داده ساختارها و الگوریتم ها

مدرس:

کیارش شکیبائی

گروه مهندسی کامپیوتر دانشگاه آزاد اسلامی واحد تبریز

زمستان ۱۴۰۴

فهرست مطالب

مقدمه

- ساختار داده روشی متقارن برای سازمان‌دهی داده‌ها برای استفاده کارآمد از آن‌ها محسوب می‌شود.
- ساختارهای داده انواع داده‌های اولیه مانند اعداد، کاراکترها، مقادیر بولین و اعداد صحیح را در یک قالب منسجم ترکیب می‌کنند. به تنهایی، هر کدام از این نوع داده‌های اولیه فقط یک مقدار واحد دارند. اما وقتی در یک ساختار داده ترکیب شوند، عملیات سطح بالاتری مانند مرتب‌سازی، جستجو، درج و حذف را امکان‌پذیر می‌کنند.

چند اصطلاح

- داده: منظور از داده مقادیر یا مجموعه مقادیر هستند.
- خصوصیت و موجودیت: منظور از موجودیت چیزی است که خصوصیات یا مشخصات معینی دارد که به بیان دیگر مقادیر انتسابی آن هستند.
- مجموعه موجودیت: موجودیت‌های با مشخصات یکسان، یک مجموعه موجودیت را تشکیل می‌دهند.
- فیلد: فیلد یک واحد ابتدایی از اطلاعات است که خصوصیت یک موجودیت را نشان می‌دهد.
- رکورد: رکورد مجموعه‌ای از مقادیر فیلد با موجودیت‌های مفروض است.
- فایل: منظور از فایل مجموعه‌ای از مقادیر فیلدها برای یک موجودیت مفروض است.

انتزاع

- نوع داده مجرد یا انتزاعی (ADT) که فقط شامل مشخصات داده ها و اعمالی که بر روی آنها انجام می شود نه نحوه اجرا. این دو اصطلاح تقریباً معادل Abstract در زبان انگلیسی هستند .چکیده کمی با خلاصه فرق دارد. خلاصه شکل کوتاه‌شده یک متن است. اما چکیده، عصاره و مهم‌ترین نکته‌های آن است.
- سطوح انتزاع شامل سه لایه اصلی است: سطح منطقی (تعریف عملیات و روابط)، سطح مفهومی (مدل کلی داده‌ها) و سطح فیزیکی (ذخیره‌سازی واقعی).

سیکل زندگی نرم افزار-نیازمندی ها

نیازمندیها

- تمام پروژه های بزرگ برنامه نویسی با مجموعه ای از مشخصات و خصوصیاتی که اهداف پروژه را مشخص می کند، شروع می شود.
- این نیازمندیها اطلاعاتی را به برنامه نویسان می دهند(ورودی) و نیز نتایجی را که باید ایجاد گردد(خروجی) تعیین می کنند.

۱-۱ سیکل زندگی نرم افزار- طراحی

طراحی

- این مرحله ادامه کاری است که در مرحله تحلیل انجام می شود.
- طراح سیستم را از دو نقطه نظر بررسی می کند:

1. از نظرداده های مقصود(data objects) مورد نیاز برنامه

ایجاد نوع داده مجرد

2. از نظر اعمالی که بر روی آنها انجام می شود. این دیدگاه به مشخصات الگوریتم ها و فرضیات خط مشی ها ی طراحی الگوریتم نیاز دارد.

۱-۱ سیکل زندگی نرم افزار- ...

- **پالایش(اصلاح) و کدنویسی:** در این مرحله، نمایشی برای داده های مقصد خود انتخاب می شود و برای هر عملی که بر روی آنها انجام می شود، الگوریتم نوشته می شود.

- **بازبینی:** در این مرحله درستی برنامه ها اثبات می شود و برنامه ها با انواع داده های ورودی مختلف تست و خطاهای برنامه رفع می شود.

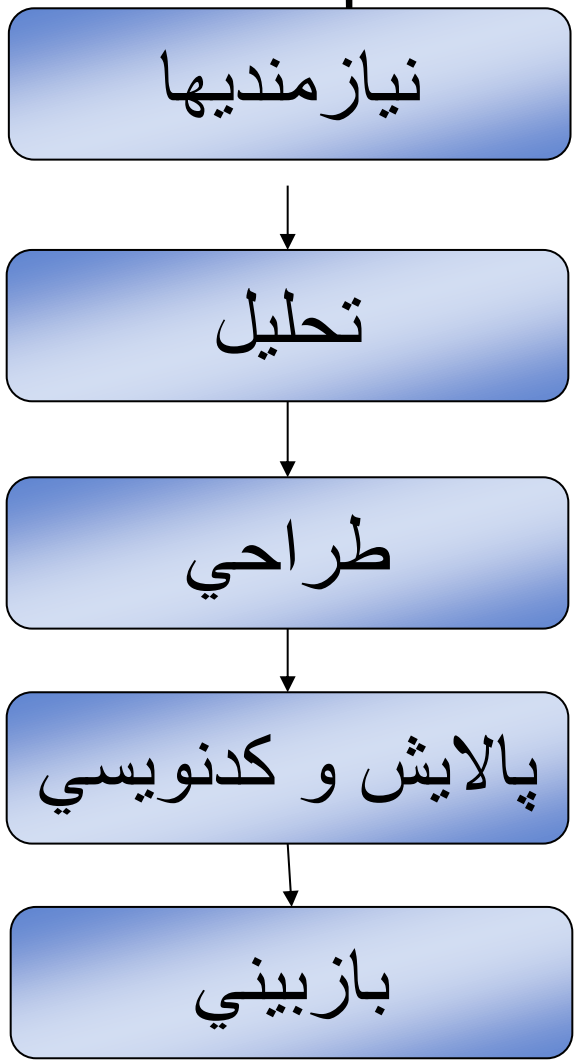
جنبه های مهم بازبینی:

اثبات درستی

تست

اشکال زدایی

۱-۱ نمودار سیکل زندگی نرم افزار



۲-۱ تعریف الگوریتم

الگوريتم مجموعه اي از دستورالعمل ها است كه
اگر دنبال شوند، موجب انجام كار خاصي مي گردد

- **یک الگوریتم یک توالی از گام های محاسباتی است که ورودی را به خروجی تبدیل می کند.**
- **الگوریتم را می توان به عنوان وسیله ای برای حل یک مساله محاسباتی خوش تعریف در نظر گرفت**

۲-۱ شرایط الگوریتم

- **ورودی:** یک الگوریتم می تواند هیچ یا چندین کمیت ورودی داشته باشدکه از محیط خارج تامین می شود.
- **خروجی:** الگوریتم بایستی حداقل یک کمیت به عنوان خروجی داشته باشد.
- **قطعیت:** هر دستورالعمل باید واضح و بدون ابهام باشد.
- **محدودیت:** اگر دستورالعمل های یک الگوریتم را دنبال کنیم ،برای تمام حالات ، الگوریتم باید پس از طی مراحل محدودی خاتمه یابد.
- **کارایی:** هر دستورالعمل باید به گونه ای باشد که با استفاده از قلم و کاغذ بتوان آن را با دست نیز اجرا نمود.

۲-۱ مثالی از الگوریتم: الگوریتم مرتب سازی

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

for (i=0 ; i<n ;i++){

Examine list [i] to list [n-1] and suppose that the smallest integer is at list [min];

Interchange list [i] and list [min];

مثال الگوریتم جستجوی دودویی

```
int binsearch ( int  list [ ]  ،int  searchnum  ،int  left  ،int  right )
{
/* search  list [0] <=  list [1] <=  … <=list [ n-1 ]  for  searchnum  Return  its  position  if  found .
Otherwise  return  -1  */
        int  middle ;
        if (left <= right )  {
                middle = ( left + right ) / 2 ;
                switch  ( COMPARE ( list [ middle ]  ،searchnum ))  {
                        case -1 :  return
                binsearch ( list  ،searchnum  ،middle +1  ،right ) ;
                        case 0 :  return  middle ;
                        case 1 :  return
                binsearch ( list  ،searchnum  ،left  ،middle -1 ) ;
                }
        }
        return  -1 ;
}
```

۳-۱ آرایه ، ساختار و نوع داده

آرایه — مجموعه ای از عناصر از یک نوع داده می باشد

ساختار — مجموعه ای از عناصر است که لزومی ندارد داده های آن یکسان باشد

نوع داده — مجموعه ای از انواع داده مقصد (object) و عملکردهایی است که بر روی این نوع داده ها عمل می کنند

۳-۱ نوع داده ای مجرد

- **نوع داده مجرد یا انتزاعی (ADT) که شامل مشخصات داده ها و اعمالی که بر روی آنها انجام می شود.**
- **جهت جداسازی پیاده سازی و نمایش داده ها از یکدیگر.**

۱-٤ تحلیل نحوه اجرای یک برنامه

معیارهای محک زدن یک برنامه

- **آیا برنامه اهداف اصلی کاری را که می خواهیم، انجام می دهد؟**
- **آیا برنامه درست کار می کند؟**
- **آیا برنامه مستند سازی شده است تا نحوه استفاده و طرز کار با آن مشخص شود؟**
- **آیا برنامه برای ایجاد واحدهای منطقی ، به طور موثر از توابع استفاده می کند؟**
- **آیا کد گذاری خوانا است؟**
- **آیا برنامه از حافظه اصلی و کمکی به طور موثری استفاده می کند؟**
- **آیا زمان اجرای برنامه برای هدف شما قابل قبول است؟**

این قسمت مربوط به تخمین های حافظه و زمان مورد نیاز بوده و مستقل از ماشین است . این قسمت را تحلیل نحوه اجرای برنامه می نامیم.

۱-٤ میزان حافظه یا پیچیدگی فضای یک برنامه

میزان حافظه یا پیچیدگی فضای یک برنامه مقدارحافظه
مورد نیازبرای اجرای کامل یک برنامه است.
میزان یا پیچیدگی زمان یک برنامه مقدار زمان کامپیوتر
است که برای اجرای کامل برنامه لازم است.

٤-١ میزان حافظه

فضای مورد نیاز یک برنامه شامل موارد زیر است :

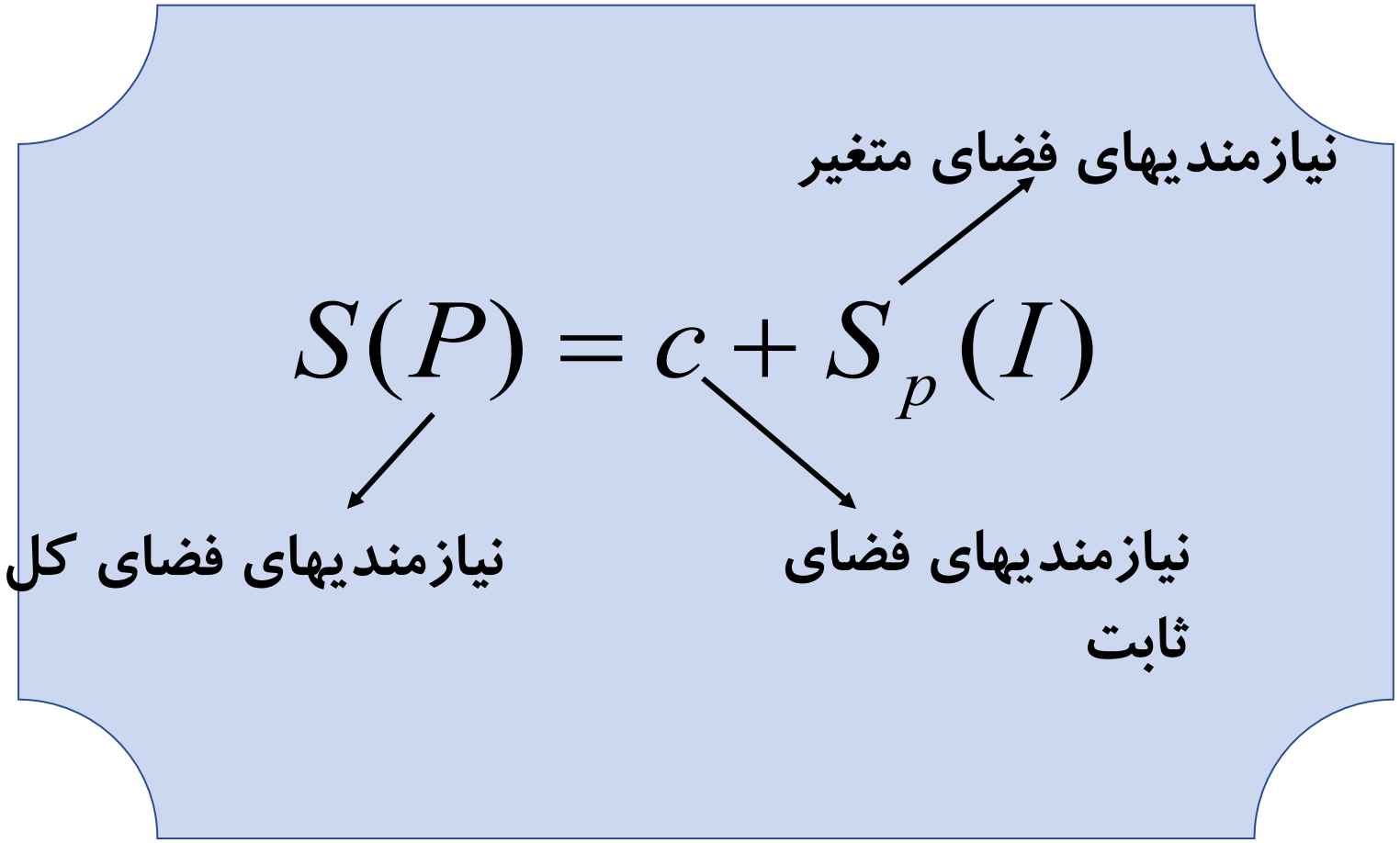
نیازمندیهای فضای ثابت

این مطلب به فضای مورد نیازی که به تعداد و اندازه ورودی و خروجی بستگی ندارد، اشاره دارد.

نیازمندیهای فضای متغیر

این مورد شامل فضای مورد نیاز متغیرهای ساخت یافته ای است که اندازه آن بستگی به نمونه I از مساله ای که حل می شود، دارد.

۱-٤ میزان حافظه(ادامه)



۱-٤ زمان T(P) برنامه

زمان T(P) برنامه عبارتست ازمجموع زمان کامپایل و زمان اجرای برنامه.
زمان کامپایل مشابه اجزای فضای ثابت است زیرابه خصیصه های نمونه بستگی ندارد.

$T_p(n)$ را به صورت زیر تعریف می شود:

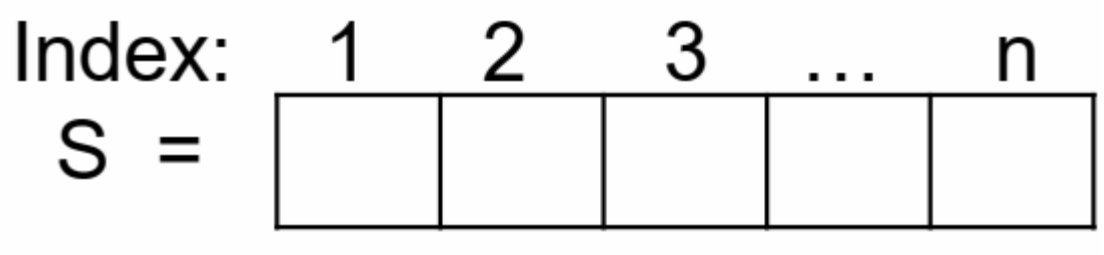
$$T_p(n) = c_a ADD(n) + c_s SUB(n) + c_l LDA(n) + c_{st} STA(n)$$

۱-۴ مرحله برنامه

یک مرحله برنامه ، قسمت با معنی برنامه است
(از لحاظ معنایی یا نحوی) که زمان اجرای آن
مستقل از خصیصه های نمونه باشد

الگوریتم جستجوی ترتیبی

- مساله: آیا کلید xدر آرایه Sبا nکلید قرار دارد؟
- ورودی ها (پارامترها):
- عدد صحیح و مثبت n
- آرایه S از کلیدها با اندیس ۱تا n و کلید x



- خروجی:
- Location: مکان x در S
- اگرx در S نباشد خروجی صفر است.

یادآوری در خصوص نماد سیگما

$$\sum_{i=1}^{n} 1 = n \qquad \sum_{i=1}^{n} kf(i) = k\sum_{i=1}^{n} f(i)$$

K عدد ثابتی است.

$$\sum_{i=1}^{n} i = 1+2+3+\ldots+n = \frac{n(n+1)}{2} \qquad \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

مثال

- دستور مبنایی در کد زیر چند بار تکرار می شود؟

```
int n,m;
 for (int i = 1; i <= m; i++)
     for (int j = 1; j <= n; j++)
         x = x + 1;
```

$$\text{تعداد اجرای دستور اصلی} = \sum_{i=1}^{m}\sum_{j=1}^{n} 1 = \sum_{i=1}^{m} n = n\left(\sum_{i=1}^{m} 1\right) = nm$$

مثال

- دستور مبنایی در کد زیر چند بار تکرار می شود؟

```
int x;
for (int j = 1; j <= n; j++)
    for (int i = 1; i <= j; i++)
        x = x + 1;
```

- راه حل اول

حلقه‌های داده شده به یکدیگر وابسته‌اند :

j	تغییرات i	تعداد اجرا شدن دستور اصلی
1	1	1 بار
2	1,2	2 بار
3	1,2,3	3 بار
	.....	
n	1,2,3,...,n	n بار

x: = x +1; تعداد اجرا شدن دستور $= 1+2+3+\ldots+n = \frac{n(n+1)}{2}$

- راه حل دوم:

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

آنالیز پیچیدگی زمانی برای مرتب سازی تعویضی

```
void exchangesort(int n, keytype S[ ] )
{
    index  i, j ;
    for( i = 1; i <= n-1; i++)
       for( j = i+1; j <= n; j++)
          if( S[j] < S[i] )
              exchange  S[i],S[j];
}
```

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

$$=(n-1)+(n-2)+\ldots+1$$

$$=1+2+3+\ldots+(n-2)+(n-1)$$

$$=[1+2+\ldots+n]-n$$

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

آنالیز پیچیدگی زمانی برای ضرب ماتریس ها

```
void matrixmult( int n,
                 const number A[ ][ ],
                 const number B[ ][ ],
                  number C[ ][ ] )
{
    index i, j, k ;
    for( i=1; i <= n; i++)
    {
        for(j=1; j <= n; j++)
        {
            C[i][j] = 0 ;
            for( k=1; k <= n; k++)
              C[i][j] = C[i][j] + A[i][k] * B[k][j] ;
        }
    }
}
```

$$T(n) = \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n} 1 = \sum_{i=1}^{n}\sum_{j=1}^{n} n = \sum_{i=1}^{n} n^2 = n^3$$

مثال

```
int i,n,x;
i = n;
while (i > 1)
{
    x = x + 1;
    i = i / 2;
}
```

تعداد اجرا شدن دستور اصلی	شرط i>1	i
۱ بار	درست	14
۱ بار	درست	7
۱ بار	درست	3
–	غلط	1
جمعاً ۳ بار		

تعداد اجرا شدن دستور اصلی	شرط i>1	i
۱ بار	درست	16
۱ بار	درست	8
۱ بار	درست	4
۱ بار	درست	2
–	غلط	1
جمعاً ٤ بار		

پس در حالت کلی دستور اصلی به تعداد $\lfloor \log_2^n \rfloor$ بار اجرا می‌شود.

مثال

```
int i, j, n, x,k;
  for (i = 1; i < n / 2; i++)
      for (j = n / 2; j < n; j++)
          for (k = 0; k < i + j; k++)
              x = x + 1;
```

- حلقه اول $\frac{n}{2} - 1$ بار تکرار می شود، حلقه دوم از $\frac{n}{2}$ تا $n - 1$ تکرار می شود و حلقه سوم از
  تا $i + j - 1$ تکرار می شود و یک دستور با هزینه ثابت اجرا می شود.

$$T(n)=\sum_{i=1}^{\frac{n}{2}-1}\sum_{j=\frac{n}{2}}^{n-1}\sum_{k=0}^{i+j-1}C$$

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

$$=C\sum_{i=1}^{\frac{n}{2}-1}\sum_{j=\frac{n}{2}}^{n-1}i+C\sum_{i=1}^{\frac{n}{2}-1}\sum_{j=\frac{n}{2}}^{n-1}j$$

$$=C\sum_{i=1}^{\frac{n}{2}-1}\frac{n}{2}i+C\sum_{i=1}^{\frac{n}{2}-1}\frac{n}{4}\left(\frac{3n}{2}-1\right)$$

$$=C\frac{n}{2}\times\frac{n}{4}\left(\frac{n}{2}-1\right)+C\frac{n}{4}\left(\frac{3n}{2}-1\right)\left(\frac{n}{2}-1\right)=\theta(n^3)$$

تحلیل پیچیدگی زمانی

- تعیین تعداد دفعاتی است که عمل مبنایی به ازای هریک از مقادیر اندازه ورودی انجام می شود.
- برای یافتن عمل مبنایی معمولا هیچ قاعده مشخص و صریحی وجود ندارد و دستوراتی که در ساختارهای تکرار، فراخوانی های بازگشتی و... وجود دارند را می توان به عنوان اعمال مبنایی در نظر گرفت.
- در هر الگوریتم بر اساس دستورات و ساختار آن باید اعمال مبنایی مشخص و مرتبه رشد و دفعات انجام آن که نتیجه آن تابع پیچیدگی الگوریتم است، بدست آید.
- در اکثر موارد منظور از پیچیدگی یک الگوریتم، پیچیدگی زمانی است.

نمونه ای از تحلیل در پیچیدگی زمانی جستجوی ترتیبی

- بدترین حالت $W(n) = n$
- حالت متوسط(فرض بر این که مقدار مورد جستجو در آرایه وجود دارد.)

$$A(n) = \sum_{k=1}^{n} (k \times \frac{1}{n}) = \frac{1}{n} \times \sum_{k=1}^{n} k = \frac{1}{n} \times \frac{n(n+1)}{2} = \frac{n+1}{2}$$

- حالت متوسط(فرض بر این که مقدار مورد جستجو در آرایه وجود ندارد.)

$$A(n) = \sum_{k=1}^{n} (k \times \frac{P}{n}) + n(1-p) = \frac{P}{n} \times \frac{n(n+1)}{2} + n(1-p) = \frac{n+1}{2} = n(1 - \frac{P}{2} + \frac{P}{2}$$

ادامه تحلیل پیچیدگی زمانی جستجوی ترتیبی

- با انتخاب مقدار ۱ برای P

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

- و مقدار ۱/۲ برای P

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

- بهترین حالت

$$B(n) = 1$$

-

رشد توابع –ترتیب(Order)

- مرتبه رشد زمان اجرای یک الگوریتم یک توصیف ساده ای از کارآیی الگوریتم ها و سنجشی از کارآیی در مقادیر بسیار بزرگ ورودی هاست.
- انتخاب اندازه ورودی بسیار بزرگ برای ساختن مرتبه رشد مناسب برای زمان اجرا جهت بررسی کارآیی مجانبی الگوریتم ها
- معرفی نمادهای استاندارد برای بیان کارآیی الگوریتم ها و نمایش آن ها

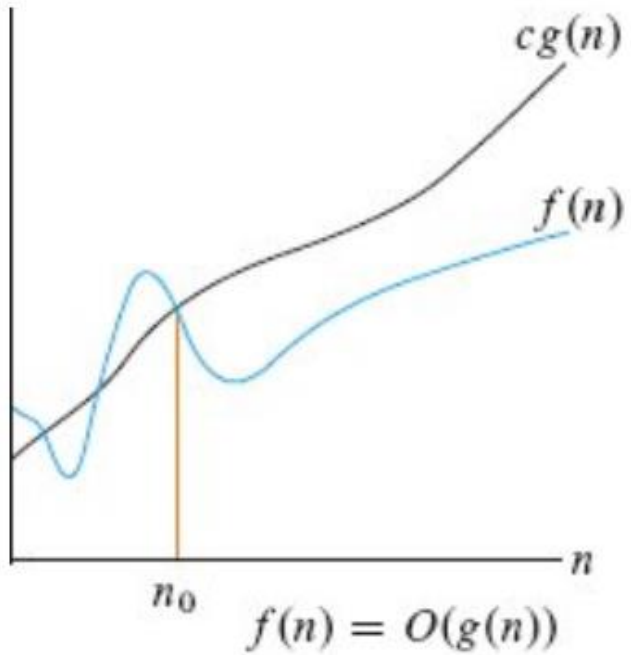
نمادهای مجانبی O، Ω و θ

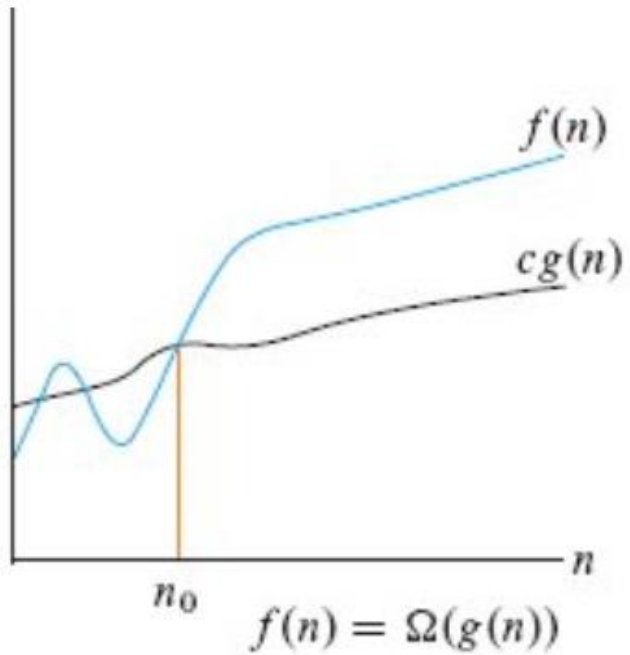
$$\forall n \geq n_0 \quad \exists\, c, n_0 > 0;\ 0 \leq f(n) \leq cg(n) \Leftrightarrow f(n) \in O\big(g(n)\big) \quad or\ f(n) = O(g(n))$$

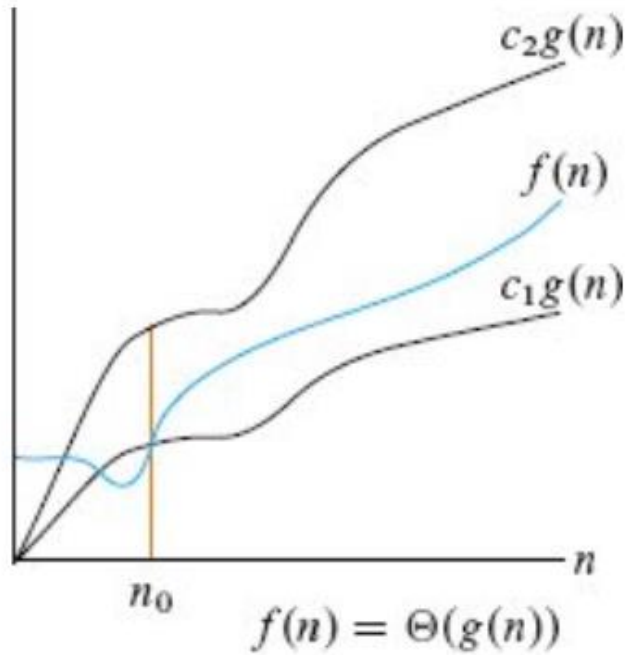
$$\forall n \geq n_0 \quad \exists\, c, n_0 > 0;\ 0 \leq cg(n) \leq f(n) \Leftrightarrow f(n) \in \Omega\big(g(n)\big) \quad or\ f(n) = \Omega(g(n))$$

$$\forall n \geq n_0 \quad \exists\, c_1, c_2, n_0 > 0;\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \Leftrightarrow f(n) \in \theta\big(g(n)\big) \quad or\ f(n) = \theta(g(n))$$

نمودارهای مربوط به نمادهای مجانبی







نمونه ای از اثبات درستی نمادهای مجانبی

- نماد O

$$f(n) = 4n^2 + 100n + 500 = O(n^2)) \; if \; g(n) = n^2$$

$$\forall \; n \geq n_0 \; 4n^2 + 100n + 500 \leq cn^2$$

$$4 + \frac{100}{n} + \frac{500}{n^2} \leq c \; if \; n_0 = 1 \; then \; c = 604 \; \rightarrow \; f(n) = O(g(n))$$

- نماد Ω

$$f(n) = 4n^2 + 100n + 500 = \Omega(n^2)) \; if \; g(n) = n^2$$

$$\forall \; n \geq n_0 \; 4n^2 + 100n + 500 \geq cn^2$$

$$4 + \frac{100}{n} + \frac{500}{n^2} \geq c \; if \; n_0 > 0 \; then \; c = 4 \; \rightarrow \; f(n) = \Omega(g(n))$$

نماد o و ω

$$\forall\, c > 0 \quad \exists\, n_0 > 0;\ 0 \le f(n) < cg(n) \Leftrightarrow f(n) \in o\big(g(n)\big)\ or\ f(n) = o(g(n))$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

$$\forall\, c > 0 \quad \exists\, n_0 > 0;\ 0 \le cg(n) < f(n) \Leftrightarrow f(n) \in \omega\big(g(n)\big)\ or\ f(n) = \omega(g(n))$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

مقایسه توابع

- تعدی

- $f(n) = \theta(g(n)) , g(n) = \theta(h(n)) \iff f(n) = \theta(h(n))$
- $f(n) = O(g(n)) , g(n) = O(h(n)) \iff f(n) = O(h(n))$
- $f(n) = \Omega(g(n)) , g(n) = \Omega(h(n)) \iff f(n) = \Omega(h(n))$
- $f(n) = o(g(n)) , g(n) = o(h(n)) \iff f(n) = o(h(n))$
- $f(n) = \omega(g(n)) , g(n) = \omega(h(n)) \iff f(n) = \omega(h(n))$

- بازتابی(انعکاسی)

- $f(n) = O(f(n))$
- $f(n) = \theta(f(n))$
- $f(n) = \Omega(f(n))$

مقایسه توابع

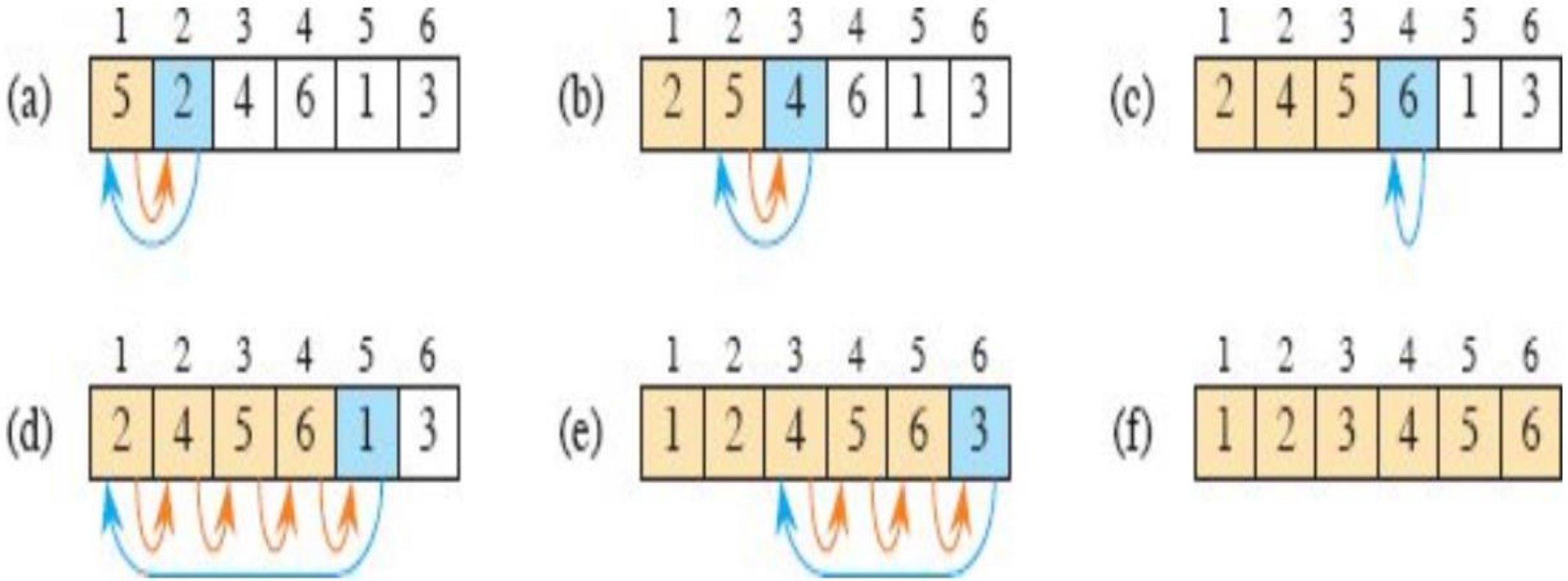
- تفارن
- $f(n) = \theta(g(n)) \iff g(n) = \theta(f(n))$
- ترانهاده تقارن
- $f(n) = O(g(n)) \iff g(n) = \Omega(f(n))$
- $f(n) = o(g(n)) \iff g(n) = \omega(f(n))$
- تشبیه مقابسه ای بین دو تابع f و g و عددهای حقیقی a و b
- $f(n) = O\big(g(n)\big) \approx a \leq b$
- $f(n) = \Omega\big(g(n)\big) \approx a \geq b$
- $f(n) = \theta\big(g(n)\big) \approx a = b$
- $f(n) = o\big(g(n)\big) \approx a < b$
- $f(n) = \omega\big(g(n)\big) \approx a > b$

۴-۱مرتب سازی درجی (Insertation Sort)

۴-۱مرتب سازی درجی (Insertation Sort)

```
INSERTION-SORT(A, n)
1  for i = 2 to n
2      key = A[i]
3      // Insert A[i] into the sorted subarray A[1 : i – 1].
4      j = i – 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j – 1
8      A[j + 1] = key
```

۴-۱مرتب سازی درجی (Insertation Sort)



۱-۵ تحلیل الگوریتم مرتب سازی درجی (Insertation Sort)

INSERTION-SORT(A, n)	*cost*	*times*
1 **for** $i = 2$ **to** n	c_1	n
2 $key = A[i]$	c_2	$n-1$
3 // Insert $A[i]$ into the sorted subarray $A[1 : i-1]$.	0	$n-1$
4 $j = i-1$	c_4	$n-1$
5 **while** $j > 0$ and $A[j] > key$	c_5	$\sum_{i=2}^{n} t_i$
6 $A[j+1] = A[j]$	c_6	$\sum_{i=2}^{n} (t_i - 1)$
7 $j = j-1$	c_7	$\sum_{i=2}^{n} (t_i - 1)$
8 $A[j+1] = key$	c_8	$n-1$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{i=2}^{n} t_i + c_6 \sum_{i=2}^{n} (t_i - 1) + c_7 \sum_{i=2}^{n} (t_i - 1) + c_8(n-1) .$$

تحلیل پیچیدگی زمانی

- بهترین حالت زمانی اتفاق می افتد که ارایه ورودی به طور پیش فرض مرتب است.
- در این حالت حلقه **while** همواره درست خواهد بود در هر بار اجرا و در نتیجه $t_i=1$ شده و زمان اجرا به صورت یک رابطه خطی تبدیل می شود.

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \ . \end{aligned}$$

پیچیدگی زمانی در بدترین حالت

- زمانی اتفاق می افتد که آرایه ورودی به صورت معکوس(نزولی) مرتب باشد.
- هر عنصر آرایه باید با تمام عناصر زیر آرایه از ۱ تا i-1 مقایسه شود و حلقه while زمانی نقض خواهد شد که j برابر صفر باشد.

$$\sum_{i=2}^{n} i = \left(\sum_{i=1}^{n} i\right) - 1$$

$$= \frac{n(n+1)}{2} - 1 \quad \text{(by equation (A.2) on page 1141)}$$

and

$$\sum_{i=2}^{n} (i-1) = \sum_{i=1}^{n-1} i$$

$$= \frac{n(n-1)}{2} \quad \text{(again, by equation (A.2))} ,$$

we find that in the worst case, the running time of INSERTION-SORT is

$$
\begin{aligned}
T(n) = {} & c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\
& + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\
= {} & \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\
& - (c_2 + c_4 + c_5 + c_8) . \qquad (2.2)
\end{aligned}
$$

پیچیدگی زمانی در بدترین حالت و حالت متوسط

- ***حد بالا برای زمان اجرا روی هر ورودی که تضمین می کند هرگز الگوریتم زمانی طولانی تر از این را صرف نخواهد کرد.***
- ***برای برخی الگوریتم ها اغلب بدترین حالت اتفاق می افتد، مثلا جستجوی یک قطعه که در پایگاه داده موجود نیست.***
- ***حالت میانگین اغلب تقریبا به همان بدی پیچیدگی در بدترین حالت ممکن است. می خواهیم n را به طور تصادفی مرتب کنیم.فرض کنید نیمی از اعداد در آرایه A[1...i-1] کوچکتر از A[i] هستند و نیمی دیگر بزرگتر هستند. بنابراین تنها در حالت میانگین نیمی از آرایه نیاز به مرتب سازی خواهد داشت و در نهایت t_i برابر i/2 خواهد بود. که در صورت محاسبه حالت متوسط نیز تابعی درجه دو همانند بدترین حالت بدست خواهد آمد.***

تقسیم و حل(Divide and Conquer)

- روشی که به طور بازگشتی زیر مسائل را حل می کند و سپس این زیر مسائل را برای بدست آوردن حل مساله اصلی با هم ترکیب می کند. مراحل این روز به شکل زیر است:

1) **تقسیم** مساله به تعدادی زیر مساله
2) **حل** زیر مسائل بوسیله حل بازگشتی ان ها
3) **ترکیب** زیر مسائل برای حل مساله اصلی

نمونه هایی از الگوریتم های تقسیم و حل

- جستجوی دودویی
- مرتب سازی ادغام
- مرتب سازی سریع
- مساله تورنومنت
- ضرب ماتریس ها
- ضرب اعداد صحیح بزرگ
- یافتن بیشترین و کمترین مقدار عناصر در آرایه
- و...

تکرارهای الگوریتمی

- ما به طور خاص به بازگشت‌هایی علاقه‌مند خواهیم بود که زمان اجرای الگوریتم‌های تقسیم و حل را توصیف می‌کنند. یک بازگشت T(n) الگوریتمی است اگر برای هر ثابت آستانه به اندازه کافی بزرگ $n_0 > 0$، دو ویژگی زیر برقرار باشد:

- برای همه $n < n_0$، داریم $T(n) = \theta(1)$
- برای همه $n \geq n_0$، هر مسیر بازگشتی در یک حالت پایه تعریف شده در تعداد محدودی از فراخوانی‌های بازگشتی خاتمه می‌یابد.

قراردادهای مربوط به تکرارها

- ما قرارداد زیر را می‌پذیریم:
- هرگاه یک بازگشت بدون یک حالت پایه صریح بیان شود، فرض می‌کنیم که بازگشت الگوریتمی است.
- این بدان معناست که شما می‌توانید هر ثابت آستانه به اندازه کافی بزرگ n_0 را برای محدوده حالت‌های پایه که در آن $T(n) = \Theta(1)$ است، انتخاب کنید. جالب توجه است که راه‌حل‌های مجانبی اکثر بازگشت‌های الگوریتمی که احتمالاً هنگام تجزیه و تحلیل الگوریتم‌ها مشاهده خواهید کرد، به انتخاب ثابت آستانه بستگی ندارند، مادامی که به اندازه کافی بزرگ باشد تا بازگشت را به خوبی تعریف کند.

قراردادهای مربوط به تکرارها–ادامه

- ممکن است گاهی اوقات با بازگشت‌هایی مواجه شوید که معادله نیستند، بلکه نابرابری‌هایی مانند $T(n) \leq 2T\left(\frac{n}{2}\right) + \theta(n)$ هستند. از آنجا که چنین بازگشتی فقط یک حد بالا روی $T(n)$ بیان می‌کند، ما راه‌حل آن را با استفاده از نماد **O** به جای نماد θ بیان می‌کنیم. به طور مشابه، اگر نابرابری به $T(n) \geq 2T\left(\frac{n}{2}\right) + \theta(n)$ معکوس شود، آنگاه، از آنجا که بازگشتی فقط یک حد پایین روی $T(n)$ می‌دهد، ما از نماد Ω در راه‌حل آن استفاده می‌کنیم.

مرتب سازی ادغام(Merge Sort)

- تقسیم: توالی n عنصری تقسیم می شود تا به ۲ توالی n/2 عنصری مرتب دست پیدا کنیم.
- مرتب سازی دو زیر توالی با استفاده از مرتب سازی ادغام
- ترکیب دو زیر توالی برای تولید یک لیست مرتب
- زمانی که توالی به طول ۱ ایجاد شود. کار خاتمه می یابد.

گام های الگوریتم مرتب سازی ادغام

- زیرآرایه `A[p:r]` را که قرار است مرتب شود، به دو زیرآرایه مجاور، هر کدام با نصف اندازه، تقسیم کنید. برای انجام این کار، نقطه میانی `q` از `A[p:r]` را محاسبه کنید (میانگین `p` و `r` را بگیرید) و `A[p:r]` را به زیرآرایه‌های `A[p:q]` و `A[q + 1:r]` تقسیم کنید.
- با مرتب‌سازی هر یک از دو زیرآرایه `A[p:q]` و `A[q + 1:r]` به صورت بازگشتی با استفاده از مرتب‌سازی ادغامی، به حل مسئله بپردازید.
- با ادغام دو زیرآرایه مرتب‌شده `A[p:q]` و `A[q + 1:r]` در `A[p:r]`، پاسخ مرتب‌شده را تولید کنید.

الگوریتم مرتب سازی ادغام

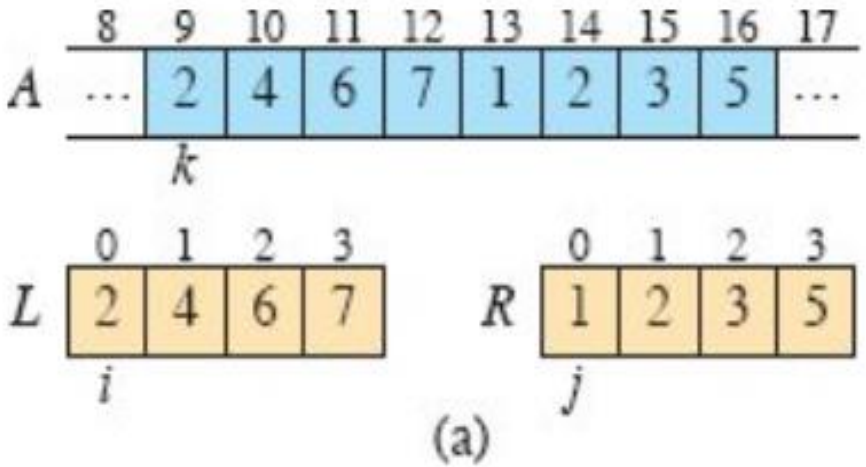
- عملیات کلیدی الگوریتم مرتب‌سازی ادغامی در مرحله «ترکیب» رخ می‌دهد که دو زیرآرایه مجاور و مرتب‌شده را با هم ادغام می‌کند. عملیات ادغام توسط رویه کمکی MERGE(A, p, q, r) انجام می شود، که در آن A یک آرایه است و p، qو r اندیس‌های آرایه هستند به طوری که p ≤ q < r. این رویه فرض می‌کند که زیرآرایه‌های مجاور A[p: q] و A[q + 1: r] قبلاً به صورت بازگشتی مرتب شده‌اند. این دو زیرآرایه مرتب‌شده را ادغام می‌کند تا یک زیرآرایه مرتب‌شده واحد تشکیل دهند که جایگزین زیرآرایه فعلی A[p: r] می‌شود.

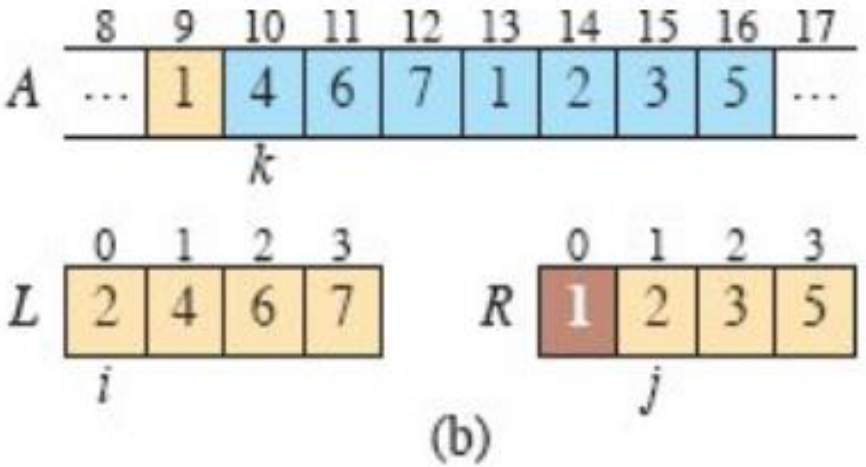
```
MERGE(A, p, q, r)
 1 n_L = q – p + 1        // length of A[p : q]
 2 n_R = r – q            // length of A[q + 1 : r]
 3 let L[0 : n_L – 1] and R[0 : n_R – 1] be new arrays
 4 for i = 0 to n_L – 1   // copy A[p : q] into L[0 : n_L – 1]
 5     L[i] = A[p + i]
 6 for j = 0 to n_R – 1   // copy A[q + 1 : r] into R[0 : n_R – 1]
 7     R[j] = A[q + j + 1]
 8 i = 0                  // i indexes the smallest remaining element in L
 9 j = 0                  // j indexes the smallest remaining element in R
10 k = p                  // k indexes the location in A to fill
11 // As long as each of the arrays L and R contains an unmerged
   element,
   //       copy the smallest unmerged element back into A[p : r].
```

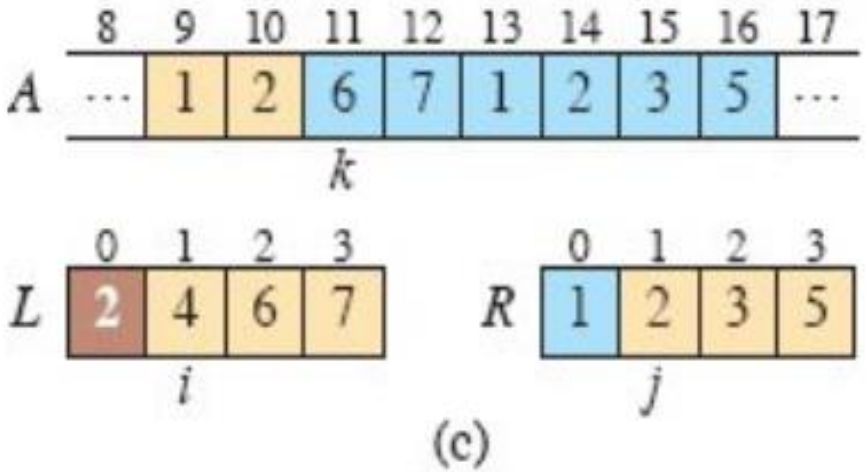
```
12 while i < n_L and j < n_R
13     if L[i] ≤ R[j]
14         A[k] = L[i]
15         i = i + 1
16     else A[k] = R[j]
17         j = j + 1
18     k = k + 1
19 // Having gone through one of L and R entirely, copy the
   //        remainder of the other to the end of A[p : r].
20 while i < n_L
21     A[k] = L[i]
22     i = i + 1
23     k = k + 1
24 while j < n_R
25     A[k] = R[j]
26     j = j + 1
27     k = k + 1
```

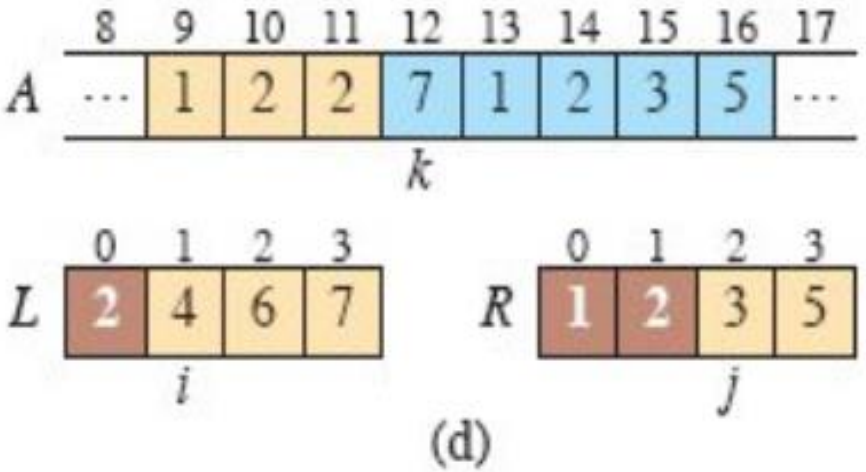
```
MERGE-SORT(A, p, r)
1  if p ≥ r                                  // zero or one element?
2      return
3  q = ⌊(p + r)/2⌋                           // midpoint of A[p : r]
4  MERGE-SORT(A, p, q)                       // recursively sort A[p : q]
5  MERGE-SORT(A, q + 1, r)                   // recursively sort A[q + 1 : r]
6  // Merge A[p : q] and A[q + 1 : r] into A[p : r].
7  MERGE(A, p, q, r)
```

مرتب سازی ادغام

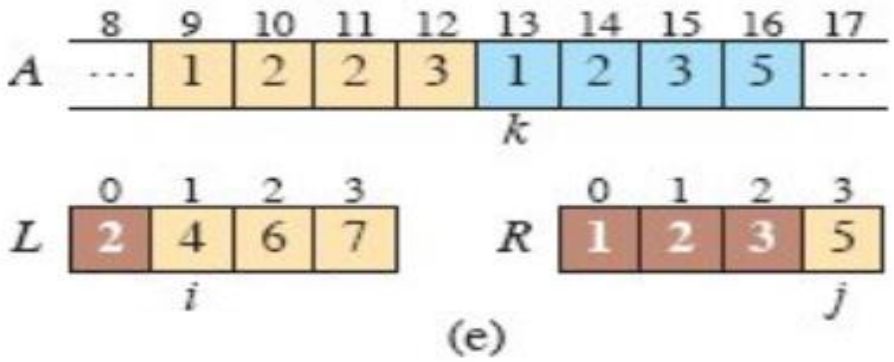




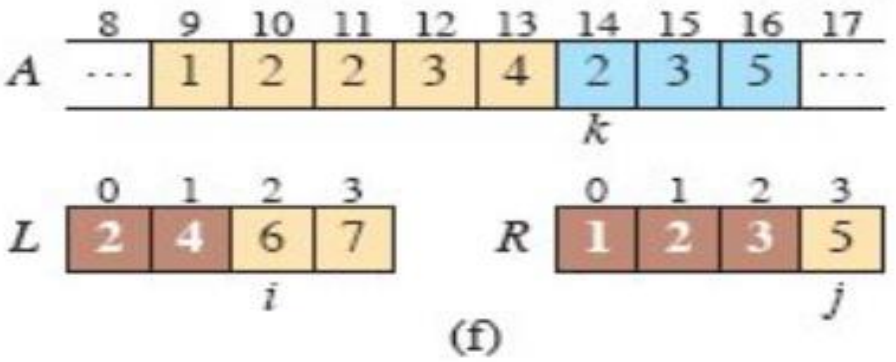




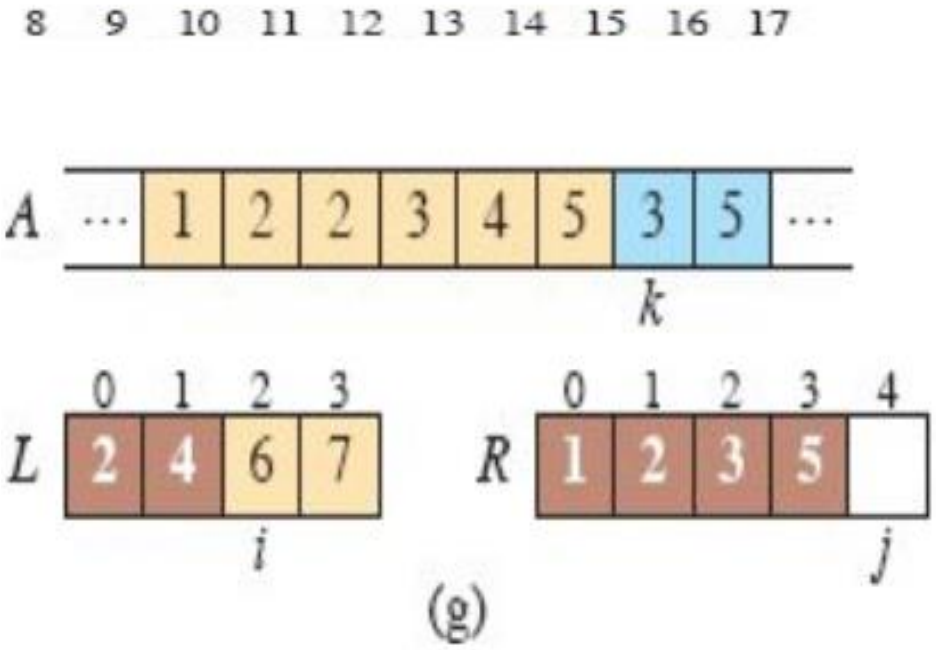
مرتب سازی ادغامی



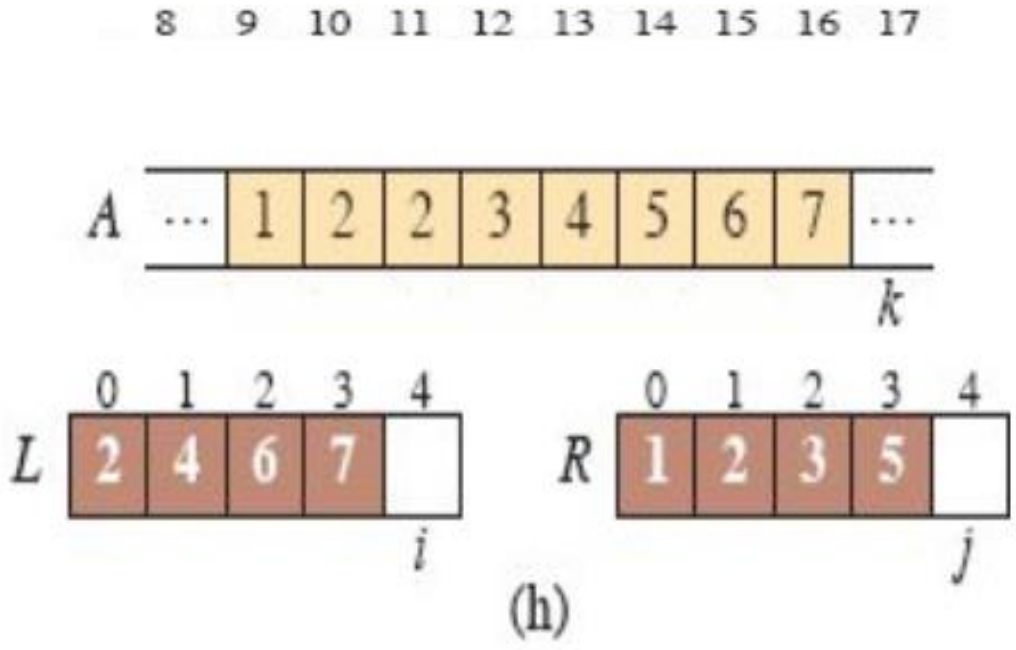
(e)



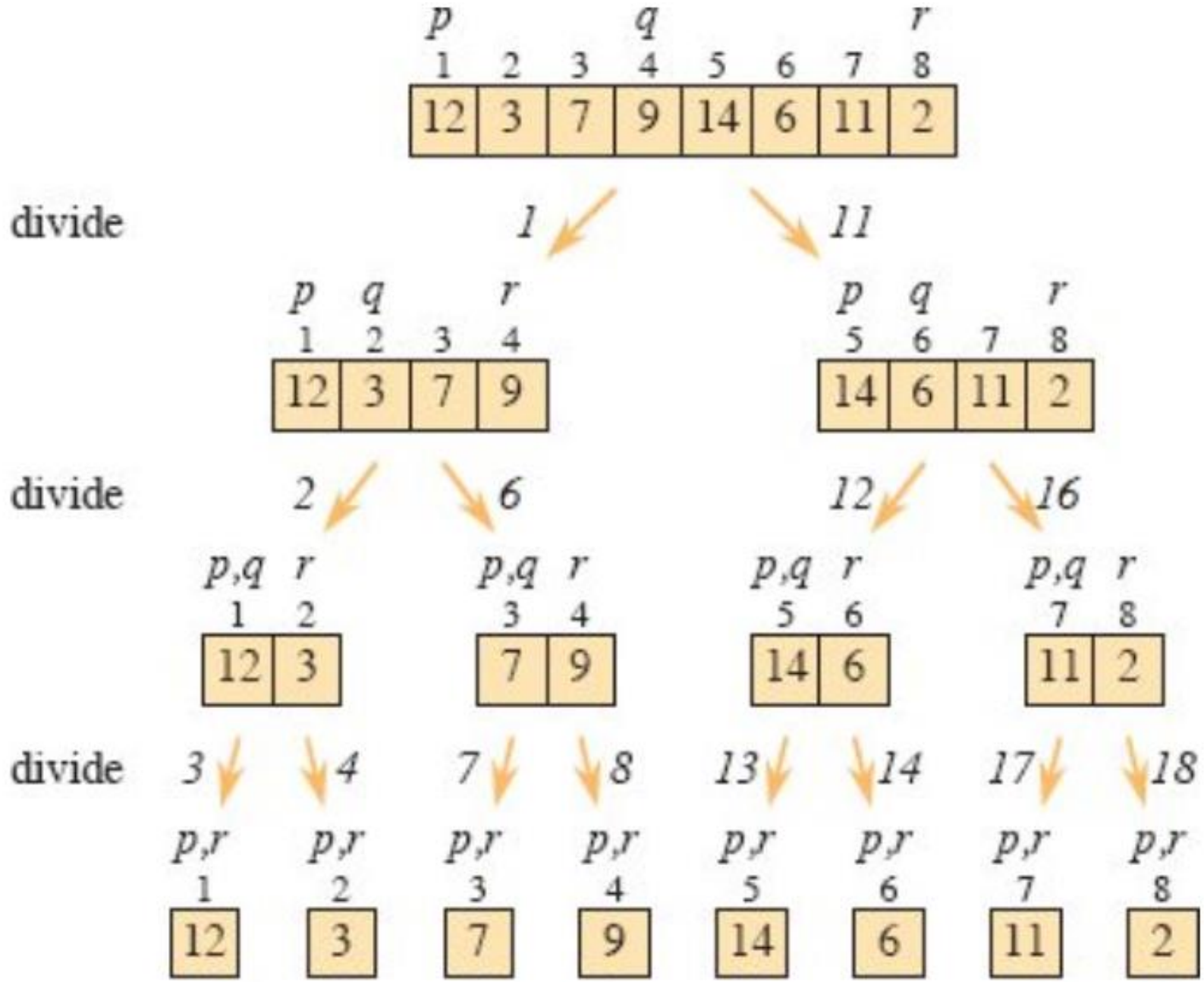
(f)

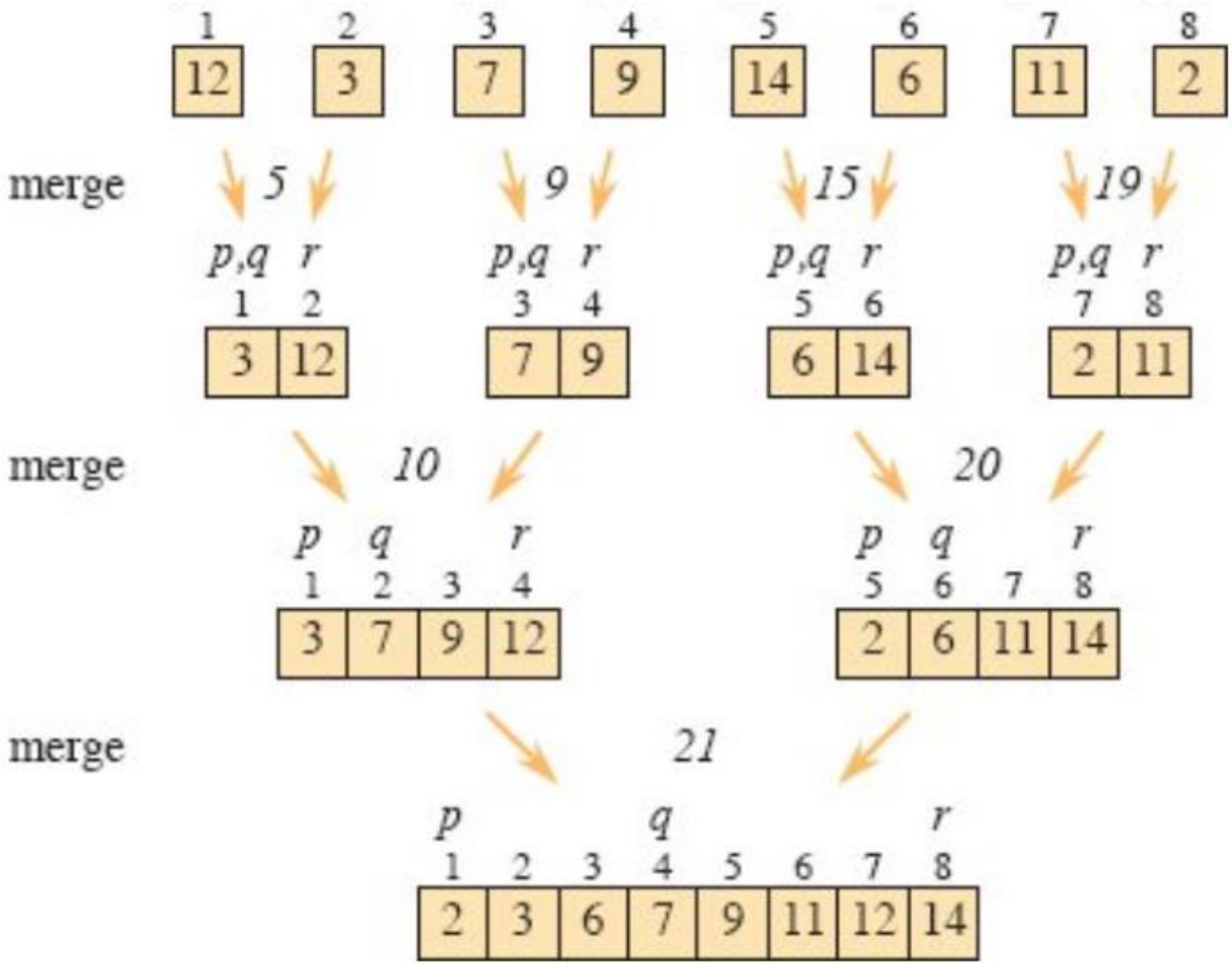


(g)



(h)


p
q
r
1 2 3 4 5 6 7 8
12 3 7 9 14 6 11 2
divide
1
11
p q r
1 2 3 4
12 3 7 9
p q r
5 6 7 8
14 6 11 2
divide
2
6
12
16
p,q r
1 2
12 3
p,q r
3 4
7 9
p,q r
5 6
14 6
p,q r
7 8
11 2
divide
3
4
7
8
13
14
17
18
p,r
1
12
p,r
2
3
p,r
3
7
p,r
4
9
p,r
5
14
p,r
6
6
p,r
7
11
p,r
8
2


1 2 3 4 5 6 7 8
12 3 7 9 14 6 11 2
merge
5
9
15
19
p,q r
1 2
3 12
p,q r
3 4
7 9
p,q r
5 6
6 14
p,q r
7 8
2 11
merge
10
20
p q r
1 2 3 4
3 7 9 12
p q r
5 6 7 8
2 6 11 14
merge
21
p q r
1 2 3 4 5 6 7 8
2 3 6 7 9 11 12 14

تحلیل الگوریتم های تقسیم و غلبه

- وقتی یک الگوریتم شامل یک فراخوانی بازگشتی است، اغلب می‌توانید زمان اجرای آن را با یک معادله بازگشتی یا تابع بازگشتی توصیف کنید، که زمان اجرای کلی روی یک مسئله با اندازه nرا بر حسب زمان اجرای همان الگوریتم روی ورودی‌های کوچکتر توصیف می‌کند. سپس می‌توانید از ابزارهای ریاضی برای حل تابع بازگشتی استفاده کنید و حدودی برای عملکرد و کارآیی الگوریتم ارائه دهید.

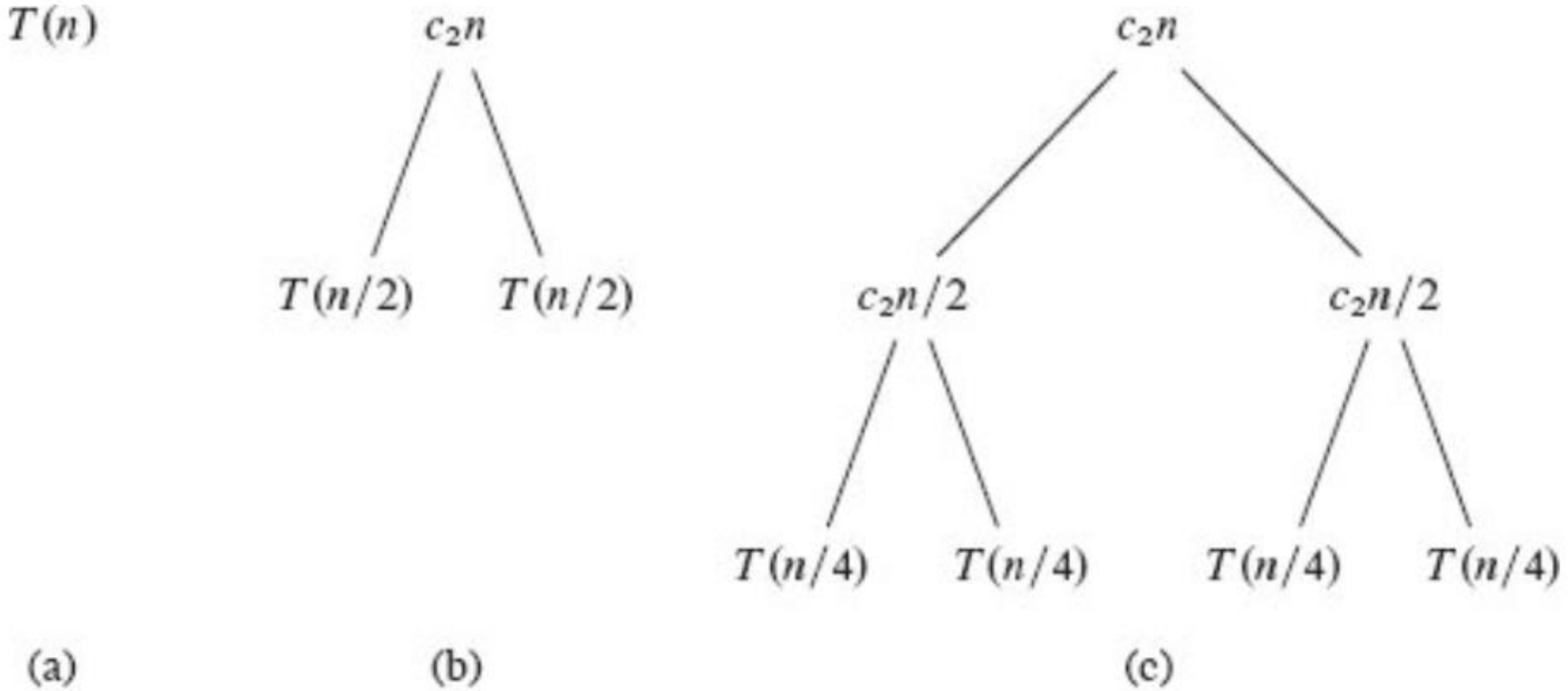
$$T(n) = \begin{cases} \Theta(1) & \text{if } n < n_0 , \\ D(n) + aT(n/b) + C(n) & \text{otherwise} . \end{cases}$$

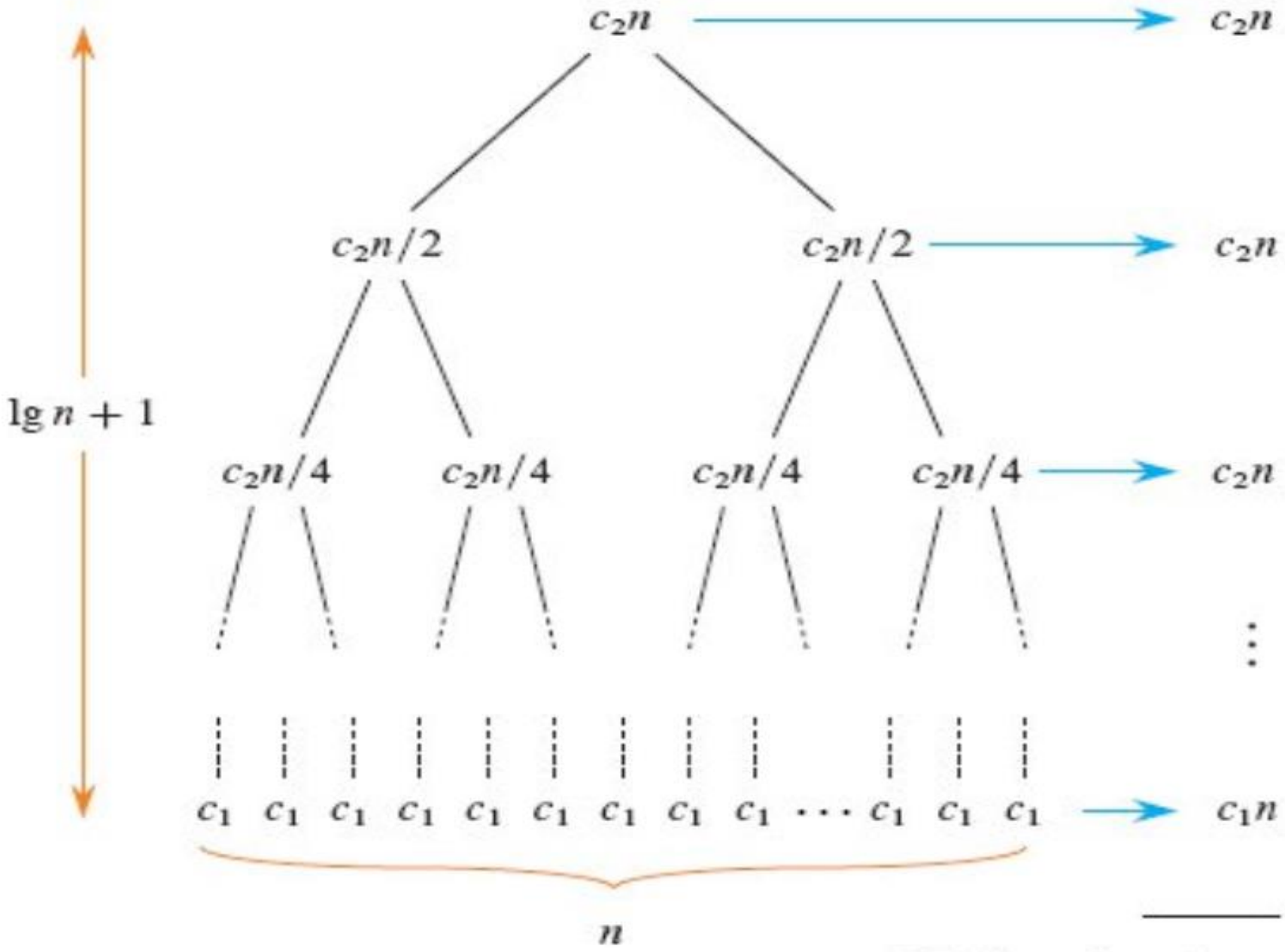
تحلیل الگوریتم مرتب سازی ادغام

- تقسیم: مرحله تقسیم فقط وسط زیرآرایه را محاسبه می‌کند که زمان ثابتی طول می‌کشد. بنابراین، $D(n) = \theta(n)$
- حل: حل بازگشتی دو زیرمسئله، هر کدام با اندازه n/2، 2T (n/2) به زمان اجرا کمک می‌کند (همانطور که بحث کردیم، کف‌ها و سقف‌ها را نادیده می‌گیریم).
- ترکیب: از آنجایی که روش MERGE روی یک زیرآرایه nعنصری زمان $\theta(n)$ طول می‌کشد، داریم $C(n) = \theta(n)$

$$T(n) = \begin{cases} c_1 & \text{if } n = 1 , \\ 2T(n/2) + c_2 n & \text{if } n > 1 , \end{cases}$$

تحلیل مرتب سازی ادغام-ادامه




c_2n
c_2n
$c_2n/2$
$c_2n/2$
c_2n
$\lg n + 1$
$c_2n/4$
$c_2n/4$
$c_2n/4$
$c_2n/4$
c_2n
c_1 c_1 c_1 c_1 c_1 c_1 c_1 c_1 c_1 $\cdots$ c_1 c_1 c_1
c_1n
n
Total: $c_2n \lg n + c_1n$

الگوریتم استراسن برای ضرب ماتریس ها

- در حالت معمول پیچیدگی زمانی الگوریتم ضرب دو ماتریس از مرتبه $\theta(n^3)$ است.
- روش معمول ضرب با استفاده از روش تقسیم و غلبه نیز دارای مرتبه اجرایی $\theta(n^3)$ است.
- استراسن با ارائه الگوریتم خود که مبتنی بر روش تفسیم و حل بود این پیچیدگی زمانی را به مرتبه $\theta(n^{2.81})$ کاهش داد و تا حدودی این الگوریتم را بهینه کرد.
- در روش معمول تقسیم و غلبه ۸ عمل ضرب برای ماتریس هایی با ابعاد $\frac{n}{2} \times \frac{n}{2}$ نیاز است.
- استراسن تعداد عملیات ضرب برای ماتریس هایی با ابعاد $\frac{n}{2} \times \frac{n}{2}$ را به ۷ ضرب کاهش داد.

الگوریتم استراسن برای ضرب ماتریس ها

- با توجه به توضیحات فوق تابع پیچیدگی زمانی روش استراسن به صورت زیر خواهد بود. $T(n) = 7T\left(\frac{n}{2}\right) + \theta(n^2)$

- در این گام جمع یا تفریق ۱۰ ماتریس $\frac{n}{2} \times \frac{n}{2}$ با استفاده از روابط زیر محاسبه می شود که زمانی حدود $\boldsymbol{\theta(n^2)}$ طول می کشد.

$$S_1 = B_{12} - B_{22}, \quad S_5 = A_{11} + A_{22},$$
$$S_2 = A_{11} + A_{12}, \quad S_6 = B_{11} + B_{22}, \quad S_9 = A_{11} - A_{21},$$
$$S_3 = A_{21} + A_{22}, \quad S_7 = A_{12} - A_{22}, \quad S_{10} = B_{11} + B_{12}.$$
$$S_4 = B_{21} - B_{11}, \quad S_8 = B_{21} + B_{22},$$

الگوریتم استراسن برای ضرب ماتریس ها

- در این مرحله به صورت بازگشتی ماتریس‌های $\frac{n}{2} \times \frac{n}{2}$ را ۷ بار در هم ضرب می‌کند تا ماتریس‌های $\frac{n}{2} \times \frac{n}{2}$ زیر را محاسبه کند که هر کدام از آنها مجموع یا تفاضل حاصلضرب زیرماتریس‌های A و Bاست:

$$P_1 = A_{11} \cdot S_1 \ (= A_{11} \cdot B_{12} - A_{11} \cdot B_{22}),$$
$$P_2 = S_2 \cdot B_{22} \ (= A_{11} \cdot B_{22} + A_{12} \cdot B_{22}),$$
$$P_3 = S_3 \cdot B_{11} \ (= A_{21} \cdot B_{11} + A_{22} \cdot B_{11}),$$
$$P_4 = A_{22} \cdot S_4 \ (= A_{22} \cdot B_{21} - A_{22} \cdot B_{11}),$$

الگوریتم استراسن برای ضرب ماتریس ها

$$P_5 = S_5 \cdot S_6 \quad (= A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22}),$$
$$P_6 = S_7 \cdot S_8 \quad (= A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22}),$$
$$P_7 = S_9 \cdot S_{10} \quad (= A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12}).$$

$$C_{11} = C_{11} + P_5 + P_4 - P_2 + P_6.$$

$$\begin{array}{l}
A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} \\
\qquad\qquad - A_{22} \cdot B_{11} \qquad\qquad + A_{22} \cdot B_{21} \\
\quad - A_{11} \cdot B_{22} \qquad\qquad\qquad\qquad - A_{12} \cdot B_{22} \\
\qquad\qquad\qquad - A_{22} \cdot B_{22} - A_{22} \cdot B_{21} + A_{12} \cdot B_{22} + A_{12} \cdot B_{21} \\
\hline
A_{11} \cdot B_{11} \qquad\qquad\qquad\qquad + A_{12} \cdot B_{21} \ ,
\end{array}$$

الگوریتم استراسن برای ضرب ماتریس ها

$$C_{12} = C_{12} + P_1 + P_2$$

$$A_{11} \cdot B_{12} - A_{11} \cdot B_{22} + A_{11} \cdot B_{22} + A_{12} \cdot B_{22}$$

$$A_{11} \cdot B_{12} + A_{12} \cdot B_{22} ,$$

$$C_{21} = C_{21} + P_3 + P_4$$

$$A_{21} \cdot B_{11} + A_{22} \cdot B_{11} - A_{22} \cdot B_{11} + A_{22} \cdot B_{21}$$

$$A_{21} \cdot B_{11} + A_{22} \cdot B_{21} ,$$

$$C_{22} = C_{22} + P_5 + P_1 - P_3 - P_7$$

$$A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} - A_{11} \cdot B_{22} - A_{22} \cdot B_{11} - A_{11} \cdot B_{11}$$

$$+ A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{11} \cdot B_{12} + A_{21} \cdot B_{11} + A_{21} \cdot B_{12}$$

$$A_{22} \cdot B_{22} + A_{21} \cdot B_{12} ,$$

روش های حل روابط بازگشتی تقسیم و غلبه

- **روش جایگزینی برای حل معادلات بازگشتی که شامل دو مرحله زیر است:**

1. با استفاده از ثابت‌های نمادین، شکل جواب را حدس بزنید.

2. با استفاده از استقرای ریاضی، نشان دهید که راه حل درست است و ثابت‌ها را بیابید.

- **روش درخت بازگشتی برای حل مسائل بازگشتی**

A. درخت بازگشتی بهترین استفاده را برای ایجاد شهود برای یک حدس خوب دارد، که می‌توانید آن را با روش جایگزینی تأیید کنید. با این حال، اگر هنگام ترسیم یک درخت بازگشتی و جمع کردن هزینه‌ها دقیق باشید، می‌توانید از یک درخت بازگشتی به عنوان اثبات مستقیم راه حل یک تکرار استفاده کنید.

قضیه اصلی برای حل مسائل بازگشتی

$$\left.\begin{aligned} T(n) &= aT\left(\frac{n}{b}\right) + cn^k \\ T(1) &= c \end{aligned}\right\} \Rightarrow \begin{cases} T(n) = \theta(n^{\log_b^a}) & \text{اگر } a > b^k \\ T(n) = \theta(n^k \log_2 n) & \text{اگر } a = b^k \\ T(n) = \theta(n^k) & \text{اگر } a < b^k \end{cases}$$

$$T(n) = aT\left(\frac{n}{b}\right) + cf(n) \Rightarrow \begin{cases} T(n) = O(n^{\log_b^a}) & \text{اگر } n^{\log_b^a} > O(f(n)) \\ T(n) = O(f(n)) & \text{اگر } n^{\log_b^a} < O(f(n)) \\ T(n) = \log n \times O(f(n)) & \text{اگر } a = b \end{cases}$$

بررسی چند نمونه در خصوص قضیه اصلی

- اگر $T(n) = 3T\left(\frac{n}{3}\right) + n^2$ و $T(1) = 1$ باشد پاسخ رابطه بازگشتی فوق به صورت زیر خواهد بود.

در رابطه فوق چون k=2، a=3، b=3 است در نتیجه بر اساس شرایط ذکر شده $a < b^k$ بوده و پاسخ به صورت رابطه مقابل خواهد بود.$\theta(n^k) = \theta(n^2)$

- اگر $T(n) = 2T\left(\frac{n}{2}\right) + 1$ و $T(1) = 0$ باشد پاسخ رابطه بازگشتی فوق به صورت زیر خواهد بود.

در رابطه فوق چون k=0، a=2، b=2 است در نتیجه بر اساس شرایط ذکر شده $a > b^k$ بوده و پاسخ به صورت رابطه مقابل خواهد بود.$T(n) = \theta(n^{\log_b a}) = \theta(n^{\log_2 2}) = \theta(n)$

- اگر $T(n) = 2T\left(\frac{n}{2}\right) + n$ باشد، در رابطه فوق داریم: k = 1 و a = 2 و b = 2 در نتیجه $a = b^k$

بنابراین پاسخ رابطه فوق به صورت مقابل است: $T(n) = \theta(n^k \log n) = \theta\,(n \log n)$

بررسی چند نمونه دیگر در خصوص قضیه اصلی

$$T(n) = 4T(\frac{n}{2}) + \log n$$

$$n^{\log_b^a} = n^{\log_2^4} = n^2 \quad \text{و} \quad O(f(n)) = \log n \Rightarrow n^2 > \log n$$

$$T(n) = O(n^2)$$

$$T(n) = 3T(\frac{n}{4}) + n \log n$$

$$n^{\log_b^a} = n^{\log_4^3} = n^{0.79} \quad , \quad O(f(n)) = n \log n \Rightarrow T(n) = O(n \log n)$$

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

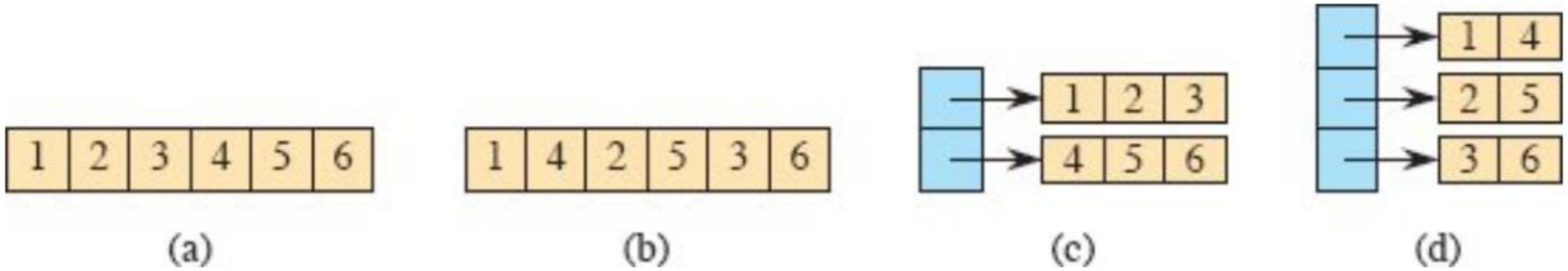
$$a = b = 2 \Rightarrow T(n) = \log n \times O(f(n)) = \log n . n \log n \Rightarrow T(n) = n \log^2 n$$

ساختارهای داده مقدماتی

- آرایه ها
- ماتریس ها
- پشته
- صف
- لیست پیوندی

آرایه ها و ماتریس ها

- ما فرض می‌کنیم که، مانند اکثر زبان‌های برنامه‌نویسی، یک آرایه به صورت یک توالی پیوسته از بایت‌ها در حافظه ذخیره می‌شود. اگر اولین عنصر یک آرایه دارای اندیس sباشد (برای مثال، در آرایه‌ای با اندیس‌گذاری از مبدا ۱، s = 1)، آرایه از آدرس حافظه aشروع می‌شود و هر عنصر آرایه b بایت را اشغال می‌کند، سپس عنصر iام بایت‌های $a + b(i - s)$ تا $a + b(i - s + 1) - 1$ را اشغال می‌کند.

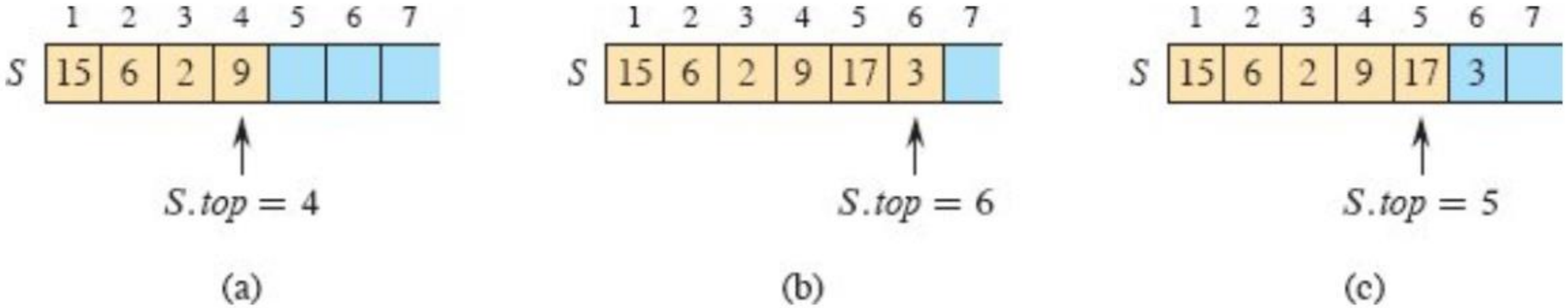


پشته و صف

- پشته‌ها و صف‌ها مجموعه‌های پویایی هستند که در آن‌ها عنصری که توسط عملیات DELETE از مجموعه حذف می‌شود، از پیش مشخص شده است. در یک پشته، عنصری که از مجموعه حذف می‌شود، عنصری است که اخیراً وارد شده است: پشته سیاست آخرین ورودی، اولین خروجی یا LIFO را پیاده‌سازی می‌کند. به طور مشابه، در یک صف، عنصری که حذف می‌شود همیشه عنصری است که بیشترین زمان را در مجموعه بوده است: صف سیاست اولین ورودی، اولین خروجی یا FIFO را پیاده‌سازی می‌کند. چندین روش کارآمد برای پیاده‌سازی پشته‌ها و صف‌ها در رایانه وجود دارد. در اینجا، نحوه استفاده از یک آرایه با ویژگی‌ها برای ذخیره آن‌ها را خواهید دید.

پشته ها

- عملیات درج ((INSERTروی یک پشته اغلب PUSH نامیده می‌شود، و عملیات حذف (DELETE) که آرگومانی برای عنصر دریافت نمی‌کند، اغلب POP نامیده می‌شود. این نام‌ها اشاره‌ای به پشته‌های فیزیکی دارند، مانند پشته‌های فنری بشقاب‌های مورد استفاده در کافه‌تریاها. ترتیب بیرون آوردن بشقاب‌ها از پشته، عکس ترتیب قرار دادن آنها روی پشته است، زیرا فقط صفحه بالایی قابل دسترسی است.



عملیات روی پشته ها

```
STACK-EMPTY(S)

1 if S.top == 0
2       return TRUE
3 else return FALSE
```

```
PUSH(S, x)

1 if S.top == S.size
2       error "overflow"
3 else S.top = S.top + 1
4       S[S.top] = x
```

```
POP(S)

1 if STACK-EMPTY(S)
2       error "underflow"
3 else S.top = S.top – 1
4       return S[S.top + 1]
```

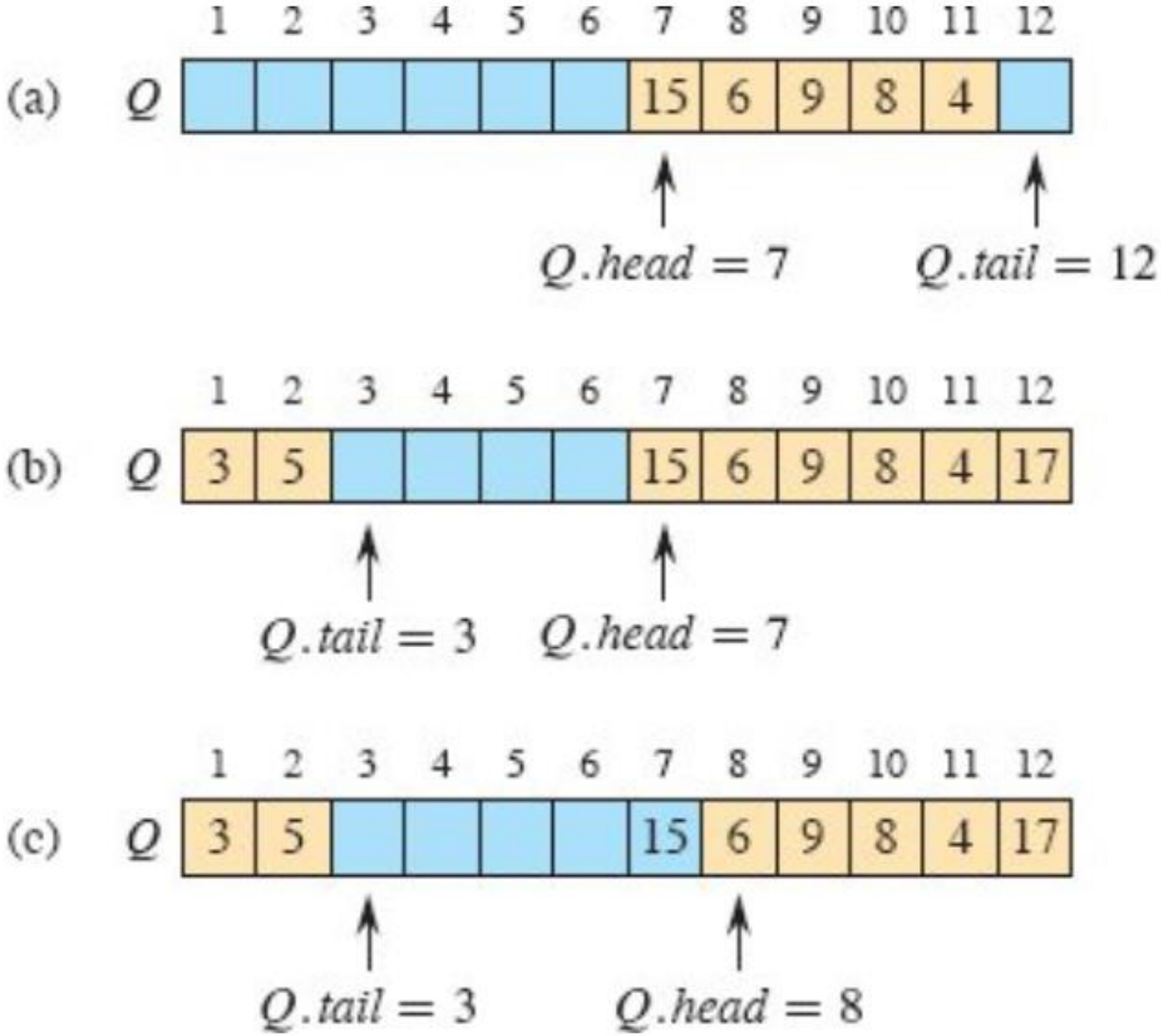
صف ها

- ما عملیات INSERT را روی یک صف ENQUEUE و عملیات DELETE را DEQUEUE می‌نامیم. مانند عملیات پشته POP، DEQUEUE هیچ آرگومانی برای عنصر دریافت نمی‌کند. خاصیت FIFO یک صف باعث می‌شود که مانند صفی از مشتریان منتظر سرویس عمل کند. صف دارای یک سر و یک دم است. وقتی یک عنصر در صف قرار می‌گیرد، در انتهای صف قرار می‌گیرد، همانطور که یک مشتری تازه وارد در انتهای صف قرار می‌گیرد. عنصر خارج شده از صف همیشه عنصری است که در سر صف قرار دارد، مانند مشتری در سر صف که بیشترین زمان انتظار را داشته است.

صف ها

- صف دارای ویژگی Q.head است که به ابتدای صف اشاره می‌کند یا آن را شاخص‌گذاری می‌کند. ویژگی Q.tail مکان بعدی را که یک عنصر تازه وارد شده در صف قرار می‌گیرد، شاخص‌گذاری می‌کند.
- عناصر موجود در صف در مکان‌های $Q.head, Q.head + 1 \ldots, Q.tail - 1$ قرار دارند، که در آن ما «به دور هم می‌چرخیم» به این معنا که مکان ۱ بلافاصله پس از مکان n به صورت دایره‌ای قرار می‌گیرد. وقتی $Q.head = Q.tail$، صف خالی است. در ابتدا، $Q.head = Q.tail = 1$ داریم. تلاش برای خارج کردن یک عنصر از صف خالی باعث سرریز شدن صف می‌شود. وقتی Q.head = Q.tail + 1 یا هر دو Q.head = 1 و Q.tail = Q.size باشد، صف پر است و تلاش برای قرار دادن یک عنصر در صف باعث سرریز شدن صف می‌شود.

صف ها


(a)
Q
1 2 3 4 5 6 7 8 9 10 11 12
15 6 9 8 4
Q.head = 7
Q.tail = 12
(b)
Q
1 2 3 4 5 6 7 8 9 10 11 12
3 5 15 6 9 8 4 17
Q.tail = 3
Q.head = 7
(c)
Q
1 2 3 4 5 6 7 8 9 10 11 12
3 5 15 6 9 8 4 17
Q.tail = 3
Q.head = 8

عملیات روی صف ها

```
ENQUEUE(Q, x)

1 Q[Q.tail] = x
2 if Q.tail == Q.size
3       Q.tail = 1
4 else Q.tail = Q.tail + 1

DEQUEUE(Q)

1 x = Q[Q.head]
2 if Q.head == Q.size
3       Q.head = 1
4 else Q.head = Q.head + 1
5 return x
```

لیست های پیوندی

- لیست پیوندی یک ساختار داده است که در آن اشیاء به ترتیب خطی مرتب شده‌اند. با این حال، برخلاف آرایه که در آن ترتیب خطی توسط اندیس‌های آرایه تعیین می‌شود، ترتیب در یک لیست پیوندی توسط یک اشاره‌گر در هر شیء تعیین می‌شود. از آنجایی که عناصر لیست‌های پیوندی اغلب حاوی کلیدهایی هستند که می‌توان آنها را جستجو کرد، لیست‌های پیوندی گاهی اوقات لیست‌های جستجو نامیده می‌شوند. لیست‌های پیوندی یک نمایش ساده و انعطاف‌پذیر برای مجموعه‌های پویا ارائه می‌دهند.

لیست پیوندی

- یک لیست ممکن است یکی از چندین شکل را داشته باشد. می‌تواند تک پیوندی یا دو پیوندی باشد، می‌تواند مرتب باشد یا نباشد، و می‌تواند دایره‌ای باشد یا نباشد. اگر یک لیست تک پیوندی باشد، هر عنصر یک اشاره‌گر بعدی دارد اما یک اشاره‌گر قبلی ندارد. اگر یک لیست مرتب باشد، ترتیب خطی لیست مطابق با ترتیب خطی کلیدهای ذخیره شده در عناصر لیست است. در این صورت عنصر حداقلی، سر لیست و عنصر حداکثری، دم لیست است. اگر لیست مرتب نشده باشد، عناصر می‌توانند به هر ترتیبی ظاهر شوند. در یک لیست دایره‌ای، اشاره‌گر قبلی سر لیست به دم و اشاره‌گر بعدی دم لیست به سر لیست اشاره می‌کند. می‌توانید یک لیست دایره‌ای را به عنوان حلقه‌ای از عناصر در نظر بگیرید. در ادامه این بخش، فرض می‌کنیم که لیست‌هایی که با آنها کار می‌کنیم، نامرتب و دو پیوندی هستند.

جستجوی یک لیست دوپیوندی

- رویه LIST-SEARCH(L, k) اولین عنصر با کلید k را در لیست L با یک جستجوی خطی ساده پیدا می‌کند و یک اشاره‌گر به این عنصر برمی‌گرداند. اگر هیچ شیء با کلید k در لیست ظاهر نشود، رویه NIL را برمی‌گرداند. برای لیست پیوندی در شکل ۱۰.٤(a)، فراخوانی LIST-SEARCH(L, 4) یک اشاره‌گر به عنصر سوم برمی‌گرداند و فراخوانی LIST-SEARCH(L, 7)، NIL را برمی‌گرداند. برای جستجوی لیستی از n شیء، رویه LIST-SEARCH در بدترین حالت $\theta(n)$ زمان می‌برد، زیرا ممکن است مجبور شود کل لیست را جستجو کند.

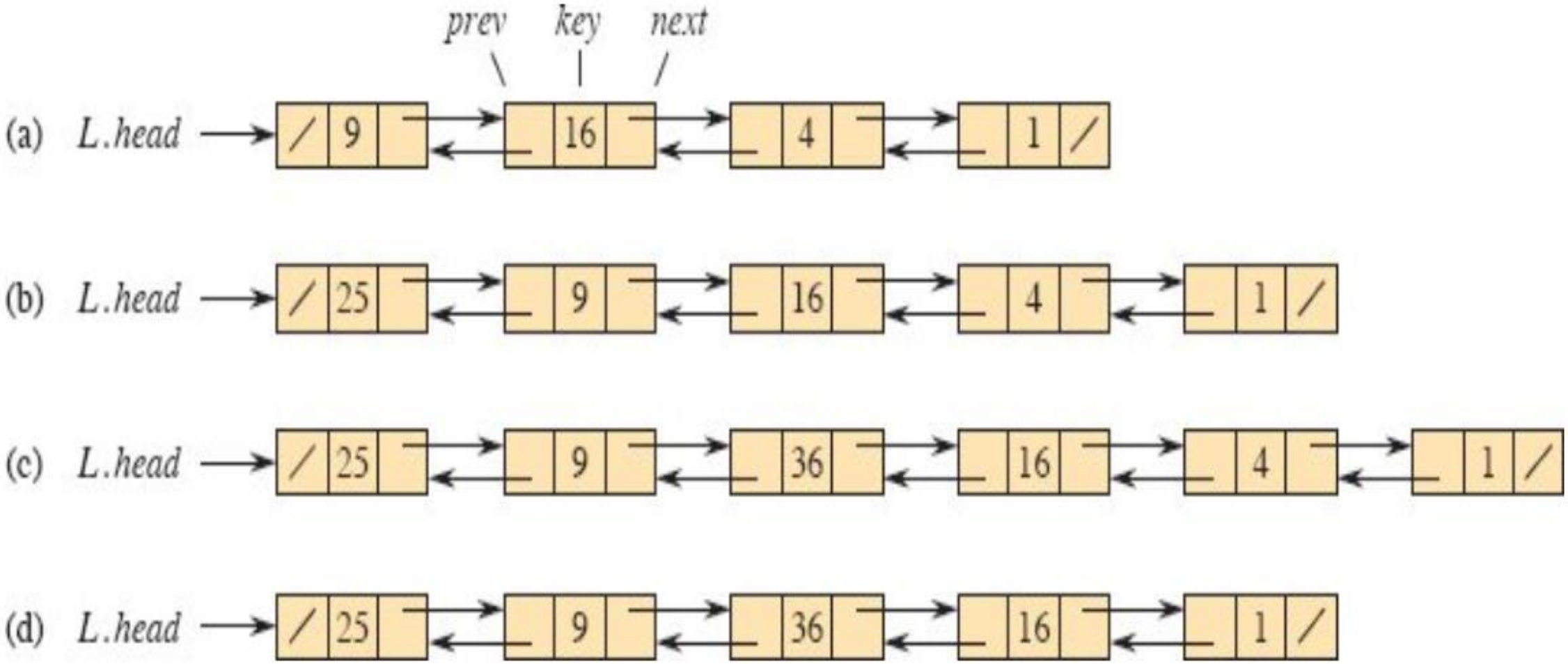
درج در لیست دوپیوندی

- با توجه به عنصر x که ویژگی کلید آن قبلاً تنظیم شده است، روال LISTPREPEND، x را به ابتدای لیست پیوندی اضافه می‌کند، همانطور که در شکل ۱۰.٤(b) نشان داده شده است. (به یاد داشته باشید که نمادگذاری ویژگی ما می‌تواند به صورت آبشاری باشد، به طوری که L.head.prev نشان‌دهنده ویژگی prev شیء است که L.head به آن اشاره می‌کند.) زمان اجرای LIST-PREPEND روی لیستی از n عنصر O(1) است.
- می‌توانید در هر جایی از یک لیست پیوندی، مقداری را وارد کنید. همانطور که شکل ۱۰.٤(c) نشان می‌دهد، اگر اشاره‌گری به نام y در یک شیء در لیست داشته باشید، روال LIST-INSERT، عنصر جدید x را بلافاصله پس از y و در زمان O(1) به لیست «پیوند» می‌دهد. از آنجایی که LIST-INSERT هرگز به شیء لیست L ارجاع نمی دهد، به عنوان پارامتر ارائه نمی‌شود.

حذف از لیست دوپیوندی

- روال LIST-DELETE عنصر x را از لیست پیوندی L حذف می‌کند. باید به آن یک اشاره‌گر به x داده شود و سپس با به‌روزرسانی اشاره‌گرها، x را از لیست «پیوند» می‌دهد. برای حذف یک عنصر با کلید مشخص، ابتدا LIST-SEARCH را برای بازیابی یک اشاره‌گر به عنصر فراخوانی کنید. شکل ۱۰.٤(d) نشان می‌دهد که چگونه یک عنصر از لیست پیوندی حذف می‌شود. LIST-DELETE در زمان O(1) اجرا می‌شود، اما برای حذف یک عنصر با کلید مشخص، فراخوانی LISTSEARCH باعث می‌شود که بدترین حالت زمان اجرا $\theta(n)$ باشد.

لیست پیوندی دو طرفه(لیست دو پیوندی)



شبه کدهای عملیات های جستجو، درج و حذف

```
LIST-SEARCH(L, k)
1 x = L.head
2 while x ≠ NIL and x.key ≠ k
3       x = x.next
4 return x
```

```
LIST-DELETE(L, x)
1 if x.prev ≠ NIL
2       x.prev.next = x.next
3 else L.head = x.next
4 if x.next ≠ NIL
5       x.next.prev = x.prev
```

```
LIST-PREPEND(L, x)
1 x.next = L.head
2 x.prev = NIL
3 if L.head ≠ NIL
4       L.head.prev = x
5 L.head = x
```

```
LIST-INSERT(x, y)
1 x.next = y.next
2 x.prev = y
3 if y.next ≠ NIL
4       y.next.prev = x
5 y.next = x
```

مقایسه زمان درج و حذف آرایه و لیست پیوندی

- عملیات درج و حذف در لیست‌های پیوندی دوگانه سریع‌تر از آرایه‌ها هستند. اگر می‌خواهید یک عنصر اول جدید را در یک آرایه وارد کنید یا عنصر اول را در یک آرایه حذف کنید، با حفظ ترتیب نسبی همه عناصر موجود، هر یک از عناصر موجود باید یک موقعیت جابجا شوند. بنابراین، در بدترین حالت، درج و حذف در یک آرایه $\theta(n)$ زمان می‌برد، در حالی که برای یک لیست پیوندی دوگانه O(1) زمان می‌برد. با این حال، اگر می‌خواهید عنصر kام را به ترتیب خطی پیدا کنید، صرف نظر از k، در یک آرایه فقط O(1) زمان می‌برد، اما در یک لیست پیوندی، باید k عنصر را پیمایش کنید و زمان $\theta(k)$ صرف کنید.

درخت

- لیست‌های پیوندی برای نمایش روابط خطی به خوبی کار می‌کنند، اما همه روابط خطی نیستند.
- ما هر گره از یک درخت را با یک شیء نمایش می‌دهیم. همانند لیست‌های پیوندی، فرض می‌کنیم که هر گره حاوی یک ویژگی کلیدی است. ویژگی‌های باقی‌مانده مورد توجه، اشاره‌گرهایی به گره‌های دیگر هستند و بسته به نوع درخت متفاوتند.
- ساختار درختی یعنی مجموعه داده های سازماندهی شده ای که عناصر اطلاعاتی شان به وسیله انشعاباتی با هم رابطه داشته باشند.

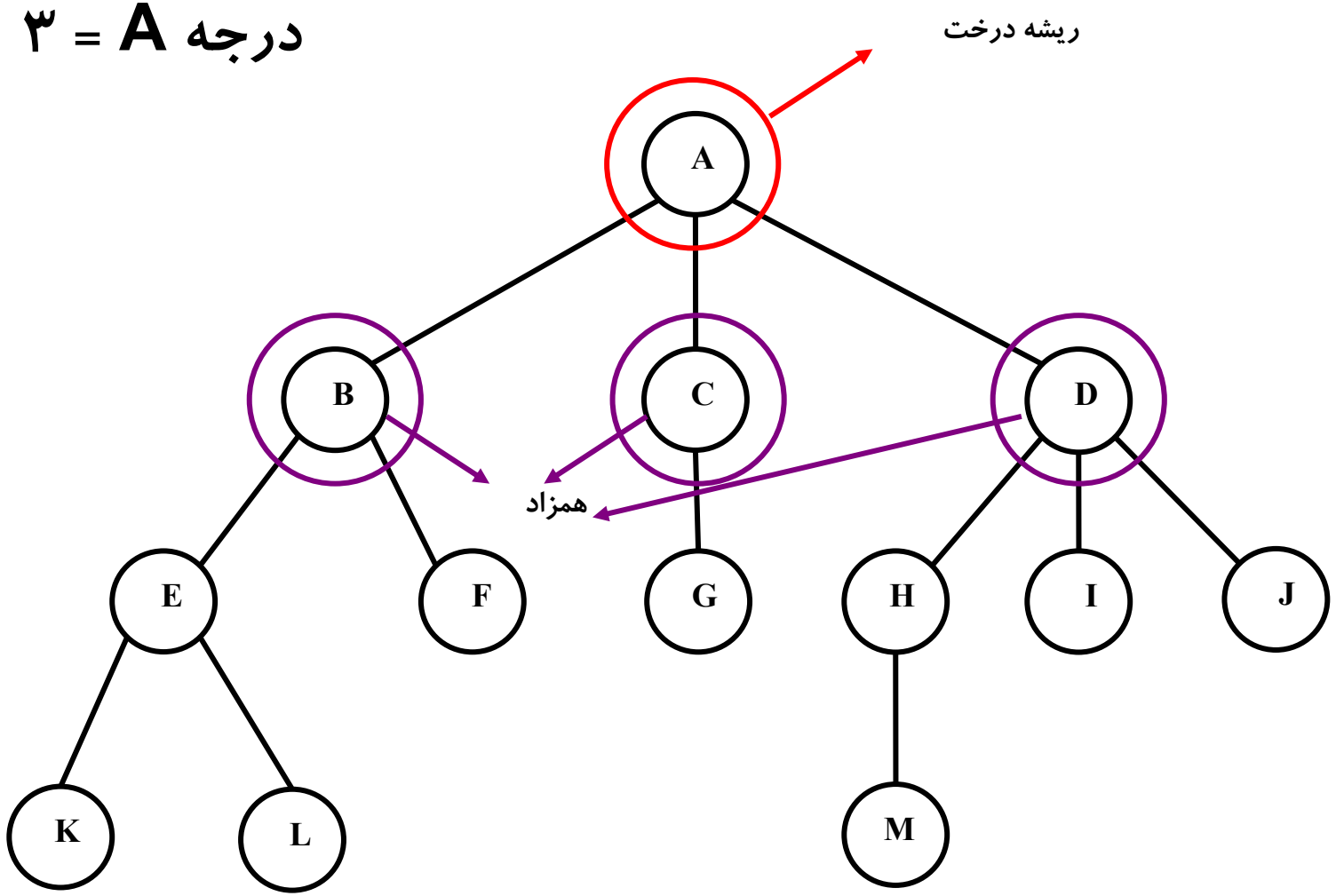
ساختار درخت

درخت مجموعه محدودی از یک یا چند گره به صورت زیر می باشد :

- **دارای گره خاصی به نام ریشه باشد.**
- **بقیه گره ها به $n \geq 0$ مجموعه مجزا $T_1,....T_n$ تقسیم شده که هر یک از این مجموعه ها خود یک درخت هستند. $T_1,....T_n$ زیر درختان ریشه نامیده می شوند.**

نمونه ای از یک درخت

درجه A = ۳



A والد B، C، D است . B، C، D همزادند.

اصطلاح هایی در مورد درخت ها

گره: گره به عنصر حاوی اطلاعات و انشعابات به دیگر عناصر، اطلاق می شود.

درجه گره: تعداد زیر درخت های یک گره را درجه آن گره می نامند.

والد: گره ای که دارای زیر درختانی است را والد ریشه های زیر درختان گویند.

فرزندان گره: ریشه های زیر درختان، فرزندان آن گره نامیده می شوند.

گره های همزاد: فرزندان یک گره، گره های همزاد یا هم نیا نامیده می شوند.

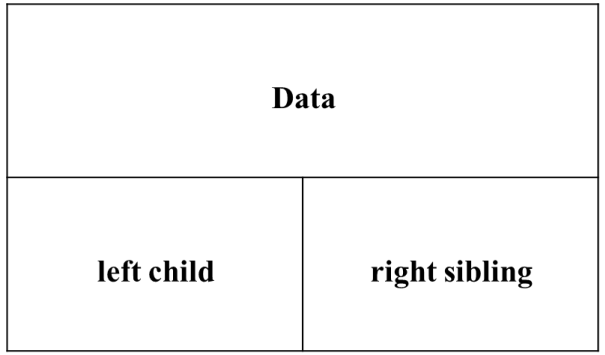
اجداد گره: اجداد یک گره، گره هایی هستند که در مسیر طی شده از ریشه تا آن گره وجود دارند.

سطح گره: سطح یک گره بدین صورت تعریف می شود که ریشه در سطح یک قرار می گیرد. برای تمامی گره های بعدی ، سطح گره برابر است با سطح والد به اضافه یک .

ارتفاع درخت: ارتفاع یا عمق یک درخت به بیشترین سطح گره های آن درخت گفته می شود.

درخت های دودویی

- یک درخت دودویی یا تهی است یا حاوی مجموعه ای محدود از گره ها شامل یک ریشه و دو زیر درخت دودویی است. این درخت ها زیر درخت های چپ و راست نامیده می شوند.
- مشخصه اصلی یک درخت دودویی بدین شکل است که هر گره آن حداکثر دو انشعاب دارد یعنی گره هایی که درجه ای بیشتر از دو نداشته باشند.
- برای درخت های دودویی زیر درخت سمت چپ و راست با یکدیگر متمایز است.
- برای نمایش درختان دودویی، دقیقا نیاز به دو اتصال یا اشاره گر به ازای هر گره است.



درخت های دودویی

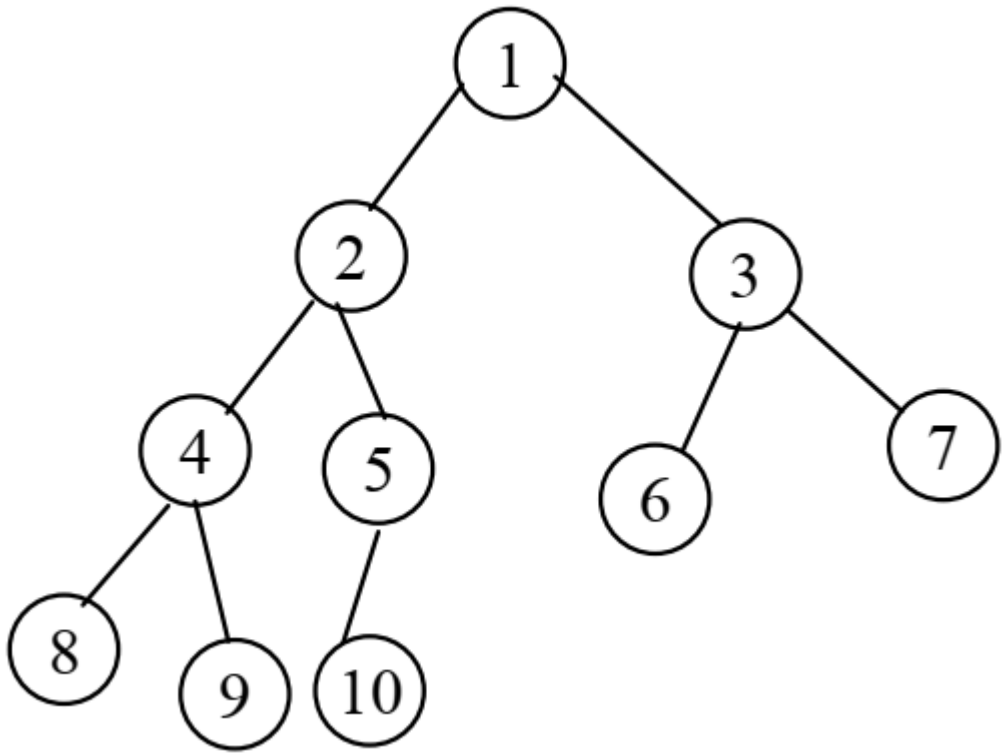
- در هیچ درخت عادی صفر گره وجود ندارد، اما درخت دودویی تهی وجود دارد.
- در یک درخت دودویی ترتیب فرزندان دارای اهمیت بوده در حالی که در درخت عادی به این صورت نیست.

- حداکثر تعداد گره ها در سطح iام یک درخت دودویی 2^{i-1} است ، $i \geq 1$.
- حداکثر تعداد گره ها در یک درخت دودویی به ارتفاع h ، $2^h - 1$ است ، $h \geq 1$
- برای هر درخت دودویی غیر تهی مانند T، اگر n_0 تعداد گره های پایانی و n_2 تعداد گره های درجه ۲ باشد، آنگاه خواهیم داشت:

$$n_0 = n_2 + 1$$

درخت دودویی کامل

- درختی که تمام سطح های ان به جز آخرین سطح(احتمالا)، حداکثر گره های ممکن را داشته باشند. ضمنا گره های آخرین سطح نیز از سمت چپ قرار گرفته باشند.



خواص درخت دودویی کامل

- در یک درخت دودویی تعریف شده با n گره عمق D_n را می توان از رابطه زیر بدست آورد:

$$D_n = \lfloor log_2 n + 1 \rfloor$$

- برای هر گره با اندیس I و $1 \leq i \leq n$ داریم:

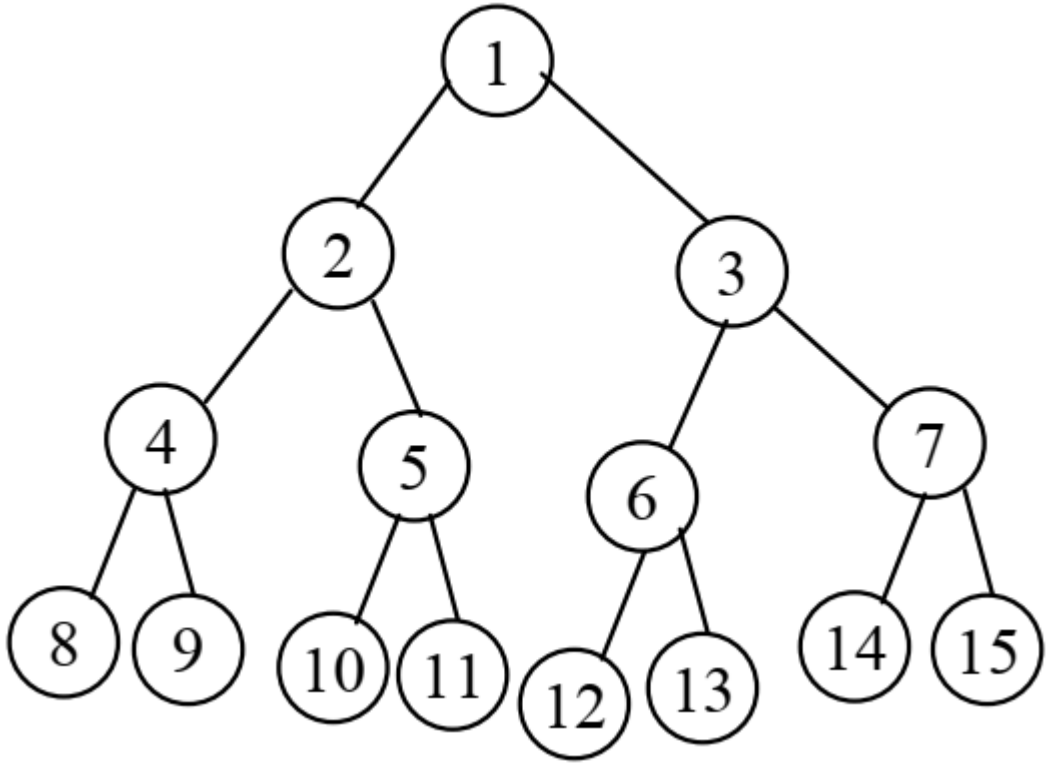
- اگر $i \neq 1$ آن گاه پدر i در $\left\lfloor \frac{i}{2} \right\rfloor$ است. اگر $i = 1$ ، i ریشه است و فرزندی نخواهد داشت.

- اگر $2i \leq n$ آن گاه فرزند چپ i در 2i است. اگر $2i > n$ آن گاه i فرزند چپ ندارد.

- اگر $2i + 1 \leq n$ آن گاه فرزند راست i در 2i+! است. اگر $2i + 1 > n$ آن گاه i فرزند راست ندارد.

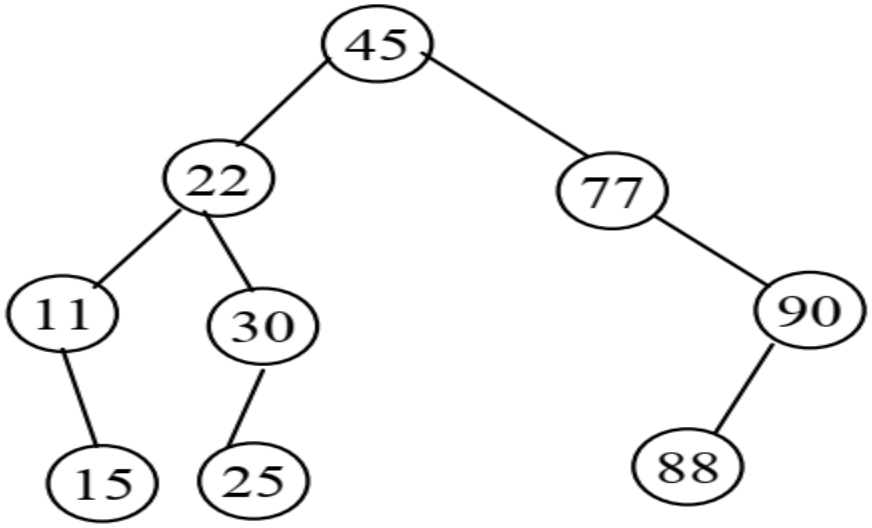
درخت دودویی پر

- درخت دودویی T را پر می گویند، هرگاه همه گره های آن به جز گره های آخرین سطح دقیقا دو فرزند داشته باشد.



نمایش درخت های دودویی در حافظه

- نمایش ترتیبی با استفاده از آرایه: ای نوع نمایش برای درخت های کامل یا نسبتا کامل مناسب است و برای سایر درختان به دلیل هدر رفت فضای اخذ شده از حافظه توصیه نمی شود.

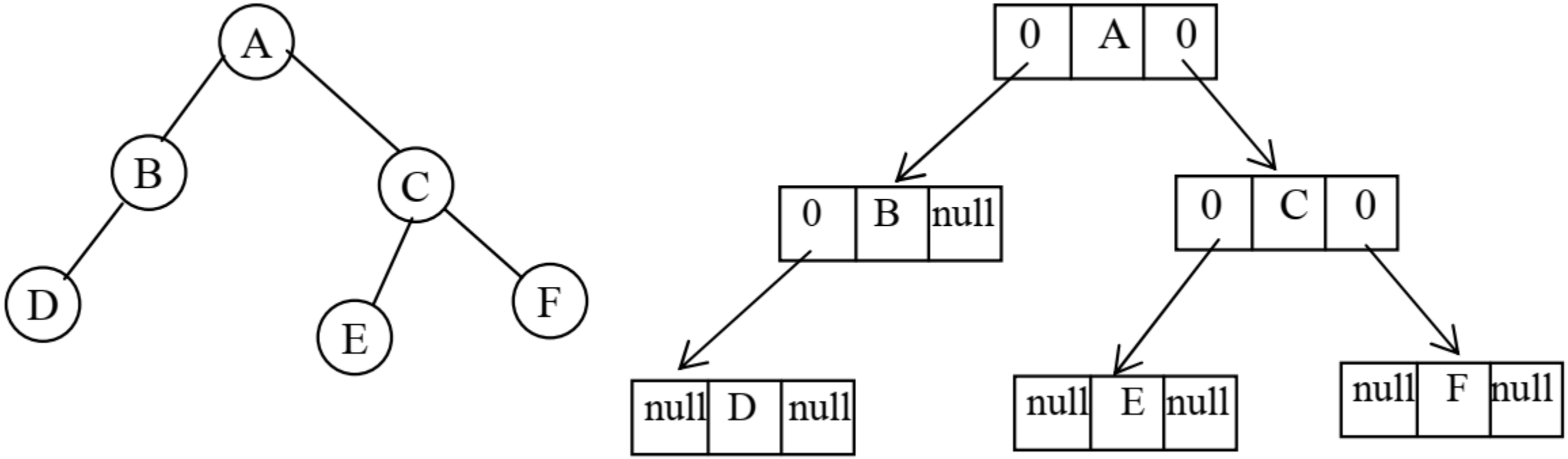


١	٢	٣	٤	٥	٦	٧	٨	٩	١٠	١١	١٢	١٣	١٤	١٥
٤٥	٢٢	١٧	١١	٣٠٠		٩٠		١٥	٢٥				٨٨	

نمایش پیوندی درخت های دودویی

- در این نوع نمایش هرگره در درخت دودویی ساختاری به صورت زیر دارد.

Lchild	Data	Rchild



پیمایش درخت های دودویی

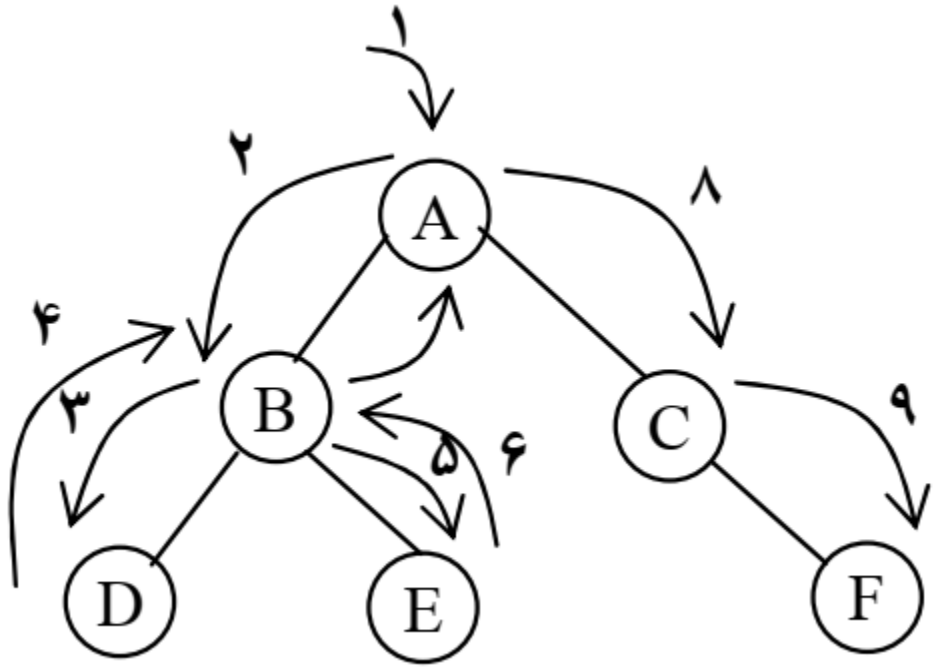
- پیش ترتیب، Preorder، VLR
- پس ترتیب، Postorder، LRV
- میان ترتیب، Inorder، LVR

پیمایش پیش ترتیب و میان ترتیب

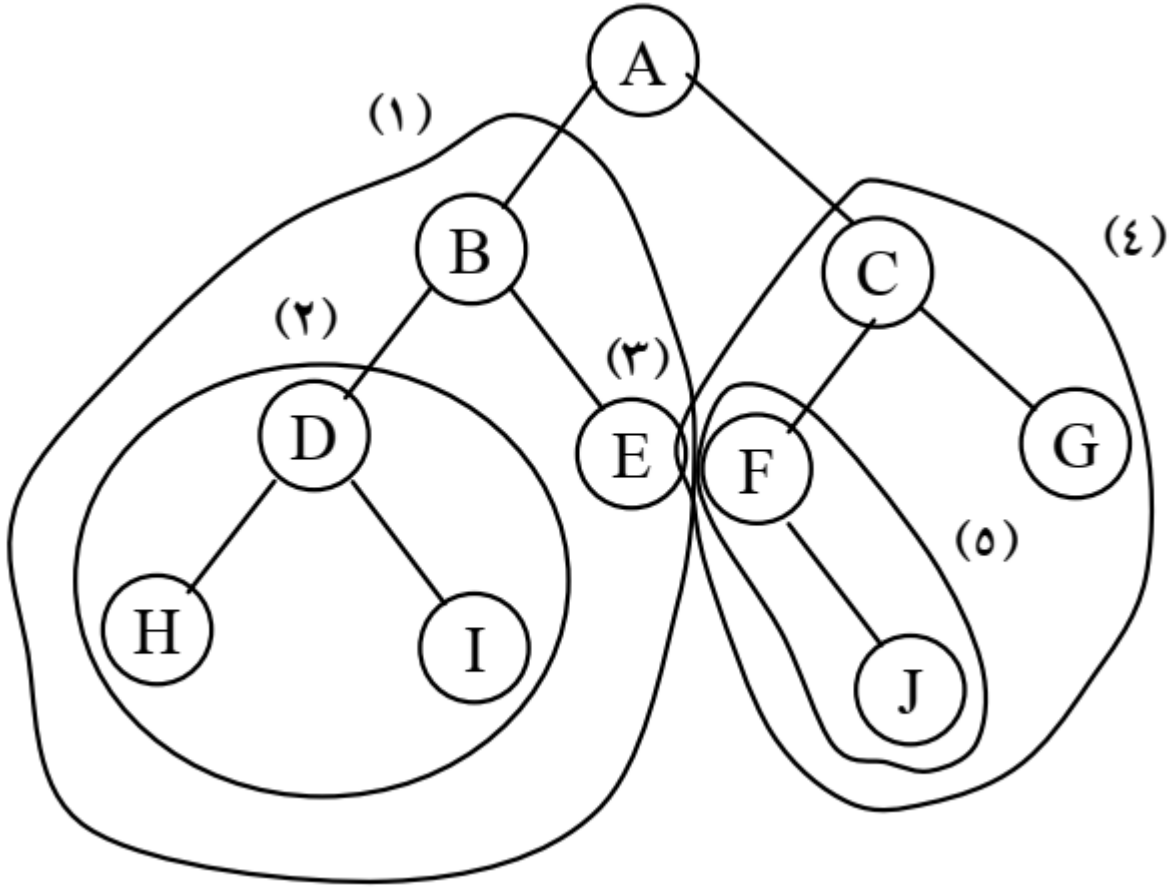
روش پیمایش Preorder
۱- ریشه را ملاقات کنید. ۲- زیردرخت چپ را به روش Preorder پیمایش کنید. ۳-زیردرخت راست را به روش Preorder پیمایش کنید.

روش پیمایش inorder
۱- زیردرخت چپ را به روش inorder پیمایش کنید. ۲- ریشه را پردازش کنید. ۳- زیردرخت راست را به روش inorder پیمایش کنید.

نمونه ای پیمایش ها



Preorder: ABDECF
Inorder: DBEACF

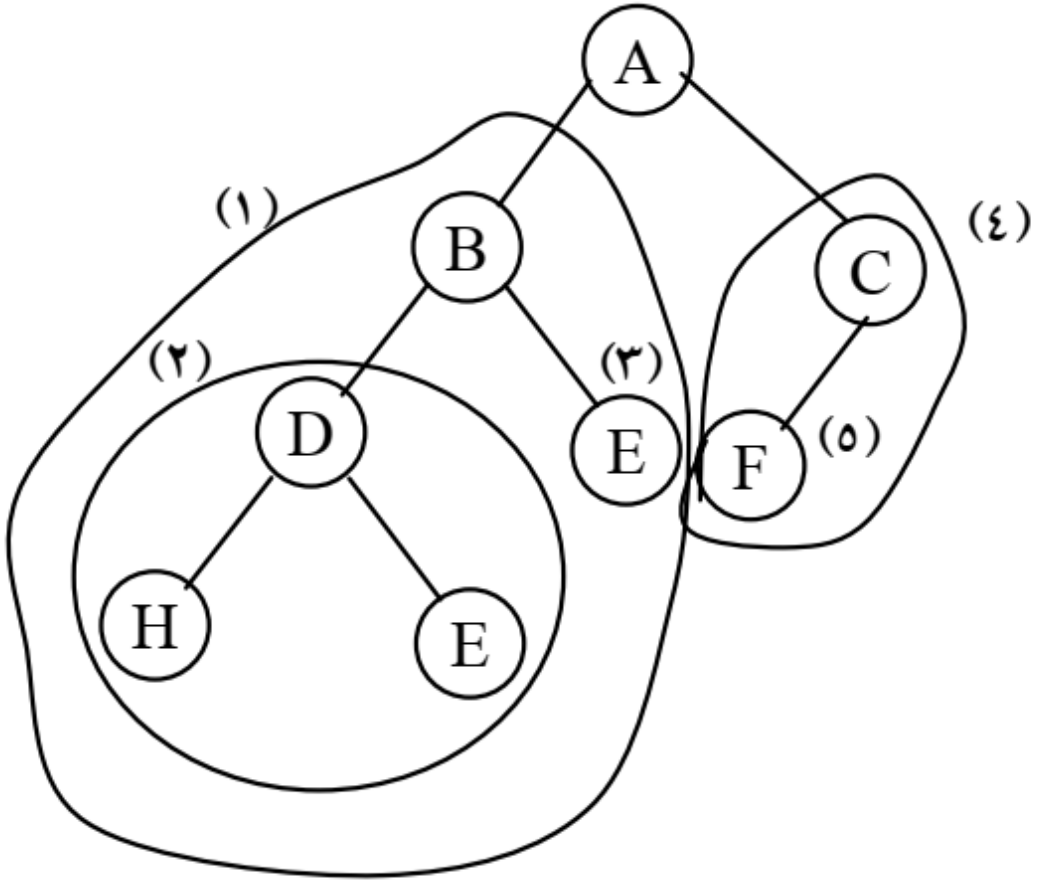


Inorder: HDIBEAFJCG
Preorder: ABDHIECFJG

پیمایش پس ترتیب

روش پیمایش Postorder
۱- زیردرخت چپ را به روش Postorder پیمایش کنید.
۲- زیردرخت راست را به روش Postorder پیمایش کنید.
۳- ریشه را پردازش کنید.

پیمایش پس ترتیب



Postorder: HEDEBFCA

Preorder: ABDHEECF

Inoredr: HDEBEAFC

ساخت درخت دودویی با استفاده از پیمایش های آن

- در صورت وجود یک پیمایش از یک درخت دودویی، نمی توان درخت منحصر به فرد ساخت.
- اما در صورت وجود پیمایش میان ترتیب و یکی از پیمایش های پس ترتیب و پیش ترتیب درخت دودویی می توان یک درخت یکتا ساخت.

نکته: اگر پیمایش های پسوندی یا پیشوندی یک درخت دودوئی را داشته باشیم ، ممکن است نتوانیم آن درخت را به صورت یکتا ترسیم کنیم.

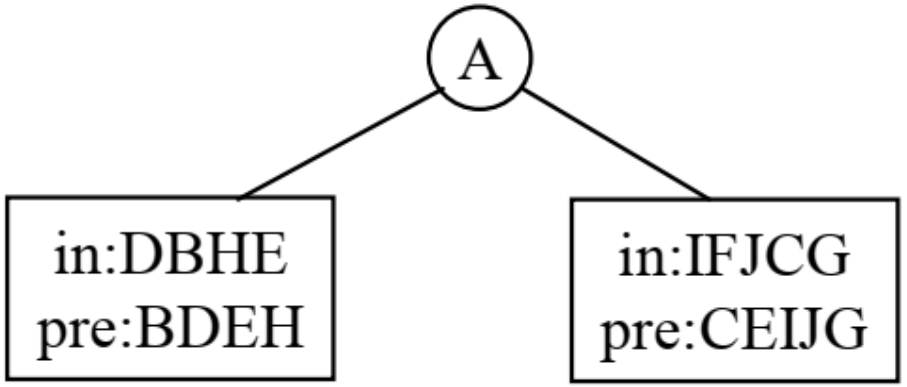
ساخت درخت دودویی با استفاده از پیمایش های آن

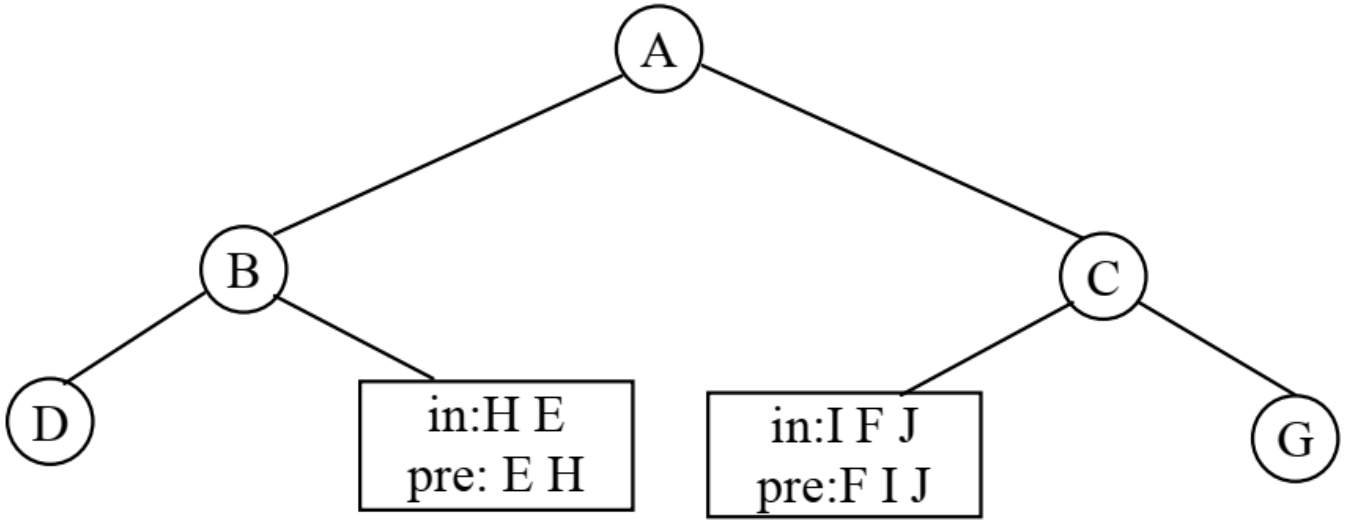
ساخت درخت دودوئی به صورت یکتا با داشتن پیمایش میانوندی و یکی از پیمایش های پسوندی یا پیشوندی
اگر پیمایش Preorder مشخص باشد، اولین گره آن به ریشه است. اگر نمایش Postorder مشخص باشد آخرین گره، ریشه است. وقتی گره ریشه مشخص شد، تمام گره‌های زیردرخت چپ و زیردرخت راست را می‌توان با استفاده از نمایش inorder پیدا کرد.

مثال

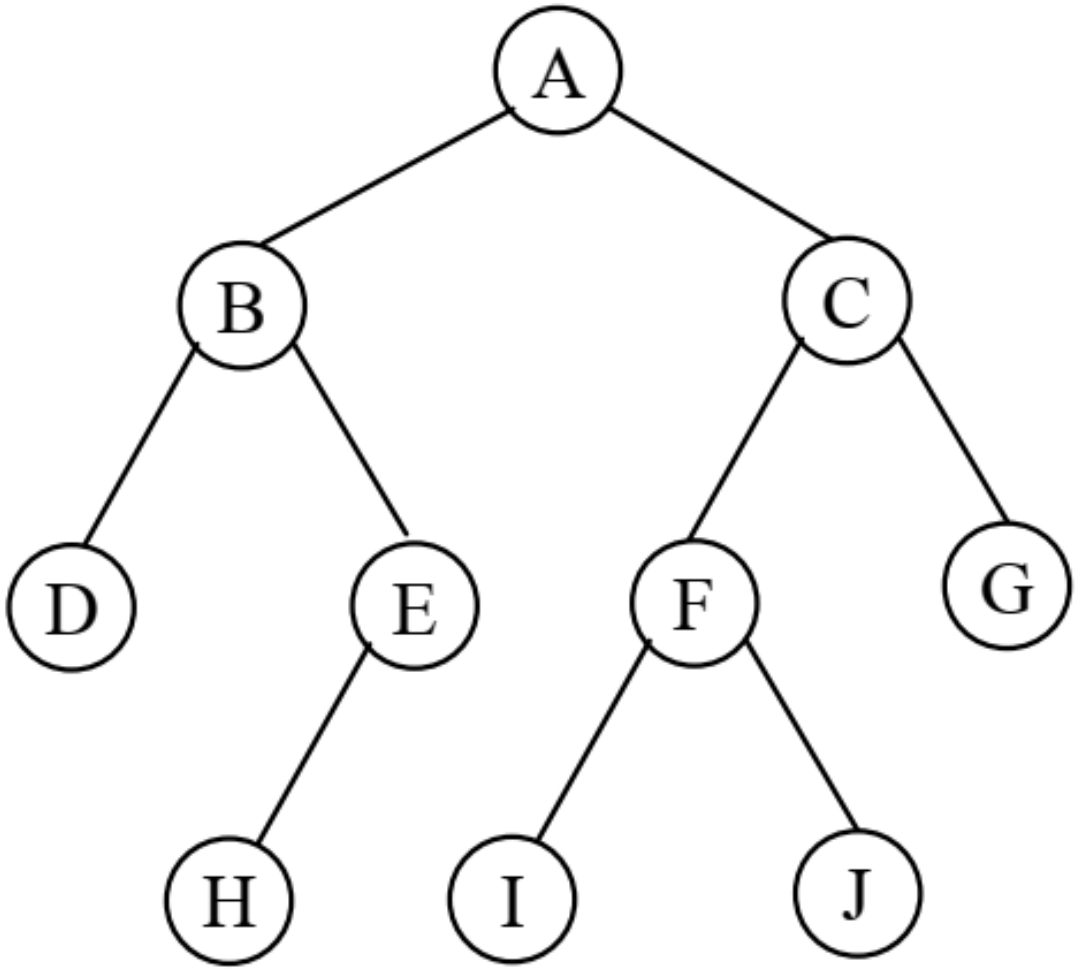
D B H E A I F J C G :inroder

A B D E H C F I J G :Preorder

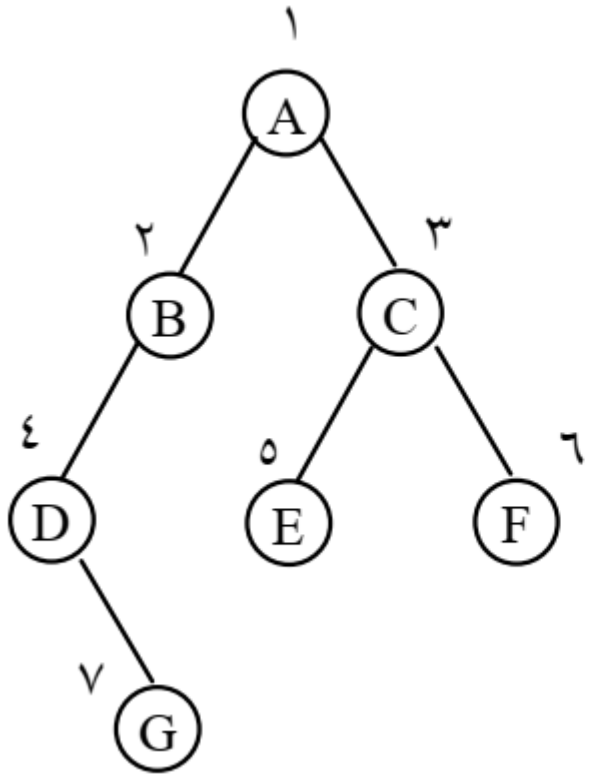




درخت نهایی



پیمایش سطحی درخت دودویی



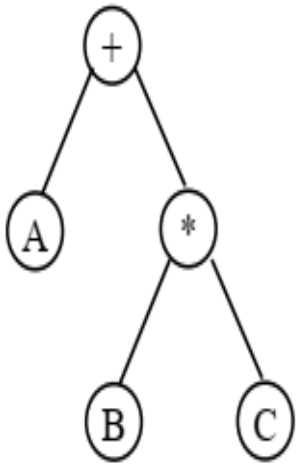
$\Rightarrow$ A B C D E F G

Preorder: +A*BC
Postoredr:ABC*+

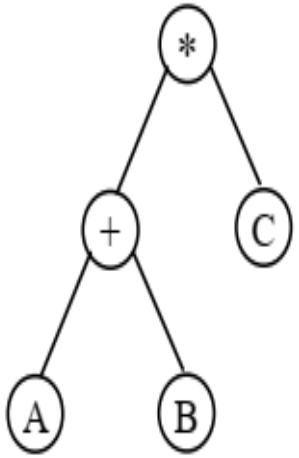
نمایش عبارت محاسباتی با درخت دودویی

- درخت دودویی محض: درختی دودوئی است که درجه تمام گره های آن از درجه صفر یا درجه دو است.
- برای نمایش عبارت های محاسباتی بوسیله درخت دودویی، علاوه بر محض بودن آن درخت، باید عملوندها در برگ های درخت و عملگرها در گره های غیربرگ درخت قرار گیرند.

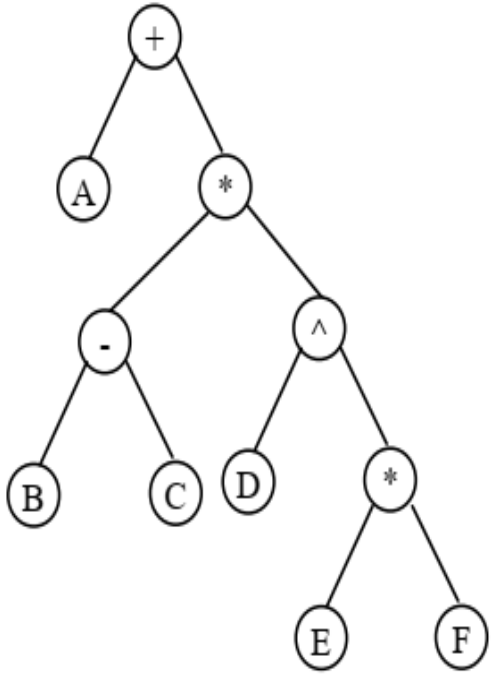
Inorder: A+(B-C)*D^(E*F)

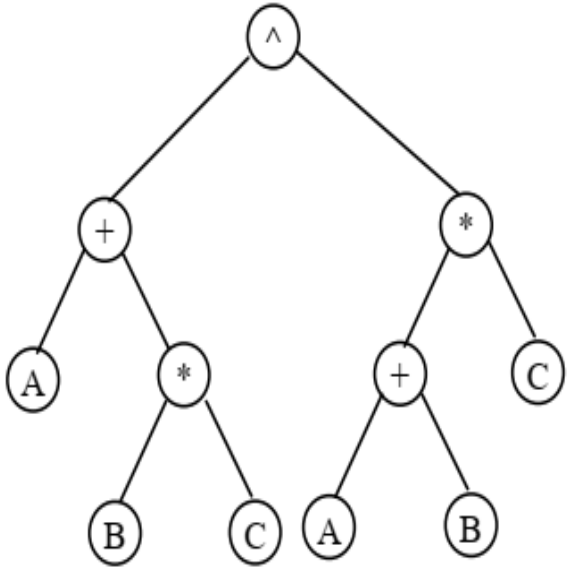


(a) A + B × C



(b) (A + B) × C





درخت جستجوی دودویی(Binary Search Tree-BST)

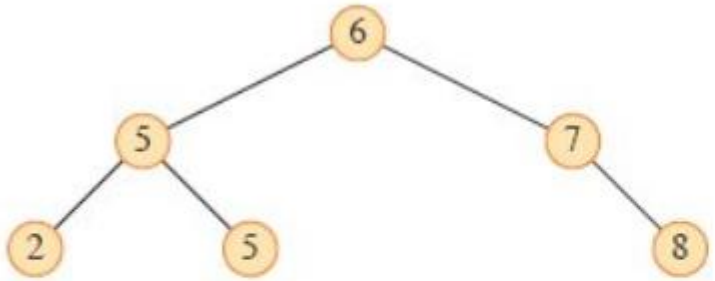
- ساختار داده درخت جستجو از هر یک از عملیات مجموعه پویا پشتیبانی می‌کند: جستجو، حداقل، حداکثر، مقدم، جانشین، درج و حذف. بنابراین، می‌توانید از یک درخت جستجو هم به عنوان یک فرهنگ لغت و هم به عنوان یک صف اولویت‌دار استفاده کنید.
- عملیات پایه روی یک درخت جستجوی دودویی، زمانی متناسب با ارتفاع درخت می‌گیرد. برای یک درخت دودویی کامل با n گره، چنین عملیاتی در بدترین حالت $\Theta(\lg n)$ اجرا می‌شوند. با این حال، اگر درخت یک زنجیره خطی از n گره باشد، همین عملیات در بدترین حالت $\Theta(n)$زمان می‌برد.

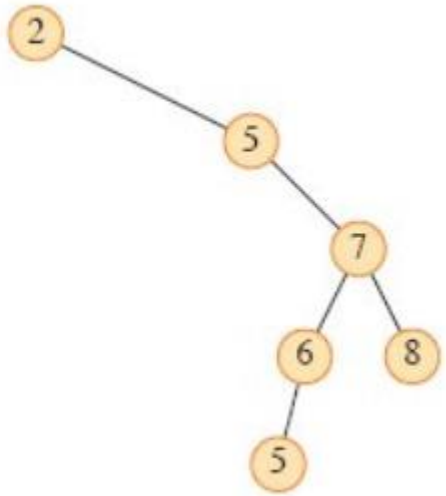
ساختار درخت جستجوی دودویی

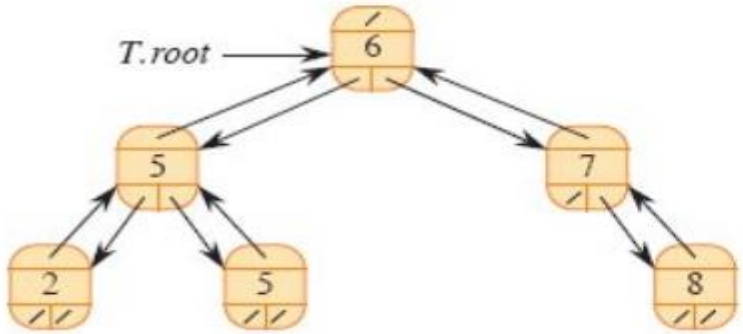
- یک درخت جستجوی یک درخت دودویی است که ممکن است تهی باشد. اگر درخت تهی نباشد خصوصیات زیر را برآورده می کند :

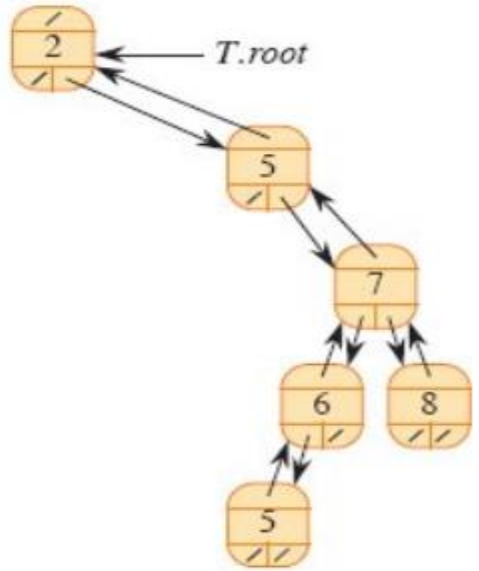
- هر عنصر دارای یک کلید است و دو عنصر نباید دارای کلید یکسان باشند ، در واقع کلیدها منحصر به فردند.
- کلیدهای واقع در زیردرخت غیرتهی چپ باید کمتر از مقدار کلید واقع در ریشه زیردرخت راست باشد.
- کلیدهای واقع در زیردرخت غیرتهی راست باید بزرگتر از مقدار کلید واقع در ریشه زیردرخت چپ باشد.
- زیردرختان چپ و راست نیز خود درختان جستجوی دودویی میباشند.

نمونه ای درختان جستجوی دودویی









نمایش تمام کلیدهای درخت جستجوی دودویی

- به دلیل ویژگی درخت جستجوی دودویی، می‌توانید تمام کلیدهای یک درخت جستجوی دودویی را به ترتیب مرتب شده با یک الگوریتم بازگشتی ساده، به نام پیمایش درخت درون‌ترتیبی، که توسط رویه INORDER-TREE-WALK ارائه می‌شود، چاپ کنید. این الگوریتم به این دلیل به این نام خوانده می‌شود که «کلید ریشه یک زیردرخت را بین چاپ مقادیر در زیردرخت چپ و چاپ مقادیر در زیردرخت راست آن چاپ می‌کند. (به طور مشابه، پیمایش درخت پیش‌ترتیبی، ریشه را قبل از مقادیر در هر یک از زیردرخت‌ها چاپ می‌کند و پیمایش درخت پس‌ترتیبی، ریشه را بعد از مقادیر در زیردرخت‌های خود چاپ می‌کند.) برای چاپ تمام عناصر در یک درخت جستجوی دودویی T، INORDER-TREEWALK(T.root) را فراخوانی کنید.

رویه مربوط به نمایش درخت جستجوی دودویی

- پیمایش یک درخت جستجوی دودویی n گره‌ای به زمان $\Theta(n)$ نیاز دارد، زیرا پس از فراخوانی اولیه، این رویه دقیقاً دو بار برای هر گره در درخت، خود را به صورت بازگشتی فراخوانی می‌کند – یک بار برای فرزند چپ و یک بار برای فرزند راست.

```
INORDER-TREE-WALK(x)
1 if x ≠ NIL
2     INORDER-TREE-WALK(x.left)
3     print x.key
4     INORDER-TREE-WALK(x.right)
```

جستجو در یک درخت جستجوی دودویی

- فرض کنید می خواهیم دنبال عنصری با کلید key بگردیم. ابتدا از ریشه (root) شروع می کنیم ، اگر ریشه تهی باشد، درخت جستجو فاقد هر عنصری بوده و جستجو ناموفق خواهد بود. در غیر این صورت key را با مقدار کلید ریشه مقایسه کرده:

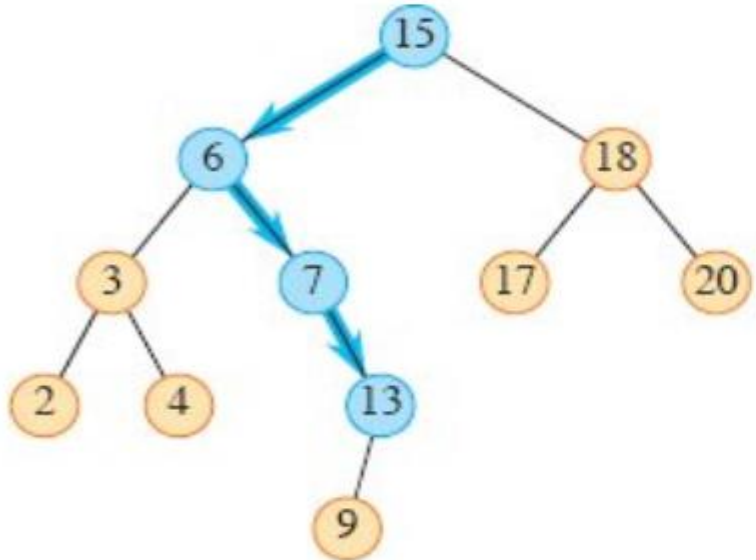
- اگر key کمتر از مقدار کلید ریشه باشد، هیچ عنصری در زیردرخت راست وجود ندارد که دارای کلیدی برابر key باشد، بنابراین زیردرخت چپ ریشه را جستجو می کنیم.

- اگر key بزرگتر از مقدار کلید ریشه باشد، زیردرخت راست را جستجو می کنیم.

- اگر h ارتفاع یا عمق یک درخت جستجوی دودویی باشد، عمل جستجو را در مدت O(h) انجام می شود.

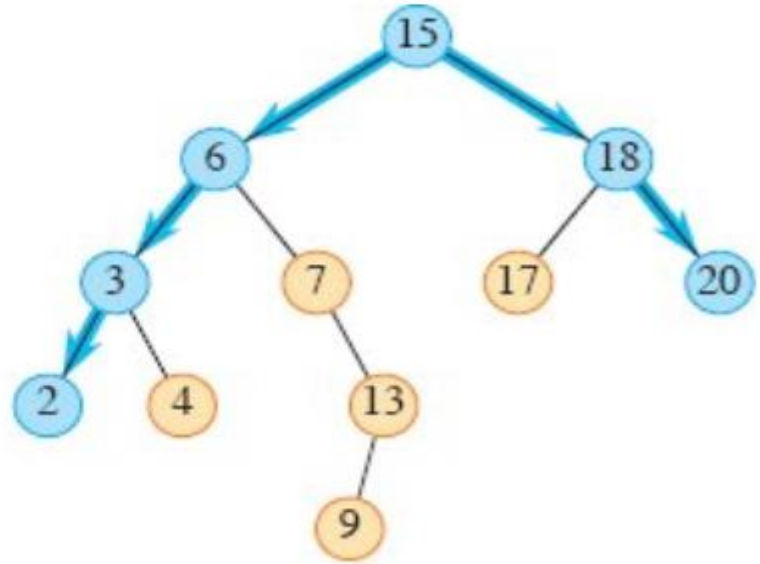
الگوریتم جستجو در درخت جستجوی دودویی

```
TREE-SEARCH(x, k)
1 if x == NIL or k == x.key
2       return x
3 if k < x.key
4       return TREE-SEARCH(x.left, k)
5 else return TREE-SEARCH(x.right, k)

ITERATIVE-TREE-SEARCH(x, k)
1 while x ≠ NIL and k ≠ x.key
2       if k < x.key
3             x = x.left
4       else x = x.right
5 return x
```

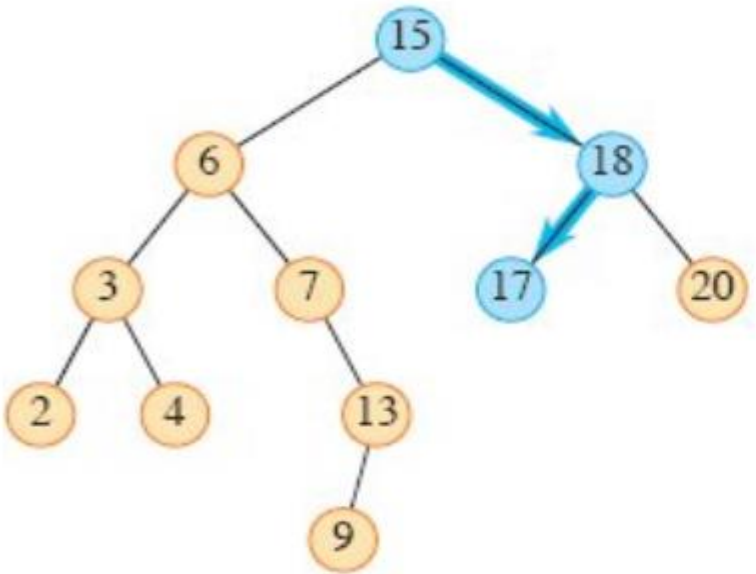


Inorder: 2 3 4 6 7 9 13 15 17 18 20

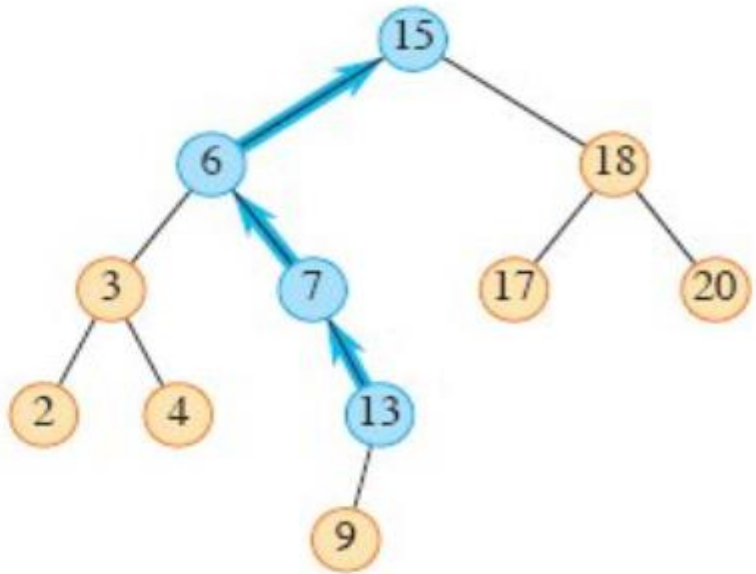
(a)



(b)



(c)

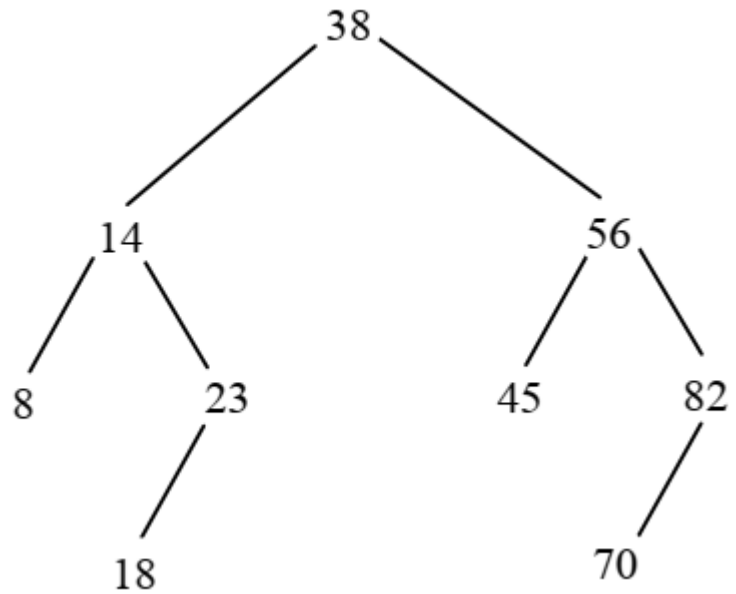


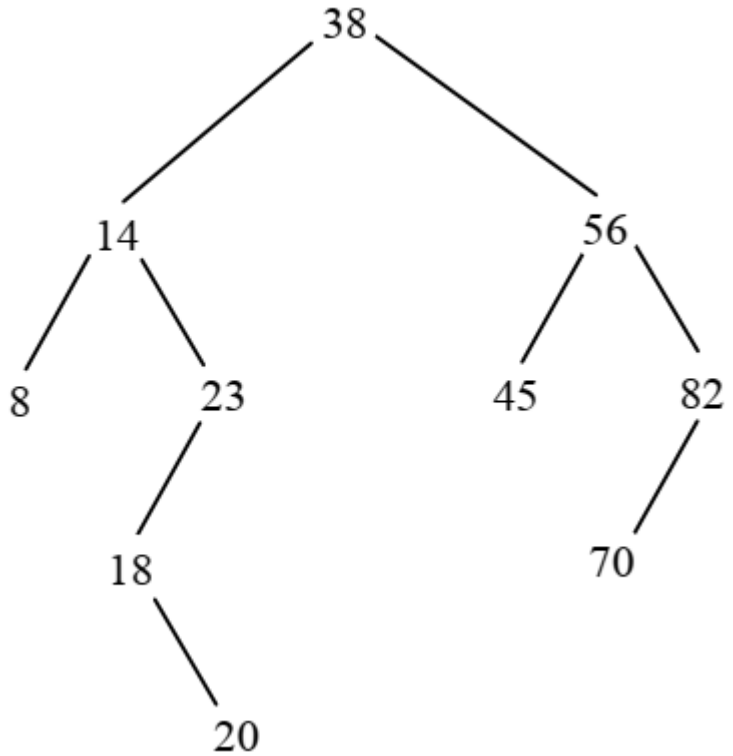
(d)

کوچکترین و بزرگترین عناصر

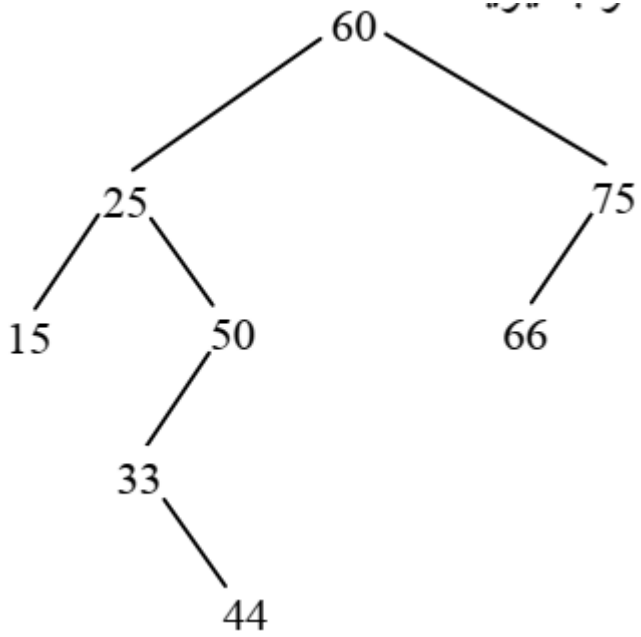
```
TREE-MINIMUM(x)
1 while x.left ≠ NIL
2       x = x.left
3 return x

TREE-MAXIMUM(x)
1 while x.right ≠ NIL
2       x = x.right
3 return x
```

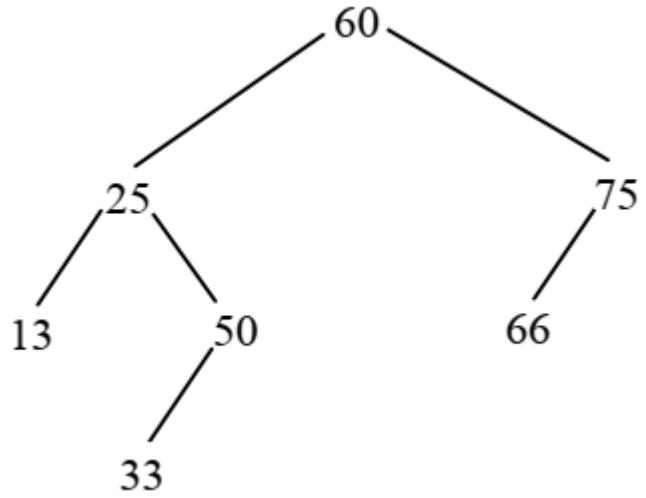
نمونه هایی از درج گره در درخت جستجوی دودویی



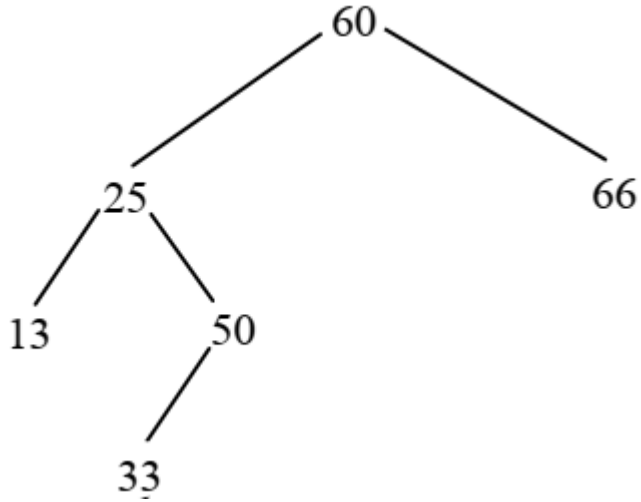


نمونه هایی از حذف گره از درخت جستجوی دودویی





پس از حذف گره با کلید ٤٤

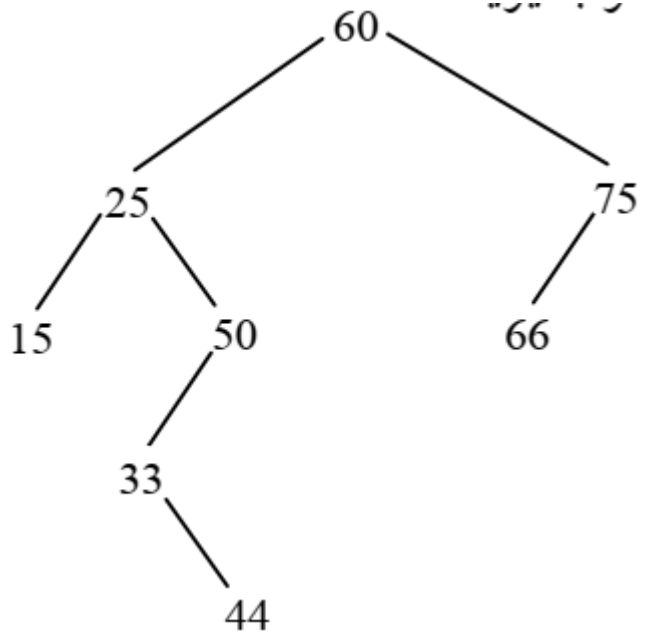


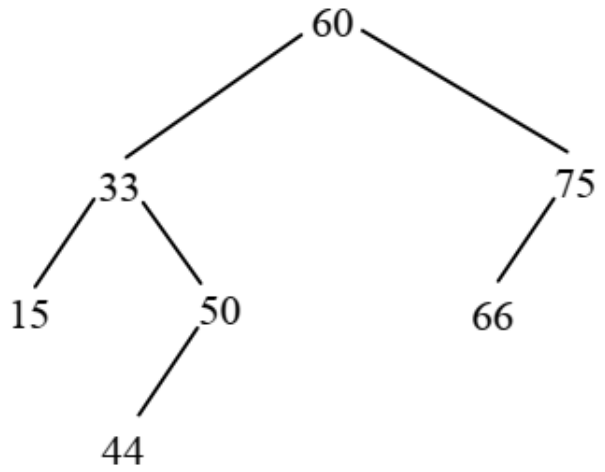
پس از حذف گره با کلید ۷۵

Root.right= x.left

حذف گرهی که از درخت جستجو که دارای دو فرزند است.

- ابتدا باید پیمایش میان ترتیب درخت را بدست آوریم، سپس گره بعدی در پیمایش میانوندی گره مورد نظر برای حذف، جایگزین آن گره می شود و این کار با تغییر اشاره گره ها در حافظه انجام می پذیرد.

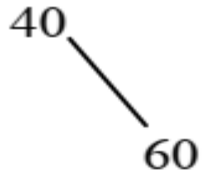




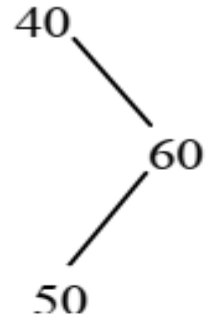
InOrder 13 25 33 44 50 60 66 75

ایجاد یک درخت جستجوی دودویی

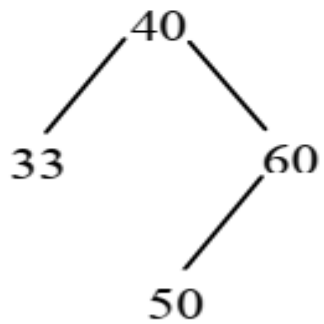
(1) key=40



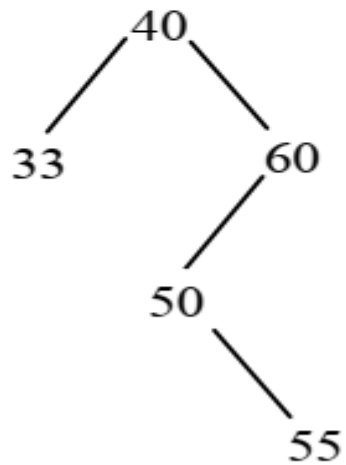
(2) key=60



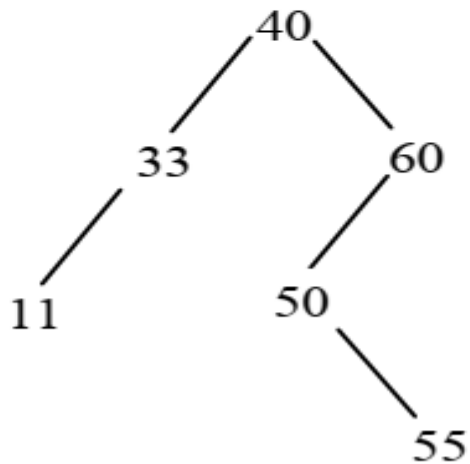
(3) key=50



(4) key=33



(5) key=55

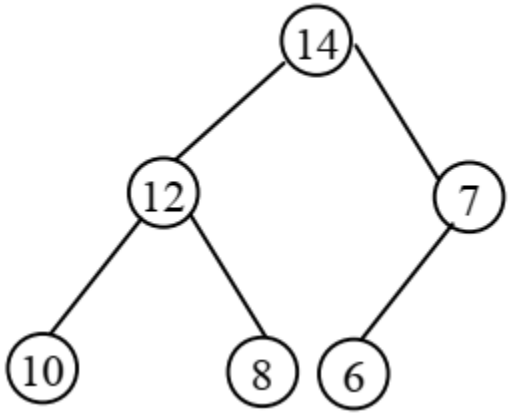


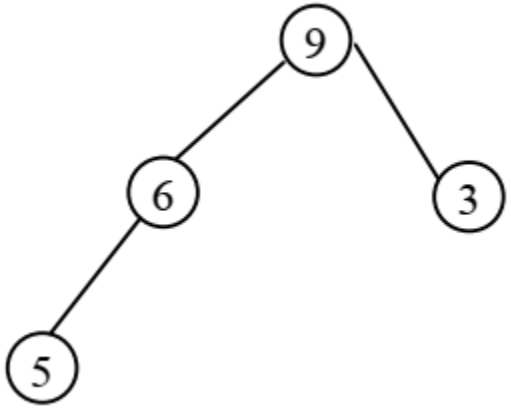
(6) key=11

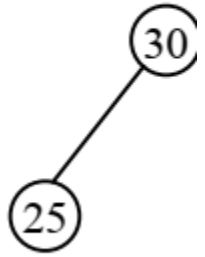
نوع داده مجرد (ADT) هرم

- **max tree** درختی است که مقدار کلید هر گره آن کمتر از مقادیر کلیدهای فرزندانش نباشد.
- **max heap** یک درخت دودویی کامل است که یک max tree نیز می باشد.
- **min tree** درختی است که مقدار کلید هر گره آن بیشتر از مقادیر کلیدهای فرزندانش نباشد.
- **min heap** یک درخت دودویی کامل است که در واقع یک min tree می باشد.

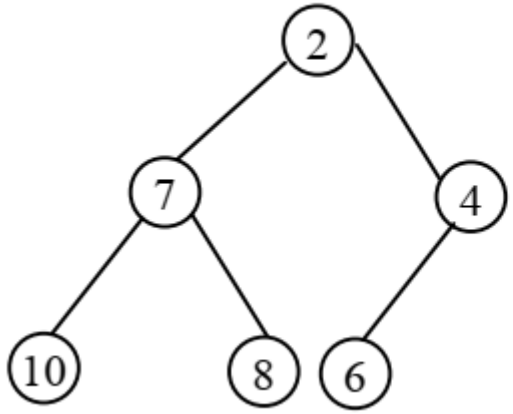
مثال از max heap و min heap

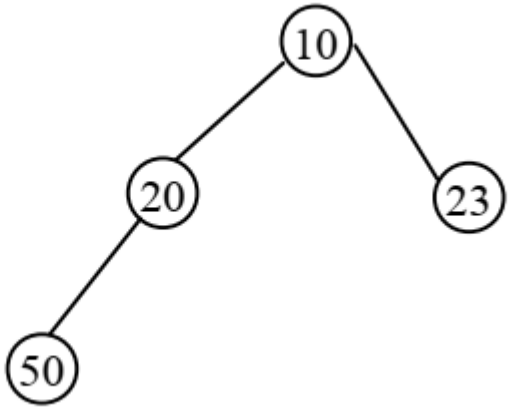


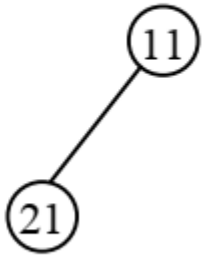




مثال از max heap

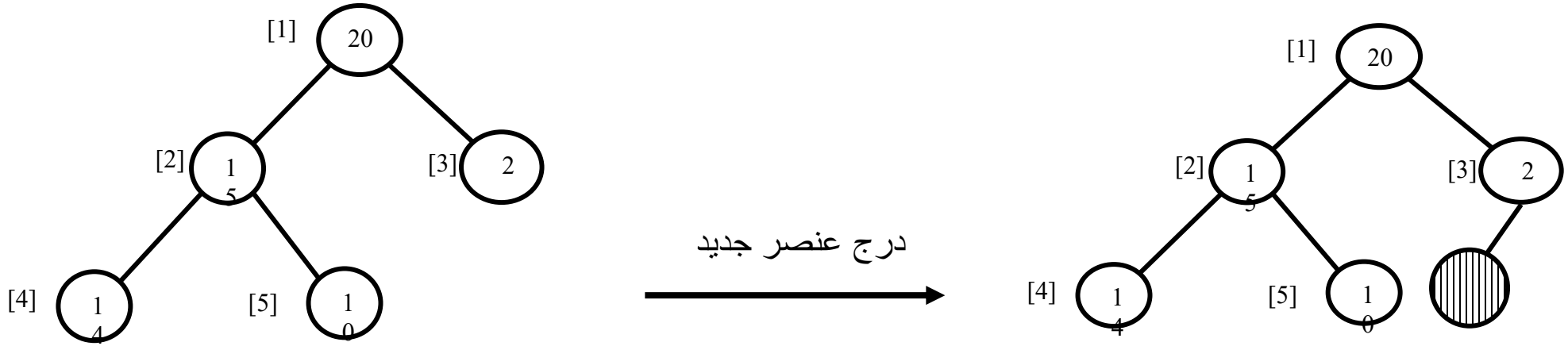






مثال از min heap

۶-۵ درج عناصر به داخل یک Max Heap

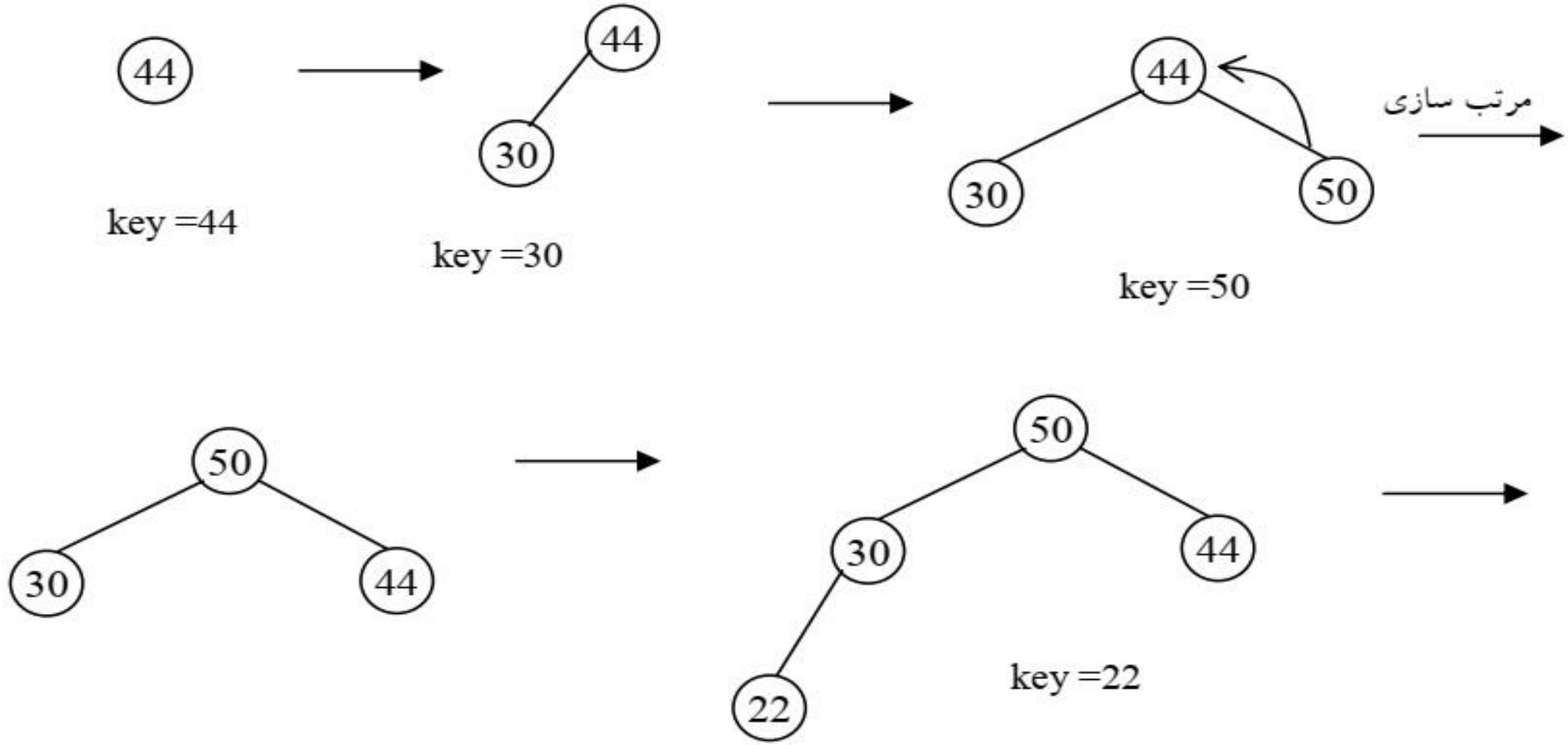


الف – درخت heap قبل از درج

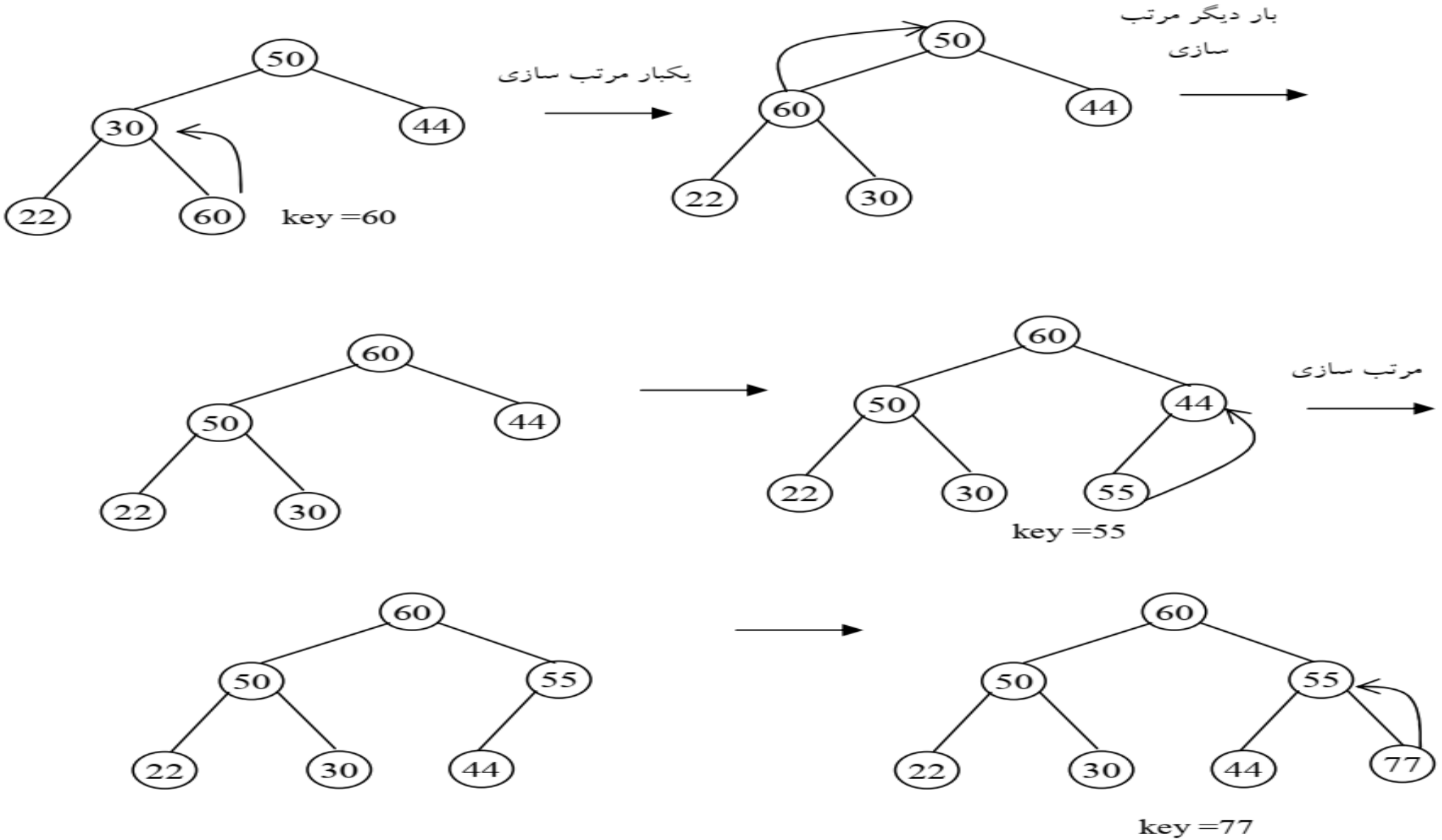
ب – محل اولیه گره جدید

اضافه کردن گره جدید در هر موقعیت دیگری ،تعریف heap را نقض می کند زیرا نتیجه یک درخت دودویی کامل نخواهد بود.

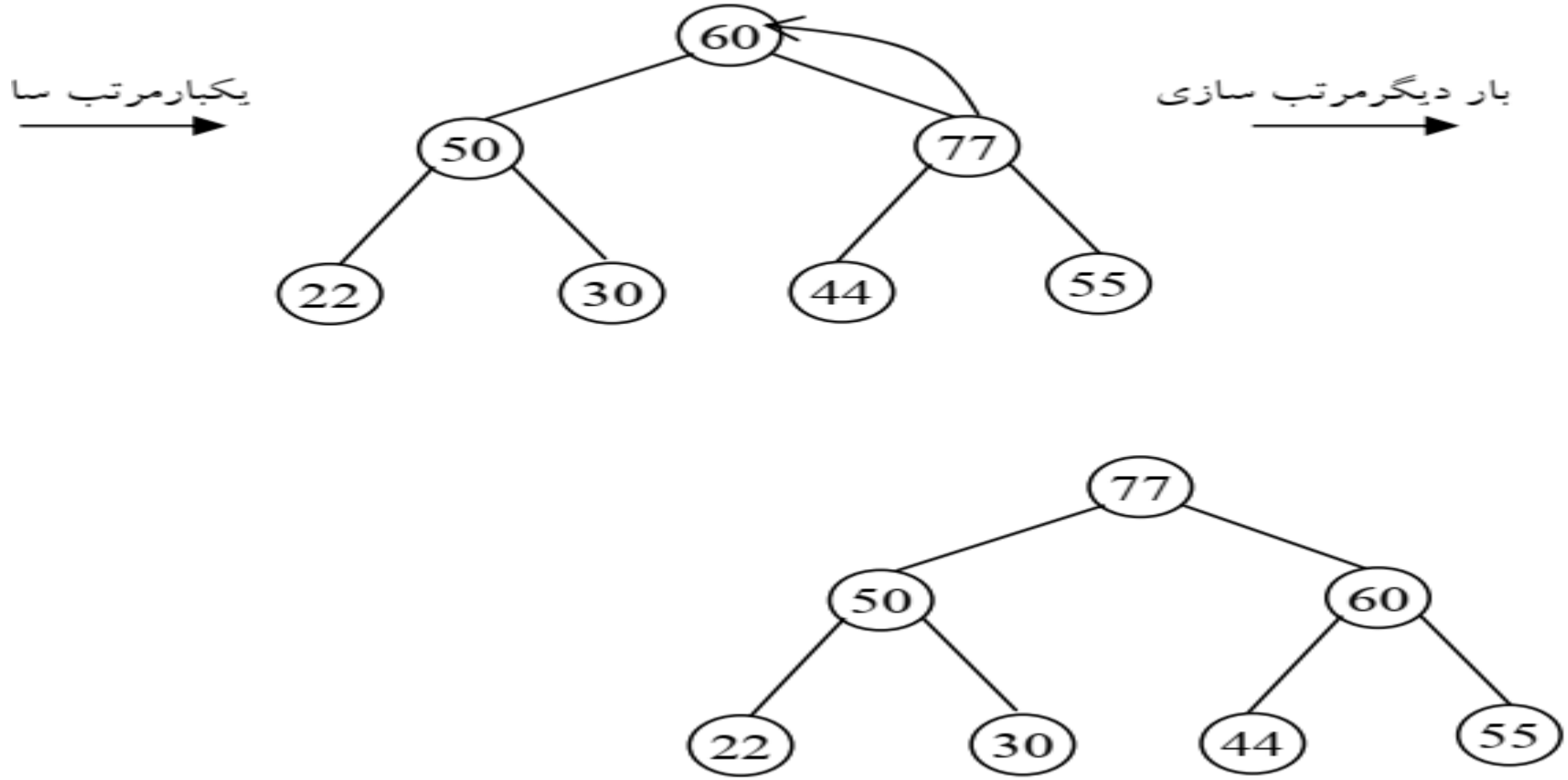
نمونه ای از ساخت یک Heap



نمونه ای از ساخت یک Heap



ساخت درخت نهایی هرم Max Heap



درخت های قرمز–سیاه

- درخت‌های قرمز–سیاه یکی از شکل های متعدد درخت جستجوی دودویی هستند که «متعادل» شده‌اند تا تضمین کنند که عملیات مجموعه پویای پایه در بدترین حالت $O(\lg n)$ زمان می‌برد.
- درخت قرمز–سیاه، یک درخت جستجوی دودویی با یک بیت حافظه اضافی به ازای هر گره است: رنگ آن که می‌تواند قرمز یا سیاه باشد. با محدود کردن رنگ گره‌ها در هر مسیر ساده از ریشه به برگ، درختان قرمز–سیاه تضمین می‌کنند که هیچ مسیری بیش از دو برابر مسیر دیگر نباشد، به طوری که درخت تقریباً متعادل باشد. در واقع، همانطور که خواهیم دید، ارتفاع یک درخت قرمز–سیاه با n کلید حداکثر $2\lg(n + 1)$ است که برابر با $O(\lg n)$ است.

ویژگی درخت قرمز–سیاه

درخت قرمز–سیاه یک درخت جستجوی دودویی است که ویژگی‌های قرمز–سیاه زیر را برآورده می‌کند:

۱. هر گره یا قرمز است یا سیاه.

۲. ریشه سیاه است.

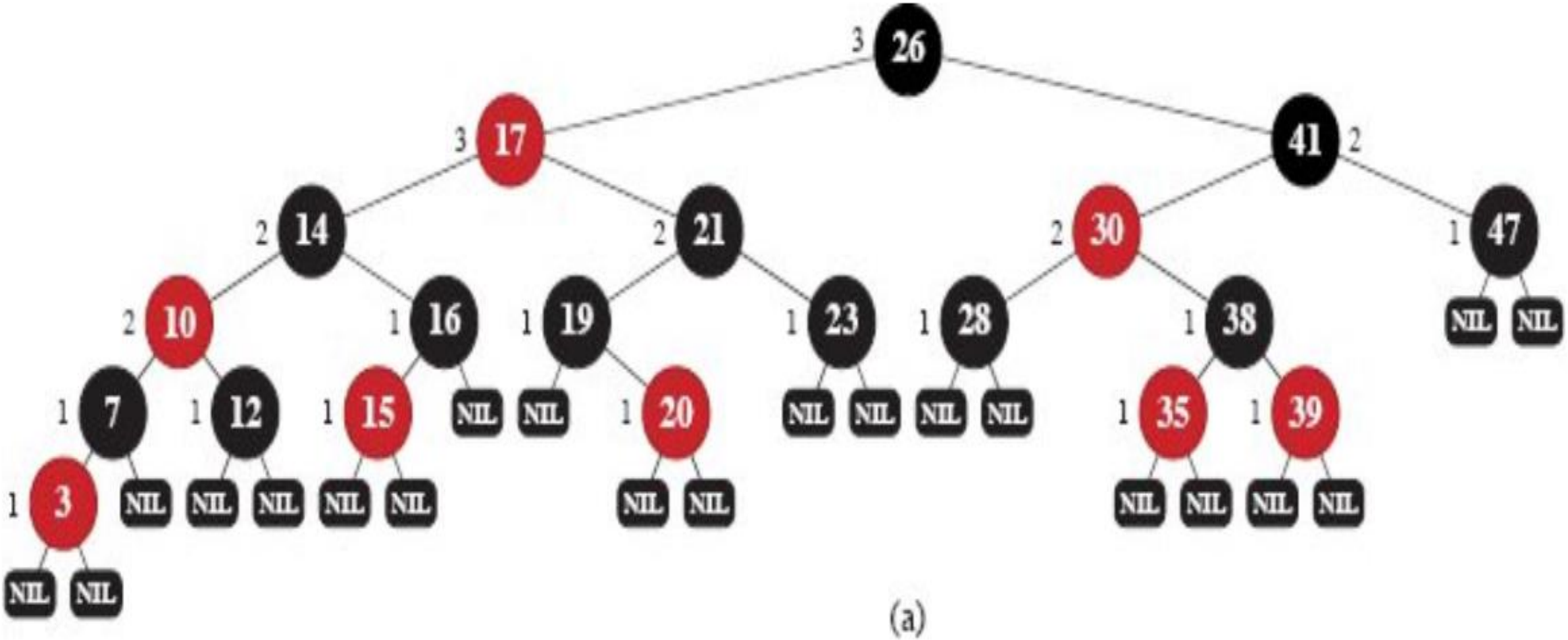
۳. هر برگ (NIL) سیاه است.

٤. اگر گره‌ی قرمز باشد، هر دو فرزند آن سیاه هستند.

۵. برای هر گره، تمام مسیرهای ساده از گره به برگ‌های فرزند شامل تعداد یکسانی گره سیاه هستند.

نمونه ای از یک درخت قرمز-سیاه

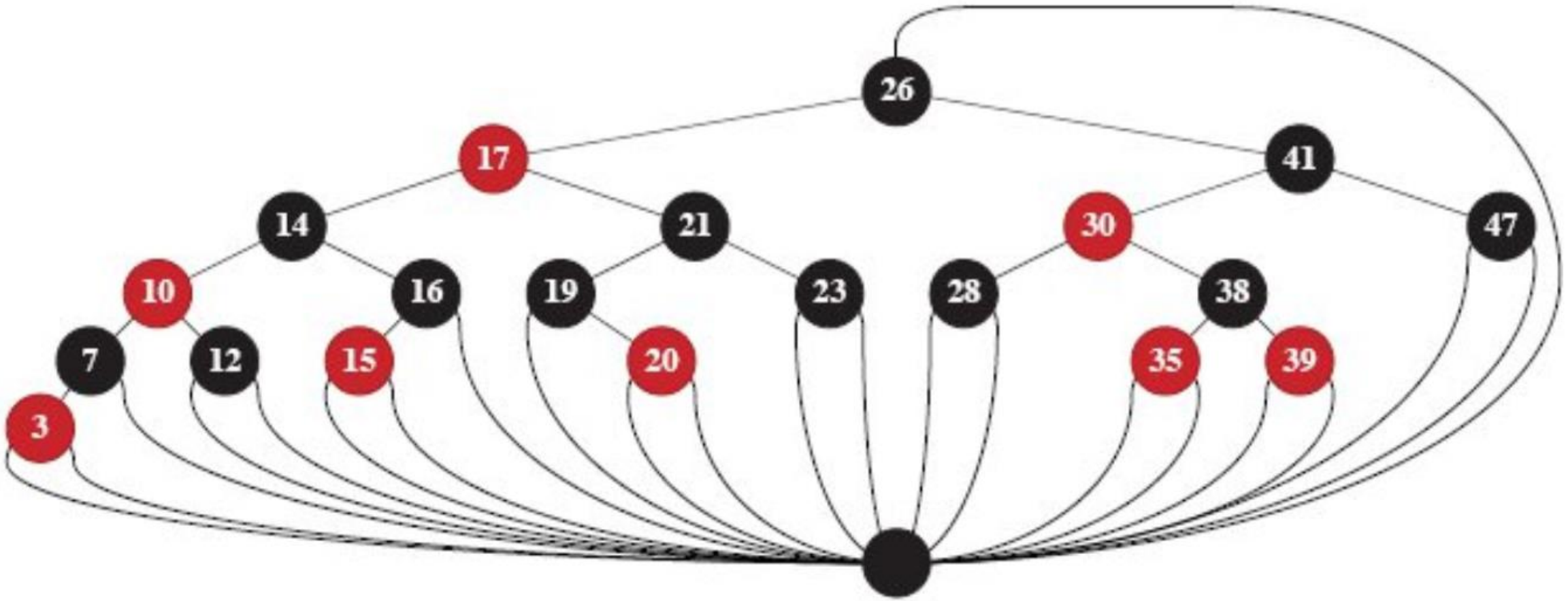
هر برگ، که به صورت NIL نشان داده شده است، سیاه است. هر گره غیر NIL با ارتفاع سیاه خود مشخص شده است، که در آن NILها دارای ارتفاع سیاه ۰ هستند.



(a)

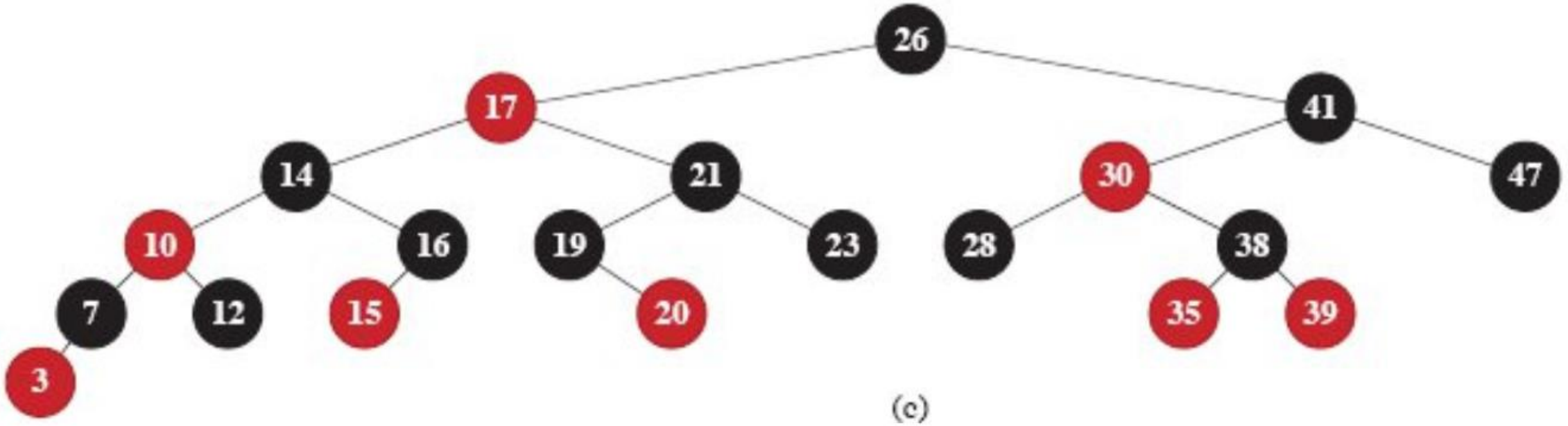
درخت های قرمز و سیاه

- **همان درخت قرمز-سیاه اما با هر NIL جایگزین شده با T.nil نگهبان واحد، که همیشه سیاه است و ارتفاع‌های سیاه حذف شده است. والد ریشه نیز نگهبان است.**



نمایش درخت های قرمز–سیاه

- همان درخت قرمز–سیاه اما با برگ‌ها و والد ریشه به طور کامل حذف شده است. بقیه این فصل از این سبک ترسیم استفاده می‌کند.



(c)

قضیه

- یک درخت قرمز–سیاه با n گره داخلی، حداکثر ارتفاعی برابر با $2\log_2 n + 1$ دارد.
- به عنوان نتیجه‌ی مستقیم این لم، هر یک از عملیات مجموعه پویای SEARCH، MINIMUM، MAXIMUM، SUCCESSOR و PREDECESSOR در زمان O(lg n) روی یک درخت قرمز–سیاه اجرا می‌شوند، زیرا هر یک می‌توانند در زمان O(h) روی یک درخت جستجوی دودویی با ارتفاع h اجرا شوند و هر درخت قرمز–سیاه روی n گره، یک درخت جستجوی دودویی با ارتفاع O(lg n) است. اگرچه رویه‌های TREE-INSERT و TREE-DELETE O(n gl) اجرا می‌شوند، اما نمی‌توانید از آنها فقط برای پیاده‌سازی عملیات مجموعه پویای INSERT و DELETE استفاده کنید. آنها لزوماً ویژگی‌های قرمز–سیاه را حفظ نمی‌کنند، بنابراین ممکن است در نهایت به یک درخت قرمز–سیاه صحیح نرسید.

ادامه قضیه

- برای تکمیل اثبات لم، فرض کنید h ارتفاع درخت باشد. طبق ویژگی ٤، حداقل نیمی از گره‌های هر مسیر ساده از ریشه به برگ، به جز خود ریشه، باید سیاه باشند. در نتیجه، ارتفاع ریشه سیاه باید حداقل h/2 باشد، و بنابراین داریم:

$$n \geq 2^{\frac{h}{2}} - 1 \implies \log_2(n+1) \geq \frac{h}{2} \implies h \leq 2\log_2(n+1)$$

- به عنوان نتیجه‌ی مستقیم این لم، هر یک از عملیات مجموعه پویا نظیر، جستجو، حداقل، حداکثر، جانشین و پیشایند در زمان O(lg n) روی یک درخت قرمز–سیاه اجرا می‌شوند، زیرا هر یک می‌توانند در زمان O(h) روی یک درخت جستجوی دودویی با ارتفاع h اجرا شوند.

گراف ها

- هر گراف G شامل دو مجموعه V وE است :
- V : مجموعه محدود و غیرتهی از رئوس است
- E : مجموعه ای محدود و احتمالا تهی از لبه ها می باشد.
- V(G) وE(G) : مجموعه رئوس و لبه های گراف G را نمایش می دهند.
- برای نمایش گراف هم می توانیم بنویسیم G = (E,V)

گراف

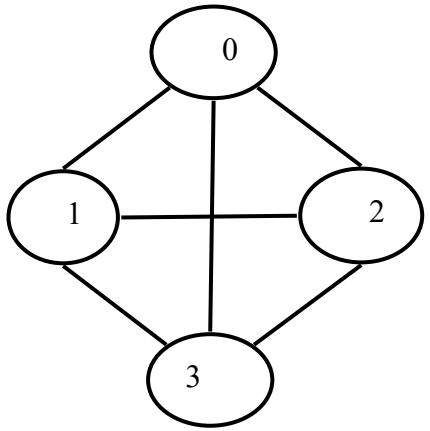
- این بخش روش‌هایی برای نمایش یک گراف و جستجوی یک گراف ارائه می‌دهد. جستجوی یک گراف به معنای دنبال کردن سیستماتیک یال‌های گراف به منظور رسیدن به رئوس گراف است. یک الگوریتم جستجوی گراف می‌تواند اطلاعات زیادی در مورد ساختار یک گراف کشف کند. بسیاری از الگوریتم‌ها با جستجوی گراف ورودی خود برای به دست آوردن این اطلاعات ساختاری شروع می‌کنند. چندین الگوریتم گراف دیگر نیز در مورد جستجوی گراف پایه توضیح می‌دهند. تکنیک‌های جستجوی گراف در قلب حوزه الگوریتم‌های گراف قرار دارند.

گراف ها

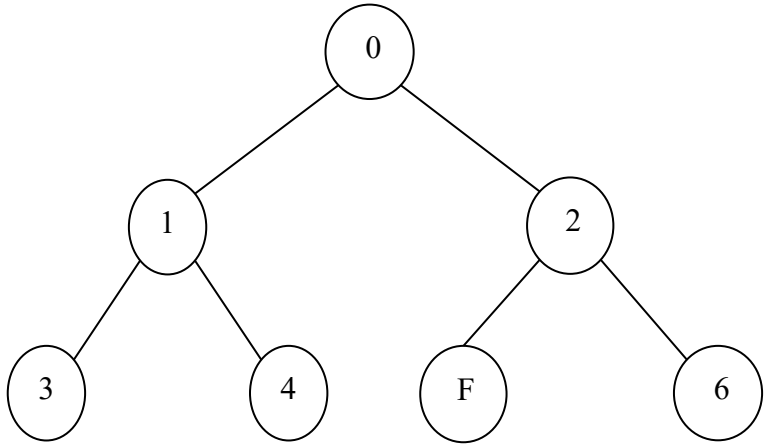
- در یک گراف بدون جهت زوج رئوس، زوج مرتب نیستند بنابراین زوج های (v_0, v_1) و (v_1, v_0) با هم یکسان هستند.
- در یک گراف جهت دار هر لبه با زوج مرتب $< v_0, v_1 >$ نمایش داده می شود که در آن v_0 ابتدای لبه و v_1 انتهای لبه است و در این حالت $< v_0, v_1 >$ و $< v_1, v_0 >$ با هم متنفاوت هستند.
- برای گراف بدون جهت ، لبه ها به صورت خطوط یا منحنی نمایش داده می شوند.
- برای گراف های جهت دار لبه ها به صورت فلش هایی که از انتها به ابتدا رسم شده است ، ارایه می گردند.

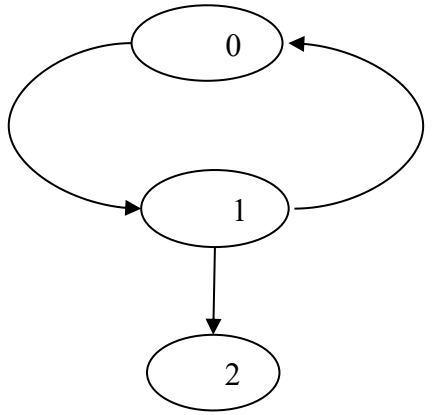
نمونه هایی از گراف های جهت دار و بدون جهت

مثال



بدون جهت





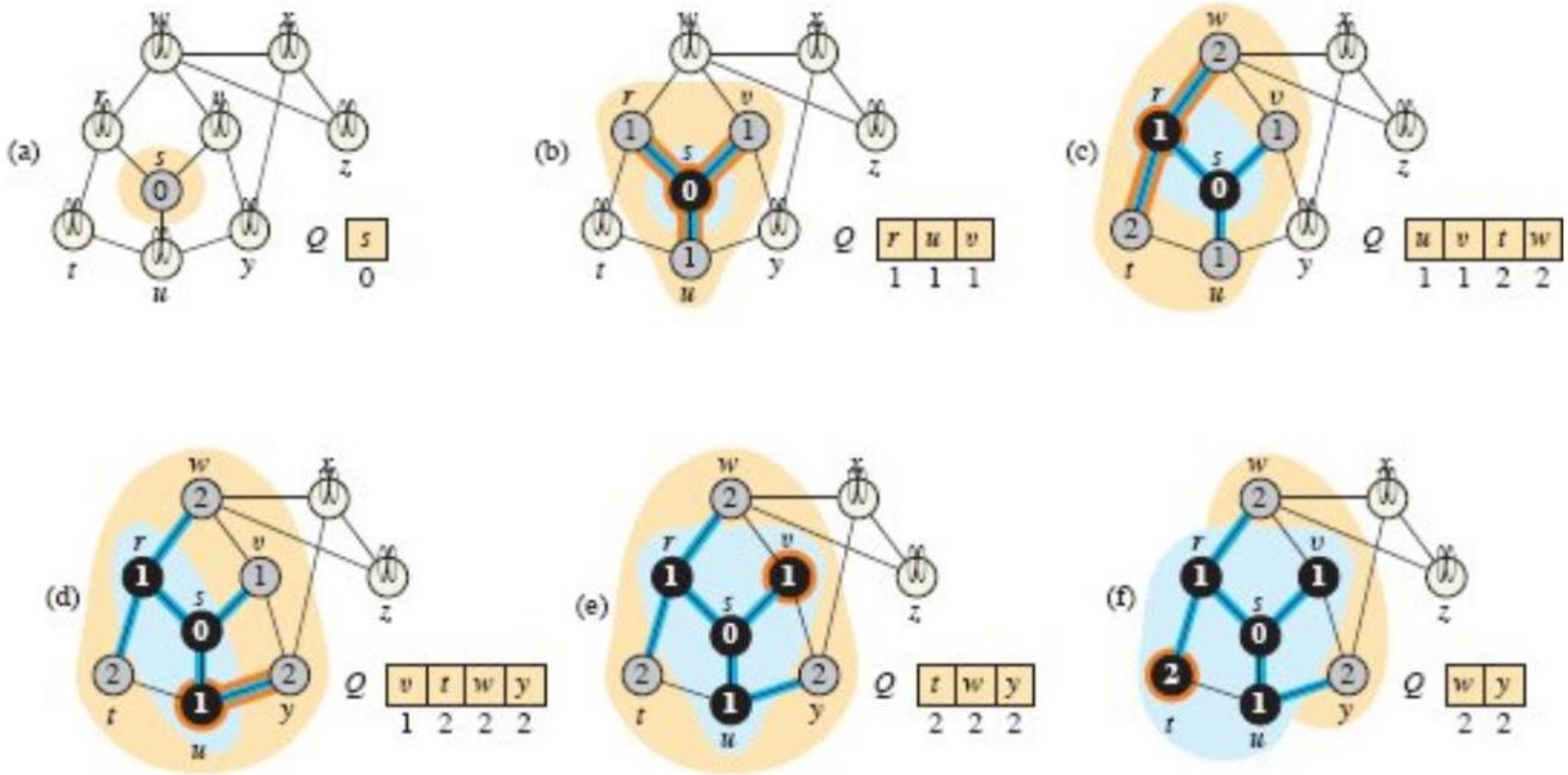
جهت دار

محدودیت های گراف ها

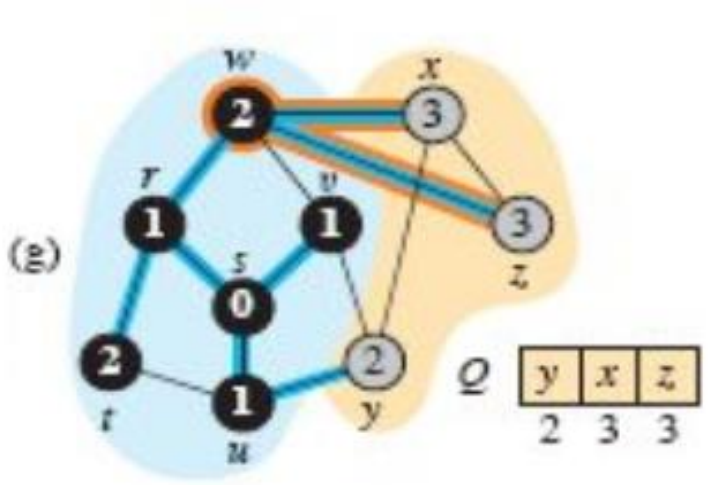
محدودیت های زیر بر روی گراف ها اعمال می شود :

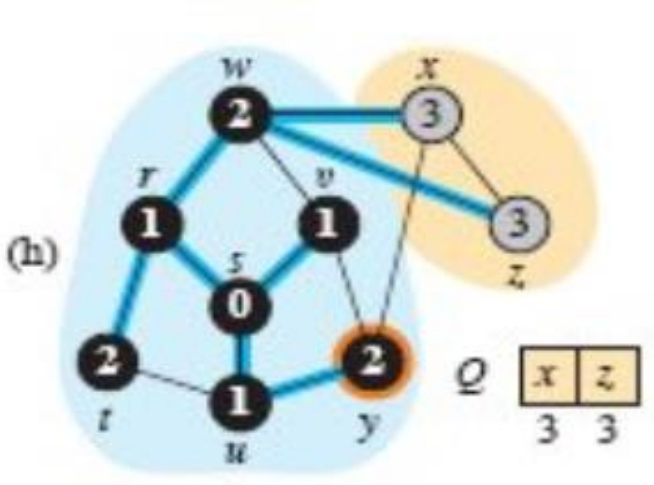
- **یک گراف فاقد لبه ای از یک راس مانند i ، به خودش می باشد. این مطلب بدین معنی است که لبه (v_i, v_i) غیر معتبر می باشد.**
- **یک گراف فاقد رویداد چندگانه از یک لبه می باشد.**
- **گراف کامل گرافی است که دارای حداکثر تعداد لبه بلشد.**

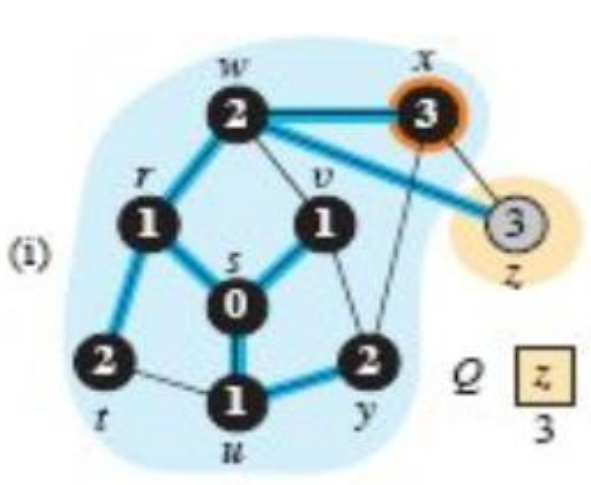
نمونه ای از پیمایش سطحی گراف

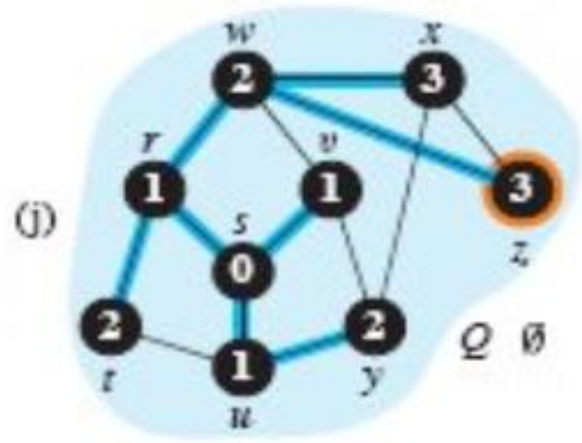


نمونه ای از پیمایش سطحی گراف-ادامه

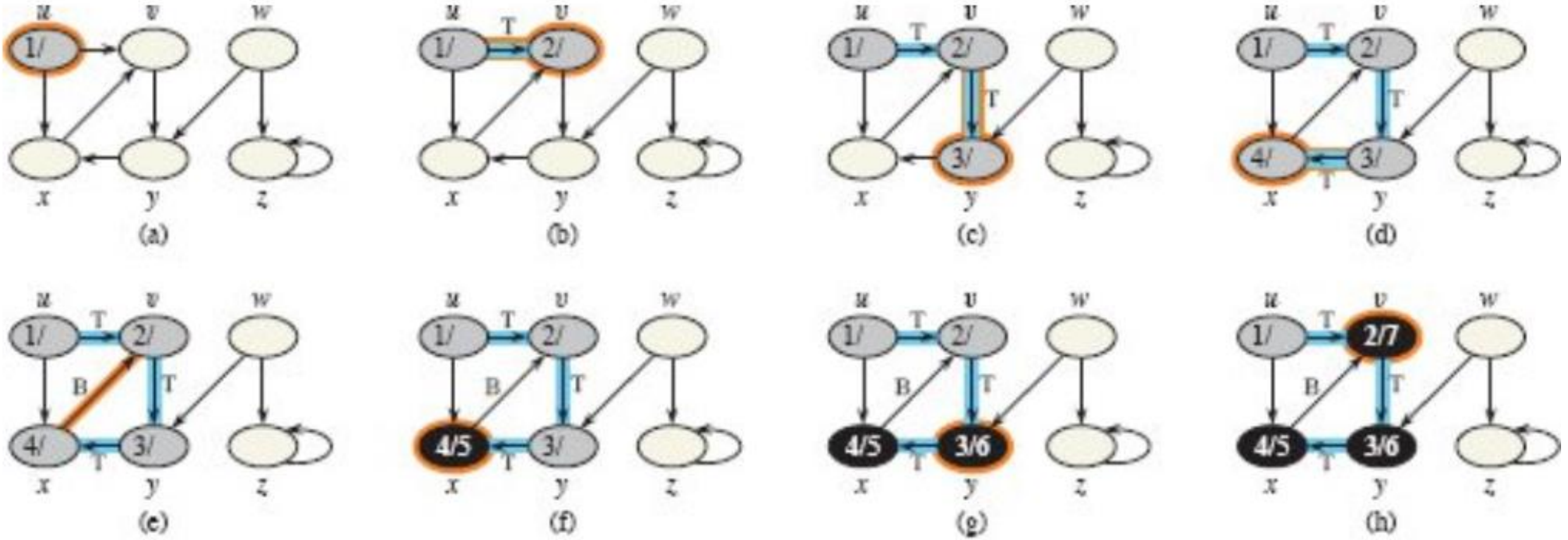








نمونه ای از پیمایش عمقی گراف



نمونه ای از پیمایش عمقی گراف-ادامه