DATA STRUCTURE

Shahrokh Amani

JUNE 14, 2026
EMPROR-DEV
Toronto, ON, Canada

فهرست مطالب

فصل 1

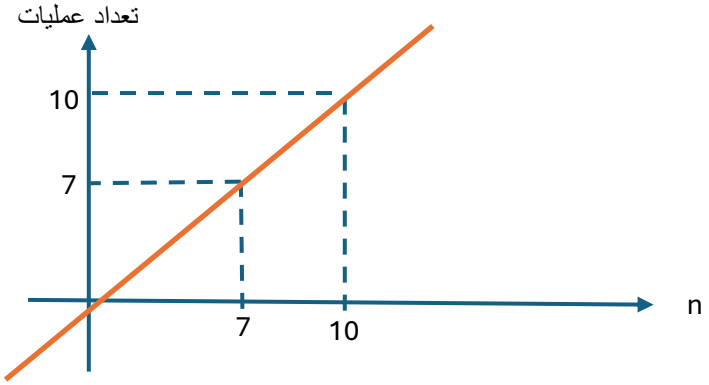
پیچیدگی اجرایی

پیچیدگی اجرائی: تابعی است که مدت زمان اجرای یک الگوریتم را بر حسب تعداد ورودی ها اندازه می گیرد.

مثال: در یک آرایه با 7 عنصر به صورت زیر داریم:

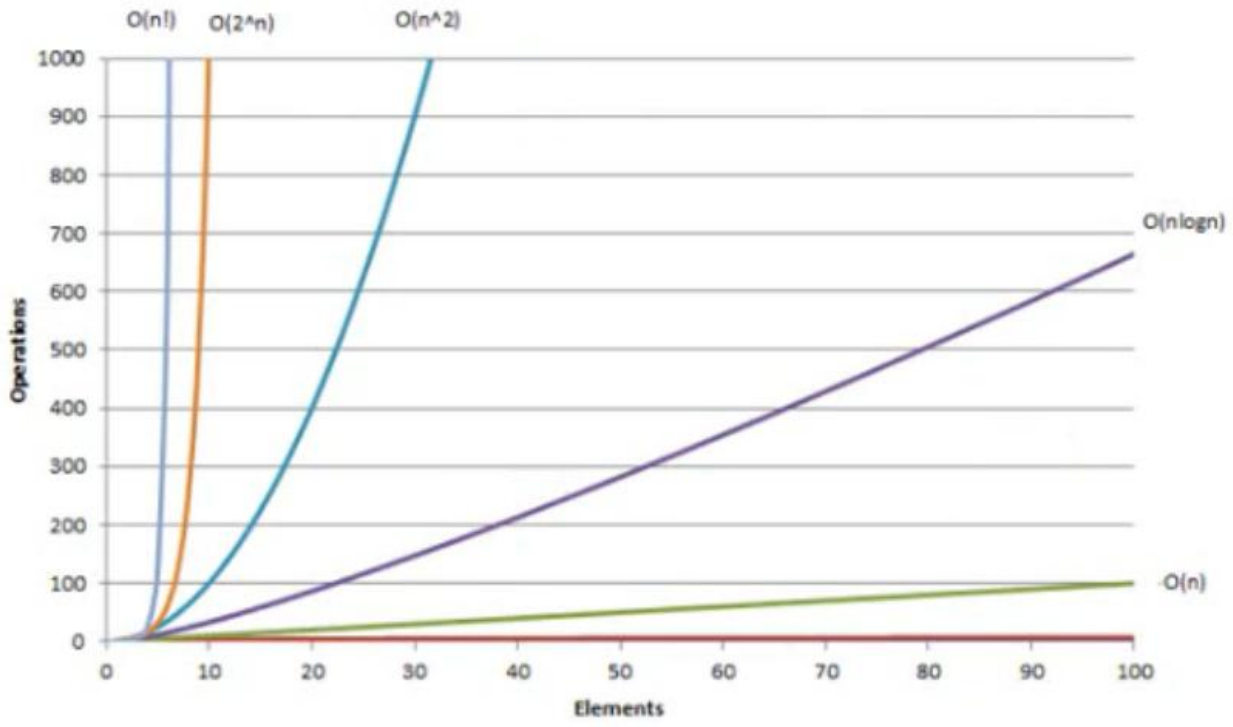
5	1	17	16	7	13	20
0	1	2	3	4	5	6

در این آرایه بنا داریم با استفاده از الگوریتم جستجوی خطی مقدار Key=13 را پیدا کنیم. برای این کار لازم است تا مقدار Key با همه درایه ها مقایسه شود تا محل مورد نظر را پیدا کنیم. در بد ترین حالت مقدار مورد نظر می تواند در خانه آخر(n) قرار داشته باشد که در این نوع الگوریتم می گوئیم پیچیدگی اجرایی از درجه n ((O(n) می باشد.



انواع پیچیدگی ها :

$$O(1) < O(\lg n) < O(n) < O(n \lg n) < O(n^2) < O(2^n) < O(n!) < O(n^n)$$



مرتبه اجرایی یک حلقه

$\text{for}(\text{int i} = a; i \leq b; i + k)$ $O(n)= \frac{b-a+1}{k}$

$Statements;$

مثال1:

$\text{for}(\text{int i} = 3; i <= n; i + 2)$ $= \frac{n-3+1}{2} == \frac{n-2}{2} = \frac{n}{2} - 1 => O(n)$

$Statements;$

مثال 2:

for(int i = a; i <= b; i *= k)
    $Statements$;

$log_k^b - log_k^a + 1$

for(int i = 1; i <= n; i *= 2)
    $Statements$;

$log_2^n - log_2^1 + 1$=O(log n)

مرتبه اجرایی در حلقه های پشت سر هم:

for(int i = 1; i <= n; i + +)
    $Statements$;
for(int j = 1; j <= m; j + +)
    $Statements$;

=➔ O(max (m, n)) OR O(m, n)

مرتبه اجرایی در حلقه های تو در تو:

مثال:

1-

```
for(int i = 1; i <= n; i++)
    for(int j = 1; j <= n; j++)        ➔        O(n^2)
        Statements;
```

2-

```
for(int i = 2; i <= n; i+= 4)
    for(int j = n; j > 3; j = j - 2)        ➔        (n-2+1)/4 * (n-4+1)/2 = O(n^2)
        Statements;
```

3-

```
for(int i = 1; i <= n; i *= 2)
    for(int j = 1; j < n; j++)        ➔ ( log_2^n - log_2^1 + 1)*n=O(nlog n)
        Statements;
```

مرتبه اجرائی حلقه تو در تو وابسته

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

در این مثال ابتدار می توانیم با فرمول ریاضی تعداد دفعات اجرای Statement را به صورت زیر محاسبه کرد:

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

از این فرمول همچنین می توان نتیجه گرفت که مرتبه اجرایی آن $O(n^2)$ می باشد.

فصل 2

آرایه ها

Array

مجموعه ای از داده های هم نوع که تحت یک نام معرفی می شوند. برای دسترسی به هر عنصر در آرایه لازم است اندیس آن به همراه نام آرایه عنوان شود.

```
int[] number=new int[5];
```

7	32	2	720	402
0	1	2	3	4

با تعریف دستور بالا یک آرایه تک بعدی با 5 عنصر ایجاد می شود که اندیس شروع از صفر می باشد. نوع هر درایه این آرایه نیز int می باشد.

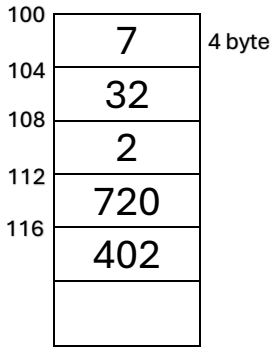
به منظور دسترسی به درایه 2 از این آرایه دستور زیر مورد استفاده قرار می گیرد.

```
number[2];
```

نحوه ذخیره عناصر آرایه در حافظه

number

7	32	2	720	402
0	1	2	3	4



عناصر آرایه به صورت پشت هم در حافظه ذخیره می شود. در نمونه بالا درایه آرایه از آدرس 100 شروع و در آدرس 116 به پایان می رسد. هر محل حافظه با توجه به نوع int روشن است 4 بایت را اشغال می کند که در شکل اعداد خانه های حافظه با اخلاف 4 به همین دلیل می باشد.

جهت بدست آوردن محل حافظه مورد نظر فرمول زیر مورد استفاده قرار می گیرد.

$x[l \dots u]$

$\alpha + (i - l) * w$

در این فرمول، α آدرس شروع، i اندیس شروع که همواره 0 است، l اندیس آرایه مد نظر و w تعداد بایت می باشد که در اینجا به دلیل نوع int این متغیر مقدار 4 را دارد.

مثال:

می خواهیم آدرس اندیس 2 آرایه را محاسبه کنیم. در این حالت داریم:

α =100, l=2,w=4➔ 100+2*4=108

آرایه های 2 بعدی

برای تعریف آرایه دو بعدی از دستور مشابه زیر استفاده می شود.

```
Int[,] numbers=new int[3,3];
```

	0	1	2
0	12	10	7
1	6	3	8
2	2	100	4

پس از اجرای دستور یک آرایه 3*3 ایجاد می شود که ما اعدادی را در آن فرض کرده ایم. جهت دسترسی به مقادیر موجود در درایه ها به صورت زیر از نامه آرایه و اندیس محل مورد نظر استفاده می شود.

```
Numbers[1,2]
```

نحوه ذخیره عناصر آرایه دو بعدی در حافظه

در آرایه های دو بعدی نیز درایه ها به صورت پشت هم در حافظه ذخیره می شود.

numbers

	0	1	2
0	12	10	7
1	6	3	8
2	2	100	4

100	12	4 byte
104	10	
108	7	
112	6	
116	3	
120	8	
124	2	
128	100	
132	4	

به منظور محاسبه آدرس یک درایه در حافظه فرموا زیر مورد استفاده قرار می گیرد.

$x[l_1 \dots u_1, l_2 \dots u_2]$

$\alpha + [(i - l_1) * (u_2 - l_2 + 1) + (j - l_2)] * w$

با توجه به اینکه در زبانهای بر پایه C اندیس آرایه ها از صفر شروع می شود، در نتیجه مقادیر l_1 و l_2 صفر می باشد که با این موضوع فرمول به صورت زیر ساده می شود.

$\alpha + [i * (u_2 + 1) + j] * w$

	l_2 → 0	1	u_2 → 2
l_1 → 0	12	10	7
1	6	3	8
u_1 → 2	2	100	4

در ماتریس بالا قصد داریم آدرس درایه numbers[1,2] را محاسبه کنیم در نتیجه مقدار i=1 و مقدار j=2 می باشد. همچنین w=4 و $\alpha = 100$ می باشد. در فرمول بالا همچنین مشخص است مقدار $u_2 = 2$ می باشد.

$$100 + [1 * (2 + 1) + 2] * 4 = 120$$

الگوریتم جستجوی خطی در آرایه

الگوریتم جستجوی خطی یا Linear search یعنی ما به صورت خطی و یکی‌یکی توی یه لیست یا آرایه دنبال یک داده می‌گردیم. مثلاً وقتی یه آرایه از عددها یا هر نوع داده دیگه‌ای داریم و می‌خواهیم ببینیم یه عدد یا داده خاص در آن آرایه هست یا خیر، از اولین درایه شروع می‌کنیم و تک‌تک چک می‌کنیم تا به داده مورد نظر برسیم . اگه داده را پیدا کردیم، جستجو به پایان می رسد ولی اگه نه، باید به جستجو ادامه بدیم تا به آخر آرایه برسیم. پیچیدگی زمانی الگوریتم جستجوی خطی برابر با O(n) است.

مثال: با فرض آرایه داده شده در ذیل قصد داریم عدد key=720 را جستجو کنیم.

7	32	2	720	402
0	1	2	3	4

7	32	2	720	402
0	1	2	3	4

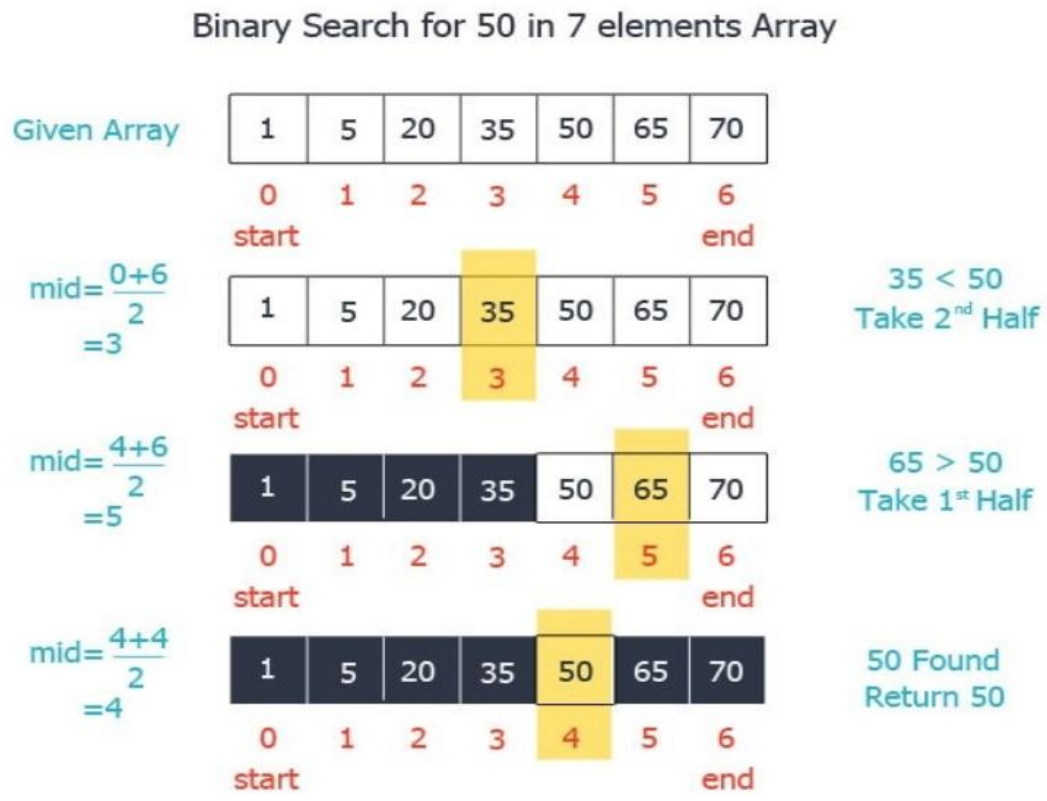
7	32	2	720	402
0	1	2	3	4

7	32	2	720	402
0	1	2	3	4

7	32	2	720	402
0	1	2	3	4

الگوریتم جستجوی دودوئی

الگوریتم جستجوی دودویی (Binary Search) در هر گام از جستجو محدوده جستجو را به نصف کاهش می دهد. روش کار به این صورت است که ابتدا عنصر مورد نظر با عنصر خانه وسط $\left(\left\lfloor\frac{start+end}{2}\right\rfloor\right)$ مقایسه می شود، اگر با مقدار این خانه برابر بود (بهترین حالت) جستجو تمام می شود. اگر مقدار عنصر مورد نظر از مقدار عنصر خانه وسط بزرگتر بود، بخش بالایی به دو قسمت تقسیم می شود و جستجو در بخش بالایی آرایه ادامه می یابد. اما اگر مقدارش از مقدار عنصر خانه وسط کوچکتر بود، بخش پایینی به دو قسمت تقسیم می شود و جستجو در بخش پایینی آرایه ادامه پیدا می کند. این عملیات تا زمانی که همه عناصر آرایه بررسی شوند، یا عنصر مورد نظر پیدا شود، ادامه می یابد.



** پیچیدگی زمانی این الگوریتم، O(Log n) است و فقط در آرایه های مرتب شده می تواند مورد استفاده قرار گیرد.

حذف عنصر از آرایه

به منظور حذف یک عنصر از یک آرایه با دو رویکرد مواجه هستیم. اول اینکه آرایه مرتب باشد و دیگری اینکه آرایه نامرتب باشد.

اگر بخواهیم عنصری را از یک آرایه نامرتب حذف کنیم، ابتدا لازم است عنصر مورد نظر را در آرایه پیدا کنیم. با توجه به اینکه آرایه نامرتب می باشد، از الگوریم جستجوی خطی استفاده می کنیم.

پس از بدست آمدن محل عنصر مورد نظر در مرحله بعد، عناصر بعد محل یافت شده، یکی یکی به سمت چپ شیفت داده می شوند.

مثال:

در آرایه زیر تصمیم داریم مقدار 35 را از داخل آرایه حذف کنیم. با استفاده از الگوریتم جستجوی خطی محل این مقدار که 4 است پیدا می شود و سپس عملیات شیفت انجام می شود.

1	20	5	50	35	70	65
0	1	2	3	4	5	6

در حذف عنصر از آرایه مرتب تنها تفاوتی که وجود دارد این است که به جای الگوریتم جستجوی خطی از الگوریتم Binary Search استفاده می شود.

درج عنصر در آرایه

در عملیات درج، نیز دو دیدگاه وجود دارد که مرتب بودن یا نامرتب بودن آرایه را در نظر می گیرد.

درج عنصر در آرایه نامرتب

اگر آرایه ای نامرتب داشته باشیم و بخواهیم عنصری را در آرایه قرار دهیم کافیست مقدار مورد نظر را در انتهای آرایه قرار دهیم.

مثال:

در آرایه با نام names که در زیر می بینید گنجایش آرایه c=7 می باشد و تعداد عناصر موجود در آرایه n=5 میباشد.

1	20	5	50	35		
0	1	2	3	4	5	6

حال می خواهیم مقدار key=13 را در آرایه قرار دهیم. برای این کار کافیست دستور ساده زیر را اجرا کنیم.

```
Numbers[n]=key;
n++;
```

بدیهی است اگر c=n باشد، آرایه پر می باشد و امکان درج نمی باشد.

درج عنصر در آرایه مرتب

برای توضیح این رویکرد آرایه مرتب زیر با نام names را فرض می کنیم.

1	5	20	35	50		
0	1	2	3	4	5	6

مشخص است که گنجایش این آرایه c=7 می باشد و تعداد عناصر موجود در آرایه نیز n=5 می باشد. می خواهیم مقدار key=22 را در آرایه قرار دهیم. برای این کار عنصر آخر آرایه (names[n-1]) را با مقدار key مقایسه می کنیم. اگر این عنصر بزرگتر از key بود به سمت راست شیفت داده می شود. اینجا چون 22<50 است این شیفت انجام می شود.

1	5	20	35	50	50	
0	1	2	3	4	5	6

در مرحله بعد این عمل برای مقدار قبلی یعنی 35 انجام می شود، و چون باز 22<35 می باشد 35 هم به سمت راست شیفت داده می شود.

1	5	20	35	35	50	
0	1	2	3	4	5	6

در مرحله بعد این فرایند تکرار می شود با این تفاوت که 20<22 می باشد و عمل درج می تواند انجام شود.

1	5	20	22	35	50	
0	1	2	3	4	5	6

ادغام دو آرایه

در این بخش می خواهیم اصول ادغام دو آرایه را بررسی کنیم. بر این اساس دو آرایه داریم که یکی با n1=4 عنصر در آن و دیگری با n2=3 عنصر که می خواهیم عمل ادغام را انجام دهیم.

			22	35	50	60
0	1	2	3	4	5	6

7	13	60
0	1	2

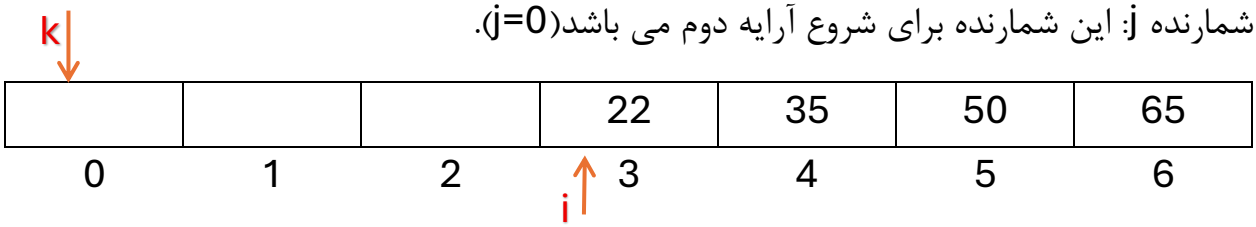
همانطور که مشخص است 3 عنصر در ابتدای آرایه اول
خالی می باشد. دلیل این کار این است که ما ادغام این دو آرایه را در آرایه اول انجام می دهیم و این کار باعث می شود فضای مورد نیاز برای ادغام وجود داشته باشد.

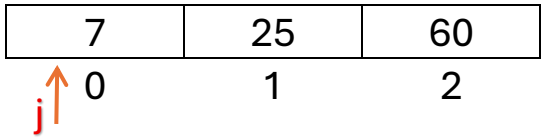
برای شروع عمل ادغام 3 شمارنده برای آرایه ها تعریف می کنیم. دو شمارنده برای آرایه اول و یک شمارنده برای آرایه دیگر می باشد.

شمارنده k: این شمارنده برای شروع آرایه اول می باشد و مقدار k=0 می باشد تا برای عمل درج مورد استفاده قرار بگیرد.

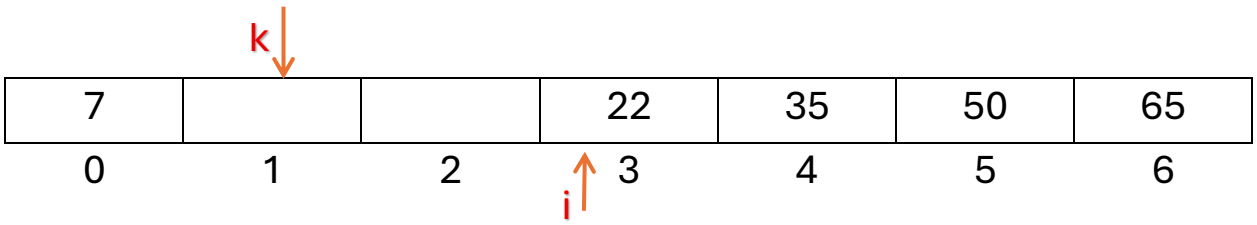
شمارنده i: این شمارنده به اولین عنصر موجود در آرایه اول اختصاص دارد و مقدار i=n2 می باشد.

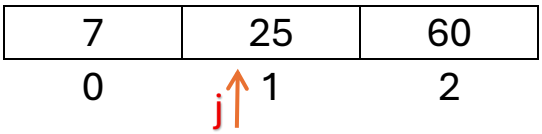
شمارنده j: این شمارنده برای شروع آرایه دوم می باشد(j=0).





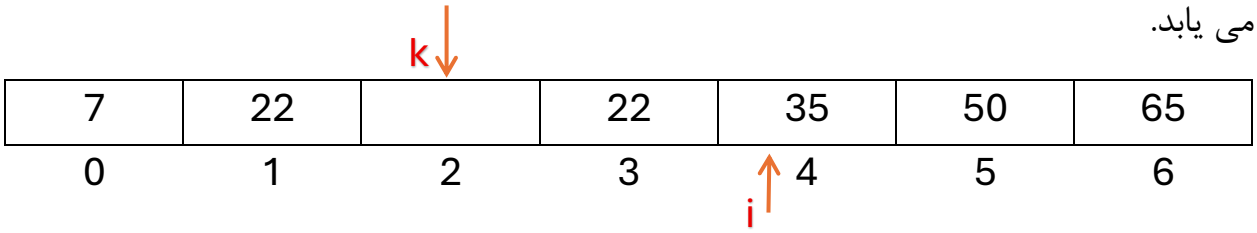
در اولین گام مقادیر موجود در درایه های i و j (22 و 7) با هم مقایسه می شود. اگر مقدار موجود در درایه j کوچکتر باشد به محل درایه k منتقل می شود و مقدار j و k یک عدد اضافه می شوند. در غیر اینصورت مقدار درایه j منتقل می شود و مقادیر k و i یک واحد اضافه می گردد.

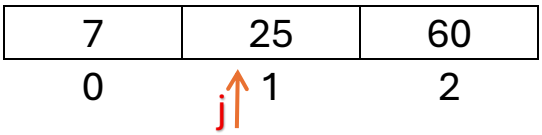


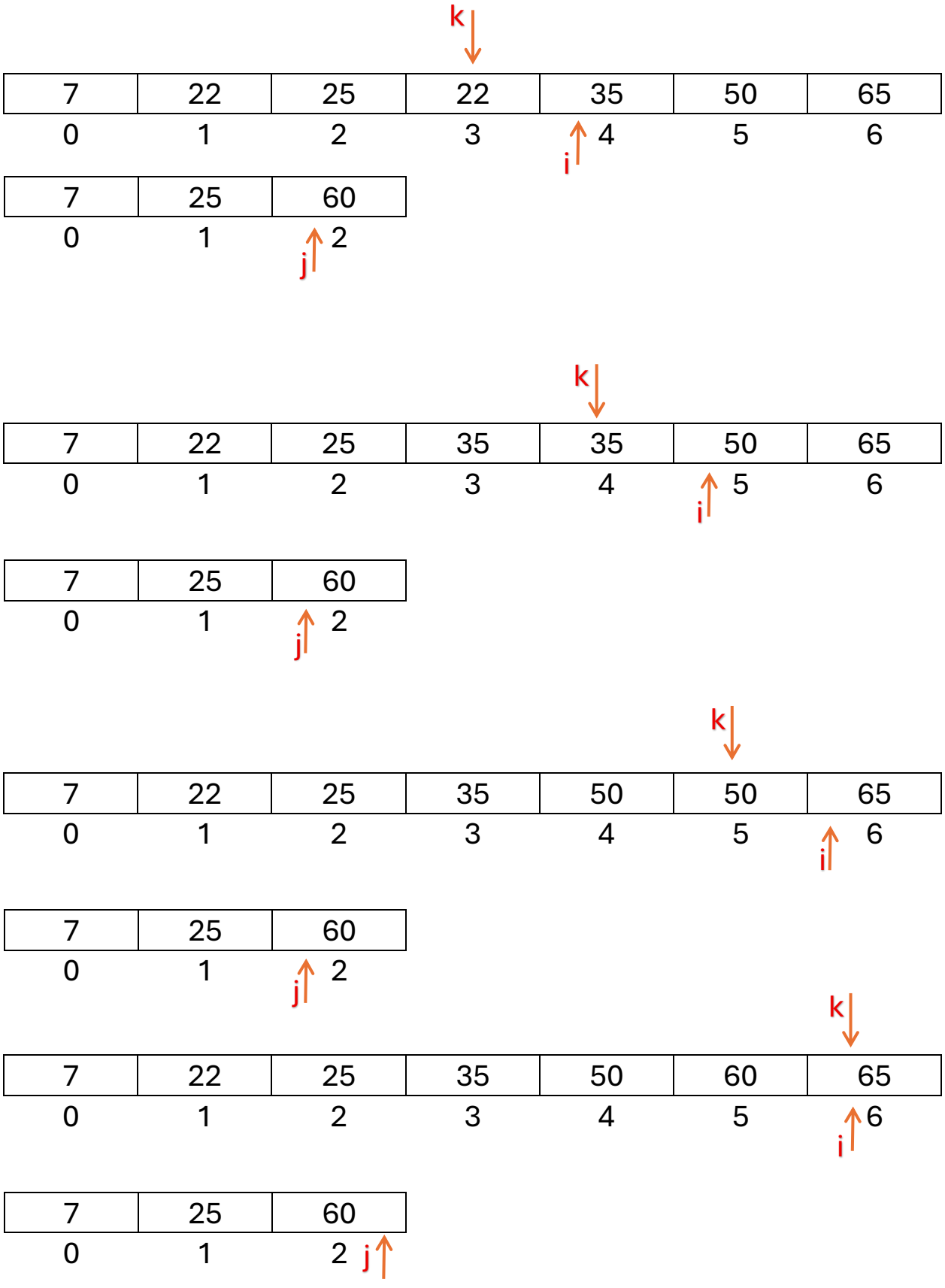


در مرحله اول همانطور که مشخص است در مقایسه مقادیر موجود در درایه i و j با توجه به اینکه عنصر موجود در درایه j کوچکتر می باشد، مقدار 7 به درایه k=0 منتقل شد و مقادیر k و j افزایش یافت.

در مرحله دوم همین روال ادامه می یابد و می بینیم که مقدار موجود در درایه j بزرگتر از مقدار موجود در درایه i می باشد. پس مقدار موچکتر به محل k اضافه می شود و مقادیر k و i افزایش می یابد.






k
7 22 25 22 35 50 65
0 1 2 3 4 5 6
i
7 25 60
0 1 2
j
k
7 22 25 35 35 50 65
0 1 2 3 4 5 6
i
7 25 60
0 1 2
j
k
7 22 25 35 50 50 65
0 1 2 3 4 5 6
i
7 25 60
0 1 2
j
k
7 22 25 35 50 60 65
0 1 2 3 4 5 6
i
7 25 60
0 1 2
j

در نهایت دیده می شود که ادغام دو آرایه در آرایه اول انجام شده است و همچنین مرتب بودن آن نیز برقرار می باشد.

7	22	25	35	50	60	65
0	1	2	3	4	5	6

فصل 3

صف

Queue

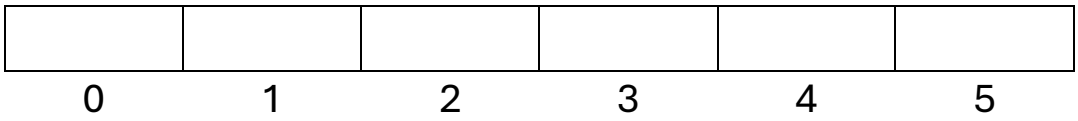
صف در ساختمان داده یکی از مهم‌ترین و اصلی‌ترین مفاهیم در حوزه ساختمان داده و برنامه‌نویسی است. در حقیقت صف در ساختمان داده نیز مانند صف‌های انسانی است و به همان صورت نیز کار می‌کند. یک صف سینما را در نظر بگیرید . اولین کسی که در صف بایستد، زودتر از بقیه می‌تواند بلیط خود را خریداری کند و از صف خارج شود(First In-First Out). در ساختمان داده نیز با چنین چیزی رو به رو هستیم . صف دارای یک ابتدا (Front) است که به آن سر صف گفته می‌شود و خروج داده‌ها از این ناحیه انجام می‌شود. همچنین صف دارای یک انتهای (Rear) نیز هست که داده‌ها از این ناحیه وارد صف می‌شوند.

عملیات مربوط به صف در ساختمان داده

- enQueue(item) : این عمل یک عنصر جدید را به صف اضافه میکند. مشاهده می کنید که عنصر جدید به عنوان پارامتر ورودی این تابع تعریف شده است. در این تابع پس از اضافه شدن عنصر طبیعی است که می بایست مقدار متغییر rear را بروز رسانی کند تا موقعیت انتهای لیست بدرستی تنظیم گردد.
- isEmpty() : این تابع خالی بودن صف را بررسی می کند و در نهایت مقدار True و یا False بر میگرداند و نیازی به پارامتر ورودی ندارد. روشن است اگر این تابع مقدار true را بازگرداند به این معنا هست که صف خالی می باشد و خواندن عنصر وجود ندارد.

- isFull() : این تابع پر بودن صف را بررسی می کند و در نهایت مقدار True یا False بر میگرداند و نیازی به پارامتر ورودی ندارد. روشن است اگر این تابع مقدار true را بازگرداند به این معنا هست که ظرفیت صف پر می باشد و امکان اضافه کردن عنصر جدید وجود ندارد.
- deQueue() : این تابع عنصر ابتدای صف را حذف می کند و باز می گرداند. به پارامتر ورودی نیاز ندارد و مقدار ابتدای صف را باز می گرداند. روشن است با حذف عنصر ابتدای صف مقدار متغییرfront بروز رسانی می شود تا محل صحیح ابتدای صف برای عملیات بعدی بدست آید.

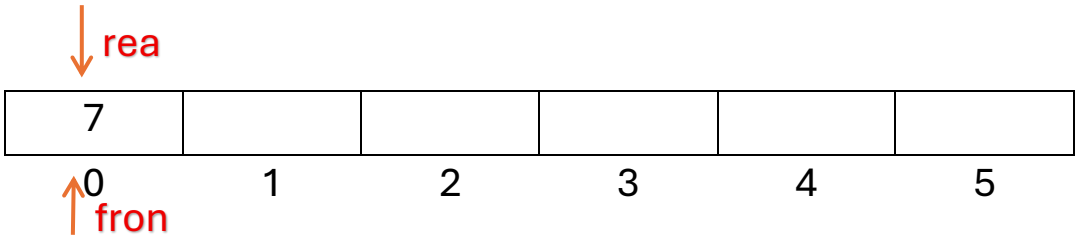
مثال: در این مثال یک صف با گنجایش c=5 داریم و عملیات مورد نظر را بر روی آن شرح میدهیم.



در مرحله اول یک صف خالی داریم که مقادیر متغییرهای rear و front را 1- درنظر میگیریم.

سپس یک عنصر به مقدار فرضی 8 به صف اضافه می کنیم.

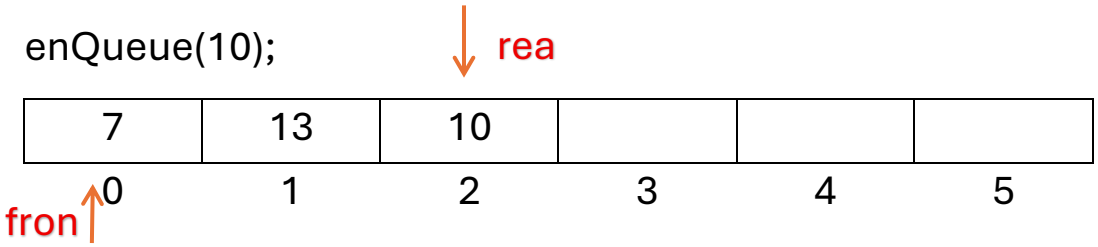
enQueue(7);



در مرحله به عنوان مثال دو عنصر به صف اضافه می کنیم.

enQueue(13);

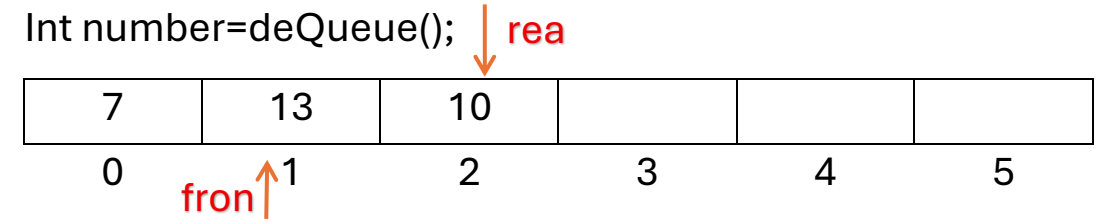
enQueue(10);



مشاهده می شود که به ازای هر عمل اضافه کردن به صف متغییر rear افزایش می یابد و موقعیت انتهای صف را مشخص می کند. مشخص است که برای هر عمل اضافه کردن باید پر بودن صف چک شود.

در گام بعدی عنصری را از ابتدای لیست برمی داریم.

Int number=deQueue();



با فراخوانی تابع حذف عنصر از صف مقدار را برداشته و متغییر front افزایش می یابد. روشن است قبل خواندن عنصر باید خالی بودن صف چک گردد.

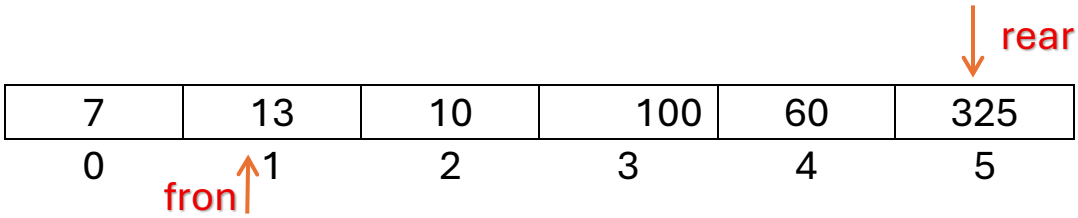
در مرحله بعد 4 عمل اضافه کردن به صف را درخواست می کنیم.

enQueue(100);

enQueue(60);

enQueue(325);

enQueue(11);

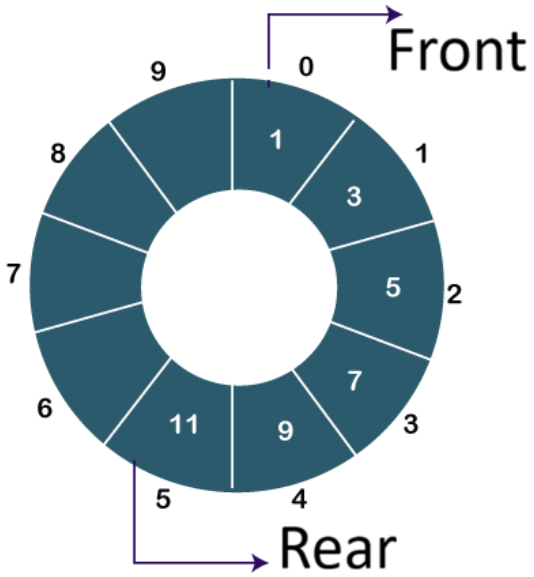


مشاهده می شود که 3 عمل اضافه کردن به لیست با موفقیت انجام می شود ولی عمل آخر به دلیل اینکه صف پر شده است با ارسال پیامی عدم موفقیت در ثبت مقدار در صف ارسال می گردد.

عمل خواندن از صف نیز تا جایی که صف خالی نباشد(front<=rear) ادامه می یابد.

صف چرخشی

صف حلقوی در ساختمان داده همان شکل و مشخصات صف خطی را دارد. از آرایه‌ها درست شده و با روش FIFO کار می‌کند. با این تفاوت که ابتدا و انتهای آن به یکدیگر متصل شده‌اند.



تمام توابع مربوط به صف خطی و کنترل های مربوط به آن را در این نوع از صف ها نیز داریم. فقط باید دقت شود با رسیدن به انتهای آرایه اصلی مقدار متغییر rear صفر می شود و به ابتدای لیست باز می گردد. در مرحله ای در کار اگر مقدار front و rear برابر شود صف پر شده است و امکان اضافه کردن به صف وجود ندارد. چرا؟

فصل 4

پشته

Stack

پشته، ساختمان داده‌ای خطی است که از اصل LIFO(Last In First Out) پشتیبانی می‌کند. آخرین ورودی اولین خروجی است. این عبارت به معنی این است که آخرین داده‌ای که وارد پشته می‌شود اولین داده‌ای خواهد بود که از پشته خارج می‌شود. زیرا پشته‌ها یک سر باز برای ورود و خروج داده‌ها بیشتر ندارند. پشته فقط یک نشانگر دارد که این نشانگر همیشه به بالاترین عنصر درون پشته اشاره می‌کند. هر وقت که عنصری به پشته اضافه شود در بالاترین خانه پشته قرار می‌گیرد و تنها عنصری که می‌تواند از پشته حذف شود همین بالاترین عنصر است.

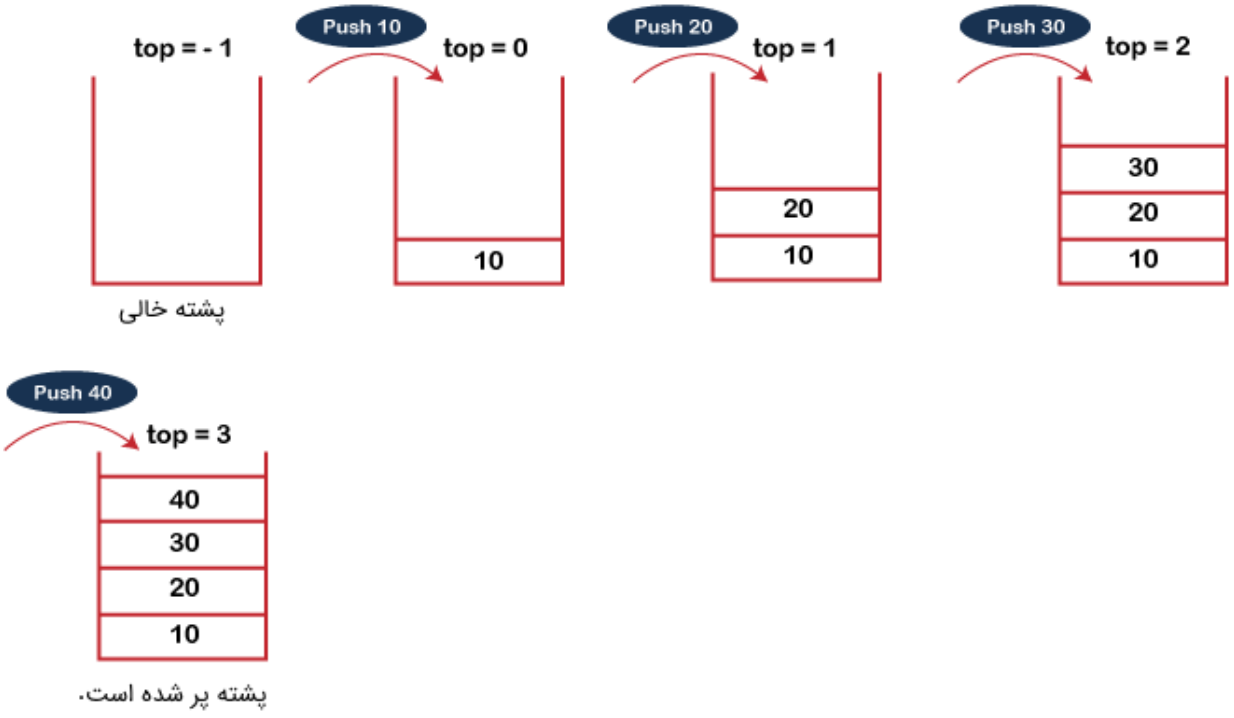
عملیات بر روی پشته

- push() : عملیات مربوط به اضافه کردن عنصرها به بالای پشته را شامل می شود. اگر پشته پر شده باشد، شرایط لازم برای احتمال وقوع سرریز (Overflow) فراهم شده است.
- pop() : این تابع بالاترین عنصر موجود در پشته را خارج میکند. اگر این تابع بر روی پشته خالی عمل کند زیر ریز (Underflow) رخ خواهد داد.
- isEmpty() این عملیات وضعیت خالی بودن یا نبودن پشته را مشخص می‌کند.
- isFull() این عملیات وضعیت پر بودن یا نبودن پشته را مشخص می‌کند.
- peek() : این عملیات عنصر بالای پشته را باز می گرداند بدون اینکه آن را از پشته حذف کند.

عمل مرتبط با تابع PUSH:

هر بار اجرای عملیات اضافه کردن عنصر به بالای پشته شامل مراحل است که در ذیل به بررسی آن خواهیم پرداخت.

1. قبل از وارد کردن هر عنصر به پشته باید بررسی کنیم که آیا پشته پُر شده یا نه برای این منظور از تابع isFull() استفاده می شود.پ
2. وقتی که پشته‌ای را مقداردهی اولیه می‌کنیم، باید مقدار بالاترین خانه پشته را برابر با -1 قرار دهیم(top=-1).
3. زمانی که عنصر جدیدی به پشته اضافه می شود، مقدار **top** افزایش پیدا می‌کند و سپس عنصر در جایگاه جدید مقدار **top** قرار می‌گیرد.
4. تا وقتی که به بیشترین اندازه (**Max** Size) پشته برسیم، می‌توان عناصر را در پشته وارد کرد.

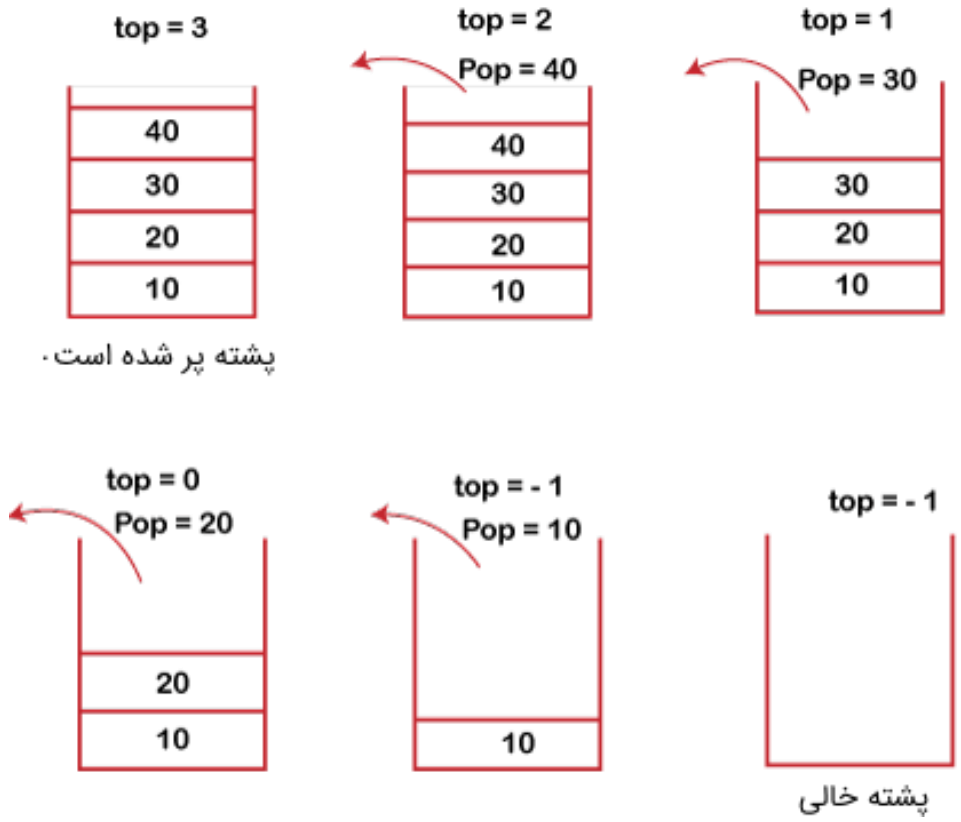


عمل مرتبط با تابع POP

هر بار اجرای عملیات حذف کردن عنصر از بالای پشته شامل مراحل است که در ذیل به بررسی آن خواهیم پرداخت.

ده‌ایم.

1. قبل از حذف کردن هر عنصر از پشته باید خالی بودن یا نبودن پشته را بررسی کنیم که این مهم توسط تابع isEmpty() انجام می گیرد.
2. اگر برای حذف کردن عنصری از پشته خالی تلاش کنیم حالت **Underflow** رخ می‌دهد.
3. اگر پشته خالی نباشد، در ابتدا به عنصری که در خانه **top** پشته قرار گرفته دسترسی پیدا می‌کنیم و سپس از مقدار **top** یک واحد کم می‌شود.



کاربردهای پشته:

1- تابع بازگشتی

یکی از کاربردهای پشته، توابع بازگشتی می باشد که در این توابع، هر تابع خودش را فراخوانی می کند و فراخوانی های آن به کمک پشته انجام می شود. در زیر به بررسی نحوه عملکرد آن می پردازیم.

- **تابع بازگشتی فاکتوریل**

فاکتوریل عملگری است که در آن یک عدد در اعداد قبل خود تا یک ضرب می شود. علامت آن ! می باشد و فاکتوریل عدد 0 مقدار 1 می باشد.

$$n! = n * (n-1) * (n-2) * (n-3) * \dots * 2 * 1$$

مثال:

5!=5*4*3*2*1=120

این عملگر را می توان به صورت بازگشتی نوشت:

```
public static int Factorial(int n)
{
    if (n == 1)
        return 1;
    else
        return n * Factorial(n - 1);
}
```

در کد بالا مشاهده می شود که تابع Factorial در داخل خودش فراخوانی می شود تا مقدار n=0 شود. و نحوه محاسبه فاکتوریل در این کد به شکل زیر می باشد.

$$Factorial(5) = 5 * Factorial(4)$$
$$5 * 4 * Factorial(3)$$
$$5 * 4 * 3 * Factorial(2)$$
$$5 * 4 * 3 * 2 * Factorial(1)$$
$$5 * 4 * 3 * 2 * 1 = 120$$

- **سری فیبوناچی**

این سری با دو عدد 0 و 1 شروع می شود و عناصر دیگر با حاصل جمع دو عنصر قبلی بدست می آید.

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

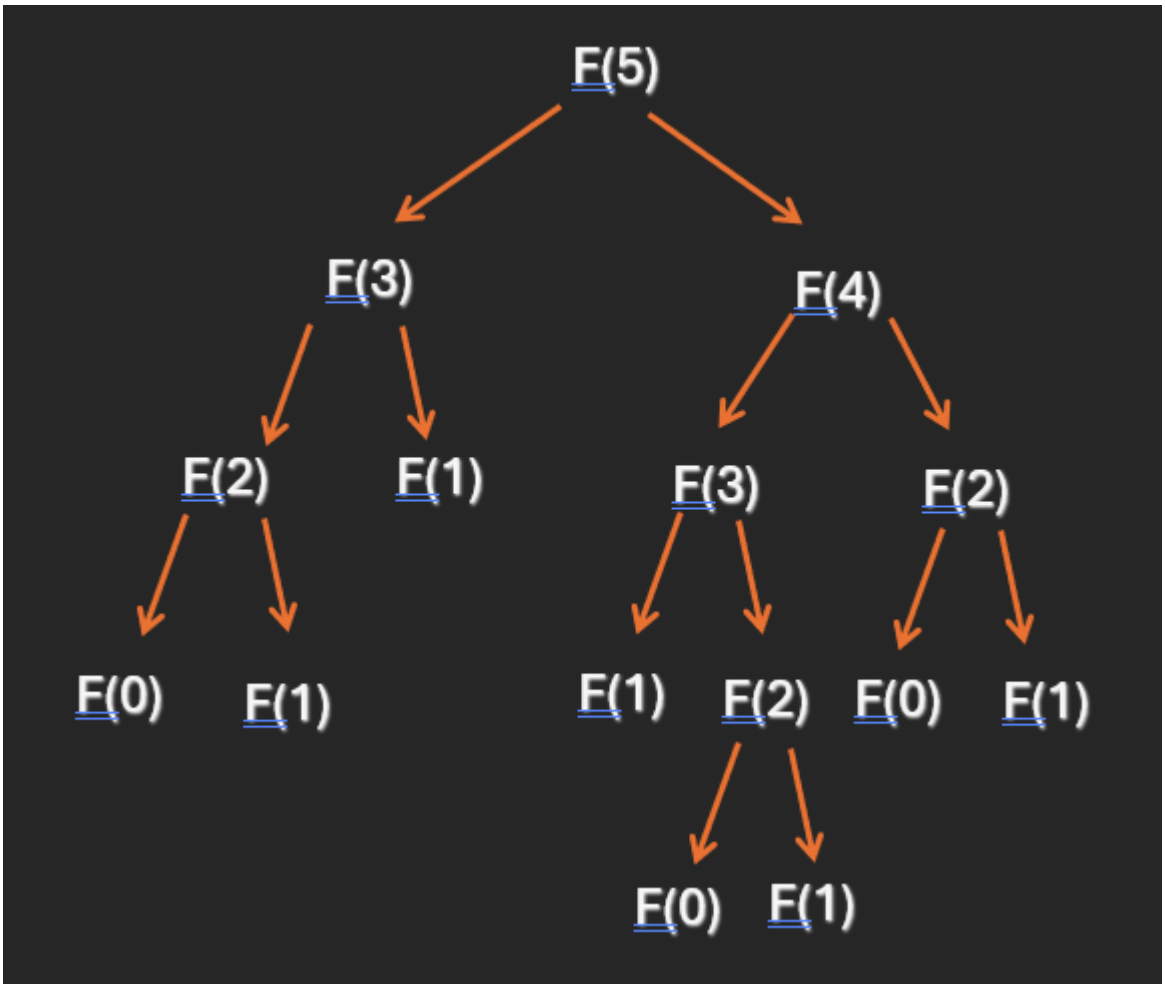
```
public static int Fibo(int n)
{
    if (n == 0 || n == 1)
        return n;
    else
        return Fibo(n - 1) + Fibo(n - 2);
}
```

تابع بالا روشن است که می توان جمله n ام این سری را محاسبه کرد و دیده می شود که در داخل تابع Fibo این تابع خودش را فراخوانی می کند.

حال به منظور درک بهتر مسئله می خواهیم با ترسیم درخت عملکر این تابع را بررسی کنیم.

0	1	1	2	3	5	8
F(0)	F(1)	F(2)	F(3)	F(4)	F(5)	F(6)

در بالا سری فیبوناچی را تا عدد 8 داریم و توابع هر کدام زیر آن نوشته شده است. حال درخت بازگشت را برای یافتن F(5) ترسیم می کنیم.



در درخت با مشاهده می شود که به ازای مقدار n چه توابعی فراخوانی می شود و این فراخوانی ها تا جایی ادامه می کند که به F(0) و یا F(1) برسیم و سپس در درخت مقادیر آنها جایگذاری می شود. اعداد هر سطح با هم جمع می شوند و حاصل تابع پدر بدست می آید. این کار ادامه پیدا می کند تا به سطح ریشه (F(5)) برسیم.

2- بررسی خروجی برنامه های بازگشتی

یکی از مسائل جالب برای بررسی آن است که بدانیم نحوه اجرای دستوراتی که بعد از فراخوانی تابع بازگشتی هستند به چه صورت می باشد.

```
public static int Example(int n)
{
    if (n == 3)
        return 0;
    else
    {
        n = n + 1;
        Example(n);
        Console.Write(n);
        return 0;
    }
}
```

در تابع بالا مشاهده می شود که دستور چاپ مقدار n بعد از تابع بازگشتی آمده است و ما می خواهیم بررسی کنیم که خروجی این برنامه به چه صورت است. به همین منظور مقدار اولیه n=1 را فرض می کنیم.

وقتی n برابر 3 نمی باشد، به بخش else میرویم و با افزایش مقدار n تابع مجدد با مقدار n=2 فراخوانی می شود و دستور بعد فراخوانی در پشته Push می شو د.

Console.Write(2)

در فراخوانی بعد هم همین روال انجام می شود و با افزایش n تابع به ازای n=3 فراخوانی می شود و دستور بعد فراخوانی باز در پشته قرار می گیرد.

Console.Write(3)
Console.Write(2)

در دور بعد چون n=3 می باشد، فراخوانی اتفاق نمی افتد ولی در اینجا عناصر موجود در پشته Pop می شوند و در خروجی اعمال می گردند که اول چاپ 3 و بعد 2 می باشد.

مثال: خروجی برنامه زیر را تعیین کنید.

```
public static int Example(int n)
{
    if (n >= 3)
    {
        Example(n-1);
        Console.Write('S');
        Console.WriteLine(n);
        Console.WriteLine('H');
        return 0;
    }
    return 0;
}
```

برای بررسی مقدار n=3 را اختیار می کنیم. در این حالت شرط موجود برقرار می باشد و وارد بدنه می شویم و تابع با مقدار n=2 مجدد فراخوانی می شود. همانطور که گفته شد دستورات بعد فراخوانی داخل پشته قرار می گیرد. اینجا که سه خط کد بعد فراخوانی داریم هر سه دستور وارد پشته می شود. البته باید توجه شود که این دستورات از آخر وارد پشته می شوند.

Console.Write(‘S’)
Console.Write(3)
Console.Write(‘H’)

حال خروجی زیر با pop شدن عناصر موجود در پشته به صورت زیر می باشد.

S3H

3- بررسی نماد گذاری

نماد گذاری میانوندی(Infix)

در نمادگذاری میانوندی برای نوشتن عبارت‌های حسابی عملگرهای مورد استفاده میان عملوندها قرار گرفته باشند.

مثال:

A+B

نماد گذاری پسوندی (Postfix)

این روش نمادگذاری که به نام لهستانی معکوس نیز نامیده می‌شود، عملگر پس از عملوند قرار می‌گیرد.

مثال:

AB+

نماد گذاری پیشوندی (Prefix)

این روش نمادگذاری که به نام نمادگذاری لهستانی نیز شناخته می‌شود، عملگر پیش از عملوند نوشته می‌شود.

مثال:

+AB

تبدیل عبارت Infix به عبارت Postfix

a+(b*c)-d^e

در اولین گام عبارا داخل پرانتز اولویت بیشتری دارد پس اولین تبدیل در پرانتز انجام می شود.

a+bc*-d^e

در گام بعد چون عملگر توان(^) اولویت بیشتری دارد تبدیل روی آن انجام می پذیرد.

a+bc*-de^

میدانیم که - و + دارای اولویت یکسان هستند پس عملگری که در سمت چپ می باشد اولویت بیشتری پیدا می کند.

abc*+-de^

و در نهایت عملگر – تبدیل می گردد.

abc*+de^-

تبدیل عبارت Infix به عبارت Postfix به کمک پشته

برای انجام این تبدیل از ابتدای رشته ورودی (عبارات infix) شروع می کنیم و هر عملوندی را که دریافت کردیم در آرایه یا رشته خروجی(عبارت postfix) قرار می دهیم و عملگر ها را در داخل پشته قرار می دهیم.

قانون 1: هیچ عملگر با اولویت پایین تر نمی تواند بر روی عملگر با اولویت بیشتر یا مساوی قرار بگیرد. در این حالت عناصر موجود در پشته آنقدر از پشته خارج می شود تا عملگر خوانده شده از رشته ورودی بتواند در پشته قرار بگیرد.

قانون 2: پرانتز باز در پشته قرار میگیرد.

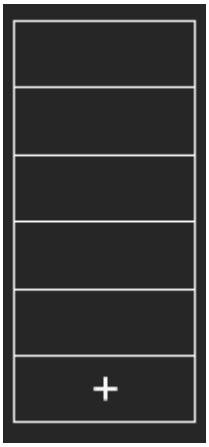
قانون 3: با خواندن پرانتز بسته، عناصر داخل پشته تا رسیدن به پرانتز باز در رشته و یا آرایه خروجی قرار می گیرند و پرانتز باز و بسته دور ریخته می شود.

قانون 4: پس از خوانده شدن آخرین عنصر در آرایه ورودی عناصر داخل پشته در آرایه خروجی نوشته می شوند.

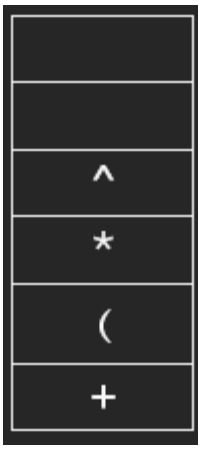
مثال: مطلوب است تبدیل عبارت infix زیر به عبارت Postfix

a+(b*c^d)^(e-f)/g

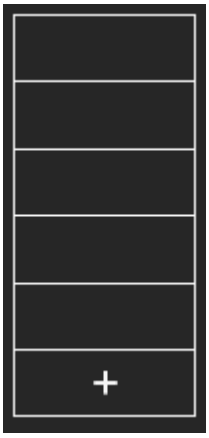
از ابتدا شروع می کنیم و عنصر a مستقیم در خروجی قرار میگیرد. سپس به جلو حرکت می کنیم و + در پشته قرار می گیرد.



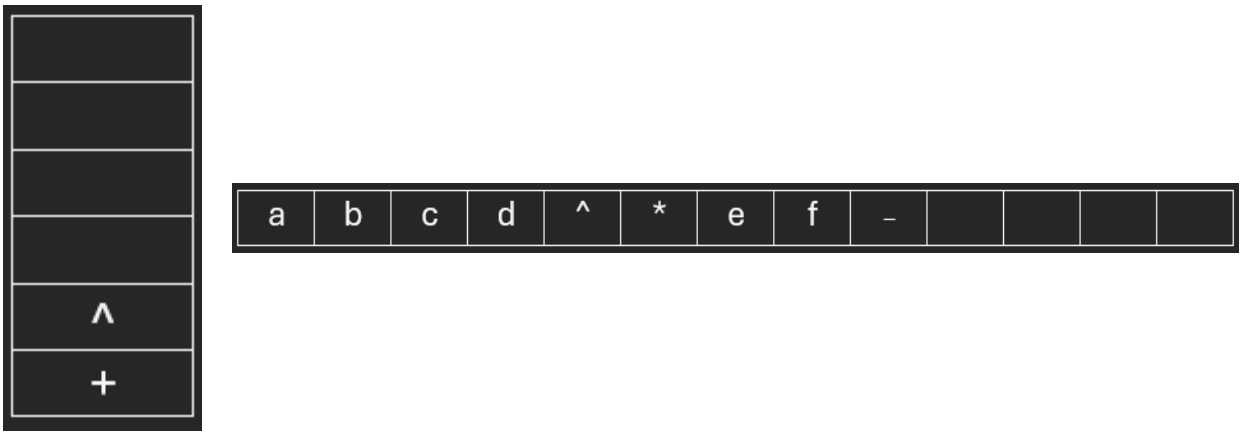
پرانتز خوانده شده و در داخل پشته قرار میگیرد و b داخل خروجی منتقل می شود. سپس * می تواند در پشته قرار بگیرد. C داخل خروجی می رود و ^ نیز می تواند در پشته قرار بگیرد و d نیز در خروجی نوشته می شود.



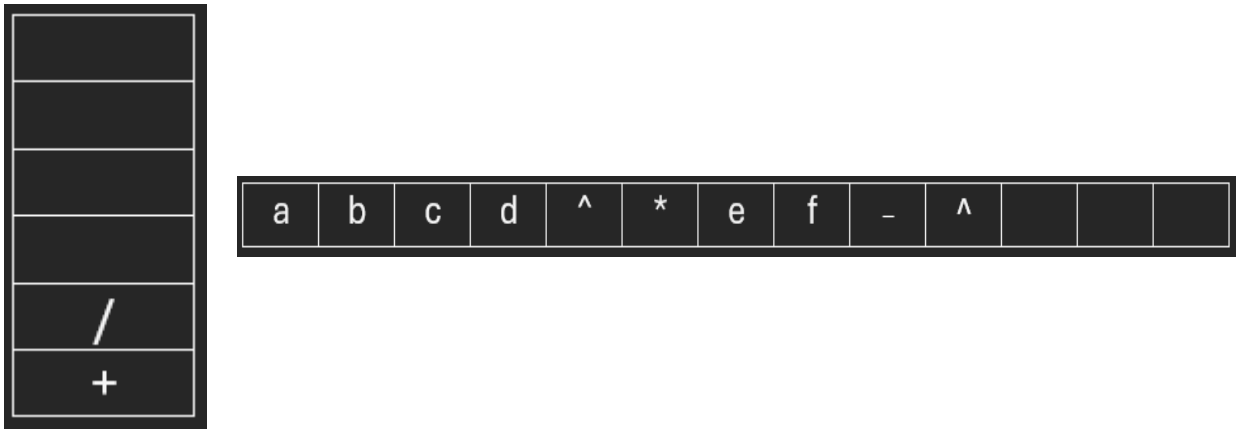
در این مرحله چون پرانتز بسته از رشته ورودی خوانده می شود، تمام عناصر موجود در پشته تا رسیدن به پرانتز باز خارج شده و در رشته خروجی قرار می گیرند و پرانتز باز هم که دور ریخته می شود.



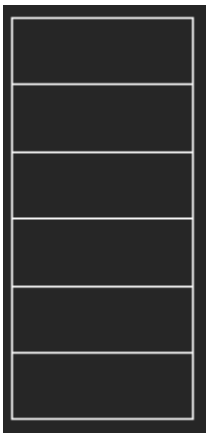
در ادامه علامت ^ خوانده می شود که در پشته قرار می گیرد. سپس) نیز در پشته قرار می گیرد و e داخل خروجی می رود و سپس – در پشته قرار می گیرد. F به رشته خروجی منتقل می شود. با خواندن پرانتز بسته همه عناصر موجود در پشته تا پرانتز باز در خروجی قرار می گیرند.



در مرحله بعد / خوانده می شود و چون اولویت کمتری از ^ دارد نمی تواند بر روی آن در پشته قرار بگیرد، پس ^ از پشته حذف می شود و به رشته خروجی می رود و حالا چون / اولویت بیشتری نسبت به + دارید در پشته اضافه می شود.



آخرین عنصر موجود در رشته ورودی یعنی g خوانده شده و در رشته خروجی قرار می گیرد و با توجه به اینکه به انتهای رشته ورودی رسیده ایم، عناصر موجود در پشته در رشته خروجی قرار می گیرند.



نتیجه تبدیل عبارت infix به postfix در خروجی در دسترس است.

a+(b*c^d)^(e-f)/g ➔ abcd^*ef-^g/+

ارزیابی عبارتهای Postfix

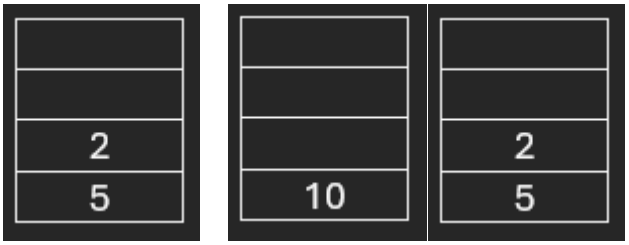
در این بخش می خواهیم بررسی کنیم که چطور می توانیم حاصل یک عبارت postfix را محاسبه کنیم.

مثال: مطلوب است حاصل عبارت زیر:

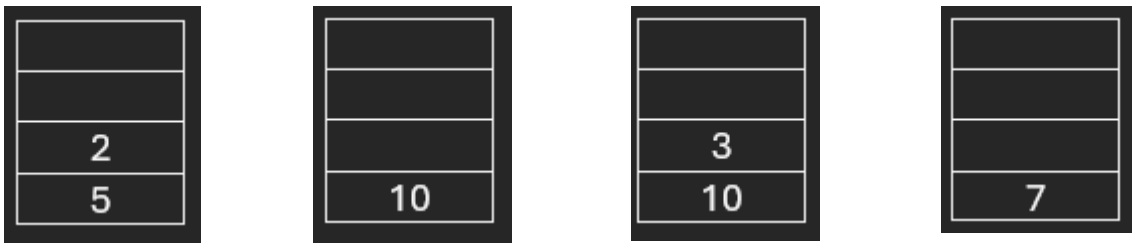
5,2,*,3,-,4,1,+,*

برای ارزیابی این نوع عبارات، از ابتدای عبارات عملوند ها داخب پشته قرار می گیرند.

با رسیدن عملگر * دو عنصر داخل پشته خوانده شده و در هم ضرب می شوند و نتیجه در پشته قرار می گیرد.

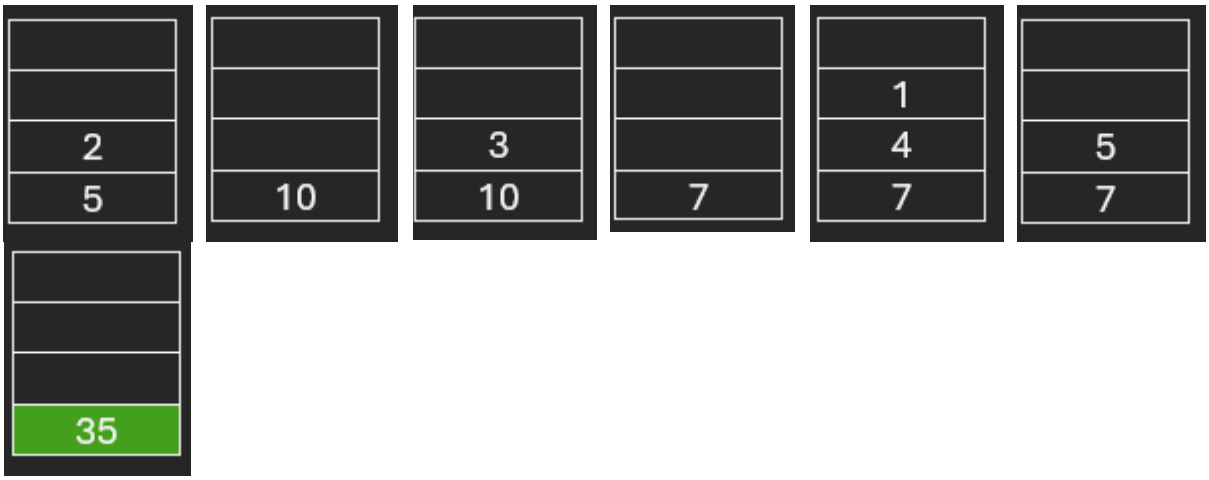


سپس مقدار 3 خوانده شده و داخل پشته قرار می گیرد. پس از آن عملگر – خوانده می شود که در نتیجه آن دو مقدار موجود در پشته خوانده شده و نتیجه محاسبه مجدد در پشته قرار می گیرند.



در مرحله بعد دو عملوند 4 و 1 خوانده شده و در پشته قرار می گیرند. و با خواندن عملگر + دو عنصر بالای پشته خوانده شده با با این عملگر جمع می شوند و نتیجه مجدد در پشته قرار می گیرد.

در نهایت عملگر * خوانده می شود و مقادیر موجود در پشته خوانده می شوند و در هم ضرب می شوند و نتیجه در پشته قرار می گیرد که این عدد نتیجه نهایی این عبارت می باشد.

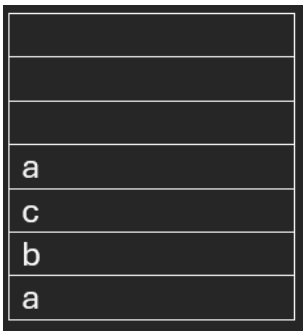


تبدیل عبارت Postfix به عبارت Infix به کمک پشته

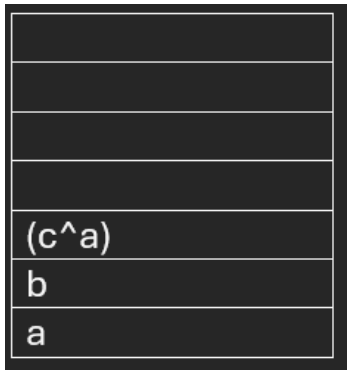
به منظور انجام این نوع تبدیل رشته زیر که یک عبارت postfix را نمایش می دهد مفروض می باشد.

Postfix: abca^*+bc/-

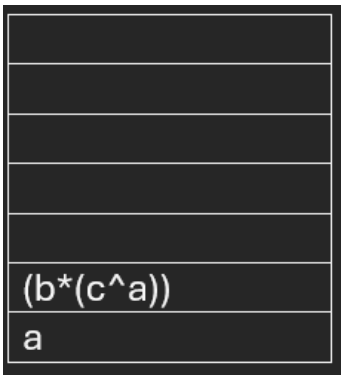
تمام عملوند ها تا رسیدن به اولین عملگر به پشته منتقل می شوند.



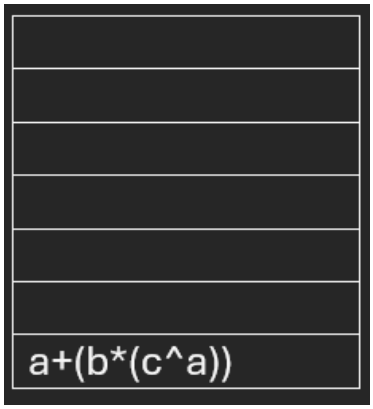
با خواندن عملگر ^ دو عنصر بالای پشته خوانده شده و به همراه عمل مورد نظر مجدد در پشته PUSH می شوند. باید دقت شود که عملوندی که در pop دوم بدست می آید (c) در سمت چپ قرار گرفته شده است.



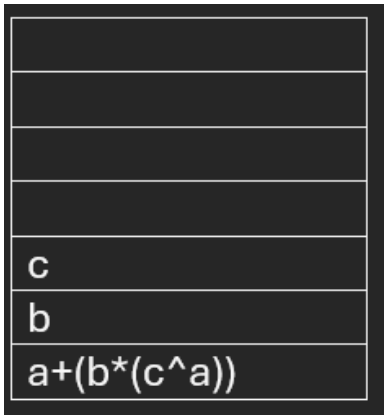
در مرحله بعد عملگر * از رشته ورودی خوانده می شود، در این حالت باز دو عنصر بالای پشته خوانده می شوند و عملگر بر روی آنها تاثیر می گذاردو نتیجه داخل پشته قرار می گیرد.



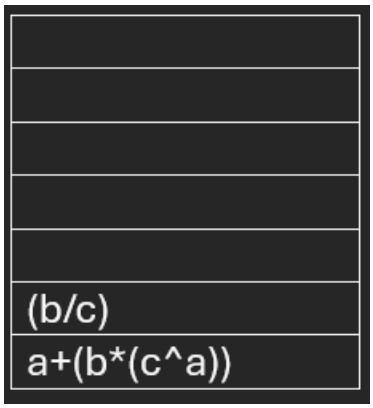
در مرحله بعد با خوانده شدن + این عمل مجدد بر روی دو عنصر بالای پشته اعمال می شود و نتیجه در پشته قرار می گیرد.



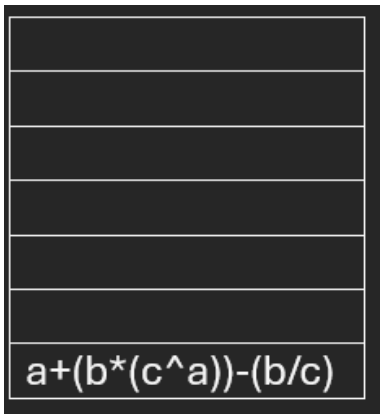
سپس دو عملوند b و c خوانده و به ترتیب خوانده شده داخل پشته قرار می گیرند.



در مرحله بعد با خوانده شدن / این عملگر بر دو عنصر بالای پشته تاثیر گذاشته و نتیجه در پشته قرار می گیرد.



در نهایت با خوانده شدن – عملگر مجدد بر روی دو عنصر بالای پشته اعمال و نتیجه مانند دفعات قبل در پشته قرار می گیرد. از آنجایی که به انتهای رشته ورودی رسیده ایم. عنصر موجود در پشته عبارت infix معدل رشته ورودی می باشد.



Infix: a+(b*(c^a))-(b/c)

تبدیل عبارت Infix به عبارت Prefix

a+b*c^a-b/c

در گام اول اولویت های عملگر ها باید در نظر گرفته شود. روشن است در این عبارت c^a دارای اولویت می باشد پس معادل پسوندی آن را جایگزین می کنیم.

a+b*^ca-b/c

سپس عبارت b/c تبدیل می شود و در عبارت جایگزین می گردد.

a+b*^ca-/bc

حال به تبدیل عملگر b*^ca می پردازیم.

a+*b^ca-/bc

پس از آن نوبت به عملگر + می باشد که باید تبدیل و اضافه شود.

+a*b^ca-/bc

و در انتها عملگر – را داریم تا عمل تبدیل به پایان برسد و خروجی مورد نظر حاصل گردد.

-+a*b^ca/bc

تبدیل عبارت Infix به عبارت Prefix به کمک پشته

برای انجام این تبدیل از انتهای رشته ورودی (عبارات infix) شروع می کنیم و هر عملوندی را که دریافت کردیم در آرایه یا رشته خروجی(عبارت prefix) قرار می دهیم و عملگر ها را در داخل پشته قرار می دهیم.

** نکته: قرار دادن هر عملوند در رشته خروجی از انتها به ابتدا می باشد.

قانون 1: هیچ عملگر با اولویت پایین تر نمی تواند بر روی عملگر با اولویت بیشتر (بر خلاف تبدیل به postfix اینجا اگر عملگرها اولویت یکسان داشته باشند می توانند بر روی هم قرار بگیرند) قرار بگیرد. در این حالت عناصر موجود در پشته آنقدر از پشته خارج می شود تا عملگر خوانده شده از رشته ورودی بتواند در پشته قرار بگیرد.

قانون 2: پرانتز بسته در پشته قرار میگیرد.

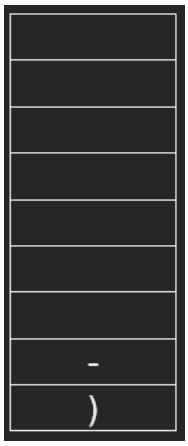
قانون 3: با خواندن پرانتز باز، عناصر داخل پشته تا رسیدن به پرانتز بسته در رشته و یا آرایه خروجی قرار می گیرند و پرانتز باز و بسته دور ریخته می شود.

قانون 4: پس از خوانده شدن آخرین عنصر در آرایه ورودی عناصر داخل پشته در آرایه خروجی نوشته می شوند.

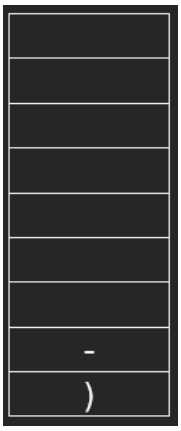
مثال: مطلوب است تبدیل عبارت infix زیر به عبارت Prefix

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

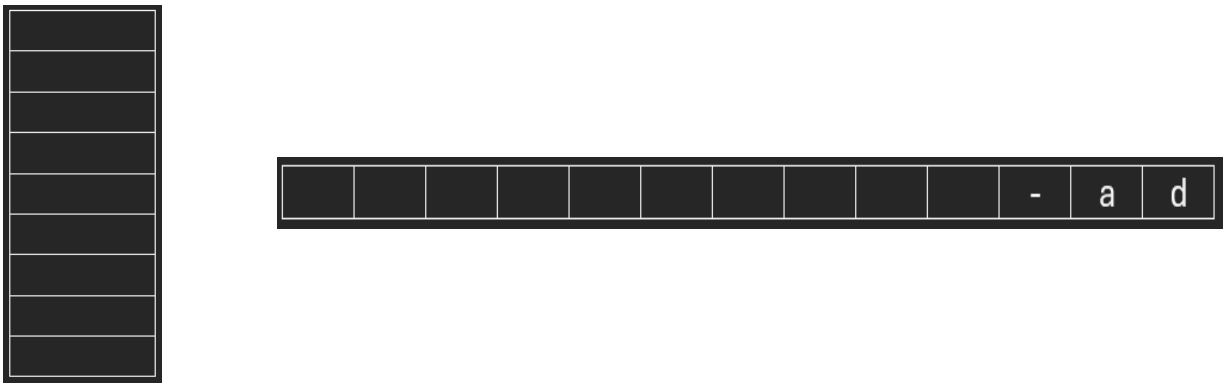
از انتهای رشته شروع می کنیم و پرانتز بسته در پشته قرار می گیرد و سپس d در انتهای رشته خروجی(Prefix) قرار می گیرد. همچنین – هم خوانده شده و در بالای پشته قرار می گیرند.



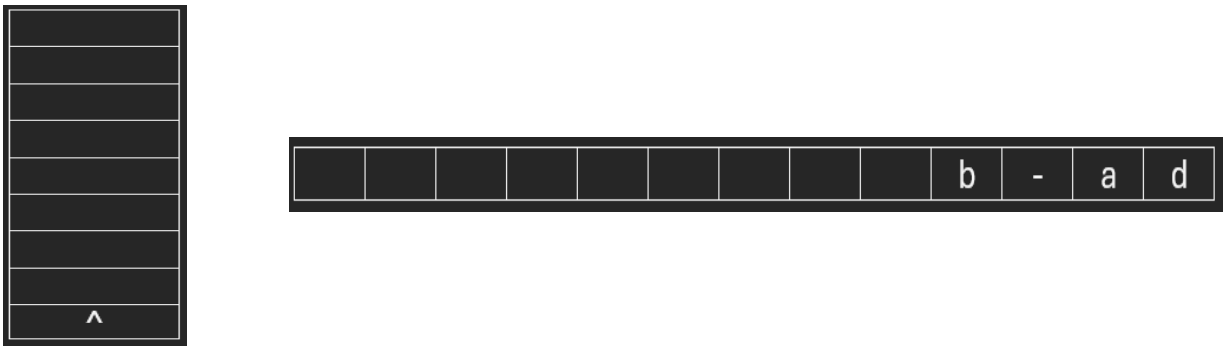
در مرحله بعد عملوند a خوانده شده و در رشته خروجی قرار می گیرد.



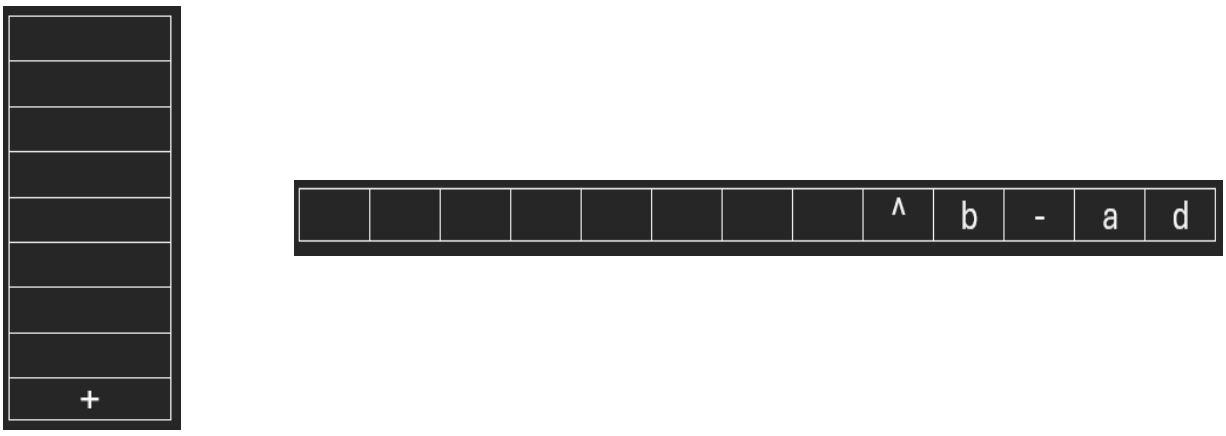
در مرحله بعد با خوانده شدن پرانتز باز عناصر موجود در پشته تا پرانتز بسته خارج شده و به داخل رشته خروجی منتقل می شوند. همچنین پرانتزها دور ریخته می شوند.



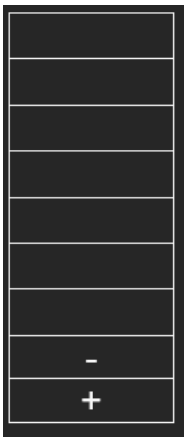
در قدم دیگر عملگر ^ خوانده شده و در پشته قرار می گیرد. سپس b در خروجی نوشته می شود.



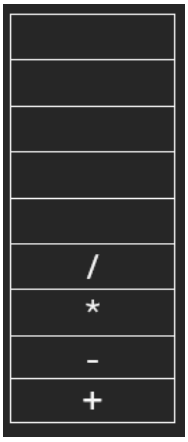
در گام بعدی عملگر + خوانده می شود ولی چون نمی تواند بر روی عملگر با اولویت بالاتر قرار بگیرد، ^ از پشته به داخل رشته خروجی منتقل می شود و سپس + داخل پشته قرار میگیرد.



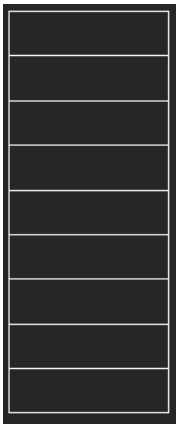
حال عملوند a خوانده می شود و در خروجی قرار می گیرد. سپس – خوانده می شود و به دلیل اینکه + در پشته هم اولویت با – است در اینجا می تواند داخل پشته قرار بگیرد.



حال c خوانده شده به در خروجی توشته می شود و سپس با خواندن * این عملگر در پشته قرار می گیرد. سپس b خوانده شده و در خروجی قرار می گیرد. در این بخش / خوانده شده و در پشته PUSH می گردد و در نهایت a به عنوان آخرین عملوند خوانده شده و در رشته خروجی قرار می گیرد.



در این مرحله که کل رشته ورودی خوانده شده است، همه عناصر موجود در پشته به رشته خروجی منتقل می گردند.



رشته حاصل خروجی مورد انتظار می باشد.

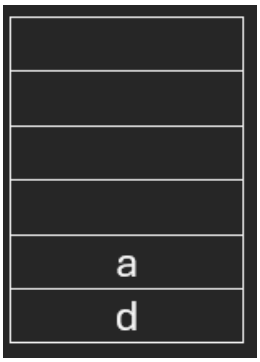
Prefix: +-*/abca^b-ad

تبدیل عبارت Prefix به عبارت Infix به کمک پشته

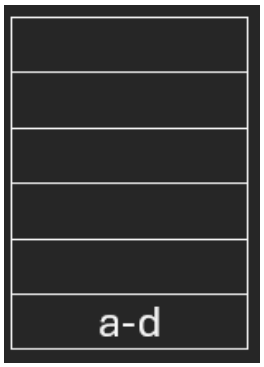
به منظور این نوع تبدیل، رشته عبارت Prefix زیر مفروض می باشد.

Prefix: +-*/abca^b-ad

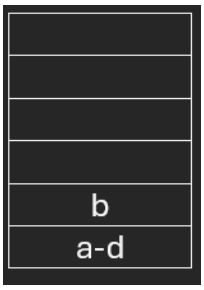
برای تبدیل از انتهای لیست شروع می کنیم و عملگر ها را داخل پشته قرار میدهیم که در اینجا d و a وارد پشته می شوند.



با خواندن عملگر – این عملگر بر روی دو عنصر بالای پشته اعمال می شود. نکته این که در اینجا عنصر بدست آمده از pop اول (a) در سمت چپ عملگر قرار می گیرد و نتیجه باز در پشته قرار میگیرد.

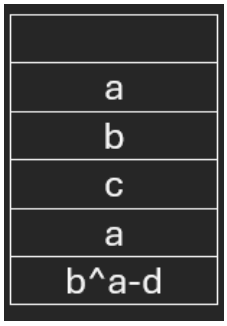


در مرحله بعد b خوانده می شود و در پشته قرار می گیرد.

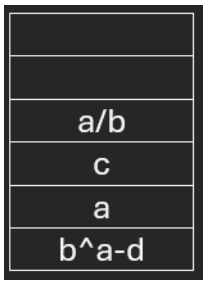


با خوانده شدن عملگر ^ تحت تاثیر این عملگر قرار می گیرند و نتیجه مجدد در پشته PUSH می شود.

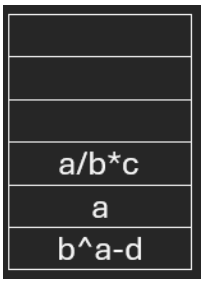
در گام بعدی عملوند های a و c و b و a خوانده شده و در پشته قرار می گیرند تا در عملیات شرکت کنند.



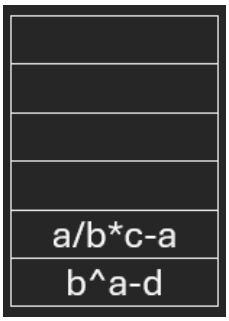
با خوانده شدن عملگر / دو عنصر a و b تحت تاثیر این عملگر قرار می گیرند و نتیجه آن داخل پشته قرار می گیرند.



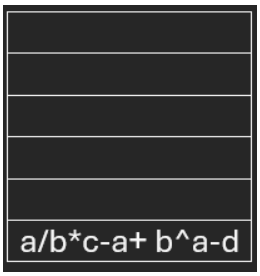
در مرحله بعد * خوانده می شود و بر روی دو عنصر بالای پشته اعمال می گردد.



سپس – خوانده می شود و بر روی دو عنصر بالای پشته اعمال شده و نتیجه در پشته PUSH می گردد.



در نهایت با خوانده شدن + عناصرنهایی موجود در پشته POP شده و عملگر بر روی آنها اعمال می گردد و عبارتی که در پشته قرار می گیرد. نتیجه مورد نظر ما می باشد و عبارت Infix حاصل شده است.



Infix: a/b*c-a+ b^a-d

فصل 5

لیست پیوندی

Link List

لیست پیوندی یک ساختمان داده خطی است که در آن، عناصر در یک مکان پیوسته ذخیره نمی‌شوند، بلکه با استفاده از اشاره گرها به هم مرتبط می‌شوند . لیست پیوندی مجموعه‌ای از گره‌های متصل را تشکیل می‌دهد که در آن هر **گره**، داده‌ها و آدرس گره بعدی را ذخیره می‌کند.

نحوه ذخیره شدن این ساختمان داده بر خلاف آرایه ها که پشت سر هم ذخیره می شدند، در جاهای مختلف از حافظه ذخیره می شوند.

همانطور که مشاهده می شود، Head متغیری است که ابتدای لیست را مشخص می کند و هر گره با ذخیره کردن آدرس گره بعدی در بدنه خود ارتباط با گره بعدی را برقرار می کند و در نهایت و در گره آخر دیده می شود که مقدار null انتهای لیست را مشخص می کند و به گره دیگری اشاره نمی کند.

سختار یک گره به صورت زیر تعریف می شود.

```
unsafe struct Node
{
    public int data;
    public Node* Next;
}
```

در اینجا یک ساختار تعریف شده که در آن یک مقدار عددی برای داده و یک اشاره گر به ساختار Node برای ذخیره گره بعدی تعریف شده است.

ایجاد گره

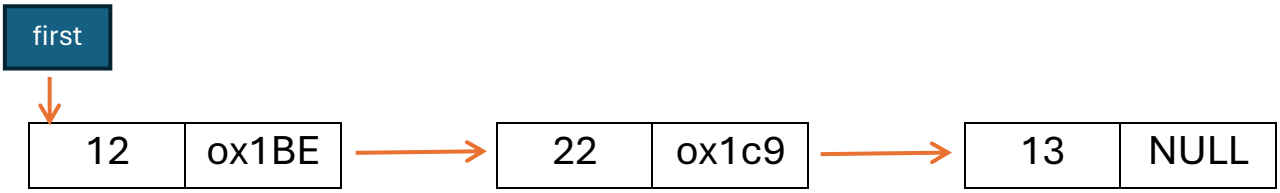
بمنظور ایجاد یک گره از لیست پیوندی لازم است یک نمونه از ساختار Node ساخته شود و خصوصیات آن مقدار دهی گردد.

```
static unsafe Node* createNode(int Data)
{
    Node node=new Node();
    node.data = Data;
    node.Next = null;
    return &node;
}
0 references
static unsafe void Main(string[] args)
{
    Node* newNode = createNode(10);
}
```

همانطور که در کد بالا نشان داده شده است. تابعی به نام CreateNode ایجاد کرده ایم که یک مقدار برای Data دریافت می کند و داخل این تابع پس از ساختن یک نمونه از ساختار Node خصوصیات مقدار دهی می شوند و در نهایت نحوه استفاده از آن در تابع Main آمده است.

نمایش محتوای لیست

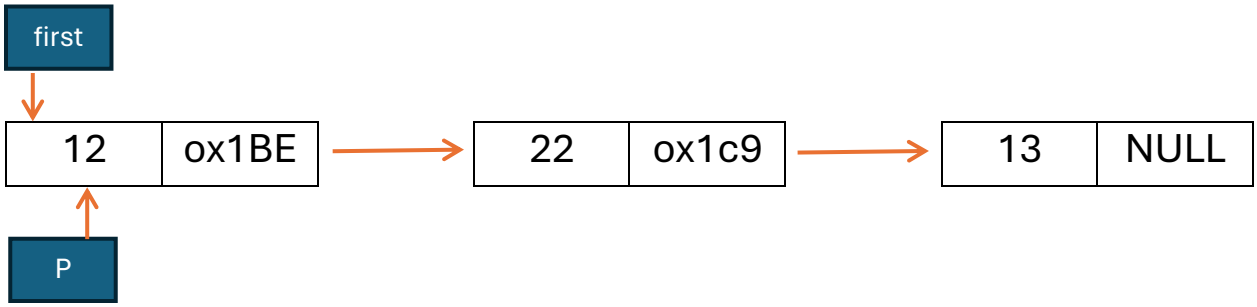
به منظور نمایش اطلاعات گره یک لیست پیوندی، لازم است از ابتدای لیست جایی که اطلاعات گره اول وجود دارد(start)، پیمایش را شروع کنیم.



در مثال بالا مقدار موجود در start آدرس اول لیست می باشد. با داشتن این مقدار گره اول را در یک متغیر مانند P قرار می دهیم تا اطلاعات start که ابتدای لیست را مشخص می کند، دستخوش تغییر نشود و ابتدای لیست را از دست ندهیم چون این باعث از دست رفتن کل لیست می شود. P خود از نوع ساختار گره می باشد.

```
P=first;
```

بعد از اختصاص بالا خواهیم دید که p هم به ابتدای لیست اشاره خواهد کرد.

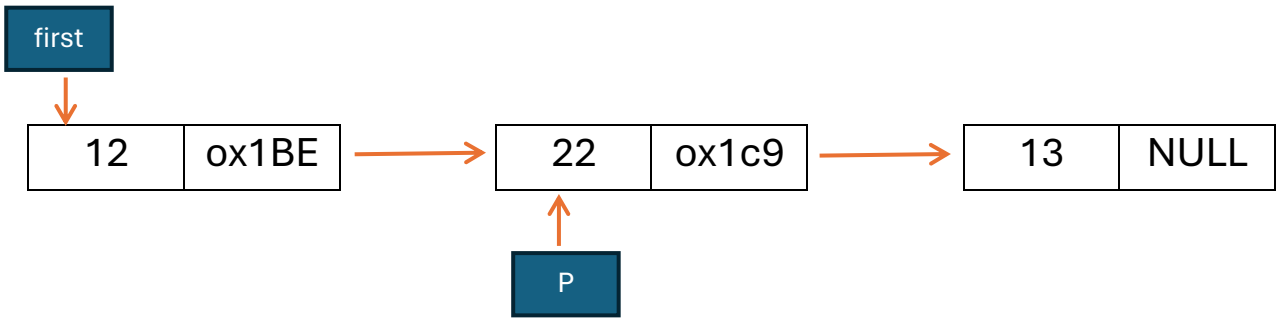


حال می توان داده موجود در گره اول را نمایش داد و به گره بعدی رفت.

```
Console.Write(p.Data)
```

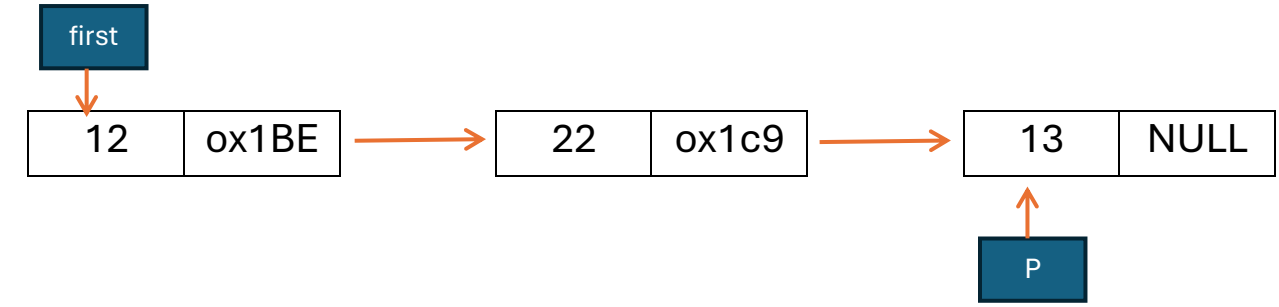
```
p=p->Next;
```

با قرار دادن قسمت next از p که آدرس گره بعدی می باشد در p حالا این متغییر به گره بعدی اشاره می کند.



حال مجدد مرحله قبل تکرار می شود، مقدار داده در خروجی چاپ می شود و p به جلو حرکت می کند.

```
Console.Write(p.Data)

p=p->Next;
```



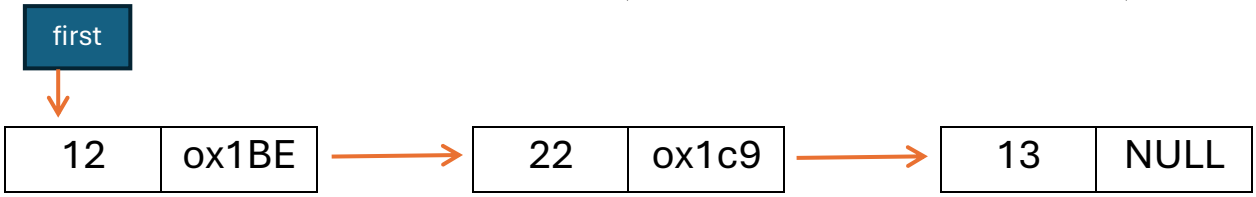
ای مراحل آنقدر ادامه پیدا می کند تا p برابر Null شود که این یعنی به انتها رسیده ایم و تمام گره ها بررسی شده اند و data آنها چاپ شده است.

```
Console.Write(p->Data)

p=p->Next;
```

روشن است اگر first برابر Null باشد لیست خالی می باشد و این اعمال انجام نمی شود.

درج گره در ابتدای لیست

برای انجام این عمل لیست زیر را در نظر میگیریم.

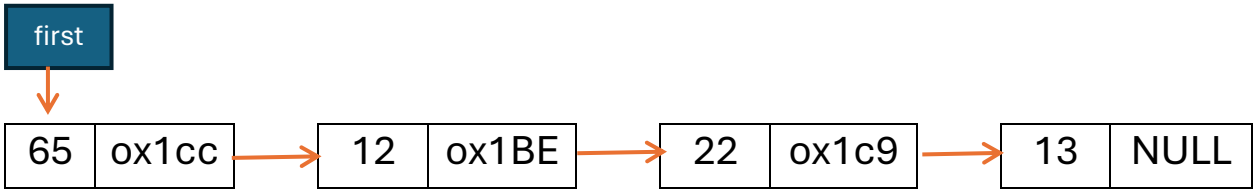


حال یک گره بصورت زیر ایجاد می کنیم.

```
Node* newNode= CreateNode(65)
```

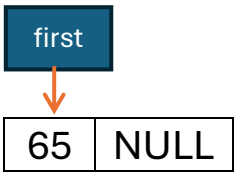
حال برای اضافه کردن این گره به ابتدای لیست باید ابتدا آدرس اولین گره را در قسمت Next از گره جدید قرار دهیم.

```
newNode->next=first;
first=newNode;
```



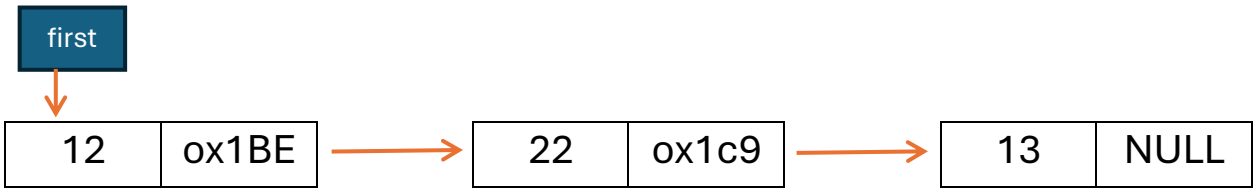
اگر لیست هم خالی باشد first=null :

```
first=newNode;
newNode.next=null;
```



درج گره در انتهای لیست

به منظور تشریح این عمل لیستی به صورت زیر را در نظر می گیریم.



در این قسمت قصد داریم یک گره ایجاد کنیم و به انتهای لیست اضافه کنیم.

```
Node* newNode= CreateNode(100)
newNode.next=Null;
```

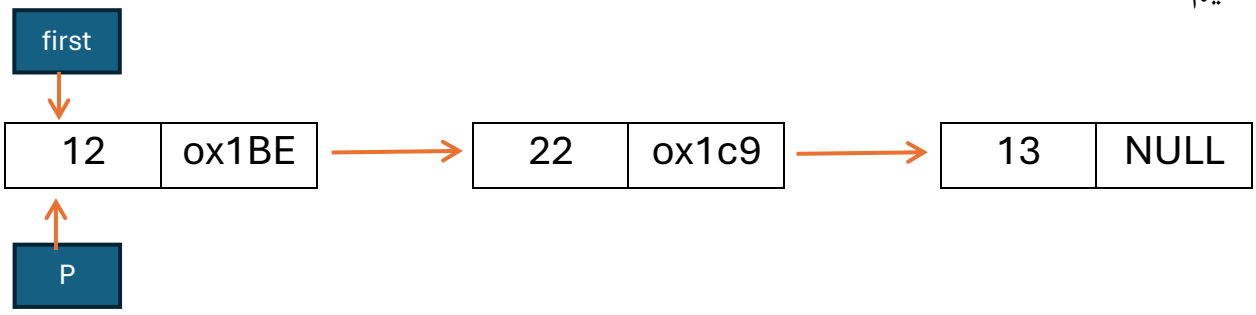
پس از ایجاد گره جدید به گره دیگری نیاز داریم تا بتوانیم به کمک آن در لیست حرکت کنیم(پیمایش) تا بتوانیم به انتهای لیست برسیم.

```
Node* p
```

روش کار به اینصورت است که در ابتدا ابتدای لیست را داخل p قرار می دهیم.

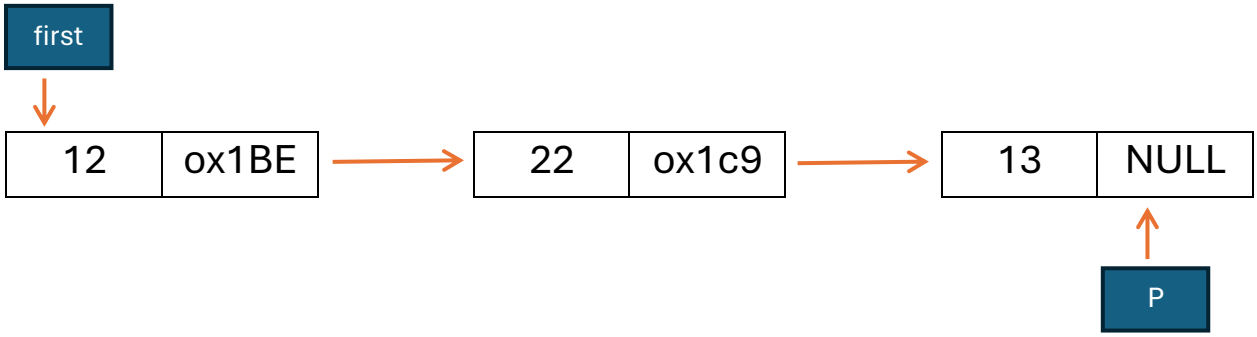
```
p=first;
```

در این حالت p نیز به ابتدای لیست اشاره می کند. حال باید گره به گره به سمت جلو حرکت کنیم.



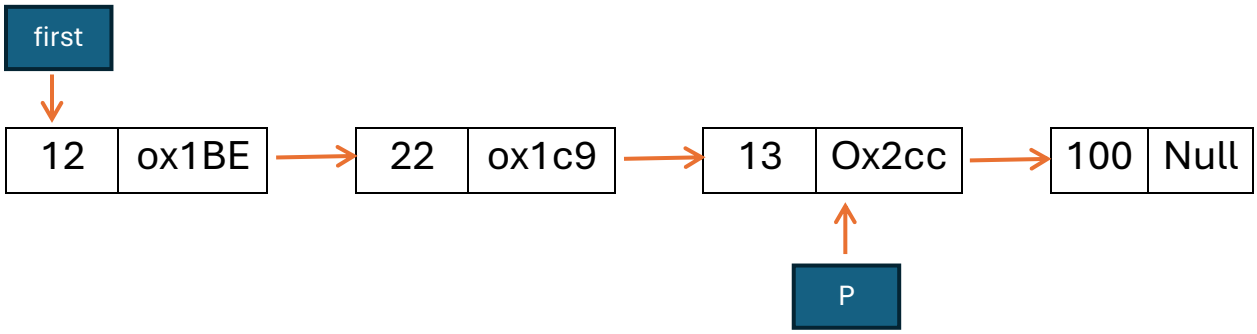
p=p.next;

p=p.next;



با قرار دادن آدرس گره بعد(p.next) داخل p گره کمکی p یکی یکی گره ها را در لیست ملاقات می کند. این کار تا وقتی ادامه پیدا می کند که p.next برابر با null باشد که یعنی به انتهای لیست رسیده ایم. حال باید عمل درج را انجام دهیم.

p.next=newNode;

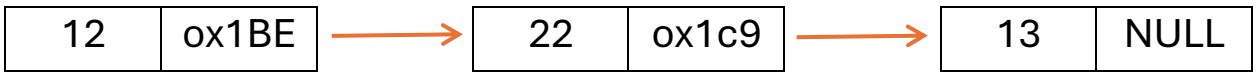


درج گره در وسط لیست

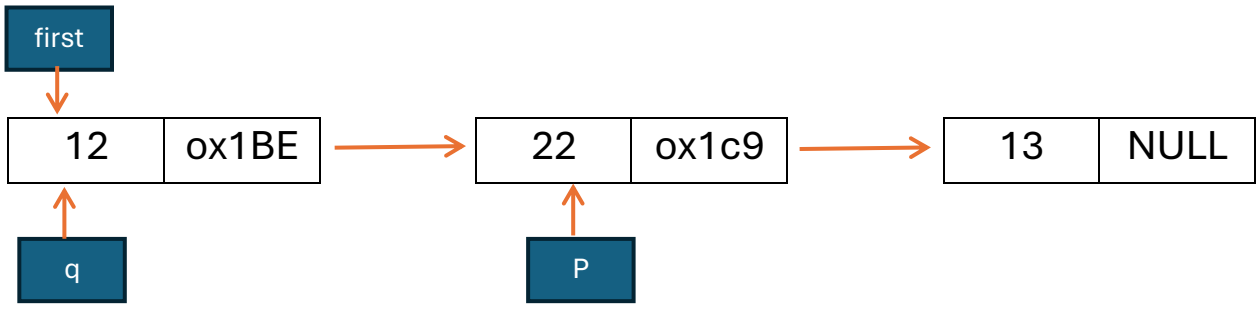
به منظور درج گره لازم است محل درج را داشته باشیم. در این مثال فرض می کنیم می خواهیم گره جدید را در محل 2 درج نماییم.

*node p,q;

*Node NewNode(100);



برای درج گره در محل دوم این لیست باید در لیست حرکت کنیم و گره های کمکی p را در محل مورد نظر قرار دهیم در همین منوال گره q به گره قبلی اشاره می کند.



newNode.Next=p;

q.Next=newNode;

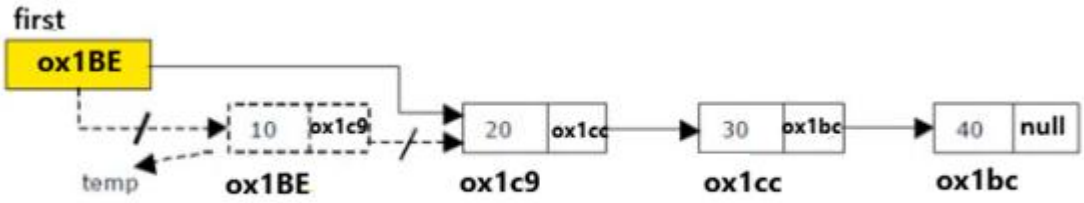
حذف گره از داخل لیست

بمنظور انجام این عمل رویکردهای مختلفی داریم که در زیر به بررسی آنها می پردازیم. هر همه این اعمال باید بررسی شود که لیست خالی نباشد(first!=null).

1- حذف از ابتدای لیست

در این کار کافیست تا first به گره دوم اشاره کند و گره اول آزاد گردد.

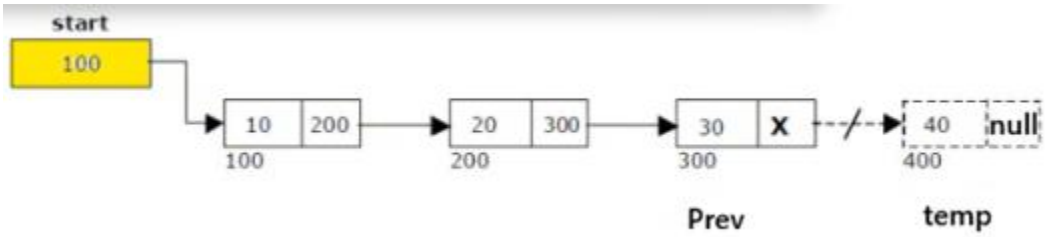
```
Node* temp;
temp=first;
first=temp.next;
free(temp);
```



2- حذف گره از انتهای لیست

برای حذف گره از انتهای لیست به دو گره کمکی temp و prev داریم که گره prev گره قبلی temp می باشد. در این نوع حذف، لیست پیمایش می شود تا وقتی که temp به آخر لیست برسد(temp.next==null). در این صورت مقدار next از گره prev برابر null شده و گره temp آزاد می شود.

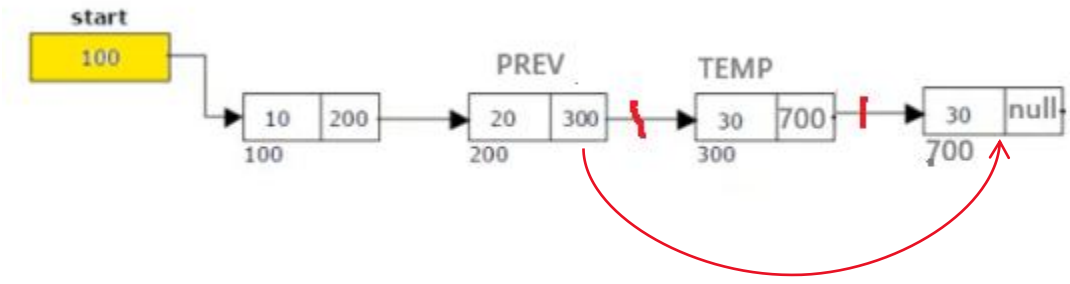
```
Node* temp,prev;
Prev=first;
temp=first;
While (temp.next!=null)
{
    prev=temp;
    temp=temp.next;
}
prev.next=null;
free(temp);
```



3- حذف گره از محل مورد نظر

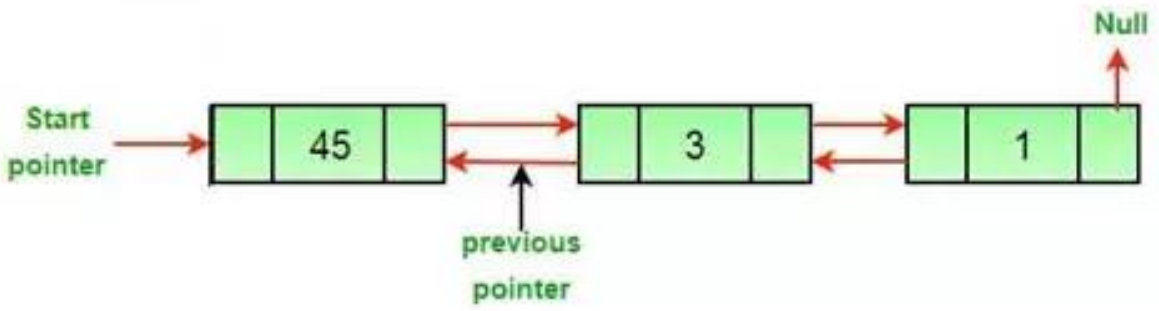
در این عمل باز هم دو گریه temp و prev را به عنوان گره کمکی در نظر می گیریم. در اینجا گره temp در مکان pos از یک لیست با c گره می خواهد حذف گردد. برای انجام گره های temp و prev در لیست حرکت می کنند بطوری که prev گره قبلی temp می باشد. این حرکت ادامه می یابد تا temp به محل مورد نظر برسد. حال گره prev با اشاره به گره بعدی temp (prev=temp.next) و آزاد سازی temp عمل حذف انجام می شود.

```
Node* prev,temp;
Int pos,c;
temp=first;
if(pos==1)
{
      Start=temp.next;
}
Else
{
    If(pos>1 && post<=c)
          {
               temp=first;
               For(int i=1;i<pos;i++)
                   {
                        Prev=temp;
                        temp=temp.next;
                   }
                   prev.next=temp.next;
          }
           Free(temp);
}
```



لیست پیوندی دو طرفه

در این نوع لیستهای پیوندی، تفاوت با لیست پیوندی یک طرفه این است یک عنصر دیگر در ساختار گره وجود دارد که آدرس گره قبلی را در خود نگهداری می کند.

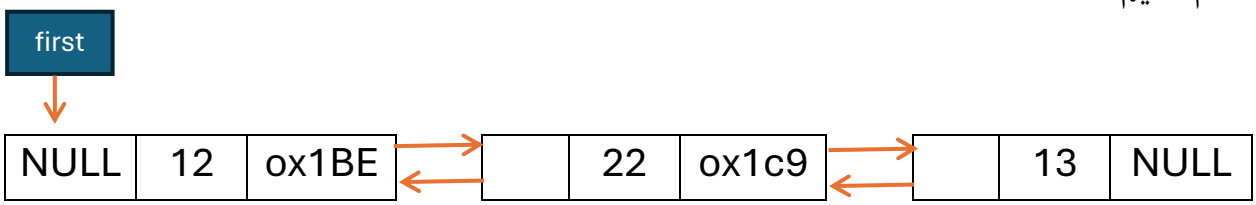


برای ایجاد گره کد زیر مفروض می باشد.

```
public unsafe struct Node
{
    public int data;
    public Node* Next;
    public Node* Prev;
}
3 references
static unsafe Node* createNode(int Data)
{
    Node n = new Node();
    n.data = Data;
    n.Next = null;
    n.Prev = null;
    Node* final = &n;

    return final;
}
```

اضافه کردن یک گره به ابتدای لیست

به منظور اضافه کردن یک گره به این لیست پس از ایجاد گره، باید نسبت به تنظیم اشاره گر ها اقدام کنیم.




```
*Node newNode(18);
newNode.Next=first;
```

با دستورات بالا یک گره جدید ایجاد می شود و این گره به گره اول اشاره می کند.

```
first.Prev=newNode;
newNode.Prev=null;
first=newNode;
```

در مراحل بعد مشاهده می شود که Prev گره اول نیز به گره جدید اشاره کرده و مقدار Prev از گره جدید چون قرار است گره اول باشد null می شود و در انتها First به گره جدید اشاره کرده و گره به ابتدای لیست اضافه میگردد.

اضافه کردن یک گره بین دو گره مورد نظر

اگر ما بخواهیم گره ای را در وسط لیست اضافه کنیم، لازم است گره قبل و بعد را داشته باشیم و گره جدید را بین آنها درج کنیم

```
*Node p,q;
*node newNode=createNode(100)
```

در اینجا گره p گره بعدی و گره q گره قبلی می باشد و همچنین گره جدید را نیز ایجاد کردیم. برای درج این گره ما لازم داریم که 4 اشاره گر را جابجا کنیم.

```
q.next=newNode;
newNode.prev=q;
newNode.Next=p;
p.prev=newNode;
```

اضافه کردن گره به انتهای لیست

برای این نوع درج ما نیاز داریم که گره آخر را داشته باشیم.

```
*Node p;
```

در اینجا گره p گره آخر لیست می باشد.

```
*node newNode=createNode(100)
p.Next=newNode;
newNode.Next=null;
newNode.Prev=p;
```

حذف کردن از ابتدای لیست

به منظور این عمل، باید گره های اول و دوم را داشته باشیم.

```
*Node p,q;
```

در اینجا گره p دومین گره و گره q اولین گره می باشد.

```
First=p;
p.Prev=null
q.Next=null;
free(q);
```

حذف کردن از انتهای لیست

به منظور این عمل، باید گره های آخر و ماقبل آخر را داشته باشیم.

```
*Node p,q;
```

در اینجا p گره آخر و q گره ماقبل آن می باشد.

```
q.next=null;
p.prev=null;
free(p);
```

حذف کردن از وسط لیست

به منظور این عمل، باید گره ای که می خواهیم حذف کنیم را پیدا کنیم و گره بعدی و قبلی آن را نیز داشته باشیم.

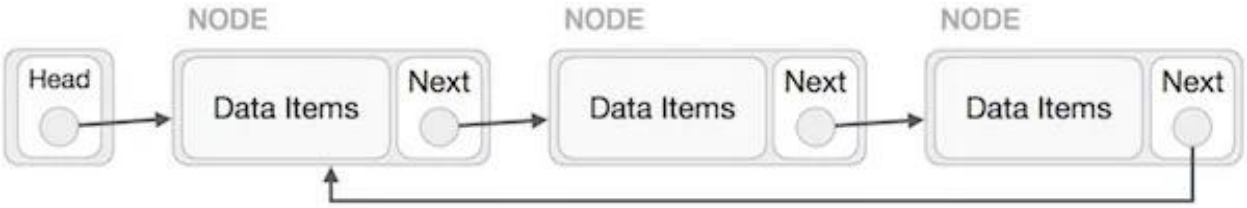
```
*Node p,q,c;
```

در این مثال قرار است گره c حذف شود و گره p گره بعدی و گره q نیز گره قبلی آن می باشد.

```
p.Prev=q;
q.next=p;
c.prev=null;
c.next=null;
free(c);
```

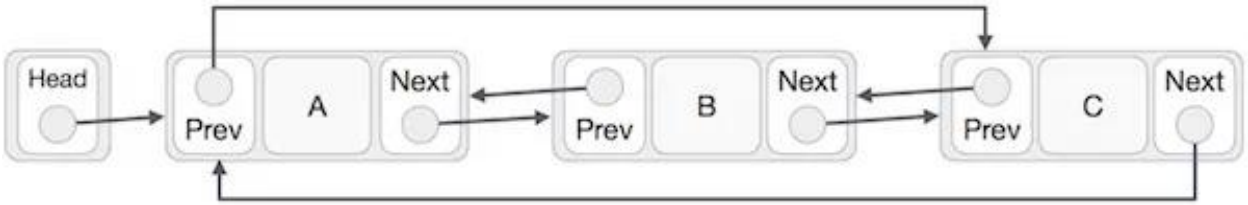
لیست پیوندی چرخشی(حلقوی)

در لیست پیوندی یک طرفه، اشاره‌گر next در عنصر آخر به گره اول لیست اشاره می‌کند:



تبدیل لیست پیوندی دو طرفه به لیست حلقوی

در لیست پیوندی دو طرفه اشاره‌گر next در عنصر آخر به گره اول لیست اشاره می‌کند و اشاره‌گر previousگره نخست نیز به عنصر آخر اشاره می‌کند تا از هر دو طرف لیست به صورت یک حلقه در آید.



همان طور که در تصویر فوق مشخص است این نوع لیست نقاط مهمی دارد که در ادامه آن‌ها را توضیح داده‌ایم:

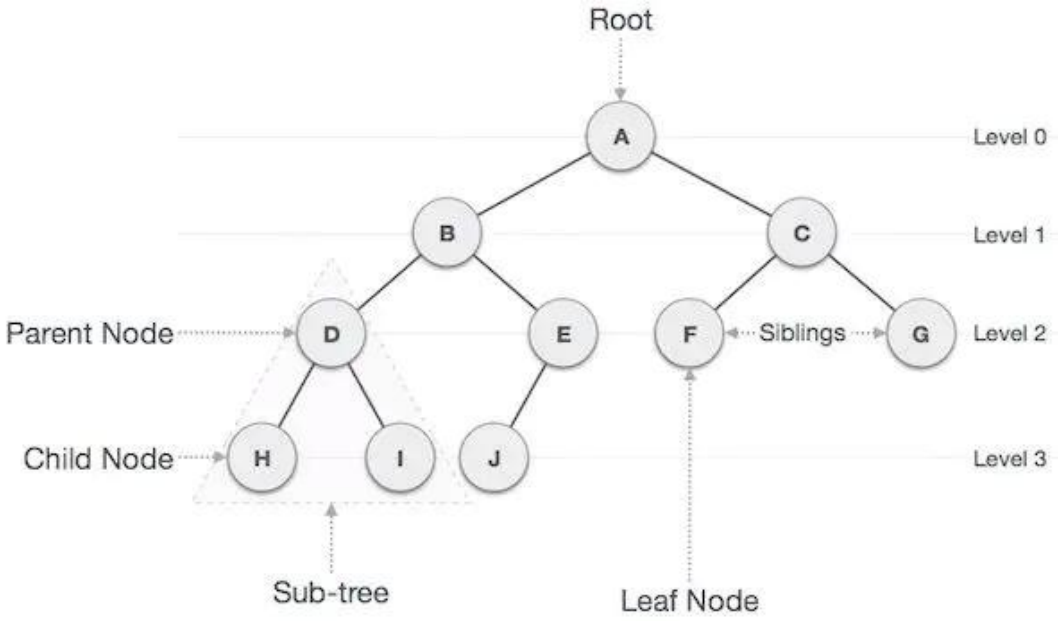
- در هر دو نوع لیست‌های پیوندی یک و دو طرفه، اشاره‌گر next لینک آخر به لینک نخست لیست اشاره می‌کند.
- در لیست پیوندی دو طرفه اشاره‌گر previous عنصر نخست به گره آخر لیست اشاره می‌کند.

فصل 6

درخت

Tree

ساختمان داده درخت به عنوان مجموعه‌ای از اشیاء یا موجودیت‌هایی به نام گره (Node) تعریف می‌شود که به صورت سلسله مراتبی (به وسیله یال) به یکدیگر مرتبط شده‌اند. درخت ساختمان داده‌ای غیرخطی است، زیرا داده‌ها را نه به صورت متوالی، بلکه به صورت **سلسله‌مراتبی** ذخیره می‌کند. در ساختمان داده درخت، اولین گره به عنوان گره ریشه شناخته می‌شود. درخت از گره ریشه شروع می‌شود و توسعه می‌یابد. هر گره حاوی داده و همچنین حاوی ارجاعاتی به گره‌های فرزند است. یک گره ریشه هرگز نمی‌تواند گره والد داشته باشد.



چند تعریف در ساختار درخت

- گره والد[1]: گره‌ای که در بالای یک گره و متصل به آن است، گره والد آن گره نامیده می‌شود .
- گره فرزند[2]: گره‌ای که در زیر و متصل به یک گره است، گره فرزند آن گره نامیده می‌شود.
- گره ریشه[3]: بالاترین گره یک **درخت** یا گره‌ای که هیچ گره والدی ندارد، گره ریشه نامیده می‌شود .گره ریشه درخت است. یک درخت غیرتهی باید دقیقاً یک گره ریشه و دقیقاً یک مسیر از ریشه تا سایر گره‌های درخت داشته باشد.

[1] Parent
[2] Child
[3] Root

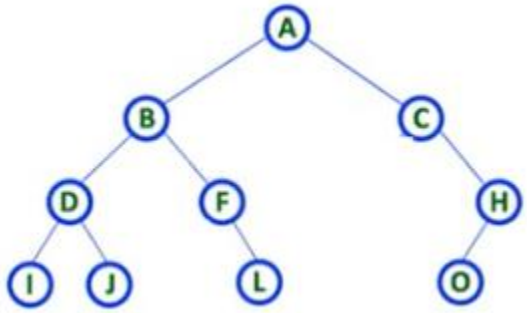
- گره برگ[1]: گره‌هایی که هیچ گره فرزندی ندارند، گره برگ نامیده می‌شوند .
- اجداد یک گره: هرگره قبلی درمسیر ریشه به یک گره، اجداد آن گره نامیده می‌شود.
- خواهر و برادر[2]: به فرزندان یک گره والد، خواهر و برادر می‌گویند.
- سطح یک گره[3]: تعداد یال‌ها درمسیر گره ریشه تا آن گره را گویند. گره ریشه دارای سطح 0 است.
- گره داخلی[4]: گره‌ای که حداقل یک فرزند داشته باشد گره داخلی نامیده می‌شود.
- زیر درخت[5]: هرگره درخت همراه با نوادگان آن.
- تعداد یال ها: یک یال را می‌توان به‌عنوان اتصال بین دو گره تعریف کرد. اگر درختی N گره داشته باشد، دارای (N - 1) یال خواهد بود. تنها یک مسیراز هرگره به هر گره دیگر درخت وجود دارد.
- عمق یک گره[6]: عمق یک گره به‌عنوان طول مسیر از ریشه تا آن گره تعریف می‌شود. هر لبه 1 واحد طول به مسیر اضافه می‌کند؛ بنابراین می‌توان آن را به‌عنوان تعداد یال‌های مسیر از ریشه درخت تا گره نیز تعریف کرد.
- ارتفاع یک گره[7]: ارتفاع یک گره را می‌توان به‌عنوان طول طولانی‌ترین مسیر از گره تا یک گره برگ درخت تعریف کرد.
- ارتفاع درخت: ارتفاع درخت طول طولانی‌ترین مسیر از ریشه درخت تا یک گره برگ درخت است.

[1] Leaf
[2] siblings
[3] Level of Node
[4] Internal Node Or Relay Node
[5] Sub Tree
[6] Depth of Node
[7] Height of Node

- درجه یک گره: تعداد کل زیر درخت‌های متصل به آن گره را درجه گره می‌گویند. درجه یک گره برگ باید 0 باشد. درجه یک درخت حداکثر درجه یک گره در بین تمام گره‌های درخت است.

درخت دودوئی[1]

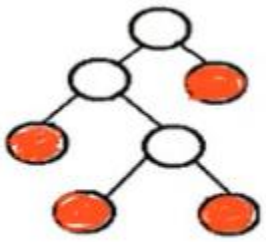
درخت دودوئی درختی است که هر گره آن حداکثر 2 فرزند دارد.



یک درخت دودویی با ارتفاع h دارای 2^h برگ و 2^{h+1} گره دارد. همچنین تعداد گره های داخلی از فرمول $floor(\frac{n}{2})$ بدست می آید. همچنین داریم n_0 تعداد برگها از رابطه زیر نیز بدست می آید که در تمام درختهای دودویی صدق می کند. در این ف

$$n_0 = n_2 + 1$$

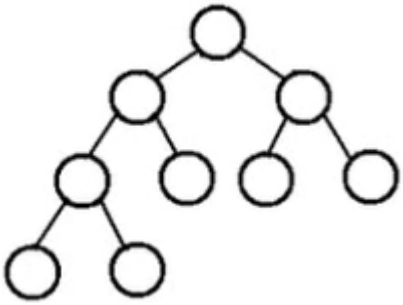
درخت دودویی پُر شده (Full Binary Tree) درختی است که در آن هر گره غیر برگ دارای 2 فرزند می باشند.



full tree

[1] Binary Tree

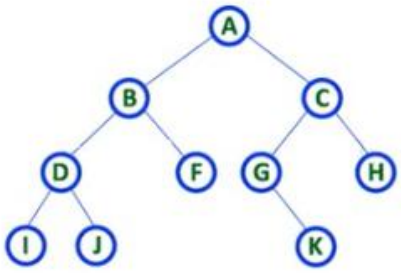
درخت دودویی کامل (Complete Binary Tree) درختی است که به جز سطح آخر تمام سطوح بطور کامل پر شد است و همه گره ها از سمت چپ کامل می شوند.

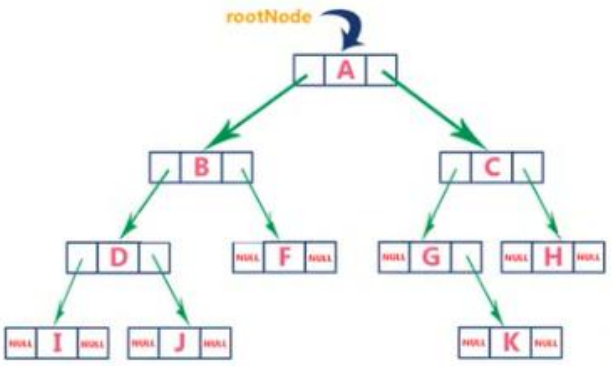


پیاده سازی درخت

برای پیاده سازی یک درخت از یک ساختار استفاده می شود.

```
public unsafe struct Node
{
    public int data;
    public Node* left;
    public Node* rigth;
}
0 references
static unsafe Node* CreateNode(int x)
{
    Node n = new Node();
    n.data = x;
    n.left = null;
    n.rigth = null;
    return &n;
}
```



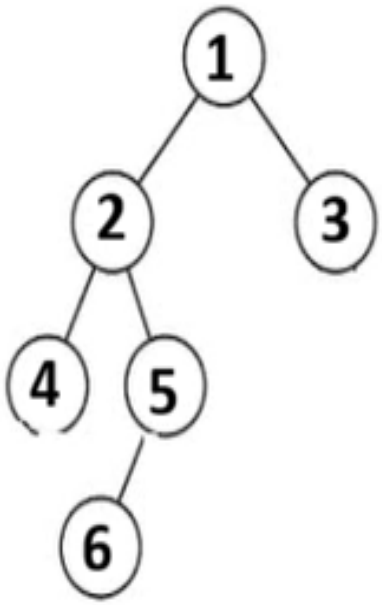


همچنین تابع ایجاد گره به شکل زیر می باشد.

```
static unsafe Node* CreateNode(int x)
{
    Node n = new Node();
    n.data = x;
    n.left = null;
    n.rigth = null;
    return &n;
}
```

با فراخوانی تابع Create و قرار دادن گره جدید در محل مورد نظر می توانیم درخت حاصل را بوجود آورد.

```
static unsafe void Main(string[] args)
{
    //Level 0
    Node* r = CreateNode(1);

    //Level 1
    r->left = CreateNode(2);
    r->rigth = CreateNode(3);

    //Level 2
    r->left->left = CreateNode(4);
    r->left->rigth = CreateNode(5);

    //Level 3
    r->left->rigth->left = CreateNode(6);
}
```



در مثال بالا کد نوشته شده و درخت حاصل را مشاهده می کنید که بمنظور تکمیل موارد در زیر به شرح آنها می پردازیم.

در مثال بالا گره ریشه به نام r تعریف شده است که سطح یک درخت می باشد. در سطح 2 مشاهده می شود که گره با مقدار 2 ایجاد و در سمت چپ درخت قرار می گیرد و گره با مقدار 3 در سمت چپ گره ریشه جایگذاری می شود.

در سطح دوم، گره با مقدار 4 به عنوان فرزند سمت چپ و گره با مقدار 5 به عنوان فرزند سمت راست گره موجود در سمت چپ گره ریشه قرار می گیرد. و در نهایت گره سطح سوم با مقدار 6 هم در جایگاه فرزند سمت چپ گره 5 جایگذاری می شود.

پیمایش درخت

اعمال زیادی روی درخت‌ها انجام می‌شود اما رایج‌ترین عملی که صورت می‌گیرد پیمایش درخت است. پیمایش درخت در ساختمان داده یعنی دستیابی صحیح به هر گره و انجام عملیات روی گره‌هایی که به آن‌ها دسترسی پیداکرده‌ایم. وقتی یک درخت را پیمایش می‌کنیم در واقع یک ترتیب خطی از گره‌های درخت را ایجاد می‌کنیم که ترتیب گره‌هایی که به آن‌ها دسترسی پیدا کرده‌ایم را نشان می‌دهد و بسیار موثر است. در هنگام پیمایش رفتار یکسانی با گره‌ها و زیر درختان انجام می‌گیرد.

الگوریتم‌های پیمایش درختی را بر اساس نظم گره‌هایی که بازدید می‌شوند، به طور گسترده می‌توان در دو دسته زیر طبقه‌بندی کرد.

- **الگوریتم‌های جست‌وجوی عمق‌اول :(Depth-First Search |DFS)** در این نوع از الگوریتم‌ها عمل پیمایش را از گره اول شروع می‌کنیم. در ابتدا قبل از عقب نشینی از هر شاخه، همه گره‌های آن شاخه را به ترتیب تا عمیق‌ترین سطح ممکن بازدید می‌کنیم. سپس گره‌های همه شاخه‌های دیگر نیز طبق همین روش باید بازدید شوند. سه نوع الگوریتم، از این روش جست‌وجو استفاده می‌کنند که در ادامه این مطلب آن‌ها را پوشش داده‌ایم .
- **الگوریتم جست‌وجوی سطح‌اول :(Breadth-First Search |BFS)** این الگوریتم نیز از ریشه شروع به کار می‌کند. روش کار به این صورت است که قبل از رفتن به هر سطحی در درخت، در ابتدا همه گره‌های سطح بالاتر بررسی می‌شود. در بخش بعدی الگوریتمی از نوع BFS را بررسی کرده‌ایم.

الگوریتم‌های جست‌وجوی عمق‌اول DFS

اگر L را حرکت به چپ، R را حرکت به راست، V را رسیدن به گره در نظر بگیریم، برای پیمایش یک درخت شش ترکیب ممکن زیر وجود خواهند داشت RLV, RVL, VRL, VLR, LRV, LVR . اما اگر تنها حالتی را در نظر بگیریم که ابتدا به سمت چپ و بعد به راست حرکت کنیم در این صورت فقط سه ترکیب وجود خواهد داشت. VLR, LRV, LVR :

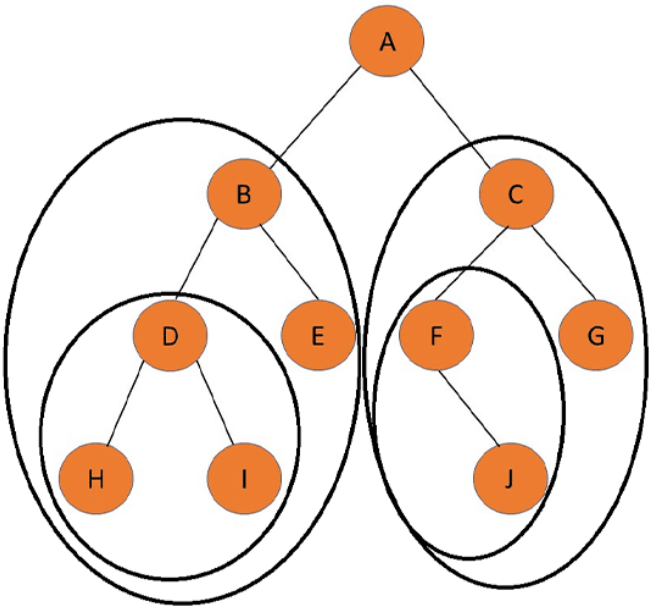
این سه حالت را به ترتیب Preorder, In order, Post order می‌نامند، این نام‌گذاری به جهت نحوه‌ی قرار گرفتن موقعیت V نسبت به L و R است. مثلا در Post order ابتدا زیردرخت‌ها پیمایش می‌شوند و بعد نوبت به گره می‌رسد درحالی‌که در Preorder, اول گره را ملاقات می‌کنیم بعد به پیمایش زیر درخت‌ها می‌پردازیم. پس سه نوع پیمایش درخت داریم:

1. پیمایش پس ترتیب Post order[1]
2. پیمایش میان ترتیب In order[2]
3. پیمایش پیش ترتیب Preorder[3]

[1] Post-Order Traversal
[2] In-order Traversal
[3] Pre-order Traversal

پیمایش میان ترتیب In order

می‌خواهیم پیمایش In order درخت زیر را به دست آوریم:

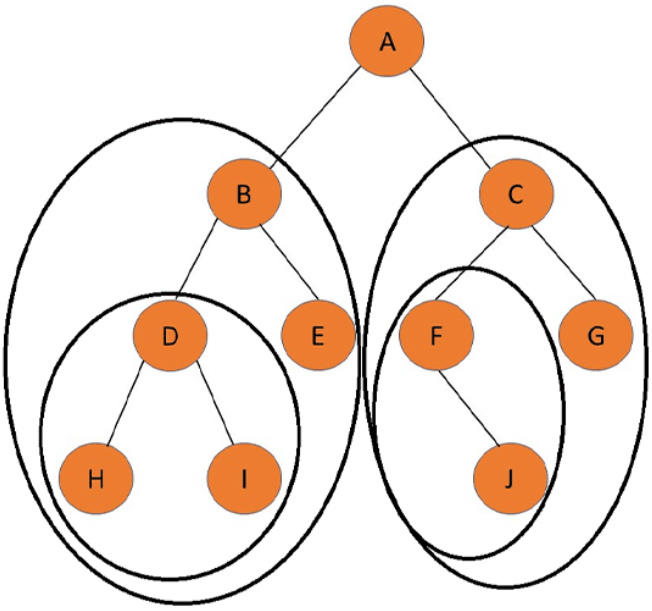


ابتدا زیر درخت سمت چپ A را پیمایش می‌کنیم که یک درخت به شمار می‌رود. در این زیر درخت هم اول به سراغ زیر درخت سمت چپ که ریشه آن D است می‌رویم. در این زیر درخت به سراغ گره سمت چپ یعنی H رفته بعد به سراغ ریشه آن یعنی D رفته و بعد گره سمت راست یعنی I رفته و بعد از پیمایش آن‌ها به سمت بالا برمی‌گردیم. گره B و بعد فرزند راست یعنی E را پیمایش می‌کنیم و مجددا به بالا برمی‌گردیم و گره A را پیمایش می‌کنیم. حالا به سمت راست درخت می‌رویم، ابتدا به سراغ زیر درخت سمت چپ باریشه F رفته و چون گره سمت چپ ندارد، اول ریشه Fپیمایش شده و بعد گره سمت راست یعنی J را پیمایش می‌کنیم. سپس به ریشه C برگشته و بعد از پیمایش آن به گره سمت راست آن یعنی G رفته و پیمایش را به‌پایان می‌رسانیم. در پیمایش LNR, آخرین گره سمت چپ در اول و آخرین گره سمت راست در آخر قرار می‌گیرند. ترتیب پیمایش گره‌ها به صورت زیر است:

H D I B E A F J C G

پیمایش پیش ترتیب Pre Order

می‌خواهیم پیمایش Pre order درخت زیر را به دست آوریم:



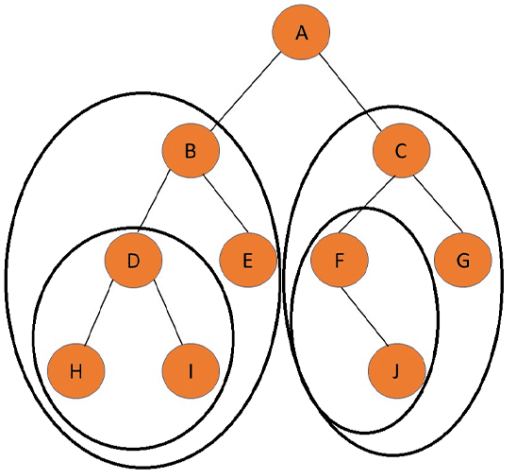
اول ریشه A را پیمایش می‌کنیم بعد به سراغ زیر درخت B رفته و بعد از پیمایش خود ریشه B به زیر درخت چپ یعنی D می‌رویم . بعد از پیمایش خود D زیر درخت سمت چپ آن یعنی H و بعد زیر درخت سمت راست یعنی I را پیمایش کرده و به ریشه D برگشته و از آن به ریشه B رفته و گره E را پیمایش می‌کنیم.

حالا نوبت سمت راست ریشه A است که پیمایش شود. برای این کار ابتدا ریشه زیر درخت سمت راست یعنی C پیمایش شده بعد به زیر درخت سمت چپ رفته و گره F را پیمایش می‌نماییم. چون گره ریشه F زیر درخت چپ ندارد به سراغ زیر درخت راست یعنی J رفته و آن را پیمایش می‌کنیم، سپس به ریشه F بازگشته و از آن به ریشه C رسیده و زیر درخت سمت راست آن یعنی G را پیمایش می‌کنیم و کار پایان می‌یابد. در پیمایش NLR ریشه درخت در ابتدا و آخرین گره در سمت راست در آخر ظاهر می‌شود، ترتیب پیمایش گره‌ها به‌صورت زیر است:

A B D H I E C F J G

پیمایش پس ترتیبPost order

می‌خواهیم پیمایش Post order درخت زیر را به دست آوریم:



ابتدا از پایین‌ترین زیر درخت سمت چپ شروع می‌کنیم یعنی گرهH ، بعد سراغ گره I می‌رویم و بعد از پیمایش آن ریشه D را پیمایش می‌کنیم. سپس نوبت گره Eو بعد از آن هم نوبت ریشه B است. حالا به زیر درخت سمت راست ریشه A می‌رویم و از پایین‌ترین زیر درخت سمت چپ شروع به پیمایش می‌کنیم. چون در سمت چپ زیر درختی وجود ندارد از سمت راست J راپیمایش کرده و بعد سراغ F رفته و آن را پیمایش می‌کنیم.

در مرحله بعد G و پس از آن هم نوبت به C می‌رسد. در آخر هم گرهA ، یعنی ریشه اصلی را پیمایش کرده و کار خاتمه می‌یابد. ترتیب پیمایش گره‌ها به صورت زیر است:

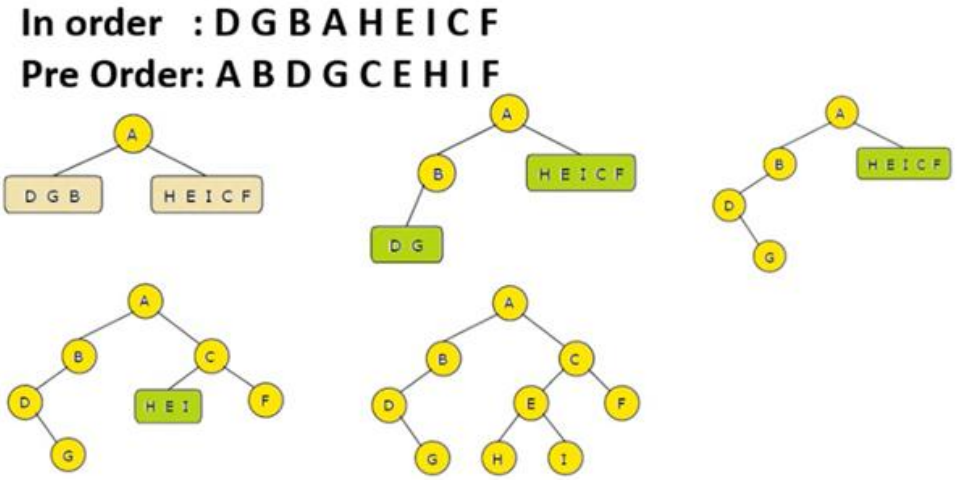
H I D E B J F G C A

الگوریتم

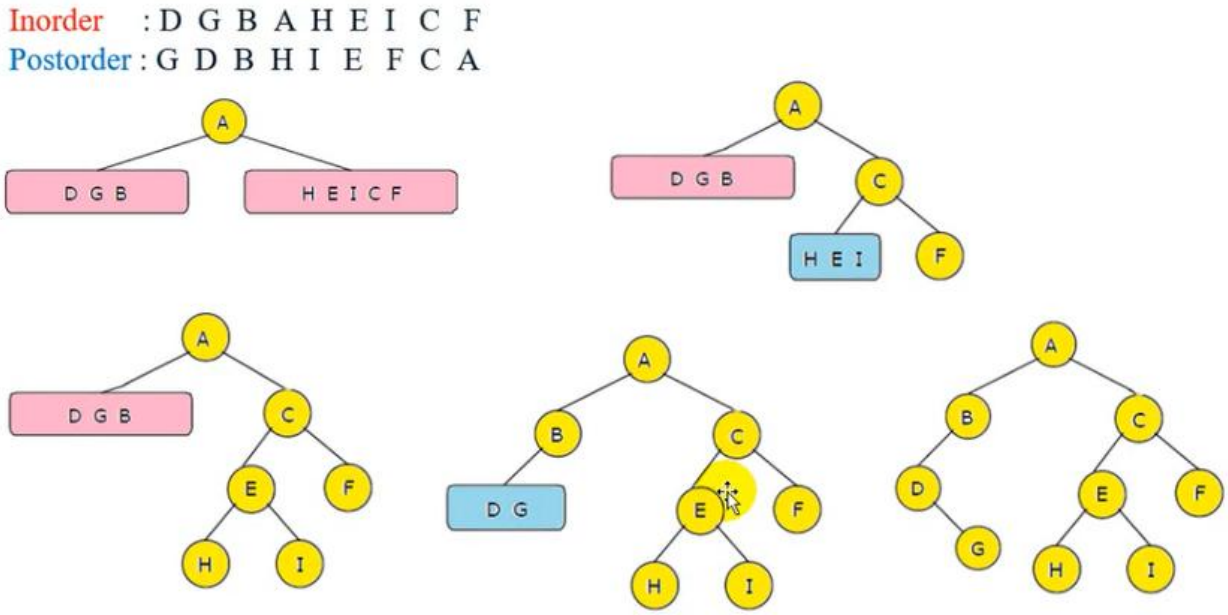
پیشوندی	میانوندی	پسوندی

pre (p){
if (p==null) return;
cout<< p -> data;
pre (p -> left);
pre (p -> right);
}
	in (p){
	if (p==null) return;
	in (p -> left);
	cout<<p -> data;
	in (p -> right);
	}
		post(p){
		if (p==null) return;
		post (p -> left);
		post (p -> right);
		cout<<p -> data;
		}

رسم درخت دودوئی با داشتن In order و Preorder



رسم درخت دودوئی با داشتن In order و Post Order

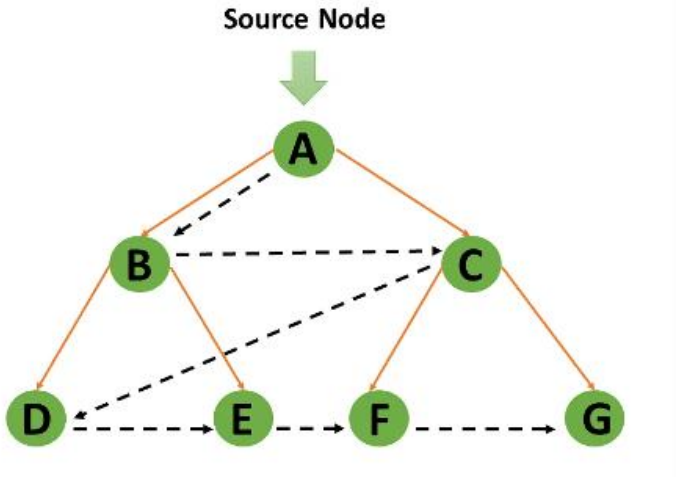


الگوریتم جست‌وجوی سطح اول BFS

الگوریتم BFS ، جزو پرکاربردترین الگوریتم‌ها به‌حساب می یادBFS ، یک رویکرد پیمایش گراف و درخت می باشد که می‌توانیم ابتدا الگوریتم رو با گره مبدأ (منبع) آغاز کنیم و بعدش لایه به لایه، گره‌هایی رو که مستقیماً با گره مبدأ ارتباط دارن، تحلیل کنیم. در مرحله بعد پیمایشBFS ، باید گره‌های مجاور سطح بعد رو مشاهده کند.

بر اساسBFS ، شما باید گراف و با درخت را در جهت عرضی پیمایش کنین:

- برای شروع، به‌صورت افقی حرکت می کنیم و از کل گره‌های لایه موجود رو بازدید می کنیم
- سپس به لایه بعدی حرکت می کنیم.



جستجوی اول سطح از ساختار داده صف برای نگهداری گره و بازدید شدن آن تا زمان بازدید از کل رئوس مجاور که ارتباط مستقیمی با اون گره دارن، استفاده می‌کنه. صف هم بر اساس اصل FIFO شکل می‌گیره و اولین گره نمایش داده شده، اون گرهی هست که اول از همه، قرار داده شده. پیمایش الگوریتم BFS در مورد گره‌ها به دو شکل انجام می‌شه:

- گره بازدید شده
- گره بازدید نشده.

ترتیب مراحل اجرای الگوریتم جستجوی اول سطح

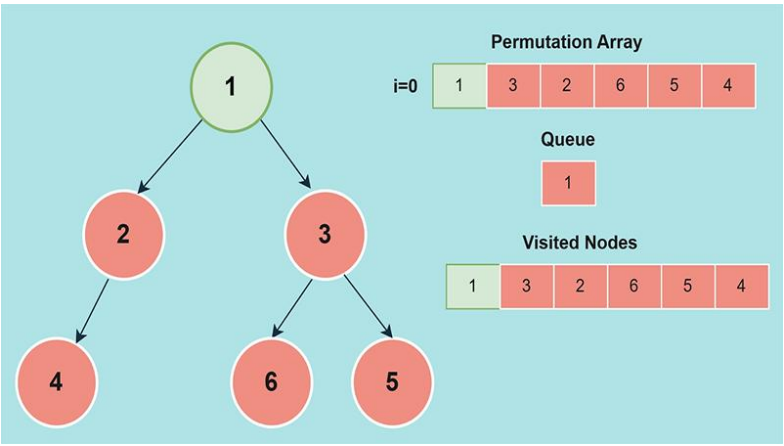
شروع با گره مبدأ (منبع)

اضافه کردن گره در جلوی صف نسبت به فهرست بازدید شده

.ایجاد فهرستی از گره‌های بازدید شده که در فاصله نزدیکی به آن رأس قرار دارند

خارج کردن گره‌های بازدید شده از صف

تکرار مراحل تا زمان خالی شدن صف



درخت های جستجو

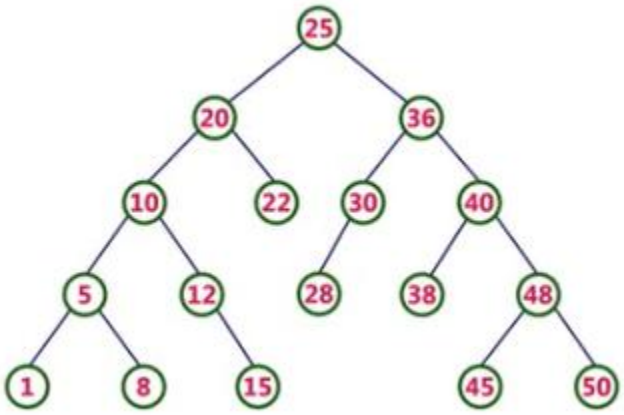
درخت های جستجو می توانند به دو صورت زیر باشند.

1- درخت جستجوی دودوئی (Binary Search Tree)

2- درخت جستجوی دودوئی متوازن (AVL[1])

درخت جستجوی دودوئی

برای تعریف درخت جستجوی دودوئی باید گفت، درختی است که در آن تمام گره های زیر درخت سمت چپ از ریشه کوچکتر هستند، همه گره های زیر درخت سمت راست از ریشه بزرگتر هستند و همچنین زیر درختهای سمت چپ و راست هر کدام خود درخت جستجو هستند. همچنین در این درخت گره با مقدار تکراری نداریم.



ارتفاع یک درخت دودوئی $logn \leq h \leq n$ می باشد.

[1] Adelson-Velsky & Landis

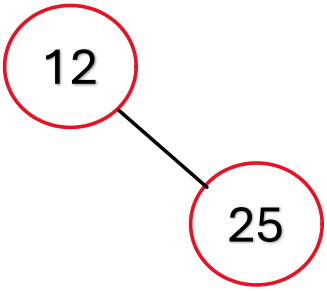
ایجاد درخت دودوئی

فرض کنیم یک سری عدد به صورت زیر در اختیار ما می باشد که می خواهیم برای آنها درخت دودوئی ترسیم کنیم.

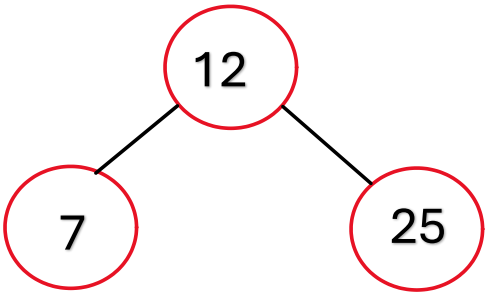
12,25,7,13,65,3

برای رسم درخت از سمت چپ به راست گره ها را می خوانیم. اولین گره با مقدار 12 در ریشه قرار می گیرد.

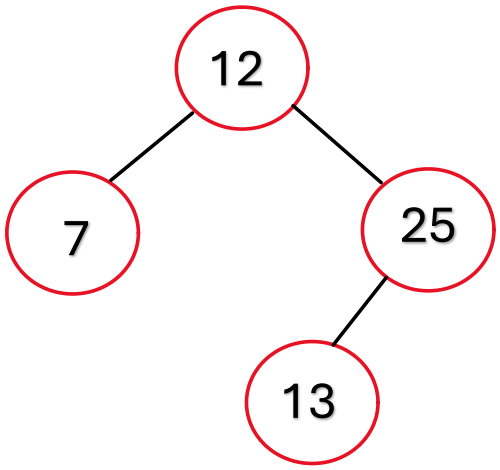
در مرحله بعد مقدار 25 خوانده می شود و چون از 12 بزرگتر است در سمت چپ قرار می گیرد.



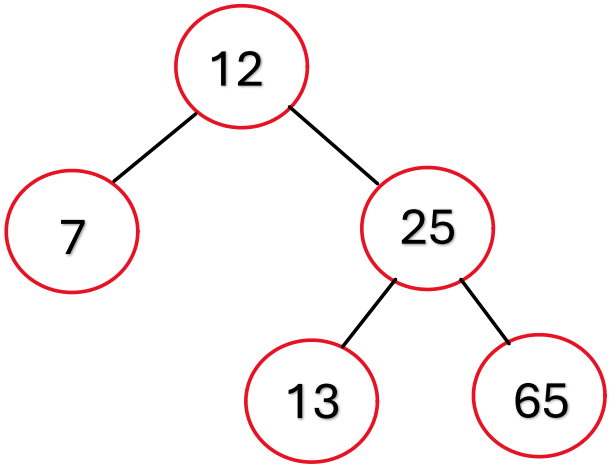
عدد بعد 7 می باشد و چون از 12 کوچکتر است در سمت راست این گره قرار می گیرد.



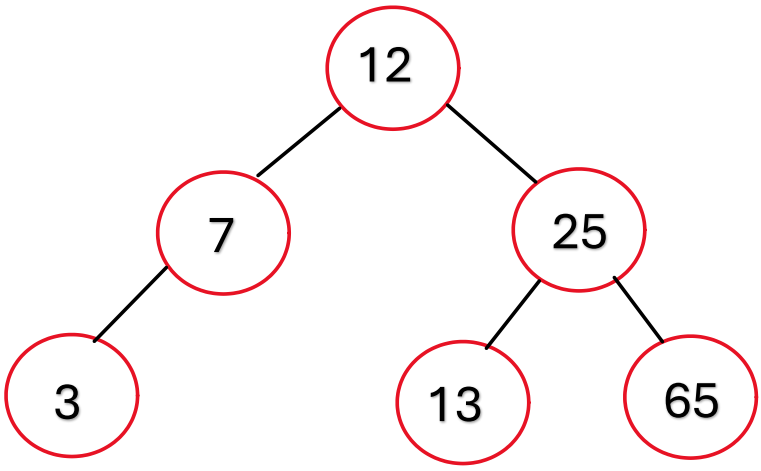
مقدار بعدی خوانده می شود 13 می باشد که از 12 بزرگتر است و باید سمت چپ درخت برود و از 25 کوچکتر است پس باید در سمت راست این گره قرار بگیرد.



در گام بعد، 65 باید در درخت قرار بگیرد. این عدد از 12 و 25 بزرگتر می باشد پس باید در سمت راست 25 قرار بگیرد.



در نهایت باید جای مناسبی برای 3 پیدا کنیم. چون این عدد از 12 و 7 کوچکتر می باشد باید در سمت راست گره 7 قرار بگیرد.

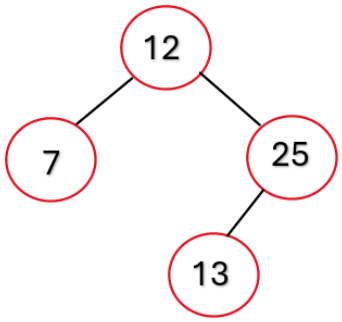


با این مراحل توانستیم درخت دودوئی برای اعداد داده شده را رسم کنیم.

درج در درخت دودوئی

الگوریتم زیر برای درج یک گره در درخت دودوئی به صورت بازگشتی ارائه شده است.

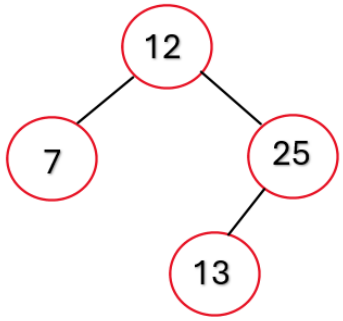
```
static unsafe Node* Insert(Node* start,int item)
{
    if (start == null)
        return CreateNode(item);
    if (item < start->Number)
        start->left=Insert(start->left,item);
    else if (item > start->Number)
        start->rigth = Insert(start->rigth, item);

    return start;
}
```



درج گره با مقدار 3 را با استفاده از الگوریتم بالا بررسی کنید.

جستجو در درخت دودوئی

الگوریتم زیر برای جستجو در درخت دودوئی به صورت بازگشتی و غیر بازگشتی ارائه شده است.




```
static unsafe Node* SearchRec(Node* root,int item)
{
    if (root == null || root->Number == item)
        return root;
    if (root->Number > item)
       return SearchRec(root->left, item);
    else
       return SearchRec(root->rigth, item);
}
```

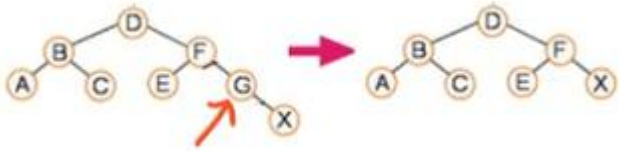
```
static unsafe bool Search(Node* root, int item)
{
   while(root!=null)
    {
        if (root->Number > item)
            root=root->left;
        else if(root->Number < item)
            root=root->rigth;
        else
            return true;
    }
    return false;
}
```

حذف گره از درخت جستجوی دودوئی

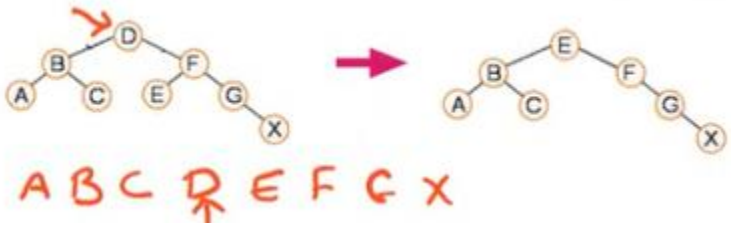
برای حذف از درخت دودوئی، پس از پیدا کردن گره ای که باید حذف شود، با سه حالت مواجه خواهیم بود و باید تنظیم درخت را انجام دهیم.

1- اگر گره ای که می خواهیم حذف کنیم برگ باشد، به راحتی حذف می شود و نیازی به تنظیم درخت نمی باشد.

2- اگر گره ای که می خواهیم حذف کنیم دارای یک فرزند باشد، گره فرزند به سمت بالا در درخت منتقل می شود.



3- اگر گره مورد نظر دارای دو فرزند باشد، گره قبل و یا گره بعد آن در پیمایش Inorder جایگزین آن می شود(گره حذف شده با بزرگترین مقدار عنصر زیر درخت سمت چپ و یا با کوچکترین مقدار عنصر زیر درخت سمت راست جایگزین می گردد).



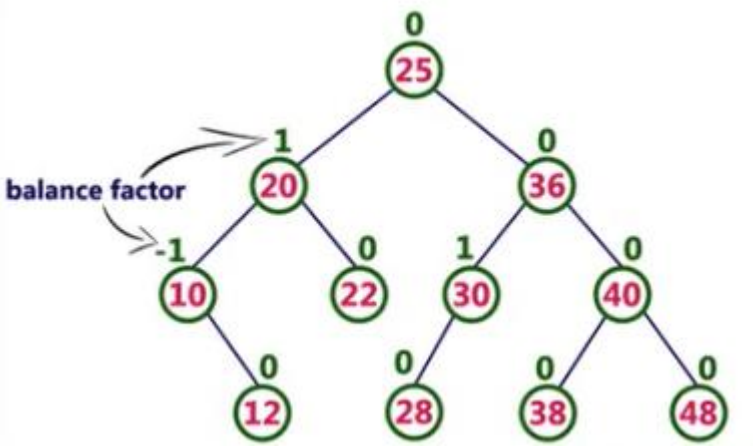
پیاده سازی به صورت زیر ارائه می گردد.

```
static unsafe Node* deleteNode(Node* root,int item)
{
    Node* temp;
    if (root == null)
        return root;
    if(item < root->Number)
        root->left=deleteNode(root->left,item);
    else if (item > root->Number)
        root->left = deleteNode(root->rigth, item);
    else
    {
        if (root->left == null) { temp = root->rigth; Free(root); return temp; }
        else if (root->rigth == null) { temp = root->left; Free(root); return temp; }

        temp = Min(root->rigth);
        root->Number=temp->Number;
        root->rigth=deleteNode(root->rigth,item);
    }
    return root;
}
```

درخت AVL

یک درخت دودوئی می باشد که حداکثر اختلاف ارتفاع دو زیر درخت چپ و راست آن 1 می باشد و هر یک از زیر درختها هم درخت AVL می باشند.

$$|h_l - h_r| \leq 1$$



حداکثر ارتفاع یک درخت AVL با n گره از o(logn) است.

مرتبه اجرایی هر یک از اعمال "جستجو، حذف، درج" در AVL ، برابر o(logn) است.

عملیات گردش

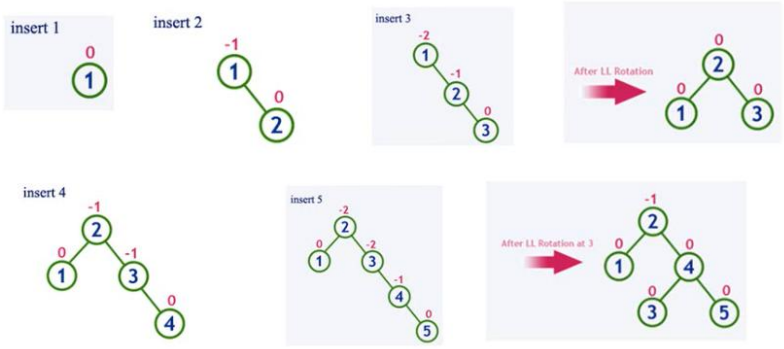
در اینجا ابتدا باید اصطلاحی را تعریف کنیم. این اصطلاح Balance Factor می باشد و برای بدست آوردن آن باید ارتفاع چپ و راست را بدست بیاوریم. در درخت شکل بالا می بینیم که برای هر گره محاسبه شده است.

در تعریف درخت AVL گفتیم که این اختلاف باید کمتر و یا مساوی 1 باشد چون این شرط باعث توازن می شود و این شرط باعث اختلاف درخت دودوئی و AVL می باشد. در تشکیل درخت AVL در هر مرحله ای که Balance Factor بیشتر از 1 باشد باید درخت را متوازن کنیم که این عمل را گردش می گوئیم.

تشکیل درخت AVL

تشکیل این نوع درخت دقیقا شبیه درخت دودوئی می باشد، فقط در حالتی که Balance Factor بیشتر از 1 شود نیاز به برقراری توازن(چرخش) می باشد.

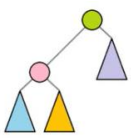
در شکل زیر و در مرحله سوم که مقدار 3 را به درخت اضافه می کنیم، Balance Factor گره ریشه 2- می شود که قدر مطلق آن بیشتر از 1 می باشد و این توازن را برهم می زند و باید با چرخش درخت را متوازن کنیم که مشاهده می شود بعد از این عمل توازن برقرار شده و درخت AVL معتبر می باشد. با ادامه ایجاد درخت مشاهده می شود که بعد از درج گره با مقدار 5 نیز مقدار Balance Factor غیر مجاز حاصل می شود که با یک چرخش دیگر توازن برقرار میشود.

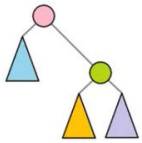


چرخش به راست

در تابع تعریف شده زیر ما عملیات چرخش را انجام داده ایم که با بررسی الگوریم و اعمال آن بر روی درخت سمت چپ شاهد متولد شدن درخت سمت راست خواهیم بود.

```
static unsafe Node* rigthRotation(Node* root)
{
    Node* x=root->left;
    Node* t=x->rigth;
    x->rigth=root;
    root->left = t;

    return root;
}
```

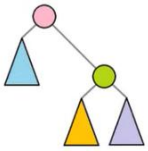


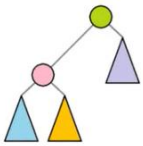


چرخش به چپ

در تابع تعریف شده زیر ما عملیات چرخش را انجام داده ایم.

```
static unsafe Node* leftRotation(Node* root)
{
    Node* x = root->rigth;
    Node* t = x->left;
    x->left = root;
    root->rigth = t;

    return root;
}
```



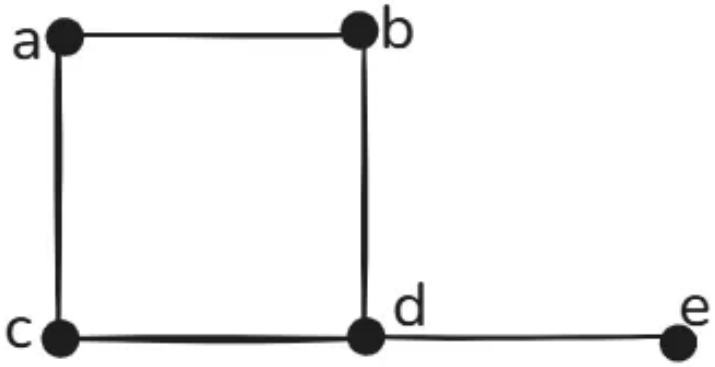


فصل 7

گراف

(Graph)

یک گراف، نمایشی تصویری از مجموعه اشیائی است که با هم ارتباط دارند. هر یک از این اشیا را راس یا گره می‌نامند. راس‌ها نیز از طریق یالها یا لبه‌ها با هم مرتبط هستند. اگر بخواهیم کمی رسمی‌تر بنویسیم، یک گراف را با (V,E) تعریف می‌کنیم که در آن، V مجموعه راس‌ها و E مجموعه یال‌ها است. شکل زیر، یک گراف را نشان می‌دهد. احتمالاً این شکل، تا حدودی شبیه همان چیزی است که تصور کرده‌اید.



در شکل بالا، V و E به‌صورت زیر هستند:

V = {a, b, c, d, e}

E = {ab, ac, bd, cd, de}

نمایش گراف

برای نماش گراف به دو صورت می شود کمل کردو

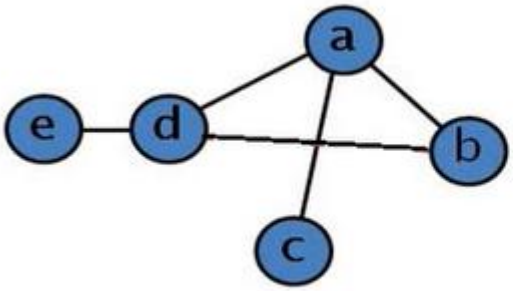
1- ماتریس همجواری[1]

2- لیست همجواری[2]

ماتریس همجواری

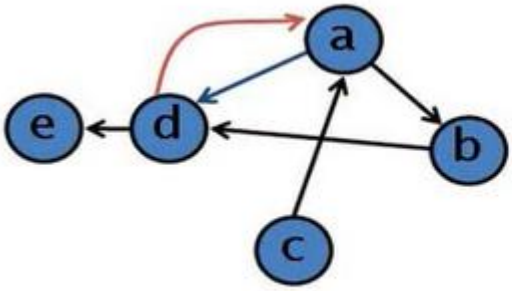
ماتریس همجواری برای گرافهای غیر جهت دار، به منظور مشخص کردن ارتباط بین گره های یک گراف ایجاد می شود و در اصل پیاده سازی گراف در حافظه می باشد. با اینجاد این گراف تمام رابطه های بین گره ها در گراف مشخص می شود. برای تشکیل این ماتریس، ماتریسی با تعداد سطر و ستون برابر و به تعداد گره های گراف ایجاد می شود. اندیس های این آرایه مقدار 0 یا 1 را خواهند داشت. سطر اول این ماتریس مربوط به گره a می باشد که مشخص می کند که گره a با چه گره هایی در ارتباط است. ستون اول ارتباط a با خودش می باشد، چون یالی از a به خودش وجود ندارد مقدار 0 در درایه 0,0 قرار می گیرد. در ستون بعد ارتباط بین a و b مشخص می شود. چون مسیر در بین این دو گره وجود دارد پس درایه 0,1 برابر 1 می شود. ارتباط بین a و c و d نیز وجود دارد پس مقدار درایه های 0,2 و 0,3 نیز 1 می شود. ولی ارتباطی بین a و e وجود ندارد پس درایه مربوط به آن برابر 0 قرار میگیرد. با این عمل سطر اول به پایان می رسد و همین اعمال در سطر های بعدی و برای گره های دیگر انجام می شود تا ماتریس کامل گردد.

[1] Adjacency Matrices
[2] Adjacency List



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

در تشکیل ماتریس همجواری برای گرافهای جهت دار دقیقا مانند بالا عمل می شود، فقط باید دقت شود که یالهای جهت دار ارتباط را محدود می کنند. مثلا در گراف زیر، گره a به d راه دارد ولی برعکس آن وجود ندارد. این مهم در ترسیم ماتریس همجواری باید مدنظر قرار گیرد.



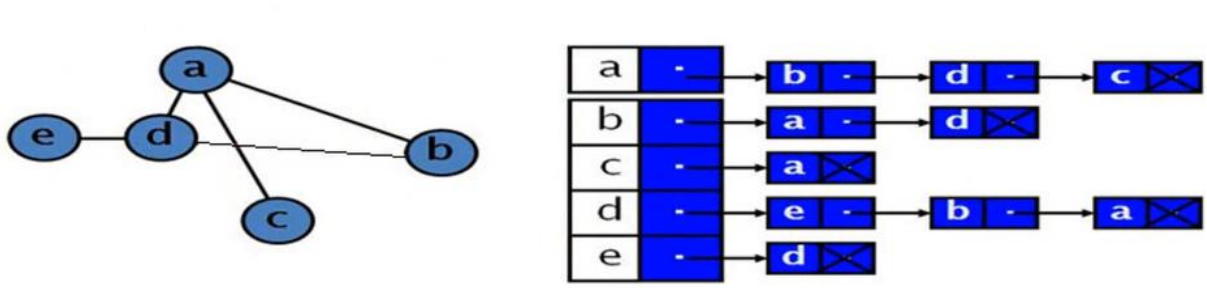
	a	b	c	d	e
a	0	1	0	1	0
b	0	0	0	1	0
c	1	0	0	0	0
d	1	0	0	0	1
e	0	0	0	0	0

لیست همجواری

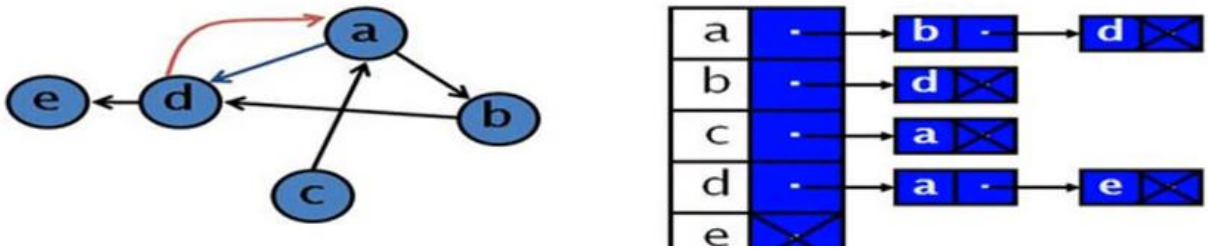
برای تشکیل لیست همجواری از لیست های پیوندی استفاده می شود. روش کار به اینصورت است که برای هر گره با کمک لیست های پیوندی تمام گره های همجوار مشخص می گردد. به عنوان مثال، برای راس a لیست پیوندی ایجاد می شود، که در داده لیست نام گره(a) قرار میگیرد و قسمت اشاره گر آن به گره دیگری اشاره می کند که گره مجاور گره a می باشد(مثلا گره b).

در گره مربوط به b و در قسمت اشاره گر آن به گره دیگر مجاور a یعنی d اشاره می شود و این فرایند ادامه پیدا می کند تا تمام گره های مجاور a در لیست قرار بگیرد.

این فرایند برای تمام گره ها و در لیست های پیوندی مجزا تکرار می شود.



در تشکیل لیست همجواری برای گرافهای جهت دار دقیقا مانند بالا عمل می شود، فقط باید دقت شود که یالهای جهت دار ارتباط را محدود می کنند. مثلا در گراف زیر، گره a به d راه دارد ولی برعکس آن وجود ندارد. این مهم در ترسیم ماتریس همجواری باید مدنظر قرار گیرد.



پیمایش گراف

گراف ها را می توان با دو روش زیر پیمایش کرد.

1- سطحی (BFS: Breadth First Search)

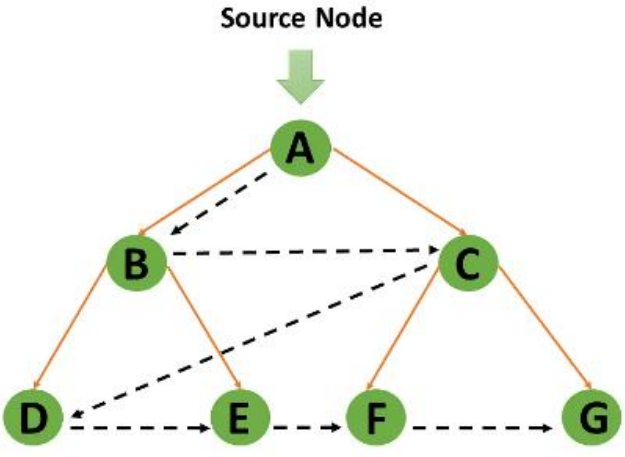
2- عمقی (DFS: Depth First Search)

پیمایش سطحی (BFS: Breadth First Search)

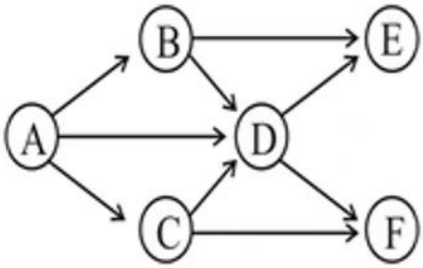
الگوریتم BFS ، جزو پرکاربردترین الگوریتم‌ها به‌حساب می یادBFS ، یک رویکرد پیمایش گراف و درخت می باشد که می‌توانیم ابتدا الگوریتم رو با گره مبدأ (منبع) آغاز کنیم و بعدش لایه به لایه، گره‌هایی رو که مستقیماً با گره مبدأ ارتباط دارن، تحلیل کنیم. در مرحله بعد پیمایشBFS ، باید گره‌های مجاور سطح بعد رو مشاهده کند.

بر اساسBFS ، شما باید گراف و با درخت را در جهت عرضی پیمایش کنین:

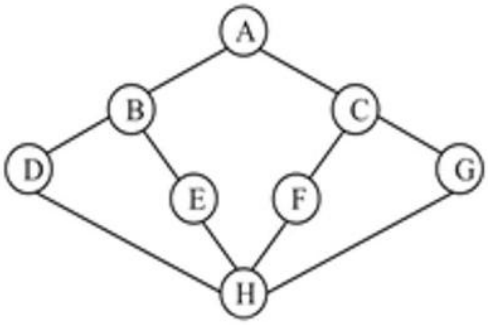
- برای شروع، به‌صورت افقی حرکت می کنیم و از کل گره‌های لایه موجود رو بازدید می کنیم
- سپس به لایه بعدی حرکت می کنیم.



مثال :



A B C D E F



A B C D E F G H

جستجوی اول سطح از ساختار داده صف برای نگهداری گره و بازدید شدن آن تا زمان بازدید از کل رئوس مجاور که ارتباط مستقیمی با اون گره دارن، استفاده می‌کنه. صف هم بر اساس اصل فایفو شکل می‌گیره و اولین گره نمایش داده شده، اون گرهی هست که اول از همه، قرار داده شده. پیمایش الگوریتم BFS در مورد گره‌ها به دو شکل انجام می‌شه:

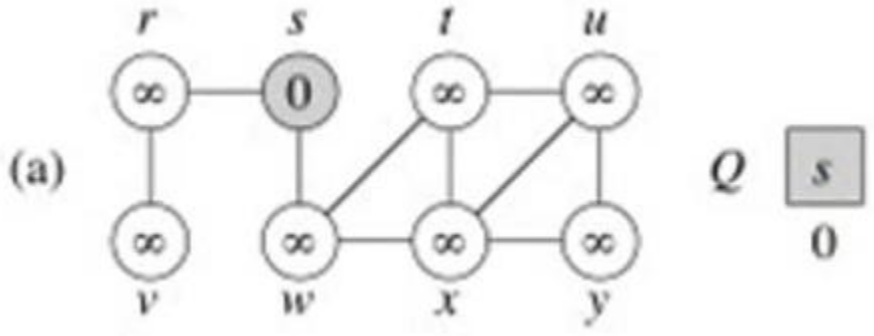
- گره بازدید شده
- گره بازدید نشده.

ترتیب مراحل اجرای الگوریتم جستجوی اول سطح

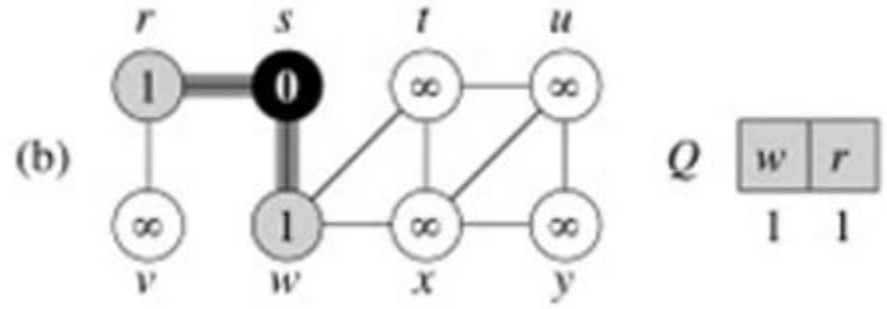
- شروع با گره مبدأ (منبع)
- اضافه کردن گره در جلوی صف نسبت به فهرست بازدید شده
- ایجاد فهرستی از گره‌های بازدید شده که در فاصله نزدیکی به آن رأس قرار دارند .
- خارج کردن گره‌های بازدید شده از صف
- تکرار مراحل تا زمان خالی شدن صف

مثال :

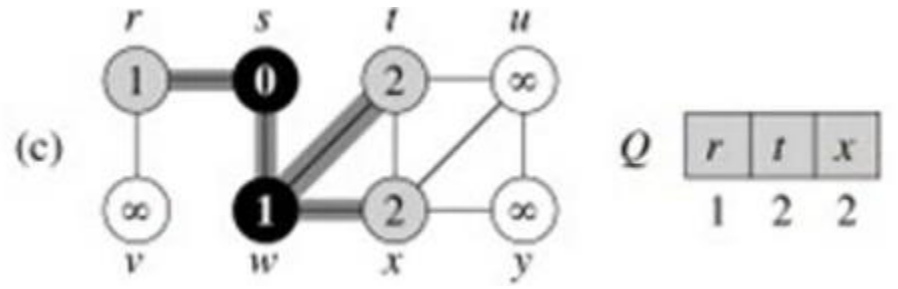
در مثال زیر s گره ریشه فرض می شود و در شروع با رنگ طوسی مشخص شده است و داخل صف Q قرار میگیرد. رنگ طوسی نشان می دهد که گره در صف می باشد و هنوز ملاقات(خوانده) نشده است. یک اندیس در صف در نظر گرفته شده است که با مراحل بالا این اندیس به عنوان برچسب ریشه انتخاب شده است که می تواند سطح گره را مشخص کند. در بقیه گره ها می بینید که برچسب گره ها نامشخص می باشد که به مرور اصلاح می شود.



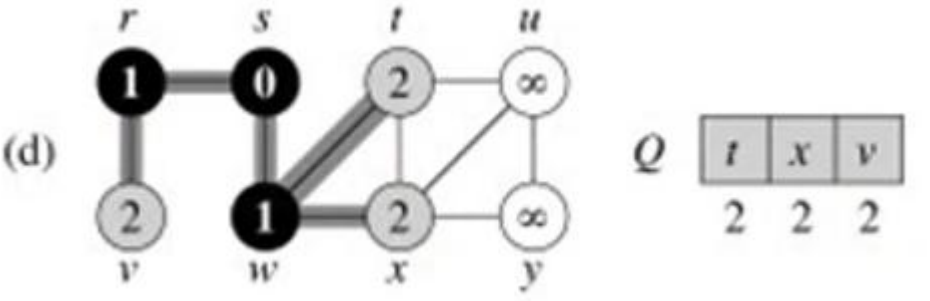
با ملاقات s و خارج شدن آن از صف رنگ گره مشکلی می شود و گره های همجوار آن(r,w) با برچسب جدید نشانه گذاری می شوند، رنگ جدید می گیرند و داخل صف قرار می گیرند. در اینجا فرض می شود که گره w در جلوی صف قرار گرفته است.



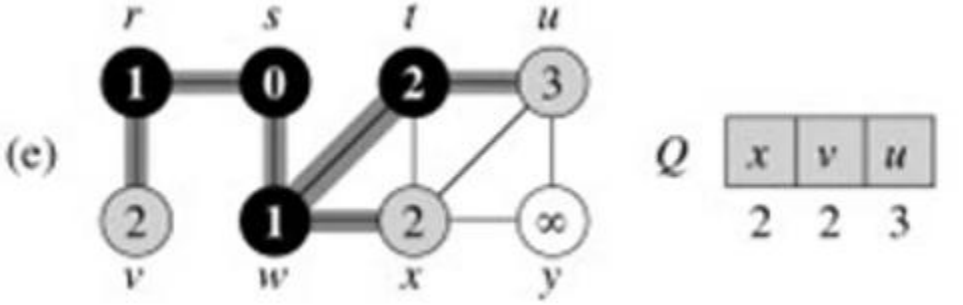
در مرحله بعد، چون w در ابتدای صف بود، ملاقات شده(رنگ آن مشکی می شود و از ابتدای صف خارج می شود) و گره های مجاور آن (t,x) با مقدار جدید در صف قرار میگیرند.



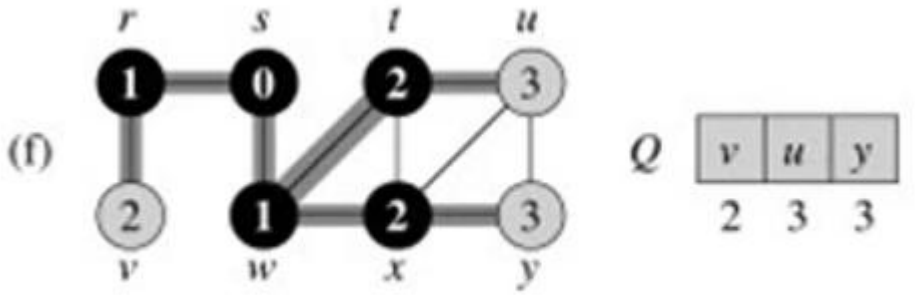
در گام بعد، r از ابتدای صف خارج شده و ملاقات می شود(رنگ آن مشکی می شود و از ابتدای صف خارج می شود). در نتیجه گره هم جوار آن(v) با برچسب جدید به انتهای صف منتقل می گردد.



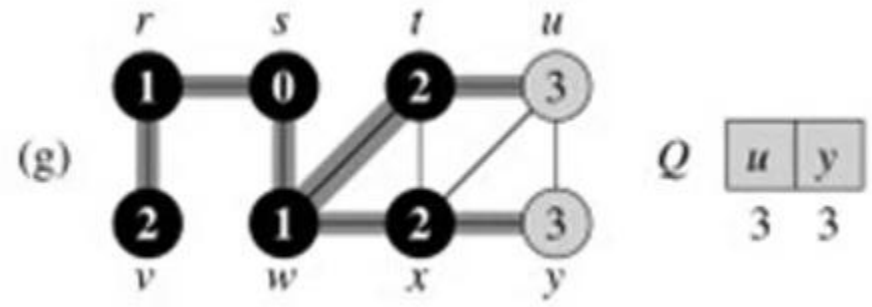
در این مرحله نوبت گره t می باشد که ملاقات شود. در نتیجه گره همجوار با آن(u) با برچسب جدید در انتهای صف قرار می گیرد.



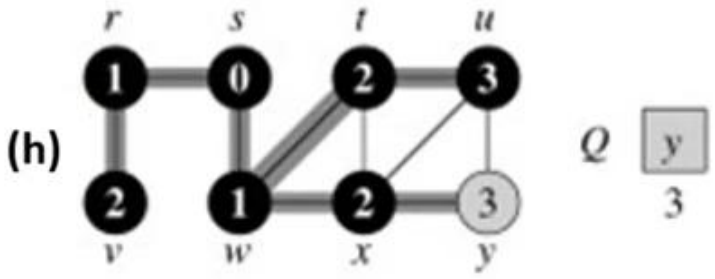
در مرحله بعد، نوبت گره x می باشد که ملاقات شود. در نتیجه گره همجوار با آن(y) با برچسب جدید در انتهای صف قرار می گیرد.



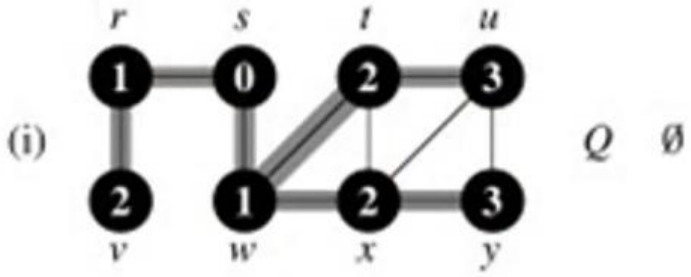
در گام بعد، نوبت گره v میرسد تا ملاقات شود و چون گره همجوار ندارد، هیچ گره ای به صف اضافه نمی شود.



در گام ماقبل آخر، نوبت گره u می باشد تا ملاقات شود و بدلیل اینکه گره همجوار آن نیز در صف می باشد و یا ملاقات شده اند، گره ای به صف وارد نمی شود.



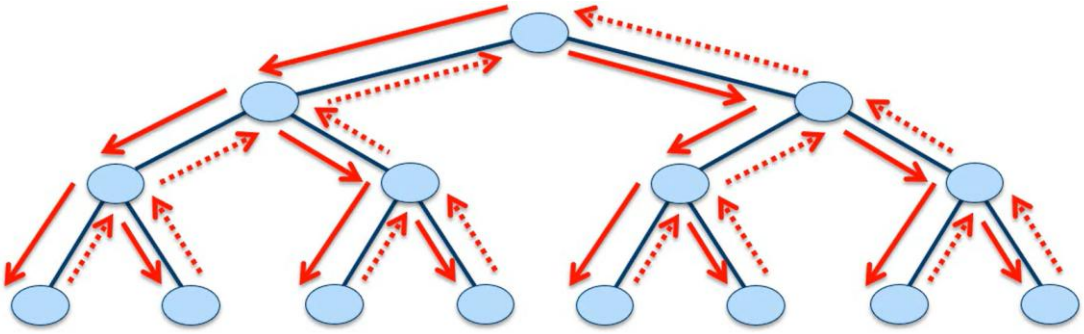
در نهایت تنها گره موجود در صف (y) ملاقات شده و با تهی شدن صف مراحل پیمایش به اتمام می رسد.



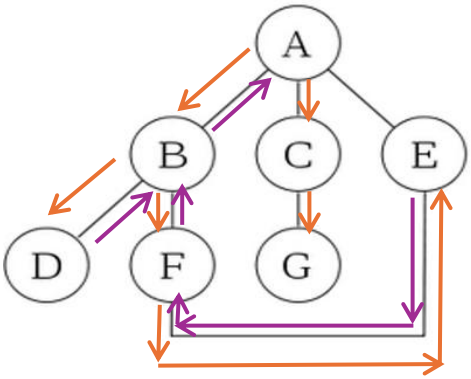

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2       u.color = WHITE
 3       u.d = ∞
 4       u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = Ø
 9  ENQUEUE(Q, s)
10  while Q ≠ Ø
11       u = DEQUEUE(Q)
12       for each v ∈ G.Adj[u]
13            if v.color == WHITE
14                 v.color = GRAY
15                 v.d = u.d + 1
16                 v.π = u
17                 ENQUEUE(Q, v)
18       u.color = BLACK
```

پیمایش عمقی (DFS: Depth First Search)

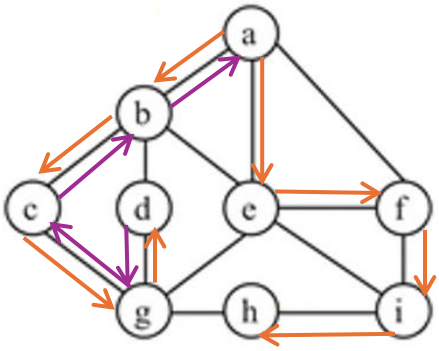
در این نوع از الگوریتم‌ها عمل پیمایش را از گره اول شروع می‌کنیم. در ابتدا قبل از عقب نشینی از هر شاخه، همه گره‌های آن شاخه را به ترتیب تا عمیق‌ترین سطح ممکن بازدید می‌کنیم. سپس گره‌های همه شاخه‌های دیگر نیز طبق همین روش باید بازدید شوند.



مثال:



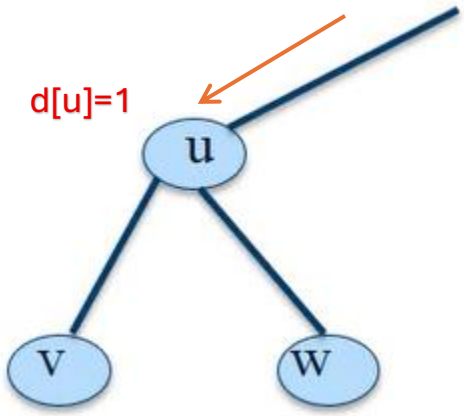
A B D F E C G



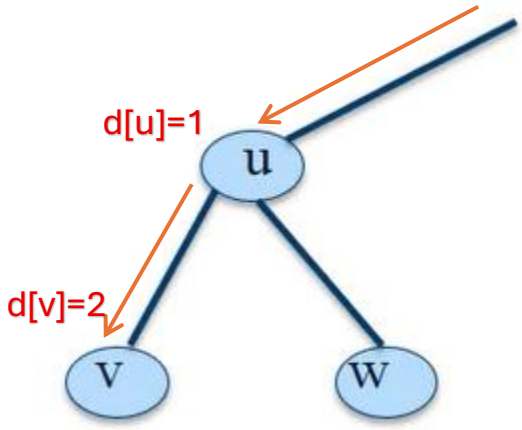
a b c g d e f i h

تعیین زمان کشف و انتظار

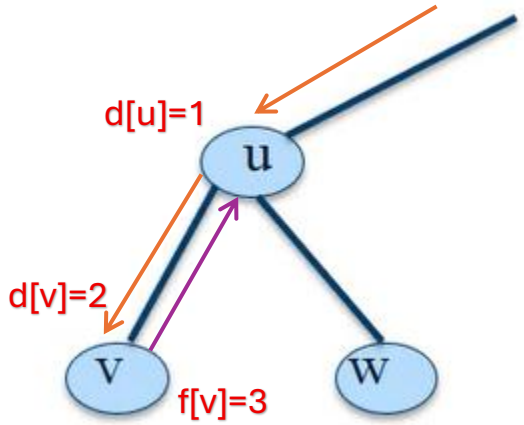
به منظور تعیین این دو زمان دو متغیر d[u] برای ملاقات یا کشف گره u و f[u] برای ترک آن در نظر می گیریم. توجه شود که این دو متغیر برای همه گره ها در نظر گرفته می شود.



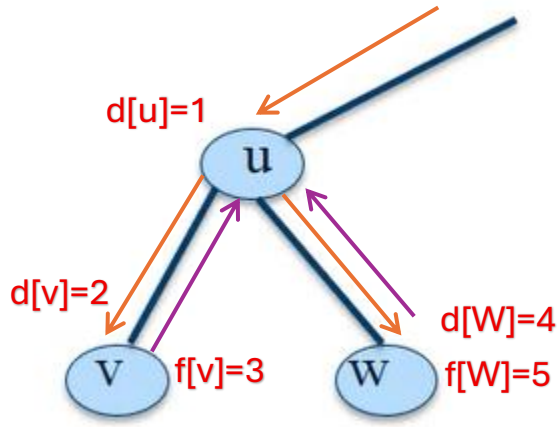
در مرحله اول وقتی گره u ملاقات می شود مقدار d[u] برابر 1 می شود که زمان اول می باشد.



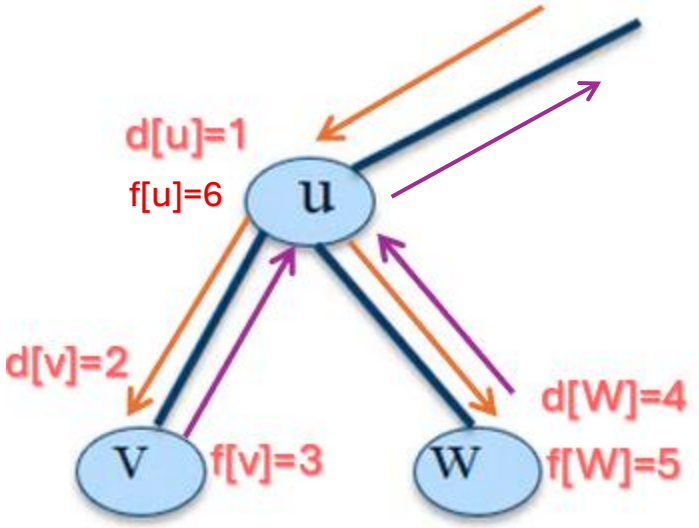
در مرحله بعد گره v ملاقات می شود و مقدار d[v] برابر 2 می شود که منظور در زمان 2 می باشد. پس تا اینجا گره u در زمان 1 و گره v در زمان 2 ملاقات شده است.



در این مرحله چون v گره همجواری ندارد ترک می شود و مقدار f[v] برابر 3 قرار می گیرد.



در مرحله بعد گره w ملاقات می شود و مقدار d[w] برابر4 می شود. در این مرحله چون W گره همجواری ندارد ترک می شود و مقدار f[W] برابر 5 قرار می گیرد و به گره u باز می گردیم.



در مرحله آخر چون همه گره های مجاور u بازدید شده و به پایان رسیده اند، u را نیز ترک می کنیم و f[u] را برابر 6 قرار می دهیم. مشاهده می شود که برای تمام گره ها زمان ملاقات و اتمام محاسبه شده است.

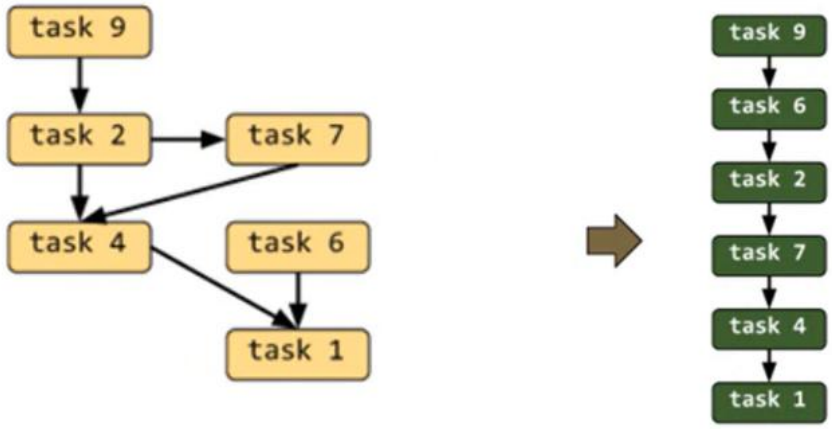
الگوریتم DFS

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
1  time = time + 1
2  u.d = time
3  u.color = GRAY
4  for each v ∈ G.Adj[u]
5      if v.color == WHITE
6          v.π = u
7          DFS-VISIT(G, v)
8  u.color = BLACK
9  time = time + 1
10 u.f = time
```

مرتب سازی موضعی[1]

یک مرتب‌سازی موضعی یا ترتیب موضعی یک گراف بدون دور جهت دار[2]، یک ترتیب خطی از همه رئوس آن است به‌طوری‌که هر گره قبل از همه گره‌هایی می‌آید که از آن به آن‌ها یال خارج شده‌است. هر درخت بدون دور جهت دار یک یا چند مرتب‌سازی موضعی دارد. کاربرد اصلی مرتب‌سازی موضعی (ترتیب موضعی) در زمانبندی یک سلسله‌ای از کارها یا وظایف است.

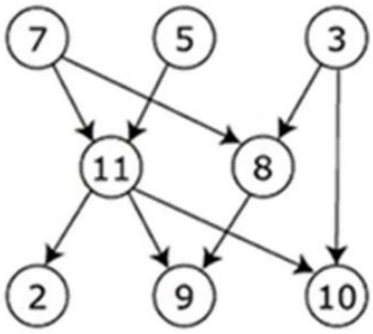


در شکل بالا که یک گراف از کارهایی که باید انجام شود داریم. کار های 9 و 6 که یال ورودی ندارند در خروجی قرار می گیرند که ترتیب آنها اهمیت ندارد. سپس چون کار 9 انجام شده است می توانیم کار 2 را در خروجی بیاوریم. حال نوبت به کار 7 می رسد که به خروجی برود. حال کار 4 می تواند در گراف ترتیب مد نظر اضافه شود و در نهایت به کار 1 می رسیم و در لیست قرار می دهیم.

در ترتیب سمت چپ کارها طوری مرتب شده اند که کار های وابسته به دیگران بعد آنها آمده است. به عنوان مثال کار 4 لازم است که بعد انجام کار 2 و 7 انجام شود. مشاهده می شود که در ترتیب کارها این مهم اتفاق افتاده و کار 4 بعد کارهای 2 و 7 آمده است.

[1] Topological Sort
[2] Directed Acyclic Graph(DAG)

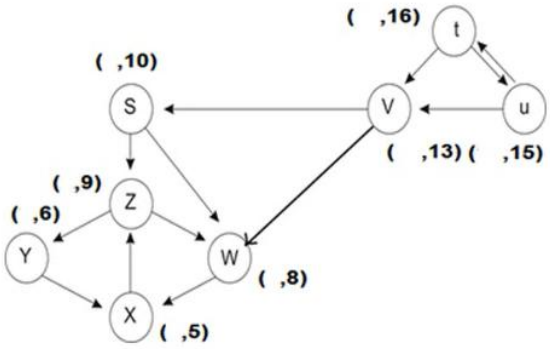
یک راه برای مرتب سازی توپولوژیکی این است که گره هایی که یال ورودی ندارند را ملاقات کنیم. پس از ملاقات آنها گره یال های خارج شده از آن را حذف کنیم و این کار را ادامه دهیم تا تمام گره ها ملاقات شوند. در تصویر زیر حالت های مختلف گراف سمت چپ را در سمت راست نشان داده شده است.

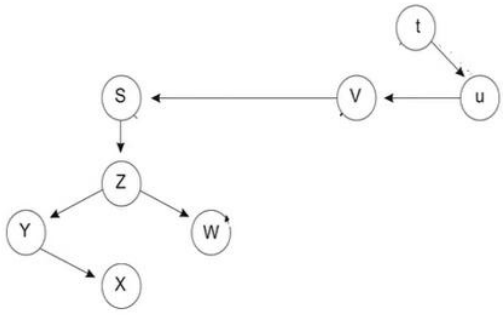


7, 5, 3, 11, 8, 2, 9, 10
3, 5, 7, 8, 11, 2, 9, 10
5, 7, 3, 8, 11, 10, 9, 2
7, 5, 11, 3, 10, 8, 9, 2
7, 5, 11, 2, 3, 8, 9, 10
3, 7, 8, 5, 11, 10, 2, 9

مرتب سازی توپولوژیکی بر اساس زمان خاتمه

گراف را به صورت عمقی پیمایش می کنیم و زمان خاتمه گره ها را بدست می آوریم. در مرحله بعد گره ها را به صورت نزولی بر اساس زمان خاتمه ملاقات می کنیم.

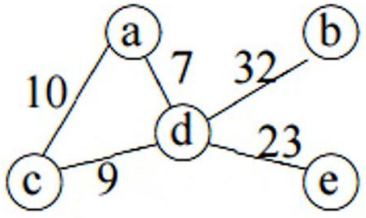




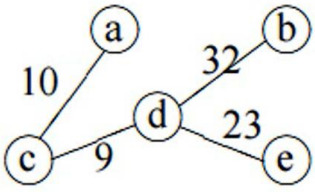
t u v s z w y x

درخت پوشای کمینه (Minimum Spanning Tree)

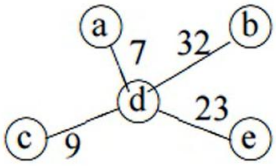
فرض کنید گراف یک گراف همبند باشد (یعنی بین هردو رأس متمایز آن یک مسیر وجود داشته باشد) منظور از یک درخت پوشا از این گراف درختی است که شامل همه رئوس این گراف باشد ولی فقط بعضی از یال‌های آن را دربر گیرد. منظور از درخت پوشای مینیمم (برای گراف همبند وزن دار) درختی است که بین درخت‌های پوشای آن گراف، مجموع وزن یال‌های آن، کمترین مقدار ممکن باشد.



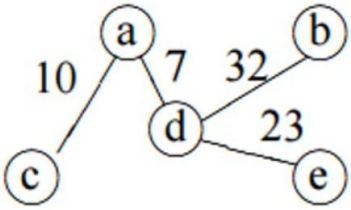
weighted graph



Tree 1. w=74



Tree 2, w=71
Minimum spanning tree



Tree 3, w=72

برای به دست آوردن درخت پوشای بهینه یک گراف جهت دار متصل می‌توان از الگوریتم‌های زیر استفاده کنیم:

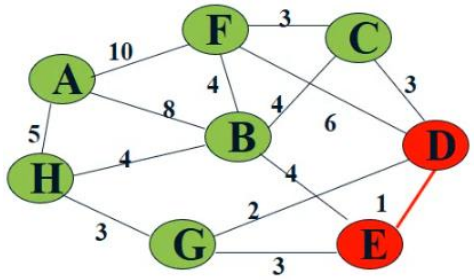
1- کروسکال(Kruskal)

2- پریم(Prim)

الگوریتم کروسکال(Kruskal)

این الگوریتم برای یافتن درخت پوشای کمینه یک گراف به کارمی رود. دراین الگوریتم ابتدا یال‌ها از کمترین وزن به بیشترین وزن مرتب می‌گردند سپس یال هابه ترتیب انتخاب شده و اگر یالی ایجاد حلقه کند، کنار گذاشته می‌شود. عملیات هنگامی خاتمه می‌یابد که تمام رأس هابه هم وصل شوند یا اینکه تعداد یال‌های موجود در F برابر n-1 شود که n تعداد رأس‌های گراف می باشد.

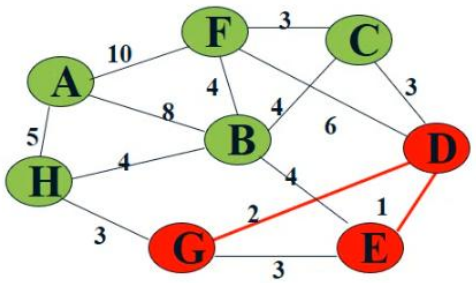
مثال: مطلوب است درخت پوشای کمینه در گراف زیر



edge	d_v	
(D,E)	1	√
(D,G)	2	
(E,G)	3	
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

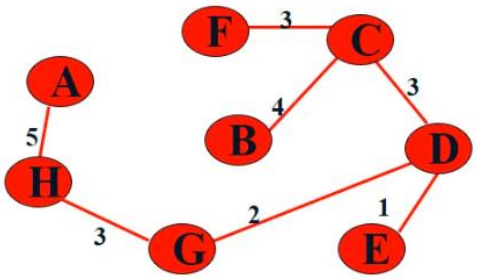
همانطور که مشاهده می کنید، یال های گراف بر اساس هزینه آنها در جدولی به صورت صعودی مرتب می شود و در انتخاب اول، کم هزینه ترین یال انتخاب می شود.



edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	√
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

در قسمت بعد، یال با وزن کمتر انتخاب می شود. ولی انتخاب بعدی (E,G) ایجاد حلقه می کند پس انتخاب نمی شود. این مراحل تکرار می شود تا تمام گره ها انتخاب شوند.



edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

edge	d_v	
(B,E)	4	χ
(B,F)	4	χ
(B,H)	4	χ
(A,H)	5	√
(D,F)	6	
(A,B)	8	
(A,F)	10	

not considered

Total Cost = 21

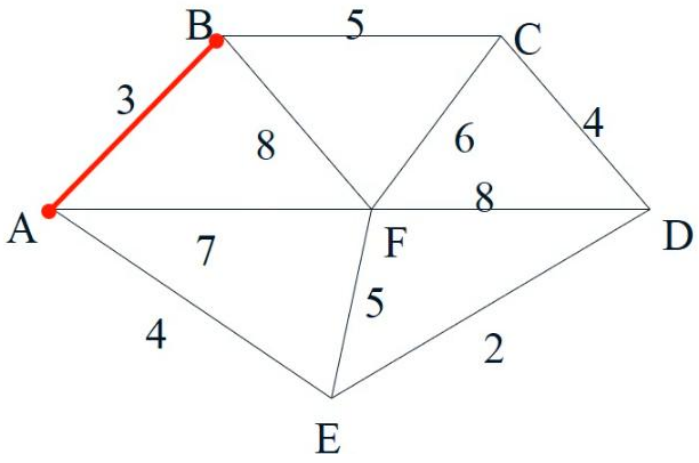
MST-Kruskal(G, w) $O(ElogE)$

```
1  A ← ∅
2  for each vertex v ∈ V[G]
3        do MAKE-SET(v)
4  sort the edges of E into nondecreasing order by weight w
5  for each edge (u, v) ∈ E, taken in nondecreasing order by weight
6        do if FIND-SET(u) ≠ FIND-SET(v)
7              then A ← A ∪ {(u, v)}
8                   UNION(u, v)
9  return A
```

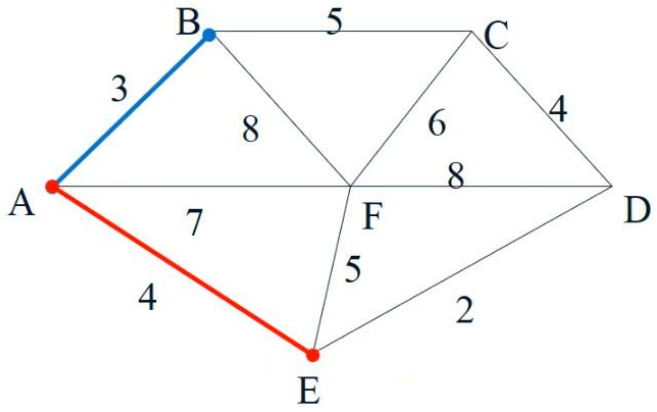
الگوریتم پریم(Prim)

در این روش از یک رأس شروع می‌کنیم و کمترین یال (یال با کمترین وزن) که از آن می‌گذرد را انتخاب می‌کنیم. در مرحله بعد یالی انتخاب می‌شود که کمترین وزن را در بین یال‌هایی که از دو گره موجود می‌گذرد داشته باشیم. به همین ترتیب در مرحله بعد یالی انتخاب می‌گردد که کمترین وزن را در بین یالهایی که از سه گره موجود می‌گذرد داشته باشد. این روال را آنقدر تکرار می‌کنیم تا درخت پوشای بهینه حاصل شود. باید توجه کرد که یال انتخابی در هر مرحله در صورتی انتخاب می‌شود که در گراف دور ایجاد نکند. تفاوت روش پریم با روش کراسکال در این است که گراف حاصل در مراحل میانی تشکیل درخت پوشای بهینه در روش پریم همیشه متصل است ولی در الگوریتم کراسکال در آخرین مرحله قطعاً متصل است.

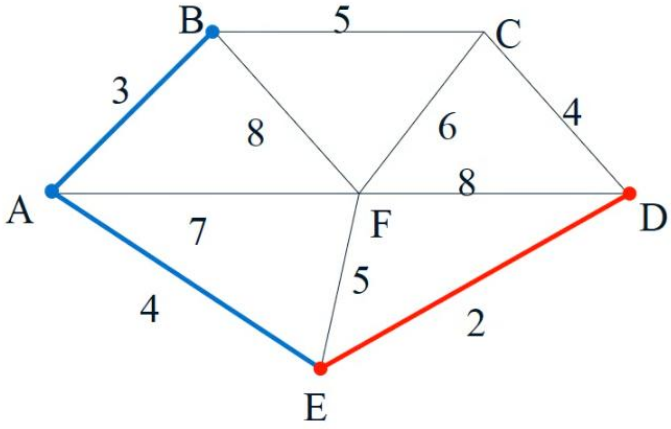
مثال: مطلوب است درخت پوشای کمینه برای گراف زیر



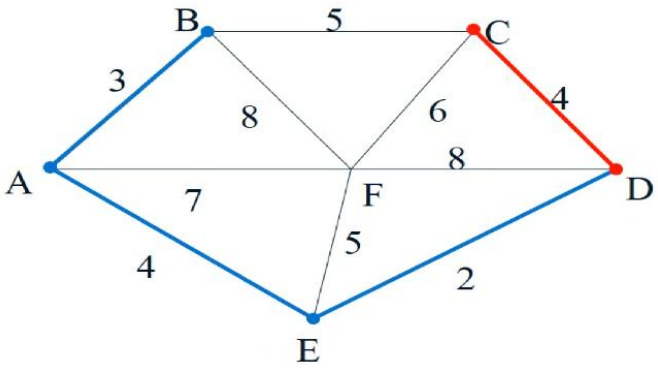
در این الگوریتم، یک گره به دلخواه انتخاب می شود که در اینجا گره A انتخاب شده است. مسیر های گره های همجوار گره انتخاب شده بررسی می شود و کم هزینه ترین مسیر برگزیده می شود. در اینجا یال متصل به B کمترین هزینه را دارد و انتخاب می شود.



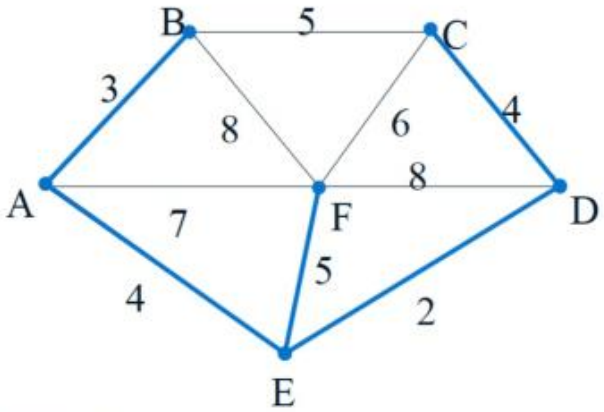
در گام بعد، باید کلیه مسیر ها بین گره های همجوار با A و B بررسی شود و کم هزینه ترین مسیر انتخاب شود. در اینجا مسیر A,E با هزینه 4 انتخاب شده است.



در این مرحله، گره ها و مسیرهای مربوط به A,B,E بررسی می شود که در این بین مسیر E,D به دلیل هزینه کمتر انتخاب می شود.



در این مرحله، مسیر ها و گره های مربوط به گره های A,B,E,D بررسی می شود و مسیر D,C انتخاب می گردد.



Total weight of tree: 18

در انتها وقتی در بین مسیر های موجود می خواهیم کم هزینه ترین مسیر را انتخاب کنیم، مسیر E,F انتخاب می شود و در نهایت درخت پوشای کمینه با هزینه 18 بدست می آید.

```
MST-Prim(G, w, r)
1   for each u ∈ V[G]
2         do key[u] ← ∞
3            π[u] ← NIL
4   key[r] ← 0
5   Q ← V[G]
6   while Q ≠ ∅
7         do u ← EXTRACT-MIN(Q)
8            for each v ∈ Adj[u]
9                  do if v ∈ Q and w(u, v) < key[v]
10                       then π[v] ← u
11                            key[v] ← w(u, v)
```