آرایه‌ها　　　Array

آرایـه نوعی ساختمان داده است که عناصر آن هم نوع بوده و هر یک از عناصر با یک اندیس به صورت مسـتقیم قابـل دسـتیابی اسـت. آرایـه مـی‌تواند یک بعدی , دو بعدی و یا چند بعدی باشد. آرایه‌های دو بعدی را با نام ماتریس می‌شناسیم.

$[L_1 \dots U_1 , L_2 \dots U_2 , L_n \dots U_n]$

Array $[L \dots U]$ of items

تعداد عناصر آرایه $= U - L + 1$

تعداد عناصر آرایه n بعدی $= [U_1 - L_1 + 1][U_2 - L_2 + 1][U_n - L_n + 1]$

فضای اشغال شده توسط آرایه (فضای مورد نیاز) $= (U - L + 1) \times n$

مـثال : در یـک آرایـه بـه نام Float [200] اگر آدرس شروع آرایه در حافظه 1000 باشد A25 در کدام آدرس قرار دارد.

$A[i] = (i - L) \times n + \alpha$

محل عنصر iام در حافظه $= (25 - 0) \times 4 + 1000 = 1100$

آرایه‌های دوبعدی یا ماتریس‌ها به دو روش در حافظه ذخیره می‌شوند.

$$\begin{bmatrix} 2 & 5 \\ 1 & 6 \\ 3 & 4 \end{bmatrix} \quad 3 \times 2$$

۱. روش سطری　　　Row Major

سطری

0	1	2	3	4	5
2	5	1	6	3	4

۲. روش ستونی　　　Column Major

ستونی

0	1	2	3	4	5
2	1	3	5	6	4

A : Array $[L_1 \dots U_1 , L_2 \dots U_2]$ of items

تعداد عناصری $= [U_1 - L_1 + 1][U_1 - L_2 + 1]$

آدرس $A[i , j]$ در روش سطری $= [(i - L_1) \times (U_2 - L_2 + 1) + (j - L_2)] \times n + \alpha$

آدرس $A[i , j]$ در روش ستونی $= [(j - L_2) \times (U_1 - L_1 + 1) + (i - L_1)] \times n + \alpha$

مثال : طبق آرایه زیر , آدرس‌های خواسته شده را محاسبه نمائید.

$$\begin{array}{l} \quad L_1 \ldots U_1 \;\; L_2 \ldots U_2 \\ A : [1 \ldots 3 , 1 \ldots 2] \end{array} \quad \Longrightarrow$$

در زبان C داریم $\Longrightarrow A[3][2]$

$$\begin{bmatrix} 2 & 5 \\ 1 & 6 \\ 3 & 4 \end{bmatrix}$$

روش سطری $A[3,2] = (3-1) \times (2-1+1) + (2-1) = 2 \times 2 + 1 = 5$

روش ستونی $A[3,2] = (2-1) \times (3-1+1) + (3-1) = 1 \times 3 + 2 = 5$

روش سطری $A[1,2] = (1-1) \times (2-1+1) + (2-1) = 1$

روش ستونی $A[1,2] = (2-1) \times (3-1+1) + (1-1) = 3$

تمرین : در یک آرایه به شکل $A[1 \ldots 100 , 1 \ldots 26]$ of integer اگر این آرایه از محل 1000 حافظـه شروع شده باشد محل داده $A[60 , 6]$ در روش سطری و محل داده $A[20 , 4]$ در روش ستونی کدام آدرس حافظه است.

$$A[60 , 6] = (60-1) \times (26-1+1) + (6-1) \times 2 + 1000 = 4078$$

$$A[20 , 4] = (4-1) \times (100-1+1) + (20-1) \times 2 + 1000 = 1638$$

در آرایـه‌هـای دو بعـدی مربعی یا ماتریس‌های مربعی که کلیه عناصر بالای قطر اصلی آن صفر باشند یـک ماتریس پایین مثلثی تشکیل می‌گردد و برعکس اگر کلیه عناصر پایین قطر اصلی آن صفر باشند یـک ماتـریس بـالا مثلثـی تشـکیل خـواهد شـد. در یـک ماتـریس پایین مثلثی یا بالا مثلثی حداکثر $\frac{n(n+1)}{2}$ عنصر غیر صفر داریم که n اندازه هر بعد ماتریس است.

$$\begin{bmatrix} 1 & 6 & 7 \\ 0 & 2 & 5 \\ 0 & 0 & 4 \end{bmatrix}$$ بالا مثلثی

حداکثر عناصر غیر صفر $= \frac{3(3+1)}{2} = 6$

ماتریس بالا مثلثی $i > j \Longrightarrow A[i , j] = 0$

ماتریس پایین مثلثی $i < j \Longrightarrow A[i , j] = 0$

اگـر انـدازه ابعـاد ماتریس‌های مثلثی افزایش یابند این ماتریس‌ها حاوی تعداد زیادی صفر خواهند بود کـه ذخیـره کردن سطری یا ستونی ماتریس به طور کامل در حافظه باعث هدر رفتن بخشی از فضای حافظـه مـی‌گردد. به همین دلیل ماتریس‌های مثلثی را بصورت سطری یا ستونی بدون در نظر گرفتن صفرها در حافظه ذخیره می‌کنند.

پایین مثلثی =====> سطری $\frac{(i-1)\times i}{2}+j$

$$\begin{bmatrix} 1 & 0 & 0 \\ 2 & 5 & 0 \\ 3 & 1 & 1 \end{bmatrix}$$ =====>

پایین مثلثی

1	2	3	4	5	6
1	2	5	3	1	1

بالا مثلثی =====> ستونی $\frac{(j-1)\times j}{2}+i$

$$\begin{bmatrix} 1 & 6 & 7 \\ 0 & 2 & 5 \\ 0 & 0 & 4 \end{bmatrix}$$ =====>

بالا مثلثی

1	2	3	4	5	6
1	6	2	7	5	4

جمع ماتریس‌ها

در جمـع دو ماتریس , حتماً باید یک ماتریس $m \times n$ با یک ماتریس $m \times n$ جمع شده و نتیجه نیز یک ماتریس $m \times n$ خواهد شد. در این عملیات عناصر دو آرایه نظیر به نظیر با یکدیگر جمع خواهند شد.

$$A_{m\times n} + B_{m\times n} = C_{m\times n}$$

for $(i = 0 , i < m , ++i)$

for $(j = 0 , j < n , ++j)$

$C_{ij} = a_{ij} + b_{ij}$

ضرب ماتریس‌ها

در عمـل ضـرب , یـک ماتـریس A_{mL} و یک ماتریس B_{Ln} با یکدیگر ضرب شده و ماتریس بدست آمده نیز دارای سطر و ستونهایی می‌باشد که سطر ماتریس بدست آمده با تعداد سطرهای ماتریس اول و ستون ماتریس بدست آمده با تعداد ستونهای ماتریس دوم برابر است.

$$C_{mn} = A_{\underset{i}{m}\,\underset{k}{L}} \times B_{\underset{k}{L}\,\underset{i}{n}}$$

$$\begin{bmatrix} 3 & 4 & 1 & 2 \\ 2 & 5 & 3 & 1 \\ 1 & 0 & 2 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 0 & 1 \\ 1 & 1 \\ 3 & 5 \end{bmatrix} = \begin{bmatrix} - & - \\ - & - \\ - & - \end{bmatrix}$$

$$3 \times 4 \quad \times \quad 4 \times 2 \quad = \quad 3 \times 2$$

for $(i = 0 , i < m , ++i)$

for $(j = 0 , j < n , ++j)$

{

$C_{ij} = 0$

for $(k = 0 , k < L , ++k)$

$C_{ij} = a_{ik} \times b_{kj} + C_{ij}$

}

تمرین : مقدار [0, 1] C را در حاصلضرب دو ماتریس مثال قبل بدست آورید.

جــواب : برای بدست آوردن مقدار خواسته شده باید حلقه‌های for بالا را Trace کنیم. پس بنابراین داریم :

$$C_{ij} = 0$$
$$C_{ij} = a_{ik} \times b_{kj} + C_{ij}$$
$$C_{ij} = 2 \times 1 + 0 = 2$$
$$C_{ij} = 5 \times 0 + 2 = 2$$
$$C_{ij} = 3 \times 1 + 2 = 5$$
$$C_{ij} = 1 \times 3 + 5 = 8$$

i	j	k	L	C_{ij}
1	0	0	4	0
		1		2
		2		2
		3		5
				8

ترانهاده

برای اینکه ترانهاده یک ماتریس را بدست آوریم جای سطرها و ستونهای ماتریس عوض می‌شوند.

$$A\begin{bmatrix} 2 & 3 & 5 \\ 1 & 0 & 7 \end{bmatrix} \rightarrow A^T\begin{bmatrix} 2 & 1 \\ 3 & 0 \\ 5 & 7 \end{bmatrix}$$

$$\begin{bmatrix} 2 & 0 & 4 \\ 1 & 2 & 5 \\ 1 & 3 & 8 \end{bmatrix} \rightarrow \begin{bmatrix} 2 & 1 & 1 \\ 0 & 2 & 3 \\ 4 & 5 & 8 \end{bmatrix}$$

$A_{m\times n}$

```
for (i = 0 , i < m , + + i)
for (j = 0 , j < n , + + j)
A [j] [i] = A [i] [j]
```

جستجوی خطی در آرایه

```
Array A[n] , x
Int search (A[n] , x) ;
{
    int i = 1 ;
    while (i <= n && A[i] != x)
        i + + ;
    if (i > n ) return – 1 // داده پیدا نشده
    else return i // داده در محل اندیس آرایه است
}
```

1	2	3	4	5
1	5	2	8	6

n	x	i	A[i]
5	8	1	1
		2	5
		3	2
		4	8

x = A[4]

جستجوی دودویی برای آرایه‌های مرتب

```
Int bsearch (A[n] , int x , int L , int U)
{
    int i ;
    while
    {
        i = ⌊(L + U) / 2⌋;
        if ( x < A[i] ) U = i – 1
        else if ( x > A[i] ) L = i + 1
            else return i // داده در اندیس i است
    }
    return – 1 // داده پیدا نشده است
}
```

$$i = \left\lfloor \frac{L+U}{2} \right\rfloor;$$

x	i	L	U	A[i]
8	5	1	10	12
	2	3	4	2
	4	4		5
	3			8

جمع دو چندجمله‌ای بوسیله آرایه

$P(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_2 x^2 + a_1 x + a_0$

a_n	a_{n-1}	a_{n-2}	....	a_1	a

$P(x) = 3x^2 + 5x + 2$

$P(x) = 3x^3 + 3$

0	3	5	2	+
3	0	0	3	=
3	3	5	5	

$P(x)\ x^{100} + 2$

ضریب	1	2
توان	100	0

Stack یا پشته

Stack لیستی است که اعمال ورودی و خروجی یا اضافه و حذف در آن از یک طرف لیست انجام می‌شود. به این جهت به آن لیست Last In First Owt (LIFO) می‌گویند. بدین معنی که آخرین ورودی به پشته , اولین خروجی خواهد بود. عنصر بالایی پشته را top پشته می‌گویند. با افزودن داده روی پشته , متغیر top یکی زیاد شده و داده در محل top از پشته قرار می‌گیرد. برای خارج کردن یک عنصر از پشته نیز داده‌ای که در محل top قرار گرفته از Stack خارج می‌گردد و متغیر top یکی کم می‌شود. مقدار اولیه top صفر است و با افزودن داده به یک پشته n عضوی , top می‌تواند تا مقدار n تغییر کند.

Top = 0 ===➔ پشته خالی است

Top = n ===➔ پشته پر است

دو عمل اصلی برای پشته‌ها را با push کردن و pop کردن می‌شناسیم. Push (x) داده x را در بالای پشته قرار می‌دهد و عمل pop عنصر بالای پشته را در متغیر x ذخیره می‌کند.

$x = pop \equiv pop\ (x)$

Stack : Array [1 .. n] of items

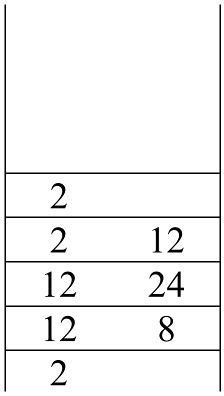
```
void push (int x)
{
    if (top = = n)
    {
        C out << " پشته پر است " ;
        return – 1 ;
    }
    else
    {
        top + + ;
        Stack [top] ;
    }
}
```

```
int pop ( )
{
    int x ;
    if (top = 0)
    {
        C out << " پشته خالی است " ;
        return – 1 ;
    }
    else
    {
        x = Stack [top] ;
        top = top – 1 ;
    }
    return x ;
}
```

مثال : مقدار نهایی A و B و C چقدر است؟

n = 5   A = 10   B = 2   C = 5

push (B)
push (A + B)
pop (C)
push (A – B)
push (C)
push (B)
pop (A)
pop (B)
push (A × B)
push (C)
push (A)
pop (B)
pop (C)
pop (A)



A	B	C
~~10~~	~~2~~	~~5~~
~~2~~	~~12~~	~~12~~
24	2	12

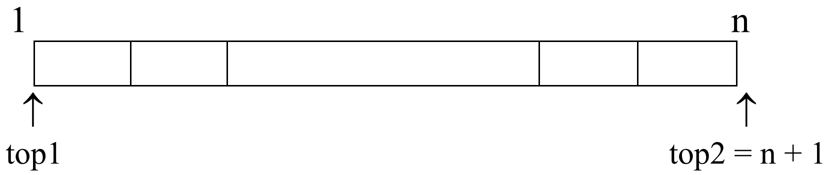
پشته‌های چندگانه

پشته دوگانه : برای پیاده‌سازی دو پشته در یک آرایه نیاز به دو متغیر top1 برای نشان دادن بالاترین عنصر پشته اول و top2 برای بالاترین عنصر پشته دوم داریم. top1 و top2 در جهت عکس یکدیگر حرکت می‌کنند. مقدار اولیه $top1 = 0$ و مقدار اولیه $top2 = n + 1$ است.

$top1 = 0$ ===➔ پشته ۱ خالی است

$top2 = n + 1$ ===➔ پشته ۲ خالی است

$top2 = top1 + 1$ ===➔ آرایه پر است



دنباله‌های قابل قبول در پشته‌ها

هرگاه اعدادی را به صورت مرتب شده صعودی داشته باشیم و بخواهیم اعداد دیگری را از آن استخراج کنیم باید این قانون را راعایت کنیم که اعداد بزرگتر در صورتیکه اعداد کوچکتر در پشته قرار نگرفته‌اند حق قرار گرفتن در پشته را ندارند. مثلاً اعداد ۴ , ۳ , ۲ , ۱ را در نظر می‌گیریم. عدد ۲ در صورتی می‌تواند push شود که حتماً عدد ۱ push شده باشد و عدد ۳ زمانی می‌تواند push شود که اعدا ۱ و ۲ قبلاً push شده باشند.

مثال : چهار عدد ۴ , ۳ , ۲ , ۱ را داریم. کدامیک از اعداد زیر را می‌توانیم تولید کنیم ؟

۲ ۱ ۳ ۴	۳ ۱ ۴ ۲	۳ ۲ ۴ ۱	۴ ۲ ۳ ۱	۴ ۳ ۱ ۲
push 1 push 2 pop 2 pop 1 push 3 pop 3 push 4 pop 4	push 1 push 2 push 3 pop 3 قابل تولید نیست	push 1 push 2 push 3 pop 3 pop 2 push 4 pop 4 pop 1	push 1 push 2 push 3 push 4 pop 4 قابل تولید نیست	push 1 push 2 push 3 push 4 pop 4 pop 3 قابل تولید نیست

اگر اعداد به صورت صعودی داده شوند (۴ ۳ ۲ ۱) و سه عدد a و b و c داشته باشیم بطوریکه $b < c < a$ در اینصورت دنباله abc قابل تولید نیست.

ارزشیابی عبارت

اولویت عملگر

بطور کلی اگر عبارت $a \times b + c / d$ را داشته باشیم اولویت عملگرها را به صورت زیر می‌نویسیم :

1. ( )
2. Not , – (قرینه) , توان
3. and , × , / , mod
4. OR , + , –
5. < , > , < = , > = , < > (! =)

نکته : بین عملگرهایی که اولویت مساوی دارند عملگری زودتر محاسبه می‌گردد که سمت چپ باشد.

روش نمایش عبارات محاسباتی

میانوندی	infix	$a + b$
پسوندی	postfix	$ab +$
پیشوندی	prefix	$+ ab$

تبدیل عبارات میانوندی به پسوندی و پیشوندی بدون استفاده از پشته

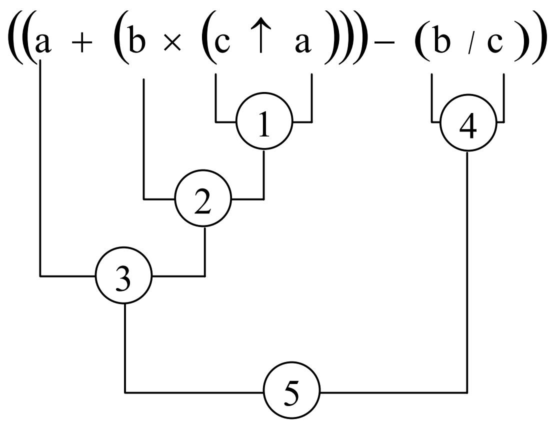
۱- پرانتز گذاری

۲- برای تبدیل به پیشوندی , درون هر پرانتز عملگر را به سمت چپ منتقل می‌کنیم.

۳- برای تبدیل به پسوندی , درون هر پرانتز عمگلر را به سمت راست منتقل می‌کنیم.

۴- پرانتزها را حذف می‌کنیم.

مثال :



$$\text{postfix} = \big(a(b(ca)\uparrow \times + (bc)/\big) = abca \uparrow \times + bc / -$$

$$\text{prefix} = - + a \times b \uparrow ca / bc$$

استفاده از پشته در تبدیل عبارات infix به postfix

۱- عبارت infix را از چپ به راست پیمایش می‌کنیم.

۲- پرانتز باز را در پشته push می‌کنیم.

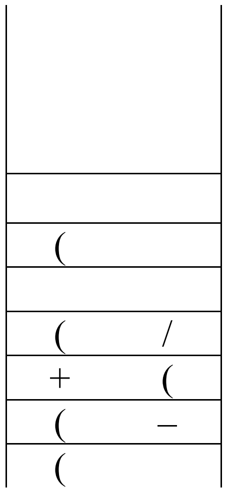
۳- عملوندها را در خروجی می‌نویسیم.

۴- در صورتیکه به یک عملگر رسیدیم اگر top پشته دارای عملگری با اولویت بیشتر یا مساوی نبود آنرا push می‌کنیم در غیر اینصورت عملگر top پشته را pop کرده و در خروجی می‌نویسیم.

۵- هرگاه به پرانتز بسته رسیدیم آنقدر pop می‌کنیم تا به اولین پرانتز باز برسیم.

مثال :

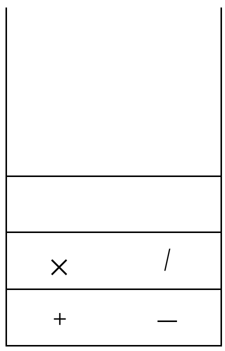
$$((a + (b \times (c \uparrow a))) - (b / c))$$



$abca \uparrow \times + bc / -$

مثال : با استفاده از پشته , عبارت زیر را به صورت postfix بنویسید.

$$a + b \times c \uparrow a - b / c$$



$abca \uparrow \times + bc / -$

مثال : با استفاده از پشته , عبارت زیر را به صورت postfix بنویسید.

$a+(b\times c)\uparrow a-b/c$

(	$\uparrow$	/
+	–	

$abc\times a\uparrow+bc/-$

بـرای تـبدیل عـبارات infix بـه عبارات prefix از دو پشته استفاده می‌کنیم. یکی پشته عملوندها و دیگری پشته عملگرها. push کردن و pop کردن در پشته عملگرها مانند تبدیل infix به postfix است. با رسیدن به هر عملوند آنرا در پشته عملوندها push می‌کنیم. در صورت pop شدن هر عملگر از پشته عملگرها , دو عملوند بالای پشته عملوندها pop شده و با عملگر مربوطه به شکل prefix در پشته عملوندها push می‌شود. بقیه قوانین مانند قوانین infix به postfix است.

مثال : عبارت زیر را بوسیله پشتبه از infix به prefix بنویسید.

$a+(b\times c)\uparrow d/a-c\times b$

$\times$			
(	$\uparrow$	/	$\times$
+	-		

c	d	a	d
b	$\times bc$	$\uparrow\times bcd$	$/\uparrow\times bcda$
a	$+a/\uparrow\times bcda$		

$-+a/\uparrow\times bcda\times cd$

مثال : عبارت infix زیر را بوسیله دو پشته به prefix تبدیل کنید.

$a+b\times c\uparrow(2-b)\times c/(d+a)$

-		
(	+	
$\uparrow$	(	
$\times$	$\times$	/
+		

b				
2	-2b	a		
c	$\uparrow$c-2b	c	d	+da
b	$\times b\uparrow$c-2b	$\times\times b\uparrow$c-2b	$/\times\times b\uparrow$c-2bc+da	
a	$+a/\times\times b\uparrow$c-2bc+da			

تبدیل عبارت postfix به infix

بـا اسـتفاده از یـک Stack مـی‌توان رشته postfix ورودی را به infix تبدیل کرد. برای این منظور

رشــته postfix را از چــپ پـردازش می‌کنیم. هر عملوند درون پشته push می‌شود. با رسیدن به هر عملگـر , دو عنصـر پشـته pop شده و بصورت infix نوشته می‌شود. سپس عبارت infix تولید شده درون پشــته push مــی‌شــود. در پایــان پردازش رشته ورودی , پشته حاوی یک عنصر است که شکل infix مــورد نظر می‌باشد. خروجی infix باید لزوماً پرانتزگذاری شده باشد. عملوند top پشته سمت راست عملگر نوشته می‌شود.

مثال :

$abca\uparrow\times+bc/-$

a			
c	$c\uparrow a$	c	
b	$b\times(c\uparrow a)$	b	b/c
a	$a+(b\times(c\uparrow a))$	$(a+b\times(c\uparrow a))-(b/c))$	

تبدیل عبارات prefix به infix

بـرای تـبدیل عبارت prefix به infix باید رشته ورودی را از سمت راست پردازش کنیم. مانند روش قبل عملوندها در پشته push می‌شوند و با رسیدن به هر عملگر , دو عملوند بالای پشته pop شده و بـا عملگـر ورودی بصـورت infix نوشته می‌شود و نتیجه در پشته push می‌شود. عملوند top پشته سمت چپ عملگر قرار می‌گیرد.

$-+a\times b\uparrow ca/bc$

c	b	a		
b	a	$(c\uparrow a)$	$(b\times(c\uparrow a))$	$(a+(b\times(c\uparrow a))$
c	(b/c)	$(a+b\times(c\uparrow a)-(b/c)$		

تبدیل عبارات postfix به prefix و بالعکس

بـرای تبدیل عبارات postfix و prefix به همدیگر می‌توان آنها را ابتدا تبدیل به حالت میانی infix کـرده و سـپس عبارت infix را با روشهای گفته شده به حالت مطلوب تبدیل نمود. همچنین می‌توان بصـورت مسـتقیم عـبارت postfix و prefix را با استفاده از الگوریتم قبلی به یکدیگر تبدیل کرد. با این تفاوت که هنگامیکه در حین پردازش رشته ورودی به یک عملگر رسیدیم , دو عملوند بالای پشته pop شــده و بــه جــای اینکه به infix با عملگر ورودی در پشته push شوند به هر کدام از حالتهای مورد نظر postfix یا prefix در پشته push می‌شوند.

صف (queue)

صـف لیستی است که عمل افزودن داده‌ها درون آن از یک طرف لیست یا انتهای لیست و عمل حذف داده‌هـا از سـمت دیگر یا ابتدای لیست انجام می‌شود. صف را لیست FIFO (First In First Out) مـی‌نامند. زیرا اولین عنصر ورودی , اولین عنصر خروجی از صف نیز هست. در ساختمان داده صف دو متغیـر front و rear بـه تـرتیب بـرای نشان دادن جلو و انتهای صف بکار می‌روند. صف را می‌توان با استفاده از آرایه‌ها یا لیست‌های پیوندی پیاده سازی کرد.

اگر صف را آرایه‌ای n عضوی از عناصر بدانیم مقادیر front و rear می‌تواند از صفر تا n تغییر کند که برای صف در ابتدا مقادیر اولیه صفر را برای front و rear تعریف می‌کنیم. front = rear = 0

در صورتیکه متغیر front با rear برابر باشد صف خالی است و در صورتیکه rear برابر با n باشد صف پر است.

rear = n ===➔ صف پر است

front = rear ===➔ صف خالی است

دو عمـل اصـلی بـرای صف , حذف کردن داده‌ها از صف و افزودن داده‌ها به صف است که به ترتیب با delqueue و Addqueue نمـایش می‌دهیم. تابع Addqueue(x) به این معنی است که عنصر x بـه انـتهای صـف اضـافه شده است و delqueue نیز مقدار جلوی صف را برداشته و در متغیر x قرار می‌دهد. x = delqueue

پیاده سازی تابع Addqueue و delqueue از صف

queue : Array [1 .. n] of item

برای حذف کردن	برای اضافه کردن
int delqueue ( )	void Addqueue (int x)

```
void Addqueue (int x)
{
      if (rear = = n)
            C out << " صف پر است " ;
      else
      {
            rear + + ;
            queue[rear] = x ;
      }
}
```

```
int delqueue ( )
{
      if (front = = rear)
      {
            C out << " صف خالی است " ;
            return 0 ;
      }
      else
      {
            front + + ;
            x = queue[front] ;
            return x ;
      }
}
```

مثال : با استفاده از توابع صفحه قبل مقادیر نهایی A و B و C را بدست آورید.

$A = 5$ $B = 10$ $C = 2$ $n = 4$

Addqueue (A + B)

Addqueue (C)

Addqueue (B × C)

A = delqueue ( )

B = delqueue ( )

Addqueue (B – A)

15	2	20	–13
1	2	3	4

Rear	Front	A	B	C
0	0	5	10	**2**
1	1	**15**	**2**	
2	2			
3				
4				

پس بنابراین داریم : **$A = 15$** **$B = 2$** **$C = 2$**

توابع Addqueue و delqueue به صورتیکه نوشته شد یک صف خطی را پیاده‌سازی می‌کنند. مشکل صف خطی این است که تنها یک بار قابل پر شدن است و در صورتیکه عناصر آن حذف شوند نیز با پیغام « صف پر است » مواجه می‌شوید به همین دلیل صف را بصورت حلقوی تعریف می‌کنیم. در صف حلقوی (دوار) rear و front بعد از رسیدن به آخرین مقدار خود در صورت وجود شرایط لازم مجدداً مقادیر اولیه را می‌توانند بگیرند. صف حلقوی n عضوی را بصورت آرایه صفر تا $n - 1$ تعریف می‌کنیم.

queue : Array [0 .. n – 1] of item

در این حالت وقتی $rear = n - 1$ عنصر بعدی در queue[0] قرار می‌گیرد.

در صف حلقوی $front = rear$ به معنای خالی بودن صف است ولی شرط پر بودن صف بدین ترتیب تغییر می‌یابد.

$front = (rear + 1) \bmod n$ ===➔ شرط پر بودن

$front = rear$ ===➔ صف خالی است

برای اضافه کردن به صف حلقوی , rear یکی اضافه می‌شود و در صورتیکه $rear = n - 1$ باید صفر بشود. بدین منظور rear را با رابطه زیر در هر شرایطی مقداردهی می‌کنند.

$$rear = (rear + 1) \bmod n$$

این مسئله برای front نیز برقرار است.

$$front = (front + 1) \bmod n$$

برای حذف کردن

```
int delqueue ( )
{
    if (front = = rear)
    {
        C out << " صف خالی است " ;
        return 0 ;
    }
    else
    {
        front = (front + 1) mod n
        x = queue[front] ;
    }
}
```

برای اضافه کردن

```
void Addqueue (int x)
{
    rear = (rear + 1) mod n
    if (front = = rear)
        C out << " صف پر است " ;
    else
        queue[rear] = x ;
}
```

مثال :

			0	1	2	3
Addqueue [50]	r = 1	f = 0		50		
Addqueue [20]	r = 2	f = 0		50	20	
Addqueue [30]	r = 3	f = 0		50	20	30
delqueue ( )	r = 3	f = 1		50	20	30
Addqueue [10]	r = 0	f = 1	10	~~50~~	20	30

مثال : عبارت زیر را بصورت prefix و postfix بنویسید.

$$\sqrt{a^2 - bc} \Rightarrow (a \uparrow 2 - b \times c) \uparrow (1/2)$$

$$postfix = a2 \uparrow bc \times -12/\uparrow$$

$$prefix = \uparrow - \uparrow a2 \times bc/12$$

مرتب‌سازی

در مرتب‌سازی تعدادی عنصر که از ورودی داده شده‌اند را بر اساس کلیدشان بصورت صعودی یا نزولی مرتب می‌کنیم.

مرتب‌سازی انتخابی (Selection Sort)

در مرتب‌سازی انتخابی یک آرایه n عنصری (A[1..n]) , n − 1 بار پیمایش می‌شود. در هر پیمایش بزرگـتـرین عنصر در محل درست خود یعنی انتهای آرایه قرار می‌گیرد. با این روش آرایه از انتها مرتب می‌شود. در مرتب‌سازی انتخابی می‌توان با انتخاب کوچکترین عنصر در هر پیمایش و قرار دادن آن در محل درست خود یعنی ابتدای آرایه در هر پیمایش , مرتب‌سازی را از ابتدای لیست انجام داد.

مثال :

10	5	8	20	25	12	
10	5	8	20	12	25	پویش اول
10	5	8	12	20	25	پویش دوم
10	5	8	12	20	25	پویش سوم
8	5	10	12	20	25	پویش چهارم
5	8	10	12	20	25	پویش پنجم

برنامه کلی مرتب‌سازی انتخابی به شرح ذیل می‌باشد :

```
for (i = n ; i > 1 ; – – 1)
{
        max = A[1] ;
        index = 1 ;
        for (i = 2 ; j < = i ; + + j)
        if (A[j] > max)
        {
                max = A[j] ;
                index = j ;
        }
        A[index] = A[i] ;
        A[i] = max ;
}
```

مثال :

3	2	5	1
1	2	3	4

i	j	n	max	index
4	2	4	3	1
3	4		5	3
	4		3	1
	2			
	3			

نکـته : در مـرتب‌سازی انتخابی , حداکثر و حداقل n^2 مقایسه داریم. حداقل جابجایی صفر و حداکثر جابجایی نیز n بار خواهد بود.

مرتب‌سازی حبابی (Bubble Sort)

در مرتب‌سازی حبابی یک آرایه n عنصری (A[1..n]) , n − 1 بار پیمایش می‌شود و در هر پیمایش دو عنصــر متوالــی با یکدیگر مقایسه شده که در صورت لزوم جابجا خواهند شد. در هر پیمایش , طول آرایه پیمایش شده نسبت به مرحله قبل یکی کم می‌شود.

10	30	50	40	90	20	
10	30	~~50~~	~~40~~	~~90~~	~~20~~	مرحله اول
		40	50	20	90	
10	30	40	~~50~~	~~20~~	90	مرحله دوم
			20	50		
10	30	~~40~~	~~20~~	50	90	مرحله سوم
		20	40			
10	~~30~~	~~20~~	40	50	90	مرحله چهارم
	20	30				
10	20	30	40	50	90	مرحله پنجم

مرتب‌سازی از انتهای لیست

```
for (i = 1 ; i < n ; + + i)
        for (j = 1 ; j < = n ; + + j)
        if (A[j]) > A[j + 1])
        swap (A[j] , A[j + 1]) ;
```

مرتب‌سازی از ابتدای لیست

```
for (i = 1 ; i < n ; + + i)
        for (j = n ; j > = i ; – – j)
        if (A[j]) < A[j – 1])
        swap (A[j] , A[j – 1]) ;
```

مثال :

i	j	n
1	4	4
2	3	
3	2	
	1	
	4	

5	3	7	2

1 2 3 4

پویش اول

2	5	3	7

پویش دوم

2	3	5	7

« الگوریتم متعادل است »

در هر پویش امکان n^2 جابجایی وجود دارد. حداقل تعداد جابجایی نیز صفر است.

مرتب‌سازی حبابی بهینه شده

```
for (i = 1 ; i < = x ; + + i)
{
    sw = 0 ;
    for (j = 1 ; j < n – 1 ; + + j)
    if (A[j] > A[j + 1])
    {
        sw = 1 ;
        swap (A[j] , A[j + 1]) ;
    }
    if (sw = = 0) break ;
}
```

مرتب‌سازی دَرجی (Insertion Sort)

در مـرتب‌سـازی درجـی فـرض شده است که $i - 1$ عنصر اول لیست مرتب هستند. پس عنصر iام در جای صحیح خود قرار دارد.

```
For (i = 2 ; i < = n ; + + i)
{
    y = A[i] ;
    j = i – 1 ;
    while (j > 0 & & (y < A[j]))
    {
        A[j + 1] = A[j] ;
        j = j – 1 ;
    }
    A[j + 1] = y ;
}
```

i	n	y	j
2	6	60	1
3		40	2
4		20	1
5		10	0
			3
			2
			1
			0
			4
			3
			2
			1
			0

50	60	40	20	10	30
1	2	3	4	5	6

50	60	40			
50	40	60			
40	50	60			
10	40	50	60		

در این جابجایی حداقل n تا پویش داریم و در بهترین حالت نیز جابجایی نداریم.

مثال : آرایه زیر را به روش درجی مرتب کنید.

	1	2	3	4	5
A	4	8	5	2	6
	4	8			
	4	5	8		
	2	4	5	8	
	2	4	5	6	8

n	i	j	y
5	2	1	8
	3	2	5
	4	1	2
		3	6
		2	
		1	
		0	
		4	
		3	

مرتب سازی ادغامی (merge sort)

مرتب سازی ادغامی مبتنی بر تقسیم و حل است و در روش ادغامی لیست n عنصری تبدیل به لیست‌های یک عنصری شده (با تقسیمات متوالی بر ۲) و سپس لیست‌های یک عنصری که مرتب هستند ادغام شده و لیست‌های دو تایی مرتب تشکیل می‌دهند و سپس لیست‌های دو تایی مرتب شده با هم ادغام می‌شوند و این فرآیند تا تولید لیست اولیه به صورت مرتب ادامه پیدا می‌کند که این حالت نیز بصورت بازگشتی است.

11 2 20 18 1 8 7 12 17 5

11 2 20 18 1 | 8 7 12 17 5

11 2 20 | 18 1 | 8 7 12 | 17 5

11 2 | 20 | 18 | 1 | 8 7 | 12 | 17 | 5

11 | 2 | 1 18 | 8 | 7 | 5 17

2 11 | 7 8

2 11 20 | 7 8 12

1 2 11 18 20 | 5 7 8 12 17

1 2 5 7 8 11 12 17 18 20

```
Void mergsort (int L , int U)
{
        int i ;
        if ( L < U )
        {
                i = ( L + U ) / 2 ;
                mergsort ( L , i ) ;
                mergsort ( i + 1 , U ) ;
                merg ( L , i , U ) ;
        }
}
```

مثال :

L = 3	U = 3	L = 4	U = 4
L = 3	U = 3	i = 3	
L = 1	U = 1	L = 2	U = 2
L = 1	U = 2	i = 1	
L = 1	U = 4	i = 2	

5	1	7	2

5	1		7	2	

5   1   7   2

1	5		2	7

1	2	5	7

مثال : آرایه زیر را بوسیله merge sort مرتب کنید.

L = 4	U = 4	
L = 3	U = 4	
L = 3	U = 4	i = 3
L = 2	U = 2	
L = 1	U = 1	
L = 1	U = 2	i = 1
L = 1	U = 4	i = 2

	1	2	3	4
A	5	3	1	4

Merge sort (1 , 4)

1	2	3	4
3	5	1	4

1	3	4	5

تمرین : برنامه‌ای بنویسید که دو آرایه مرتب را بگیرد و در هم ادغام کند.

مرتب سازی سریع (quick sort)

در روش مـرتب‌سازی سریع , یک عنصر بعنوان عنصر محوری در نظر گرفته می‌شود که عنصر محوری را معمـولاً اولـین عنصـر آرایه در نظر می‌گیرند. بعد از اولین پیمایش , عنصر محوری در محل مناسب خـود در لیسـت قـرار مـی‌گیـرد و لیسـت به دو بخش مجزا تقسیم می‌گردد. عناصر سمت چپ عنصر محـوری که کوچکتر از عنصر محوری هستند و عناصر سمت راست عنصر محوری که همگی بزرگتر از عنصـر محـوری مـی‌باشند. این عمل مجدداً بر روی هر یک از دو بخش انجام می‌شود تا به لیست‌های یـک عنصری مرتب برسیم. متوسط زمان اجرای این الگوریتم $0(n \text{ Log } n)$ و بدترین زمان اجرای آن $0\ (n^2)$ می‌باشد که زمانی اتفاق می‌افتد که آرایه از پیش مرتب باشد.

1	2	3	4	5	6	7	8	9
12	10	2	8	15	7	3	1	14
محوری (Pivot)						j		I

3	10	2	8	1	7	12	15	14
	i	j	i	j				

2	1	3	8	10	7	12	14	15

```
Void quicksort (int L , int U )
{
        int i , j , pivot ;
        if ( L < U )
        {
                i = L + 1 ; j = U ; pivot = A[L] ;
                while ( i < j )
                {
                        while ( A [i] < pivot ) i + + ;
                        while ( A[j] > pivot ) j - - ;
                        if ( i < j ) swap ( A[i] , A[j]) ;
                }
                swap ( A[L] , A[j] ) ;
                quicksort ( L , j – 1 ) ;
                quicksort ( j + 1 , U ) ;
        }
}
```

لیست پیوندی (Link List)

لیست‌ها ساختمان داده‌ای هستند که اندازه آنها بصورت پویا تغییر می‌کند. پیمایش در لیست‌های پیوندی بصورت ترتیبی (خطی) است. بنابراین برای حذف , اضافه یا جستجو باید لیست را از ابتدا بصورت خطی پیمایش کرد. هر گره (node) در لیست پیوندی ساختاری با دو فیلد اصلی دارد. یکی فیلد داده که می‌تواند از هر نوع داده‌ای باشد و دیگری فیلد آدرس که به محل عنصر بعدی در لیست پیوندی اشاره می‌کند. در ساختمان داده لیست پیوندی اعمال اصلی حذف داده از لیست , اضافه کردن داده به لیست و جستجو در لیست انجام می‌شود. عنصر اول لیست پیوندی را هد (Head) یا هدر (Header) لیست می‌گویند و معمولاً این عنصر را برای سادگی پیمایش خالی نگه می‌دارند.

برای افزودن داده جدید به لیست پیوندی ۴ عمل اصلی انجام می‌گیرد.

۱- تشکیل گره (node) جدید بر اساس اطلاعات جدید افزوده شدنی

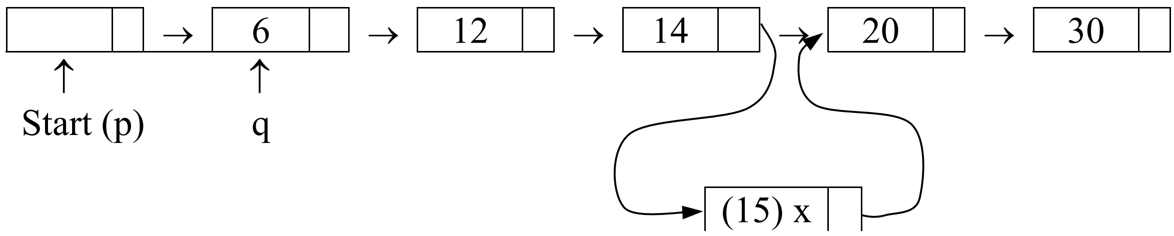
۲- بدست آوردن آدرس گره‌ای که باید قبل از گره جدید قرار گیرد (مثلاً گره p)

۳- آدرس گره جدید که به محل اشاره‌گر p اشاره می‌کند.

۴- آدرس گره p را به محل new node تغییر می‌دهیم.

```
Void insert ( int x , node * start )
{
        node * p , * q , * new node ;
        q = start → next ;
        p = start ; new node = new ( node ) ; new node → data = x ;
        while ( q → data < newnode → data)
                             x
        {
                p = q ;
                q = q → next ;
        }
        new node → next = p → next ;
        p → next = new node ;
}
```

مثال : می‌خواهیم گره ۱۵ را به لیست اضافه کنیم :



p	q
Start	6
6	12
12	14
14	20

مراحل حذف یک گره از لیست پیوندی

۱- یافتن گره قبل از گره مشخص شده برای حذف (q)

۲- تغییر دادن $q \rightarrow next$ به $p \rightarrow next$ (p حذف می‌شود)

۳- آزاد کردن حافظه‌ای که برای p در نظر گرفته‌ایم.

```
void dellinklist ( int x , node * store )
{
        node * p , * q ;
        p = start ;
        q = p ;
        while ( p → data ! = x )
        {
                q = p ;
                p = p → next ;
        }
        q → next = p → next ;
        delete (p) ;
}
```

مثال : گره شماره 3 را حذف کنید.

5		→	8		→	3		→	2	\

5		→	8		→	2	\

p	q
5	5
8	8
3	2

تمرین : برنامه‌های زیر چه کاری انجام می‌دهند؟

```
node * f ( int x , node * start )
{
        node * p ;
        p = start ;
        while ( p ) ;
        if ( p → data ! = x && p ) p = p → next ;
        else return p ;
}
void g ( node * start )
{
        if (start ! = Null )
        {
                C out << start → data ;
                g ( start → next ) ;
        }
}
```

جــواب : در قسمت اول یعنی f عمل جستجو را انجام می‌دهد و در قسمت دوم یعنی g عناصر لیست را به ترتیب چاپ می‌کند.

<u>لیست پیوندی چرخشی</u>

اگر اشاره‌گر عنصر انتهای لیست به جای Null به هد لیست اشاره‌گر کند (start) لیست ما تبدیل به لیست تک پیوندی چرخشی می‌شود.

تمرین : عمل حذف و اضافه را در یک لیست پیوندی چرخشی بنویسید.

لیست دو پیوندی

در لیست دو پیوندی سه بخش وجود دارد.

۱- بخش data

۲- بخش سمت راست که به گره بعدی اشاره می‌کند.

۳- بخش سمت چپ که به گره قبلی اشاره می‌کند.

```
struct Linklist
{
        struct Linklist * left ;
        data ;
        struct Linklist * right ;
}
typedeg struct Linklist node
node * p , * q ;
```

مراحل اضافه کردن داده به لیست دو پیوندی

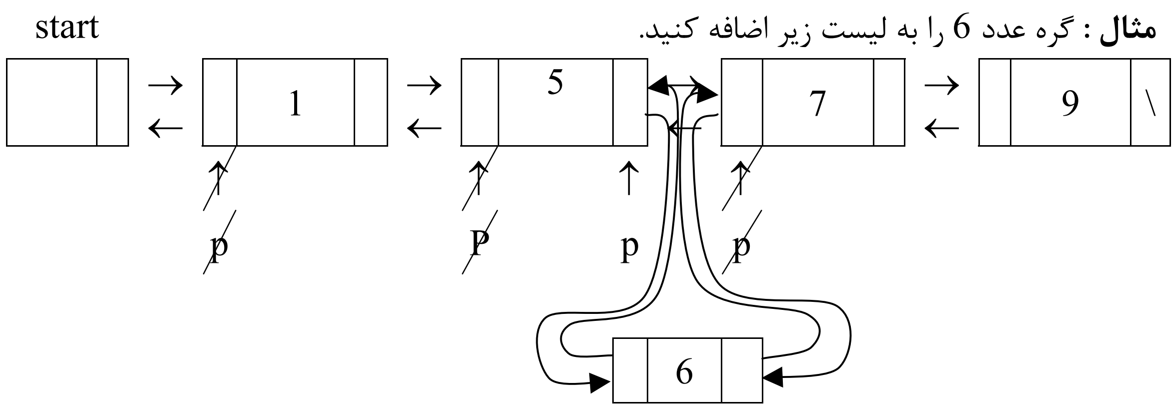
۱- تشکیل گره جدید با استفاده از اطلاعات اضافه شدنی (new node)

۲- پیدا کردن محل درج گره جدید

۳- انتساب مقادیر مورد نظر به بخشهای آدرس چپ و آدرس راست گره‌های p و new node

```
void insert (int x , node * start )
{
          node * newnode , * p ;
          newnode = new (node) ;
مرحله 1   newnode → data = x ;
          newnode → right = Null ;
          newnode → left = Null ;
          p = start → right ;
مرحله 2   while ( p → data < x ) p = p → right ;
          p = p → left ;
          ( p → right ) → left = newnode ;
مرحله 3   newnode → left = p ;
          newnode → right = p → right ;
          p → right = newnode ;
}
```

مثال : گره عدد 6 را به لیست زیر اضافه کنید.



حذف از لیست دو پیوندی

```
void delete ( int x , node * start )
{
        node *p ;
        p = start → right ;
        while ( p && p → data < x ) p = p → right ;
        if ( p = = Null | | p → data > x ) return “ داده در لیست نبوده است “ ;
        ( p → Left ) → right = p → right ;
        ( p → right ) → Left = p → Left ;
        delete ( p ) ;
}
```

نمایش چندجمله‌ایها با استفاده از لیست‌های پیوندی

همانطور که چندجمله‌ایها بوسیله آرایه‌ها قابل نمایش بودند با استفاده از لیست‌های پیوندی نیز می‌توان چندجمله‌ایها را نمایش داد. برای این منظور باید ساختار متناسب با چندجمله‌ای تولید کرد. این ساختار شامل یک توان , یک ضریب و یک اشاره‌گر به جمله بعدی چند جمله‌ای است. به این ترتیب در نمایش چندجمله‌ایها بوسیله لیست‌های پیوندی از حافظه بصورت بهتری استفاده شده است ولی چون پیمایش در لیست‌های پیوندی , ترتیبی یا خطی است زمان محاسبات روی چندجمله‌ایها در صورت استفاده از لیست پیوندی افزایش می‌یابد.

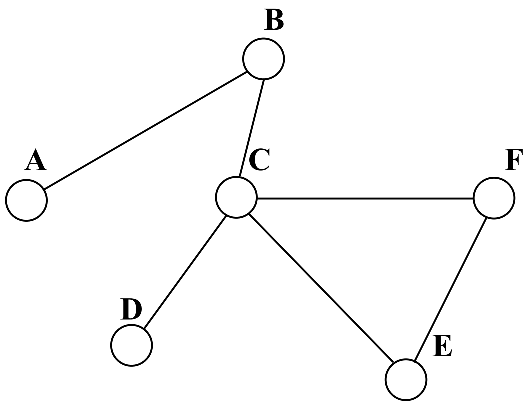
2	1		→	3	5		→	2	0	

$$2x + 3x^5 + 2$$

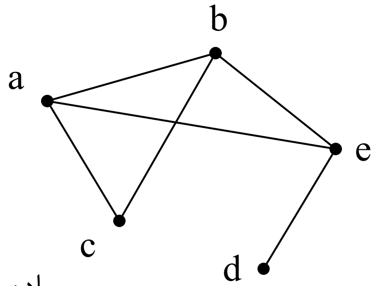
گراف (Graph)

گراف G مجموعه‌ای است از گره‌ها (گره = Vertex) که بوسیله یالهایی (لبه , یال = Edge) به یکدیگر متصل شده‌اند. گراف‌ها می‌توانند جهت‌دار یا غیر جهت‌دار باشند. اگر هر یال در گراف دارای جهت مشخصی باشد بدین معنی که مبدأ و مقصد آن مشخص بوده و غیرقابل جابجایی , این گراف , گراف جهت‌دار خواهد بود. هر دو گره که مستقیماً با یک یال به هم متصل باشند را دو گره مجاور یا همسایه گویند (Adjacement). گرافی که بین تمام گره‌ها مسیر مستقیم وجود داشته باشد را گراف کامل می‌گویند. گراف کامل با n گره را با k_n نمایش می‌دهند.

تعداد کل یالها در گراف کامل غیرجهت‌دار $n(n-\frac{1}{2})$ و در گراف کامل جهت‌دار $n(n-1)$ خواهد بود.



اگر از گره A با عبور از تعدادی یال و گره میانی به گره B در گراف برسیم گوئیم بین A و B یک مسیر از طول n وجود دارد. n تعداد یالهایی است که در مسیر پیموده می‌شوند. اگر در مسیری گره‌ای بیش از یکبار دیده شده باشد آن مسیر , مسیر غیر ساده است در غیر اینصورت مسیر ساده خواهد بود. اگر در یک مسیر , گره مبدأ و گره مقصد بر هم منطبق بودند یا گره مبدأ همان گره مقصد بود , گراف دارای سیکل یا دور است.



مسیر ساده = c - b - e

مسیر غیر ساده = c - a - e - b

سیکل = a - b - e - a

فاصله دو گره در گراف برابر است با کوتاهترین مسیر بین آن دو گره

قطر گراف برابر است با بزرگترین فاصله بین دو گره در گراف

گراف متصل : گرافی که بین هر دو گره مسیری وجود داشته باشد را گراف متصل گویند.

گراف غیر متصل : گرافی است که حداقل بین دو گره آن هیچ مسیر وجود نداشته باشد.

شرط لازم برای اینکه گرافی با n گره متصل باشد این است که حداقل n - 1 یال وجود داشته باشد.

گراف تهی : گرافی است که مجموعه‌ای از گره‌ها باشد و هیچ یالی بین گره‌ها وجود نداشته باشد.

درجه هر گره : تعداد یالهایی که از یک گره عبور می‌کند را درجه آن گره گویند.

تذکر : درجه خروجی برای گراف‌های جهت‌دار تعداد یالهای خارج شده از یک گره را نشان می‌دهد و درجه ورودی برای گراف‌های جهت‌دار تعداد یالهایی که به یک گره وارد شده‌اند می‌باشد.

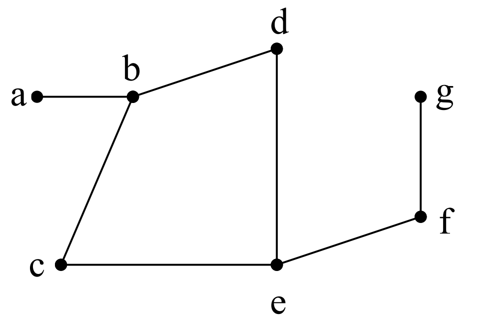
مجموع درجات گره‌ها در گراف‌های بدون جهت دو برابر تعداد یالهاست و مجموع درجات ورودی یا مجموع درجات خروجی در گراف‌های جهت‌دار تعداد یالها را نشان می‌دهد.

روشهای نمایش گراف‌ها

۱- ماتریس مجاورتی

ماتریس مجاورتی روشی عمومی برای پیاده‌سازی گراف‌ها است. در این روش از یک ماتریس $n \times n$ برای نمایش گراف استفاده می‌کنیم که n تعداد گره‌های گراف است.

گراف وزن‌دار : گراف وزن دار گرافی است که به هر یال آن یک وزن (ارزش) منتصب شده باشد.



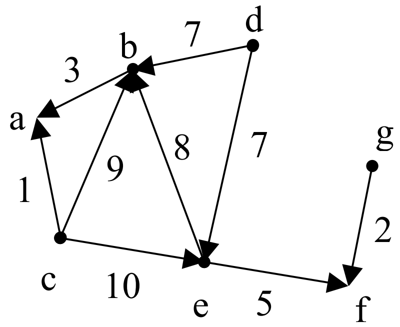
$$
\begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \\ g \end{array}
\begin{array}{c} \begin{array}{ccccccc} a & b & c & d & e & f & g \end{array} \\
\begin{bmatrix}
0 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 1 & 1 & 0 & 0 \\
1 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 1 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0
\end{bmatrix} \end{array}
$$

7×7

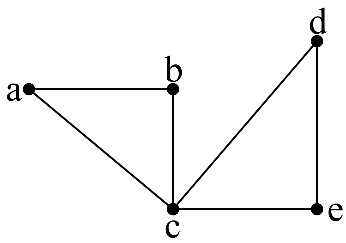
$A[i,j] = 1$ یا w if i, j متصل باشند

$A[i,j] = 0$ if i و j متصل نباشند

نکته : ماتریس گراف‌های غیر جهت‌دار , ماتریس متقارن است.

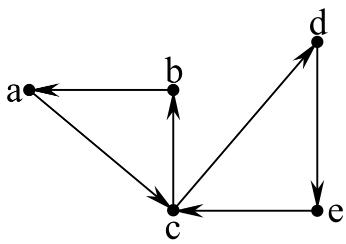


	a	b	c	d	e	f	g
a	0	0	0	0	0	0	0
b	3	0	0	0	0	0	0
c	1	9	0	0	10	0	0
d	0	7	0	0	7	0	0
e	0	8	0	0	0	5	0
f	0	0	0	0	0	0	0
g	0	0	0	0	0	2	0



A =

	a	b	c	d	e
a	0	1	1	0	0
b	1	0	1	0	0
c	1	1	0	1	1
d	0	0	1	0	1
e	0	0	1	1	0



	a	b	c	d	e
a	0	0	1	0	0
b	1	0	0	0	0
c	0	1	0	1	0
d	0	0	0	0	1
e	0	0	1	0	0

مجموع سطری یا ستونی هر گره در ماتریس برای گراف‌های بدون جهت برابر با درجه هر گره است و مجموع سطری هر گره در ماتریس برای گراف‌های جهت‌دار برابر است با درجه خروجی هر گره و مجموع ستونی هر گره در ماتریس برای گراف‌های جهت‌دار برابر است با درجه ورودی هر گره.

تمرین : ماتریس A صفحه قبل را در نظر گرفته و ماتریس‌های A^2 و A^3 آنرا حساب کنید.

$$\begin{array}{c} \\ a\\ b\\ c\\ d\\ e \end{array}\begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix} \end{array} \times \begin{array}{c} \\ a\\ b\\ c\\ d\\ e \end{array}\begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix} \end{array} \Rightarrow A^2 = \begin{array}{c} \\ a\\ b\\ c\\ d\\ e \end{array}\begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{bmatrix} 2 & 1 & 1 & 1 & 1 \\ 1 & 2 & 1 & 1 & 1 \\ 1 & 1 & 4 & 1 & 1 \\ 1 & 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 1 & 2 \end{bmatrix} \end{array}$$

$$\begin{array}{c} \\ a\\ b\\ c\\ d\\ e \end{array}\begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{bmatrix} 2 & 1 & 1 & 1 & 1 \\ 1 & 2 & 1 & 1 & 1 \\ 1 & 1 & 4 & 1 & 1 \\ 1 & 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 1 & 2 \end{bmatrix} \end{array} \times \begin{array}{c} \\ a\\ b\\ c\\ d\\ e \end{array}\begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix} \end{array} \Rightarrow A^3 = \begin{array}{c} \\ a\\ b\\ c\\ d\\ e \end{array}\begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{bmatrix} 2 & 3 & 5 & 2 & 2 \\ 3 & 2 & 5 & 2 & 2 \\ 5 & 5 & 4 & \boxed{5} & 5 \\ 2 & 2 & 5 & 2 & 3 \\ 2 & 2 & 5 & 3 & 2 \end{bmatrix} \end{array}$$

حال بـرای ایـنکه ماتـریس بدست آمده را امتحان کنیم تا درست بودن آن ثابت گردد یک عدد را در نظر گرفته (مثلاً عدد ۵ که از خانه‌های c - d بدست می‌آید) و حالات موجود را بررسی می‌کنیم :

c - d - c - d

c - b - c - d

c - e - c - d

c - a - c - d

c - d - e - d

نکته : قطر اصلی ماتریس A^2 درجه هر گره را نشان می‌دهد.

عنصر $[\,i\,,j\,]$ در ماتریس A^k نشان می‌دهد که چه تعداد مسیر به طول k بین i و j وجود دارد.

فضـای لازم برای نمایش دادن یک گراف با استفاده از ماتریس مجاورتی از مرتبه n^2 است که n تعداد گره‌های گراف می‌باشد.

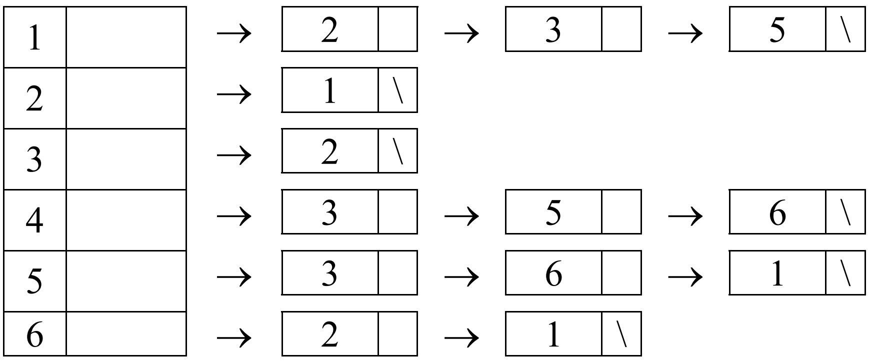
۲- لیست همجواری

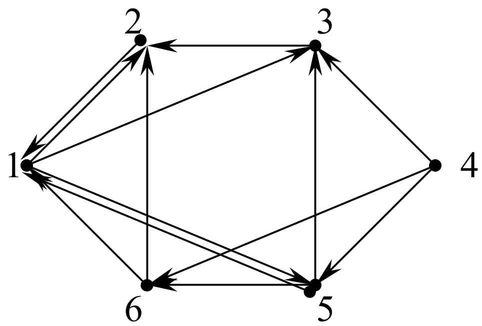
بـرای نمایش گراف‌ها می‌توانیم از لیست‌های همجواری استفاده کنیم. بدینصورت که آرایه‌ای به تعداد گـره‌هـای گـراف ایجاد می‌کنیم. هر عنصر این آرایه اشاره‌گری است به ابتدای یک لیست پیوندی. این لیست پیوندی شامل گره‌هایی است که به گره متناظر با عنصر آرایه متصل‌اند.

در گـراف‌هـای غیـر جهت‌دار تعداد کل گره‌ها در لیست‌های پیوندی دو برابر تعداد یالهای گراف است ولـی در گـراف‌هـای جهت‌دار مجموع تعداد گره‌های لیست‌های پیوندی برابر تعداد یالهای گراف است. فضـای مصرفی در نمـایش بوسـیله لیسـت همجـواری در گـراف‌هـای جهت‌دار از مرتبه $n + e$ و در گراف‌های غیر جهت‌دار $n + 2e$ است. e تعداد یالهای گراف و n تعداد گره‌های گراف است.

$n = | V |$

$e = | E |$



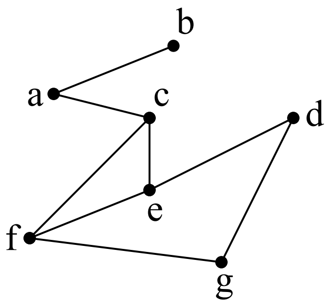


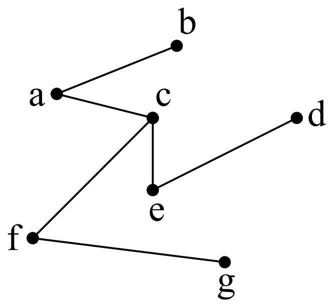
روشهای پیمایش گراف

دو روش کلی برای پیمایش گراف وجود دارد.

۱. اول سطح breadth first search (bfs)

۲. اول عمق depth first search (dfs)





a - b - c - f - e - g – d

(bfs)

تعریف درخت : درخت گراف متصل بدون سیکل است.

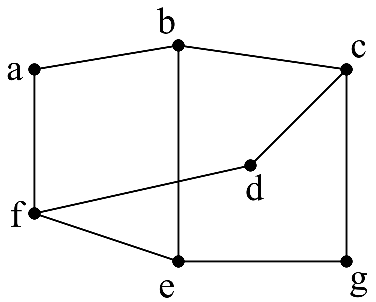
روش اول سطح

در پیمایش اول سطح با شروع از یک گره و ملاقات آن , کلیه گره‌های مجاور آن نیز ملاقات می‌شوند. سپس این رویه به ترتیب برای هر یک از گره‌های مجاور تکرار می‌شود. برای پیاده‌سازی پیمایش اول سطح (bfs) از ساختمان داده صف استفاده می‌کنیم. بدین ترتیب که رئوس مجاور هنگام ملاقات وارد صف می‌شوند , سپس از سر صف یک عنصر را حذف کرده و گره‌های مجاور آنرا ضمن ملاقات به صف اضافه می‌کنیم. هر گره در صورتی ملاقات می‌شود (وارد صف می‌گردد) که قبلاً ملاقات نشده باشد.

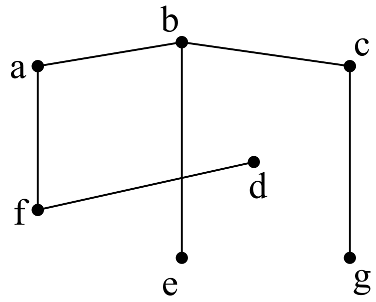
نتیجه پیمایش اول سطح , درخت پوشای اول سطح (درخت bfs گراف) می‌باشد.

پیمایش اول سطح از یک گراف و در نتیجه درخت پوشای bfs لزوماً منحصر به فرد نیست.

مثال : گراف زیر را بوسیله روش اول سطح پیمایش کرده و درخت پوشای آنرا بکشید.

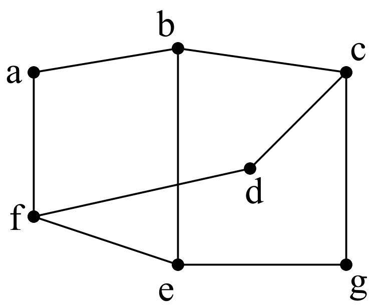


a - b - f - c - e - d - g

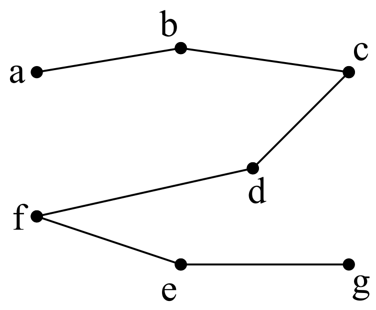


روش اول عمق

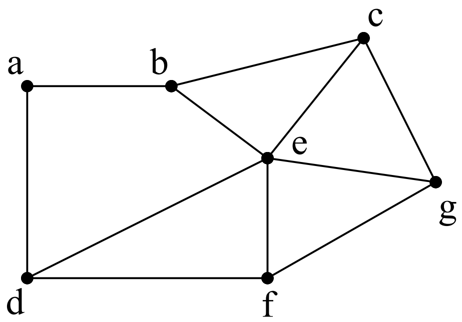
در پیمایش اول عمق با شروع از یک گره و ملاقات آن , یکی از گره‌های مجاوری که ملاقات نشده را ملاقات می‌نمائیم و این عمل را متناوباً تکرار می‌کنیم. اگر در یک گره بودیم و همه گره‌های مجاور آن ملاقات شده بود , یک مرحله به عقب برمی‌گردیم. برای پیاده‌سازی درپیمایش اول عمق از پشته استفاده می‌کنیم. نتیجه پیمایش اول عمق , درخت پوشای dfs است که لزوماً منحصر به فرد نیست.



a - b - c - d - f - e - g



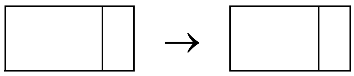
تمرین : از گره e شروع کرده و اول سطح و اول عمق گراف زیر را بنویسید.



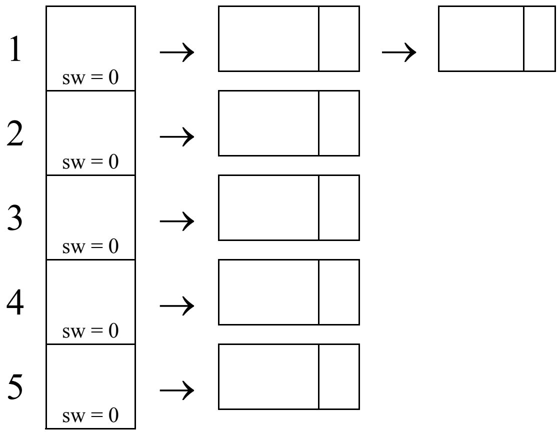
اول سطح = e - b - c - d - f - g - a

اول عمق = e - c - b - a - d - f - g

کدنویسی روش اول سطح و اول عمق



```
Struct Linklist
{
        int data ;
        struct Linklist  *Next ;
}
typedef struct Linklist node ;
struct LinkArray
{
        int sw ;
        node  *Link ;
}
typedef struct LinkArray pointer ;
```



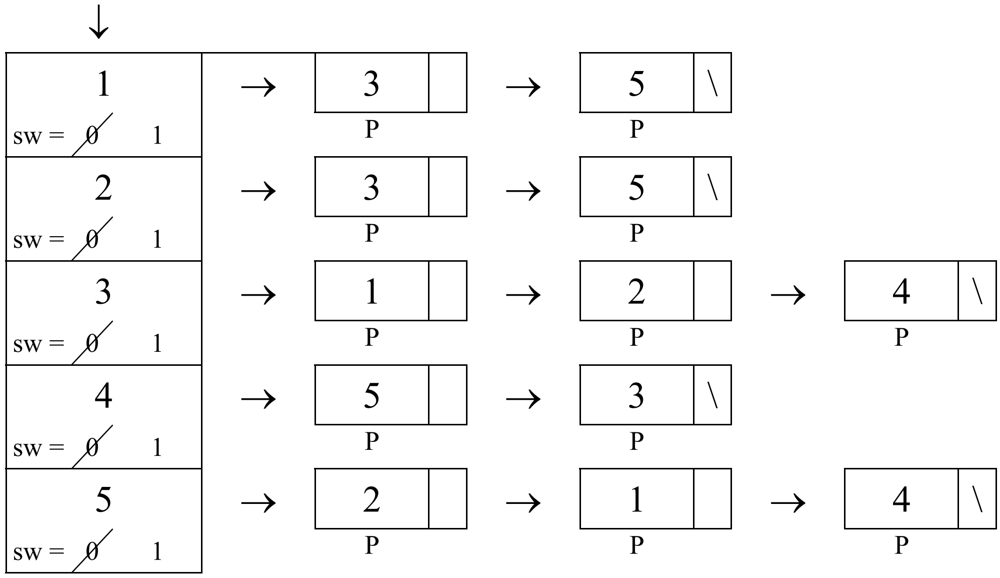

```
pointer  *graphnodes ;
graphnodes = New   pointer [n] ;
```

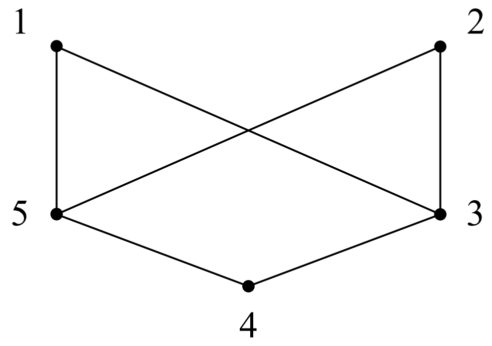
الگوریتم پیمایش اول سطح

در این پیمایش از رأس k شروع می‌کنیم. پس بنابراین داریم :

```
Void bfs ( pointer graphnodes [ ] , int k )
{
        node  *p ;
        graphnodes [ k ]  sw = 1 ;
        C out << k ;
        Addqueue ( k ) ;                    مقدار k را به انتهای صف اضافه می‌کند.
        while ( ! ( queue . empty ( )))    چک می‌کند که آیا صف خالی است یا خیر
        {
                k = delqueue ( ) ;    از صف یک مقدار حذف کرده و در k قرار می‌دهد.
                p = graphnodes [ k ] . Link ;
                do
                {
                        if ( graphnodes [ p → data ] . sw = = 0 )
                        {
                                addqueue ( p → data ) ; C out << p → data ;
                                graphnodes [ p → data ] . sw = 1 ;
                        }
                        p = p → Next ;
                }
                while ( p ) ;
        }
}
```

Graphnodes [5]





1	3	5	2	4

1 3 5 2 4

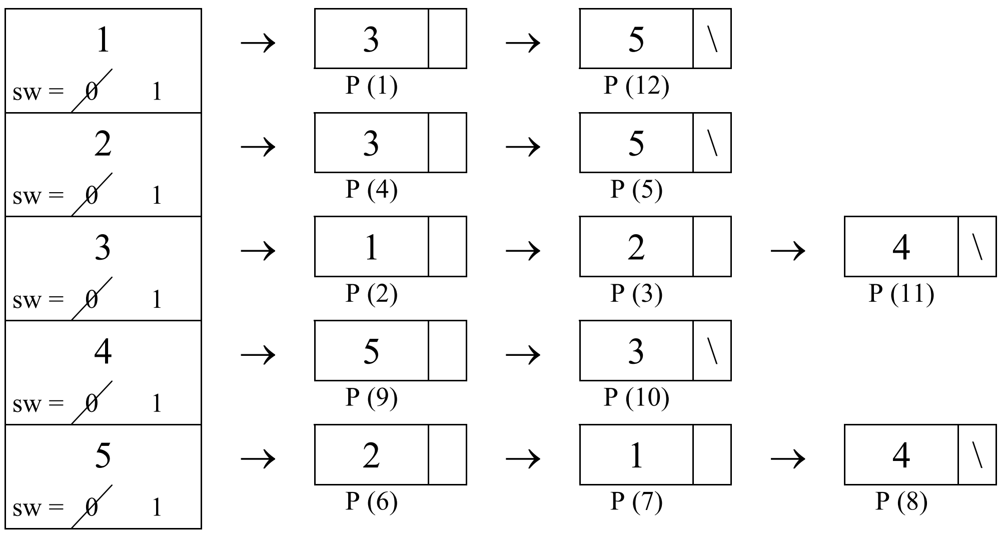
k
1
3
5
2
4

الگوریتم اول عمق

در این پیمایش از رأس k شروع می‌کنیم. پس بنابراین داریم :

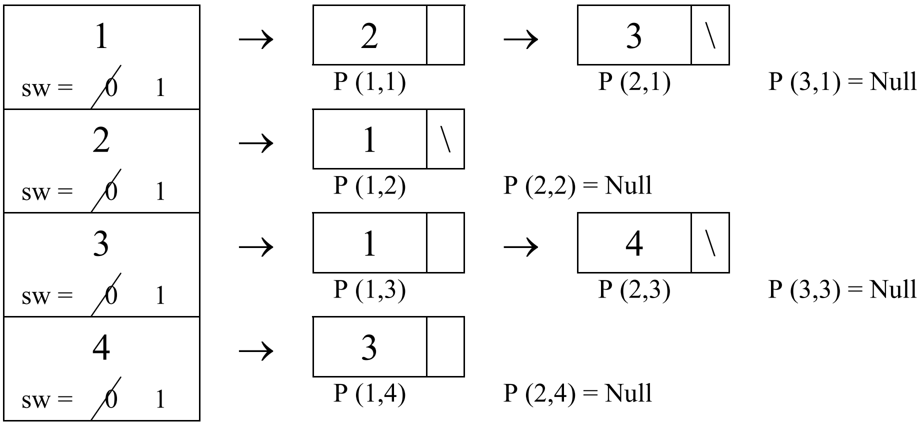
```
Void dfs ( pointer graphnodes [ ] , int k ) //
{
        node  *p ;
        graphnodes [ k ] . sw = 1 ;  C out << k ;
        for ( p = graphnodes [ k ] . Link ;  p ! = Null ;  p = → Next )
        if ( graphnodes [ p → data ] . sw = = 0 )
        dfs ( graphnodes , p → data ) ;
}
```

همان گراف بالا را در نظر می‌گیریم :



1 3 2 5 4

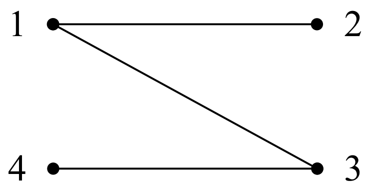
مثال :



از یک شروع می‌کنیم.

k
1
2
3
4

1 2 3 4

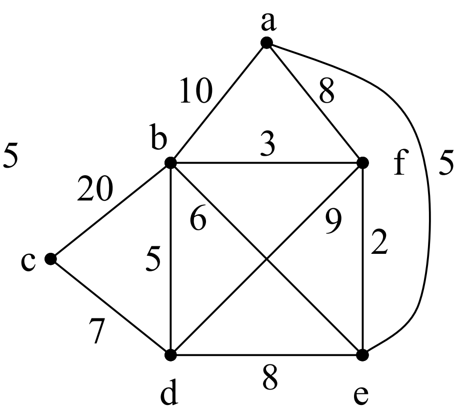


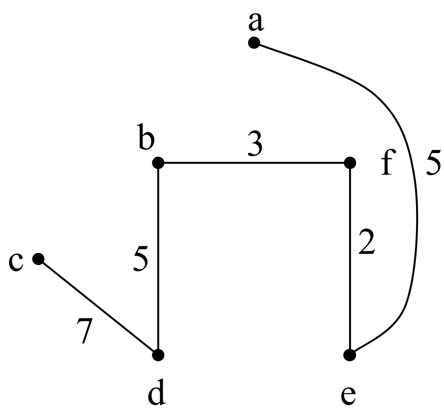
درخت پوشای بهینه (حداقل هزینه)

درخـت پوشای بهینه در گراف‌های ارزش‌دار (وزن‌دار) ساخته می‌شود و آن درختی است که اگر ارزش تمام گراف‌های آن را جمع کنیم کوچکترین عدد ممکن حاصل گردد.

- **روش اول الگوریــتم کراســکال (kraskal)** : در الگوریــتم کراسـکال , یالهای گراف را به تـرتیب صـعودی مـرتب می‌کنیم. از اولین (کوچکترین) یال شروع کرده و هر یال را به گراف اضـافه مـی‌کنیم به شرط اینکه دور در گراف ایجاد نگردد. این روال را آنقدر ادامه می‌دهیم تا درخت پوشای بهینه تشکیل گردد.

✓ fe = 2
✓ bf = 3
✓ bd = 5 , ✓ ae = 5
× be = 6
✓ cd = 7
× de = 8 , af = 8
× df = 9
× ab = 10
× bc = 20





2 + 3 + 5 + 5 + 7 = 22