الگوریتم

مجموعه دستورالعملهایی که برای رسیدن به یک هدف خاص ارائه می شوند را الگوریتم گویند. به بیان دیگر، مجموعه ای متناهی از دستورالعمل هایی است که به ترتیب خاصی اجرا می شوند و مسئله را حل می کنند و لزوما منحصر به فرد نیستند.

تمام الگوریتم ها باید شرایط و معیار های زیر را دارا باشند:

- ورودی : یک الگوریتم می تواند هیچ یا چندین پارامتر را به عنوان ورودی بپذیرد.
- خروجی : یک الگوریتم باید حداقل یک کمیت را به عنوان خروجی "نتیجه عملیات "تولید کند.
- قطعیت : دستور های الگوریتم باید با زبانی دقیق و بدون ابهام بیان شود.(در اجرای آن هیچ ابهامی نباشد) هر دستورالعمل نیز باید انجام پذیر باشد.
- خاتمه پذیر باشد : الگوریتم باید دارای شروع و پایان مشخصی باشد، به نحوی که اگر دستورهای آن را دنبال کنیم پس از طی مراحلی که قابل شمارش و متناهی باشد، خاتمه یابد.

لازم به ذکر است مدت زمان لازم برای خاتمه یک الگوریتم باید به گونه ای معقول و کوتاه باشد.

ارزیابی کارایی الگوریتم

الگوریتم‌های مختلفی برای حل یک مسئله ممکن است طراحی شده باشند. برای انتخاب بهترین الگوریتم باید معیاری جهت مقایسه کارایی الگوریتم‌ها داشته باشیم. کارایی یک الگوریتم با دو معیار:

- **پیچیدگی زمانی** time complexity
- **پیچیدگی حافظه** space complexity

سنجیده می‌شود؛ که رفتار الگوریتم را در زمان اجرا با مجموعه‌ای از ورودی‌های منتخب توصیف می‌کند.

پیچیدگی زمانی

زمان اجرای یک الگوریتم به موارد زیر بستگی دارد:

- **سرعت سخت‌افزار**
- **نوع کامپایلر**
- **اندازه داده ورودی**
- **ترکیب داده‌های ورودی**
- **پیچیدگی زمانی الگوریتم**

از این عوامل سرعت سخت‌افزار و نوع کامپایلر به صورت ثابت در زمان اجرای برنامه‌ها دخیل هستند. پارامتر مهم پیچیدگی زمانی الگوریتم است که خود تابعی از اندازه مسئله است. ترکیب داده‌های ورودی نیز با بررسی الگوریتم در شرایط مختلف قابل اندازه‌گیری است. در ادامه سعی در بررسی پیچیدگی زمانی الگوریتم خواهیم داشت .

منظور از زمان در محاسبه ی پیچیدگی زمانی الگوریتم، واحدهای زمانی واقعی مانند میکرو یا نانو ثانیه نیست بلکه منظور واحدهای منطقی است که رابطه بین بزرگی (n) یک فایل یا یک آرایه و زمان مورد نیاز برای پردازش داده‌ها را شرح می‌دهد. (توجه کنید که هر دستور یک واحد زمانی اشغال می‌کند)

برای بررسی پیچیدگی زمانی الگوریتم تابعی به نام $T(n)$ که تابع زمانی الگوریتم نامیده می‌شود در نظر می‌گیریم که در آن n اندازه ورودی مسئله است. مسئله ممکن است شامل چند ورودی باشد. به عنوان مثال اگر ورودی یک گراف باشد علاوه بر تعداد راس ها (n) تعداد یال‌ها(m) هم یکی از مشخصه‌های داده ورودی است. در این صورت زمان اجرای الگوریتم را با $T(n,m)$نمایش می‌دهیم. در صورتی که تعداد پارامترها بیشتر باشند آن‌هایی که اهمیت بیشتری در زمان اجرا دارند را در محاسبات وارد می‌کنیم و از بقیه صرف نظر می‌کنیم .

زمان اجرا برای هر عبارت یک برنامه بستگی به؛ نوع عبارت دارد.

1- در عبارات توضیحی برابر صفر
2- در دستور جایگزینی یا محاسباتی برابر یک
3- در دستورهای غیربازگشتی حلقه for, while, repeat until به تعداد تکرار حلقه

مجموع تعداد عملکردهای اجرایی، زمان اجرای برنامه را می‌رساند و مستقل از ماشین است.

تعداد کل مراحل برنامه زیر را بیابید

float sum(float list[], int n)	0	
{	0	
float s=0;	1	چون عدد صفر در s ریخته می شود
int i;	0	
for(i=0;i<n;i++)	n+1	for شرط حلقه
s=s+list[i];	n	
return s;	1	دستور return توسط cpu باید اجرا شود
}	0	
	2n+3	

{ و } و نیز خط اول تعریف تابع و تعریف متغیر دستوراتی نیستند که توسط CPU اجراء شوند پس مرحله اجرائی آنها صفر است.

زمان اجرای هر عبارت جایگزینی یا محاسباتی را مساوی ۱ واحد زمانی فرض می‌کنیم. هم چنین دستور داخل حلقه n بار انجام می‌شود ولی آزمایش کردن شرط حلقه در خط for به تعداد n+1 بار صورت می‌گیرد. دستور return نیز مساوی یک واحد زمانی است.

- نکته : خطوط { } و نیز خط اول تعریف تابع و تعریف متغیر دستوراتی نیستندکه توسط cpu اجرا شوند و زمان اجرای آن‌ها برابر صفر است.

- نکته : اگر عمل اصلی را فقط خط s=s+list[i]; فرض کنیم آنگاه T(n)=n خواهد بود.

مثال: دستور اصلی a=a+1 در قطعه برنامه زیر چند بار اجرا می‌گردد؟

```
for(i=1 ;i<=m ;i++)

  for(j=1 ; j<=n ; j++)
    a=a+1;
```

- نکته : حلقه‌های for برنامه مستقل از یکدیگر هستند

T(N)=MN

مثالی دیگر:

```
int sum(float list[][y], int x)   0
{                                 0
 float s=o;                       1
 int i,j;                         0
 for(i=o;i<x;i++)                 x+1      شرط حلقه for
{                                 0
  for(j=o;j<y;j++)                x(y+1)
    s=s+list[i][j];               xy
}                                 0
 return s;                        1
}                                 0
```

$1+ x+1 + x^2+ x + x^2 +1$

اگر x=y باشد

$2x^2+ 2x +3$

مثالی دیگر:

```
int F(int A[],int x)
{
  int s=0;
 int i;
 for(i=1; i<x; i=i*2)
{
  s=s+ A[i];
}
```

برای حل مسئله یک مقدار اولیه برای X فرض می کنیم . که در این مسئله باید توانی از 2 باشد. مثلا 16

i	تعداد اجرا شدن s=s+A[i];
1	1 بار
2	2 بار
4	3 بار
8	4 بار
16	

$$\log_2 16 = 4$$

لذا تعداد دفعات اجرای عبارت s=s+A[i] ، با استفاده از محاسبات انجام شده برابر است با:

$$\log_2 X$$

و تعداد دفعات بررسی شرط حلقه ی for یک واحد بیشتر است. بنابر پیچیدگی زمانی اجرای حلقه ی for برابر است با:

$$T(n)= \log_2 X +1$$

اما پیچیدگی زمانی اجرای کل برنامه برابر است با:

$$T(n)= 2 \log_2 X +2$$

مثال: دستور اصلی a=a+1 در قطعه برنامه زیر چند بار اجرا می‌گردد؟

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

- نکته : حلقه‌های for برنامه به یکدیگر وابسته‌اند:

j	i	تعداد اجرا شدن a=a+1;
۱	۱	۱ بار
۲	۱،۲	۲ بار
۳	۱،۲،۳	۳بار
-	-	-
-	-	-
-	-	-
n	1,2,3,... ,n	nبار

تعداد اجرا شدن دستور اصلی =1,2,3,... ,n = n(n+۱)/ ۲

هدف از محاسبه پیچیدگی زمانی یک الگوریتم این است که بفهمیم نیازمندی یک الگوریتم به زمان با چه تابعی رشد می‌کند و هدف اصلی بدست آوردن این تابع رشد است.

پیچیدگی زمانی در بهترین و بدترین حالت

تابع زیر را در نظر بگیرید که در n عنصر اول آرایه‌ی arr مقدار x را جستجو کرده و اندیس آن را باز می‌گرداند.

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

در فرآیند جستجو ممکن است عنصر مورد نظر در هر یک از خانه‌های آرایه باشد یا اصلا وجود نداشته باشد که مقدار 1− توسط تابع بازگردانده می‌شود. بنابراین نمی‌توان رابطه‌ی ثابت و مشخصی برای تعداد اعمال اصلی تعریف کرد. در چنین شرایطی می‌توان گفت بهترین حالت اجرا از مرتبه‌ی 1 است و بدترین حالت آن از مرتبه‌ی n.

مرتبه‌ی 1 مستقل بودن زمان اجرا از اندازه‌ی ورودی را مشخص می‌کند. عنصر مورد نظر چه در ابتدای یک آرایه با 10 عنصر باشد و چه در ابتدای یک آرایه با 10000 عنصر، یافتن آن تنها یک مقایسه نیاز دارد و در زمان ثابتی اتفاق می‌افتد.

پیچیدگی زمانی متوسط الگوریتم

دو حالت بهترین و بدترین ممکن است کم اتفاق بیفتند. محاسبه‌ی مرتبه‌ی بدترین حالت این حسن را دارد که نهایت منابع یا زمان مورد نیاز را برآورد می‌کنیم. اما عموما اجرا به ازای ورودی‌های مختلف به این میزان زمان یا حافظه نیاز ندارد. به همین دلیل حالت متوسط اجرا نیز محاسبه می‌شود تا بدانیم به طور متوسط الگوریتم در چه پیچیدگی زمانی سیر می‌کند.

حالت متوسط اجرا را می‌توان از محاسبه‌ی امید ریاضی به دست آورد. در مورد تابع جستجوی خطی فوق، با احتمال 1/n تعداد i مقایسه)مقایسه تا خانه ی i آرایه(ما را به جواب می‌رساند .

پس الگوریتم جستجوی خطی به طور متوسط نیز از مرتبه‌ی n است. هرچند ضریب n نسبت به بدترین حالت کمتر است. این ضریب به صورت شهودی هم قابل درک است که به طور متوسط نصف عناصر جستجو می‌شوند.

یک مثال دیگر روش حل مسئله برج هانوی است. برای جابجا کردن n دیسک از میله‌ی مبدأ به میله‌ی مقصد ابتدا نیاز است $n-1$ دیسک بالایی را به میله‌ی کمکی منتقل کرده، سپس دیسک بزرگ را به میله‌ی مقصد برده و در نهایت دیسک‌های موجود در میله‌ی کمکی را به میله‌ی مقصد منتقل کنیم. اگر T(n) تعداد حرکات مورد نیاز برای حل کامل مسئله باشد، بر اساس این توضیح می‌توان نوشت:

$$T(n) = T(n-1) + 1 + T(n-1) = 2T(n-1) + 1, \quad T(1) = 1$$

حل این رابطه‌ی بازگشتی به $T(n) = 2^n - 1$ می‌رسد. پس حل این مسئله از مرتبه‌ی پیچیدگی 2^n است. این مسئله بهترین و بدترین حالت ندارد و همواره یک رابطه برای آن برقرار است و نمی‌توان الگوریتم کاراتری نیز برای آن نوشت. پس می‌توان نتیجه گرفت به ازای مقادیر بزرگ n حل آن بسیار زمانبر خواهد بود. به همین دلیل نیز از آن افسانه ساخته‌اند!

برای محاسبه پیچیدگی زمان الگوریتم ابتدا تعداد قدم‌های الگوریتم به صورت تابعی از اندازه مسئله مشخص می‌شود، برای انجام این کار تعداد تکرار عملیات اصلی الگوریتم محاسبه می‌شود و به صورت تابع f بیان می‌شود. سپس تابع g، که مرتبه بزرگی تابع f را وقتی اندازه ورودی به اندازه کافی بزرگ است نشان می‌دهد، بدست می‌آید. در نهایت پیچیدگی الگوریتم برای نشان دادن رفتار الگوریتم با ورودی‌های مختلف با استفاده از نمادها O، Θ و Ω که در بخش بعدی با آن‌ها آشنا می‌شویم، بیان می‌شود.

پیچیدگی حافظه

پیچیدگی حافظه‌ای میزان فضائی از حافظه‌است که برنامه برای اجرای کامل به آن نیاز دارد. فضای مورد نیاز در هربرنامه مجموع قسمت‌های زیر است:

- بخش ثابت فضا که معمولاً شامل فضای دستورالعمل، فضای متغیرهای با اندازه ثابت و فضای لازم برای ذخیره ورودی و خروجی‌های برنامه‌است.
- بخش متغیر فضا شامل فضای پشته و فضای موردنیاز برای مقادیر متغیرهایی که اندازه آن‌ها بستگی به مسئله و مشخصات ورودی دارد.

در تحلیل فضای لازم روی تخمین بخش متغیر تأکید نداریم زیرا برای هر مسئله ابتدا باید مشخصات موردی را تعیین کنیم که کار دشواری است.

نمایش پیچیدگی الگوریتم

برای نمایش پیچیدگی الگوریتم‌ها از تعاریف زیر استفاده می‌شود:

Big-O (حدبالا)

تابع $f(n)$ را برای $n\geq 0$ در نظر بگیرید. می‌گوئی $f(n)=O(g(n))$ است اگر ثابت مثبت و حقیقی c و عدد صحیح و غیر منفی N وجود داشته باشند به طوریکه به ازای تمام مقادیر $n\geq N$:

$f(n)\leq cg(n)$ برقرار باشد.

این نماد حد بالائی برای تابع $f(n)$ می‌دهد و وقتی بکار می‌رود که رفتار الگوریتم بدترین حالت و بیشترین زمان اجرا را برای مقادیرمعین ورودی دارد.

تتا/Θ (حدمتوسط)

تابع f(n) را برای n≥0 در نظر بگیرید. می‌گوئیم(f(n) = Θ(g(n)) است اگر ثابت‌های مثبت و حقیقی c و d و عدد صحیح غیر منفی N وجود داشته باشند به طوریکه به ازای تمام مقادیر n≥N:

c g(n) ≤f(n) ≤ d g(n) برقرار باشد.

به عبارت دیگر برای تابع پیچیدگی مفروض f(n):

```
Θ(f(n)) = O(f(n)) ∩ Ω(f(n))
```

این نماد حدمتوسطی برای تابع f(n) می‌دهد و زمان اجرای الگوریتم را به صورت میانگینی از تعداد عملیات انجام شده با کلیه نمونه ورودی‌های مسئله نشان می‌دهد .

اگر زمان الگوریتم وابسته به ورودی نباشد با نماد O(۱) نشان داده می‌شود؛ و برای تحلیل الگوریتم باید به اندازه کافی الگوریتم را درک کرده باشیم تا بهترین و بدترین رفتار را تولید و محاسبه کنیم

چون برآورد رفتار آماری ورودی‌ها امری دشوار است، در اکثر موارد به بدترین حالت قناعت می‌کنیم

نکته. اگر الگوریتم شامل بخش های مختلفی باشد که هر قسمت پیچیدگی متفاوتی دارد، مرتبه بزرگی هر قسمت را پیدا کرده و بزرگترین مرتبه را به عنوان پیچیدگی کل الگوریتم در نظر می‌گیریم در زیر مربته اجرائی چند تابع به ترتیب صعودی نوشته شده است.

notation	name
O(1)	constant
O(log n)	logarithmic
O(n)	linear
O(n^2)	quadric
O(n^c)	polynomial
O(c^n)	exponential
O(n!)	factorial

$$\log_2 n, \quad n, \quad n\log_2 n, \quad n^2, \quad n^3, \quad 2^n$$

تابع بازگشتی(recursive) :

تابع بازگشتی دارای دستوری است که تابع را فراخوانی می کند و تعداد دستورات محدودی را اجرا می کند و دارای شرط خاتمه است.

مثالهایی از تابع بازگشتی:

فاکتوریل

مجموع اعداد 1 تا n

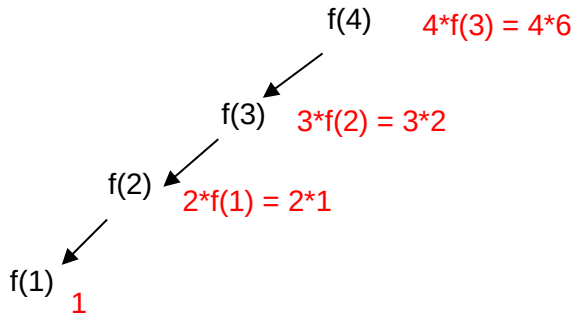
توان

فیبوناچی

تابع بازگشتی فاکتوریل:

$$f(n)=\begin{cases} 1 & n=1 \\ n*f(n-1) & n>1 \end{cases}$$

```
long int f(int  n)
{  if(n==1)
       return 1;
    else
        return n*f(n-1);
}
```



تابع بازگشتی مجموع اعداد 1 تا n :

$$sum(n)=\begin{cases} 1 & n=1 \\ n+sum(n-1) & n>1 \end{cases}$$

```
long int sum(int  n)
{  if(n==1)
       return 1;
    else
        return n+ sum(n-1);
}
```

sum(4) 4+sum(3) = 4+6

sum(3) 3+sum(2) = 3+3

sum(2) 2 +sum(1) = 2+1

sum(1) 1

تابع بازگشتی توان :

$$pow(n,m)=\begin{cases} n & m=1 \\ n*pow(n,m-1) & m>1 \end{cases}$$

```
long int pow(int  n, int m)
{  if(m==1)
       return n;
    else
        return n* pow(n,m-1);
}
```

pow(3,4) 3*pow(3,3) = 3*27

pow(3,3) 3*pow(3,2) = 3*9

pow(3,2) 3*pow(3,1) = 3*3

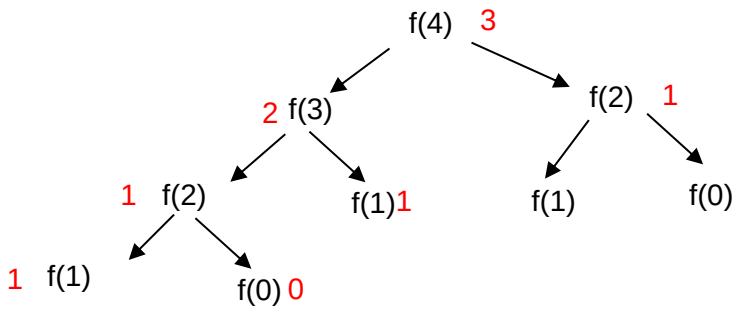
pow(3,1) 3

تابع بازگشتی فیبوناچی

0,1,1,2,3,5,8,13,21,34,....

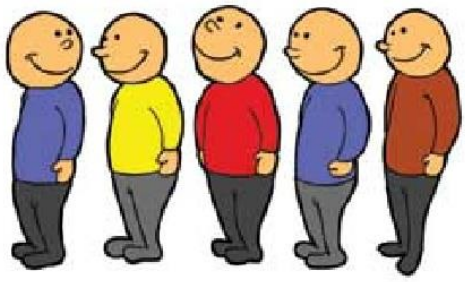
$$f(n)=\begin{cases} 0 & n=0 \\ 1 & n=1 \\ f(n-1) * f(n-2) & n>=2 \end{cases}$$

```
int f(int  n)
{  if ((n==0) || (n==1))
       return n;
    else
        return f(n-1)+ f(n-2);
}
```



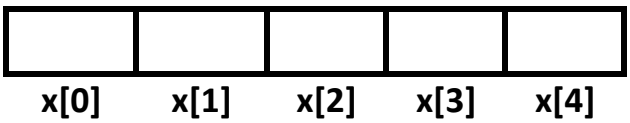
صف (Queue):

صـف لیستی است که عمل افزودن داده ها درون آن از یک طرف (انتهای لیست) و عمل حذف داده هـا از سـمت دیگر (ابتدای لیست) انجام می شود. صف را لیست FIFO (First In First Out) مـی نامند. زیرا اولین عنصر ورودی، اولین عنصر خروجی از صف نیز هست .

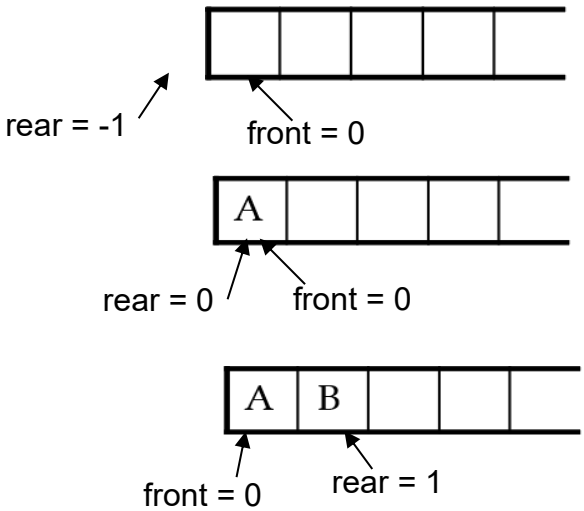


در ساختمان داده صف دو متغیـر front و rear بـه تـرتیب بـرای نشان دادن ابتدا و انتهای صف بکار می روند . صف را می توان با استفاده از آرایه ها یا لیست های پیوندی پیاده سازی کرد.

اگر صف را آرایه ای n عضوی از عناصر بدانیم مقدار front می تواند از 0 تا n تغییر کند و مقدار rear از 1- تا n-1 متغیر است.



برای صف در ابتدا مقدار اولیه صفر را برایfront و مقدار اولیه ی 1- را برای rear در نظر می گیریم.



ADT صف

مجموعه عناصر :

مجموعه ای از عناصر مرتب که در آن عناصر از ابتدای صف حذف و از انتهای صف به آن اضافه می شوند.

عملیات اصلی :

creator : صف خالی را ایجاد می کند.

isEmpty : خالی بودن صف را تست می کند.

isFull : پر بودن صف را تست می کند.

add : عنصری را به آخر صف می افزاید.

remove : عنصری را از ابتدای صف حذف می کند.

retrieve : عنصر ابتدای صف را بازیابی می کند.

کلاس پیاده سازی صف به زبان C++

```
#define SIZE 5
class queue{
   int front;
   int rear;
   int a[SIZE];
  public:
   queue();
   int IsEmpty();
   int IsFull();
   void add(int );
   int remove();
   int retrieve();
};
```

تابع سازنده ی صف

یک صف خالی ایجاد می کند.

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

بررسی خالی بودن صف

اگر هیچ عنصری در صف وجود نداشته باشد، این تابع مقدار 1 و در غیر اینصورت مقدار 0 را بر می گرداند.

```
int queue:: isEmpty()
{  if( rear< front)
      return 1;
   return 0;
}
```

بررسی پر بودن صف

اگر همه ی مکانهای صف پر باشند، این تابع مقدار 1 و در غیر اینصورت مقدار 0 را بر می گرداند.

```
int queue:: isFull()
{
   if( rear == SIZE-1)
      return 1;
   return 0;
}
```

حذف عنصر از صف

در پیاده‌سازی صف با استفاده از آرایه، عنصر داده‌ای واقعاً حذف نمی‌شود، بلکه تنها مقدار front یک واحد افزایش می‌یابد. اما در پیاده‌سازی صف با استفاده از لیست پیوندی، واقعاً عنصر داده‌ای از فضای تخصیص یافته صف پاک می گردد. یک عملیات remove می‌تواند شامل گام‌های زیر باشد:

- گام 1: بررسی کن که صف خالی است یا نه؟
- گام 2: اگر صف خالی بود، یک خطا اعلام کن و خارج شو.
- گام 3: اگر صف خالی نبود، عنصر داده‌ای را که front مورد اشاره قرار می‌دهد برگردان.
- گام 4: مقدار front را یک واحد افزایش بده.

```
int queue:: remove()
{
 if (empty ())
  { cout<<"Queue is empty!";
   exit();  }
else
    x=a[front++];
}
```

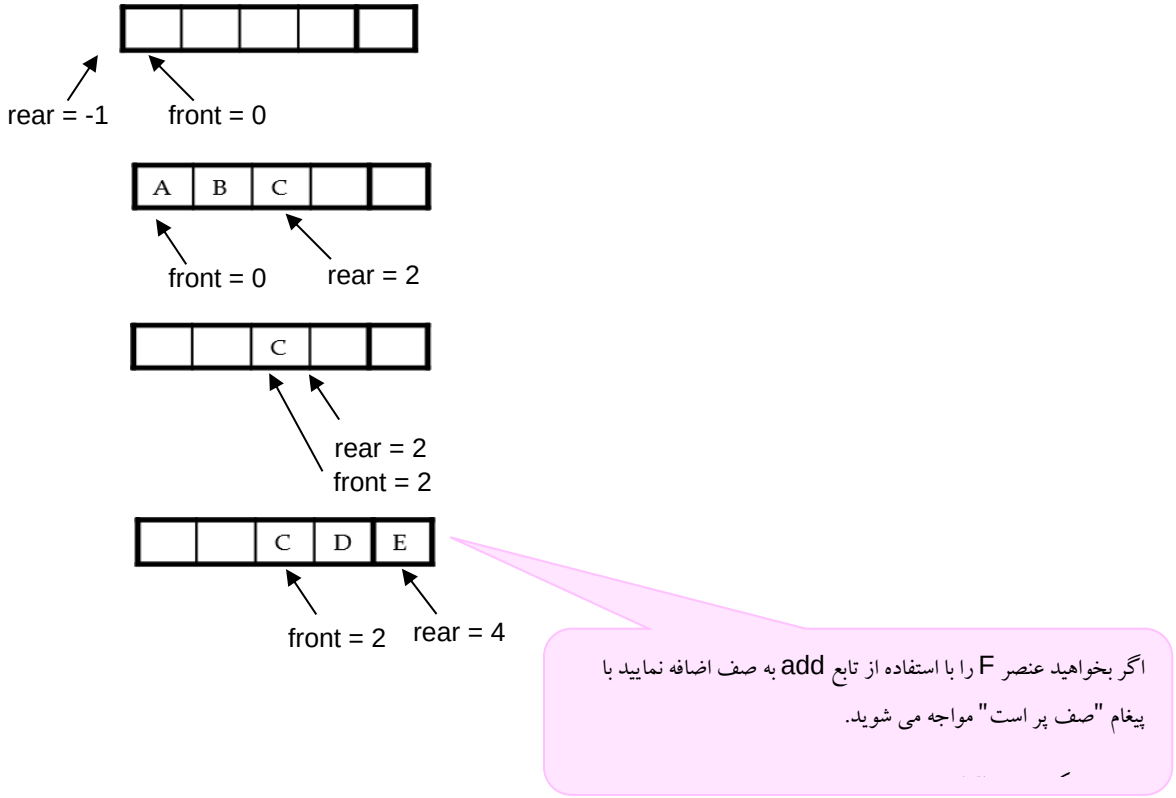
اضافه کردن عنصر به صف

این عملیات شامل مراحل زیر است:

- گام 1: بررسی کن که صف پر است یا نه.
- گام 2: اگر صف پر باشد، اعلام خطا کرده و خارج شو.
- گام 3: اگر صف پر نبود، مقدار rear را یک واحد افزایش بده تا به فضای خالی بعدی اشاره کند.
- گام 4: عنصر داده مد نظر را در مکانی که rear به آن اشاره می کند، اضافه کن.

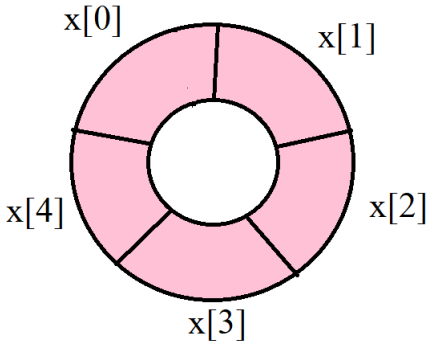
```
void queue:: add(int x)
{
  if (isFull())
   { cout<<"queue is full ! ";
     exit(); }
  else
    a[++rear] = x;
}
```

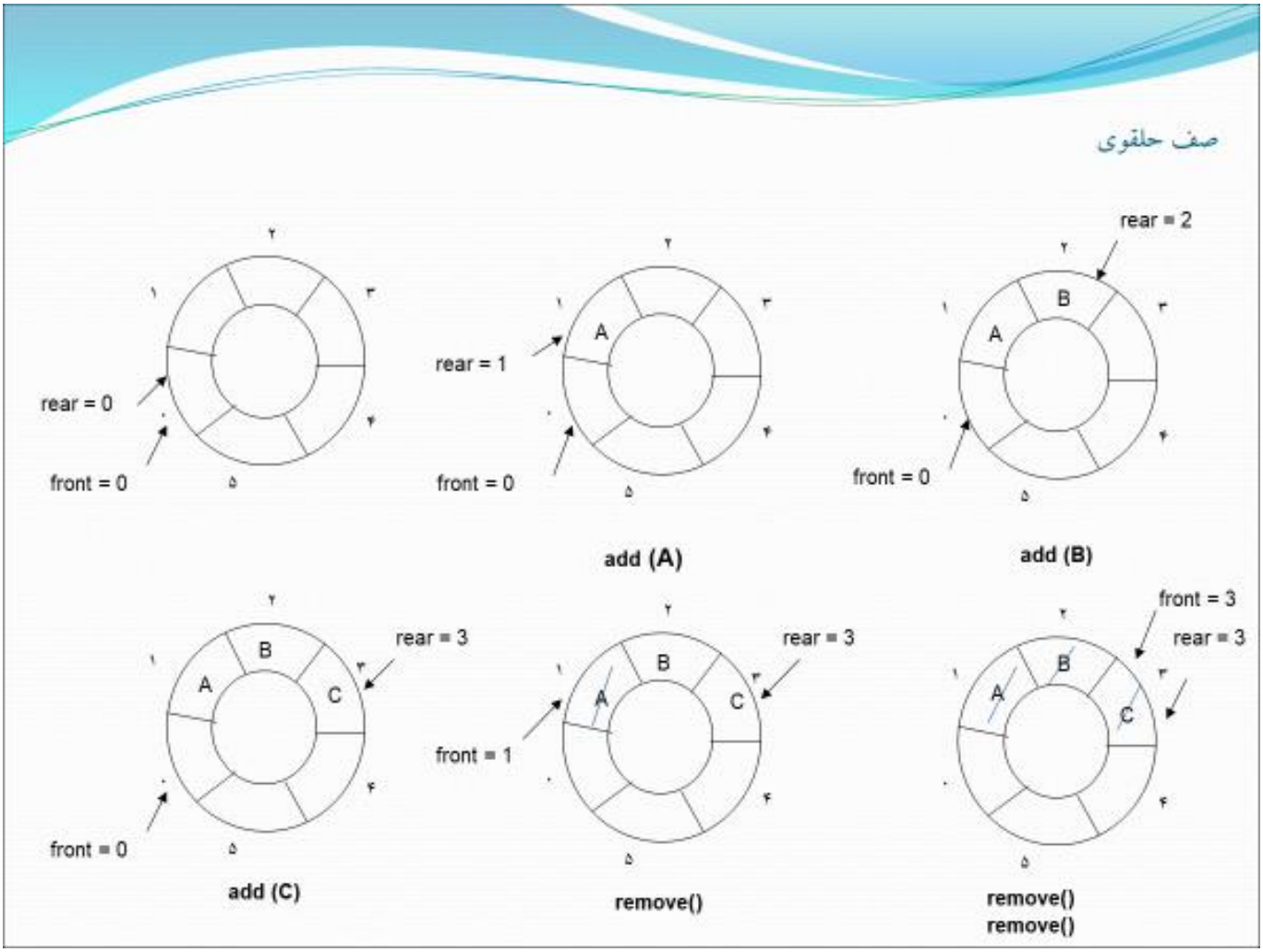
مشکل صف خطی این است که تنها یک بار قابل پر شدن است و در صورتیکه عناصر آن حذف شوند نیز با پیغام " صف پر است" مواجه می شوید .



لذا به منظور حل مشکل صف خطی آن را بصورت حلقوی تعریف می کنیم . در صف حلقوی (دوار) rear وfront بعد از رسیدن به آخرین مقدار خود در صورت وجود شرایط لازم مجددا مقادیر اولیه را می توانند بگیرند. صف حلقوی n عنصری را بصورت آرایه ی 0 تا n – 1 تعریف می کنیم.

در این حالت وقتی rear=n-1 ، عنصر بعدی در مکان صفرم آرایه قرار می گیرد.





بـرای اضافه کردن به صف حلقوی، rear یکی اضافه می شود و در صورتیکه rear = n – 1 باشد باید صفر بشود . بدین منظور rear را با رابطه زیر در هر شرایطی مقداردهی می کنیم.

```
rear = (rear + 1) % n
```

این مسئله برای front نیز برقرار است.

```
front = (front + 1) % n
```

در صـف حلقوی، شرط front == rear به معنای خالی بودن صف است. **ولی به منظور ایجاد تمایز بین شرط خالی بودن و پر بودن صف،** شرط پر بودن صف بدین ترتیب تغییر می یابد.

```
front == (rear + 1) % n
```

تابع سازنده ی صف حلقوی

یک صف حلقوی خالی ایجاد می کند.

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

بررسی خالی بودن صف حلقوی

اگر هیچ عنصری در صف وجود نداشته باشد، این تابع مقدار 1 و در غیر اینصورت مقدار 0 را بر می گرداند.

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

بررسی پر بودن صف حلقوی

اگر همه ی مکانهای صف پر باشند، این تابع مقدار 1 و در غیر اینصورت مقدار 0 را بر می گرداند.

```
int queue:: isFull()
{
  if( front == (rear+1) % SIZE)
    return 1;
  return 0;
}
```

حذف عنصر از صف حلقوی

در پیاده‌سازی صف حلقوی با استفاده از آرایه، عنصر داده‌ای واقعاً حذف نمی‌شود، بلکه تنها مقدار front یک واحد افزایش می‌یابد. اما در پیاده‌سازی صف با استفاده از لیست پیوندی، واقعاً عنصر داده‌ای از فضای تخصیص یافته صف پاک می گردد. یک عملیات remove می‌تواند شامل گام‌های زیر باشد:

- گام 1: بررسی کن که صف حلقوی خالی است یا نه؟
- گام 2: اگر صف خالی بود، یک خطا اعلام کن و خارج شو.
- گام 3: اگر صف خالی نبود، مقدار front را یک واحد افزایش داده و باقیمانده تقسیم آن بر سایز صف را در front قرار بده.
- گام 4: عنصر داده‌ای را که front مورد اشاره قرار می‌دهد برگردان.

```
int queue:: remove()
{
 if (empty()){
   cout<<"queue is empty !";
    exit(); }
 else
   {
     front = (front +1) % SIZE;
     x = a[front];
   }
}
```

اضافه کردن عنصر به صف حلقوی

این عملیات شامل مراحل زیر است:

- گام 1: بررسی کن که صف حلقوی پر است یا نه.
- گام 2: اگر صف پر باشد، اعلام خطا کرده و خارج شو.
- گام 3: اگر صف پر نبود، مقدار rear را یک واحد افزایش داده و باقیمانده تقسیم آن بر سایز صف را محاسبه کن و در rear قرار بده تا به فضای خالی بعدی اشاره کند.
- گام 4: عنصر داده مد نظر را در مکانی که rear به آن اشاره می کند، اضافه کن.

```
void queue:: add(int x)
{
  if (isFull()){
     cout<<"queue is full !";
      exit(); }
  else
    {
        rear = (rear +1) % SIZE;
        a[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

تمرین) توابع DelQueue وAddqueue به ترتیب توابع لازم برای حذف از صف خطی و درج درون صف خطی می باشند، اگر این توابع به ترتیب زیر فراخوانی گردند، مقدار نهایی متغییرهای A,B,C,D را مشخص نمایید.

A = 10 B = 2 C = 5 D = 20 (مقادیر اولیه)

Addqueue (C) Addqueue (A+C) Addqueue (D/A) A = Delqueue() C = Delqueue()

B = Delqueue() Addqueue (D/B) Addqueue (C/A) Addqueue (A+C) B = Delqueue()

D= Delqueue() A = Delqueue()

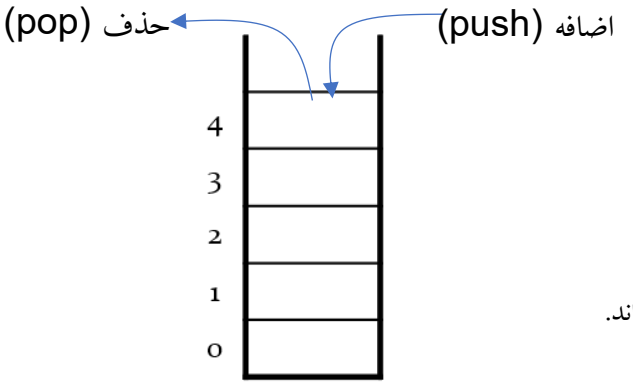
تمرین: اگر یک صف دایره ای را در آرایه ای به طول 15 پیاده سازی کنیم و rear = 13 و front = 7 باشد. تعداد عناصر صف را مشخص نمایید.

پشته (stack):

پشته یک نوع داده مجرد (ADT) است که در اکثر زبان‌های برنامه‌نویسی کاربرد رایجی دارد. دلیل این که این نوع داده، پشته نامیده شده، این است که از نظر ظاهری شبیه پشته است، یعنی به یک دسته از بشقاب‌های روی هم شباهت دارد.

در دنیای واقعی شما تنها می‌توانید بشقابهای جدید را روی دسته بشقابها بگذارید یا از روی آن بردارید. به طور مشابه ADT پشته نیز تنها از یک سمت امکان انجام عملیاتها بر روی داده ها را فراهم می‌کند. ما در هر زمان تنها به عناصر فوقانی پشته دسترسی داریم.

این ویژگی باعث می‌شود که پشته یک ساختار داده LIFO (Last-in-first-out) باشد یعنی ورودی-آخر- خروجی-اول. در پشته عملیات درج به نام عملیات PUSH نامیده و عملیات حذف به نام POP خوانده می‌شود. ساختمان داده پشته می تواند بوسیله ی آرایه ها و لیست پیوندی نمایش یابد. پشته می تواند دارای اندازه ثابت باشد و یا اندازه آن به طور دینامیک تغییر کند.



ADT پشته

مجموعه عناصر :

مجموعه ای از عناصر مرتب که از یک طرف (بالای پشته) قابل دستیابی اند.

عملیات اصلی :

creator : پشته خالی را ایجاد می کند.

isEmpty : خالی بودن پشته را بررسی می کند.

isFull : پر بودن پشته را بررسی می کند.

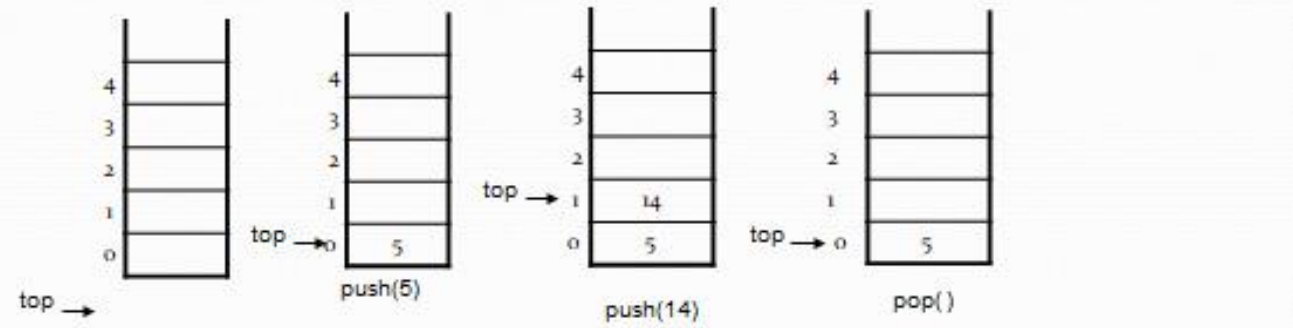
push : عنصری را به بالای پشته می افزاید.

pop : بالاترین عنصر پشته را حذف و ارائه می دهد.

peek : بالاترین عنصر پشته را بدون حذف آن ارائه می دهد.

ما باید یک متغییر برای مشخص کردن آخرین داده push شده به پشته در اختیار داشته باشیم. از آنجا که این اشاره‌گر همواره به عنصر فوقانی پشته اشاره می‌کند، نام آن top تعیین شده است. اشاره‌گر top بالاترین مقدار پشته را بدون حذف آن به دست می‌دهد.

top: به عنصر بالای پشته اشاره می کند. در ابتدا top=-1 است.



کلاس پیاده سازی پشته به زبان C++

```
#define SIZE 5
class stack{
  private:
    int top;
    int a[SIZE];
  public:
    stack();
    int isEmpty();
    int isFull();
    void push(int x);
    int pop();
    int peek();
    void  display();
};
```

تابع سازنده ی پشته :

یک پشته ی خالی ایجاد می کند.

```
 stack:: stack()
  {
    top= -1;
  }
```

بررسی خالی بودن پشته

اگر هیچ عنصری در پشته وجود نداشته باشد، این تابع مفدار 1 و در غیر اینصورت مقدار 0 را بر می گرداند.

```
int stack:: isEmpty()
{
  if( top== -1)
    return 1;
  return 0;
}
```

بررسی پر بودن پشته

اگر همه ی مکانهای پشته پر باشند، این تابع مقدار 1 و در غیر اینصورت مقدار 0 را بر می گرداند.

```
int stack:: isFull()
{
  if( top== SIZE-1)
    return 1;
  return 0;
}
```

عملیات push

قرار دادن یک عنصر جدید در پشته را push می‌گویند. این عملیات شامل مراحل زیر است:

- گام 1: بررسی کن که پشته پر است یا نه.
- گام 2: اگر پشته پر باشد، اعلام خطا کرده و خارج شو.
- گام 3: اگر پشته پر نبود، مقدار top را یک واحد افزایش بده تا به فضای خالی بعدی اشاره کند.
- گام 4: عنصر داده مد نظر را در مکانی که top به آن اشاره می کند، اضافه کن.

```
void stack:: push(int x)
{
  if (isFull())
   { cout<<“stack is full”;
     exit(); }
  else
    a[++top] = x;
}
```

عملیات pop

دسترسی به محتوای بالای پشته در هنگام حذف آن را pop می‌گویند. در پیاده‌سازی پشته با استفاده از آرایه، عنصر داده‌ای واقعاً حذف نمی‌شود، بلکه تنها مقدار top یک واحد کاهش می‌یابد. اما در پیاده‌سازی پشته با استفاده از لیست پیوندی، واقعاً عنصر داده‌ای از فضای تخصیص یافته پشته پاک می گردد. یک عملیات pop می‌تواند شامل گام‌های زیر باشد:

- گام 1: بررسی کن که پشته خالی است یا نه؟
- گام 2: اگر پشته خالی بود، یک خطا اعلام کن و خارج شو.
- گام 3: اگر پشته خالی نبود، عنصر داده‌ای را که top مورد اشاره قرار می‌دهد برگردان.
- گام 4: مقدار top را یک واحد کم کن.

```
int stack:: pop()
{
  if (isEmpty ())
    { cout<<“stack is empty”;
      exit(); }
  else
    return a[top - -];
}
```

عملیات peek

دسترسی به محتوای بالای پشته بدون حذف آن را peek می‌گویند. این عملیات می‌تواند شامل گام‌های زیر باشد:

- گام 1: بررسی کن که پشته خالی است یا نه؟
- گام 2: اگر پشته خالی بود، یک خطا اعلام کن و خارج شو.
- گام 3: اگر پشته خالی نبود، عنصر داده‌ای را که top مورد اشاره قرار می‌دهد برگردان .

```
int stack:: peek()
{
  if (isEmpty())
   { cout<<“stack is empty”;
      exit(); }
  else
     return a[top] ;
}
```

مثال: فرض کنید یک پشته داشته باشیم که به ترتیب از پایین به بالا دارای عناصر 1، 2 و 3 باشد. وضعیت پشته را پس از انجام دستورات زیر از چپ به راست مشخص کنید.

pop(A): بالاترین عنصر موجود در پشته را حذف کرده و در متغیر A قرار می دهد.

push(A) : مقدار موجود در متغیر را در پشته قرار می دهد.

pop(x), pop(y), push(x), pop(z), pop(x), push(y), push(z)

pop(x)	pop(y)	push(x)	pop(z)	pop(x)	push(y)	push(z)
2		3				3
1	1	1	1		2	2
x=3	x=3	x=3	x=3	x=1		
y	y=2	y=2	y=2	y=2		
z		z	z=3	z=3		

مثال: مقدار نهایی متغییر های A و B و C را پس از اجرای دستورات زیر از چپ به راست مشخص کنید.

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), pop(A), pop(C), pop(B)

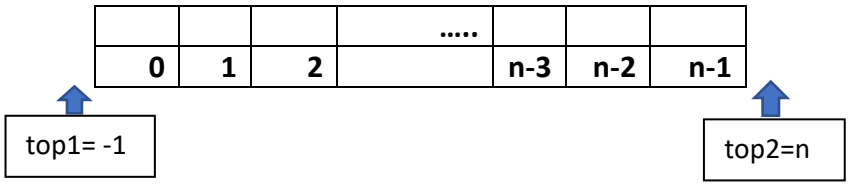
push(B)	push(A+B)	pop(C)	push(A-B)	push(C)	push(B)	pop(A)	pop(B)	Push(A*B)
					2			
				12	12	12		24
	12		8	8	8	8	8	8
2	2	2	2	2	2	2	2	2
A=10	A=10	A=10	A=10	A=10	A=10	A=2	A=2	A=2
B=2	B=2	B=2	B=2	B=2	B=2	B=2	B=12	B=12
C=5	C=5	C=12	C=12	C=12	C=12	C=12	C=12	C=12

push(C)	pop(A)	pop(C)	pop(B)
12			
24	24		
8	8	8	
2	2	2	2
A=2	A=12	A=12	A=12
B=12	B=12	B=12	B=8
C=12	C=12	C=24	C=24

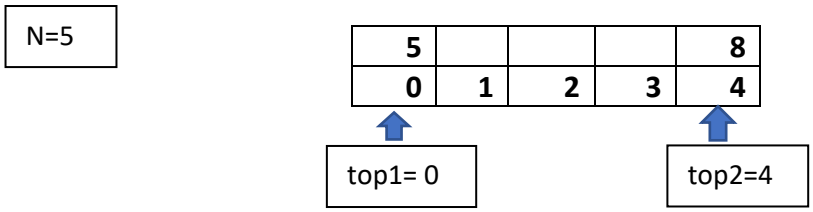
پشته دو گانه

برای پیاده‌سازی دو پشته در یک آرایه نیاز به دو متغیر برای نشان دادن بالاترین عنصر هر پشته می باشد. بالاترین عنصر پشته اول با متغیر top1 و بالاترین عنصر پشته دوم با متغیر top2 مشخص می گردد و این دو پشته در جهت عکس یکدیگر حرکت می کنند .

مقدار اولیه top1 = -1 و مقدار اولیه ی top2 = n می باشد.



اگر عنصری به پشته اول اضافه گردد top1 = top1 + 1 و اگر عنصری به پشته دوم اضافه گردد top2 = top2 -1



شرط پر بودن پشته دوگانه

top2 = top1 +1

اولویت عملگر

بطور کلی اولویت عملگرها به صورت زیر می باشد:

1. ( ) ++ و– پیشوندی
2. عملگر منطقی not – (منفی) توان
3. عملگر منطقی and * /
4. عملگر منطقی or + –
5. ++ و -- پسوندی

بین عملگرهایی که اولویت مساوی دارند عملگری زودتر محاسبه می گردد که سمت چپ باشد.

روش نمایش عبارات محاسباتی

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

مثال:

6 * 5 – 17 + 30 / 2 / 3 + 7

(6 * 5) – 17 + (30 / 2) / 3 + 7

((6 * 5) – 17) + ((30 / 2) / 3) + 7

تبدیل عبارات میانوندی به پسوندی و پیشوندی بدون استفاده از پشته

1. پرانتزگذاری
2. برای تبدیل به پیشوندی، عملگر درون هر پرانتز را به سمت چپ منتقل می‌کنیم.
3. برای تبدیل به پسوندی، عملگر درون هر پرانتز را به سمت راست منتقل می کنیم.
4. پرانتزها را حذف می کنیم.