقـدر فـضای حافظـه

هدر می‌رود.

لم: اگر T درخت mتایی با n گره باشد آنگاه $n(m-1)+1$ تعداد از $n*m$ پیوند تهی خواهند بود $(n \geq 1)$.

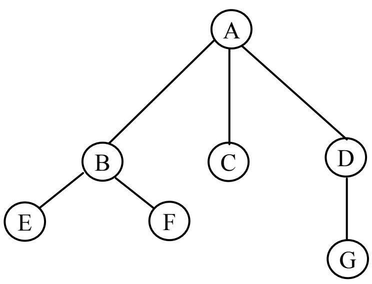
براساس این لم، در یک درخت سه‌تایی، بیش از $\frac{۲}{۳}$ پیوندها تهی‌اند. میزان فضایی که به هدر می‌رود با افزایش درجه درخت افزایش می‌یابد.

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 به‌صورت زیر تعریف کرد که در آن، فرزندان هر گره در یک لیست پیوندی قرار می‌گیرند.

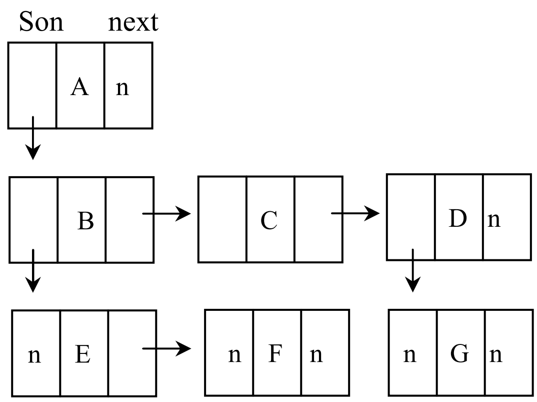
تعریف ساختار درخت
```Struct tree { elementtype info; tree *son; tree *next; };```

```
Struct    tree {
          elementtype  info;
          tree  *son;
          tree  *next;
};
```

مثال ۱۱-۶: شکل (الف) را در نظر بگیرید. نمایش پیوندی این درخت عمومی در شکل (ب) نمایش داده شده است. (n نشان‌دهنده null می‌باشد)



الف



ب

۳. نمایش درخت عمومی به‌صورت درخت دودوئی

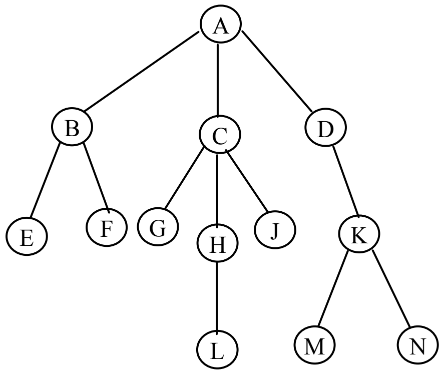
نمایش درخت عمومی با استفاده از درخت دودوئی، کارآمد و عملی است. هر درخت را می‌توان به‌صورت یک درخت دودوئی منحصر به فرد نمایش داد. با الگوریتم زیر می‌توان یک درخت عمومی را به درخت دودوئی معادل و منحصر به فردش تبدیل کرد:

i. در هر سطح کلیه گره‌های کنار هم، که فرزند یک پدر هستند را به یکدیگر وصل کنید.

ii. ارتباط کلیه گره‌ها به پدر را به جزء اتصال سمت چپ‌ترین فرزند، قطع کنید.

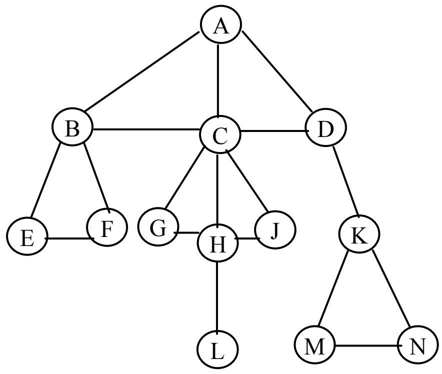
iii. گره‌های متصل به هم، در هر سطح افقی را ۴۵ درجه در جهت حرکت عقربه‌های ساعت بچرخانید.

مثال ۱۲-٦: درخت شکل زیر را در نظر بگیرید:

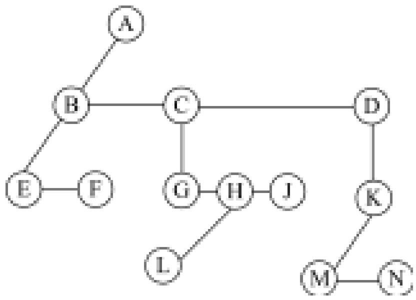


این درخت را به‌صورت دودوئی درمی‌آوریم.

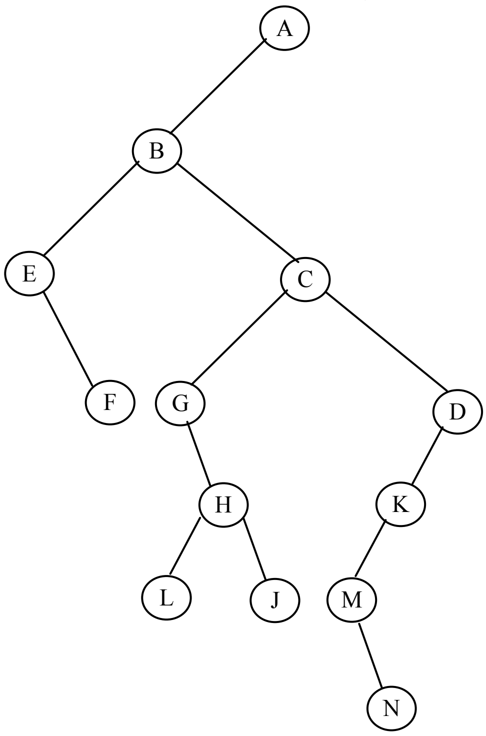
۱. ابتدا در هر سطح کلیه گره‌های کنار هم را به یکدیگر وصل می‌کنیم. (توجه کنید باید متعلق به یک پدر باشند)



۲. ارتباط کلیه گره‌ها به پدر خودش به جزء سمت چپ‌ترین فرزند را قطع می‌کنیم. بنابراین شکل زیر حاصل می‌شود:



۳.گره‌های متصل به هم در سطح افقی را ٤٥ درجه در جهت عقربه‌های ساعت می‌چرخانیم. لذا خواهیم داشت:



• **پیمایش درخت‌ها**

پیمایش درخت‌ها نیز مشابه درخت‌های دودوئی به سه روش انجام می‌شود که عبارتند

از:

۱. روش میانوندی (inorder)

۲. روش پسوندی (postrorder)

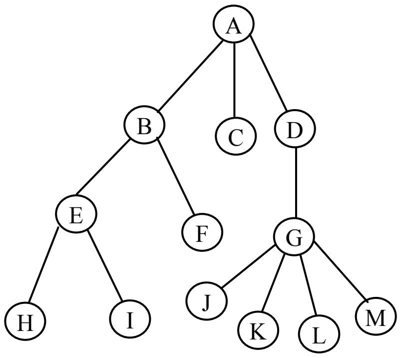
۳. روش پیشوندی (preorder)

در اینجا ما، تابع روش پیمایش inorder را به‌عنوان نمونه ارائه می‌دهیم. پیاده‌سازی بقیه روش‌ها به‌عنوان تمرین به خواننده واگذار می‌شود:

روش پیمایش inorder درخت

```
void   inorder (tree * P)
{
        if (P!=NULL)
        {
                inorder ( p → Son );
                cout<< p → inf o ;//Printf("%d", P → info);
                inorder ( p → next );
        }
}
```

درخت شکل ۱۵–٦ را در نظر بگیرید:



شکل ۱۵–٦ درخت عمومی

پیمایش‌های مختلف درخت بالا به‌صورت زیر خواهد بود:

inorder:HIEFBCJKLMGDA
preorder: ABEHIFCDGJKLM
postrorder: HIEFBCJKLMGDA

۲-۸-٦ درختان نخی دودوئی

اگر نمایش پیوندی درخت دودوئی T را در نظر بگیرید، ملاحظه می‌کنید که تعداد اتصالات تهی در ورودیهای فیلدهای left و right بیشتر از تعداد اتصال‌های غیرتهی است. در یک درخت دودوئی با n گره، تعداد کل اتصالات آن 2n می‌باشد، که از این تعداد n+1 اتصال تهی است. این فضا با قرار دادن نوع دیگری از اطلاعات به جای ورودی‌های پوچ می‌تواند به شکل کاراتری مورد استفاده قرار گیرد. به‌طور مشخص ما اشاره‌گرهای خاصی را جانشین ورودیهای تهی می‌کنیم که به گره‌های بالاتر درخت اشاره می‌کند. این اشاره‌گرهای خاص را نخ‌کشی‌ها و درخت دودوئی حاصل را درخت‌های نخ‌کشی یا درخت‌های نخی می‌گویند.

نخ‌کشی‌ها در یک درخت نخ‌کشی شده باید از اشاره‌گرهای معمولی تمیز داده شوند. در نمودار یک درخت نخ‌کشی شده، نخ‌کشها را معمولاً با خط‌چین نمایش می‌دهند.

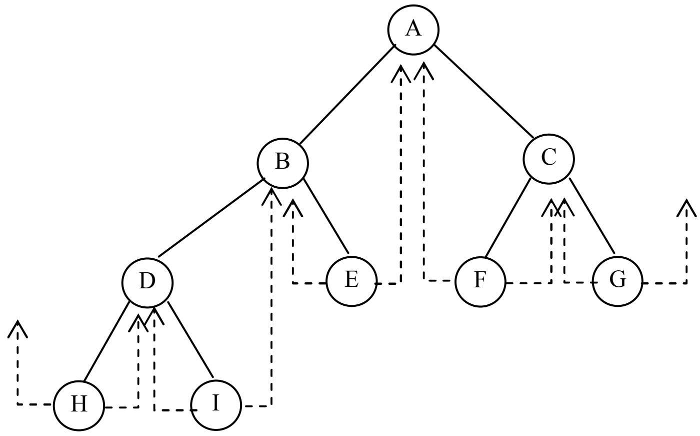
برای نخ‌کشی یک درخت دودوئی راه‌های متعددی وجود دارد اما هر نخ‌کشی متناظر با یک پیمایش خاص درخت T است. نخ‌کشی ما متناظر با پیمایش inorder درخت T است. برای ایجاد اتصالات نخی، می‌توان از قوانین زیر استفاده نمود (فرض کنید که ptr نشان‌دهنده یک گره می‌باشد):

۱. اگر $ptr \to left$ تهی باشد آن را طوری تغییر می‌دهیم که به گره‌ای که در پیمایش inorder قبل از ptr قرار دارد، اشاره کند.

۲. اگر $ptr \to right$ تهی باشد، آن را طوری تغییر می‌دهیم که به گره‌ای که در پیمایش inorder بعد از ptr قرار دارد، اشاره کند.

شکل ۱٦-٦ نمونه‌ای از درخت نخی را نشان می‌دهد که در آن اتصالات نخی به‌صورت نقطه‌چین مشخص شده است. این درخت دارای ۹ گره و ۱۰ اتصال تهی است که با اتصالات نخی تعریف شده‌اند. اگر درخت را به روش inorder پیمایش کنیم، گره‌های حاصل به‌صورت H D I B E A F C G خواهند بود. برای مشاهده

اینکه چگونه اتصالات نخی ایجاد می‌شوند، گره E را به‌عنوان نمونه انتخاب مـی‌کنـیم. چون فرزند چپ E یک اتصال تهی است آن را به گونه‌ای تغییر می‌دهیم که بـه گـره‌ای که قبل از E قرار دارد، یعنی B اشاره کند. به‌طور مشابهی، چون فرزند راست E نیز یک اتصال تهی است، آن را با اشاره‌گری به گره از E قرار می‌گیرد یعنی A تعویض می‌کنیم. بقیه اتصالات به‌طور مشابهی ایجاد می‌گردند.



شکل ۱۶–۶ نمونه‌ای از درخت نخی

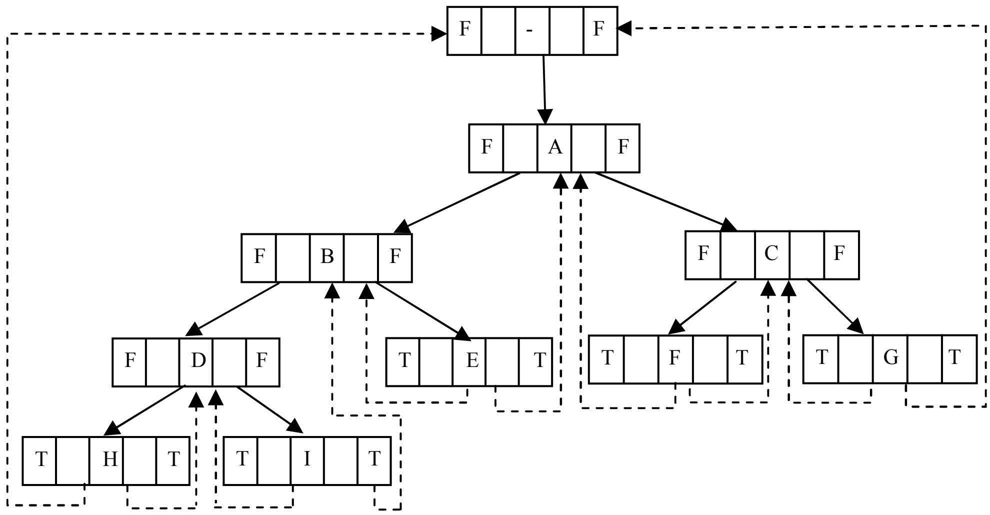
در شکل فوق اشاره‌گر سمت راست یعنی G و اشاره‌گر سمت چپ یعنـی H بـه هیچ جائی اشاره نمی‌کنند و در حالی که هدف از ارائه درخت‌های نخی ایـن بـود هـیچ اشاره‌گر تهی نداشته باشند. پس یک گره Head برای هر درخت دودوئی نخـی در نظـر می‌گیریم. یعنی همواره درخت نخی تهی دارای یک گره بنام گـره Head مطـابق شـکل زیر وجود دارد:

left-thread	left	info	right	right-thread
TRUE		-		FALSE

فرزند left به نقطه شروع گره اول درخت واقعی اشاره می‌کند. توجه داشته باشید اشاره‌گرهای تهی رها شده (loose threads) به گره head اشاره می‌کنند.

نمایش حافظه‌ای کامل درخت شکل ۱۶–۶ در شـکل ۱۷–۶ ارائـه شـده اسـت.

متغیر root به گره Head درخت اشاره می‌کند و $root \rightarrow left$ به شروع گره اول درخت واقعی اشاره می‌کند.



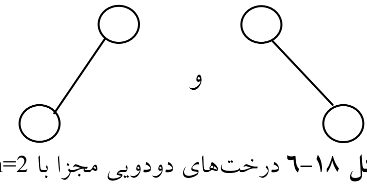
شکل ۱۷-۶ درخت نخی دودوئی

۹-۶ شمارش درخت‌های دودوئی

در اینجا قصد داریم، سه مسئله جدا از هم را که به‌طور جالبی دارای راه حل یکسانی هستند، بررسی کنیم. می‌خواهیم تعداد درخت‌های متمایز با n گره، تعداد جایگشت‌های مجزا اعداد ۱ تا n توسط پشته و در نهایت تعداد ضرب‌های متمایز n+1 ماتریس را مشخص کنیم.

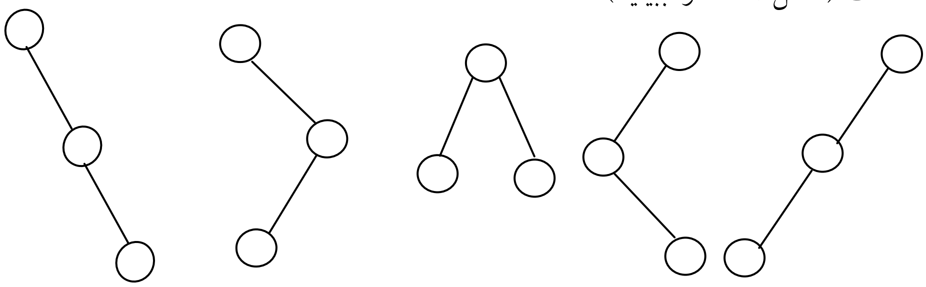
- **درخت‌های دودوئی متمایز**

می‌خواهیم در مورد تعداد درخت‌های حاصل از n گره بحث کنیم. می‌دانیم اگر n=0 یا n=1 باشد، آنگاه فقط امکان ساخت یک درخت دودوئی وجود دارد. اما اگر n=2 باشد، می‌توانیم دو رخت دودوئی داشته باشیم(شکل ۱۸-۶ را ببینید).



شکل ۱۸-۶ درخت‌های دودویی مجزا با n=2

برای n=3 همان‌طور که مشاهده می‌کنید می‌توان 5 درخت دودوئی متمایز ساخت (شکل ۱۹–٦ را ببینید):



شکل ۱۹–٦ درخت‌های دودوئی متمایز با n=3

در اینجا این سؤال مطرح است که، با n گره چند درخت دودوئی می‌توان ساخت؟ قبل از پاسخ به این سؤال، دو مسئله که معادل با این سؤال هستند را بیان می‌کنیم.

- **جایگشت پشته**

فرض کنید اعداد ۱، ۲ و ۳ (n=3) به ترتیب از راست به چپ وارد پشته‌ای شوند. در اینصورت پنج خروجی زیر امکان‌پذیر است (از چپ به راست):

(1,2,3) , (1,3,2) , (2,1,3) , (2,3,1) , (3,2,1)

توجه: ما این مسئله را طبق قضیه‌ای در مسائل حل شده فصل پشته سوم بیان کردیم.

حال سؤال اینجاست که با n عدد ورودی که به ترتیب وارد یک پشته می‌شوند چند حالت خروجی می‌توان داشت؟

- **ضرب ماتریس‌ها**

موضوع دیگری که به‌طور جالبی به دو مسئله قبل مربوط است به‌صورت زیر مطرح می‌شود:

فرض کنید می‌خواهیم حاصلضرب چند ماتریس را به‌دست آوریم:

$$M_1 * M_2 * ... * M_n$$

با توجه به اینکه ضرب ماتریس‌ها شرکت‌پذیر است می توان عمل ضرب را با ترتیب‌های گوناگونی انجام داد. سؤال اینجاست، چند راه برای انجام این حاصلضرب 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:

$$((M_1 * M_3) * M_3) * M_4$$
$$(M_1 * (M_2 * M_3)) * M_4$$
$$M_1 * ((M_2 * M_3) * M_4)$$
$$(M_1 * (M_2 * (M_3 * M_4)))$$
$$((M_1 * M_2) * (M_3 * M_4))$$

به راحتی می‌توان نشان داد که تعداد حالات فوق با هم مساویند و از فرمول زیر محاسبه می‌شوند: $\frac{1}{n+1}\binom{2n}{n}$

به‌عنوان تمرین رابطه بالا را ثابت کنید.

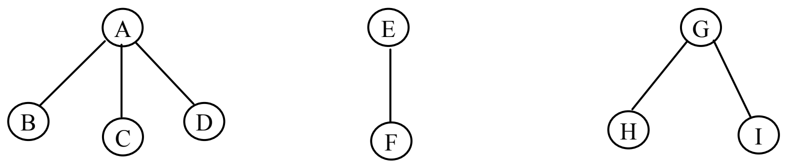
۱۰-٦ جنگل‌ها

یک جنگل مجموعه‌ای مرتب از صفر یا چند درخت متمایز است. به عبارتی دیگر جنگل مجموعه‌ای $N \geq 0$ درخت مجزا است. مفهوم جنگل خیلی نزدیک به مفهوم درخت است. زیرا اگر ریشه درخت را حذف کنیم، جنگل به‌وجود می‌آید.

- **تبدیل جنگل به درخت دودوئی**

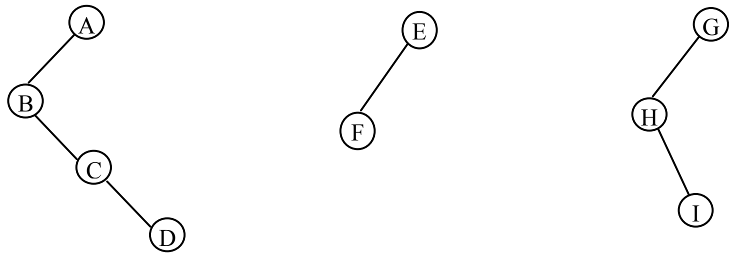
در اینجا قصد داریم الگوریتم تبدیل یک جنگل به درخت دودوئی را ارائه دهیم. برای تبدیل جنگل به درخت دودوئی، نخست هر کدام از درختان جنگل را به‌دست می‌آوریم. سپس تمام درختان دودوئی را از طریق فیلد همزاد گره ریشه به یکدیگر متصل می‌کنیم. برای روشن شدن مطلب کار را با ارائه مثالی شروع می‌کنیم.

فرض کنید جنگل G از سه درخت زیر تشکیل یافته باشد:



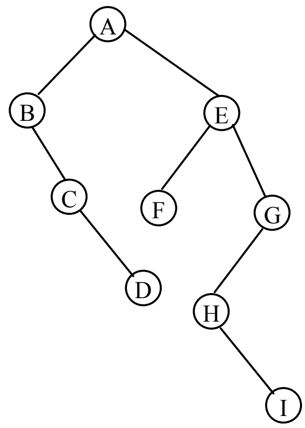
شکل ۲۰-٦ جنگل G

ابتدا این سه درخت را به درخت دودوئی تبدیل می‌کنیم (شکل ۲۱–٦ را ببنید):.



شکل ۲۱–٦ درخت‌های دودوئی حاصل

سپس از چپ به راست ریشه هر درخت را به‌عنوان فرزند راست درخت سمت چپی در نظر می‌گیریم. به‌عبارت دیگر به‌صورت زیر عمل می‌کنیم (شکل ۲۲–٦ را ببینید):



شکل ۲۲–٦ درخت دودوئی حاصل

همان‌طور که ملاحظه می‌کنید، توانستیم جنگل را تبدیل به درخت دودوئی نمائیم.

۱۱–٦ درختان با ساختار مشخص

درختانی که تا حال بررسی کردیم، هیچکدام دارای ساختار مشخصی نبودند. به بیان دیگر درخت بر اساس مقادیر گره‌ها تشکیل نمی‌شد در اینجا قصد داریم با استفاده از مباحثی که در این فصل عنوان شده، درختانی با ساختار معین را ارائه دهیم. همه این درختان براساس مقادیری که در گره‌ها قرار خواهند گرفت، تشکیل می‌شوند.

۱-۱۱-٦ هرم‌ها (HEAPS)

در اینجا نخستین درخت دودوئی با ساختار مشخص را ارائه می‌دهیم. بخش‌های قبل درخت دودوئی کامل تعریف شد. در این بخش با استفاده از درخت دودوئی کامل که قبلا بحث شده، درخت HEAP را بررسی می‌کنیم، که در اکثر کاربردها مخصوصاً مرتب‌سازی اطلاعات مورد استفاده قرار می‌گیرد.

• درخت HEAP (هرم)

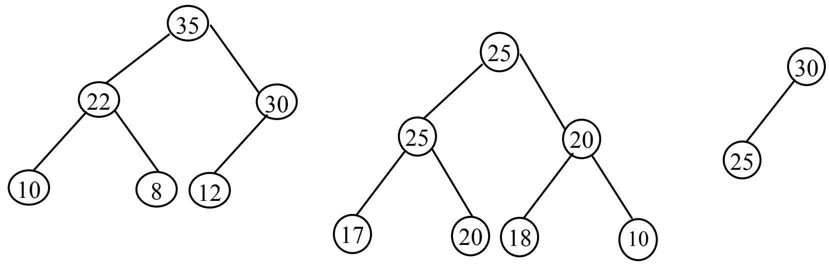
قبل از تعریف درخت HEAP نیازمند بعضی تعاریف هستیم، که از آن جمله می‌توان به تعریف درخت دودوئی کامل و تعریف max tree اشاره کرد. همان‌طورکه قبلاً مشاهده کردید درخت دودوئی کامل را تعریف کردیم. لذا در اینجا قبل از تعریف HEAP درخت max را بررسی می‌کنیم.

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 (یا اصطلاحا HEAP)یک درخت دودوئی کامل است که یک max tree نیز می‌باشد.

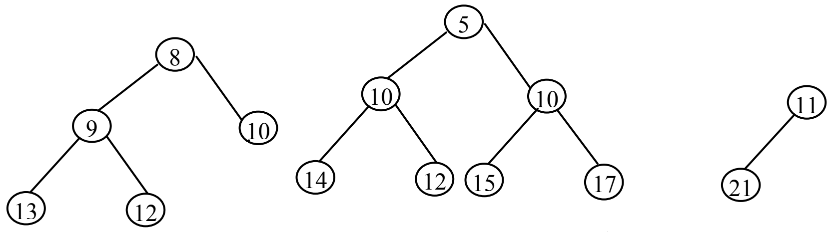
تعریف: min tree درختی است که مقدار کلید هر گره آن بیشتر از مقادیر کلیدهای فرزندانش (اگر وجود داشته باشند) نباشد (کوچک‌تر یا مساوی باشد). Min heap یک درخت دودوئی کامل است که در واقع یک min tree باشد.

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 ۲۳-٦ چند مثال heap و شکل ۲٤-٦ نیز چند نمونه از min heap را ارائه می‌کند.

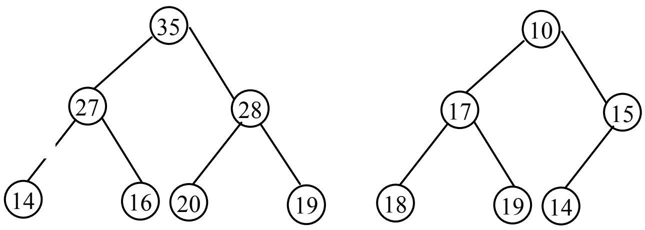


شکل ۲۳-٦ نمایش دو درخت heap



شکل ۲۴-٦ نمایش دو درخت Min heap

حال شکل ۲۵-٦ الف را در نظر بگیرید. این درخت heap نمی‌باشد. زیرا درخت دودوئی کامل نیست. هر چند که خاصیت heap را داراست. همچنین درخت شکل ۲۵-٦ ب نیز درخت min heap نیست زیرا خاصیت min tree را دارا نیست. هر چند که درخت دودوئی کامل می‌باشد.



شکل ۲۵-٦ دو نمونه درخت دودوئی

همان‌طور که ملاحظه کردید در بررسی heap بودن باید دو شرط را با هم بررسی نمائیم.

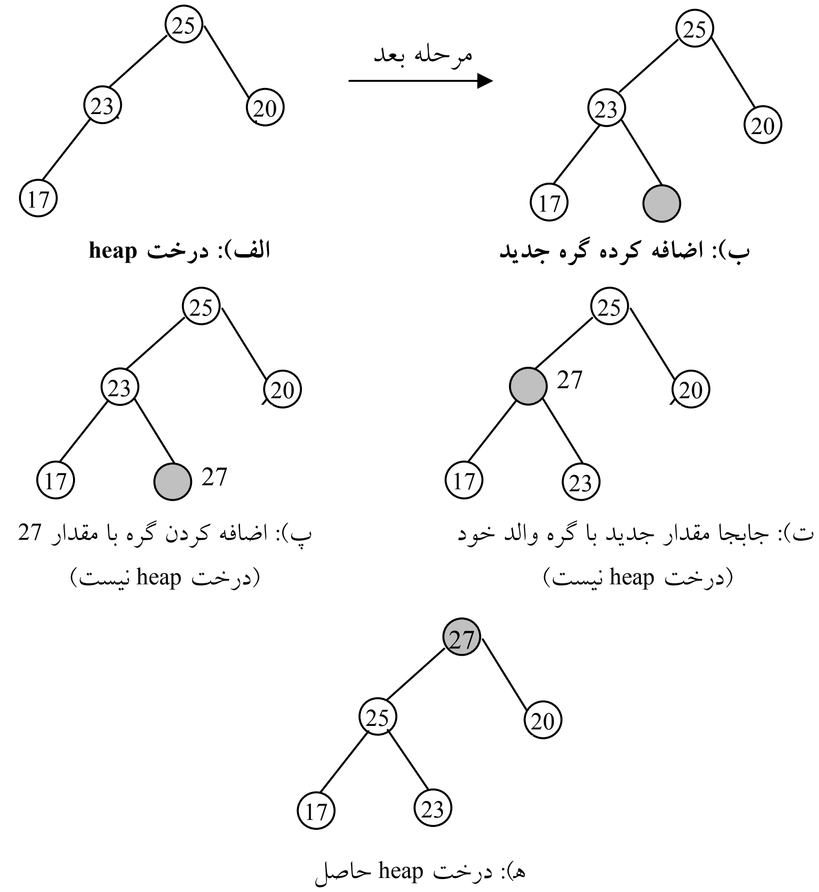
۱-۱-۱۱-٦ درج یک عنصر در heap

در اینجا قصد داریم ساخت درخت heap را مرحله به مرحله تشریح کنیم، لذا در هر مرحله با یک عمل درج در درخت heap مواجه هستیم.در استفاده از درخت هرم، اعمال اضافه کردن یک عنصر در درخت و حذف عنصری از آن باید طوری صورت پذیرد، که درخت به‌صورت هرم (یعنی درخت دودوئی کامل و خاصیت هرم را داراست) باقی بماند. بنابراین پس از عملیات درج کردن و یا حذف کردن، اعمالی برای تنظیم درخت لازم است.

عمل درج کردن به این صورت است که: هر سطح درخت از چپ به راست پر

می‌شود و پس از پر شدن یک سطح، پرکردن سطح بعدی اغاز می‌شود. پس از درج کردن عنصر، خاصیت heap بودن بررسی می‌شود. اگر درخت heap نباشد عمل جابجائی مقادیر گره‌ها انجام می‌شود. در این عمل یک گره در پایین‌ترین سطح تا جایی که لازم باشد با گره پدرش جابجا می‌شود تا خاصیت heap بودن برقرار باشد.

حال می‌خواهیم یک عنصر جدید به درخت heap شکل ۲۶-۶ درج کنیم. برای اینکار همان‌طور که اشاره کردیم، نخست یک گره خالی ایجاد می‌کنیم سپس heap بودن درخت را بررسی می‌کنیم. توجه کنید که، چون درخت دودوئی کامل است بنابراین از چپ به راست ساخته و پر می‌شود.

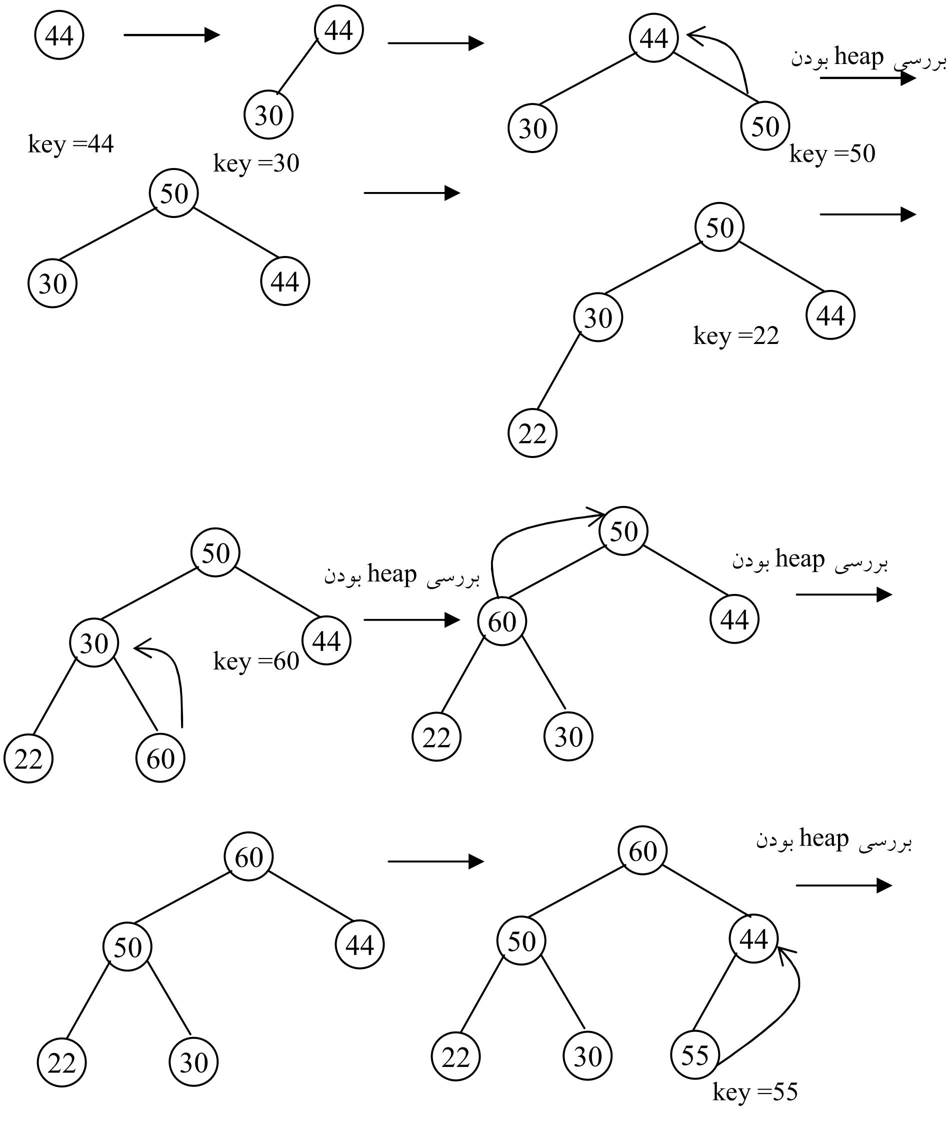


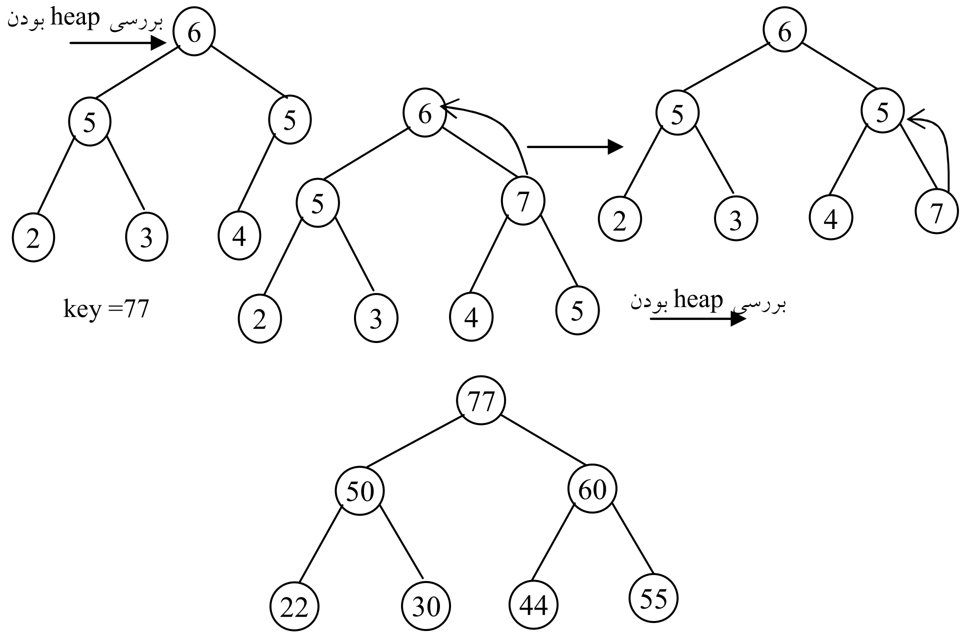
شکل-۲۶ مراحل درج یک گره جدید به درخت heap

حال 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 گره یکی پس از دیگری در درخت خالی انجام داد. شکل ۲۷-٦ (الف) تا (ح) تصویرهای مربوطه Heap را پس از درج کردن هر یک از هشت عنصر نشان می‌دهد.

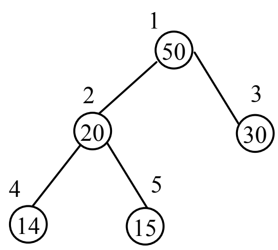




شکل ۲۷-٦ درج تعدای گره در درخت heap

ما هرم‌ها (heap) را با استفاد از آرایه‌ها، البته بدون استفاده از موقعیت 0 آنها، پیاده‌سازی می‌کنیم.

همانگونه که اشاره گردید معمو لاً درخت Heap به‌صورت آرایه پیاده‌سازی می‌شودبه این صورت که مکان هر گره در درخت دقیق اً متناظر با اندیس در یک خانه آرایه می‌باشد.



1	2	3	4	5
50	20	30	14	15

همان‌طور که می‌دانید چون درخت heap درخت دودوئی کامل است لذا، فرزند چپ گره با اندیس i در خانه‌ای با اندیس 2i قرار می‌گیرد و فرزند راست گره با اندیس i در خانه‌ای با اندیس 2i+1 قرار دارد. لذا با توجه به این اصل تابع درج در درخت

heap را به‌صورت زیر ارائه می‌دهیم:

ساختار درخت heap

```
struct   element{
                    int   key;
                    // other fields
                  };
element  heap[MAX_SIZE];
```

حال تابع insert را که عمل درج کردن عنصری به heap حاوی n عنصر را انجام می‌دهد به‌صورت زیر پیاده‌سازی می‌کنیم.

پیاده‌سازی تابع درج کردن عنصری به heap

```
void  insert  (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 درج کردن به درخت heap**

همان‌طور که مشاهده کردید بعد از درج یک عنصر در درخت heap، در بدترین حالت به اندازه ارتفاع درخت، جابجائی صورت می‌گیرد. در تابع بالا حلقه while این موضوع را نشان می‌دهد. از آنجا که درخت heap یک درخت دودوئی کامل با n گره می‌باشد. بنابراین طبق اصول موضوعی بیان شده داریم:

$$n \le 2^h - 1$$

که در آن h ارتفاع درخت می شد لذا طبق رابطه بالا، ارتفاع درخت $\lceil \log_2(n+1) \rceil$

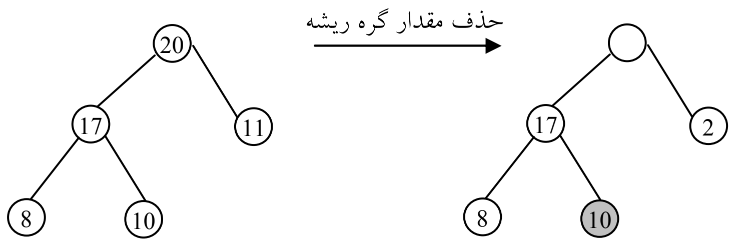
می‌باشد. این بدین معنی است که حلقه while بـه میـزان $O(\log_2 n)$ تکـرار مـی‌شـود. بنابراین پیچیدگی تابع درج کردن یک عنصر جدید به درخت heap برابر با $O(\log_2 n)$ می‌باشد.

۲-۱-۱۱-۶ حذف عنصری از درخت heap

همان‌طور که مشاهده کردید در درخت heap بیشترین مقدار در ریشه قرار داشت. بـه همین دلیل عمل حذف از درخت همواره از ریشه درخت heap صورت می‌گیـرد. بعـد از عمل حذف، سمت راست‌ترین عنصر موجود در پـایین‌تـرین سـطح در ریـشه قـرار می‌گیرو درخت مجدد اً تنظیم می‌شود(خاصیت heap بودن بررسی می‌شود).

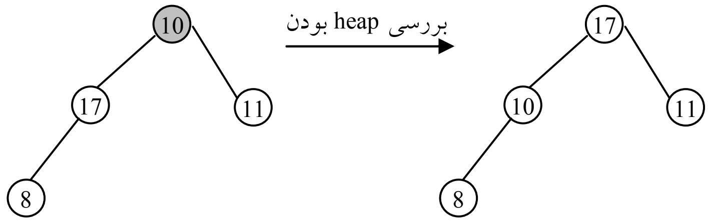
برای روشن شدن مطلب کار را با یک مثال ادامه می‌دهـیم. درخـت heap شـکل ۲۸-۶ (الف) را در نظر بگیرید، می‌خواهیم گره با کلید ۲۰ را که ریشه درخت می‌باشـد حذف کنیم، از آنجایی که heap حاصل دارای تنها چهار عنصر می‌باشد، باید درخـت را به گونه‌ای سازماندهی کنیم که متناظر با درخت دودوئی کامل با چهـار عنـصر گـردد و همچنین خاصیت heap بودن نیز برقرار باشد.

ساختار مطلوب در شکل ۲۸-۶ (ب) آورده شده است. بعـد از حـذف گـره 20، نصر 10 را (که راست‌ترین عنصر در پایین‌ترین سـطح قـرار دارد) در گـره ریـشه قـرار می‌دهیم. شکل ۲۸-۶ (ج) یک درخت دودوئی کامل است ولی درخت حاصل، درخت heap نمی‌باشد. برای برقراری مجدد heap بودن، به سمت پایین حرکت نمـوده و گـره پدر را با فرزندان آن مقایسه و عناصر خارج از ترتیب را تا برقراری مجدد heap جابجـا می‌کنیم. شکل ۲۸-۶ (د) درخت heap نهایی را نشان می‌دهد.



الف) درخت heap

ب) حذف مقدار گره ریشه



ب): مقدار ۱۰ در ریشه قرار می‌گیرد.   ت): درخت heap حاصل

شکل ۲۸-۶ حذف یک عنصر از درختheap

در زیر تابع delete از درخت heap ارائه شده است:

پیاده‌سازی تابع حذف عنصری از heap

```
element  delete (int *n)
{
          int parent , child;
          element item, temp;
          if (heap_Empty (*n))
            {
                    cout<<"The heap is empty";
                  //fprintf(stderr, "The heap is empty\n");
                    exit(1);
            }
          item = heap [1];
          temp = heap [(*n)--];
          parent =1;
          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 ;
 }
```

حال در زیر پیچیدگی تابع بالا را محاسبه می‌کتیم.

- **تحلیل تابع delete**

همان‌طور که مشاهده کردید بعد از حذف یک عنصر از درخت heap، در بدترین حالت به اندازه ارتفاع درخت، جابجائی صورت می‌گیرد (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). در تابع بالا حلقه while این موضوع را نشان می‌دهد. از آنجا که درخت heap یک درخت دودوئی کامل با n گره می‌باشد. بنابراین طبق اصول موضوعی بیان شده داریم:

$$n \le 2^h - 1$$

که در آن h ارتفاع درخت می‌باشد. لذا طبق رابطه بالا از آنجایی که ارتفاع یک درخت heap با n عنصر برابر با $[\log_2(n+1)]$ می‌باشد. حلقه while، به میزان $O(\log_2 n)$ مرتبه تکرار می‌گردد. بنابراین پیچیدگی تابع حذف برابر با $O(\log_2 n)$ می‌باشد.

همان‌طور که ملاحظه کردید ساختار داده heap، ساختار داده‌ای با زمان‌های درج و حذف لگاریتمی می‌باشد، که زمان‌هائی پذیرفته شده می‌باشند. اما برای جستجوی یک عنصر این ساختار داده الگوی خاصی را دنبال نمی‌کند. و زمان آن از مرتبه $O(n)$ می‌باشد. همچنین حذف دلخواه از درخت heap براحتی امکانپذیر نیست و از مرتبه زمانی $O(n)$ می‌باشد، که زمان مناسبی نیست.

با این حال این ساختار داده کاربردهای زیادی دارد. که از آن جمله می‌توان به کاربرد آن در مرتب‌سازی و صف‌اولویت اشاره کرد.

۳-۱-۱۱-٦ صف اولویت (priority queue)

یکی از کاربردهای هرم‌ها برای پیاده سازی صف اولویت‌دار می‌باشد. همانطور که قبلاً اشاره 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. ولی در موقع حذف عنصر با اولویت بالا حذف می‌شود.

همان‌طور که اشاره کردیم، آرایه ساده‌ترین 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 $\theta(1)$ است. برای حذف، ابتدا باید عنصر با بزرگ‌ترین (یا کوچک‌ترین) اولویت را جستجو و سپس آن را حذف کنیم. لذا زمان جستجو برابر با $O(n)$ می‌باشد و زمان جابجائی عناصر دیگر برابر $O(n)$ می‌باشد. استفاده از لیست پیوندی غیرمرتب زمان اجرای برنامه را تا حدودی بهبود می‌بخشد. می‌توانیم عنصری را به ابتدای لیست در $\theta(1)$ اضافه نمائیم. ولی همچنان برای پیدا نمودن عنصری با بزرگ‌ترین کلید باید جستجو کنیم، با این نمایش، زمان مورد نیاز جهت تغییر مکان عناصر، گرفته بنابراین زمان حذف برابر با θ(n) می‌باشد.

نمایش صف اولویت به‌صورت هرم (heap) این امکان را فراهم می‌سازد که هم درج و هم حذف را در زمان $O(log_2 n)$ انجام دهیم و همین امر از آن نمایش مطلوبی می‌سازد و ارجحیت آن را نسبت به بقیه نشان می‌دهد.

ساختار داده	درج	حذف
آرایه غیرمرتب	θ(۱)	θ(n)
لیست پیوندی غیرمرتب	θ(۱)	θ(n)
آرایه مرتب	θ(n)	θ(۱)
لیست پیوندی مرتب	θ(n)	θ(۱)
هرم (max heap)	O(log۲ n)	$O(\log_2 n)$

زمان انواع نمایش صف اولویت

- **کاربرد heap در مرتب کردن اطلاعات**

فرض کنید آرایه A با N عنصر داده شده است. الگوریتم Heap sort که عناصر لیست A را مرتب می‌کند از دو مرحله زیر تشکیل یافته است:

مرحله اول: یک درخت heap از عناصر آرایه A بسازید.

مرحله دوم: حذف از درخت heap به تعداد عناصر درخت.

از آنجا که ریشه درخت heap همواره بزرگ‌ترین گره درخت است، مرحله دوم، عنصرهای آرایه A را به ترتیب نزولی حذف می‌کند.

۲-۱۱-۶ درخت‌های جستجوی دودوئی (Binary Search Tree)

همان‌طور که در بخش قبل مشاهده کردید درخت heap در برخی کاربردها از کارائی لازمی برخوردار نیست. بنابراین در این بخش یکی از مهمترین ساختمان داده‌های علم کامپیوتر یعنی درخت جستجوی دودوئی(BST) را مورد بحث و بررسی قرار می‌دهیم، که نسبت به هر ساختار داده‌ای که تا حال مطرح شده، کارائی بهتری دارد. 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 به دلخواه 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 $O(\log_2 n)$ پیدا کرد. اما اضافه کردن و حذف کردن پرهزینه است.

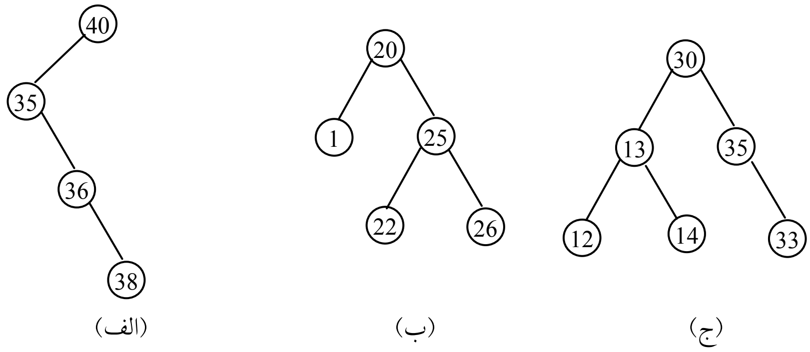
ب) لیست پیوندی: در اینجا به سادگی می‌توان عناصر را اضافه یا حذف کرد. اما در این روش جستجوی عنصر و پیدا کردن آن پرهزینه است. چون باید از جستجوی خطی با زمان اجرای $O(n)$ استفاده کند.

حال درخت جستجوی دودوئی(BST) را به‌صورت زیر تعریف می‌کنیم:

تعریف: یک درخت جستجوی دودوئی، درخت دودوئی است که ممکن است تهی باشد. اگر درخت تهی نباشد دارای خصوصیات زیر است:

- 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.
- زیر درختان چپ و راست نیز خود درختان جستجوی دودوئی می‌باشند.

چند نمونه از درختان دودوئی در شکل ۲۹-۶ 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 کلید ۲۵ باید بزرگ‌تر از آن باشد درحالی‌که مقدار آن ۳۳ می‌باشد.



شکل ۲۹-۶ نمونه‌ای از درخت‌های جستجوی دودوئی

از آنجایی که درخت جستجوی دودوئی شکل خاصی از یک درخت دودوئی است، لذا عملگرها برای یک درخت جستجوی دودوئی تفاوتی با عملگرهایی‌که قبلاً برای یک درخت دودوئی به کار گرفتیم، ندارد. تمام عملگرهای درخت‌های دودوئی که قبلاً مورد بحث قرار گرفت نیز برای درخت‌های جستجوی دودوئی نیز اعمال می‌شود. به‌عنوان مثال می‌توانیم از پیمایش‌های postroder, preorder, inorder بدون هیچگونه تغییری استفاده کنیم. به این اعمال می‌توان، درج، حذف و جستجو را نیز با کمی دقت اضافه نمود.

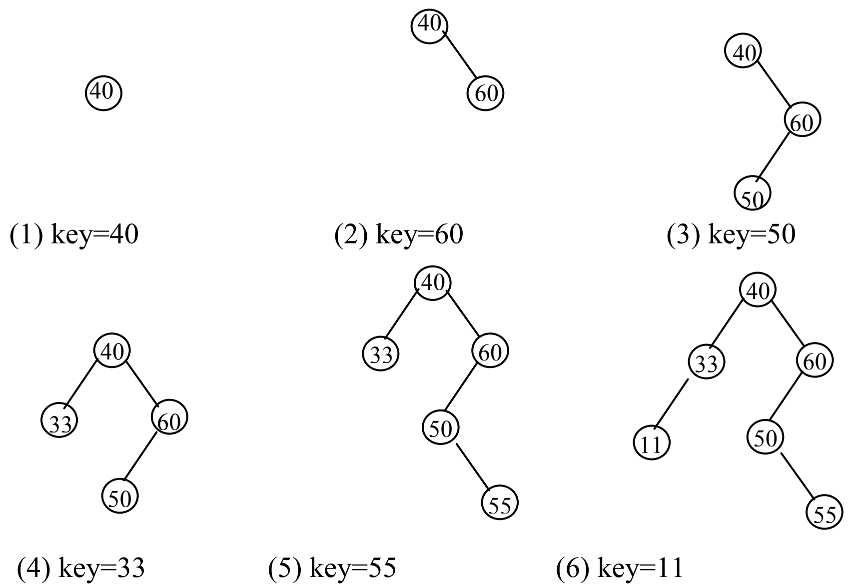
برای درک بهتر درخت BST در زیر ساخت چنین درختی را مرحله به مرحله نشان می‌دهیم:

فرض کنید شش عدد زیر به ترتیب در یک درخت جستجوی دودوئی خالی اضافه شده است:

40 , 60 , 50 , 33 ,   55, 11

شکل ۳۰-۶ شش مرحله از درخت را نشان می‌دهد.

تاکید می‌کنیم که اگر شش عدد داده شده با ترتیب مختلف داده شده باشد، آنگاه ممکن است، درخت‌های حاصل نیز با هم فرق کنند و عمق مختلفی داشته باشند.



شکل ۳۰-٦: ساخت مرحله به مرحله BST

ساختار هر گره در درخت جستجوی دودوئی همان‌طور که در درختـان دودوئـی اشاره کردیم از فیلدهای اطلاعات و اشاره‌گر بـه زیـر درخـت چـپ و راسـت تـشکیل می‌شود. این ساختار به‌صورت زیر می‌باشد:

نحوه تعریف گره در BST
`struct node { node *left ; int info; node * right ; };`

```
struct   node {
                node   *left ;
                int     info;
                node   * right ;
            };
```

در بخش‌های بعد عملگرهای این ساختار داده را بررسی می‌کنیم.

۱-۲-۱۱-٦ جستجوی یک عنصر در درخت جستجوی دودوئی

از آنجائی که تعریف درخت جستجوی دودوئی را به‌صورت بازگشتی انجام دادیم، لـذا

بیان و ارائه یک روش جستجوی بازگشتی نیز ساده می‌باشد. اگر بخواهیم در درخت BST به جستجوی عنصری با کلید key بگردیم، برای اینکار 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 به‌صورت بازگشتی جستجو می‌کند.

تابع جستجوی یک عنصر در BST

```
node    *search (node  *tree ,  int   key )
{
/*return a pointer to the node that contains key, if there isn't  such node,
return Null.*/
        if (!tree)
            return NULL ;
        if (key = =  tree → inf o )
            return tree ;
        if (key <  tree → inf o )
                return search (  tree → left , key) ;
        return search  (  tree → right   , key) ;
}
```

- **پیچیدگی الگوریتم جستجوی عنصر در BST**

فرض کنید در یک درخت جستجوی دودوئی T، می‌خواهیم یک عنصر 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 $A_1, A_2, ..., A_N$ 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 به‌وجود 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 $f(n) \in O(\log_2 n)$.

۲-۲-۱۱-٦ درج عنصری در درخت جستجوی دودوئی

فرض کنید T یک درخت جستجوی دودوئی باشد، می‌خواهیم عنصری با مقدار key را در درخت درج کنیم. در واقع، جستجو و وارد کردن یک عنصر، تنها با یـک الگـوریتم جستجو و وارد کردن انجام می‌شود.

فرض کنید عنصر key داده شده است. الگـوریتم زیـر محـل key را در درخـت جستجوی دودوئی پیدا می‌کند. اگر جستجو ناموفق باشد، key را در محلی که جـستجو خاتمه پیدا یافته است، درج می‌کنیم. برای اینکار الگوریتم مراحل زیر را انجام می‌هد:

الف) key را با info، که مقدار کلید گره ریشه است، مقایسه کنید:

i) اگر key < info باشد، به‌طرف زیر درخت چپ ریشه حرکت کنید.

ii) اگر key > info باشد، به‌طرف زیر درخت راست ریشه حرکت کنید.

ب) مرحله (الف) را تکرار کنید تا یکی از حالتهای زیر اتفاق بیفتد:

i) گره با کلیـد info را وقتـی key = info اسـت ملاقـات کنیـد. در ایـن حالـت جستجو موفق است.

ii) یک زیردرخت خالی را ملاقات کنید که بیان می‌کند جستجو موفق نیـست و key را به جای زیردرخت خالی اضافه کنید.

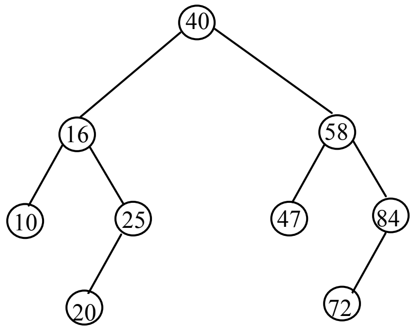
بنابراین الگوریتم اضافه کردن شبیه الگوریتم جستجو است و فقط باید به انتهـای الگوریتم تابع زیر را اضافه کنید:

تابع درج عنصری در درخت جستجوی دودوئی

```
void  insert (node  *tree , int  key)
{
   node  *ptr;
   ptr = getnode();
   ptr -> info = key  ;
   ptr -> left = NULL;
   ptr -> right = NULL;
  if ( tree -> info > key )
```

```
        tree → left = ptr;
   else if ( tree → info < key )
        tree → right = ptr;
}
```

مثال ۱۳-۶: درخت جستجوی دودوئی T شکل ۳۱-۶ را در نظر بگیرید.



شکل ۳۱-۶ درخت جستجوی دودوئیT

T یک درخت جستجوی دودوئی است. یعنی هر گره N در T از هر عدد زیردرخت چپ آن بزرگ‌تر و از هر عدد زیردرخت راست آن کوچک‌تر است.

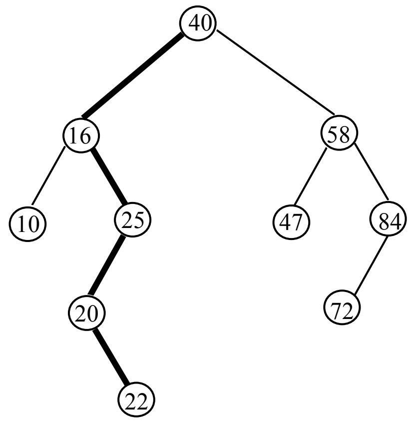
فرض کنید عدد 37 جای‌گزین عدد 25 شود آنگاه T همچنان یک درخت جستجوی دودوئی باقی خواهد ماند ولی اگر عدد 42 را جایگزین عدد 25 کنیم T یک درخت جستجوی دودوئی نخواهد بود، چون در زیردرخت چپ هیچ عددی نباید بزرگ‌تر از آن باشد.

حال فرض کنید می‌خواهیم عدد ۲۲ را در درخت درج کنیم عملیات موردنظر به‌صورت زیر خواهد بود:

- key=22 را با ریشه یعنی 40 مقایسه می‌کنیم چون از آن کوچک‌تر است به طرف فرزند چپ 40 یعنی 16 می‌رویم.
- Key=22 را با 16 مقایسه می‌کنیم. چون از آن بزرگ‌تر است به طرف فرزند راست 16 یعنی 25 می‌رویم.
- Key=22 را با 25 مقایسه می‌کنیم. چون از آن کوچک‌تر است به طرف فرزند

چپ 25 یعنی 20 می‌رویم.

- Key=22 را با 20 مقایسه می‌کنیم چون از آن بزرگ‌تر است به طرف فرزند راست 20 می‌رویم و چون 20 فرزند راست ندارد، 22 را به‌عنوان فرزند راست به درخت اضافه می‌کنیم.



شکل ۳۲-٦ درخت جستجوی دودوئی T بعد از عمل درج

- **تحلیل الگوریتم insert**

برای درج یک عنصر ابتدا باید بوسیله جستجو محل درج عنصر جدید مشخص شود و سپس عنصر جدید درج شود. زمان لازم برای جستجوی کلید key برابر با h(ارتفاع درخت) می‌باشد و بعد از جستجو، عمل درج نیاز به زمان دارد. بنابراین زمان کل مورد نیاز insert برابر با O(h) می‌باشد.

۳-۲-۱۱-٦ 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 محل گره N که حاوی عنصر key است و همچنین محل پدر آن P(N) را پیدا می‌کند. روشی که با آن N از درخت حذف می‌شود بسته به تعداد فرزندهای N از سه حالت زیر تشکیل می‌شود:

- گره N فرزندی ندارد. در این صورت با جایگزین شدن محل گره N در گره

پدر P(N) به وسیله اشاره‌گر Null، گره N از درخت حذف می‌شود.

• گره N دقیقاً یک فرزند دارد. در این صورت با جایگزین شدن محل گره N در P(N) به‌وسیله محل تنها فرزند گره N، گره N از درخت حذف می‌شود (یعنی فرزند گره N جایگزین خود گره N می‌شود.)

• گره N دو فرزند دارد. فرض کنید S نمایش گره بعدی پیمایش Inorder گره N باشد. آنگاه می‌توان گفت S فرزند چپ ندارد (به‌عنوان تمرین ثابت کنید). در این‌صورت با حذف S از T (با استفاده از حالت اول یا دوم) و سپس جانشین کردن گره S به جای گره N در درخت T، گره N از T حذف می‌شود.

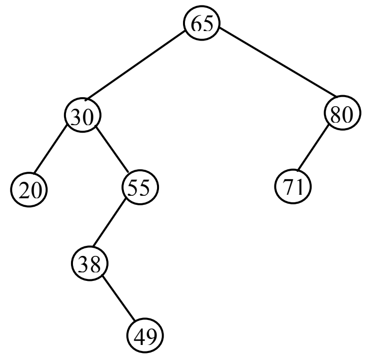
سه حالت الگوریتم حذف از BST را می‌توان به‌صورت زیر خلاصه کرد:

• اگر گره N فرزند نداشته باشد براحتی قابل حذف است.

• اگر گره N فقط یک فرزند داشته باشد، فرزندش جایگزین آن می‌شود.

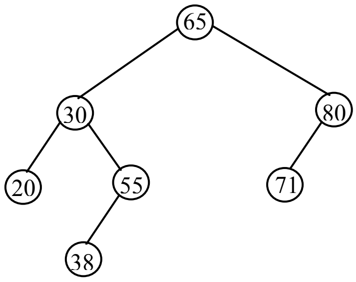
• اگر گره N دو فرزند داشته باشد، آنگاه گره بعد از N در پیمایش inorder جایگزین N می‌شود.

برای روشن شدن مطلب کار را با ارائه یک مثال ادامه می‌دهیم. درخت جستجوی دودوئی شکل ۳۳-۶ را در نظر بگیرید.



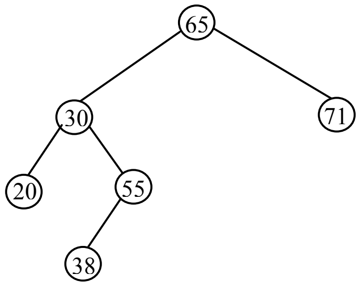
شکل ۳۳-۶ درخت جستجوی دودوئی

فرض کنید بخواهیم گره ۴۹ را از درخت حذف کنیم. با توجه به اینکه گره ۴۹ فرزندی ندارد، کافی است اشاره‌گر پدر آن (یعنی ۳۸) را Null کنیم. شکل ۳۴-۶ این درخت را پس از حذف گره ۴۹ نشان می‌دهد.



شکل ۳٤–٦ درخت جستجوی دودوئی بعد از حذف گره ٤۹

حال فرض کنید بخواهیم، گره ۸۰ را از درخت حذف کنیم. با توجه به اینکه گره ۸۰ تنها یک فرزند دارد، بنابراین گره ۸۰ حذف و فرزند آن یعنی گره ۷۱ جایگزین آن می‌شود. شکل ۳۵–٦ این درخت را پس از حذف گره ۸۰ نمایش می دهد.



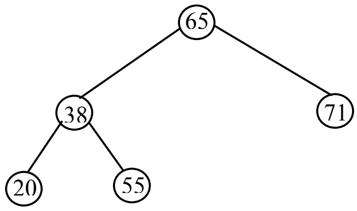
شکل ۳۵–٦ درخت جستجوی دودوئی بعد از حذف گره ۸۰

حال می‌خواهیم گره ۳۰ را از درخت حذف کنیم. توجه کنید که گره ۳۰ دو فرزند دارد. ابتدا پیمایش میانوندی (inorder) درخت را به‌دست می‌آوریم:

20 30 38 55 65 71

ملاحظه می‌کنید که گره ۳۸، گره بعدی گره ۳۰ در پیمایش میانوندی می‌باشد. بنابراین طبق الگوریتم، باید گره ۳۸ را جانشین گره ۳۰ بکنیم

بنابراین، ابتدا گره ۳۸ را حذف کرده و سپس آن را به جای گره ۳۰ قرار می‌دهیم. تاکید می‌کنیم که جانشینی گره ۳۸ به جای گره ۳۰ درحافظه تنها با تغییر اشاره‌گرها انجام می‌گیرد نه با جابجایی محتوای یک گره از یک محل به محل دیگر. شکل ۳٦–٦ درخت را پس از حذف گره ۳۰. نمایش می



شکل ۳۶-۶ درخت پس از حذف گره ۳۰

۴-۲-۱۱-۶ حذف عناصر تکراری به‌عنوان کاربردی از BST

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 روش ساده برای حل این مسئله به شرح زیر است:

الگوریتم اول: عناصر را از a_1 تا a_N یعنی از چپ به راست بخوانید.

(i) هر عنصر a_k را با $a_1, a_2, ..., a_{k-1}$ مقایسه کنید. (یعنی a_k را با عناصری که قبل از a_k هستند مقایسه کنید)

(ii) اگر a_k در بین $a_1, a_2, ..., a_{k-1}$ 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 حذف خواهند شد.

- **پیچیدگی زمانی الگوریتم اول**

پیچیدگی زمانی الگوریتم اول، به وسیله تعداد مقایسه‌ها تعیین می‌شود. هر مرحله بررسی a_k، به‌طور تقریبی به $k-1$ مقایسه احتیاج دارد. چون a_k با عنصرهای $a_1, a_2, ..., a_{k-1}$ مقایسه می‌شود، بنابراین $T(n)$ تعداد مقایسه‌های مورد نیاز در الگوریتم، تقریباً برابر است با:

$$۱ + ۲ + ۳ + ... + (n-۲) + (n-۱) = \frac{n(n-۱)}{۲} = O(n^۲)$$

برای مثال برای n=1000 عنصر، الگوریتم 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 کند.

الگوریتم دوم: با استفاده از عناصر $a_N,...,a_2,a_1$ 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 دوم 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 دوم تقریباً به $n\log_2 n$ مقایسه نیاز دارد.

برای مثال برای $n=1000$ عنصر، الگوریتم دوم مستلزم تقریباً ۱۰۰۰۰ مقایسه است درحالی‌که در الگوریتم اول تعداد مقایسه‌ها برابر ۵۰۰۰۰۰ است. قابل توجه است که در بدترین حالت (حالتی که درخت جستجوی دودوئی به‌صورت مورب باشد) تعداد مقایسه‌های الگوریتم‌های اول و دوم با هم برابر هستند.

۳-۱۱-٦ درخت‌های انتخابی (selection trees)

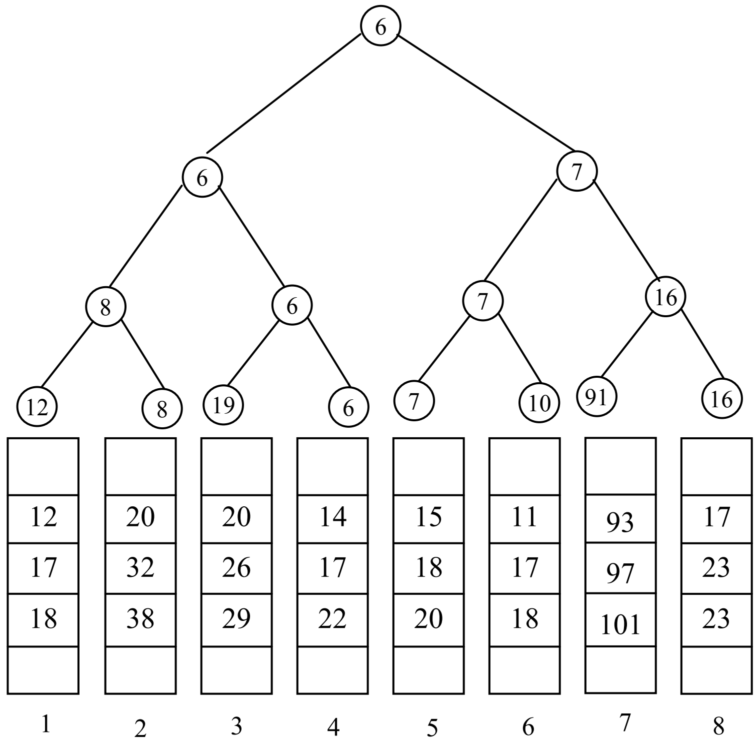
فرض کنید k آرایه مرتب شده غیرنزولی از عناصر داریم و می‌خواهیم این k آرایه را در یک آرایه واحد ادغام کنیم. n مجموع کل خانه‌های k آرایه می‌باشد. ساده ترین روش برای ادغام k آرایه مرتب، آن است که عنصر اول تمام آرایه‌ها را با هم مقایسه کرده و کوچک‌ترین آنها را به‌دست آوریم. این عمل به k-1 مقایسه نیاز دارد. کوچک‌ترین عنصر را در خروجی چاپ می‌کنیم چون در هر بار تکرار الگوریتم، k-1 مقایسه نیاز است، از طرف دیگر، چون در کل n عنصر در آرایه‌ها موجود است، بنابراین، مرتبه اجرایی الگوریتم $O(n.k)$ می‌باشد.

مسئله فوق را می‌توان با استفاده از درختان انتخابی حل کرد. ایده درخت انتخابی، تعداد مقایسه‌های لازم را کاهش می‌دهیم و راه‌حل کاراتری نیز می‌باشد.

تعریف: درخت انتخابی، یک درخت دودوئی است که مقدار هر گره آن، کوچک‌تر از دو فرزند خود می‌باشد. بنابراین، گره ریشه نشان‌دهنده کوچک‌ترین گره در درخت می‌باشد.

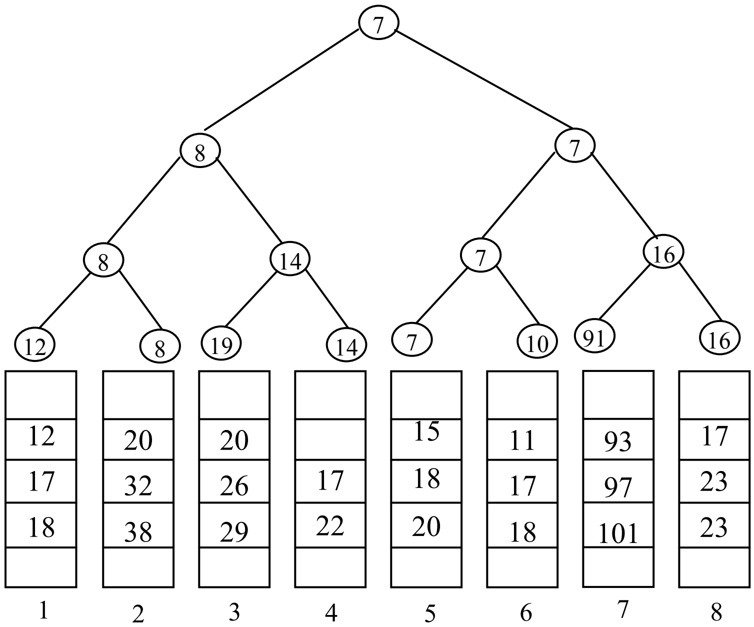
حال برای درک بهتر درختان انتخابی از شکل ۳۷-٦ استفاده می‌کنیم. این شکل برای حالت k=8 یک درخت انتخابی را نشان می‌دهد. در این درخت هر گره غیربرگ

نشان‌دهنده گره با مقدار کوچک‌تر است و گره ریشه کوچک‌ترین کلید را نشان می‌دهد.



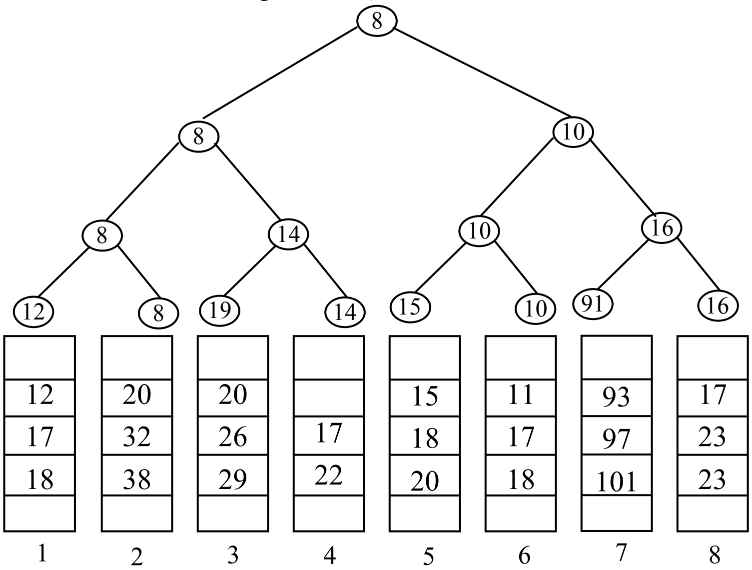
شکل ۳۷–٦ درخت انتخابی با k=8

همان‌گونه که مشاهده می‌کنید ابتدا از لیست اول کوچک‌ترین عنـصر یعنـی گـره با مقدار 12 و از لیست دوم کوچک‌ترین عنصر یعنی گره با مقدار8 را انتخاب کرده، بـا هم مقایسه می‌کنیم و عنصر کوچک‌تر یعنی 8 به بالا انتقال داده می‌شود. سـپس گـره بـا مقدار 19 و گره با مقدار 6 به ترتیب از لیست‌های سوم و چهارم انتخاب شده و بـا هـم مقایسه می‌شوند و عنصر کوچک‌تر یعنی گره 6 به طرف بالا انتقال داده می‌شـود و الـی آخر. سپس در سطح بعدی گره با مقدار 8 با گره 6 مقایسه شده و عنصر کوچک‌تـر بـه بالا انتقال داده می‌شود و الی آخر. در نهایت 6 کوچک‌ترین گره خواهد شد و به‌عنـوان اولین خروجی خواهد بود. چون کوچک‌ترین گره از آرایه شماره 4 می‌باشد در مرحلـه بعدی خانه دوم از آرایه شماره چهار یعنی 14 را به جای گیدرینظو میا حـل بالا را تکرار می‌کنیم. شکل ۳۸–٦، مرحله دوم الگوریتم را نشان می‌دهد.



شکل ۳۸-۶ مرحله دوم الگوریتم

همان‌طور که مشاهده می‌کنید خروجی این مرحله عدد ۷ از لیست پنجم می‌باشد. بنابراین در مرحله بعدی عنصر ۱۵ انتخاب می‌شود. شکل۳۹-۶ مرحله بعدی را نمایش می‌دهد.



شکل-۳۹ مرحله دوم الگوریتم

خروجی مرحله سوم عدد ۸ می‌باشد. با ادامه مراحل کار لیست مرتب که حاصل ادغام هشت لیست می‌باشد، به‌عنوان خروجی ارائه می‌گردد.

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. غالبا این درختان برای پیاده‌سازی بازی به‌کار می‌روند. در هر مرحله از این درخت، خروجی برنده آن مرحله می‌باشد و در نهایت برنده کل بازی توسط این نوع درخت مشخص می‌شود.

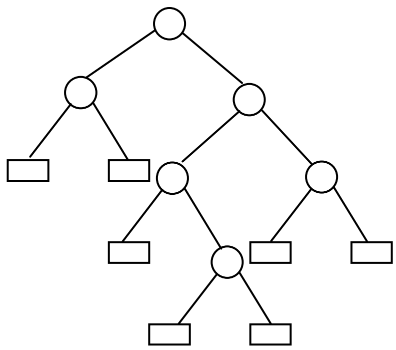
۱۲-۶ الگوریتم‌هافمن

در این بخش قصد داریم الگوریتم‌هافمن را برای حل مسائل ارائه دهیم. برای اینکار نخست تعاریفی را ارائه می‌دهیم.

تعریف ۲- درخت (یا یک درخت دودوئی گسترش یافته): درخت دودوئی است که در آن هر گره 0 یا 2 فرزند دارد. گره‌هایی که 0 فرزند دارند گره‌های خارجی (External) نام دارند و همچنین گره‌هایی که 2 فرزند دارند، گره‌های داخلی(Internal) نام دارند.

شکل۶-٤ 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:

$$N_E = N_I + 1$$



شکل ٤٠-٦ نمونه ۲-درخت

در شکل فوق $N_E = 7, N_I = 6$ می‌باشد.

حال طول مسیر را برای گره‌های داخلی و خارجی یک ۲- درخت برای تکمیل

بحث، تعریف می‌کنیم.

طول مسیر خارجی (L_E):طول مسیر خارجی L_E یک ۲-درخت T به صورت مجموع طول مسیرهای از ریشه درخت تا گره های خارجی است.

طول مسیر داخلی L_I به‌طور مشابه تعریف می‌شود که در آن به جای گره‌های خارجی گره های داخلی استفاده می‌شود.

طبق تعریف بالا برای درخت شکل ٤٠-٦ خواهیم داشت:

$$L_I=0+1+1+2+2+3=9 \quad , \quad L_E=2+2+3+4+4+3+3=21$$

فرض کنید T یک ۲- درخت با n گره خارجی باشد و به هر گره خارجی یک وزن (غیر منفی) نسبت داد شده باشد. در اینصورت طول مسیر وزن دار (خارجی) درخت T به‌صورت زیر تعریف می‌شود:

$$P=W_1L_1 + W_2L_2 + \ldots + W_nL_n$$

که در آن W_i و L_i به ترتیب وزن و طول مسیر یک گره خارجی N_i است.

حال فرض کنید یک لیست با n وزن داده شده است:

$$w_۱, w_۲, \ldots, w_n$$

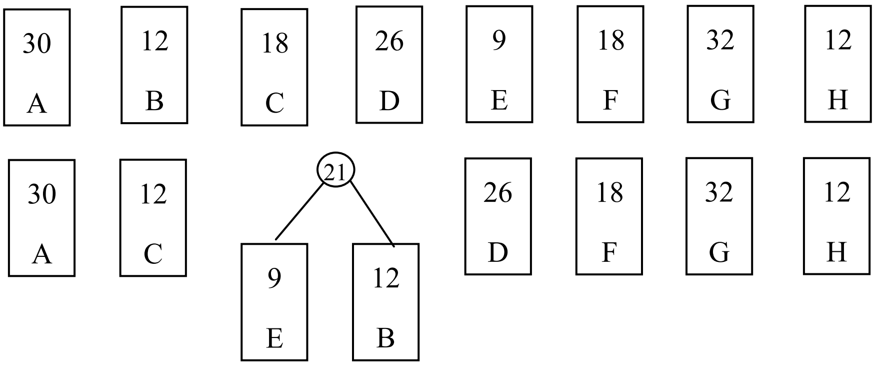
می‌خواهیم از میان تمام ۲- درخت دارای n گره خارجی و وزن دار یک درختT با حداقل طول مسیر وزن دار به‌دست آوریم.

هافمن الگوریتمی برای پیدا کردن چنین درخت Tای ارائه می‌دهد. که ما اکنون آن را بیان می‌کنیم. برای درک بهتر کار را با یک مثال شروع می‌کنیم.

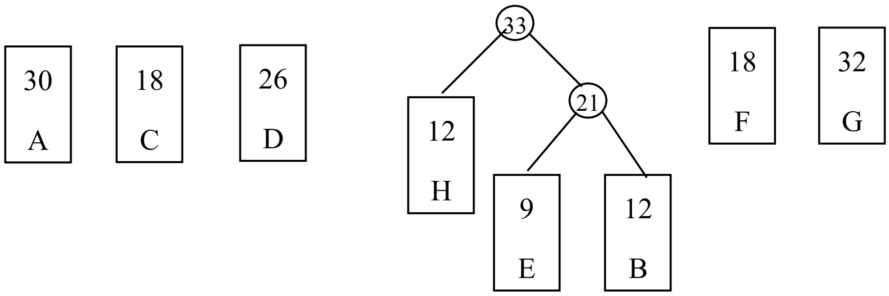
فرض کنید H , G , F, E, D, C, B, A، کارکترهای یک متن هستند. که تعداد تکرار هر کدام از آنها در متن در زیر مشخص شده است:

عنصر:	A	B	C	D	E	F	G	H
وزن :	30	12	18	26	9	18	32	12

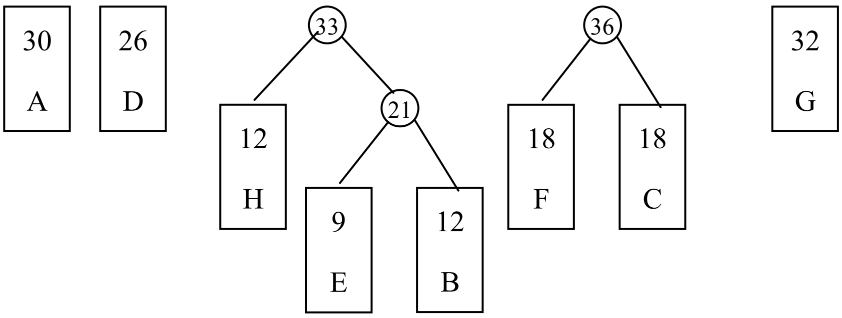
شکل ٤٠-٦ (الف) تا (د) چگونگی ساخته شدن درخت T را با حداقل طول مسیر وزن داده شده با استفاده از اطلاعات بالا و الگوریتم‌هافمن نشان می‌دهد در هر مرحله دو درختی که ریشه کمتری دارند با هم ترکیب می‌کنیم وزن آنها را با هم جمع می‌کنیم.)



(الف)

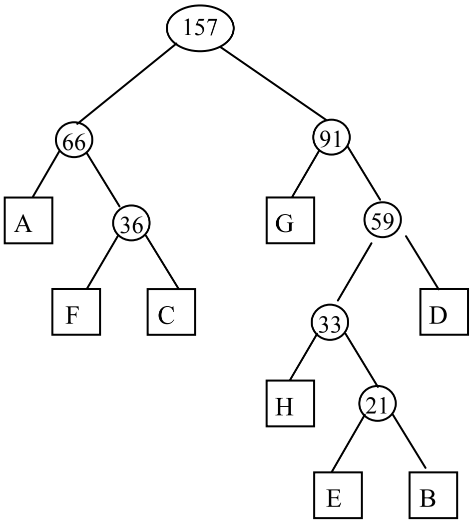


(ب)



(ج)

اگر همین‌طور ادامه دهیم شکل نهایی به‌صورت زیر می‌شود:



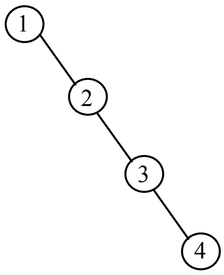
(د)

شکل ۴۱-۶ درخت‌هافمن

برای مطالعه بیشتر به کتاب طراحی الگوریتم انتشارات پیام نور مراجعه فرمائید.

۱۳-۶ درخت جستجوی متعادل

درخت‌های جستجوی دودوئی، ابزار مناسبی برای جستجوی عناصر می‌باشند. علاوه بر این، ترکیب‌های مختلفی از یک مجموعه از داده‌ها، درخت‌های دودوئی مختلفی را به‌وجود می‌آورند. که زمان جستجوی عناصر در آنها یکسان نیست. اگر به اندازه کافی دقت نشود عمق یک درخت جستجوی دودوئی (BST)تواند n عنصر بزرگی ی n باشد. به‌عنوان مثال درخت جستجوی دودوئی برای عناصر ٤، ۳، ۲، ۱ به‌صورت زیر خواهد بود.

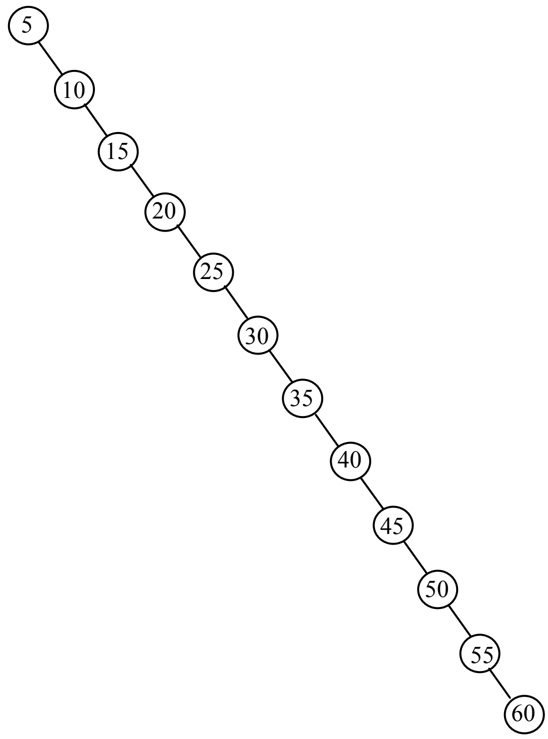


که دارای عمق ٤ می‌باشد. اما اگر دودوئی به‌صورت تصادفی باشد، به‌طور متوسط عمق درخت جستجوی دودوئی $O(\log_2 n)$ خواهد بود. بنابراین هدف این است که درخت‌های جستجوی دودوئی را طوری ساخت که زمان جستجو در آنها به حداقل برسد. یا به‌عبارت دیگر عمق درخت $O(\log_2 n)$ باشد. درخت‌های جستجوی با بیشترین عمق $O(\log n)$ را درخت‌های جستجوی متعادل می‌نامند. در این درخت‌ها عمل جستجو، درج، حذف در زمان $O(h)$ (h ارتفاق یا عمق درخت) انجام می‌گیرد. قابل توجه‌ترین موارد در بین این نوع درخت‌ها، درخت‌های AVL، ۳-۲، red-black می‌باشد.

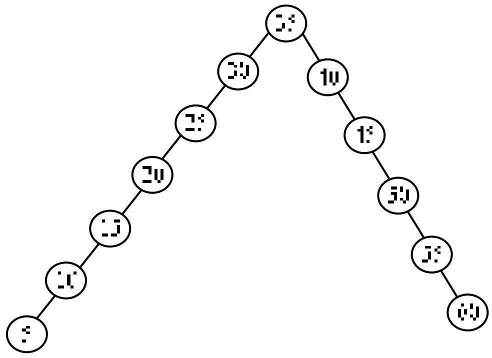
داده‌های زیر را به‌عنوان ورودی درنظر بگیرید:

5 10 15 20 25 30 35 40 45 50 55 60

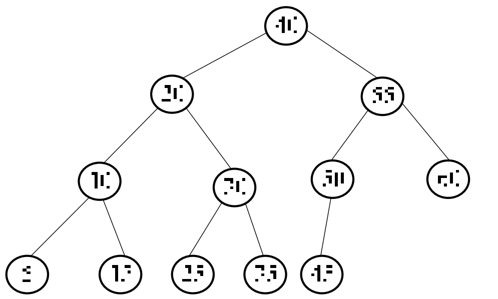
با استفاده از جایگشت‌های مختلفی از این داده‌ها، می‌توان !۱۲ درخت جستجوی دودوئی به‌دست آورد که بعضی از آنها را در شکل ۴۲-۶ می‌بینید:



(الف)



(ب)



(ج)

شکل ۴۲-٦ درخت جستجوی دودوئی

اگر T میانگین زمان جستجوی درخت باشد، داریم:

$$T = \frac{\sum_{i=1}^{n} t_i}{n}$$

که در آن t_i تعداد مقایسه‌ها برای گره iام و n برابر تعداد گره‌های درخت جستجوی دودوئی می‌باشد. اگر T را برای درخت‌های شکل ۴۲-٦ محاسبه کنیم خواهیم داشت:

- ❖ زمان جستجوی درخت (الف) 6.50
- ❖ زمان جستجوی درخت (ب) 4
- ❖ زمان جستجوی درخت (ج) 3.08

برای مثال چگونگی به دست آوردن زمان جستجوی درخت (ج) را توضیح می‌دهیم.

در این درخت، برای مقایسه گرهی با شماره ٤٠ یک مقایسه لازم است. برای گرهی با شماره ۲۰، دو مقایسه برای گرهی با شماره ۱۰ سه مقایسه و برای گرهی با شماره ۳۵ چهار مقایسه و... لازم است.

1+2+2+3+3+3+3+4+4+4+4+4=37

از طرف دیگر، تعداد گره‌های درخت، برابر ۱۲ می‌باشد. پس زمان جستجو برابر است با:

$$\frac{۳۷}{۱۲} = ۳/۰۸$$

همان‌طور که مشاهده می‌کنید، بیشترین زمان جستجو مربوط به شکل (الف) و کمترین زمان جستجو مربوط به شکل (ج) می‌باشد. بنابراین، این پرسش مطرح می‌شود که چگونه می‌توان از مجموعه‌ای از داده‌ها، یک درخت جستجوی دودوئی با کمترین زمان جستجو ایجاد کرد. برای این کار باید درخت متوازن را ساخت.

١-١٣-٦ تعریف درخت متوازن

قبل از تعریف درخت متوازن، درجه توازن هر گره از درخت را تعریف می‌کنیم

اگر h_L ارتفاع سمت چپ گره و h_R ارتفاع سمت راست آن گره باشد، درجه توازن آن گره برابر با $h_L - h_R$ (یا $h_R - h_L$) می‌باشد. معمولا درجه توازن را با bf نمایش می‌دهند. بنابراین خواهیم داشت:

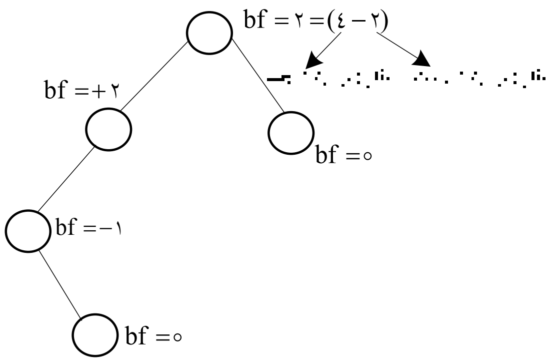
$$bf = h_L - h_R$$

با توجه به تعریف درجه توازن، درخت متوازن را به‌صورت زیر تعریف می‌کنیم

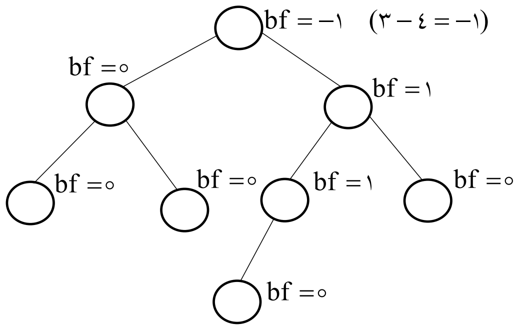
تعریف درخت متوازن: درخت جستجوی دودوئی است که ضریب توازن هر گره آن ۱، ۰ یا ۱- باشد. یعنی برای هر گره داشته باشیم:

$$bf = |h_L - h_R| \leq ۱$$

به‌عنوان مثال شکل ٤٣-٦ سه درخت را به همراه درجه توازن هر گره نشان می‌دهد.



(الف)



(ب)

شکل ۴۳-۶ درخت‌های دودوئی و محاسبه درجه توازن

براساس تعریف درخت متوازن شکل (الف) متوازن نیست چون یکی از گره‌ها دارای درجه توازن ۲ است ولی درخت (ب) درختی متوازن است.

حال بعد از تعاریف بالا درخت AVL را تعریف می‌کنیم:

تعریف درخت AVL: درخت AVL، درخت جستجوی دودوئی است که ارتفاع آن متوازن باشد. یعنی درجه توازن تمام گره‌های آن ۱، ۰ یا −۱ باشد.

هدف عمده درخت AVL، اجرای کارآمد اعمال جستجو، درج و حذف در آن است. این اعمال در درخت دودوئی متوازن نسبت به سایر درخت‌ها با کارایی بیشتری انجام می‌گیرد.

۱۴-۶ حل تعدادی مثال

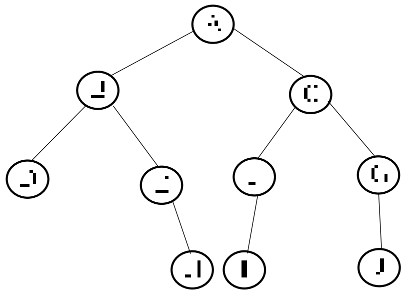
در این بخش قصد داریم با حل تعدادی مسئله مفاهیم درخت را بطور کامل بحث کنیم.

مثال-۱: تابعی بنویسید که برگ‌های درخت دودوئی را محاسبه نماید.

حل: فرض می‌کنیم درخت دودوئی در نظر گرفته شده، کامل نباشد بنابراین درخت با لیست پیوندی پیاده‌سازی شده است. بنابراین خواهیم داشت:

```
int     Count (node *tree)
{
        if ( tree==NULL )
                return 0 ;
        else if ((tree→left)= = NULL && (tree→right)= = NULL )
                return 1;
        else
                Count: = Count (tree→left)+Count (tree→right);
}
```

مثال-۲: سه روش پیمایش را در نظر گرفته و درخت زیر را با سه روش پیمایش نمائید.



حل: همان‌طور که می‌دانید در روش Postorder اول زیر درخت چپ سپس زیر درخت راست و بعد ریشه ملاقات می‌شود. بنابراین خواهیم داشت:

DHEBIFJGCA

در روش Inorder اول زیر درخت چپ بعد ریشه و بعد زیر درخت راست ملاقات می‌شوند لذا خواهیم داشت:

DBEHAIFCJG

در روش preorder همان‌طور که می‌دانید اول ریشه بعد زیر درخت چپ و بعـد زیردرخت راست ملاقات می‌شود. بنابراین:

ABDEHCFIGJ

خروجی پیمایش به روش preorder خواهد بود.

مثال ۳–٦: پیمایش پیشوندی یک درخت دودوئی به‌صورت :

ABDFCEG

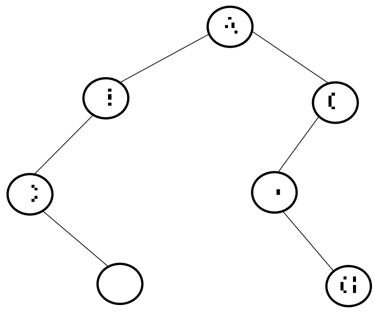
می‌باشد و پیمایش میانوندی آن به‌صورت:

DFBAEGC

است. درخت دودوئی مربوطه را ترسیم نمائید.

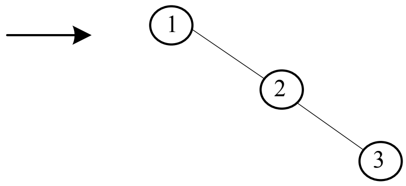
حل: همان‌طور که می‌دانید در پیمایش پیشوندی اولین گره ملاقات شـده، ریـشه می‌باشد. بنابراین ریشه درخت، گره با مقدار A می‌باشـد. از پیمـایش Inorder درخـت متوجه می‌شویم که DFB در زیر درخت چـپ و EGC در زیـر درخـت راسـت قـرار دارند.

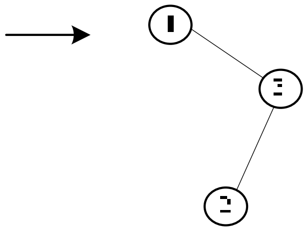
بنابراین با توجه به خواص گفته شده برای زیر درخت‌ها، درخت دودوئـی زیـر حاصل می‌شود:

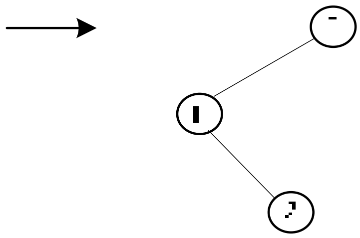


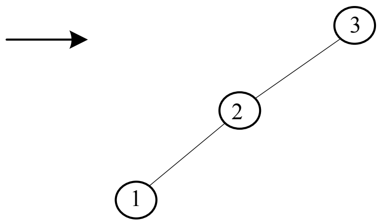
مثال–۵: با مقادیر ۱، ۲، ۳ چند درخت جستجوی دودوئی می‌توان ساخت.

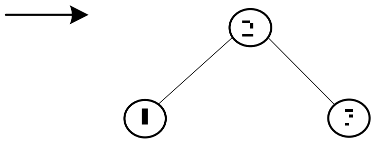
حل: مقادیر فوق را با حالت‌های مختلف در نظر گرفتـه و بـا هـر ترتیـب یـک درخت دودوئی می‌سازیم. بنابراین خواهیم داشت:

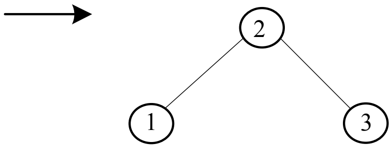
1, 2, 3


1, 3, 2


3, 1, 2


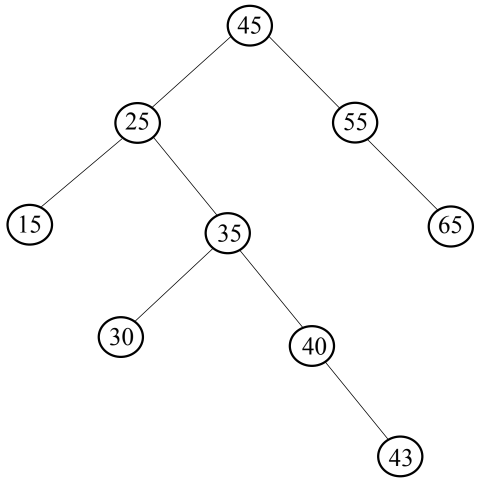
3, 2, 1


2, 3, 1


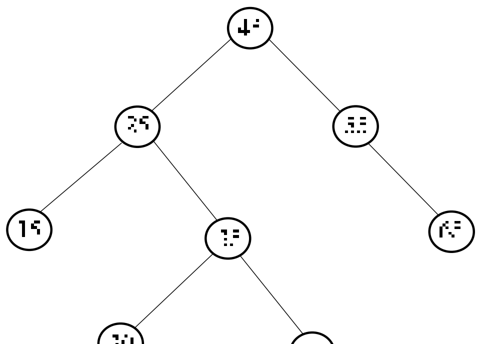
2, 1, 3


همان‌طور که مشاهده می‌کنید به غیر از درخت مرحله ششم، بقیه درخت‌ها متمایز می‌باشند. بنابراین ۵ درخت bst می‌توان ساخت.

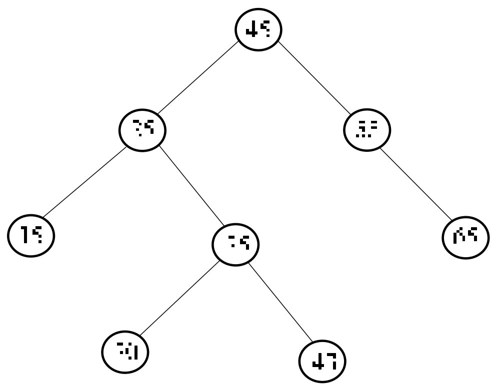
مثال ۵–٦: درخت جستجوی دودوئی زیر را در نظر بگیرید:



الف) گره با مقدار ٤۳ را از درخت حذف کنید. مطابق قسمت اول الگوریتم حذف، درخت زیر حاصل می‌شود:



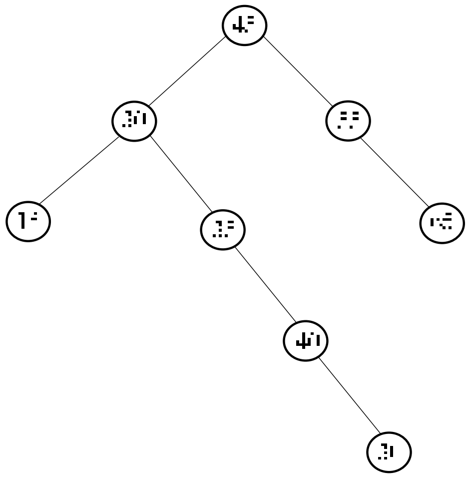
ب) به جای گره با مقدار ٤۳، گره با مقدار ٤۰ را حذف کنید طبق قسمت دوم الگوریتم، درخت زیر حاصل می‌شود:



ج) گره با مقدار ۲۵ را از درخت حذف کنید. طبق قسمت سوم الگوریتم حـذف، پیمایش Inorder درخت عبارت است از:

15 25 30 35 40 43 45 55 65

گره بعد از گره ۲۵، گره با مقدار ۳۰ می‌باشد که طبق الگوریتم جایگزین گـره ۲۵ می‌شود. بنابراین خواهیم داشت:



۱۵-۶ تمرین‌های فصل

۱. درخت‌های 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. درخت‌های جستجوی دودوئی حاصل از درج اعداد زیر را رسم کنید:

d. 1, 2 , 3 , 4 , 5

e. 5 , 4 , 1 , 7 , 8

f. 6 , 5 , 3 , 1

g. 4 , 1 , 5 , 3 , 6 , 2

۳. برای عبارات ریاضی زیر، درخت دودوئی رسم کنید. سپس با پیمایش درخت، عبارات پیشوندی و پسوندی آن‌ها را چاپ کنید:

h. (A – B) – C

i. A – (B – C)

j. A / (B – (C – (D – (E – F))))

k. (A * C + (B – C) / D) * (E – F % G)

l. (A * B + D / (C – K))

٤. پیمایش‌های میانوندی و پسوندی درختی به‌صورت زیر هستند، درخت را رسم کنید:

Inorder: GFHKDLAWRQPZ

Postorder: FGHDALPQRZWK

٥. پیمایش‌های میانوندی و پیشوندی درختی به‌صورت زیر هستند، درخت را رسم کنید:

Inorder: GFHKDLAWRQPZ

Perorder: ADFGHKLPQRWZ

٦. با مثالی نشان دهید که اگر نتایج پیمایش‌های پیشوندی و پسوندی درختی معلوم باشد، نمی‌توان درخت منحصربه فردی را رسم کرد.

۷. الگوریتم بازگشتی و غیربازگشتی برای تعیین موارد زیر بنویسید:

الف) تعداد گره‌های در یک درخت دودوئی.

ب) مجموع محتویات کلیه گره‌ها در یک درخت دودوئی.

ج) عمق درخت دودوئی.

۸. 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 کپی هم و زیردرخت راست آنها نیز کپی هم باشند. الگوریتمی بنویسید که مشخص کند دو درخت دودوئی کپی هستند یا خیر. 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 به‌صورت 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 به‌عنوان زیردرخت چپ و درخت فیبوناچی مرتبه n-2 به‌عنوان زیردرخت راست است.

الف) تابعی بنویسید که اشاره گر به یک درخت فیبوناچی را برگرداند.

ب) آیا چنین درختی، دودوئی محض است.

ج) تعداد برگ‌های درخت فیبوناچی مرتبه n چیست؟

د) عمق درخت فیبوناچی مرتبه n چیست؟

۱۵. تابعی بنویسید که تعداد کل گره‌ها، تعداد برگها، تعداد گره‌های تک فرزندی، تعداد گره‌های دو فرزندی و تعداد شاخه‌های یک درخت را محاسبه و برگرداند.

۱۶. تابعی بازگشتی و غیربازگشتی بنویسید که زیردرخت‌های چپ و راست یک درخت را جابجا کند.

۱۷. تابعی بنویسید که یک درخت عمومی را از ورودی خوانده تبدیل به درخت دودوئی معادل کرده و پیمایش‌های این درخت حاصل را چاپ کند.

۱۸. تابعی بنویسید که جنگلی را از ورودی خوانده تبدیل به درخت دودوئی معادل کرده و پیمایش‌های این درخت حاصل را چاپ کند.

۱۹. تابعی بنویسید که فرم پرانتزی یک درخت دودوئی را به‌صورت رشته از ورودی خوانده و آن را به فرم لیست پیوندی در حافظه پیاده‌سازی کند و سپس پیمایش‌های مختلف آن را چاپ کند.

۲۰. چند درخت با n گره وجود دارد؟

۲۱. چند درخت مختلف با n گره و حداکثر m سطح 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 می‌شود.

۲۳. ثابت کنید که اگر m فیلد اشاره گر در هر گره درخت عمومی برای اشاره به حداکثر m فرزند وجود داشته باشد و تعداد گره‌های درخت برابر با n باشد، تعداد فیلدهای اشاره گر فرزند که برابر null هستند برابر با $n*(m-1)+1$ است.

۲۴. چگونه می‌توان 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:

m. 2 , 4 , 7 , 3 , 1 , 8
n. 1 , 2 , 3 , 5 , 6 , 9
o. 9 , 6 , 5 , 3 , 2 , 1
p. 3 , 5 , 6 , 4 , 2

۲۶. تابعی بنویسید که اعدادی را به ترتیب از ورودی خوانده و در یک درخت Heap (به‌صورت آرایه) ذخیره کند. سپس این آرایه را چاپ کند.

۲۷. الگوریتم‌های درج، حذف و جستجو در درخت AVL را بنویسید.

۲۸. تابعی بنویسید که تعدادی داده با درصد احتمال بروز هر یک را گرفته و کدهافمن معادل را چاپ کند.

۲۹. حروف a, d, b, c, e با جدول فراوانی زیر داده شده است. درخت‌هافمن این مسئله را رسم کنید.

حروف	a	b	c	d	e
فراوانی	0.05	0.1	0.25	0.28	0.32

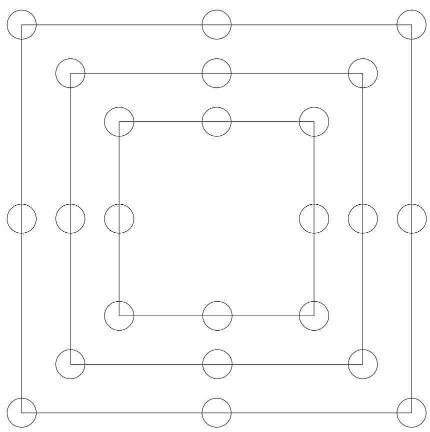
۱۶–۶ پروژه‌های برنامه‌نویسی

۱. بازی nim به این صورت است که:

تعدادی چوب کبریت موجود می‌باشند. دو بازیکن وضعیت آن مجموعه را با حذف یک یا دو چوب کبریت تغییر می‌دهند. بازیکنی که آخرین چوب کبریت را حذف می‌کند، بازنده است.

برنامه‌ای بنویسید که این بازی را شبیه‌سازی نماید.

۲. شکل زیر را در نظر بگیرید:



دو بازیکن A و B را در نظر بگیرید که هر کدام ۱۲ مهره (مثلاً A مهره‌های سفید و B مهره‌های قرمز) در اختیار دارند و می‌توانند در این شکل قرار دهند در صورتی که بازیکن A بتواند سه مهره را در یک راستا قرار دهد یک مهره بازیکن B از دور خارج می‌شود و برعکس.

برنامه‌ای بنویسید که بازی بالا را شبیه‌سازی کند.

فصل هفتم

گراف‌ها (Graphs)

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 به‌دست آورید.

✓ درخت پوشا را تعریف کرده و الگوریتم راشال و پریم را برای به‌دست آوردن درخت پوشا به کار ببرید.

سؤال‌های پیش از درس

۱. به نظر شما آیا گراف را می‌توان یک نوع داده جدید معرفی کرد؟

۲. چه مسائلی را می‌توان با گراف مدلسازی نمود.

۳شما معمو لاً برای پیدا کردن کوتاه‌ترین مسیر از شهری به شهر دیگر چگونه عمل می‌کنید؟

مقدمه

در این فصل یکی دیگر از ساختمان داده‌های 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 هستند که درآنها سیگنال‌ها، پالس‌های الکتریکی و مانند اینها در مسیرهای گوناگونی از یک راس به راس دیگر جریان می‌یابند.

۱-۷ چند اصطلاح نظریه گراف

در اینجا، اصطلاحات مربوط به گراف‌ها را بیان می‌کنیم تا بتوانیم الگوریتم‌های لازم را به‌راحتی مورد بررسی قرار دهیم.

تعریف گراف: یک گراف G از دو مجموعه E و V تشکیل می‌شود:

○ V مجموعه‌ای از عناصر که راس‌ها یا رأس‌ها(vertex) نامیده می‌شود.

○ E مجموعه‌ای از یالها(edge)، طوری که هر یال e درمجموعه E به وسیله یک جفت منحصر به فرد نامرتب (u,v) از راس‌ها در v مشخص می‌شود و آن را با e(u,v) نشان می‌دهند.

راس‌های همجوار: دو راس x , y را در صورتی همجوار گویند هرگاه یالی از x به y وجود داشته باشد.

تعریف مسیر: یک مسیر P به طول n از یک راس u به راس v به‌صورت دنباله‌ای از n+1 راس تعریف می‌شود.

$$P = (v_o, v_1, v_2, ..., v_n)$$

به‌طوری $u = v_o$ و $v = v_n$ است و v_{i-1} به ازای $i = ۱,۲,...,n$ مجاور v_i می‌باشد.

مسیر بسته: مسیر P را بسته گویند اگر $v_o = v_n$ باشد. همچنین مسیر P را ساده گویند اگر تمام راس‌های آن متمایز باشد.

دور یا حلقه: یک دور یا حلقه، مسیر ساده است که اولین و آخرین راس آن یکی است.

گراف همبند: گراف G را همبند گویند اگر بین هر دو راس آن مسیری وجود داشته باشد.

گراف کامل: گراف G را کامل گویند اگر هر راس u در G مجاور هر راس v در G باشد. به‌عبارت دیگر بین هر دو راس آن یک یال وجود داشته باشد. واضح است چنین گرافی همبند است.

بنابراین براحتی می‌توان گفت که گراف کامل با n راس، $\frac{n(n-۱)}{۲}$ یال دارد.

گراف برچسب‌دار: گراف G را برچسب‌دار گویند هرگاه اطلاعاتی به یال‌های آن نسبت داده شود.

گراف وزن‌دار: گراف G را وزن‌دار گویند اگر به هر یال e در G یک مقدار عددی غیرمنفی w که وزن e نامیده می شود.

یال حلقه: در گراف G یال e را حلقه گویند اگر نقاط شروع و پایانی یکسانی داشته باشد.

حال با ارائه یک مثال همه موارد بالا را نشان می‌دهیم.

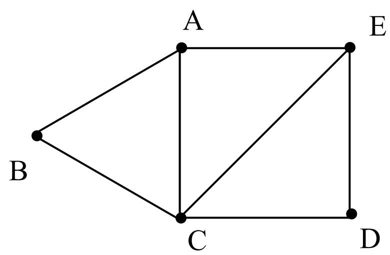
مثال ۱-۷: شکل ۱-۷ (الف) نمودار یک گراف همبند با 5 راس A , B, C, D , E و 7 یال

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

را نشان می‌دهد.

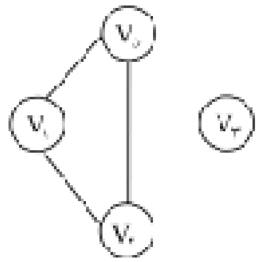
همچنین از B به E دو مسیر ساده به طول 2 وجود دارد. (B,C,E) , (B,A,E)

در این گراف deg(A)=3 می‌باشد، چون A به 3 یال تعلق دارد و به‌طور مشابه deg(C)=4 و deg(D)=2 می‌باشد.

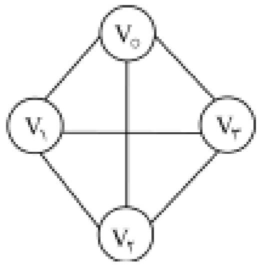


شکل ۱-۷ گراف همبند

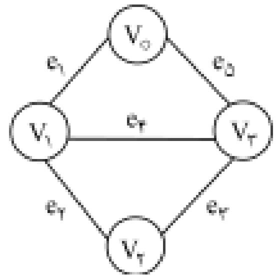
مثال ۲-۷: گراف‌های زیر انواع اصطلاحات گراف‌ها را معرفی می‌کنند:



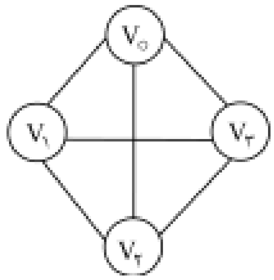
ب): گراف غیرهمبند



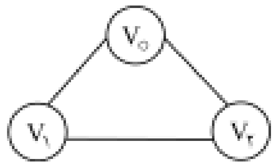
الف): دارای دور یا حلقه



ت): گراف وزن‌دار



پ): گراف کامل



و) گراف با یال دارای حلقه

شکل ۲-۷ انواع گراف‌ها

- **گراف جهت‌دار**

در گرافی که یالها، جهت را نشان دهند گراف جهت‌دار گویند. درجه خروجی راس u در گراف G که به‌صورت 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 به‌صورت indeg(u) نمایش داده می‌شود نشانگر تعداد یالهایی است که در u به پایان می‌رسند.

راس منبع: راس u، یک راس منبع نامیده می‌شود اگر درجه خروجی مثبت داشته باشد و درجه ورودی‌اش صفر باشد.

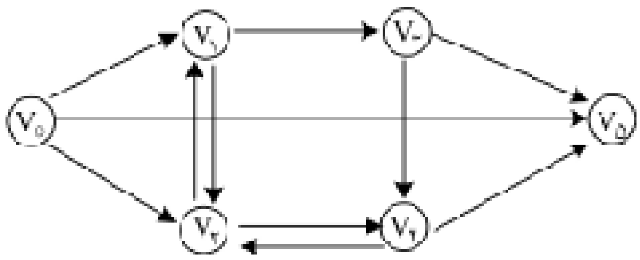
راس مقصد: راس u یک راس مقصد نامیده می‌شود اگر درجه خروجی صفر و درجه ورودی مثبت داشته باشد.

راس u از راس v قابل دسترس است اگر u به v یک مسیر جهت‌دار وجود داشته باشد.

گراف جهت‌دار همبند یا همبند قوی: گراف جهت‌دار G را همبند یا همبند قوی گویند اگر برای هر زوج v, u از راس‌ها 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 راس‌ها در G یک مسیر از u به v یا یک مسیر از v به u وجود داشته باشد.

مثال ۳-۷: شکل زیر گراف جهت‌دار G را نمایش می‌دهد:

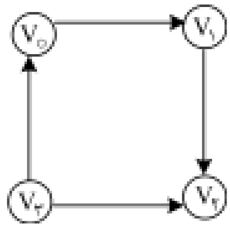


شکل ۳-۷ گراف جهت‌دار G

در شکل بالا $Outdeg\ (v_0)$ برابر ۲ می‌باشد. همچنین $Indeg\ (v_0)$ برابر صفر می‌باشد. بنابراین گره v_0 یک گره منبع می‌باشد.

برای گره v_5، $Indeg\ (v_5)$ برابر ۳ است و $Outdeg\ (v_5)$ برابر صفر می‌باشد. بنابراین این گره، گره مقصد می‌باشد.

مثال-۷: گراف زیر یک گراف جهت‌دار قویا همبند را نمایش می‌دهد:



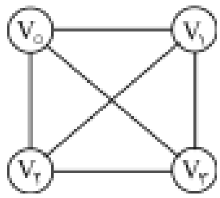
شکل ٤-۷ گراف جهت‌دار همبند

در گراف بالا هم از گره v_0 به v_2 و هم از v_2 به v_0 مسیر وجود دارد برای هر دو گره دلخواه دو مسیر وجود دارد. بنابراین گراف جهت‌دار همبند یا قویا همبند است.

قضیه ۱: اگر G=(V,E) گراف غیرجهت‌دار باشد آنگاه داریم:

$$۲|E| = \sum_{v \in V} \deg(v)$$

مثال ۵-۷: گراف غیر جهت‌دار زیر را در نظر بگیرید:



شکل ۵-۷ گراف غیر جهت‌دار

حال می‌خواهیم صحت قضیه ۱ را در گراف شکل ۵-۷ بررسی کنیم. حال مجموع درجه‌های رئوس از v_0 به v_3 را محاسبه می‌کنیم:

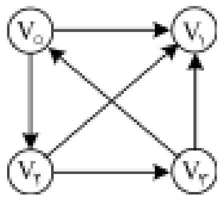
$$\sum_{v \in V} \deg(v) = \deg(v_۰) + \cdots + \deg(v_۳)$$
$$= 3+3+3+3=12$$

و همان‌طور که مشاهده می‌کنید تعداد یال‌های ما ٦ می‌باشد بنابراین قضیه برقرار است.

قضیه ۲: در گراف جهت‌دار G=(V,E) رابطه زیر برقرار است:

$$\sum_{v\in V} \operatorname{Ind}eg(v) = \sum_{v\in V} \operatorname{Out}deg(v) = |E|$$

مثال ٦-۷: گراف جهت‌دار زیر را در نظر بگیرید:



شکل ٦-۷ گراف جهت‌دار

نخست، مجموع یال‌های ورودی گره‌ها را به‌صورت زیر محاسبه می‌کنیم

$$\sum_{v\in V} \operatorname{In}deg(v) = \operatorname{In}deg(v_۰) + \cdots + \operatorname{In}deg(v_۳)$$

$$= ۱ + ۳ + ۱ + ۱ = ٦$$

حال مجموع درجه یال‌های خروجی گره‌های شکل ٦-۷ را محاسبه می‌کنیم:

$$\sum_{v\in V} \operatorname{Out}deg(v) = ۲ + ۰ + +۲ + ۲ = ٦$$

بنابراین همانطور که مشاهده می‌کنید قضیه ۲ برقرار است.

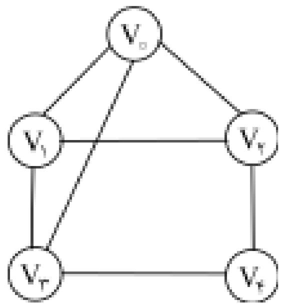
نکته: حداکثر تعداد یال‌های یک گراف جهت‌دار همبند برابر است با:

$$|E| = n(n-۱)$$

که در آن n تعداد رئوس گراف می‌باشد.

قضیه ۳: فرض کنید G=(V,E) یک گراف غیرجهت‌دار باشد آنگاه تعداد رأس‌های با درجه فرد گراف، همواره زوج است.

مثال ۷-۷: گراف غیرجهت‌دار شکل ۷-۷ را در نظر بگیرید:



شکل ۷-۷ گراف غیرجهت‌دار

می‌خواهیم تعداد رئوس با درجه فرد را محاسبه کنیم، همانطور که مشاهده می‌کنید در گراف شکل ۷-۷ رئوس:

V۰ , V۱ , V۳ , V۲

از درجه فرد می‌باشند بنابراین قضیه ۳ برای گراف برقرار است.

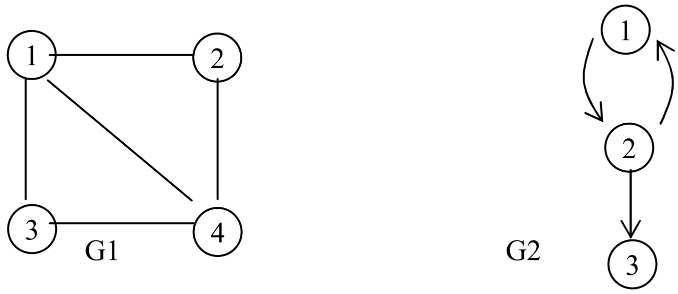
۷-۲ نحوه نمایش گراف‌ها

حال می‌خواهیم روش‌های نمایش گراف‌ها را بررسی کنیم. در کل، دو روش متداول و استاندارد برای نمایش گرافها وجود دارد. یک روش که نمایش ترتیبی گراف G نامیده می‌شود به وسیله ماتریس مجاورت انجام می‌شود. روش دوم نمایش گراف‌ها، نمایش بوسیله لیست پیوندی می‌باشد. در ادامه بحث دو روش را به تفصیل بررسی می‌کنیم.

۷-۲-۱ ماتریس مجاورتی

گراف G=(V,E) با n راس را 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 به‌صورت $<v_i, v_j>$ 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 ۸–۷ چند گراف و ماتریس همجواری آنها نمایش داده شده است.



$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 1 & 1 \\ 2 & 1 & 0 & 0 & 1 \\ 3 & 1 & 0 & 0 & 1 \\ 4 & 1 & 1 & 1 & 0 \end{array} \qquad \begin{array}{c|ccc} & 1 & 2 & 3 \\ \hline 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 1 \\ 3 & 0 & 0 & 0 \end{array}$$

شکل۸–۷ چند گراف و ماتریس مجاورتی آنها

همان‌طور که در شکل ۸–۷ مشاهده می‌کنید، ماتریس همجواری مربوط بـه گـراف بدون جهت، متقارن است. یعنی به ازای هر $n \geq j$ و $i \leq j$ 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 همانطور که مشاهده کردید، اگر تعداد راس‌های گراف بـدون جهـت زیـاد باشد، در نتیجه ماتریس همجواری آن بزرگ خواهد شد. می‌توانیم برای صـرفه‌جـویی در حافظه فقط ماتریس بالا مثلثی ماتریس همجواری را ذخیره کنیم.

با استفاده از ماتریس همجواری به سادگی می‌توان تعیین نمود که آیـا بـین دو راس یالی وجود دارد یا خیر. این عمل را می‌توان در زمان $O(1)$ انجام داد. بـرای گـراف بـدون جهت، درجه هر راس مثل i برابر با مجموع عناصر سطر iام ماتریس همجواری است.

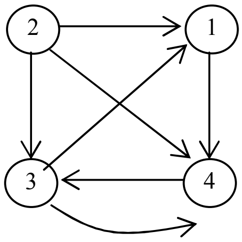
درجه راس i در گراف بدون جهت $= \sum_{J=1}^{n} T[i][j]$

در گراف جهت‌دار، برای به‌دست آوردن درجه خروجی راسی مثل i عناصـر سـطر i را با هم جمع می‌کنیم و برای محاسبه درجه ورودی، عناصر ستون i را با هم جمع می‌کنیم.

درجه خروجی راس i گراف جهت‌دار $= \sum_{J=1}^{n} T[i][j]$

$\sum_{J=1}^{n} T[j][i] =$ درجه ورودی i در گراف جهت‌دار

مثال ۸–۷: گرافلو جهت G، شکل ۹–۷ را در نظر بگیرید:



شکل ۹–۷ گراف جهت‌دار

ماتریس مجاورتی گراف جهت‌دار G چنین است:

$$A = \begin{pmatrix} ۰ & ۰ & ۰ & ۱ \\ ۱ & ۰ & ۱ & ۱ \\ ۱ & ۰ & ۰ & ۱ \\ ۰ & ۰ & ۱ & ۰ \end{pmatrix}$$

توجه کنید تعداد 1ها در A برابر تعداد یال‌های گراف G است.

توان‌های $A^۱, A^۲, A^۳, ...$ از ماتریس مجاورتی A گراف G را در نظر بگیرید.
فرض کنید:

A^k درایه ijام ماتریس $a_k(i,j)$

باشد، ملاحظه می‌کنید که $a_1(i,j) = a_{ij}$ تعداد مسیرهای به طول ۱ از راس v_i به v_j را به‌دست می‌دهد. می‌توان نشان داد که $a_2(i,j)$ تعداد مسیرهای به طول 2 از v_i به v_j است.

قضیه ٤: فرض کنید A ماتریس مجاورتی گراف G باشد. آنگاه $a_k(i,j)$ درایه ijام ماتریس A^k، تعداد مسیرهایی از v_i به v_j را به‌دست می‌دهد که طول k دارند.

فرض کنیم اکنون ماتریس B_r را به‌صورت زیر تعریف کرده‌ایم:

$$B_r = A + A^۲ + A^۳ + ... + A^r$$

آنگاه درایه ij ام ماتریس B_r تعداد مسیرهای به طول r یا کمتر از r را از راس v_i به v_j محاسبه می‌کند.

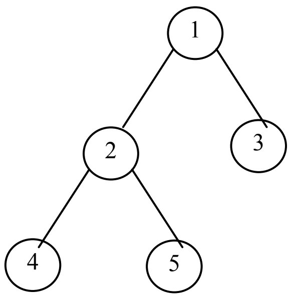
۷-۲-۲ نمایش گراف با استفاده از لیست پیوندی

فرض کنید G یک گراف با m راس باشد. نمایش ترتیبی G یعنی نمایش G به کمک ماتریس مجاورتی A دارای چند اشکال عمده است. مهمترین مشکل ماتریس مجاورتی، اضافه و حذف راس‌ها می‌باشد. براینکه با تغییر اندازه A، راس‌ها را می‌بایست از نو مرتب کرد، و این باعث می‌شود اعمال زیادی برای این تغییرات نیاز است.

علاوه بر این اگر تعداد یال‌ها $O(m)$ یا $O(m\log m)$ باشد، آنگاه ماتریس A، خلوت(اسپارس) خواهد بود. چون دارای صفرهای بسیار زیادی می‌باشد. از این رو مقدار زیادی از حافظه به هدر می‌رود. بنابراین G را معمولاً در حافظه به‌صورت پیوندی نمایش می‌دهند.

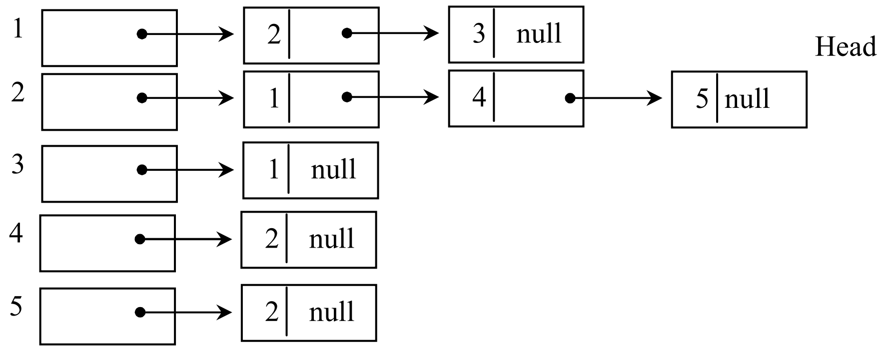
در این نمایش برای هر رأس از گراف G یک لیست وجود خواهد داشت. در هر لیست مشخصی مانند i، راس‌های لیست، حاوی رئوس مجاور از رأس i می‌باشد. هر لیست یک راس Head دارد که به ترتیب شماره‌گذاری شده‌اند و این امر دستیابی سریع به لیستهای مجاورتی برای رأس خاصی را به آسانی امکانپذیر می‌سازد.

مثال۴: گراف G شکل زیر را در نظر بگیرید:



شکل ۱۰-۷ گراف G

لیست مجاورتی گراف شکل ۱۰-۷ به‌صورت زیر ترسیم می شود:



با توجه به شکل ۱۰-۷ درجه هررأس یک گراف بدون جهت را می‌توان به سادگی با شمارش تعداد راس‌های آن در لیست مجاورتی مشخص نمود. بنابراین در حالت کلی می‌توان گفت که، اگر تعداد رئوس گراف G برابر n باشد تعداد کل یال‌ها در زمان o(n+e) تعیین می‌شود.

۳-۷ عملیات بر روی گراف‌ها

حال در اینجا مانند سایر ساختار داده‌ها عملگرهای لازم بر روی این ساختار داده را ارائه می‌کنیم. بعضی از عملگرها مربوط به گراف را به‌طور کامل مورد بحث و بررسی قرار خواهیم داد. این عمگرها در حالت کلی عبارتند از:

۱. پیمایش گراف‌ها ۲. جستجو در گراف

۳. اضافه کردن راسی به گراف ٤. حذف راسی از گراف

۱-۳-۷ پیمایش گراف‌ها

جستجو و پیمایش در گراف، ارتباط تنگاتنگی با یکدیگر دارند. به‌طوری‌که عمل جستجو می‌تواند با عملیات پیمایش انجام شودقب ًلا با مفهوم پیمایش در درخت‌ها آشنا شدیم 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 گرهی رسید. اما در یک گراف ممکن است نتوان از هر راسی به راس دیگر رسید. لذا ممکن است از راسی که به‌عنوان شروع استفاده می‌کنیم، به تمام

راس‌های گراف نرسیم. تعریف پیمایشی که به ساختار گراف مربوط می‌شود، به سه دلیل پیچیده‌تر از پیمایش درخت است:

۱. در گراف گرهی به عنوان اولین گره وجود ندارد که پیمایش از آن شروع شود. در حالی که در درخت چنین گرهی (به نام ریشه) موجود است. علاوه بر این در گراف ، وقتی کلیه گره‌هایی که از گره شروع قابل دسترسی‌اند ملاقات شدند، ممکن است برخی گره‌ها ملاقات نشده باشند، زیرا ممکن است مسیری از گره شروع به گره مزبور وجود نداشته باشد. اما در درخت تمام گره ها از ریشه قابل دسترسی هستند.

۲. بین جانشین‌های (successor) یک راس، ترتیب خاصی وجود ندارد. بنابراین هیچ ترتیبی وجود ندارد که راس‌های جانشین یک راس، بر اساس آن پیمایش شوند.

۳. برخلاف راس‌های درخت، هر راس گراف ممکن است بیش از یک راس پیشین داشته باشد. اگر xها جانشین (فرزند) دو راس y و z باشد، x ممکن است بعد از y و قبل از z ملاقات شود. بنابراین ممکن است راسی، قبل از یکی از راس‌های پیشین خود (پدر خود) ملاقات شود.

با توجه به این سه مورد تفاوت که بیان شد، الگوریتم‌های پیمایش گراف، سه ویژگی مشترک باید داشته باشند:

۱. الگوریتم ممکن است طوری تهیه شود که پیمایش را از راس خاصی شروع کنیم یا راس‌ها را به‌طور تصادفی انتخاب نماید و پیمایش را از آن راس شروع کند. چنین الگوریتمی، بر اساس این که از کدام راس شروع به پیمایش می‌کند، ترتیب گوناگونی از راس‌ها را به خروجی می‌برد.

۲. به‌طور کلی پیاده سازی گراف، ترتیب ملاقات راس‌های جانشین یک راس را مشخص می‌کند. به‌عنوان مثال، اگر از ماتریس همجواری برای پیاده‌سازی گراف استفاده شود، شماره‌گذاری راس‌ها از 0 تا n-1، این ترتیب را مشخص می‌کند. اگر از پیاده‌سازی لیست همجواری استفاده شود، ترتیب یالها در لیست همجواری، ترتیب ملاقات راس‌های جانشین را تعیین می‌کند.

۳. اگر راسی بیش از یک راس پیشین داشته باشد، ممکن است بیش از یک بار در پیمایش ظاهر شود. الگوریتم پیمایش بایدبررسی کند که آیا راس 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 می‌تواند یکی از شرایط زیر را داشته باشد:

- حالت آماده (راس 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 راسی در این گراف باشد، در پیمایش گراف G لازم است مشخص کنیم چه راس‌هایی از طریق راس x قابل دستیابی‌اند. برای این منظور از دو الگوریتم استاندارد استفاده می‌شود:

- جستجوی عمقی یا پیمایش عمقی (depth – first search = dfs)
- جستجوی عرضی با پیمایش عرضی (breadth – first search = bfs)

۷-۳-۲ جستجوی عرضی

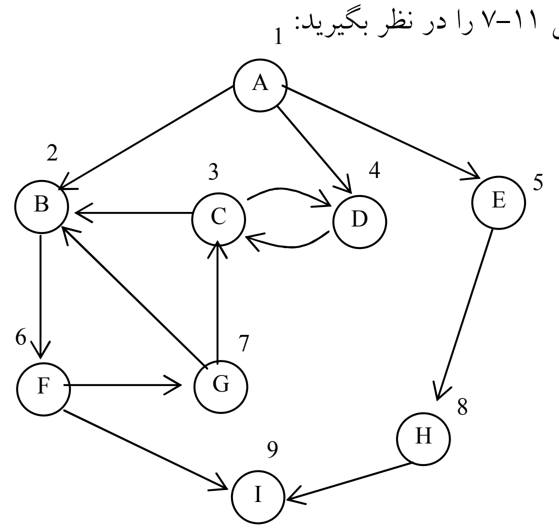
ایده کلی جستجوی عرضی بدین صورت است که کار را با یک آغازین، به شرح زیر شروع می کنیم. نخست راس آغازین A را ملاقات می‌کنیم. آنگاه تمام همسایه‌ها یا راس‌های مجاور A را ملاقات می‌کنیم. سپس تمام همسایه‌های راس‌های مجاور A را ملاقات می‌کنیم و این روند تا ملاقات تمام رئوس ادامه می‌دهیم. لازم است اطمینان داشته باشیم که هیچ راسی بیشتر از یک بار پردازش نشود. این کار با استفاده از یک صف، جهت نگه داشتن راس‌هایی که در انتظار پردازش بسر می‌برند و با استفاده از فیلد STATUS که وضعیت جاری هر راس را به ما اطلاع می‌دهد، انجام می‌شود. حال الگوریتم جستجوی ردیفی یا عرضی را به شرح زیر ارائه می‌دهیم:

الگوریتم جستجوی عرضی

الگوریتم: الگوریتم جستجوی عرضی، با شروع از راس آغازین A روی یک گراف G اعمال زیر را اجرا می‌کند:

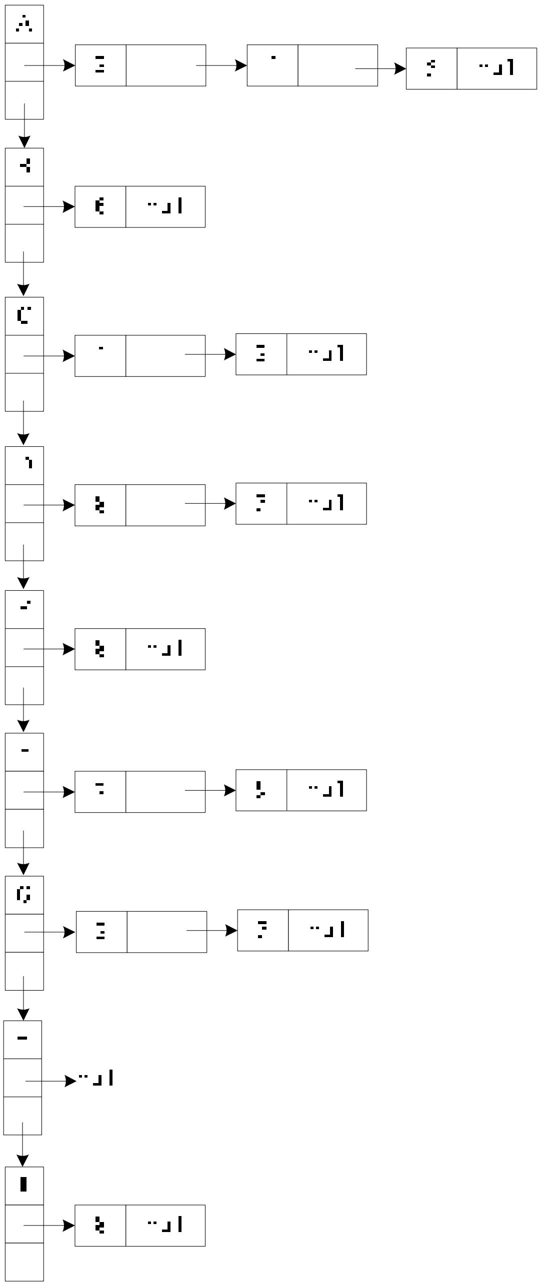
۱. تمام راس‌هایی که در حالت آماده (STATUS=1) هستند مقدار اولیه می‌دهد.
۲. راس آغازین A را در صف (QEUEUE) قرار دهید و وضعیت آن را به حالت انتظار (STATUS=2) تغییر دهید.
۳. مرحله‌های ٤ و ۵ را تا وقتی صف خالی نشده تکرار کنید.
٤. راس V را از ابتدای صف حذف کنید. V را پردازش کنید. و وضعیت آن را به حالت پردازش شده (STATUS=3) تغییر دهید.
۵. تمام همسایه‌های V را به انتهای صف اضافه کنید که در حالت آماده هستند (STATUS=1) و وضعیت آنها را به حالت انتظار (STATUS=2) تغییر دهید.

گراف G شکل ۱۱–۷ را در نظر بگیرید:



شکل ۱۱–۷ گراف جهت‌دار

لیست مجاورتی گراف شکل ۱۱–۷ به‌صورت زیر می‌باشد (در لیست برای راحتی کار از اعداد به‌عنوان برچسب استفاده شده است):



شکل ۱۲-۷ لیست مجاورتی گراف جهت‌دار

اکنون الگوریتم جستجوی عرضی را بر روی گراف شکل۱۲-۷ اجـراء مـی‌کنـیم

(فیلد توضیح: وضعیت تمام راس‌ها 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 آنها 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 به 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 را در صف قرار دهید و فیلد وضعیت آن را به ۲ تغییر دهید.

۸. راسE را از صف خارج کرده و در خروجی بنویسید و فیلد وضعیت آن را به ۳ تغییر دهید.

۹. راس‌های همجوار E را که در حالت آماده قرار دارند در صف قرار دهید. راس همجوار E راس H است 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. راس‌های همجوار F را که در حالت آماده قرار دارند در صف قرار دهید و حالت آن را به ۲ تغییر دهید این راس‌ها I ,G هستند.

۱۲. C را از صف خارج کرده و در خروجی بنویسید و فیلد وضعیت آن را به ۳ تغییر دهید.

۱۳. راس‌های همجوار C راس‌های D و G هستند که قبلاً بررسی شده‌اند.

- راس H را از صف خارج کرده و در خروجی بنویسید و فیلد وضعیت آن را به ۳ تغییر دهید.

- راس H همجوار ندارد که در صف قرار گیرد.

• راس G را از صف خارج کرده و در خروجی بنویسید و وضعیت آن را به ۳ تغییر دهید.
• راس‌های همجوار G قبلاً بررسی شده‌اند.
• I را از صف خارج کرده و در خروجی بنویسید و وضعیت آن را به ۳ تغییر دهید.
• چون صف خالی شد، الگوریتم به پایان می‌رسد، خروجی حاصل از این الگوریتم به‌صورت زیر خواهد بود:

ABDEFCHGI

جستجوی عرضی

ایده کلی جستجوی عرضی بدین صورت است که کار را با گره آغاز به شرح زیر شروع می‌کنیم. 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 یکبار پردازش نشود. این کار با استفاده از یک صفت جهت نگهداشتن گره‌هایی که در انتظار پردازش به سر می‌برند و با استفاده از STATUS که وضعیت جاری هر گره را به ما اطلاع می‌دهد انجام می‌شود.

حال تابع این الگوریتم را ارائه می‌دهیم. الگوریتم زیر پیاده‌سازی bfs را به‌صورت غیربازگشتی و با استفاده از صف q نشان می‌دهد (در تابع زیر به جای فیلد STATUS از لیست visited با مقادیر False و True استفاده می‌کنیم):

الگوریتم پیمایش عرضی یا ردیفی

```
void    bfs ( int   v  )
{
        visited [v] = true;
        addq(q , v) ;
        while (!Empty_queue(q))
        {
                delq (q , v) ;
                for(all vertex W adjacent with V)
                {
                        addq (q , w) ;
                        visited [w] = true ;
```

```
            }
        }
}
```
در الگوریتم فوق V نشان‌دهنده راس آغازین گراف 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 راس در لیست‌های مجاورتی یک بار ملاقات می‌شود، کل زمان جستجو $O(|E|)$ (E تعداد یال‌ها) است اگر G توسط ماتریس مجاورتی نمایش داده شود، 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.

۳-۳-۷ جستجوی عمقی

ایده کلی الگوریتم جستجوی عمقی بدین صورت است که، کار را با راس آغازین A به‌صورت زیر شروع می‌کنیم:

ابتدا، راس آغازین A را ملاقات می‌کنیم. آنگاه هر راس v را که در مسیر P قرار دارد و با A شروع می‌شود را ملاقات می‌کنیم یعنی یک÷ همسایه A را پردازش می‌کنیم سپس همسایه همسایه A را پردازش می‌کنیم. پس از رسیدن به نقطه پایان P، بطرف P برمی‌گردیم تا بتوانیم در طول مسیر دیگری مانند Q پردازش را ادامه دهیم این پروسه را برای تمام راس‌ها ادامه می‌دهیم. این روش پیمایش، مشابه پیمایش Preorder یک درخت دودوئی است.

به بیان دقیق‌تر، در آغاز رأس A را ملاقات می‌کنیم. بعد رأسی مانند y| که قبلاً ملاقات نشده و مجاور A است را انتخاب کرده و روش جستجوی عمقی را با آن ادامه می‌دهیم در هر مرحله، همچنین موقعیت جاری راس A در لیست مجاورتی با قرار دادن آن در یک پشته صورت می‌گیرد. 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.

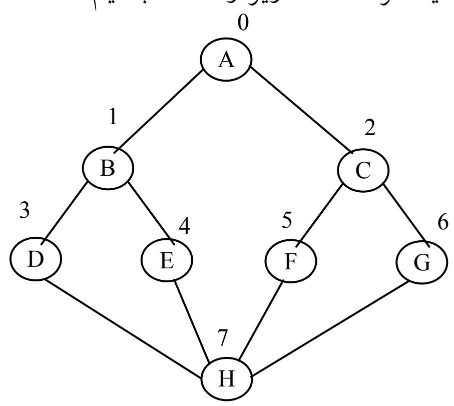
در این الگوریتم، رئوس ملاقات شده، از پشته خارج شده و رئوس ملاقات

نشده، در پشته قرار می‌گیرند و جستجو زمانی پایان می‌یابد، که پشته خالی شود.

جدول زیر، الگوریتم جستجوی عمقی را با شروع از راس آغازین A روی گراف G اجرا می‌کند:

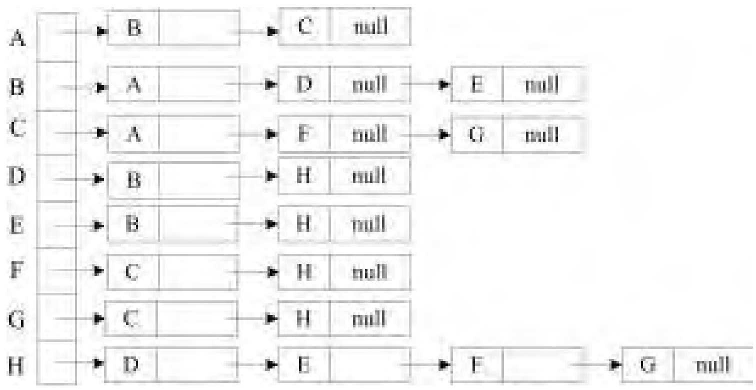
الگوریتم جستجوی عمقی
الگوریتم جستجوی عمقی را با شروع از راس آغازین A روی یک گراف G اجرا می‌کند. ۱. تمام راس‌هایی را که در حالت آماده STATUS=1 هستند مقدار اولیه می‌دهد. ۲. راس آغازین A را در پشته push کنید و وضعیت آن را به حالت انتظار STATUS=2 تغییر دهید. ۳. مرحله‌های ٤ و ٥ را تا وقتی پشته خالی نشده است تکرار کنید. ٤. راس V بالای پشته را pop کنید و آن را پردازش کنید و وضعیت آن را به STATUS=3 تغییر دهید. ٥. راس مجاور، راس V را به داخل پشته Push کنید طوری که همچنان در حالت آماده STATUS=1 باشد و وضعیت آن را به حالت انتظار (STATUS=2) تغییر دهید.

حال با استفاده از مثال ساده‌ای، مراحل کار را به‌صورت صوری و نه الگوریتمیک بیان می‌کنیم فرض کنید گراف ساده زیر را داشته باشیم:



شکل ۱۳-۷ گراف G

شکل زیر لیست مجاورتی گراف شکل ۱۳–۷ را نشان می‌دهد:



شکل ۱٤–۷ لیست مجاورتی گراف G

اکنون الگوریتم جستجوی عمقی را بر روی گراف شکل۱٤–۷ اجراء می‌کنیم (فیلد وضعیت تمام راس‌ها در ابتدا ۱ است که به معنای حالت آماده است.):

- A را در پشته قرار دهید و فیلد وضعیت آن را به ۲ تغییر دهید.
- A را از پشته حذف کنید و آن را در خروجی بنویسید و فیلد وضعیت آن را به ۳ تغییر دهید.
- راس همجوار A، یعنی B را در پشته قرار دهید و فیلد وضعیت آنها را به ۲ تغییر دهید.
- B را از پشته حذف کنید، در خروجی بنویسید و فیلد وضعیت آن را به ۳ تغییر دهید.
- راس همجوار B را که در حالت آماده است، یعنی D را در پشته قرار دهید و فیلد وضعیت آن را به ۲ تغییر دهید.
- D را از پشته حذف کنید، در خروجی بنویسید و فیلد وضعیت آن را به ۳ تغییر دهید.
- راس همجوار D را که در حالت آماده است، یعنی H را در پشته قرار دهید و فیلد وضعیت آن را به ۲ تغییر دهید.
- راس H را از پشته خارج کرده و در خروجی بنویسید و فیلد وضعیت آن را به

۳ تغییر دهید.

• راس همجوار H را که در حالت آماده قرار دارد، یعنی E را در پشته قرار دهید و فیلد وضعیت آن را به ۲ تغییر دهید.

• E را از پشته خارج کرده و در خروجی بنویسید و فیلد وضعیت آن را به ۳ تغییر دهید.

• راس همجوار E در حالت آماده قرار ندارند، زیرا قبلا راس‌های مجاور آن ملاقات شده‌اند. یک مرحله عقب برمی‌گردیم.

• F را در پشته قرار دهید و فیلد وضعیت آن را به ۲ تغییر دهید.

• F را از پشته حذف کنید، در خروجی بنویسید و فیلد وضعیت آن را به ۳ تغییر دهید.

• راس همجوار F را که در حالت آماده است، یعنی C را در پشته قرار دهید و فیلد وضعیت آن را به ۲ تغییر دهید.

• راس C را از پشته خارج کرده و در خروجی بنویسید و فیلد وضعیت آن را به ۳ تغییر دهید.

• راس همجوار C را که در حالت آماده قرار دارد، یعنی G را در پشته قرار دهید و فیلد وضعیت آن را به ۲ تغییر دهید.

• G را از پشته حذف کنید، در خروجی بنویسید و فیلد وضعیت آن را به ۳ تغییر دهید.

پشته خالی می‌شود و خروجی زیر حاصل می‌شود:

A B D H G F C G

در زیر تابع الگوریتم پیمایش عمقی را ارائه می‌دهیم(در تابع زیر به جای فیلد STATUS از لیست visited با مقادیر False و True استفاده می‌کنیم):

الگوریتم پیمایش عمقی

```
void dfs (int  v)
 {
        cout<<Data (v) ;
        visited [v]= ture ;
```

```
        for (each vertex w adjacent to v)
                if (!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.

۷-٤ درخت‌های پوشا و درخت پوشای کمینه

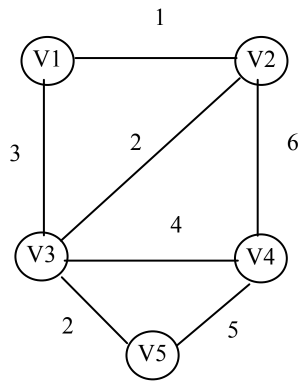
فرض کنید می‌خواهیم چند شهر معین را با جاده به هم وصل کنیم، به‌طوری‌که مردم بتوانند از هر شهر، به شهر دیگر بروند. اگر محدودیت‌های بودجه‌ای در کار باشد، ممکن است طراح بخواهد این کار را با حداقل جاده‌کشی انجام دهد. در این بخش می‌خواهیم الگوریتمی ارائه دهیم که این مسئله و مسئله‌های مشابه را حل کند.

می‌توانیم از گراف بدون جهت و وزن‌دار G، یالهایی را حذف کنیم به‌طوری‌که زیر گراف به‌دست آمده متصل باقی بماند و حاصل جمع وزن‌های یال‌های باقیمانده را کمینه کند. این مسئله کاربردهای متعددی دارد. به‌عنوان مثال در ارتباطات راه دور می‌خواهیم حداقل طول کابل و در لوله‌کشی می‌خواهیم حداقل مقدار لوله مصرف شود. یک زیرگراف با حداقل وزن باید درخت باشد.

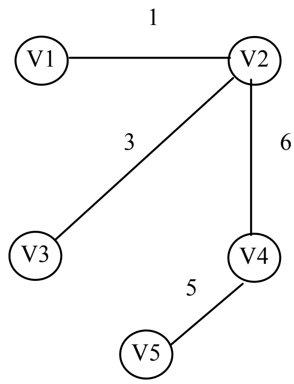
تعریف درخت پوشا: فرض کنید $G=(V,E)$ یک گراف همبند و بدون جهت باشد که در آن V مجموعه رئوس و E، مجموعه یال‌ها می‌باشد. یک زیرگراف $T \subseteq G$، یک درخت پوشای G است اگر و فقط اگر T یک درخت باشد.

به‌عبارت دیگر، درخت پوشای گراف G، زیرگراف همبندی است که حاوی تمام راس‌های گراف G بوده، و همچنین فاقد چرخه یا دور باشد (یعنی درخت باشد). یک گراف همبند با n راس، حداقل می‌تواند $n-1$ یال داشته باشد، که اگر فقط $n-1$ یال داشته باشد یک درخت نامیده می‌شود.

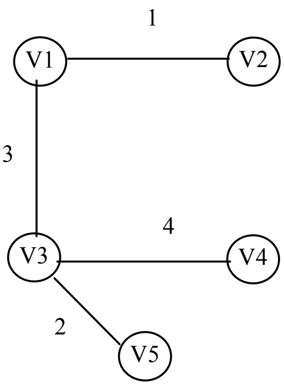
گراف وزن‌دار شکل ۱۵-۷ (الف) را در نظربگیرید. درخت‌های شکل ۱۵-۷ (ب) و (ج) درخت‌های پوشایی برای گراف G هستند.



الف) گراف همبند، موزون و بدون جهت G



ب) درخت پوشا برای G



ج) درخت پوشای کمینه برای G

شکل ۱۵-۷ گراف وزن‌دار و درخت‌های پوشای آن

در عمل می‌توان به یال‌های یک گراف کمیتی را به‌عنوان وزن به آنها نسبت داد. این وزنها ممکن است هزینه ساخت، طول مسیر، میزان ترافیک و غیره باشد. با معلوم شدن یک گراف وزنی، می‌توان مسیرها را طوری طی نمود که هزینه ساخت طول مسیر، کمترین مقدار ممکن را در گراف داشته باشد. هزینه یک درخت پوشا، مجموع هزینه یال‌های آن درخت می‌باشد.

تعربف درخت پوشای کمینه: درخت پوشایی است که حداقل وزن را داشته باشد. به‌عنوان مثال، وزن درخت‌پوشای شکل ۱۵-۷(ج) برابر با ۱۰ و وزن درخت‌پوشای

شکل ۱۵-۷ (ب) برابر با ۱۵ است. بنابراین درخت شکل۱۵-۷ (ج) یک درخت‌پوشای کمینه برای گراف است. توجه داشته باشید که یک گراف ممکن است بیش از یک درخت پوشای کمینه داشته باشد.

در بخش بعدی برای به‌دست آوردن درخت پوشای کمینه یک گراف – الگوریتم‌های راشال و پریم را بررسی خواهیم کرد.

روش ما برای تعیین درخت پوشا با حداقل وزن، داشتن سه شرط زیر می‌باشد:

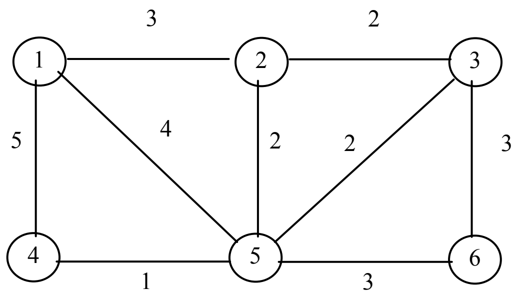
- ✓ فقط باید از یال‌های داخل گراف استفاده کند.
- ✓ باید دقیقاً از n-1 یال استفاده کند (n تعداد رأسها)
- ✓ نباید از یال‌هایی که ایجاد حلقه می‌کنند استفاده کند.

۵-۷ الگوریتم راشال برای ساخت درخت پوشای کمینه

در این الگوریتم درخت پوشای کمینه، یال به یال ساخته می‌شود. برای این منظور، یال‌های گراف بر حسب وزن به ترتیب صعودی مرتب می‌شوند. یال جدید وقتی به درخت T اضافه می‌شود که با یال‌های موجود در T چرخه‌ای ایجاد نکند. این الگوریتم را می‌توان به‌صورت زیر نوشت (برای مطالعه بیشتر به کتاب طراحی الگوریتم مراجعه نمائید):

الگوریتم راشال برای تهیه درخت پوشای کمینه
۱. یال‌ها را به ترتیب صعودی، از کمترین وزن به بیشترین وزن مرتب کنید. ۲. یال‌های مرتب شده را به ترتیب به درخت T اضافه کنید. اگر با افزودن این یال، چرخه ایجاد شود، از آن یال صرفنظر کنید و یال بعدی را بررسی نمائید. ۳. مرحله ۲ را آنقدر تکرار کنید تا به انتهای لیست یال‌های مرتب شده برسید.

گراف شکل ۱٦-۷ را در نظر بگیرید. می‌خواهیم با اعمال الگوریتم راشال بر روی گراف، درخت پوشای کمینه آن را به‌دست آوریم.



شکل ۱٦-۷ گراف وزن‌دار

همانطور که اشاره کردیم، در مرحله اول یال‌ها را به‌صورت صعودی مرتب می‌کنیم:

یال‌هایی با وزن ۱ (4,5)

یال‌هایی با وزن ۲ (5,3) (2,5) (2,3)

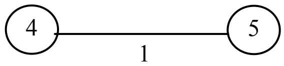
یال‌هایی با وزن ۳ (5,6) (3,6) (1,2)

یال‌هایی با وزن ٤ (1,5)

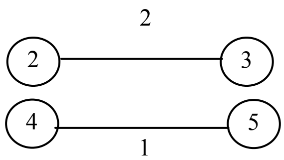
یال‌هایی با وزن ٥ (1,4)

حال مراحل الگوریتم را مرحله به مرحله ادامه می‌دهیم:

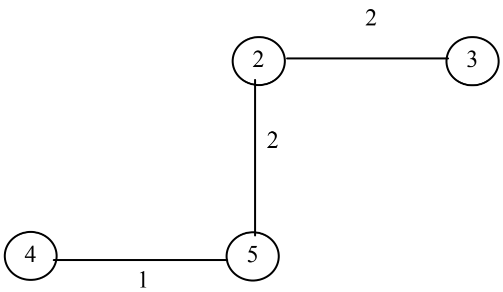
۱. ابتدا یال با وزن ۱ را به درخت اضافه می‌کنیم.



۲. یکی از یال‌های با وزن ۲ را به درخت اضافه می‌کنیم

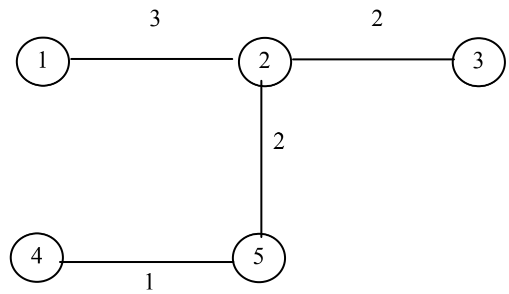


۳. یکی دیگر از یال‌هایی با وزن ۲ را به درخت اضافه می‌کنیم.

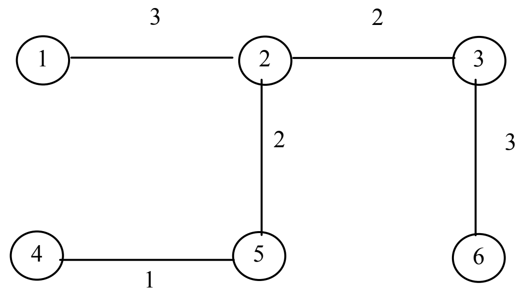


٤. یال (5,3) را که وزن ۲ دارد به درخت اضافه نمی‌کنیم چرا که در ایـن صـورت یـک چرخه درست خواهد شد.

٥. یالی با وزن ۳ که یال (1,2) می‌باشد، را به درخت اضافه می‌کنیم

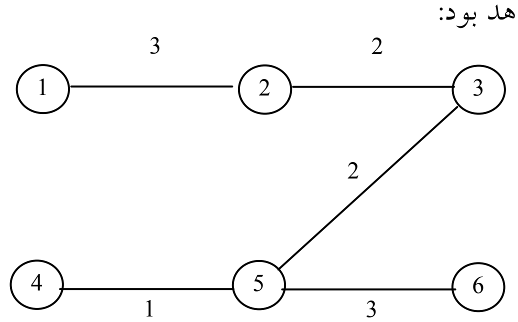


٦. یال (3,6) را که دارای وزن ۳ می‌باشد را به درخت اضافه می‌کنیم



۷. هیچ کدام از یال‌های (5,6) , (1,5) , (1, 4) را نمی‌توان اضـافه نمـود چـون تـشکیل چرخه می‌دهند.

بنابراین وزن درخت پوشانی کمینه خواهد شد. توجـه کنیـد کـه ایـن درخت پوشا به دلیل داشتن یال‌هایی با وزن یکسان، منحصر به فرد نخواهد بـود. بـرای نمونه، یکی دیگر از درخت‌های پوشـای کمینـه بـا وزن ۱۱ بـرای گـراف شـکل ۱٦-۷ به‌صورت زیر خواهد بود:



٦-۷ الگوریتم پریم برای تعیین درخت پوشای کمینه

در این روش، نخست با یک گره دلخواه کار را آغاز می‌کنیم و درخت را یال به یال می‌سازیم. بنابراین در هر مرحله یک یال درخت ساخته می‌شود.

در هر مرحله بهینه‌بودن بررسی می‌شود. بدین‌صورت که یالی را انتخاب می‌کنیم که منجر به حداقل افزایش در مجموع هزینه‌هایی گردد که تا به حال در نظر گرفته شده است (بهینه محلی). الگوریتم زمانی پایان می‌یابد که کلیه گره‌ها به دقت افزوده شود. مجموع هزینه یال‌های این درخت کمترین مقدار است. این الگوریتم به الگوریتم پریم مشهور است.

در هر مرحله پیاده‌سازی این الگوریتم به‌صورت زیر عمل می‌کنیم

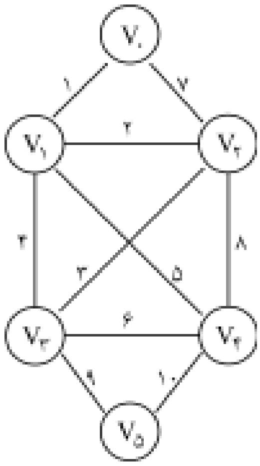
رأسی را به‌عنوان نقطه شروع در نظر می‌گیریم رئوس مجاور این راس را به‌دست می‌آوریم. وزن‌های روی یال‌های مجاور گراف را بررسی می‌کنیم سپس، یال با کمترین هزینه را انتخاب می‌کنیم و این راس را به مجموعه رئوس انتخاب شده تا حالا، اضافه می‌کنیم. این فرایند تا زمانیکه که کلیه رئوس ملاقات نشده‌اند، تکرار می‌شود.

الگوریتم بالا نخست، F را که منظور مجموعه یالهای انتخاب‌شده خواهد بود، ϕ در نظر میگیرید و راس اول مثلاً V_1 را انتخاب می‌کند و داخل مجموعه‌ای مانند Y قرار می‌دهد. سپس تا زمانیکه مسئله حل نشده اعمال زیر را انجام می‌دهد:

- از مجموعه $V-Y$، رئوس مجاور را انتخاب می‌کند (V مجموعه کل رئوس می‌باشد).
- نزدیکترین رأس انتخاب شده را به Y اضافه می‌کند.
- یال مربوطه را به F اضافه می‌کند.
- اگر Y برابر V شده باشد حل مسئله تمام است.

حال با یک مثال الگوریتم **پریم** را به‌طور کامل بررسی می‌کنیم

گراف زیر را در نظر بگیرید:



شکل ۱۷-۷ یک گراف وزن‌دار

آرایه زیر، هزینه یال‌های گراف را نمایش می‌دهد:

$$\begin{array}{c} \\ V_0 \\ V_1 \\ V_2 \\ V_3 \\ V_4 \\ V_5 \end{array}
\begin{array}{c} \begin{array}{cccccc} V_0 & V_1 & V_2 & V_3 & V_4 & V_5 \end{array} \\
\begin{bmatrix} \infty & 1 & 7 & \infty & \infty & \infty \\ 1 & \infty & 2 & 4 & 5 & \infty \\ 7 & 2 & \infty & 3 & 8 & \infty \\ \infty & 4 & 3 & \infty & 6 & 9 \\ \infty & 5 & 8 & 6 & \infty & 10 \\ \infty & \infty & \infty & 9 & 10 & \infty \end{bmatrix} \end{array}$$

طبق الگوریتم، ابتدا رأس $Y=\{V_0\}$ انتخاب می‌شود و $F=\phi$ خواهد بود.

مرحله اول: تمام رئوس مجاور V_0 را پیدا می‌کنیم.

بنابراین خواهیم داشت:

$$\{V_1, V_2\} \text{ و } e_{01}=1 \text{ و } e_{02}=7$$

به وضوع رئوس V_1 رأس انتخابی خواهد بود و یال e_{01} به F اضافه مـی‌شـود و داریم:

$$F=\{e_{01}\}$$

مرحله دوم: حال رئوس مجاور به Y را انتخاب می‌کنیم.

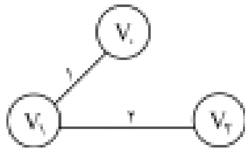
لذا:

$$e_{۰۲} = ۷ \text{ و } e_{۱۲} = ۲ \text{ و } e_{۱۳} = ۴ \text{ و } \{V_۲, V_۳, V_۴\}$$

$$e_{۱۴} = ۵$$

حاصل می‌شود. بنابراین نزدیکترین رأس که $V_۲$ بوده، انتخاب مـی‌شـود. بنـابراین یـال $e_{۱۲}$ به F اضافه می‌شود و داریم:

$$F = \{e_{۰۱}, e_{۱۲}\}$$

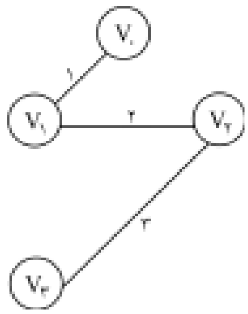


مرحله سوم: رئوس مجاور Y را انتخاب می‌کنیم:

$$e_{۲۳} = ۳ \text{ و } e_{۲۴} = ۸ \text{ و } e_{۱۴} = ۵ \text{ و } e_{۱۳} = ۴ \text{ و } \{V_۳, V_۴\}$$

به‌وضوح، نزدیکترین رأس به Y رأس $V_۳$ بوده و یال $e_{۲۳}$ به F اضافه می‌شود و داریم:

$$F = \{e_{۰۱}, e_{۱۲}, e_{۲۳}\}$$

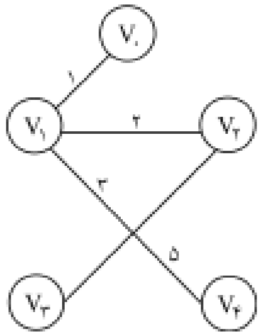


مرحله چهارم: رئوس مجاور Y را انتخاب می‌کنیم:

$$V_{۱۵} = ۵ \text{ و } e_{۳۴} = ۶ \text{ و } e_{۳۵} = ۹ \text{ و } e_{۲۴} = ۸ \text{ و } \{V_۴, V_۵\}$$

نزدیکترین رأس به Y رأس $V_۴$ بوده و یال $e_{۱۵}$ به F اضافه می‌شود و داریم:

$$F = \{e_{۰۱}, e_{۱۲}, e_{۲۳}, e_{۱۵}\}$$

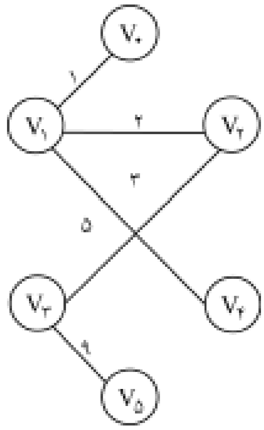


مرحله پنجم: رئوس مجاور Y را انتخاب می‌کنیم:

$e_{۴۵} =$ ۱۰ و $e_{۳۵} =$ ۹ و $\{V_۵\}$

نزدیکترین رأس به Y، رأس $V_۵$ بوده و بنابراین خواهیم داشت:

$F = \{e_{۰۱}, e_{۱۲}, e_{۲۳}, e_{۱۵}, e_{۳۵}\}$

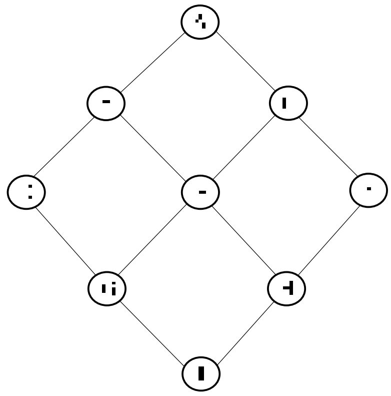


مرحله ششم: چون V=Y می‌باشد. بنابراین گراف حاصل به‌عنوان درخت پوشای مینیمم با مقدار هزینه 20 می‌باشد.

۷-۷ ارائه مسائل حل شده

در این بخش برای روشن شدن مفاهیم گراف چند مسئله حل می‌کنیم:

مثال ۱-۷: گراف زیر را در نظر بگیرید

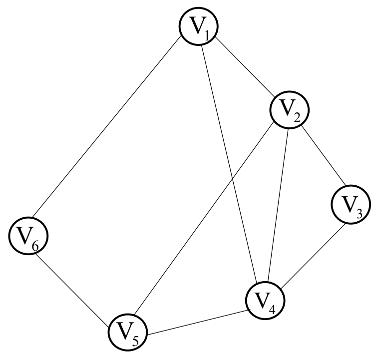


پیمایش عمقی یا DFS گراف که از رأس A شروع می‌شود، به‌صورت زیر خواهد بود:

نخست رأس A ملاقات می‌شود سپس رأس B، D، G و I پردازش می‌شوند. سپس گره H را ملاقات می‌کنیم به طرف رأس همجوار چپ رأس H رفته و رأس E را ملاقات می‌کنیم. سپس چون رأس B که رأس همجوار E می‌باشد قبلاً ملاقات شده، رأس C را پردازش می‌کنیم و در نهایت گره F را ملاقات می‌کنیم. بنابراین خواهیم داشت:

A B D G I H E C F

مثال ۲-۷: گراف زیر را در نظر بگیرید:



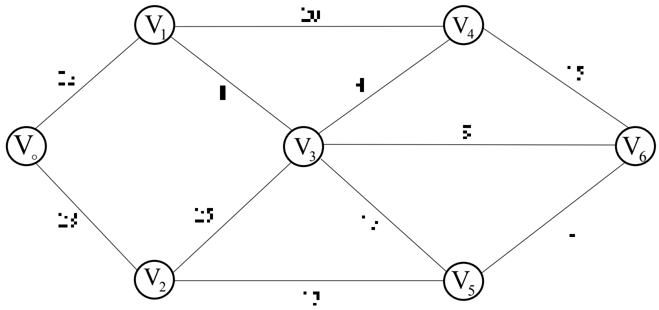
پیمایش ردیفی یا BFS گراف بالا را با شروع از رأس V_1 به‌صورت زیر ارائه می‌دهیم:

نخست رأس V_1 را ملاقات می‌کنیم سپس رأس همجوار با V_1 را ملاقات می‌کنیم. بنابراین رئوس V_6 و V_4 و V_2 را پردازش می‌کنیم. سپس V_6 را از صف

حذف کرده و رأس‌های همجوار V_6 را که عبارت است از V_5 ملاقات می‌کنیم. حال رأس همجوار V_5 را پردازش می‌کنیم. ولی V_4 قبلاً ملاقات شده است. بنابراین رأس‌های همجوار V_4 که عبارت است از V_3 را پردازش می‌کنیم. لذا خواهیم داشت:

$$V_1\ V_6\ V_4\ V_2\ V_5\ V_3$$

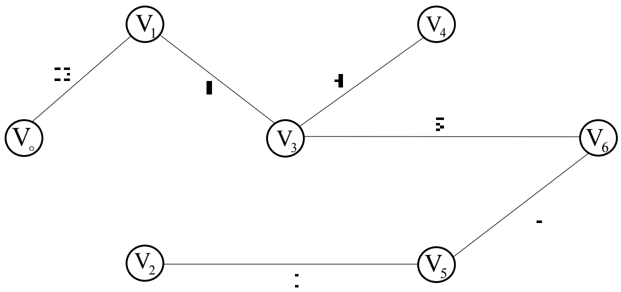
مثال ۳-۷: گراف وزن‌دار زیر را در نظر بگیرید:



می‌خواهیم با استفاده از الگوریتم راشال، درخت پوشای کمینه را به‌دست آوریم:

- نخست از رأس V_1 به V_3 حرکت می‌کنیم.
- سپس از رأس V_6 به V_5 حرکت می‌کنیم.
- در مرحله بعدی از رأس V_3 به V_4 حرکت می‌کنیم.
- یال e_{46} را انتخاب نمی‌کنیم چون یک دور ایجاد می‌شود.

به همین ترتیب بقیه یال‌ها را انتخاب می‌کنیم در نهایت درخت پوشای کمینه زیر حاصل خواهد شد:



مجموع هزینه‌های درخت برابر است با:

مجموع هزینه‌ها $= 1+3+4+9+17+23=57$

۸-۷ تمرین‌های فصل

۱. تابعی بنویسید که با توجه به یک ماتریس همجواری و دو راس از گراف، موارد زیر را محاسبه کند:

الف) تعداد مسیرهای با طول معین بین آنها

ب) تعداد کل مسیرهای موجود بین آنها

۲. تابعی بنویسید که گرافی را دریافت کرده و مشخص کند آن گراف متصل است یا خیر.

۳. تابعی بنویسید که الگوریتم پریم را برای ساخت درخت پوشای پیاده‌ساز

٤. تابعی بنویسید که تعداد راس‌ها و اینکه هر رأس مجاور کدام راس‌هاست را از کاربر گرفته و ماتریس مجاورتی آن را چاپ کند. ورودی برنامه به‌صورت مجموعه‌های V و E باشد. برنامه باید برای گراف‌های جهت دار و بدون جهت کار کند.

۵. تابعی بنویسید که گرافی را خوانده و جستجوی عمقی آن را چاپ کند. گراف یکبار به‌صورت لیست مجاورتی و بار دیگر به‌صورت ماتریس مجاورتی باشد.

٦. تابعی بنویسید که گرافی را خوانده و جستجوی ردیفی آن را چاپ کند. گراف یکبار به‌صورت لیست مجاورتی و بار دیگر به‌صورت ماتریس مجاورتی باشد.

۷. تابعی بنویسید که گرافی وزن‌دار را خوانده و با استفاده از الگوریتم پریم درخت پوشای مینممم آن را چاپ کند.

۸. الگوریتمی بنویسید که کوتاه‌ترین مسیرهای رأس 0 تا تمام رئوس دیگر گراف را به ترتیب غیرنزولی پیدا نماید.

۹. با توجه به گراف کامل با n رأس، نشان دهید که حداکثر تعداد مسیرهای بین رئوس برابر $((n-1)!)$ می‌باشد.

۱۰. نشان دهید که اگر T یک درخت پوشا برای گراف بدون جهت G باشد، آنگاه اضافه کردن یک یال مانند e موجب ایجاد یک حلقه منحصر بفرد می‌گردد.

۱۱. با توجه به گراف کامل با n راس، نشان دهید که تعداد درخت‌های پوشای حداقل، برابر با $2^{n-1}-1$ می‌باشد.

۱۲. برای گراف بدون جهت G با n راس، ثابت کنید که موارد زیر یکسان و معادل

هستند:

الف) G یک درخت می‌باشد.

ب) G متصل می باشد، اما اگر هر یک از یال‌های آن حذف شود گراف حاصل متصل نمی‌باشد.

ج) G فاقد حلقه بوده و دارای n-1 یال می‌باشد.

د) برای هر راس مجزا تنها یک مسیر ساده از u به v وجود دارد.

۹-۷ پروژه‌های برنامه‌نویسی

۱. برنامه‌ای بنویسید که یک گراف وزن‌دار را از ورودی دریافت نماید، سپس کوتاه‌ترین مسیر در این گراف را پیدا نموده و آن را به‌عنوان خروجی به‌صورت گرافیکی نمایش دهد.

فصل هشتم

مرتب‌سازی (sorting)

اهداف

در پایان این فصل شما باید بتوانید:

- ✓ مفهوم مرتب‌سازی را بیان کرده و دلیل استفاده از آن را بیان کنید.
- ✓ انواع مرتب‌سازی‌ها و جستجوها را مقایسه کرده و تحلیل کنید.
- ✓ کاربردهای مرتب‌سازی را بیان کنید.
- ✓ روش‌های مرتب‌سازی را تشریح کرده و در مورد مرتبه زمانی آنها بحث کنید.
- ✓ روش‌های مرتب‌سازی را با توجه به شرایط مسئله با یکدیگر مقایسه کنید؟

سؤال‌های پیش از درس

۱. اگر بخواهید یک شماره تلفن از دفترچه شماره تلفن پیدا کنید چه کارهایی را انجام می‌دهید؟
۲. اگر بخواهید دفترچه شماره تلفن خود را مرتب کنید چه کارهایی را انجام می‌دهید؟
۳. آیا برای اینکه به اطلاعات به‌صورت مرتب دسترسی پیدا کرد، مرتب‌سازی تنها راه حل آن است؟

مقدمه

مرتب کردن و جستجوی اطلاعات از عملیات اساسی و اصلی در علم کامپیوتر است. مرتب کردن عبارت است از، عمل تجدید آرایش داده‌ها بلک ترتیب مشخص، مثلاً برای داده‌های عددی به ترتیب صعودی یا نزولی اعداد یا برای داده‌های کاراکتری ترتیب الفبایی آنها مرتب‌سازی صورت می‌گیرد. به همین ترتیب، جستجو کردن عبارت است از عمل پیدا کردن محل یک عنصر داده شده در بین مجموعه‌ای از عناصر می‌باشد.

مرتب کردن و جستجوی اطلاعات، اغلب روی یک فایل از رکوردها به کار می‌رود، از این رو لازم است چند اصطلاح استاندارد را یادآوری کنیم. هر رکورد در یک فایل می‌تواند چند فیلد داشته باشد اما فیلد ویژه‌ای وجود دارد که مقدارهای آن به‌طور منحصر به فردی، رکوردهای داخل فایل را معین می‌کند. چنین فیلدی یک کلید اولیه یا اصلی نامیده می‌شود، مرتبکردن فایل معمو لاً به مرتب کردن نسبت به کلید اولیه خاصی گفته می‌شود و جستجوی اطلاعات در فایل به جستجوی رکورد با مقدار کلیدهای معین گفته می‌شود

۱–۸ مرتب کردن

فرض کنید A یک لیست از عناصر $A_1, A_2, ..., A_n$ در حافظه باشد، منظور از مرتب کردن A عمل تجدید آرایش محتوای A است به‌طوری‌که با ترتیب صعودی (عددی یا فرهنگ لغتی) یا نزولی باشند، به‌طوری‌که:

$$A_1 \le A_2 \le A_3 ... \le A_n$$

اگر عمل مرتب‌سازی بر روی رکوردهای موجود در حافظه انجام شود، مرتب‌سازی را مرتب‌سازی داخلی(internal) می‌نامند و اگر بر روی رکوردهای موجود در حافظه جانبی (مانند دیسک‌ها)صورت گیرد، مرتب‌سازی را مرتب‌سازی خارجی (External) می‌نامند. در این کتاب بیشتر 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 اگر $i<j$ و در لیست ورودی $k_i=k_j$ باشد(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، رکوردهای با کلیدهای مساوی را به همان ترتیب قبل از عمل مرتب‌سازی نگهداری می‌کند.

به‌عنوان مثال فرض کنید اگر رشته ورودی به‌صورت $٤,٢^{(١)},٥,١,٢^{(٢)}$ باشد اگر رشته مرتب شده به‌صورت $٤,٢^{(٢)},٢^{(١)},١$ باشد الگوریتم مرتب‌سازی پایدار می‌باشد و اگر رشته مرتب شده به‌صورت $٤,٥,٢^{(١)},٢^{(٢)},١$ باشد الگوریتم مرتب‌سازی پایدار نمی‌باشد. توجه کنید شماره‌های بالای عدد۲ نشان 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.

۲-۸ مرتب‌سازی با آدرس

عمل مرتب‌سازی می‌تواند بر روی خود رکوردها یا بر روی جدولی از اشاره‌گرها صورت گیرد. به‌عنوان مثال شکل ۱-۸ (الف) را که در آن فضای فایلی با ۵ رکورد نشان داده شده در نظر بگیرید. اگر فایل بر حسب شماره کلید به‌طور صعودی مرتب گردد، نتیجه آن در شکل ۱-۸ (ب) مشاهده می‌گردد.

	کلید	سایر فیلدها
رکورد ۱	4	DDD
رکورد ۲	2	BBB
رکورد ۳	1	AAA
رکورد ٤	5	EEE
رکورد ٥	3	CCC

فایل

(الف) فایل اصلی

کلید	سایر فیلدها
1	AAA
2	BBB
3	CCC
4	DDD
5	EEE

فایل

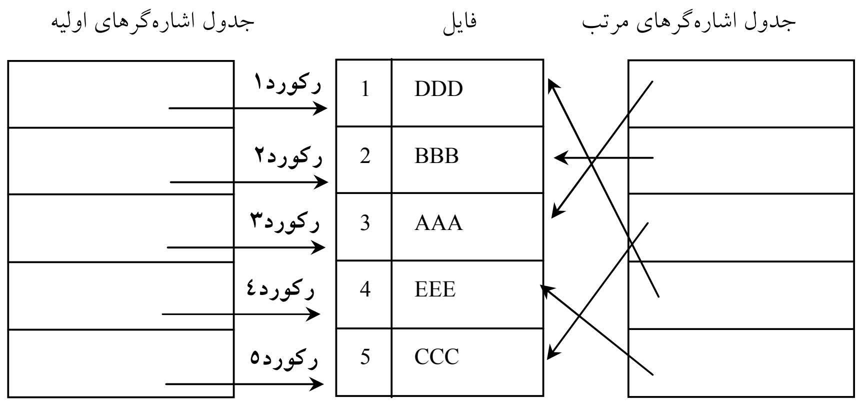
(ب) فایل مرتب

شکل ۱-۸ دو فایل با ۵ رکورد

Deleted: :

فرض کنید مقدار داده‌های هر رکورد فایل شکل ۱-۸ (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 ۲-۸)



شکل۲-۸ 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.

۳-۸ مرتب‌سازی یا جستجو

به دلیل ارتباط تنگاتنگ بین مرتب‌سازی و جستجو، معمولاً در هر کاربردی این سؤال 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 داردبه 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 مسئله تصمیم بگیرد. وقتی تصمیم به عمل مرتب‌سازی گرفته شد،

باید تصمیماتی راجع‌به روشهای مرتب‌سازی و فیلدهایی که باید مرتب گردند، اخذ شود. هیچ روش مرتب‌سازی وجود ندارد که از هر جهت از سایر روش‌ها ممتاز باشد. برنامه‌نویس باید مسئله را به دقت بررسی کرده و با توجه به نتایجی که انتظار می‌رود، روش مناسبی را انتخاب کند.

۸–۴ ملاحظات کارایی

همان‌طور که در این فصل مشاهده خواهید کرد روشهای متعددی برای مرتب‌سازی 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 مرتب‌سازی فقط یک بار اجرا شود در اینصورت زمان و فضای کافی برای آن وجود دارد و جالب نیست که برنامه‌نویس روزها وقت صرف کند تا بهترین روش مرتب‌سازی را جهت به‌دست 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 و غیره.

در محاسبه زمان همانطور که در فصل اول مشاهده کردید 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 است‌همچنین 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.

۵-۸ مقایسه روش‌های مرتب‌سازی

با توجه به مفهوم مرتبه یک روش مرتب‌سازی، 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 مرتب‌سازی $O(n\log n)$ است، دلیلی برای انتخاب این روش مرتب‌سازی نیست! ارتباط بین طول لیست یا 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) به‌طور کامل مرتب می‌گردند در حالی که اگر داده‌ها به ترتیب معکوس مرتب باشند. زمان لازم برای مرتب‌سازی 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 براساس بدترین حالت یا حالت متوسط انتخاب کنیم. به‌طور کلی می‌توان گفت که بهترین روش مرتب‌سازی که در همه موارد قابل استفاده باشد وجود ندارد.

به‌طورکلی 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 مرتب‌سازی 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 الگوریتم‌های 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 مرتب‌سازی درجا است. در این مرتب‌سازی فضای اضافی مورد نیاز $O(n)$ است. یعنی مرتب‌سازی درجا عمل مرتب‌سازی را در آرایه یا لیستی که حاوی این عناصر است انجام می‌دهد. فضای اضافی مورد نیاز، صرفنظر از اندازه مجموعه‌ای که باید مرتب گردد، به‌صورت 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 اصطلاحا هوشمندی وجود دارند که از حداقل زمان و حافظه استفاده می‌کنند. این الگوریتم‌ها همان الگوریتم‌های درجا هستند که از درجه $O(n \log n)$ می‌باشند.

۶–۸ روش‌های مرتب‌سازی

در این بخش سعی کنیم الگوریتم‌های مرتب‌سازی را بطور مبسوط مورد بررسی قرار دهیم. همچنین زمان‌های الگوریتم‌های ارائه شده را مورد ارزیابی قرار داده و آنها را با هم مقایسه می‌کنیم.

۱-٦-۸ 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 لیست مرتب شود.

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.

در هر یک از مثالها، آرایه‌ای از اعداد صحیح مانند A را در نظر می‌گیریم که باید مرتب گردند.

به‌عنوان مثال، لیست 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 می‌گیرد.
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 لیست به‌صورت 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.

به‌طور کلی پس از تکرار iام، x[n-i] در موقعیت مناسب خود قرار می‌گیرد. پس از گذر دوم، محتویات لیست به‌صورت 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. لیستی به طول n حداکثر در n-1 تکرار، مرتب می‌گردد.

تکرارهای مختلف منجر به مرتب شدن لیست موردنظر می‌گردد که 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 الگوریتم 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[1] , A[0]) صورت می‌گیرد.

۲. نشان دادیم که برای مرتب‌سازی لیستی به طول n، حداکثر n-1 تکرار لازم است، اما در مثال قبلی که تعداد عناصر لیست ۸ بود، لیست 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 باشد، 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 مرتب‌سازی حبابی

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

• **پیچیدگی الگوریتم مرتب کردن حبابی**

اگر بهبودهای مطرح شده، به الگوریتم اعمال نشوند، تحلیل آن ساده است. برای مرتب‌سازی لیستی 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:

$$(n-1)+(n-2)+(n-3)+\ldots+(n-k)=\frac{(2nk-k^{2}-k)}{2}$$

می‌توان 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 مقداردهی اولیه به متغیر (flag) در هر

گذر و قرار دادن مقدار ۱ در این متغیر (یک بار برای هر جابجایی) باید صورت گیرد.

مرتب‌سازی حبابی در صورتی که لیست به‌طور کامل یا تقریباً کامل) مرتب باشد از مرتبه O(n) است. بنابراین مرتب‌سازی حبابی در بردار مرتب بهترین عملکرد و در بردار نامرتب بدترین عملکرد را دارد. این الگوریتم پایدار (stable) می‌باشد.

ویژگی‌های مرتب‌سازی حبابی
✓ مرتب‌سازی حبابی در صورتی که لیست به‌طور کامل یا تقریباً کامل مرتب باشد از مرتبه O(n) است. ✓ مرتب‌سازی حبابی در بردار مرتب بهترین عملکرد و در بردار نامرتب بدترین عملکرد را دارد. ✓ این الگوریتم پایدار (stable) می‌باشد.

۲-۶-۸ مرتب‌سازی انتخابی (selection sort)

فرض کنید آرایه A با n عنصر A[0] , A[1] ,… , A[n-1] در حافظه موجود باشد. الگوریتم مرتب کردن انتخابی برای مرتب کردن آرایه A به‌صورت زیر عمل می‌کند:

نخست کوچک ترین عنصر داخل لیست را پیدا می‌کند و آن را در مکان اول لیست قرار می‌دهد (جای آن را با عنصر اول لیست عوض می‌کند) آنگاه کوچک‌ترین عنصر دوم داخل لیست را پیدا میکند و آن را در مکان دوم لیست قرار می‌دهد و الی آخر تا در نهایت لیست مرتب شود.

به‌عنوان مثال فرض کنید لیست زیر باید به‌طور 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 موقعیت 4 می‌یابیم و این عنصر را با عنصر اول تعویض می‌کنیم و در نتیجه کوچک‌ترین عنصر لیست در ابتدای لیست قرار می‌گیرد.

11 , 33 , 44 , 77 , 88 , 22 , 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.

در تابع زیر الگوریتم مرتب‌سازی انتخابی ارائه شده است:

الگوریتم مرتب‌سازی انتخابی

```
void  selection (int A[ ] , int  n )
 {
        int i,j , minpost , temp ;
       for (i= 0  ; i< n – 1 ; i++)
         {
            minpos = i ;
            for (j= i + 1 ; j< n ; j++)
               if (A[j] < A[minpos])
                    minpos=j ;
            temp=A[minpos];
            A[minpos]=A[i];
            A[i]=temp ;
         }
  }
```

• **پیچیدگی الگوریتم مرتب‌سازی انتخابی**

در اولین تکرار حلقه خارجی i، حلقه تکرار داخلی j به تعداد n-1 بار اجرا می‌شود. در مرحله دوم به تعداد n-2 بار تکرار می‌شود تا الی آخر.

بنابراین خواهیم داشت:

$$(n-1)+(n-2)+\ldots+1=\frac{n(n-1)}{2}\in O(n^2)$$

این الگوریتم پایدار (stable) نیست و ممکن است ترتیب عناصر مساوی را در آرایه حفظ نکند.

ویژگی‌های مرتب‌سازی انتخابی
✓ مرتب‌سازی انتخابی در همه موارد دارای مرتبه زمانی $O(n^2)$ می‌باشد. ✓ این الگوریتم پایدار (stable) نیست و ممکن است ترتیب عناصر مساوی را در آرایه حفظ نکند.

۳-۶-۸ مرتب‌سازی سریع (Quick sort)

مرتب‌سازی سریع الگوریتمی از نوع تقسیم و غلبه است که دارای میانگین زمانی بسیار مناسبی می‌باشد (به کتاب طراحی الگوریتم مراجعه فرمائید). روش مرتب‌سازی سریع ارائه 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 تعویض‌ها 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 به‌طور 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 به‌عنوان محور در نظر 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 به‌طور بازگشتی مرتب کنیم (یعنی، عمل فوق را روی آن انجام دهیم).

برای عمل فوق بدین ترتیب عمل می‌کنیم:

از انتهای سمت راست لیست اولین کوچک‌ترین عنصر از محور (عدد 68) را

پیدا می‌کنیم و ابتدای سمت چپ لیست اولین بزرگ‌ترین عنصر از محول (عـدد 84) را پیدا می‌کنیم.

75 , 70 , 65, 84 , 98 , 78, 100 , 93 , 55 , 61, 81 , 68

سپس جای این عناصر را تعویض می‌کنیم.

75 , 70 , 65, 68 , 98 , 78 , 100 , 93 , 55 , 61 , 81 , 84

جستجو را از سمت راست ادامه می‌دهیم تا عنصر دیگری کـه کوچـک‌تـر از 75 (عدد 61) پیدا شود و سمت چپ ادامه می‌دهیم تا عنصر دیگـر بـزرگ‌تـر از 75 (عـدد 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 پیـدا مـی‌کـنیم که در جستجوی قبلی از چپ پیدا شده بود.

75 , 70 , 65, 68 , 61 , 55 , 100 , 93 , 78 , 98, 81 , 84

در این جا اشاره‌گرهای مربوط به جستجوهای چـپ و راسـت بـا هـم برخـورد می‌کند و بیانگر این است که جستجو خاتمـه یافتـه اسـت. اکنـون 55 را بـا محـور 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 کوچک‌تر و تمـام عناصـر راست آن از آن بزرگ‌تر 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

برای هر کدام از این لیست‌ها، با انتخاب عنصر محوری در هر کدام، روند قبلی را تکرار کنید.

الگوریتم مرتب‌سازی سریع را پیاده‌سازی می‌کنیم:

الگوریتم مرتب‌سازی سریع

تابع ()Split برای تقسیم کردن آرایه به‌کار 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], temp;
          while (left  < right)
          {
               while (A[ right ] > pivot)
                     right - - ;
              while (left < right && A[left] <=  pivot)
                   left++;
                if (left < right)
                {
                      temp= A[left] ;
                     A[left]=A[right];
                     A[right]=temp;
                }
             /*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 معمو لاً با تعداد دفعات مقایسه مورد نیاز 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 $O(n^۲)$ 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 می‌گیرد و الی آخردر نتیجه، مجموع اً تعداد:

$$f(n) = n + (n-1) + \ldots + 2 + 1 = \frac{n(n+1)}{2} \in O(n^2)$$

مقایسه انجام شود. ملاحظه می‌کنید که این عدد برابر پیچیدگی الگوریتم مرتب کردن حبابی است.

پیچیدگی O(nlogn) حالت میانگین، از این واقعیت ناشی می‌شود که به‌طور متوسط، هر مرحله ساده‌سازی در الگوریتم دو لیست کوچک‌تر تولید می‌کند. بنابراین:

با ساده شدن لیست اول، ۱ عنصر در جای خود قرار می‌گیرد 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 می‌کند و از این دو تقریب اً $\log n$ سطح ساده سازی وجود دارد. علاوه بر این هر سطح حداکثر از n مقایسه استفاده می کند. بنابراین: $f(n) \in O(n \log n)$ می‌باشد. در واقع تحلیل ریاضی و ملاحظات تجربی هر دو نشان می دهند که:

$$f(n) \cong ۱.٤[n \log n]$$

تعداد مقایسه‌های مورد انتظار برای الگوریتم quick sort می‌باشد.

قابل ذکر است که این الگوریتم پایدار نمی‌باشد. و پیچیدگی این الگوریتم در حالت کلی به‌صورت زیر است:

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

بنابراین با توجه به مطالب ارائه شده داریم:

ویژگی‌های مرتب‌سازی سریع
✓ در این الگوریتم انتخاب عنصر محوری (pivot) تاثیر مهمی در سرعت اجرای آن دارد.
✓ در مرتب‌سازی سریع بدترین حالت زمانی رخ می‌دهدکه عنصر محوری کوچک‌ترین یا بزرگ‌ترین عنصر آرایه باشد.
✓ در مرتب‌سازی سریع اگر آرایه از قبل مرتب باشد، بدترین حالت رخ می‌دهد 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) می‌باشد.
✓ اگر مرتب‌سازی سریع در بهترین حالت خود باشد در آنصورت سریع‌ترین روش مرتب‌سازی خواهد بود.
✓ این الگوریتم پایدار (متعادل) نیست.
✓ در حالت کلی در آرایه‌های مرتب دارای بدترین عملکرد و در آرایه‌های نامرتب دارای بهترین عملکرد می‌باشد.

٤-٦-٨ مرتب‌سازی درجی (Insertion sort)

در مرتب‌سازی درجی، ابتدا عنصر دوم با عنصر اول لیست مقایسه می‌شود و در صورت لزوم با عنصر اول جابجا می‌شود به‌طوری‌که عناصر اول و دوم تشکیل یک لیست مرتب دوتایی را بدهند. سپس عنصر سوم به ترتیب با دو عنصر قبلی خود یعنی

عناصر دوم و اول مقایسه و درجای مناسبی قرار می‌گیرد به طوریکه عناصــر اول و دوم و سوم تشکیل یک لیست مرتب سه تایی را بدهند. سپس عنصر چهارم به ترتیب با سه عنصر قبلی خود یعنی با عناصــر ســوم و دوم و اول مقایســه و در جــای مناسـب قــرار می‌گیرد به طوریکه عناصــر اول و دوم و ســوم و چهــارم تــشکیل یــک لیــست مرتــب چهارتایی را بدهند و در حالت کلی عنصر i ام با i-1 عنصر قبلی خود مقایسه می‌گردد تا در مکان مناسب قرار گیرد به‌طوری‌که i عنصر تشکیل یــک لیــست مرتــب iتــایی را بدهند و این روند تا مرتب شدن کامل لیست ادامه می‌یابد. یا به‌صورت دقیق‌تر:

:Deleted

:Deleted می

:Deleted به

مرحله ۱: A[1] خودش به‌طور بدیهی مرتب است.

مرحله ۲: A[2] را یا قبل از یا بعـد از A[1] درج مـی‌کنـیم طـوری کـه 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.

مرحله n: A[n] را در مکان صحیح خود در A[1]، A[2]، ...، A[n-1] به گونه‌ای درج می‌کنیم که کل آرایه مرتب باشد.

:Deleted

تابع زیر الگوریتم مرتب‌سازی درجی را پیاده‌سازی می‌کند:

الگوریتم مرتب‌سازی درجی

```
void        Insertion sort (int A[ ] , int  n)
{
             int i,j,temp;
             for (i= 1 ; i< n ; i++)
             {
                    temp= A[i] ;
                    for (j=i ; j> 0  &&  A[j-1]  > temp ; j--)
                        A[j] = A[j-1];
                    A[j]=temp ;
             }
}
```

• 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:

$$f(n)=1+2+....+(n-1)=\frac{n(n-1)}{2}\in O(n^2)$$

اگر آرایه‌ای که در اختیار الگوریتم مرتب‌سازی درجی قرار میگیرد تقریباً مرتب باشد آنگاه الگوریتم از مرتبه O(n) خواهد بود.

$$\underbrace{1+1+...+1+1}_{n-1}=n-1\in O(n)$$

این الگوریتم جزء الگوریتمهای پایدار می‌باشد و همانگونه که اشاره شد در یک بردار مرتب بهترین حالت و برای یک بردار مرتب شده معکوس بدترین حالت را دارد. این الگوریتم برای nهای کوچک روش بسیار مناسبی است و ثابت شده است که برای $n \leq ۲۰$ سریع‌ترین روش مرتب‌سازی است.

ویژگی‌های مرتب‌سازی 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 مرتب‌سازی می‌باشد.

۵-۶-۸ مرتب‌سازی هرمی

قبلاً در فصل درخت‌ها، درخت Heap و مرتب‌سازی هرمی را به‌طور کامل بررسی کردیم‌هرم 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 به‌دست 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 به‌صورت بازگشتی نیست.

ویژگی‌های مرتب‌سازی هرمی
✓ کلیه اعمال در مرتب‌سازی هرمی از مرتبه O(nlogn) است ✓ در این روش درخت heap روی آرایه ساخته می‌شود. ✓ مرتب‌سازی هرمی از نوع درجا می‌باشد. ✓ این الگوریتم پایدار (stable) نمی‌باشد.

٦-٦-٨ مرتب‌سازی ادغامی (Merge sort)

در این نوع مرتب‌سازی، نخست لیست اقلامی که قرار است براساس کلید خاصی مرتب شوند به دو قسمت تقسیم می‌شوند. هر کدام از لیست‌ها دوباره براساس نیاز به زیرلیست‌های کوچک‌تر تقسیم می‌شوند، زیرلیست مرتب‌شده، سپس نتیجه آنها با هم ادغام می‌شوند. این عمل تا زمانی‌که کل لیست مرتب نشده است، ادامه می‌یابد. این روش، را مرتب‌سازی ادغامی می‌نامند.

در کل منظور از ادغام دو لیست عبارتست از:

فرض کنید دو لیست مرتب موجود است، هدف ایجاد یک لیست مرتب از ترکیب دو لیست می‌باشد، لیست حاصل را ادغام در لیست اولیه می‌گویند.

همان‌طور که تا حال متوجه شدید، برای این مسئله، روش تقسیم و حل را به‌راحتی می‌توان به‌کار برد.

مراحل زیر را با توجه به روش تقسیم و حل برای این مسئله مرتب‌سازی در نظر بگیرید:

- تقسیم مسئله به دو زیرمسئله کوچک‌تربا طول تقریباً یکسان
- مرتب‌سازی دو زیرمسئله به‌طور بازگشتی با به‌کارگیری مرتب‌سازی ادغامی
- ترکیب زیرمسئله‌های مرتب‌شده با هم برای تولید لیست مرتب‌شده.

الگوریتم زیر، پیاده‌سازی مراحل فوق را نمایش می‌دهد.

• **الگوریتم مرتب‌سازی ادغامی**

مسئله: مرتب‌سازی n عنصر به ترتیب غیرنزولی

ورودی: عدد صحیح و مثبت n، لیستی (آرایه‌ای) از اقلام که از 0 تا n اندیس‌گذاری شده‌اند.

خروجی: لیست مرتب براساس کلید به ترتیب غیرنزولی.

الگوریتم مرتب‌سازی ادغامی

```
void    MergeSort  (low  ,  high)
{
    if  (low  <  high)
    {
        // Divide P into Subproblems.
        mid = (low  +  high) / 2  ;
        // Solve the Subproblems.
        Mergesort (low , mid) ;
        Mergesort (mid + 1 , high) ;
        // Combine the Solutions.
        Merge (low , mid , high) ;
    }
}
```

الگوریتم طراحی شده در بالا با روش تقسیم و حل بوده و مراحل این روش را دارا می‌باشد. تابع Merge در الگوریتم بالا کار ترکیب زیر لیست‌های مرتب‌شده را بر عهده دارد. در زیر تابع Merge را ارائه می‌دهیم.

• **الگوریتم ادغام دو لیست مرتب (آرایه‌ای از اقلام)**

مسئله: ادغام دو زیرلیست مرتب‌شده S که در MergeSort ایجاد شده‌اند.

ورودی: high, mid, low و زیرلیست S که از low تا high اندیس‌گذاری شده است. که در آن کلیدها از low تا mid و از mid+1 تا high از قبل مرتب شده‌اند.

خروجی: زیرلیست S که از low تا high اندیس‌گذاری شده است به‌صورت مرتب شده باشد.

الگوریتم ادغام دو لیست مرتب (آرایه‌ای از اقلام)

```
viod     Merge  (low , mid ,  high)
{
     // A local Array needed for the merging
     elementtype  L [low.. high] ;
     h = low ;  i = low ;  j = mid + 1 ;
     while ((h ≤  mid) &&  (j ≤  high))
     {
          if (S[ h ]  ≤  S[ j ]) {
                    L[i] = S[h] ;
                    h + +;
          }
          else{
                    L[ i ] = S[ j ] ;
                    j + +;
          }
          i + + ;
     } // End  of  while
     if (h  >  mid)
          for (k = j   ;  k <= high ;  k++)
          {
                    L[ i ] = S[ k ] ;
                    i++ ;
          }
     else
          For (k = h ;  k <= mid ;  k++)
          {
                    L[ i ] = S[ k ] ;
                    i++ ;
          }
     for (k = low   ;  k <= high ; k++)
                    S[ k ] = L[ k ] ;
}// End of Function
```

این تابع برای ادغام دو لیست مرتب $S[low..mid]$ و $S[mid+1,high]$ از یک لیست کمکی به نام $L[low..mid]$ داخل تابع به‌طور محلی استفاده می‌کند و بعد از ادغام دو لیست در L، نتیجه را مجدداً در لیست S جهت ارسال به تابع MergeSort قرار می‌دهد.

روش کار این تابع بدین صورت است که، عناصر اول دو بخش لیست S را با هم مقایسه می‌کند عنصر کوچک‌تر را در لیست L قرار می‌دهد. این کار را تا زمانیکه یک لیست به اتمام نرسیده انجام می‌دهد. در نهایت اگر عناصری از یک لیست واقع در S باقی مانده باشند، تمامی عناصر باقیمانده را، در لیست محلی L قرار می‌دهیم.

برای روشن‌شدن مطالب به مثال توجه کنید:

فرض کنید لیست حاوی اعداد زیر موجود باشند:

۲۰	۱۰	۳۷	۲۵	۸	۱۵	۱۷	۱۲

آرایه از ۸ عنصر تشکیل‌شده که داخل لیست S، از اندیس 0 تا ۷ قرار گرفته‌اند. اعمال زیر انجام می‌گیرد:

۱. تقسیم $S[0..7]$ به دو زیر لیست:

$S[0..3]$ و $S[4..7]$

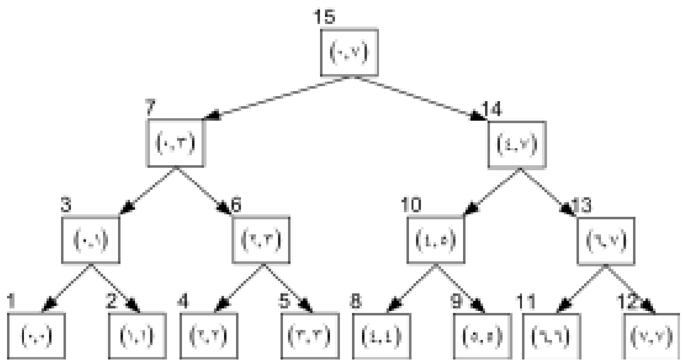
۲. مرتب‌سازی دو زیرلیست با روش مرتب‌سازی ادغامی.

۳. ادغام دو لیست و ارائه لیست مرتب:

۳۷	۲۵	۲۰	۱۷	۱۵	۱۲	۱۰	۸

مرحله دوم کار ممکن است خود شامل چند مرحله تقسیم به زیرمسائل را در بر داشته باشد. با به‌کارگیری الگوریتم بالا درخت فراخوانی زیر برای مثال بالا تشکیل می‌شود:

در شکل (۳-۸) اعدادی که در گوشه چپ گره‌ها ظاهر شده‌اند ترتیب فراخوانی و مرتب‌سازی مسائل کوچک که ناشی از تقسیم مسئله اصلی به زیرمسئله‌ها باشد، را نمایش می‌دهند.



شکل ۳–۸ درخت فراخوانی MergeSort (0,7)

درخت ارائه شده در شکل (۲–٤)، فراخوانی‌های بازگشتی تولید شده توسط الگوریتم MergeSort روی ۸ عنصر را نمایش می‌دهد. که در آن، در هر گره مقدار اندیس‌های low و high به ازای هر فراخوانی نمایش داده شده است.

- **تحلیل پیچیدگی زمانی در بدترین حالت برای الگوریتم مرتب‌سازی ادغامی**

در تحلیل این الگوریتم باید توجه داشته باشیم که دستور مقایسه و دستور انتساب را می‌توان به‌عنوان اعمال اصلی در نظر گرفت. که در اینجا، دستور مقایسه را به‌عنوان عمل اصلی در نظر می‌گیریم. بنابراین خواهیم داشت:

$$T(n)=\begin{cases}\theta(1) & \text{اگر } n=1\\ 2T\left(n/2\right)+\theta(n) & \text{اگر } n>1\end{cases} \qquad (8-1)$$

که در آن $T(n/2)$ زمان بازگشت وحل بوده و $\theta(n)$ زمان لازم برای ادغام می‌باشد. با کمی دقت در معادله (۱–۸) و الگوریتم می‌توان چنین نوشت:

$$T(n)=\begin{cases}a & \text{اگر } n=1\\ 2T\left(n/2\right)+Cn & \text{اگر } n>1\end{cases}$$

رابطه بازگشتی بالا با روش‌های مختلفی (که در فصل اول اشاره کردیم) قابل حل می‌باشد.

چنانچه n توانی از ۲ باشد آنگاه می‌توان نوشت $n=2^k$ و می‌توانیم با روش

تکرار و جایگزینی چنین بنویسیم:

$$\begin{aligned} T(n) &= ۲\left(۲T\left(\tfrac{n}{۴}\right) + C\tfrac{n}{۲}\right) + Cn \\ &= ۴T\left(\tfrac{n}{۴}\right) + ۲C \\ &= ۴\left(۲T\left(\tfrac{n}{۸}\right) + C\tfrac{n}{۴}\right) + ۲Cn \\ &= \ldots \\ &= ۲^k T(۱) + kCn \end{aligned}$$

که در آن:

$$n = ۲^k \Rightarrow k = \text{Log } n$$

بنابراین:

$$T(n) = an + Cn \text{ Log } n \qquad \textbf{(۸-۲)}$$

از رابطه (**۲-۸**) بوضوح خواهیم داشت:

$$T(n) \in \theta(n \text{ Log } n)$$

حال ویژگی‌های مرتب‌سازی ادغامی را در زیر ارائه می‌دهیم:

ویژگی‌های مرتب‌سازی ادغامی
✓ مرتبه اجرایی این الگوریتم همواره O(nlogn) است ✓ مرتب‌سازی ادغام معمولا برای مرتب کردن فایلها استفاده می‌شود. ✓ 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 مرتب‌سازی درخت دودوئی

در این روش از درخت‌های جستجوی دودوئی (BST) برای مرتب‌سازی استفاده می‌شود. اگر درخت BST به‌صورت inorder پیمایش شود، دنباله به‌دست آمده

به‌صورت صعودی خواهد بود.

کارایی نسبی این روش به ترتیب اولیه داده‌ها بستگی دارد. اگر آرایه ورودی کاملاً مرتب باشد، درخت جستجوی دودوئی به‌صورت 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:

$$۱+۲+۳+...+n=\frac{n*(n+۱)}{۲}$$

تعداد مقایسه‌ها از $o(n^2)$ است.

از طرف دیگر، اگر داده‌ها طوری سازماندهی شده باشند. برای عدد خاصی مانند a، نصفی از اعداد کوچک‌تر و نصف دیگر بزرگ‌تر از a باشند، درخت متعادل ایجاد می‌گردد. در چنین موردی عمق درخ ت 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 دامنه زیر می‌باشد:

$$d+\sum_{L=1}^{d-1}2^{L-1}*(L)\ ,\ \sum_{l=1}^{d}2^{L-1}*(L)$$

که در آن d عمق درخت می‌باشد.

می‌توان از طریق ریاضی نشان داد که عبارت بالا از مرتبه $O(n\log n)$ 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 مرتبه $O(n\log n)$ است. اما در بدترین حالت (ورودی مرتب) مرتب‌سازی درخت دودوئی از مرتبه $O(n^2)$ است.

توجه کنید روش مرتب‌سازی درخت دودوئی همانند مرتب‌سازی ادغامی به فضای کمکی به طول آرایه ورودی نیاز دارد که به‌صورت درخت BST جلوه می‌کند. پس این روش مرتب‌سازی درجا نمی‌باشد.

ویژگی‌های **مرتب‌سازی درختی** دودوئی
✓ مرتبه اجرایی این الگوریتم در بهترین حالت و حالت متوسط $O(n\log n)$ 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. ✓ مسئله پایدار بودن به دلیل فرض عدم وجود عناصر با کلیدهای یکسان مطرح نیست.

۸-۶-۸ مرتب کردن مبنایی (Radix sort)

مرتب کردن مبنایی روشی است که افراد بسیاری به‌طور شهودی از آن استفاده می‌کنند یا هنگامی که لیست بزرگی از اسامی را به‌صورت الفبا مرتب می‌کنیم از آن استفاده می‌کنیم. در اینجا مبنا ۲۶ است، علت آن ۲۶ حروف الفبای انگلیسی است. به‌طور مشخص لیست اسامی نخست بر اساس حرف اول هر اسم مرتب می‌شود. به بیان دیگر اسامی به ۲۶ 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 به‌صورت الفبایی مرتب می‌شود و الی آخر. اگر هیچ اسمی برای مثال، بیشتر از ۱۲ حرف نداشته باشد، اسامی حداکثر در ۱۲ مرحله به‌صورت 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 ۱۰ صف قرار می‌دهیم. سپس هر صف را با شروع از صفی که با رقم صفر شماره‌گذاری شده تا صفی که با رقم ۹ 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 این روش مرتب‌سازی ابتدا بر اساس ارقام کم ارزش صورت می‌گیرد.

شکل زیر مراحل مرتب‌سازی آرایه را با روش مرتب‌سازی مبنائی 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]		

12,92,33,25,86,57,37,48: آرایه بعد از گذر اول

گذر دوم: اعداد آرایه به‌دست 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	

12,25,33,37,48,57,86,92: آرایه بعد از گذر دوم

چون اعداد ۲ رقمی بودند، در ۲ گذر آرایه مرتب شد. اگر اعداد ۵ رقمی بودند، به 5 گذر نیاز داریم. الگوریتم این مرتب‌سازی به‌صورت زیر است:

الگوریتم مرتب‌سازی 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 n عنصری $A_1, A_2, ..., A_n$ داده شده است.فرض کنید d نمایش مبنا باشدمثلاً 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.

$$A_i = d_{i1}d_{i2}...d_{is}$$

الگوریتم مرتب کردن مبنایی نیازمند s مرحله، یعنی تعداد ارقام هر عنصر است. در مرحله k هر رقم d_{ik} با هر یک از d رقم مقایسه می‌شود. از این رو C(n) تعداد مقایسه‌ها برای الگوریتم به‌صورت زیر 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 $c(n) \in o(n^2)$. در بهترین حالت $s = \log_d^n$، از این رو $c(n) \in o(n \log n)$.

عیب دیگر مرتب‌سازی مبنایی این است که ممکن است به $d * n$ خانه حافظه احتیاج داشته باشد. این عیب را می‌توان با استفاده از لیست‌های پیوندی به جای آرایه، به حداقل رساند. با وجود این همچنان به 2*n خانه حافظه نیازمندیم.

ویژگی‌های مرتب‌سازی 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 روش‌های مرتب‌سازی

از چندین روش مرتب‌سازی ارا ه شده هیچکدام روش مناسب و خوبی نیستند. برخی از روش‌ها برای مقادیر کوچک n و برخی دیگر برای مقادیر بزرگ n مناسب هستند. مرتب‌سازی درجی زمانی که لیست به‌صورت جزئی مرتب شده باشد، خوب کار می‌کند و از آنجا که این روش حداقل سربار 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 سربار آن بیشتر از مرتب‌سازی سریع می‌باشد. مرتب‌سازی سریع بهترین میانگین را دارد، اما در بدترین حالت، زمان آن از مرتبه $o(n^2)$ خواهد شد. عملکرد مرتب‌سازی مبنائی بستگی به کلید و انتخاب مبناء دارد.

آزمایش‌هایی که روی روشهای مرتب انجام شده است، نشان می‌دهد که به ازای $n \leq ۲۰$ مرتب‌سازی درجی سریع‌ترین روش است. برای مقادیر بین 20 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 به‌طوری‌که مرتب‌سازی ادغامی برای زیرلیست‌های کمتر از ٤٥ از مرتب‌سازی سریع استفاده کند و مرتب‌سازی سریع نیز زمانی که طول زیرلیست‌ها کمتر از ۲۰ باشد از مرتب‌سازی درجی استفاده کند.

حال این مقایسه را از دیدگاه دیگری نیز بیان می‌کنیم.

کارایی مرتب‌سازی 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 عمل مقایسه کلیدها ارزان تمام می‌شود. اگر عکس این وضعیت برقرار باشد، مرتب‌سازی درجی پیشنهاد می‌گردد. اگر ورودی دو لیست پیوندی باشد، حتی اگر رکوردها بزرگ باشند، مرتب‌سازی درجی پیشنهاد می‌گردد. زیرا نیاز به جابجایی داده‌ها نیست. البته کارایی مرتب‌سازی 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 که در ورودی تصادفی، زمان لازم در heapsort دو برابر زمان لازم در quicksort است. اما در بدترین حالت heapsort مناسب‌تر از quicksort است. Heapsort همچنین برای مقادیر کوچک n مفید نیست. علتش این است که ایجاد اولیه heap و محاسبه محل پدر و فرزندان 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
مرتب‌سازی 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
مرتب‌سازی سریع	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	پایدارنیست و درجا
مرتب‌سازی ادغام	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	پایداراست و غیردرجا
مرتب‌سازی هرمی	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	پایدار نیست و درجا
مرتب‌سازی درختی	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	غیردرجا

۸-۸ تمرین‌های فصل

۱. عمل اصلی در الگوریتم مرتب‌سازی انتخابی، پیمایش لیست $x_1,x_2,\ldots,x_n$ 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 $x_2,\ldots x_{n-1}$ انجام می‌شود و غیره

الف) تابعی برای پیاده‌سازی این روش مرتب‌سازی بنویسید.

ب) زمان تابع را محاسبه کنید

ج) با استفاده از یک آرایه فرضی این روش مرتب‌سازی را نشان دهید.

۲. یک تابع بازگشتی برای مرتب‌سازی انتخابی بنویسید.

۳. یک تابع بازگشتی برای مرتب‌سازی درجی بنویسید.

٤. یک الگوریتم حبابی بنویسید که برای لیست پیوندی مناسب باشد.

۵. برای اعداد زیر، مرتب‌سازی 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، مرتب‌سازی حبابی را دنبال کنید:

d. 13, 57, 85, 70, 22, 64, 48

e. 13, 57, 12, 39, 40, 54, 55, 2, 68

f. 70, 57, 99, 34, 56, 89

۷. برای اعداد زیر، مرتب‌سازی انتخابی را دنبال کنید:

g. 13, 57, 85, 70, 22, 64, 48

h. 13, 57, 12, 39, 40, 54, 55, 2, 68

i. 70, 57, 99, 34, 56, 89

۸. برای اعداد زیر، مرتب‌سازی سریع را دنبال کنید:

j. 13, 57, 85, 70, 22, 64, 48

k. 13, 57, 12, 39, 40, 54, 55, 2, 68

l. 70, 57, 99, 34, 56, 89

۹. برای اعداد زیر، مرتب‌سازی هرمی را دنبال کنید:

m. 13, 57, 85, 70, 22, 64, 48

n. 13, 57, 12, 39, 40, 54, 55, 2, 68

o. 70, 57, 99, 34, 56, 89

۱۰. برای اعداد زیر، مرتب‌سازی درختی را دنبال کنید:

p. 13, 57, 85, 70, 22, 64, 48

q. 13, 57, 12, 39, 40, 54, 55, 2, 68

r. 70, 57, 99, 34, 56, 89

۱۱. برای اعداد زیر، مرتب‌سازی ادغام را دنبال کنید:

s. 13, 57, 85, 70, 22, 64, 48

t. 13, 57, 12, 39, 40, 54, 55, 2, 68

u. 70, 57, 99, 34, 56, 89

۱۲. برای اعداد زیر، مرتب‌سازی مبنایی را دنبال کنید: (یکبار به‌صورت صعودی و بار دیگر به‌صورت نزولی)

v. 13, 57, 85, 70, 22, 64, 48

w. 13, 57, 12, 39, 40, 54, 55, 2, 68

x. 70, 57, 99, 34, 56, 89

۱۳. تابعی بنویسید که ادغام سه طرفه را پیاده‌سازی کند. یعنی سه فایل مرتب را در یک فایل دیگر به‌طور مرتب ادغام کند.

۱٤. اثبات کنید که بهترین حالت ممکن برای هر الگوریتمی که n عنصر را مرتب می‌کند O(nlogn) می‌باشد.

۱۵. نشان دهید که هر فرایندی که یک فایل را مرتب می‌کند می‌تواند 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 مرتب‌سازی حبابی، قبل از این که فایل مرتب شود (غیر از آخرین گذر، که کشف می‌کند فایل مرتب شده است) برابر با

بزرگ‌ترین فاصله‌ای است کـه یـک عنـصر از یـک انـدیس بـزرگ‌تـر بـه انـدیس کوچک‌تر منتقل شود.

۱۸. مرتب‌سازی با شمارش به این صورت انجام می گیرد: آرایه‌ای به نام count تعریف کنید و count[i] را برابر با تعداد عناصری که کوچک‌تر از x[i] هستند قـرار دهیـد. سپس x[i] را در موقعیت count[i] یک آرایه خروجـی قـرار دهیـد. (امـا، مواظـب تساوی عناصر باشد) یک تابع بنویسید که یک آرایه x به اندازه n را بـه ایـن روش مرتب نماید.

۱۹. مرتب‌سازی جابجایی «فرد ـ زوج» به این صورت انجام می‌گیرد: در سراسـر فایـل چند بار گذر کنید. در گذر اول x[i] را با x[i+1] برای کلیـه مقـادیر فـرد i مقایـسه 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 ادامـه دهید.

الف) شرط پایان ری‌شچیمرتب؟ساز

ب) این روش مرتب‌سازی را توضیح دهید؟

ج) کارایی این روش در حالت متوسط چگونه است؟

۲۰. روش پیدا کردن عنصر pivot در مرتب‌سازی سریع به این صورت تغییر دهیـد کـه مقدار میانی عنصر اول، عنصر میانی و عنصر آخر را به کار ببـرد. در چـه مـواردی استفاده از این روش مرتب‌سـازی نـسبت بـه مرتـب‌سـازی مطـرح شـده در مـتن کارآمادتر است؟ در چه مواردی کارایی کمتری دارد؟

۹-۸ پروژه‌های برنامه‌نویسی

۱. فرض کنید یک فایل حجیم کـه شـامل مشخـصات شخـصی و درسـی دانـشجویان می‌باشد، موجود باشد. می‌خواهیم این فایل را بر اسـاس شـماره دانـشجوئی مرتـب کنیم.

توجه داشته باشید که فایل حجیم بوده و نمی‌تواند کل آن در حافظه قرار گیرد.

برنامه‌ای بنویسید که با ترکیب روش‌های مختلف مرتب‌سازی فایل مذکور مرتب شود.

سؤالات چهارگزینه‌ای

۱) مجموع مراحل خطوط در برنامه زیر چند است؟

```
float  sum (int num [ ] , int n )
{
        int  i,  temp = ∘ ;
        for (i=∘ ; i < n ; i  + + )
           temp + = num [i];
        return temp ;
}
```

۱) ۲n + ۳

۲) ۲n + ۱

۳) تعداد مراحل بستگی به n دارد و نامشخص است.

٤) n + ۱

۲) تعداد مجموع مراحل خطوط در برنامه زیر چند است؟ (MS ثابتی است که حداکثر انـدازه ماتریسها را مشخص می‌سازد.)

```
void   add (int a[ ] [MS], int b [ ] [MS] , int c [ ] [MS], int r, int c)
{
        int i, j ;
        for ( i =∘ ; i < r ; i + + )
             for ( j = ∘ ; j < c ; j + +)
             c [i][j] = a[i][j] + b[i][j];
}
```

۱) r ∗ c　　۲) ۲r ∗ c + r + c　　۳) ۲r ∗ c + ۲r + ۱　　٤) MS ∗ r ∗ c

۳) کدامیک از مجموع توابع زیر برحسب افزایش مرتبـه (order) از چـپ بـه راسـت مرتـب هستند؟

۱) $(1.005)^n, n^{1000}, n!$
۲) $n^{1000}, (1.005)^n, n!$
۳) $(1.005)^n, n!, n^{1000}$
۴) $n^{1000}, n!, (1.005)^n$

۴) در برنامه زیر تعداد دفعات تکرار دستورالعمل انتساب برابر است با ....

```
for(k = 0; k <= n - 1; k++)
  for(i = 1; i <= n - k; i++)
      a[i][i + k] = k;
```

۱) n^2 ۲) $\frac{n(n+1)}{2}$ ۳) $\frac{n^2}{2}$ ۴) $\frac{n(n-1)}{2}$

۵) اگر $F(n) = 5n + 100$ و $C=6$ و $g(n) = n$ باشد به ازای کدام مقدار n، رابطه $F(n) = O(g(n))$ برقرار است؟

۱) ۱۲۰ ۲) ۶۳ ۳) ۵۸ ۴) ۱۰۰

۶) مرتبه اجرای الگوریتم زیر چیست؟

```
For( j=1 ; j<= m ; j++ )
   For(  k=1 ; k<= j ; k++ )
          x++;
```

۱) $O\left(\frac{m+1}{2}\right)$ ۲) $O(m^2)$ ۳) $O(\log_3 m)$ ۴) $O(m^3)$

۷) در الگوریتم زیر در صورتی که n=m باشد مرتبه اجرایی کدام است؟

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

۱) $O\left(\frac{m+1}{2}\right)$
۲) $O\left(\frac{m(m+1)}{2}\right)$
۳) $O(n^2)$
۴) $O(n^3)$

۸) کدامیک از روابط زیر نشان‌دهنده رابطه صحیح زمان محاسبه الگوریتمهای مختلف است؟

۱) $O(\log n) < O(n) < O(n\log n) < O(2^n) < O(n^2)$

۲) $O(n) < O(\log n) < O(n\log n) < O(2^n) < O(n^2)$

۳) $O(n) < O(\log n) < O(n\log n) < O(n^2) < O(2^n)$

٤) $O(\log n) < O(n) < O(n\log n) < O(n^2) < O(2^n)$

۹). زمان جمع دو چندجمله‌ای یک متغیر که براساس توان X به صورت نزولی مرتب شده‌انـد و یکی m جمله و دیگری n جمله دارد از چه مرتبه‌ای است؟

۱) $O(m\times n)$ ۲) $O(m+n)$ ۳) $O(2^{m\times n})$ ٤) $O(2^{m+n})$

۱۰) مرتبه اجرای برنامه زیر کدام است؟

```
i= n ;
while ( i> 1 ) {
        i =  i/2 ;  j= n ;
        while ( j> 1 )
        j=j/3 ;
}
```

۱) $O(\log_2 n)$ ۲) $O(\log_3 n)$ ۳) $O(\log_6 n)$ ٤) $O(\log_2 n \times \log_3 n)$

۱۱) الگوریتمهای بازگشتی چه معایبی دارند؟

۱) اتلاف حافظه، سرعت اجرای کمتر

۲) اتلاف حافظه، طولانی‌بودن سورس

۳) سرعت اجرای کمتر، طولانی بودن سورس

٤) طولانی بودن سورس، اتلاف حافظه، سرعت اجرای کمتر

۱۲) تابع زیر روی عدد طبیعی X چه عملی انجام می‌دهد؟

```
int    g( int  X )
{
      if (  X > 1 )
```

```
        return  X∗ g( X – 1 );
      else
        return 1 ;
}
```

۱) ۱   ۲) X^X ۳) $X!$ ۴) $\sum_{i=۱}^{X} i$

۱۳) با توجه به تابع روبرو (۱۰۰)Func چه خواهد بود؟

```
int     Func (int  n)
{
      if (n = = ∘)
         return ∘;
      return ( n + Func ( n - 1 ) ) ;
}
```

۱) ۱۹۹   ۲) ۲۰۰   ۳) ۵۰۵۰   ۴) ۱۰۰۰۰

۱۴) تابع زیر چه کاری را انجام می‌دهد و L(۲۵) را محاسبه نمائید.

$$L(n) = \begin{cases} ۰ & \text{if} \quad n = ۱ \\ (\lfloor n/۲ \rfloor) + ۱ & \text{if} \quad n > ۱ \end{cases}$$

۱) نصف عدد داده شده + ۱و۱۳

۲) بزرگترین عدد صحیح به طوری که $۲^L \leq n$ ،۴

۳) $L = \lfloor \log_۲ n \rfloor$ ،۴

۴) ۳ صحیح است.

۱۵) خروجی تابع ر که به صورت F(۲،۱) صدا زده شده است کدام است؟

```
int   F( int  m , int  n )
{
    if ( m = = ∘)
        return + + n ;
    if (n = = ∘)
        return F ( m – 1 , 1 ) ;
    return F ( m – 1 , F( m ,  n – 1 ) ) ;
```

```
}
```

۱) ۷   ۲) ۵   ۳) ۸   ٤) ۳

۱٦) اگر تابع A به صورت بازگشتی زیر تعریف شده باشد:

$$A(m,n)\begin{cases} n+1 & m=0 \quad \text{اگر} \\ A(m-1,1) & m\neq 0\ ,\ n=0 \quad \text{اگر} \\ A(m-1,A(m,n-1)) & m\neq 0\ ,\ n\neq 0 \quad \text{اگر} \end{cases}$$

آنگاه مقدار خروجی (۲تابع) کدام است:

۱) ۹   ۲) ۱۵   ۳) ٤   ٤) ۷

۱۷) خروجی این تابع به ازای n =۲۰ چیست؟

```
float   F( int  n )
{
    if ( n = = 1 )
        return  Sqrt (12) ;
    else
        return Sqrt ( 12 + F( n – 1 )  ) ;
}
```

۱) ٤.۰   ۲) ۳.۰   ۳) ۱۲.۰   ٤) ٦/۰

۱۸) تابع بازگشتی زیر را در نظر بگیرید:

```
int      recursive ( int  n )
{
    if (    n = =1)
        return 1 ;
    else
        return( recursive (n-1) + recursive (n-1) );
}
```

۱) ۱٦   ۲) ۸   ۳) ۲۳   ٤) ۱٤

۱۹) در برنامه زیر مقدار F(۳،٦) برابر است با:

```
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 ) ) ;
}
```

۱) ۲۰ ۲) ۱۰ ۳) ۱۸ ۴) ٤

۲۰) تابع بازگشتی را در نظر بگیرید:

```
int  test  (int  n)
{
  if (n <= 2)
     return 1;
  else
     return test (n-2) * test (n-2);
}
```

زمان اجرای تابع فوق برابر است با:

۱) $O(n^2)$ ۲) $O(n\log n)$ ۳) $O\left(2^{n/2}\right)$ ۴) $O(2^n)$

۲۱) تابع ACK به صورت زیر تعریف می‌شود. مقدار ACK(۱،۱) برابر است با:

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

۱) ۵ ۲) ٤ ۳) ۳ ۴) ٦

۲۲) با توجه به دو تابع زیر $F_1(4)$ و $F_2(4)$ چیست؟

```
void  F1 ( int  X )
{
    if ( X )  F2 ( X – 1 ) ;
    Printf ( X ) ;
}
```

```
void  F2 ( int  Y )
{
        if (Y){
            Printf ( Y + 1 ) ;
            F1 ( Y – 1 ) ;
        }
}
```

۱) به ترتیب از چپ به راست بـرای $F_1(4)$ خروجـی ۴۴۲۲۰ و بـرای $F_2(4)$ خروجـی ۵۳۱۱۳ است.

۲) به ترتیب از چپ به راست بـرای $F_1(4)$ خروجـی ۲۰۲۴۴ و بـرای $F_2(4)$ خروجـی ۱۳۳۵ است.

۳) به ترتیب از چپ به راست بـرای $F_1(4)$ خروجـی ۴۲۲۰۴ و بـرای $F_2(4)$ خروجـی ۵۳۳۱ است

۴) به ترتیب از چپ به راست بـرای $F_1(4)$ خروجـی ۴۲۰۲۴ و بـرای $F_2(4)$ خروجـی ۵۳۱۳ است

۲۳) در روال (routine) بازگشتی زیر مقدار Rec(۵،۳) کدام است؟

```
int   Rec ( int P , int q )
{
        int R ;
        if (q <= 0) return 1;
        R = Rec (P, q/2);
        R = R * R;
        if ( q % 2 = = 0 )
            return R;
       else
            return R * P;
}
```

۱) ۱۵　　۲) ۲۵　　۳) ۷۵　　۴) ۱۲۵

۲۴) تعداد مراحل کل خطوط در برنامه زیر چقدر است؟

```
float   rsum (float  list  [ ] , int  n)
{
       if ( n )
           return ( rsum (list, n-1) + list [n-1] ) ;
       return list [ ० ] ;
}
```

۱) $۲n+۴$ ۲) $۲n+۲$ ۳) $۲n^۲$ ۴) $۲(n-۱)^۲$

۲۵) در ضرب سه آرایه A(۳،۴) و B(۴،6) و C(۶،۲) به ترتیب $A*B*C$ چند عمل ضرب انجام می‌شود؟

۱) ۲۵ ۲) ۱۰۸ ۳) ۲۵۹۲ ۴) ۳۴۵۶

۲۶) برای یافتن یک عنصر درون آرایه N عنصری به چه تعداد مقایسه نیاز است (روش جستجوی خطی)؟

۱) N ۲) $N+۱$ ۳) $\frac{N+۱}{۲}$ ۴) $\frac{N-۲}{۲}$

۲۷) افزودن یک عنصر به یک آرایه به طور متوسط چند جابه‌جائی نیاز دارد؟

۱) $N-۱$ ۲) $\frac{N-۱}{۲}$ ۳) $N+۱$ ۴) $\frac{N+۲}{۲}$

۲۸) برای حذف عنصر kام از یک آرایه N عنصری چند جابه‌جائی لازم است؟

۱) N-K-۱ ۲) K ۳) N-K ۴) N-K+۱

۲۹) در یک آرایه، n عدد به ترتیب نزولی قرار دارد. اگر از روش جستجوی دودوئی برای یافتن عدد استفاده کنیم، حداکثر تعداد مقایسه چقدر خواهد بود؟

۱) $n\log_۲ n$ ۲) $\lfloor \log_۲ n \rfloor+۱$ ۳) $\log_۲(n-۱)$ ۴) $n-\log_۲ n$

۳۰) مرتبه اجرائی الگوریتم جمع ماتریسها و ضرب ماتریسها به ترتیب از راست به چپ می‌شود؟

۱) $O(n^۳),O(n^۲)$ ۳) $O(n^۲),O(n\log n)$

۲) $O(n^۳),O(n\log n)$ ۴) $O(n^۲\log n),O(n^۲)$

۳۱) در یک جستجوی خطی حداکثر تعداد جستجو برابر کدام است؟

۱) n　　۲) $n-۱$　　۳) $\frac{n}{۲}$　　٤) $n^{۲}$

۳۲) کار تابع F بر روی رشته S با n کاراکتر چیست؟ تـابع Sub(S,i,n) تعـداد n کـاراکتر از موقعیت i در رشته S را برمی‌گرداند.

$$F(S,n)=\begin{cases} S & \text{اگر } n=۱ \\ F\big(\mathrm{Sub}(S,۱,n-۱),n-۱\big)+\mathrm{Sub}(S,n,۱) & \text{اگر } n>۱ \end{cases}$$

۱) رشته S را برمی‌گرداند.
۲) معکوس رشته S را برمی‌گرداند.
۳) یک کاراکتر از انتها به رشته S اضافه می‌کند.
٤) یک کاراکتر به ابتدای رشته S اضافه می‌کند.

۳۳) فرض کنید یک آرایه ۲۰۰ عنصری مرتب شده باشد. زمان اجرای بدترین، برای پیداکردن عنصر معلوم X در آرایه A با استفاده از جستجوی دودوئی چیست؟

۱) ۲۰۰　　۲) ۸　　۳) ۷　　٤) ۱۲

۳٤) رشته ABCD داده شده است.ین رشته چند زیررشته دارد؟

۱) ٤　　۲) ۵　　۳) ۱۱　　٤) ۱۰

۳۵) میخواهیم حاصلضرب ABCD (A یک ماتریس ۵×۱۳، B یک ماتریس ۸۹×۵، C یـک ماتریس ۳×۸۹، D یک ماتریس ۳٤×۳ می‌باشد) را پیدا کنیم بطوریکه کمتـرین تعـداد عمـل ضرب انجام گیرد. ترتیب ضرب ماتریسها عبارت است از:

۱) $\big(A(BC)\big)D$　　۲) $A\big((BC)D\big)$　　۳) $(AB)(CD)$　　٤) $\big((AB)C\big)D$

۳٦) در صورتیکه آرایه مورد جستجو در جستجوی دودوئی به صـورت ۷,٦,۵,٤,۳,۲,۱,۰,۱− باشد، متوسط تعداد مقایسه‌ها برای جستجوی موفق چیست؟

۱) $\frac{۲۷}{۹}$　　۲) $\frac{۲۵}{۹}$　　۳) $\frac{۳۱}{۹}$　　٤) هیچکدام

۳۷) فرض کنید آرایه مورد جستجو توسط جستجوی دودوئی به صورت (۱۰۱,۸۲,۵٤,۳۰,۲۰,۹,۷,۰,٦-) باشد، متوسط تعداد مقایسه‌های مورد نیاز برای حالت جستجوی موفق چیست؟

۱) $\frac{۲۸}{۹}$ ۲) $\frac{۱۸}{۹}$ ۳) $\frac{۲۵}{۹}$ ٤) $\frac{۲٦}{۹}$

۳۸) تابع بازگشتی چه می‌کند؟

```
int  X ( int a[ ] , int k , int m , int n )
{
     int  f ;
     if ( m <=  n ){
        f = ( m + n ) / 2 ;
        Switch ( Compare ( a[P] , k ) {
        Case -1:  return  X ( a , k , f + 1 , n ) ;
        Case ∘ : return  f ;
        Case 1 : return  X( a , k , m , f – 1 ) ;
        }
  }
return -1;
}
```

۱) جستجوی عنصری در آرایه
۲) مرتب کردن آرایه
۳) پیدا کردن اولین عنصر بزرگتر یا کوچکتر از عدد معین در آرایه
٤) پیدا کردن اولین عنصر کوچکتر از عددی معین در آرایه

۳۹) می‌خواهیم چهار ماتریس $A(۱\times۳۰)$ و $B(٤۰\times۱)$ و $C(۱۰\times٤۰)$ و $D(۱۰\times۲۵)$ را در هم ضرب کنیم $(A\times B\times C\times D)$ حداقل ممکن تعداد عمل ضرب عناصر این ماتریس‌ها چند تا است؟

۱) ۱۲۵۰ ۲) ۱٤۰۰ ۳) ۲۰۷۰۰ ٤) ۱۱۷۵۰

٤۰) اگر $Pat = 'abcabcacab' = P_۰P_۱...P_{n-۱}$ باشد و داشته باشیم:

$$F(j)=\begin{cases} P_۰P_۱...P_k = P_{j-k}\,P_{j-k+۱}...P_j & k<j \quad \text{بزرگترین} \\ -۱ & \text{در غیر اینصورت} \end{cases}$$

مقادیر تابع Pat برای F

برابر است با ....

۱) ۰،۰،۱،3،2،۱،۰،۰،۰،۱-

۲) ۰،۰،۱،3،2،۰،۰،۰،۰،۱

۳) ۰،۰،۰،۰،۰،۱،۱،۱،۱،۱،۱

۴) ۱،۰،۱-،3،2،۱،۰،۱-،۱-،۱-

۴۱) مینیمم و ماکزیمم اعداد ذخیره شده در یک آرایه یک بعدی با n خانه، با چند مقایسه بین اعداد ذخیره شده در این خانه‌ها بدست خواهد آمد؟

۱) $\frac{3n}{2}$ ۲) $\frac{3n}{2}-2$ ۳) $\frac{n}{2}$ ۴) $\frac{n+1}{2}$

۴۲) اگر $a=2$ و $b=4$ و $c=8$ و $d=10$ باشد، ارزش عبارت پسوندی $ab*c+d\,a-/c*$ چیست؟

۱) ۲- ۲) ۱- ۳) ۱ ۴) ۲

۴۲. یک پشته خالی با اعداد ۱ تا ٦ 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.

کدامیک از گزینه‌های زیر را نمی‌توان با هیچ ترتیبی از اعمال فوق به‌دست آورد؟ (اعداد را از چپ به راست بخوانید)

۱) ٤ ٦ ٥ ٣ ٢ ۱

۲) ۱ ٥ ٦ ٤ ٢ ٣

۳) ٥ ٦ ۱ ٢ ٣ ٤

۴) ٦ ٤ ٣ ٥ ۱ ٢

۴۳. یک *stack* با عملیات زیر مفروض است:

الف) عملیات stack:

الف) عملیات stack:	مقدار x را وارد stack می‌کند	PUSH (X):
	مقدار بالای stack را در x می‌گذارد	pop (x):
ب) عملیات صف:	مقدار x را در انتهای صف می‌گذارد	Enqueue (x):
	مقدار سرصف را در x می‌گذارد	Dequeue (x):

فرض کنید در ابتدا *stack* حاوی ۳ عدد صحیح متفاوت دلخواه است؛ مثلاً ۳ و ۱ و ۲) اگر در پایان انجام عملیات زیر، *stack* حاوی همان ۳ عدد به ترتیب صعودی از پایین به بالا؛ مثلاً ۳ و ۲ و ۱) باشد، چند ترتیب اولیه در *stack* برای ۳ عدد مزبور ممکن است؟

pop (x): Enqueue (x); pop (x); Enqueue (x); Dequeue (x); push (x); pop (x); Enqueue (x); pop (x): Enqueue (x); Dequeue (x); push (x); Dequeue (x); push (x); Dequeue (x); push (x);

۱) ۳ ۲) ۲ ۳) ٤ ٤) ۱

٤٤. بر روی پشته s اعمال زیر تعریف شده‌اند:

Push (S.x): درج x در بالای پشته با هزینه *O(1)*. *Pop (S.x)*: حذف عنصر بالای پشته با هزینه *o(1)*. *Multipop (s.k)*: حذف k عنصر بالای پشته (با فرض آنکه k حداکثر برابر تعداد عناصر موجود s است) با هزینه *o(k)*. اگر N عمل از اعمال فوق به ترتیب دلخواه بر روی پشته S که در ابتدا تهی است انجام شود مجموع هزینه این N عمل در بدترین حالت چقدر است؟

۱) $O(N\log N)$ ۲) $O(N^۲)$ ۳) $O(NK)$ ٤) $O(N)$

٤٥. عناصر صف‌های $Q_۱$ و $Q_۲$ از چپ به راست به‌صورت زیر است (عنصر سمت چپ ابتدای صف است).

$Q_۱ =$ ۱۰,۲۵,۱۷,٤۱,۱۹,۲٦,۷۵

$Q_۲ =$ ۱,٥,۷,٤,۹,٦

اگر x و y عناصر صف باشند، پس از اجرای‌رقطعه برنامه ز :

```
Makenull (Q٣)
{
    i = ٠;
    while (not empty (Q١) and not empty (Q٢))
    {
        i = i + l;
        x = Delete Q (Q١);
        y = Delete Q (Q٢);
        it (y = i) then Add Q (Q٣Q)
    }
}
```

محتوی صف $Q_۳$ برابر است با:

۱) $Q_۳=۱,٤,٦$ ۲) $Q_۳=۱۰,۲٥,۱۷$ ۳) $Q_۳=۱۰,٤۱,۲٦$ ٤) $Q_۳=۱,٥,۷$

٤٦. اگر رشته اعداد ۱ و ۲ و ۳ و ٤ و ٥ را به ترتیب وارد کنیم *stack* کدامیک از خروجی‌های زیر از این *stack* امکان‌پذیر خواهد بود. (خروجیهای پشته را از سمت چپ به راست بخوانید)

۱) ٤-۱-٥-۳-۲ ۲) ۱-۲-۳-٤-٥

۳) ۲-٤-٥-۳-۱ ٤) ٤-۲-۳-۱-٥

٤۷. اگر رشته اعداد ۱ و ۳ و ٤ و ٥ و ۷ را به ترتیب از سمت راست وارد یک *stack* کنیم کدامیک از خروجیهای زیر از این *stack* امکان‌پذیر نیست. (خروجیها را از سمت راست بخوانید)

۱) ۷ ٥ ٤ ۳ ۱ ۲) ۱ ۳ ۷ ٥ ۷

۳) ۱ ۷ ۳ ٥ ٤ ٤) ۱ ٤ ۳ ۷ ٥

٤۸. عبارت پسوندی *(Postfix)* معادل عبارت ریاضی $a/b-c+d*e-a*c/d$ کدام است؟ (با توجه به اولویت عملگرها)

۱) $ab/c-de*+ac*d/-$ ۲) $ab/c-d*e+a-c/d/$

۳) $ab/c-de*ac*-d/$ ٤) $abc/-d+e*a-c*d$

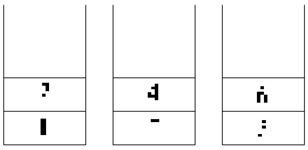
٤۹. عبارت پیشوندی *(prefix)* معادل عبارت ریاضی $a/b-c+d*e-a*c$ کدام

است؟ (با توجه به اولویت عملگرها)

۱) $+-/abcd*e-ac*$ ۲) $-+-/abc*de*ac$

۳) $a/b-c+d*e-a*c/d$ ۴) $+-/abcde*-ac*$

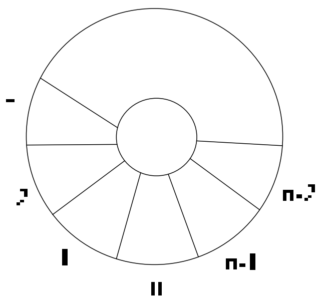
۵۰. سه پشته S_1 و S_2 و S_3 هر یک حاوی دو عدد به شکل زیر می‌باشند:



دو عملگر $pop(i), poppush(i,j)$ به‌صورت زیر تعریف شده‌اند.
$poppush(i,j)$ یک قلم از پشته S_i حذف و به پشته S_j اضافه می‌کند.
$pop(i)$ یک قلم از پشته S_i حذف و سپس آن را چاپ می‌کند. برای چاپ اعداد ۱ تا ٦ به‌صورت ٦،٤،٥،٣،۱ عملگر $poppush$ بایستی حداقل چندبار مورد استفاده قرار گیرد؟ اولین عددی که چاپ می‌شود عدد ۱، دومین عدد، عدد ۳ و... می‌باشد.

۱) ۳ بار ۲) ۵ بار ۳) ٦ بار ٤) ٤ بار

۵۱. کدامیک از فرمولهای زیر، N، تعداد اقلام در یک صف دایره‌ای را محاسبه می‌کند؟ متغیر F به خانه‌ای که بلافاصله قبل از جلوی صف قرار دارد، (در جهت عکس عقربه‌های ساعت) اشاره می‌کند و متغیر R به عقب صف اشاره می‌نماید.



۱) $N = R - F$ ۲) $N = n(R - F)$

۳) $N = \begin{cases} n-(F-R) & \text{if } F > R \\ R-F & \text{if } R > F \end{cases}$ ٤) $N = \begin{cases} n-(R-F) & \text{if } F < R \\ R-F & \text{if } R > F \end{cases}$

۵۲. عبارت *infix* زیر را به *polish* تبدیل کنید.

$A*(B-D)/E-F*(G+H/K)$

۱) $ABD+*E/FGHK/+*-$ ۲) $A+B*DEF/GHK/-*-$

۳) $-*+/KHG/FED*B+A$ ۴) $AB*-DEF/G+HK/*$

۵۳. عبارت *post fix* زیر مساوی است با:

۱۲ ۷ ۳ − / ۲ ۱ ۵ + * +

۱) ۸ ۲) ۱۵ ۳) ۰ ٤) هیچکدام

٥٤. کم‌هزینه‌ترین (از نظر تخصصی حافظه) برای اینکه عناصر یک پشته *(Stack) S1* را به پشته دیگری *S2*، بدون اینکه ترتیب عناصر تغییر یابند، انتقال دهیم کدام است؟

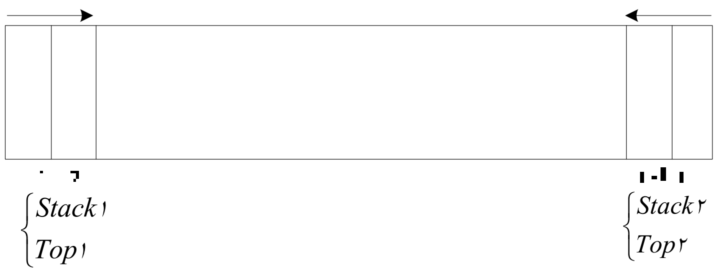
۱) از طریق یک متغیر ۲) از طریق یک پشته *(Stack)* اضافی

۳) از طریق دو پشته *(Stack)* اضافی ٤) از طریق چند متغیر

۵۵. مینیمم تعداد متغیرهای میانی در محاسبه عبارت جبری *ab+cd*/a+* که به‌صورت *postfix* است، برابر است با ...

۱) ۱ ۲) ۲ ۳) ۳ ٤) ٤

٥٦. دو پشته در یک آرایه به اندازه *(1..n)* پیاده‌سازی شده‌اند. اگر *Top* یکی به خانه خالی و *Top* دیگری به خانه پر اشاره کند، شرط پر بودن بردار کدام گزینه می‌باشد؟



۱) $Top1=Top2-1$ ۲) $Top1=Top2$

۳) $Top1>Top2$ ٤) هیچکدام

۵۷. عبارت *postfix* معادل عبارت $(A+B)*D+E/(F+A*D)$ برابر است با ...

۱) $AB+D*E+F/A+D*$ ۲) $AB+D*EFAD*+/+$

۳) $ABDEFAD+*+/+*$ ٤) $AB+DE+FAD*+/$

۵۸. عبارت پیشوندی *(prefix)* زیر داده شده است:

$$++a/b-cd/-ab-+c*d5/a-bc$$

معادل پسوندی *(postfix)* آن کدام است؟

۱) $abcd-/+ab-cd5*+abc-/-/+$ ۲) $ab+cd-ab-cd+/5*+ab/c--+$

۳) $ab+cd-ab-+/cd5*+/abc-/-$ ۴) $abcd/-+abc-d5*abc-+/-/+$

۵۹. یک لیست دو طرفه خطی موجود است.

```
Struct   Node{
        char Info;
        Node *next,*perv;
}
```

فرض کنید لیست دارای سرلیست است. زیرا برای کپی کردن این لیست پیشنهاد شده است:

```
Node    * Copy(Node   *List)
{
        Node   *LC;
        LC=NULL;
        if (List!=NULL){
            LC = get node( );
            LC →Info=List→Info;
            LC → next=Copy(List →next);
            if (List →next!=NULL)
                List→next→prev=List;
        }
        return LC;
}
```

۱) این الگوریتم کاملاً درست است.

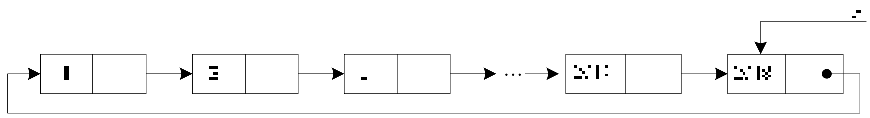
۲) این الگوریتم هیچ وقت کاملاً درست عمل نمی‌کند.

۳) این الگوریتم ممکن است ایجاد خطای زمان اجرا نماید.

۴) این الگوریتم فقط برای بعضی از لیستها کاملاً درست است.

۶۰. با وجود بودن‌لیست پیوندی حلقوی به شکل زیر، خروجی حلقه قطعه کد داده شده چیست؟ فرض کنید هر عنصر لیست پیوندی از دو مولفه *CellData* از نوع *Integer* و *Next* از نوع اشاره‌گر تشکیل شده باشد.

```
While (p → next != p)
  {
       p → next = p → next → next;
       p = p → next;
  }
Cout << p → CellData;
```



١) ٢٠٤٧ ٢) ١ ٣) ٢٠٤٨ ٤) ١٠٢٤

٦١. روال زیر چه عملی انجام می‌دهد؟

```
procedure f (x, y : prt; var z : prt);
var p : prt;
begin
     p := x;   z := x;
     while p↑.Link <> nil do
          p := p↑.Link;
     p↑.Link := y;
end;
```

١) دو لیست پیوندی را به هم وصل *(ConCat)* می‌کند.

٢) دو لیست حلقوی *(Circular)* را به هم وصل می‌کند.

٣) دو لیست پیوندی که حداقل یکی از آنها غیر تهی می‌باشد را به هم وصل می‌کند.

٤) دو لیست حلقوی غیرمنفی را به هم وصل می‌کند.

٦٢. تابع چه عملی انجام می‌دهد؟ (با فرض اینکه نوع *list* اشاره‌گر است)

```
list x (list L)
{
    list m, t;
    m = NULL;
    while (L) {
        t = m;  m = L;
        L = L → link
        m → link = t;
    }
    return m;
}
```

۱) محل دو عنصر در لیست L را جابه‌جا می‌کند.

۲) لیست پیوندی L را معکوس می‌کند.

۳) عنصری را از لیست L جابه‌جا می‌کند.

۴) لیست L را مرور می‌کند.

٦٣. شرط پایان در حلقه *Repeat-Until* قطعه برنامه پاسکال زیر که ر پیمایش یک لیست یکطرفه دوار بدون گره آغازین *(Header Node)* نوشته شده است چیست؟

```
P := Start;
If p <> nil then
    Repeat
        Writeln (P↑data);
        P := P↑Link;
Until شرط ;
```

۱) P <> Start	۲) p = Start

۳) P↑Link = Start.	۴) P↑Link <> Start.

٦٤. زیر برنامه g بر روی یک لیست پیوندی یکطرفه کدام عمل را انجام می‌دهد؟

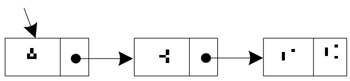
```
Procedure   g(Var Start : Nodeptr);
Var
    p, q, r : Nodeptr
Begin
    p := Start;
    q := NIL;
    while p <> NIL do
        Begin
            r := q;
            q := p;
            p := P ↑ .Link;
            q ↑ .Link := r;
        End;
    Start := q;
End;
```

۱) پیمایش لیست ۲) حذف کردن لیست

۳) معکوس کردن لیست ٤) مرتب کردن لیست

٦٥. خروجی رویه برگشتی *WHAT* برای لیست پیوندی یکطرفه زیر چیست؟

```
procedure WHAT (L)
begin
    if L ≠ ۰ then begin
        WHAT (Link(L));
        Print (Data (L));
        WHAT (Link(L));
        Print (Data (L));
    End
End;
```



۱) *CCBCCBACCBCCBA* ۲) *CBCCBCACBCCBCA*

۳) *ACBBACBBACBBA* ٤) *CBCBCCACBCBCCA*

٦٦. درخت *T* با *n* رأس مفروض است. اگـر ایـن درخـت دارای پـنج رأس *(node)* از درجه *(degree)* چهار و شش رأس از درجه سـه و پـنج رأس از درجـه دو باشـد، تعداد برگ‌های آن چقدر است؟

۱) $n-1$ ۲) $2n-2$ ۳) 18 ٤) هیچکدام

۶۷. رویه زیر را برای یک درخت دودوئی در نظر بگیرید.

```
function Count (tree : pointer) : integer;
begin
   if tree = nil then Count := 0
  else if tree^.Left = nil and tree^.rigt = nil then
    Count := 1
  else
     Count := Count (tree^.Left) + Count (tree^.right)
end;
```

این رویه:

۱) تعداد گره‌های یک درخت دودوئی را محاسبه می‌کند.

۲) تعداد برگ‌های یک درخت دودوئی را محاسبه می‌کند.

۳) تعداد گره‌هایی که دارای دو فرزند می‌باشند را محاسبه می‌کند.

۴) تعداد گره‌هایی را که دارای یک فرزند می‌باشند را محاسبه کنید.

۶۸. در یک درخت دودوئی کامل با n گره، حداکثر فاصله بین دو گره کدامیک از پاسخهای زیر است: طول هر بال را واحد فرض کنید.

۱) حدود $log_2 n$ ۲) $2 log_2 n$

۳) حدود $3 log_2 n$ ۴) $4 log_2 n$

۶۹. در یک درخت دودوئی کامل با n گره، برای هر گره با اندیس i داریم:

۱) اگر $i <> 1$ باشد، i ریشه است و پدری نخواهد داشت.

۲) $2i > n$ باشد آنگاه i فرزند راست ندارد.

۳) اگر $i <> 1$ باشد آنگاه پدر i در $[i/2]$ است.

۴) اگر $2i + 1 > n$ باشد آنگاه i فرزند چپ ندارد.

۷۰. کدام گزینه نادرست است؟

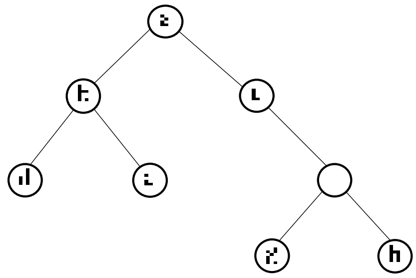
۱) بیشترین تعداد گره‌ها در یک درخت دودوئی به عمق k، $2^k - 1$ است $(k >= 1)$.

۲) بیشترین تعداد گره‌ها روی سطح i ام یک درخت دودوئی 2^{i-1} است $(i >= 1)$.

۳) در هیچ درخت عادی گره صفر وجود ندارد.

۴) در هیچ درخت دودوئی گره صفر وجود ندارد.

۷۱. پیمایش درخت زیر به روش $postorder$ کدام است؟

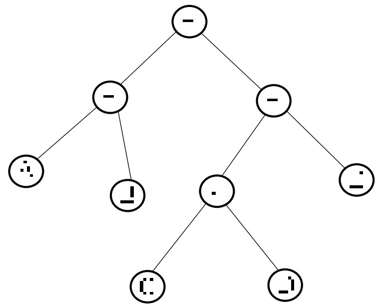


۱) *dbeacgfh* ۲) *acfhgbed* ۳) *abdecfgh* ٤) *debghfca*

۷۲. پیمایش پس ترتیبی *(postorder)* یک درخت به‌صورت *DEBFCA* می‌باشد. کدامیک از گزینه‌های زیر درخت پیش ترتیبی *(preorder)* آن را نشان می‌دهد؟

۱) *DBEACF* ۲) *DABCEF* ۳) *ABDECF* ٤) *ACEDBF*

۷۳. پیمایش *postorder* درخت مقابل کدامیک از گزینه‌های زیر است؟



۱) $++AB+{*}CDE$ ۲) $AB+CD*E++$

۳) $A*B+C*D+E$ ٤) $BA+DC*E++$

۷٤. تابع زیر برای درخت دودوئی T چه عملی را انجام می‌دهد؟

```
Function n(T : tree) : integer;
begin
      n := 0;
      if T ≠ nil then
      if Rchild(1) = nil and Lchild(t) = nil then
            n := 1
      else n := n(R Child(t)) + n(Lchild(t)) + 1;
end;
```

۱) تعداد برگ‌های T را می‌شمارد.

۲) تعداد گره‌های T را می‌شمارد.

۳) تعداد گره‌های دو فرزندی T را می‌شمارد.

٤) تعداد گره‌های غیربرگ را می‌شمارد.

۷۵. یک درخت دودویی کامل با ارتفاع ۷، می‌بایست حداقل چند تعداد گره داشته باشد؟

۱) ٦٤ ۲) ۱۲۷ ۳) ۱۲۸ ٤) ٦۳

۷٦. یک درخت دودویی کامل به ارتفاع h می‌باید … قل چند nod داشته باشد؟

۱) $\left(\sum_{i=0}^{h-2} 2^i\right)$ ۲) $\sum_{i=0}^{h} 2^i$ ۳) $\left(\sum_{i=0}^{h-1} 2^i\right)+1$ ٤) $\sum_{i=0}^{h-1} 2^i$

۷۷. کدام گزینه نادرست است؟

۱) در هر درخت تعداد یالها یکی کمتر از تعداد رأس‌هاست.

۲) یک درخت، یک گراف همبند بدون دور است.

۳) در هر درخت هر دو رأس با یک مسیر منحصر به فرد به هم متصل هستند.

٤) در هر درخت تعداد یالها یکی بیشتر از تعداد رأس‌هاست.

۷۸. عمق یک درخت دودویی کامل با n گره برابر است با:

۱) $h=[log_2 n]-1$ ۲) $h=[log_2 n]$

۳) $h=[log_2 n]+1$ ٤) $h=[log_2 n]-2$

۷۹. بیشترین تعداد گره‌ها در یک درخت دودویی *(Binary Tree)* با عمق h کدام است؟

۱) 2^{h+1} ۲) 2^h+1 ۳) 2^h-1 ٤) 2^{h-1}

۸۰. کدام گزینه در مورد درخت دودویی نادرست است؟

۱) در درخت دودویی تهی گره صفر وجود دارد.

۲) درخت دودویی در حقیقت درختی است که هر گره آن حداکثر ۲ شاخه دارد.

۳) بیشترین گره روی سطح kام یک درخت دودویی 2^{k-1} است.

٤) بیشترین گره‌ها در یک درخت دودویی به عمق k 2^k-10k است.

۸۱. کدام جمله در ارتباط با درخت دودویی‌کام لاً صحیح است؟ (i اندیس گره و n تعداد کل گره می‌باشند).

۱) اگر $2i \le n$ آنگاه فرزند سمت راست i در $2i$ است.

۲) اگر $2i > n$ آنگاه فرزند سمت راست i در $2i$ است.

۳) اگر $n \leq 2i$ آنگاه فرزند سمت چپ i در $2i$ است.

٤) اگر $2i < n$ آنگاه فرزند سمت چپ i در $2i$ است.

۸۲. برای هر درخت دودویی غیرتهی، اگر n_0 تعداد گره‌های پایانی و n_2 تعداد گره‌هایی باشد که درجه ۲ دارند آنگاه:

۱) $n_0 = n_2 + 1$ ۲) $n_2 = n_0 + 1$ ۳) $n_2 + n_0 = 1$ ٤) $n_2 + n_0 = 2$

۸۳. الگوریتم زیر:

```
procedure X (Left, Right, Root, Count)
    if Root = NULL then Count = 0
    else
        Call X (Left, Right, Left[Root], count1)
        Call X (Left, Right, Right[Root], count)
        if Count1 >= Count2 then
            Count := Count 1 + 1
        else
            Count := Count 2 + 1
        endif
    endif
    return
end x
```

۱) تعداد نودهای یک درخت را محاسبه می‌کند.

۲) تعداد نودهای یک درخت را که بزرگ‌تر از مقدار معینی است به‌صورت محاسبه بازگشتی می‌کند.

۳) عمق یک درخت باینری را محاسبه می‌کند.

٤) هیچکدام

۸٤. پردازهٔ زیر را برای پیمایش درخت دودویی نوشته‌ایم: کدامیک از عبارتهای زیر درست است؟

```
procedure trav (r : tree);
begin
        if r <> nil then begin
          trav (r^.left);
         writeln (r^.info);
         trav (r^ right);
    end;
end;
```

۱) اگر جای سطرهای ٤ و ٥ پـردازه را عـوض کنـیم، پـردازه بـرای پیمـایش پـس ترتیبی به‌کار می‌رود.

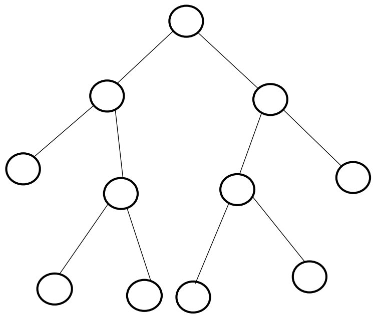
۲) این پردازه برای پیمایش پس ترتیبی *(postorder)* به‌کار می‌رود.

۳) این پردازه برای پیمایش میان ترتیبی *(inorder)* به‌کار می‌رود.

٤) اگر جای سطرهای ٥ و ٦ پـردازه را عـوض کنـیم، پـردازه بـرای پیمـایش پـس ترتیبی *(preorder)* به‌کار می‌رود.

۸۵. تابع *Count* برای درخت دودویی نوشته شده است. مقدار این تـابع بـرای درخـت دودویی زیر چیست؟

```
Function Count(T);
begin
       Count := 0;
       if T ≠ 0 then
          Count := 2 Count (Leftchild (rightchild(T))) +
                    Count (rightchild (leftchild(T))) + 1
end;
```

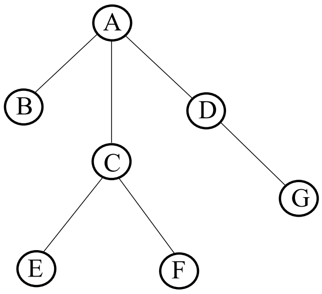


۱) ٤	۲) ٥	۳) ٦	٤) ٨

۸٦. عمق درخت دودویی معادل با عبارت محاسباتی $(-a)*b*c-d/e*g+h$ برابر است با:

۱) ٤	۲) ٥	۳) ٦	٤) ٧

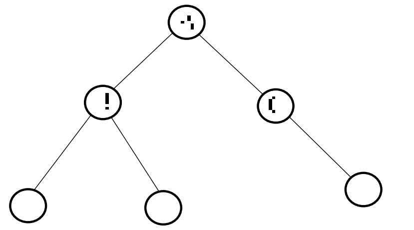
۸۷. خروجی پیمایش *postorder* درخت (توجه کنید که این درخت دودویی نیست) به قرار .... است.

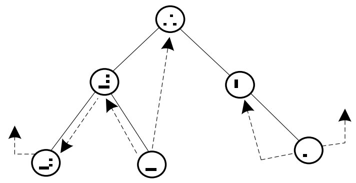


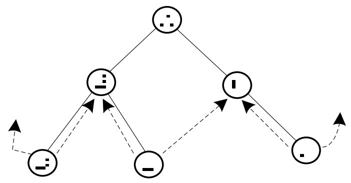
۱) *BEFCGDA*	۲) *FEGDCBA*

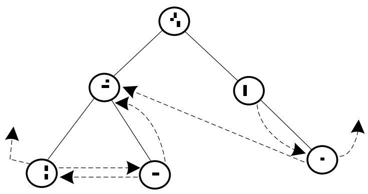
۳) *GDFECBA*	٤) *ABCEFDG*

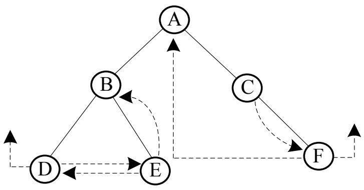
۸۸. درخت زیر را در نظر بگیرید. می‌خواهیم این درخت را برای پیمایش *inorder* نخی *(threaded)* کنیم. گزینه صحیح را انتخاب کنید.



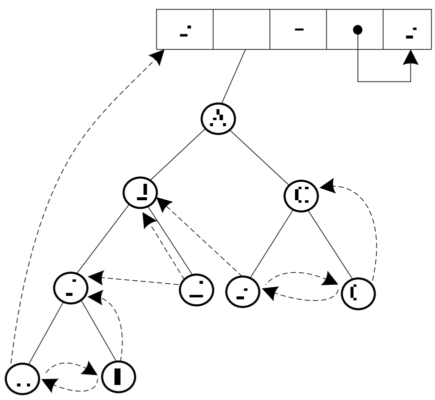
۱)


۲)


۳)


۴)


۸۹. درخت زیر به چه صورت نخ‌کشی شده است؟

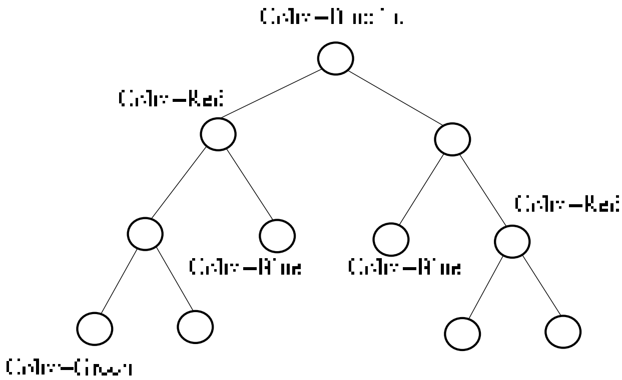


۱) *preorder* ۲) *postorder*

۳) *inorder* ٤) هیچکدام

۹۰. در درخت زیر، برای تعدادی از گره‌ها مشخصه *Color* تعریف شده است که رنگ آن گره‌ها را نشان می‌دهد. برای مشخص کردن رنگ یک گره *n* برنامه زیر را اجرا می‌کنیم:

```
Function FindColor(n) :Color;
begin
        if n has Color attribute then
           Find Color := Color(n);
        else
          FindColora := Find Color(parent(n))
end;
```



اگر *Find Color (n)* برای هر یک از ۱۱ گره‌های درخت فوق اجرا شود، در مجموع چند تعداد رنگ قرمز *(Red)* بازمی‌گردد؟

۱) ۲ ۲) ٦ ۳) ٤ ٤) ٥

۹۱. یک مجموعه *A* از اعداد صحیح دو به دو نامساوی با *n* عنصر داده شده است. می‌خواهیم ساختمان داده‌ای برای *A* طراحی کنیم تا اعمال زیر را بتوان همواره در $O(\log_2 n)$ انجام داد:

Insert(x) درج عنصر x در L.

Delete(x) حذف عنصر x از L.

Search(x) جستجو برای عنصر x در L.

Next(x) [illegible] عدد بعدی x یعنی کوچکترین عدد بزرگتر از x در L.

کدام یک از ساختمان‌های داده زیر جواب است؟

۱) درخت دودویی جستجو ۲) درخت دودویی جستجوی متوازن

۳) لیست مرتب ۴) درخت نیمه مرتب *(Heap)*

۹۲. کدام یک از گزینه‌های زیر در ارتباط با هر درخت دودویی جستجو با n عنصر غلط است؟

۱) در حذف تعدادی عناصر از درخت، ترتیب حذف تأثیری در درخت حاصل ندارد.

۲) متوسط ارتفاع کلیه درخت‌های دودویی جستجو با n المان متناسب است با $log_۲ n$

۳) اگر n عنصر را از قبل داشته باشیم می‌توان یک درخت دودویی با ارتفاع متناسب با $log_۲ n$ ایجاد کرد.

۴) می‌توان کوچکترین عنصر را در این درخت با مرتبه $O(log_۲ n)$ حذف کرد.

۹۳. صف اولویت *(Priority Queue)* با استفاده از *Heap* پیاده‌سازی شده است. کدام یک از عبارات زیر صحیح است؟ n تعداد اقلام در صف اولویت می‌باشد.

۱) عملگر «پیدا کردن ماکزیمم و حذف آن» از صف اولویت دارای پیچیدگی زمانی $O(1)$ می‌باشد.

۲) عملگر «اضافه کردن» یک قلم جدید به صف اولویت دارای پیچیدگی زمانی $O(n log_۲ n)$ می‌باشد.

۳) عملگر «پیدا کردن ماکزیمم» دارای پیچیدگی $O(log_۲ n)$ می‌باشد.

۴) عملگر «اضافه کردن» یک قلم جدید به صف اولویت دارای پیچیدگی زمانی $O(log_۲ n)$ می‌باشد.

۹۴. می‌خواهیم با وارد کردن مقادیر ۱ و ۲ و ۳ به هر ترتیب دلخواه در یک درخت

تهی دودویی جستجو *(Null Binary Search Tree)* یک درخت دودویی جستجو با ۳ گره بسازیم. چند درخت دودویی جستجوی متفاوت ممکن است ساخته شود؟

۱) ۲ درخت ۲) ۳ درخت ۳) ٦ درخت ٤) ۵ درخت

۹۵. آرایه مرتب A داده شده است. تابع زیر برای جستجوی دودویی در A (از اندیس L تا R) برای پیدا کردن x نوشته شده است.

```
Function Search (x : real; L, R : integer) : boolean;
Var M : integer;
begin
    M := (L + R) div 2;
    if (x = A[M]) then return (true);
    if (x < A[M] and (M > L) then return (search (n, L, M - 1));
    if (x > A[M] and (M < R) then return (search (n, M + 1 , R))
    return false;
end;
```

کدامیک از گزینه‌های زیر در مورد الگوریتم فوق برای جستجوی دودویی در $A[1..N]$ غلط است؟ (منظور از مقایسه، مقایسه x با یک عنصر از A است)

۱) اگر x در $A[1..N]$ باشد، همواره با حداکثر $O(\log_2 N)$ پیدا می‌شود.

۲) اگر x در $A[1..N]$ نباشد، پاسخ *false* همواره با حداکثر $O(\log_2 N)$ مقایسه به‌دست می‌آید.

۳) در جستجوی ناموفق برای دو مقدار متفاوت x (که در A موجود هستند) همواره تعداد مقایسه‌ها برابر است.

٤) حداکثر تعداد مقایسه‌ها (چه موفق چه ناموفق) همواره از $O(N)$ کمتر است.

۹٦. n عنصر با کلیدهای مختلف را می‌توان با استفاده از یک درخت دودویی جستجو T که در ابتدا تهی است به صورت زیر مرتب کرد:

۱) عناصر را به ترتیب در T درج کن.

۲) T را به روش «بین ترتیب» *(inorder)* پیمایش کن و عناصر را به همین ترتیب در خروجی بنویس. مرتبه زمان اجرای این الگوریتم به ترتیب در بهترین حالت، بدترین حالت و حالت متوسط (از راست به چپ) کدام است؟

۱) $O(n^2), O(n^2), O(n\log n)$ ۲) $O(n\log n), O(n\log n^2), O(n)$

۳) $O(n\log n), O(n^3), O(n\log n)$ ٤) $O(n\log n), O(n^3), O(n)$

۹۷. درخت AVL یک درخت دودویی است که ارتفاع دو زیر درخت هر گره آن حداکثر یک واحد با هم اختلاف دارد. (ارتفاع یک درخت تهی را ۱- فرض کنید) اگر $T(h)$ کمترین تعداد گره برای یک درخت AVL به ارتفاع h باشد، کدام یک از رابطه‌های بازگشتی زیر برای $T(h)$ درست است؟

$(T(0)-1, T(1)=2)$

۱) $T(h)=2T(h-2)+1$ ۲) $T(h)=T(h-1)+T(h-2)+1$

۳) $T(h)=2T(h-1)+1$ ٤) $T(h)=T([h/2])+T([h/2])+1$

۹۸. یک درخت دودویی جستجوی T با ۵ گره با برچسب‌های $a_1<a_2<a_3<a_4<a_5$ را در نظر بگیرید. به گره a_i عدد x_i را نسبت می‌دهیم. فرض کنید $x_1=2$ و $x_i=1$ برای $i=2..5$ می‌خواهیم T را طوری بسازیم که عبارت $C_T=\sum_{i=1}^{5}x_i(depth(a_i)+1)$ مینیمم شود. عمق گرهٔ a_i در T است. کمترین مقدار C_T چند است؟

۱) ۳ ۲) ١٤ ۳) ۱۵ ٤) ١٦

۹۹. کدامیک از گزاره‌های زیر در مورد درخت‌های دودویی جستجو غلط است.

۱) n را می‌توان بااستفاده از درخت دودویی جستجو با الگوریتم از مرتبه $O(n\log n)$ مرتب کرد.

۲) ترتیب حذف دو المان مختلف از یک درخت دودویی جستجو مهم نیست و درخت حاصل برای هر دو ترتیب یکسان خواهد بود.

۳) ارتفاع یک درخت جستجو با n المان می‌توان $n-1$ باشد.

٤) اگر درخت دودویی جستجوی T شامل عناصر $a_1<a_2<\ldots<a_n$ و اگر a_i در T هر دو فرزند چپ و راست را داشته باشد در آن صورت a_{i+1} نمی‌تواند فرزند چپ و a_{i-1} نمی‌تواند فرزند راست داشته باشد.

۱۰۰. کدامیک از عبارات زیر صحیح است؟

۱) یک درخت تصمیم‌گیری که n عدد منحصر به فرد را مرتب می‌کند دارای

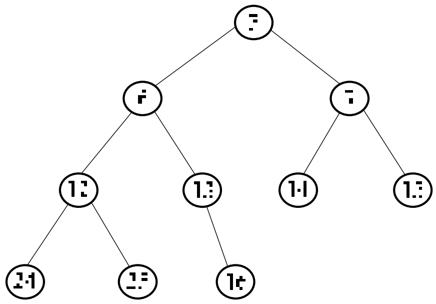
حداکثر عمق $log_۲\, n$ می‌باشد.

۲) یک درخت دودویی که دارای عمق n می‌باشد دارای حداکثر $n_۰^n$ برگ می‌باشد.

۳) یک درخت تصمیم‌گیری که n عدد منحصر به فرد را مرتب می‌کند دارای حداقل عمق $log_۲\, n$ می‌باشد.

٤) یک درخت دودویی که دارای عمق n می‌باشد دارای حداقل ۲^n برگ می‌باشد.

۱۰۱. درخت زیر:



۱) یک درخت $(Full)$ پر است.

۲) یک درخت کامل $Complete$ است.

۳) یک درخت جستجوی باینری است.

٤) هیچکدام

۱۰۲. فرض کنید فرض کنید اعداد ۱ تا ۱۰۰۰ در یک درخت دودویی جستجو ذخیره شده‌اند و ما می‌خواهیم عدد ۳٦۳ را پیدا کنیم. کدامیک از ترتیب‌های زیر (از چپ به راست) نمی‌تواند بیانگر ترتیب دسترسی به عناصر درخت در این قسمت جستجو باشد؟

۱) ۳٦۳ و ۲٤٥ و ۹۱۲ و ۲٤۰ و ۹۱۱ و ۲۰۲ و ۹۲٥

۲) و ۳٦۳ و ۳٦۲ و ۲٥۸ و ۸۹۸ و ۲٤٤ و ۹۱۱ و ۲۲۰ و ۹۲٤

۳) و ۳٦۳ و ۳۹۷ و ۳٤٤ و ۳۳۰ و ۳۹۸ و ٤۰۱ و ۲٥۲ و ۲

٤) ۳٦۳ و ۲۷۸ و ۳۸۱ و ۳۸۲ و ۲٦٦ و ۲۱۹ و ۳۸۷ و ۳۹۹ و ۲

۱۰۳. در یک درخت دودویی جستجوی T، اگر x برگ و y پدر x باشد، کدام گزینه در مورد مقادیر $a = key[x]$ و $b = key[y]$ درست است؟

۱) b بزرگترین کلید در T است که کوچکتر از a باشد.

۲) a کوچکترین کلید در T است که بزرگتر از b باشد.

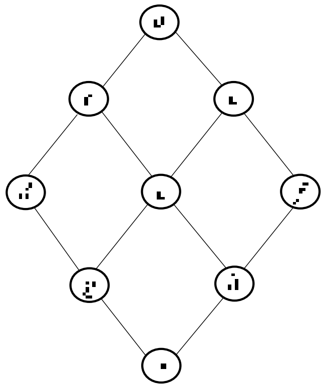
۳) b کوچکترین کلید در T است که بزرگتر از a باشد.

٤) هیچکدام از گزینه‌های فوق همواره درست نیست.

۱۰٤. حداکثر تعداد لبه‌های یک گراف جهت‌دار شامل n گره برابر است با:

۱) $۲n-۱$ ۲) $n^۲-n$ ۳) $n^۲-۱$ ٤) $۲n-n$

۱۰۵. پیمایش عمقی *(depth-first search)* گراف زیر چیست اگر از یک گره a شروع کنیم. (به ترتیب از چپ به راست)



۱) $a.b.c.d.e.f.g.h.i$ ۲) $a.b.d.g.i.h.e.c.f$

۳) $a.b.c.f.h.e.i.g.d$ ٤) $a.b.d.g.i.h.f.c.e$

۱۰٦. کدام گزینه نادرست است؟

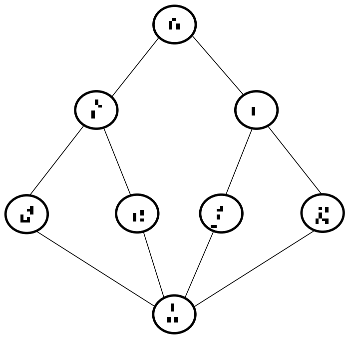
۱) تفاوتی میان گراف و گراف چندگانه وجود ندارد.

۲) حلقه یک مسیر ساده که اولین و آخرین رأس آن یکی باشد.

۳) در یک گراف بدون جهت با n رأس بیشترین تعداد لبه‌ها $n(n-۱)/۲$ است.

٤) در یک گراف جهت‌دار با n رأس بیشترین تعداد لبه‌ها $n(n-۱)$ است.

۱۰۷. پیمایش عمقی *Depth-First* گراف زیر چه خواهد بود؟ اگر از گره a شروع کنیم. (پاسخ‌ها از چپ به راست نوشته شده‌اند)



۱) $h.d.e.f.g.b.c.a$

۲) $a.c.g.g.f.b.c.d$

۳) $a.b.c.d.e.f.g.h$

٤) $a.b.d.h.e.f.c.g$

۱۰۸. در گراف G با داشتن درجه تک‌تک رئوس (یعنی d_1 برای رأس i) و تعداد رئوس (یعنی n) تعداد لبه‌ها عبارتند از:

۱) $e=\frac{1}{2}\sum_{i=1}^{n} d_i$

۲) $e=\sum_{i=1}^{n} d_i$

۳) $e=\frac{1}{2}\sum_{i=1}^{n}(d_i+d_n)$

٤) $e=\frac{1}{2}\sum_{i=1}^{n}\left(d_1-d_n\right)$

۱۰۹. کدام عبارت صحیح نیست؟

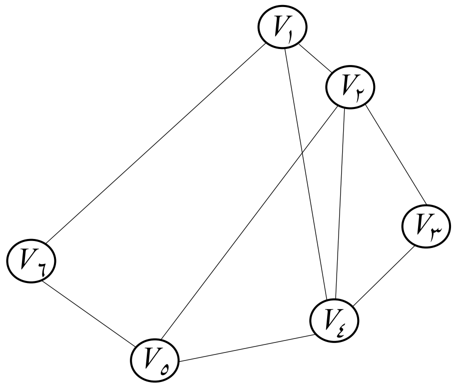
۱) درختی که تعدادی از لبه‌ها از رئوس را دربردارد، درخت پوشا گفته می‌شود.

۲) الگوریتم DFS لبه‌های T را به شکل یک درخت پوشا تعیین می‌کند.

۳) الگوریتم DFS لبه‌های T را به شکل یک درخت پوشا تعیین می‌کند.

٤) همه گراف‌های متصل با $n-1$ لبه درخت هستند.

۱۱۰. در گراف جستجوی DFS از رأس V_1 منجر می‌شود به ....



۱) $V_1.V_2.V_3.V_4.V_5.V_6$ ۲) $V_1.V_6.V_4.V_2.V_5.V_3$

۳) $V_1.V_6.V_5.V_4.V_2.V_3$ ۴) $V_1.V_6.V_4.V_2.V_3.V_5$

۱۱۱. گراف جهت‌دار *G(V, E)* که یک جنگل *(forest)* است که از ۵ مؤلفـه همبنـد بـه ب با تعدلوتیالهای ۵ و ۱۰ و ۱۵ و ۲۰ و ۲۵ تشکیل شده است. تعـداد رئـوس این گراف $(|V|)$ کدام است؟

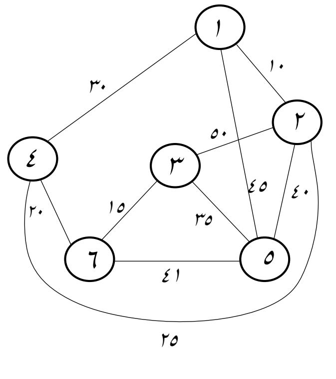
۱) ۷۰ ۲) ۷۵ ۳) ۸۰ ۴) ۸۵

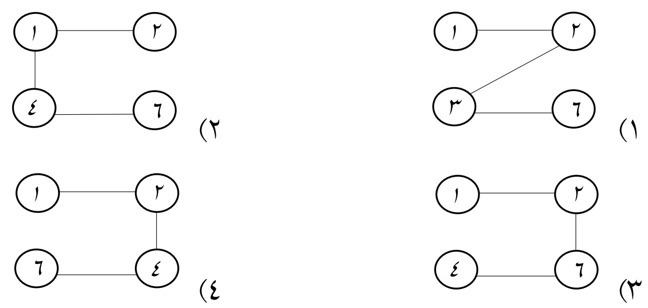
۱۱۲. فرض کنید که u و v دو نود در یک گراف بدون جهت باشند. اگر دو مسیر p_1 و p_2 وجود داشته باشد آنگاه:

۱) u و v مجاورند. ۲) G دارای سیکل است.

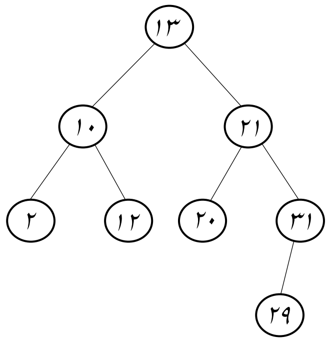
۳) نمی‌تواند یک گراف باشد. ۴) هیچکدام

۱۱۳. اگر برای پیدا کردن درخت پوشای مینیمم زیـر از الگـوریتم *Prim* اسـتفاده کنـیم کدامیک از گزینه‌های حاصله در انتهای مرحله سوم الگوریتم را به ما می‌دهد؟





۱۱۴. در شکل زیر کدامیک از انتخاب‌ها روش *Breadth-Frist Traversal* یک درخت است؟



۱) ۲، ۲۹، ۱۲، ۲۰، ۳۱، ۱۰، ۲۱، ۱۳

۲) ۲۹، ۲، ۱۲، ۲۰، ۳۱، ۲۱، ۱۰، ۱۳

۳) ۲۹، ۳۱، ۲۰، ۲۱، ۱۲، ۲، ۱۰، ۱۳

۴) ۲۹، ۳۱، ۲۰، ۱۲، ۲، ۲۱، ۱۰، ۱۳

۱۱۵. فضای موردنیاز برای نمایش یک گراف $G(V,E)$ به روش لیست همسایگی *(Adjacency List)* کدام است؟

۱) $O(|E|+|V|)$ ۲) $O(|E|)$ ۳) $O(|V|)$ ۴) $O(|E|,|V|)$

۱۱۶. الگوریتم *Quick Sort* یک رشته n تایی را با چه سرعتی مرتب می‌کند؟

۱) $O(n\log n)$ ۲) $O(n)$ ۳) $O(n^2)$ ۴) $O(\log_2 n)$

۱۱۷. اگر یک لیست مرتب شده با n خانه را با استفاده از الگوریتم *Binary Search* برای یک مقدار خاص جستجو کنیم، تعداد دفعات مقایسه چه خواهد بود؟

۱) $O(n/2)$ ۲) $O(n^2)$ ۳) $O(\log n)$ ۴) $O(\log_2 n)$

۱۱۸. کدامیک از موارد زیر صحت دارد؟

۱) یک *Heap* همیشه از نوع درخت *Binary Search* می‌باشد.

۲) یک *Heap* همیشه یک درخت دوتایی کامل می‌باشد.

۳) یک درخت دوتایی کامل همیشه یک *Heap* می‌باشد.

٤) یک درخت *Binary Search* همیشه یک *Heap* می‌باشد.

۱۱۹. کدامیک از موارد زیر در مورد (درخت انتخابی) صحیح است؟

۱) گره ریشه کوچکترین مقدار را دارد.

۲) هر گره آن بزرگ‌تر از دو فرزند است.

۳) گره ریشه بزرگترین مقدار را دارد.

٤) مقدار گره‌ها ترتیب خاصی ندارد.

۱۲۰. بدترین حالت در روش مرتب‌سازی سریع *(Quick Sort)* چیست؟

۱) عناصر لیست به ترتیب معکوس باشند.

۲) عناصر لیست از قبل مرتب باشند.

۳) یک نیمه لیست مرتب باشد.

٤) یک نیمه لیست معکوس باشد.

۱۲۱. چنانچه بخواهیم داده‌های تکراری را از لیستی حذف کنیم، از کدام ساختار داده‌ای برای لیست مزبور استفاده می‌کنیم؟

۱) درخت جستجوی دودویی ۲) درخت *Heap*

۳) پشته ٤) صف

۱۲۲. اگر N رکورد داشته باشیم تعداد کل مقایسه در روش *Linear Insertion Sort* برابر است با:

۱) $\frac{N^2}{4}$ ۲) $\frac{N(N-1)}{2}$ ۳) $\frac{N(N+1)}{4}$ ٤) $\frac{N^2}{4}$

۱۲۳. کدام عبارت صحیح نیست؟

۱) الگوریم *Heap Sort* برای مرتب کردن یک لیست n قلمی احتیاج به $O(n\log_2 n)$ زمان و $O(n\log_2 n)$ حافظه دارد.

۲) الگوریتم *Quick Sort* برای مرتب کردن یک لیست n قلمی دارای پیچیدگی زمانی $O(n^2)$ در بدترین حالت و $O(n\log_2 n)$ در بهترین حالت می‌باشد.

۳) الگوریتم *Quick Sort* برای مرتب کردن یک لیست n قلمی دارای پیچیدگی زمانی $O(n)$ در بهترین حالت و $O(n^2)$ در بدترین حالت و حالت متوسط می‌باشد.

٤) الگوریتم *Quick Sort* برای مرتب کردن یک لیست n قلمی دارای پیچیدگی زمانی $O(n^2)$ در بدترین حالت و $O(n\log_2 n)$ در حالت متوسط می‌باشد.

۱۲٤. کدامیک از گزینه‌های زیر صحیح است؟

۱) پیچیدگی الگوریتم *Quick Sort* در حالت متوسط و بدترین حالت با هم متفاوت است.

۲) پیچیدگی الگوریتم *Heap Sort* در حالت متوسط و بدترین حالت با هم یکسان است.

۳) پیچیدگی الگوریتم *Bubble Sort* در حالت متوسط و بدترین حالت با هم متفاوت است.

٤) پیچیدگی الگوریتم *Selection Sort* در بهترین حالت و حالت متوسط با هم متفاوت است.

۱۲۵. الگوریتم *Quick Sort*:

۱) همواره $O(n^2)$

۲) فقط در بدترین حالت یعنی زمانی که داده‌ها مرتب باشند $O(n^2)$ است.

۳) همواره $O(n\log n)$ است.

٤) سریعترین روش مرتب کردن است.

۱۲٦. کدامیک از جملات زیر در مورد الگوریتم‌های مرتب‌سازی درست است؟

۱) الگوریتم‌سازی مرتب‌سازی حبابی *(Bubble)* و کوئیک سورت در بدترین حالت یکسان عمل می‌کنند.

۲) الگوریتم‌های مرتب‌سازی کوئیک سورت و *Heap Sort* در بهترین حالت دارای پیچیدگی یکسانند.

۳) الگوریتم‌های کوئیک سورت و *Heap Sort* در بدترین حالت دارای پیچیدگی یکسانند.

٤) ۱ و ۲

۱۲۷. الگوریتم *Merge Sort* از نظر زمان محاسبه دارای:

۱) $O(n\log n)$ است.　　　　۲) $O(n)$

۳) $O(n\log n)$ است.　　　　۴) هیچکدام

۱۲۸. رویه زیر را در نظر بگیرید:

```
Procedure MergeSort (low, high, A)
// A[low...high] is an array Containing Values which
   represents the elements to be sorted //
begin
  if low < high then
  begin
      mid ← [(low + high) / 2]
      MergeSort (low, mid, A)
      MergeSort (mid + 1, high, A)
      Merge (low, mid, high, A)
  end;
end.
```

رویه *merge* در رویه فوق دو لیست مرتب شده $A[low...mid]$ و $A[mid+1...high]$ ادغام می‌کند. تعداد صداهای انجام گرفته به رویه فوق برای مرتب کردن لیست زیر چیست؟

۵۲۰، ۴۵۰، ۲۵۴، ۸۶۱ ،۴۲۳، ۳۵۱، ۶۲۵، ۱۷۹، ۲۸۵، ۳۱

۱) ۲۳　　　۲) ۲۸　　　۳) ۱۹　　　۴) ۲۰

۱۲۹. کدامیک از گزینه‌های زیر در مورد الگوریتم *Quick Sort* درست است؟ فرض کنید که عنصر اول همواره به‌عنوان محور *(pivot)* انتخاب می‌شود. *n* تعداد عناصر آرایه ورودی است. اگر آرایه ورودی از قبل مرتب باشد.....

۱) زمان اجرای الگوریتم $O(n)$

۲) زمان اجرای الگوریتم $O(n^2)$

۳) زمان اجرای الگوریتم $O(n\log n)$ است.

۴) متوسط زمان اجرا $O(n\log n)$ است.

۱۳۰. آرایه زیر یک *Heap* است. برای درج عدد ۹۵ در آرایه به گونه‌ای که آرایه نهـایی نیز وضعیت *heap* داشته باشد، چند عمـل *exchange* (تعـویض دو کمیـت) لازم است؟

۱۰۰
۹۰
۸۲
۸۵
۷٤
۷۵
۷۳
٦۸
۷۰

۱) دو ۲) چهار ۳) شش ٤) هشت

۱۳۱. اگر در الگوریتم مرتب‌سازی سریع *(Quick Sort)* به ترتیب صعودی، عنصر لـولا *(pivot)* را همان عنصر اول لیست بگیریم و با استفاده از آن یک بار لیست مرتـب نزولی و یکبار دیگر لیست مرتب صعودی را مرتـب کنـیم. گزینـه صـحیح بـرای مرتبه تعداد عملیات اصلی (مقایسه و جابه‌جـایی) را در ایـن دو حالـت انتخـاب کنید

۱) هر دو حالت از $O(n \log n)$

۲) هر دو حالت از $O(n^2)$

۳) برای لیست صعودی $O(n)$ و برای لیست نزولی $O(n^2)$

٤) برای لیست صعودی $O(n \log n)$ و برای لیست نزولی $O(n^2)$

۱۳۲. لیست زیر را در نظر بگیرید. اگر عنصر اول لیست یعنی عدد ۹ را بـه‌عنـوان لـولا *(pivot)* اختیار کنیم کدامیک از گزینه‌هـای زیـر مـی‌تواننـد خروجـی مرحلـه اول الگوریتم مرتب‌سازی سریع *(Quick Sort)* باشد؟

(۹، ۱۰، ۸، ۷، ٦، ۱۵، ۳)

۱) (۷، ۸، ۹، ۱۰، ۳، ٦، ۱۵)

۲) (۷، ۸، ۹، ۳، ٦، ۱۰، ۱۵)

۳) (۱۰، ۱۵، ۹، ۸، ۷، ۳، ٦)

٤) (۱۵، ۱۰، ۳، ۹، ۸، ۷، ٦)

۱۳۳. n گلوله با وزنهای مختلف را می‌خواهیم با یک ترازوی دو کفه‌ای بدون وزنه و با توزین‌های متوالی مرتب کنیم (یک توزین عبارت است از قرار گرفتن دو گلوله در دو کفه ترازو و مقایسه وزنهای آنها). کدامیک از گزینه‌های زیر درست است؟

۱) ۳ گلوله را می‌توان حداکثر با ۲ بار توزین کرد.

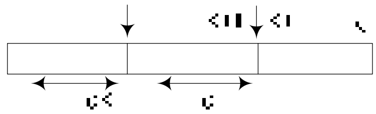
۲) ٤ گلوله را می‌توان حداکثر با ٤ بار توزین مرتب کرد.

۳) ٤ گلوله را می‌توان در مواردی با ۳ بار توزین مرتب کرد.

٤) هیچکدام

۱۳٤. الگوریتم زیر را برای مرتب‌سازی آرایه A با N عنصر (با اندیس‌های از ۱ تا N) پیشنهاد شده است. در این الگوریتم فرض کنید $Sort(A,i,j)$ عناصر I تا j از آرایه A را با یکی از الگوریتم‌های شناخته شده به‌صورت صعودی مرتب می‌کند.

[illegible]
[illegible]
[illegible]



کدام یک از گزینه‌های زیر برای هر N درست است؟

۱) تنها اگر $K \le L$ باشد این الگوریتم درست است.

۲) تنها اگر $K < L$ باشد این الگوریتم درست است.

۳) تنها اگر $N = 3K = 3L$ باشد الگوریتم درست است.

٤) تنها اگر $K = L$ باشد الگوریتم درست است.

۱۳۵. یکی از تکنیک‌های مرتب کردن اطلاعات تکنیک *Linear Insertion Sort* است که در آن برای تشخیص محل iامین رکورد باید رکورد یاد شده را بین $i-1$ رکورد مرتب شده وارد کرد. اگر N رکورد داشته باشیم. تعداد کل مقایسه برابر است با:

۱) $\frac{N^2}{4}$ ۲) $\frac{N(N+1)}{2}$ ۳) $\frac{N(N+1)}{4}$ ٤) $\frac{N^2}{4}$

۱۳٦. کدامیک از گزاره‌های زیر درست است؟

۱) الگوریتم *Quick Sort* و *Heap Sort* هر دو متعادل *(Stable)* است.

۲) الگوریتم *Quick Sort* از الگوریتم *merge sort* همواره سریعتر است.

۳) حداکثر درجه پیچیدگی الگوریتم *Binary Insertion Sort* از *Quick Sort* بهتر است.

٤) حداکثر درجه پیچیدگی الگوریتم *Quick Sort* بهتر از *Heap Sort* است.

۱۳۷. اگر بخواهیم یک لیست متصل *(Linkded List)* که آدرس اول *first* می‌باشد را با کمترین تعداد عملیات مرتب نماییم *(sort)* جای علامت ؟ در الگوریتم زیر چه مقادیری به ترتیب قرار دهیم:

$$\text{for}(p = \text{first}; ?; p = p \to \text{link})$$
$$\text{for}(q = ?; q; q = q \to \text{link})$$
$$\text{if}(p \to \text{Data} > q \to \text{Data})$$
$$\text{Swap}(\&p \to \text{data}, \&q \to \text{Data});$$

۱) P ، $P \to link$　　۲) $P \to link$ ، P　　۳) P ، P　　٤) $P \to link$ ، $P \to link$

۱۳۸. الگوریتم زیر به کدام روش عمل Sort (مرتب‌سازی) را انجام می‌دهد؟

```
XSort ( )
{
   For i=2  to  n  do
   {
      X=A[i] ;
      j= i-1;
     while ( j>◦ and A[i]>X)  d◦
     {
        A[j+1]=A[j];
        j=j-1;
     }
     A[j+1]=X;
   }
}//end X Sort
```

۱) radix　　۲) shell　　۳) Insertion　　٤) selection

۱۳۹. در الگوریتم Insertion Sort بهترین شرایط و بدترین شرایط به ترتیب از راست به چپ عبارت است از:

۱) مرتب‌شده نزولی، مرتب‌شده صعودی

۲) مرتب‌شده صعودی، مرتب‌شده نزولی

۳) مرتب‌شده صعودی، مرتب‌شده صعودی

٤) توالی عناصر ورودی اثری ندارد

۱۴۰.سه آرایه C، B، A هر کدام با n عنصر را در نظر بگیرید. به قسمی که C و A مرتب شده صعودی و B مرتب‌شده، نزولی است همچنین B و A عنصر تکراری ندارند ولی C دارد. به منظور مرتب‌سازی صعودی الگوریتمهای Selection Sort و Straight inseriom Sort و quick Sort را روی هر کدام از این آرایه‌ها اجرا می‌کنیم. در این صورت می‌توان گفت:

۱) برای B پیچیدگی زمانی quick Sort و heap Sort یکسان است.

۲) برای C برنامه quick Sort سریع‌تر از heap Sort به پایان می‌رسد.

۳) برای C پیچیدگی زمانی Selection Sort و Straight insertion Sort یکسان است.

٤) برای A اجرای Straight insertion Sort سریع‌تر از quick Sort و Selection Sort به پایان می‌رسد.

۱٤۱ .پنج فایل مرتب شده به سایزهای ۵، ۱۰، ۲۰، ۲۵ و ۳۰ داریم. می‌خواهیم از ادغام دوبه‌دو آنها یک فایل مرتب‌شده واحد شامل همه رکوردها بدست آوریم در هر ادغام رکوردهای فایل‌های ورودی ممکن است چندبار از یک فایل خوانده و در یک فایل دیگر نوشته شوند به هر کدام از این نوشتن و خواندن یک جابه‌جایی می‌گوییم. حداقل تعداد کل این جابه‌جایی برای ادغام همه فایل‌ها چقدر است؟

۱) ۱۹۵　　۲) ۲۰۰　　۳) ۱۸۵　　٤) ۲۱۵

۱٤۲.برای ادغام (merge) دو آرایه مرتب شده A و B که به ترتیب m و n عنصری هستند حداکثر چند مقایسه لازم است؟

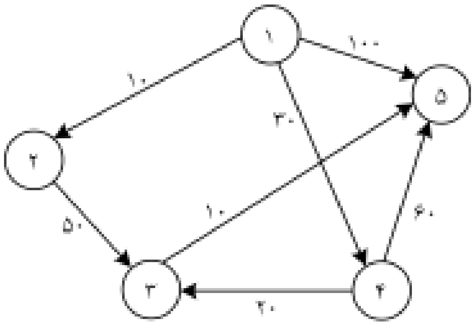
۱) n+m

۲) $n+m-1$

۳) $\max(n,m)$

۴) $\frac{(m+n)(m+n+1)}{2}$

۱۴۳.در گراف زیر کمترین هزینه از گره ۱ به ۵ دارای مسیری است با طول:

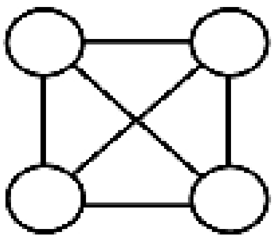


۱) ۱
۲) ۲
۳) ۳
۴) ۴

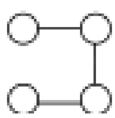
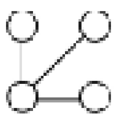
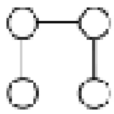
۱۴۴.پیچیدگی کدام یک از الگوریتمهای مرتب‌کننده زیر (برحسب تابعی از اندازه ورودی) در حالت متوسط (Average Case) و در بدترین حالت (Worst Case) با هم متفاوت است؟

۱) Quick Sort
۲) Binary Insertion Sort
۳) Heap Sort
۴) Merge Sort

۱۴۵.گراف زیر داده شده است:



کدامیک از انتخابهای زیردرخت پوشای این گراف است؟

۱) 
۲) 
۳) 
۴) هر سه

منابع و مراجع

1. Writing a Correct version of binary search id discussed in the papers J.Bentley, “programming peals: Writing Correct programs, CACM,” Vol. 26,1983, PP.1040-1045, and Levisse, “Some Lessons drawn form the history of the binary search Algorighm” The Computer Journal, vol. 26, 1983, pp.154-163.
2. Several texts discuss array representation in C.including T.Plum, Reliable Data Structures in C.Plum Hall.
3. Several texts discuss the Precedence hierarchy used in C.Among the references you night live to Look at are S.ltarbison and G.Steele, C.
4. Data Structure and Algorithm with C.tanen bavn 1994.
5. Prinsiple of duta structure with C.Allis Horwit Z.Sahni. Anderson. 1996.
6. For other representation of trees, see D.Knuth, The art of computer programming: Fandumental Algorithms, Second Edition Addison-wesley.
7. Alfred V.Aho, John E.Hopcroft, and Jeffrey D, Vllman. The Design and Analysis of Computer Alyoritm. Addison-We Log 1974.
8. More details about the primality testing algoritm can be found in Introduction to Algoritms, by T. H. Cormen, C. E.Leiser son, and R. L. Rivest, MIT press, 1990.
9. The Two greedy methods for obtaning minimum – Cost Spaning trees cme clue to R. C. Prim and J. B. Krusked, respectively.
10. R. Floyd, The dynamic programing formulation for the Shourtest-path problem.
11. Kruskal, J.B, on the shortest Spaning subtree of a graph and the trauelig salesman problem, proceedings of the American me The matical Jonranal, Vol 7, 1956.
12. Prim, R. C., Shortest Connection networks and Some generalizations, Bell System technical Jonrnal, Vol 36, 1957.
13. Horwowitz, E, S,. Sahni and D. Mehta, Dutu Structures and Algorithem in Ctt, Computer Science, press, 1995.
14. Horowitz, E. S. Sahni and S.Reja Sekaran, Computer Algoritms, W. H. Freeman and Company, 1998.

۱۵. جعفر تنها ـ احمد فراهی؛ تحلیل و طراحی الگوریتم، انتشارات پیام‌نور، ۱۳۸۶.