ساختمان داده‌ها

استاد: آقای دکتر رضا نورمندی

ساختمان داده‌ها (Data Structures): سازمان‌دهی منطقی و ریاضی داده‌ها در حافظه را ساختمان داده می‌گویند که به بهترین نحو می‌توان از آن استفاده کرد.

انواع روش‌های سازمان‌دهی داده‌ها (ساختمان داده)

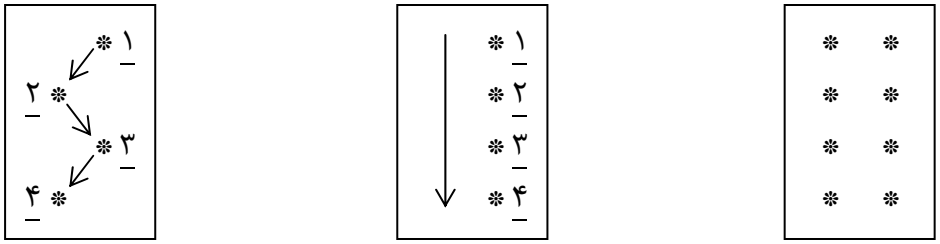
- خطی
  - ۱- آرایه (Array)
  - ۲- لیست پیوندی (Linked List)
  - ۳- پشته (Stack)
  - ۴- صف (Queue)
- غیر خطی
  - ۵- درخت (Tree)
  - ۶- گراف (Graph)

انواع ساختمان داده‌ها

۱- خطی: داده‌ها تشکیل یک لیست خطی می‌دهند.

۲- غیر خطی: داده‌ها تشکیل یک لیست خطی را نمی‌دهند.

لیست خطی: ترتیب داده‌ها مشخص باشد و پشت سرهم باشند یا نباشند.



غیر خطی — خطی پشت سرهم — خطی اشاره‌گری

انواع ساختمان داده‌های خطی

۱- داده‌ها پشت سرهم در خانه‌های متوالی از حافظه ذخیره می‌شوند. (آرایه)

۲- داده‌ها در موقعیت‌های مختلف از حافظه ذخیره می‌شوند و توسط اشاره‌گرها به هم پیوند خورده و تشکیل یک لیست خطی را می‌دهند. (لیست پیوندی)

عملیات اصلی بر روی ساختمان داده‌ها:

۱- **پیمایش:** دسترسی به عناصر ساختمان داده دقیقاً یک بار به طوری که بتوان آن عنصر را پردازش کرد.

۲- **جستجو:** پیدا کردن یک یا چند عنصر از ساختمان داده که بر اساس یک شرط یا شرایط خاصی صورت می‌گیرد.

۳- **اضافه:** اضافه کردن یک عنصر به ساختمان داده به طوری که طول ساختمان داده یک واحد اضافه شود.

٤- **حذف:** حذف کردن یک عنصر از ساختمان داده به طوری که طول ساختمان داده یک واحد کم شود.

عملیات فرعی بر روی ساختمان داده‌ها:

۱- **مرتب‌سازی:** چیدن عناصر ساختمان داده با یک نظم خاصی در کنار هم را مرتب‌سازی می‌گویند.

۲- **ادغام:** قرار دادن دو ساختمان داده از قبل مرتب شده در یک ساختمان داده مرتب شده را گویند.

الگوریتم ادغام:

مرحله اول: تا زمانی که $i <= n$ و $j <= m$ است تکرار کن

اگر $A[i] < B[j]$

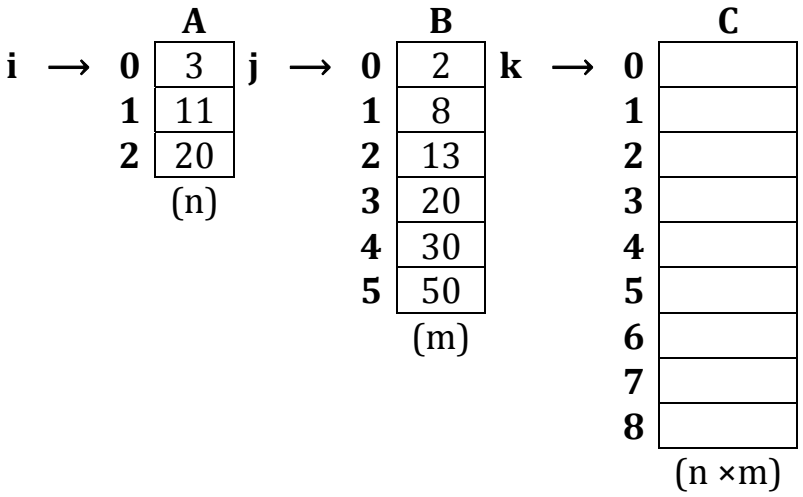
$$C[k] \longleftarrow A[i] \quad \begin{matrix} i++ \\ j++ \end{matrix}$$

در غیر این صورت

$$C[k] \longleftarrow B[j] \text{ و } \begin{matrix} j++ \\ i++ \end{matrix}$$

مرحله دوم: اگر $i > n$ است از موقعیت j تا انتهای آرایه B را یکجا در C کپی کنید.

اگر $j > m$ است از موقعیت i تا انتهای آرایه A را یکجا در C کپی کنید.



تمرین ۱) الگوریتم ادغام را بر اساس وضعیت داده شده زیر بنویسید.

```
merge (int a[], int b[], int c[], int n, int m)
{
        ...
        ...
}
```

الگوریتم: مراحلی که برای رسیدن به یک هدف طی می‌شود و متشکل از تعدادی دستورالعمل است که گام به گام اجرا می‌شوند تا ما را به هدف برسانند.

پیچیدگی الگوریتم (Complexity): تابعی است که زمـــان اجرا و مقدار حـــافظه مصرفی الگوریتم را بر حسب تعداد ورودی‌های الگوریتم می‌دهد و با $f(n)$ یا $C(n)$ نمایش می‌دهند.

معیارهای پیچیدگی الگوریتم:

زمان اجرا: با شمارش عملیات کلیدی الگوریتم بدست می‌آید.

مقدار حافظه: با شمارش حافظه مورد نیاز الگوریتم بدست می‌آید.

عمل جستجو: عمل کلیدی (در بدنه حلقه‌ها وجود دارد) یا مقایسه است.

حالت‌های مختلف پیچیدگی الگوریتم‌ها:

۱- بهترین حالت: $C(n) = 1$

۲- حالت متوسط: $C(n) = \frac{n+1}{2}$

۳- بدترین حالت: $C(n) = n$

O بزرگ (Big O): نرخ رشد تابع $C(n)$ را بر حسب رشد n می‌دهد.

$C(n) = 2n^2 + 1$

$n = 100 \longrightarrow 2 * 10000 + 1$

$n = 1000 \longrightarrow 2 * 1000000 + 1$

⋮ ⋮

$n >> \longrightarrow n^2 \Longrightarrow O(n^2)$

مثال۱) نرخ رشد تابع C(n) زیر را حساب کنید.

$C(n) = 2n^4 + n + 3 = O(n^4)$

برای بدست آوردن O باید به nهای تابع داده شده نگاه کرد و بزرگ‌ترین آن را به عنوان پیچیدگی در نظر گرفت. در تابع بالا دو n وجود دارد یکی n^4 و دیگری n که در تابع بالا n^4 از n بزرگ‌تر است و به عنوان پیچیدگی در نظر گرفته می‌شود.

مثال۲) نرخ رشد تابع C(n) زیر را حساب کنید.

$C(n) = 2n\log_2^n + 3n + 1 = O(n\log_2^n)$

تابع بالا شامل دو n است یکی n و ضریب لگاریتمی و دیگری n، در تابع بالا n با ضریب لگاریتم از n بزرگ‌تر است و به عنوان پیچیدگی در نظر گرفته می‌شود.

مثال۳) تعداد تکرار دستور cout را بدست آورید.

```
x = 1;                         ← 1
for (int i = 0 ; i < n ; i++)   (0 → n+1 , n → n-1)   (n-1)-0+1=n⟹n+1
        x *= 2;                ← n
```

برای محاسبه پیچیدگی Forهای افزایشی به صورت زیر عمل می‌شود:

1 + حد اولیه اندیس - حد نهایی اندیس

$f(n) = 1 + (n + 1) + n = 2n + 2$

دستور انتساب، ۱بار اجرا می‌شود. حلقه For چون i < n است و از صفر تا n-1 بار و در مجموع n بار و ۱ بار هم برای خروج و در نهایت n+1 بار اجرا می‌شود. خط سوم چون i از صفر تا n-1 بار اجرا می‌شود n بار عمل ضرب انجام می‌شود.

مثال٤) تعداد تکرار دستور cout را بدست آورید.

```
for (int i = 0 ; i < n ; i++)      (n-1)-0+1=n⟹n+1
        cout << i;
```

$f(n) = n + n + 1 = 2n + 1$

حلقه For چون i < n است و از صفر تا n-1 بار و در مجموع n بار و ۱ بار هم برای خروج و در نهایت n+1 بار اجرا می‌شود. خط دوم i را n بار چاپ می‌کند.

مثال٥) تعداد تکرار دستور cout را بدست آورید.

```
for (int i = 0 ; i < n ; i++)  ← n+1
        cout << a[i];          ← n
```

$f(n) = n + n + 1 = 2n + 1$

حلقه For چون i < n است و از صفر تا n-1 بار و در مجموع n بار و ۱ بار هم برای خروج و در نهایت n+1 بار اجرا می‌شود. خط دوم n بار اجرا می‌شود.

مثال٦) تعداد تکرار دستور cout را بدست آورید.

```
for (i = 0 ; i < n ; i++)          ← n+1
        for (j = 1 ; j < n ; j++)  ← n     n⟹n*n=n²
                cout << i * j;     ← n-1   n(n-1)
```

$f(n) = (n + 1) + n^2 + n(n - 1) = 2n^2 + 1$

For اول چون i < n است از صفر تا n-1 بار و در مجموع n بار و ۱ بار هم برای خروج و در نهایت n+1 بار اجرا می‌شود. For دوم چون j < n است از ۱ تا n-1 بار و در مجموع n-1 بار و ۱ بار هم برای خروج و در نهایت n بار اجرا می‌شود و چون در بدنه حلقه i قرار دارد و خود حلقه i نیز n بار انجام می‌شود، کلاً n^2 بار حلقه j اجرا می‌شود. دستور cout برای چاپ n-1 بار اجرا می‌شود و چون در بدنه حلقه j قرار دارد و حلقه j با n بار تکرار اجرا می‌شود، دستور cout کلاً n(n-1) بار اجرا می‌شود.

مثال۷) تعداد تکرار دستور cout را بدست آورید.

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

```
        cout << i * j;  ← n²
```

$f(n) = (n + 1) + n(n + 1) + n^2 = n + 1 + n^2 + n + n^2 = 2n^2 + 2n + 1$

For اول چون $i < n$ است از صفر تا $n-1$ بار و در مجموع n بار و ۱ بار هم برای خروج و در نهایت $n+1$ بار اجرا می‌شود. For دوم چون $j < n$ است از صفر تا $n-1$ بار و در مجموع n بار و ۱ بار هم برای خروج و در نهایت $n+1$ بار اجرا می‌شود و چون در بدنه حلقه i قرار دارد و خود حلقه i نیز n بار انجام می‌شود، کلاً $n(n+1)$ بار حلقه j اجرا می‌شود. دستور cout برای چاپ n بار اجرا می‌شود و چون در بدنه حلقه j قرار دارد و حلقه j با n بار تکرار اجرا می‌شود، دستور cout کلاً n^2 بار اجرا می‌شود.

مثال۸) تعداد تکرار دستور cout را بدست آورید.

```
for (i = 0 ; i <= n-1 ; i++)  ← n+1
    for (j = 0 ; j <= n-1 ; j++)  ← n(n+1)
        cout << i + j;  ← n²
```

$f(n) = (n + 1) + n(n + 1) + n^2 = n + 1 + n^2 + n + n^2 = 2n^2 + 2n + 1$

توضیحات این مثال همانند مثال ۷ می‌باشد.

مثال۹) تعداد تکرار دستور cout را بدست آورید. (حلقه بینهایت بار اجرا می‌شود)

```
for (i = 0 ; i <= n-2 ; i++)
    for (j = i+1 ; j <= n-1 ; j++)
        cout << i * j;
```

i	j	cout
0	1 ⟶ n-1	n-1
1	2 ⟶ n-1	n-2
2	3 ⟶ n-1	n-3
⋮	⋮	⋮
n-3	n-2 ⟶ n-1	2
n-2	n-1 ⟶ n-1	1

i: 0 + 1 + 2 + ... + (n-2) = n-1+1 = بارn

j: -1 + 1 + 2 + 3 + ... + n $= \frac{n(n+1)}{2} - 1$

cout: (n-1) + (n-2) + (n-3) + ... + 1 = $\frac{n(n-1)}{2}$

پیچیدگی زمانی در حلقه‌های تو در تو که شمارنده حلقه‌ها به یکدیگر وابسته هستند $\frac{n(n+1)}{2}$ می‌باشد. در نتیجه For اول چون $i <= n-2$ است از صفر تا $n-2$ بار و در مجموع $n-1$ بار و ۱ بار هم برای خروج و در نهایت n بار اجرا می‌شود. For دوم $j <= n-1$ است و از $i+1$ و به دلیل وابسته بودن به For اول، چون i از 0 شروع می‌شود در نتیجه j از 1 شروع و تا $n-1$ بار اجرا می‌شود و در هر بار اجرا ۱ بار هم از حلقه خارج شده تا به i اضافه گردد، در نتیجه به آن در هر مرحله ۱ واحد اضافه می‌شود. پس j از 2 تا n بار اجرا می‌گردد و چون جمع این اعداد $\frac{n(n+1)}{2}$ نمی‌شود با جمع این اعداد در یک +1 و -1 به $\frac{n(n+1)}{2} - 1$ دست پیدا خواهیم کرد که پیچیدگی حلقه For دوم خواهد بود. دستور cout چون در بدنه For دوم قرار دارد از ۱ تا $n-1$بار اجرا می‌گردد و با جمع کردن این اعداد $\frac{n(n-1)}{2}$ بدست می‌آید که پیچیدگی دستور cout می‌باشد.

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

مثال۱۰) تعداد تکرار دستور cout را بدست آورید.

```
for (i = 0 ; i < n-1 ; i++)   (n-2)   (n-1)
    for (j = i+1 ; j < n ; j++)
        cout << i * j;
```

i	j	cout
0	1 ⟶ n-1	n-1
1	2 ⟶ n-1	n-2
2	3 ⟶ n-1	n-3
⋮	⋮	⋮
n-3	n-2 ⟶ n-1	2
n-2	n-1 ⟶ n-1	1

i: 0 + 1 + 2 + ... + (n-2) = n-1+1 = بارn

j: -1 + 1 + 2 + 3 + ... + n $= \frac{n(n+1)}{2} - 1$

cout: (n-1) + (n-2) + (n-3) + ... + 1 = $\frac{n(n-1)}{2}$

توضیحات این مثال همانند مثال ۹ می‌باشد.

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

تمرین ۲) پیچیدگی الگوریتم زیر را محاسبه کنید.

```
for (i = 0 ; i < n-1 ; i++)
        for (j = i ; j < n ; j++)
                for (k =1 ; k < n ; k++)
                        cout << i * j * k;
```

تمرین ۳) پیچیدگی الگوریتم زیر را محاسبه کنید.

```
for (i = 0 ; i <= n-2 ; i++)
        for (j = i+1 ; j <= n-1 ; j++)
                for (k = j ; k <= n-1 ; k++)
                        cout << i * j * k;
```

تمرین ٤) پیچیدگی الگوریتم زیر را محاسبه کنید.

```
for (i = 1 ; i <= n ; i *= 2)
        cout << i;
```

تمرین ٥) پیچیدگی الگوریتم زیر را محاسبه کنید.

```
for (i = n ; i >= 1 ; i /= 2)
        cout << i;
```

تمرین ٦) پیچیدگی الگوریتم زیر را محاسبه کنید.

```
i = 1;
while (i < n)
{
        cout << i;
        i * 2;
}
```

تمرین ۷) پیچیدگی الگوریتم زیر را محاسبه کنید.

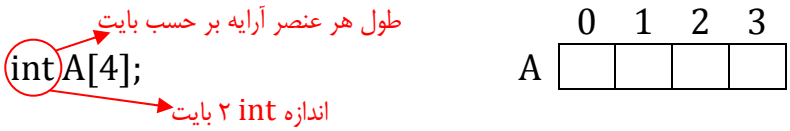
```
i = n;
while (i >= 0)
{
        cout << i;
        i /= 2;
}
```

تمرین ۸) پیچیدگی الگوریتم زیر را محاسبه کنید.

```
i = 0;
while (i < n)
{
        cout << i;
        i += 2;
}
```

«آرایه – Array»

آرایه: ساختمان داده‌ای است که از تعداد محدودی عنصر هم نوع تشکیل شده است؛ به طوری که: ۱- عناصر پشت سرهم در خانه‌های متوالی از حافظه ذخیره شده‌اند. ۲- هر عنصر توسط یک اندیس $0 \le i \le n-1$ در دسترس قرار می‌گیرد.



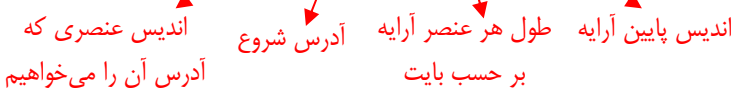
فرمول بدست آوردن طول آرایه:

طول آرایه بر اساس تعداد عناصر = UB – LB + 1

طول آرایه بر حسب بایت = W(UB – LB + 1)

فرمول بدست آوردن خانه‌ای از آرایه:

آدرس عنصر (A[K] = Base(A) + W(K – LB))



مثال ۱۱) آرایه‌ای که در حافظه از آدرس ۱۰۰ ذخیره شده است. اگر طول هر عنصر از آرایه ۲ بایت باشد و اندیس‌های آرایه از صفر شروع شده باشند. آدرس عنصری که در خانه با اندیس ۲ آرایه ذخیره شده است را بدست آورید.

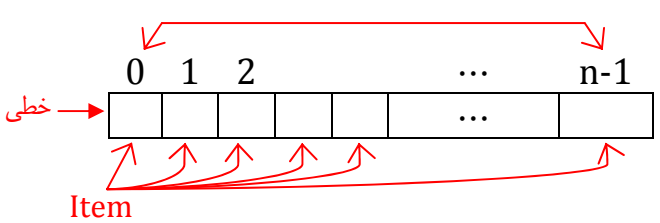
(a[2] = 100 + 2[(2 – 0) = 104

عملیات بر روی آرایه:

پیمایش آرایه:

جهت انجام کارهای گوناگون باید آرایه را پیمایش کرد. در زیر تابع پیمایش آرایه به منظور چاپ تک تک عناصر آرایه آمده است.

```
int survey (int A[], int n)
{
        for (int i = 0 ; i < n ; i++)
                cout << A[i];
}
```



O (n)

جستجو در آرایه: ۱- خطی ۲- دودویی

جستجوی خطی: در روش جستجوی خطی (Linear Search) یا جستجوی ترتیبی (Sequential Search)، عنصر مورد جستجو با هر یک از عناصر آرایه مقایسه می‌شود، چنانچه دو عنصر برابر بودند، عمل جستجو به پایان می‌رسد و اندیس عنصر برگردانده می‌شود وگرنه مقایسه با عنصر بعدی آرایه انجام می‌پذیرد. از آنجا که عناصر آرایه نامرتب می‌باشند، عنصر مورد جستجو در هر کجای آرایه می‌تواند باشد لذا عمل مقایسه تا یافتن عنصر مورد نظر و یا رسیدن به انتهای آرایه یعنی جستجو در همه عناصر آرایه ادامه می‌یابد.

```
int linearsearch (int A[], int n, int item)
{
        for (int i = 0 ; i < n ; i++) ← n+1          int ⟶ 2 byte
                if (A[i] == item) ← n
                        return i; ← 1
                return -1; ← 1          f(n) = 2n + 3 ⟹ O(n) بدترین حالت
}
```

جستجوی دودویی: روش جستجوی دودویی (Binary Search) در آرایه‌های مرتب شده قابل استفاده می‌باشد و از سرعت بالایی برخوردار می‌باشد. در این الگوریتم، در هر بار مقایسه، نیمی از عناصر آرایه حذف می‌شوند. الگوریتم عنصر میانی آرایه را می‌یابد و آن را با عنصر مورد جستجو، مقایسه می‌کند. اگر برابر بودند، جستجو به پایان رسیده و اندیس عنصر برگردانده می‌شود، در غیـر این صورت عمل جستجو روی نیمی از عناصر انجام می‌گیرد. اگر عنصر مورد جستجو کوچک‌تر از عنصر میانی باشد، جستجو روی نیمه اول آرایه صورت می‌پذیرد، در غیر این صورت نیمه دوم آرایه جستجو می‌شود. این جستجوی جدید روی زیر آرایه طبق الگوریتم جستجو روی آرایه اصلی انجام می‌شود یعنی عنصر میانی زیر آرایه یافته می‌شود و با عنصر مورد جستجو مقایسه می‌گردد، اگر برابر نباشند زیر آرایه مجدداً نصف می‌شود و در هر بار جستجو زیر آرایه‌ها کوچک‌تر می‌گردند. عمل جستجو تا یافتن عنصر مورد نظر (یعنی برابر بودن عنصر مورد جستجو با عنصر میانی یکی از زیر آرایه‌ها) و یا نیافتن عنصر مورد نظر (برابر نبودن عنصر مورد جستجو با عنصر زیر آرایه‌ای شامل تنها یک عنصر) ادامه می‌یابد.

شرایط انجام الگوریتم جستجوی دودویی:

۱- مرتب باشد.

۲- دسترسی مستقیم به عنصر وسط امکان‌پذیر باشد.

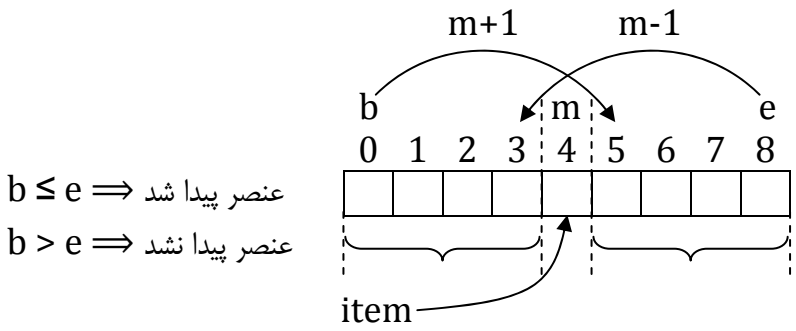
$$\text{اندیس وسط} = \frac{\text{اندیس انتها} + \text{اندیس اول}}{2}$$

برای الگوریتم جستجوی دودویی سه اندیس زیر تعریف می‌شود:

b: اندیس ابتدای آرایه مورد جستجو

e: اندیس انتهای آرایه مورد جستجو

m: اندیس وسط آرایه مورد جستجو $m = \frac{b+e}{2}$




```
int binarysearch (int A[], int n, int item)
{
        int b = 0 ; e = n-1;
```

```
while (b <= e)
{
        int m = (b + e) / 2;
        if (item == A[i])
                return m;        ← جستجوی موفق
        if (item < A[i])
                e = m - 1;
        else
                b = m + 1;
}
return -1;        ← جستجوی ناموفق
}
```

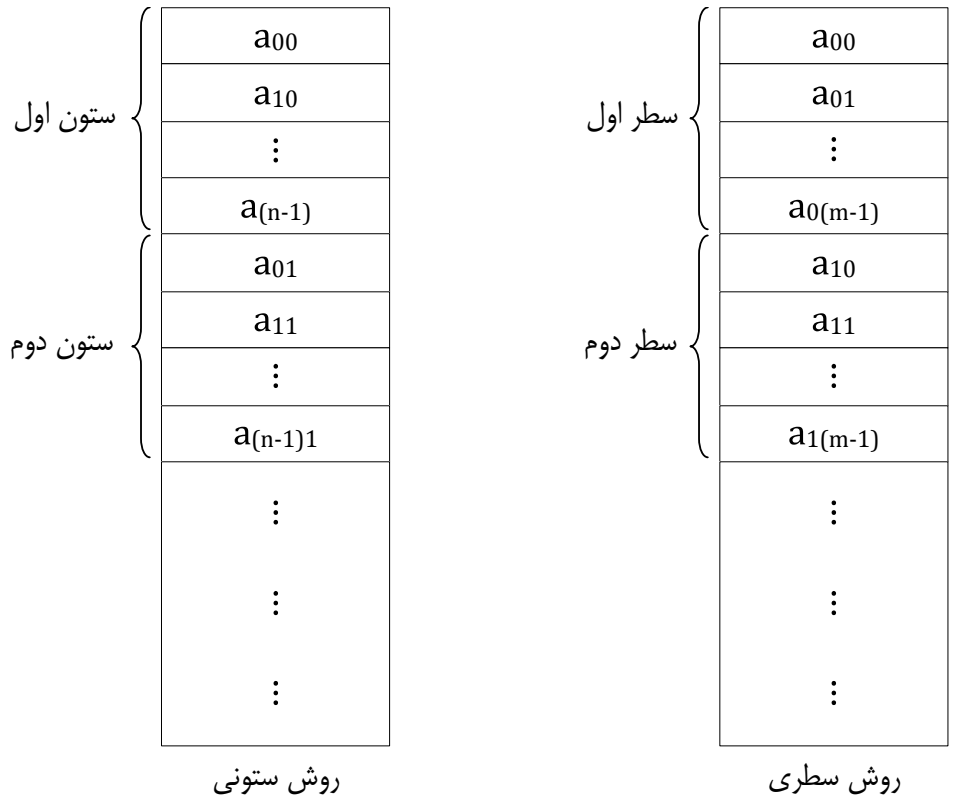
b > e

بدترین حالت $f(n) = \lfloor \log_2^n \rfloor + 1 \Longrightarrow O(\log_2^n)$

ماتریس‌ها: ماتریس یکی از مفاهیم ریاضی است که در بسیاری از مسائل از جمله علوم کامپیوتر و فیزیک کاربرد دارد. در واقع به هر آرایه دو بعدی $m \times n$ یک ماتریس یا جدول با m ستون و n سطر گفته می‌شود. که تعداد mn خانه در آن وجود دارد.

$$\begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0(m-1)} \\ a_{10} & a_{11} & \cdots & a_{1(m-1)} \\ \vdots & \vdots & a_{ij} & \vdots \\ a_{(n-1)0} & a_{(n-1)1} & \cdots & a_{(n-1)(m-1)} \end{bmatrix}_{m \times n}$$

نحوه ذخیره ماتریس در حافظه:



روش ستونی — روش سطری

$$a[LB_1 \cdots UB_1 , LB_2 \cdots UB_2]$$

کران پایین سطر — کران بالای سطر — کران پایین ستون — کران بالای ستون

$$a[0 \cdots n-1 , 0 \cdots m-1]$$

تعداد سطرها $n: UB_1 - LB_1 + 1$

تعداد ستون‌ها $m: UB_2 - LB_2 + 1$

آدرس (روش سطری) $(a[i,j]) = Base(a) + W[m(i - LB_1) + (j - LB_2)]$

آدرس (روش ستونی) $(a[i,j]) = Base(a) + W[(i - LB_1) + n(j - LB_2)]$

مثال۱۲) آرایه دو بعدی A[-3 .. 4 , 2 .. 8] در حافظه از آدرس ۱۰۰ به روش ستونی ذخیره شده‌اند. اگر طول هر عنصر از آرایه ۴ بایت باشد. مطلوب است آدرس A[-1 , 5] را بدست آورید.

$$(a[-1 , 5]) = 100 + 4[(-1 - -4) + 8(5 - 2)] = 204$$

مثال۱۳) آرایه دو بعدی A[-3 .. 5 , 6 .. 12] در حافظه از آدرس ۱۰۰ با اعداد صحیح ذخیره شده‌اند. مطلوب است آدرس A[-1 , 8] را به روش سطری بدست آورید.

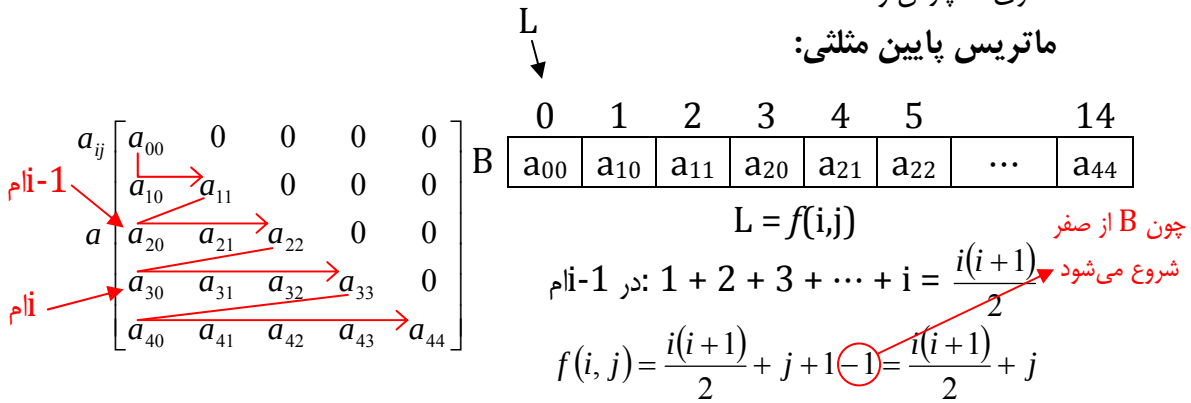
تعداد سطرها $n: UB_1 - LB_1 + 1 = 5 - -3 + 1 = 9$

تعداد ستون‌ها $m: UB_2 - LB_2 + 1 = 12 - 6 + 1 = 7$

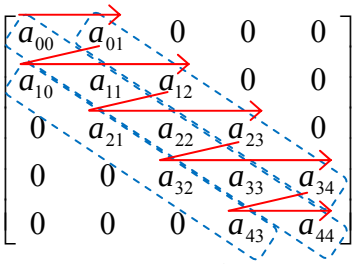
$$(a[-1 , 8]) = 100 + 2[7(-1 - -3) + (8 - 6)] = 132$$

ماتریس‌های خلوت: ماتریسی که عناصر صفر آن‌ها زیاد است و نسبتاً تعداد کمی عنصر غیر صفر دارند به عبارت دیگر اکثر درایه‌های آن صفر است، را ماتریس خلوت یا پراکنده می‌نامند. مانند: مثلثی، قطری، ۳-قطری، اسپارس و ...

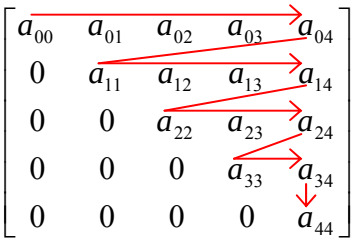
ماتریس پایین مثلثی:



تمرین۹) چگونگی ذخیره و نحوه دسترسی به عناصر ماتریس ۳-قطری زیر را بدست آورید.



تمرین۱۰) چگونگی ذخیره و نحوه دسترسی به عناصر ماتریس بالا مثلثی زیر را بدست آورید.



ماتریس پراکنده:

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

4×4
16

درایه	سطر	ستون
3	0	1
5	1	0
2	1	3
1	2	0
6	3	2

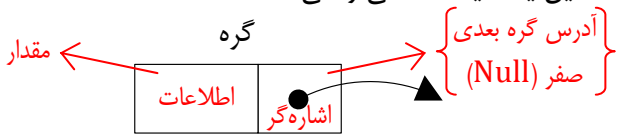
تعداد درایه‌های غیر صفر

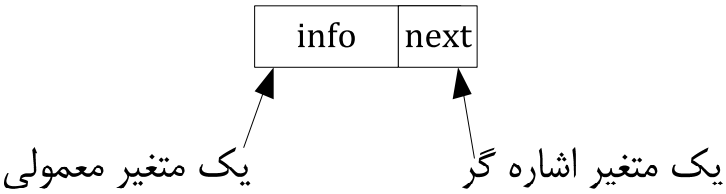
K × 3

5 × 3

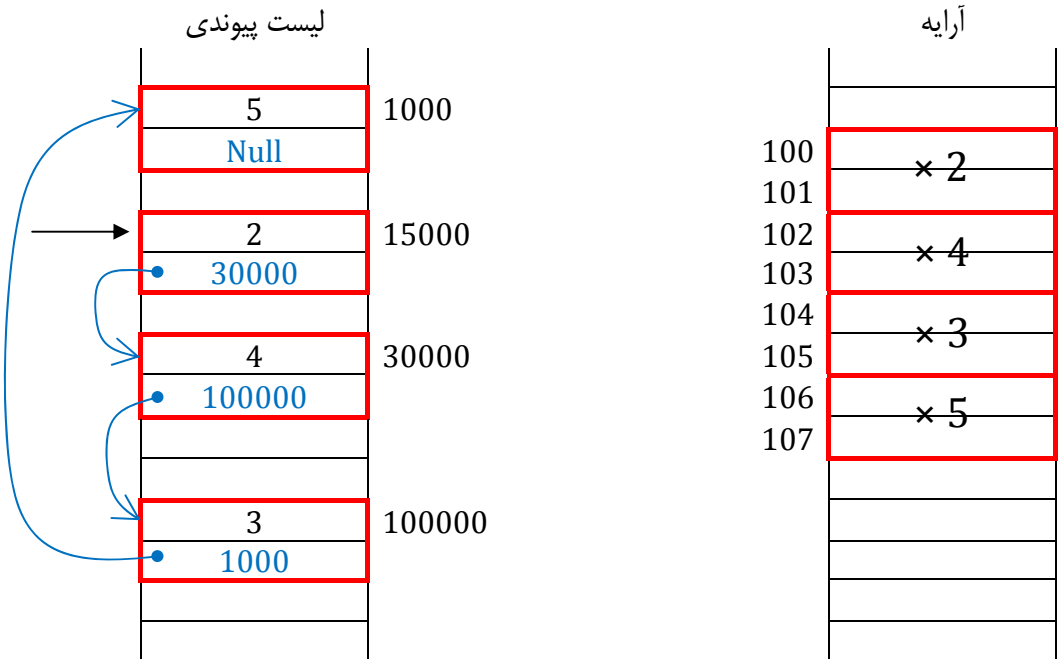
«لیست پیوندی – Linked List»

لیست پیوندی: ساختمان داده‌ای است که از مقدار محدودی گره (Node) تشکیل شده است که توسط اشاره‌گرها به هم پیوند خورده و تشکیل یک لیست خطی را می‌دهد.

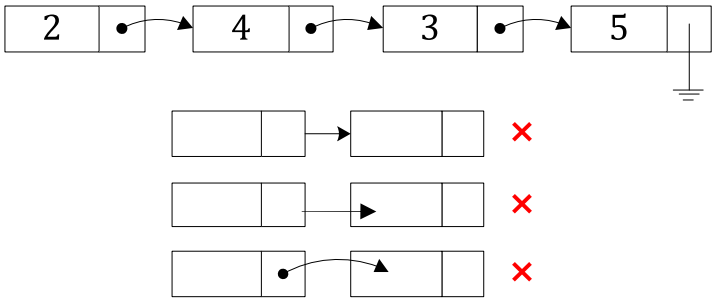




مثال ۱۴) نمایش چگونگی قرارگیری اعداد ۲، ۴، ۳ و ۵ در آرایه و لیست پیوندی



لیست پیوندی یک طرفه: یک لیست پیوندی یک طرفه (Singly-Linked List) دنباله‌ای از عناصر داده‌ای به نام گره (node) است که ترتیب خطی آن‌ها توسط اشاره‌گرها تعیین می‌گردد. عناصر لیست تنها می‌توانند به ترتیب از ابتدای لیست تا انتها مورد دسترسی قرار بگیرند. هر گره آدرس گره بعدی را شامل می‌شود که به این صورت امکان پیمایش از یک گره به گره بعدی فراهم می‌شود. برای رسم لیست پیوندی گره‌ها به صورت مستطیل‌هایی پشت سرهم رسم می‌شوند که توسط فلش‌هایی به هم متصل شده‌اند.



ترسیم اشتباه لیست‌های پیوندی

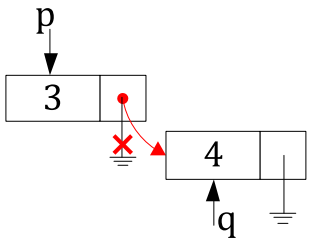
ایجاد کردن یک گره در حافظه:

۱- تعریف ساختار گره:

```
struct node
{
        int info;
        node *next;
};
```

۲- ایجاد گره (New):

```
node *p;
      p = new node;
      p → info = 3;
      p → next = null;
```



دستورات بالا باعث ایجاد یک گره می‌شوند که اشاره‌گر p به آن اشاره می‌کند و قسمت آدرس گره بعدی null می‌باشد.

```
node *q = new node;
      p → info = 4;
      p → next = null;
```

دستورات بالا گره‌ای دیگر ایجاد می‌کنند که اشاره‌گر q به آن اشاره می‌کند و قسمت آدرس گره بعدی null می‌باشد.

نکته: برای ایجاد پیوند بین دو گره باید حتماً آدرس دو گره را داشته باشیم.

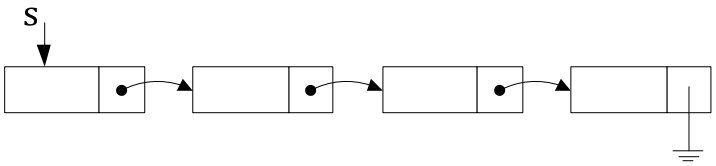
۳- تابع ایجاد گره:

```
node *makenode (int x)
{
      node *p = new node;
      p → info = x;
      p → next = null;
      return p;
}
```

برای جلوگیری از نوشتن دستورات تکراری از دستور Makenode استفاده می‌کنیم. می‌توان برای ساخت دو گره قبل با استفاده از تابع ایجاد گره، به صورت زیر عمل کرد.

```
node *p = makenode (3);
node *q = makenode (4);
p → next = q;
```

پیمایش لیست پیوندی:




```
node *p = s;
while (p != null)
{
      cout << p → info;
      p = p → next;
}
```

حرکت روی لیست

آدرس گره بعدی

تمرین ۱۱) مطلوب است شمارش تعداد گره‌های یک لیست پیوندی با آدرس شروع s.

جستجو در لیست پیوندی (خطی):

```
node *linearsearch (node *s , int item)
{
      node *p = s;
      while (p != null)
      {
            if (p → info == item)
                  return p;
            p = p → next;
      }
      return null;
}
```

پیمایش لیست و جستجوی داده مورد نظر

جستجوی موفق

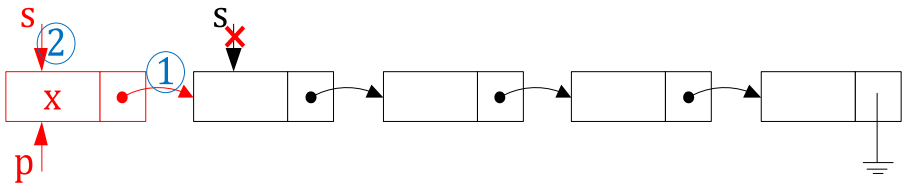
جستجوی ناموفق

نکته: روی لیست‌های پیوندی امکان جستجوی دودویی وجود ندارد.

تمرین ۱۲) در جستجوی خطی لیست پیوندی، بجای برگرداندن آدرس گره، گره‌ها را شماره‌گذاری کند و شماره گره پیدا شده را برگرداند.

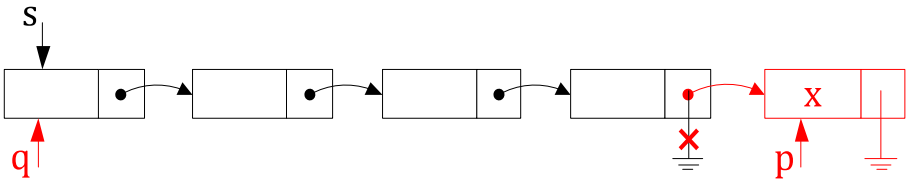
اضافه کردن به لیست پیوندی: به سه موقعیت از لیست می‌توان گره را اضافه کرد: ۱- به ابتدای لیست پیوندی ۲- به انتهای لیست پیوندی ۳- به وسط لیست پیوندی

۱- اضافه کردن گره به ابتدای لیست:




```
void InsertFirst (node *s , int x)
{
        node *p = makenode (x);
        p → next = s;
        s = p;
}
```

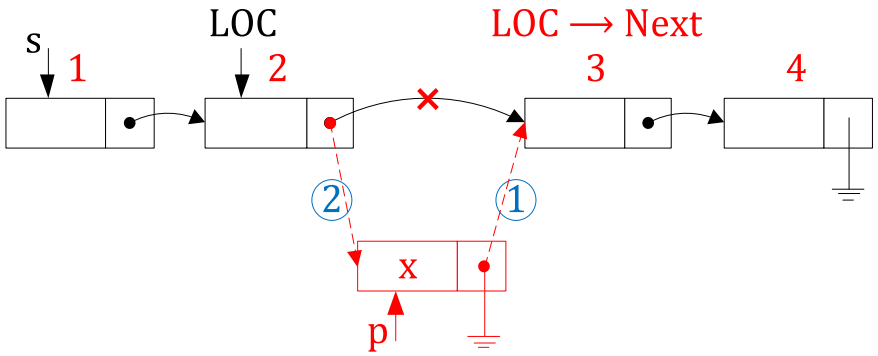
۲- اضافه کردن گره به انتهای لیست:




```
void InsertLast (node *s , int x)
{
        node *p = makenode (x);
        if (s == null)
                s = p;
        node *q = s;
        while (q → next != null)
                q = q → next;
        q → next = p;
}
```

۳- اضافه کردن گره به وسط لیست:

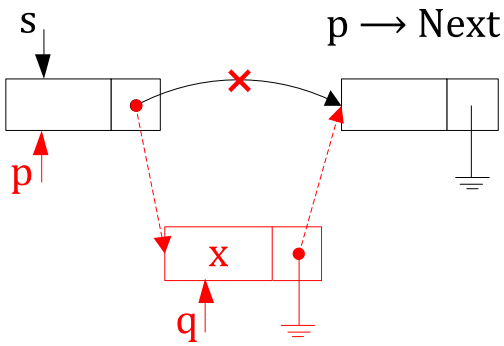
برای اضافه کردن گره به وسط لیست می‌توان به دو صورت عمل کرد. ۱- اضافه کردن گره بعد از یک گره با آدرس معلوم ۲- اضافه کردن گره بعد از یک گره با شماره معلوم



اضافه کردن گره بعد از یک گره با آدرس معلوم

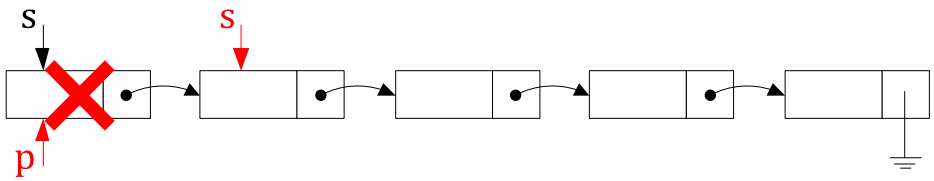
```
void InsertMiddelAddress (node *LOC , int x)
{
        node *p = makenode (x);
        if (LOC == null)
        {
                cout << "Invalid Insertion";
                return;
        }
    ① p → next = LOC → next;
    ② LOC → next = p;
}
```

اضافه کردن گره بعد از یک گره با شماره معلوم



حذف کردن از لیست پیوندی: از سه موقعیت لیست می‌توان گره را حذف کرد: ۱- از ابتدای لیست پیوندی ۲- از انتهای لیست پیوندی ۳- از وسط لیست پیوندی

۱- حذف کردن گره از ابتدای لیست:

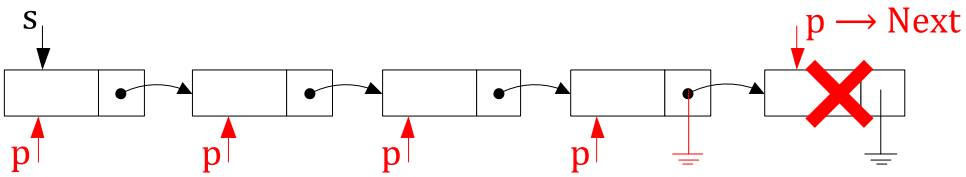



```
void DeleteFirst (node *s)
{
        if (s == null)
        {
                cout << " Invalid Deletion";
                return;
        }
        node *p = s;
        s = s → next;
        delete (p);
}
```

به صورت فیزیکی گره را از حافظه پاک می‌کند

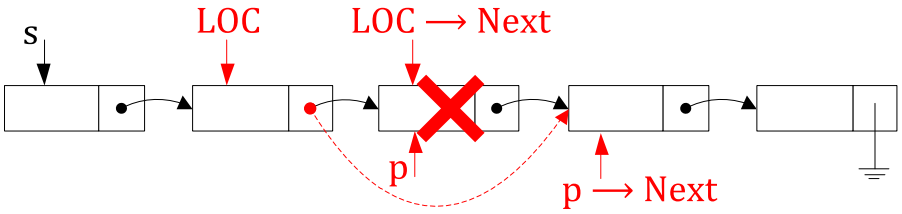
۲- حذف کردن گره از انتهای لیست:

در لیست‌های یک طرفه برای حذف یک گره باید آدرس گره قبل را داشته باشیم.




```
void DeleteLast (node *s)
{
        if (s == null)
        {
                cout << "Invalid Deletion";
                return;
        }
        if (s → next == null)
        {
                delete (s);
                s = null;
        }
        else
        {
                node *p = s;
                while (p → next → next != null)
                        p = p → next;
                delete (p → next);
                p → next = null;
        }
}
```

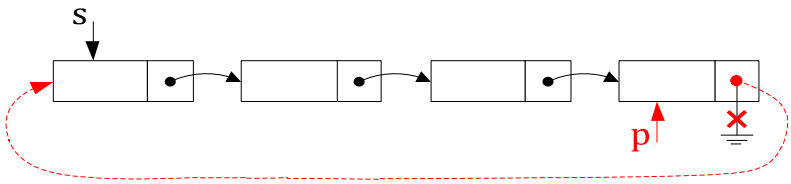
۳- حذف کردن گره از وسط لیست: به دو صورت می‌توان یک گره را از وسط لیست حذف کرد:
۱- شماره گره‌ای که قرار است حذف شود را در اختیار ما بگذارند. ۲- آدرس گره بعد از گره‌ای که قرار است حذف شود را در اختیار ما قرار دهند.



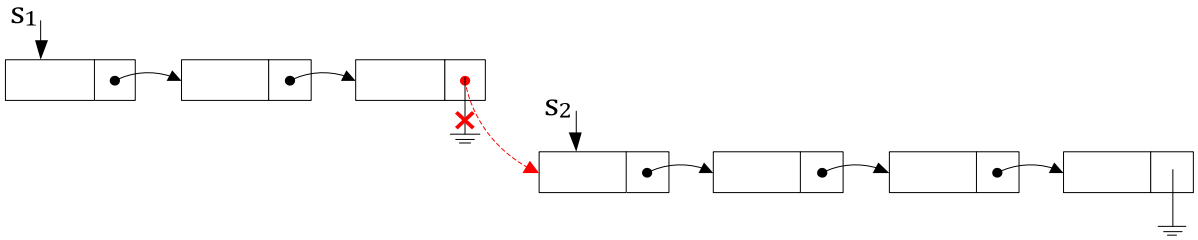

```
void DeleteMiddelAddress (node *LOC)
{
        if (LOC == null || LOC → next == null)
        {
                cout << " Invalid Deletion";
                return;
        }
        node *p = LOC → next;
        LOC → next = p → next;
```

```
    delete (p);
}
```

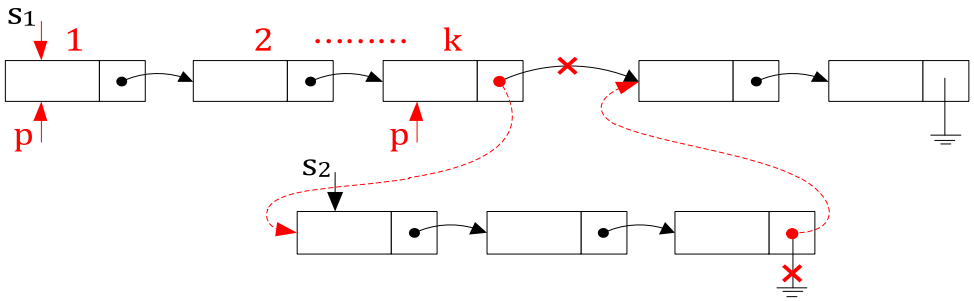
تمرین ۱۳) لیست زیر را حلقوی کنید.



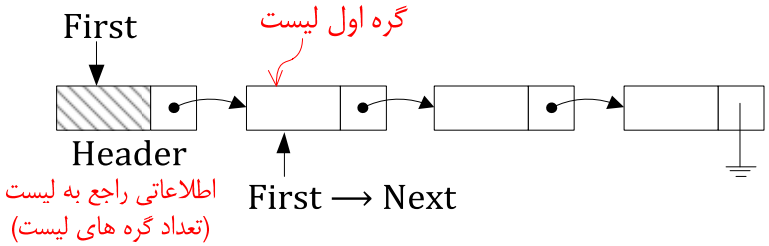
تمرین ۱۴) لیست s_2 را به انتهای s_1 بچسبانید.



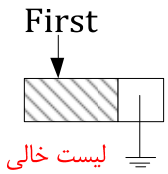
تمرین ۱۵) لیست s_2 را در لیست s_1 بعد از گره kام درج کنید.



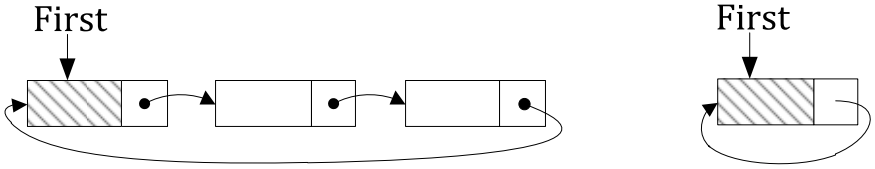
لیست پیوندی یک طرفه با سر لیست:



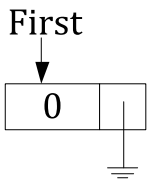
لیست وقتی خالی است که First → Next نال باشد. (First → Next == null)



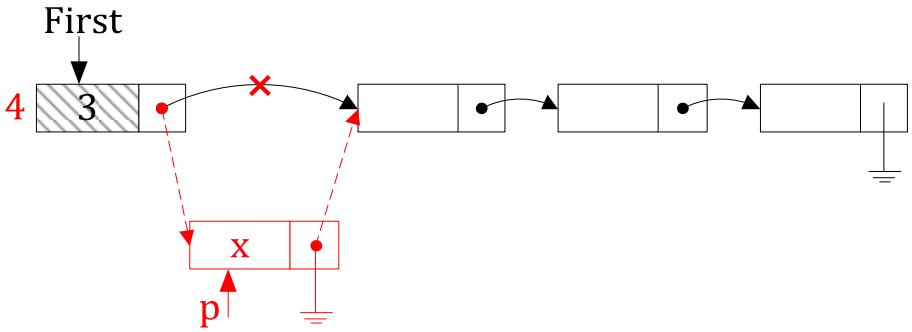
در لیست‌های حلقوی با سر لیست وقتی خالی است که First → Next با First مساوی باشد. (First → Next == First)



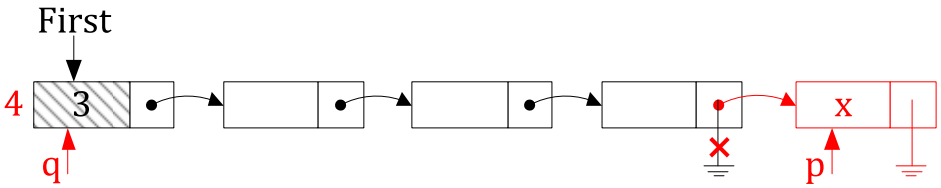
```
node *makeHeader ()
{
        node *p = new node;
        p → next = null;
        p → info = 0;
}
```



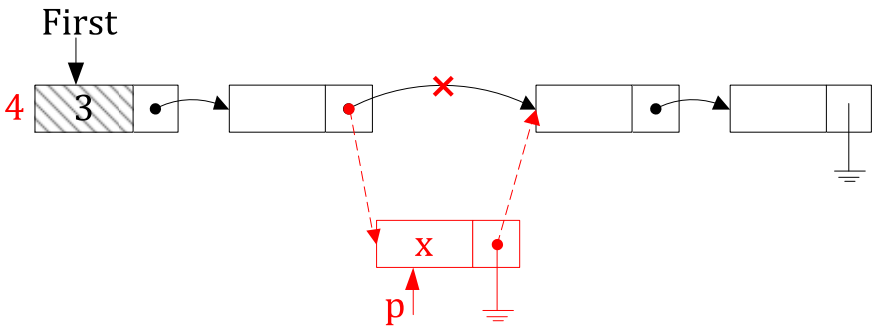
۱- اضافه کردن گره به ابتدای لیست: فرض بر این است که Header ساخته شده است.



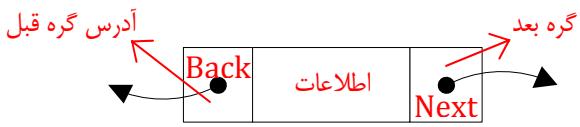
۲- اضافه کردن گره به انتهای لیست: فرض بر این است که Header ساخته شده است.

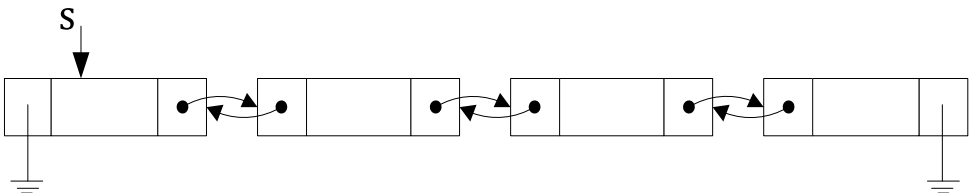


۳- اضافه کردن گره به وسط لیست: فرض بر این است که Header ساخته شده است.



لیست پیوندی دو طرفه: در لیست یک طرفه فقط می‌توانیم در یک جهت پیمایش کنیم. گاهی پیمایش روی لیست از هر دو طرف مورد نیاز است. بنابراین در هر گره به دو فیلد اشاره گر نیاز است؛ برای اشاره به گره بعدی و گره قبلی که اغلب اشاره‌گرهای Next و Back نامیده می‌شوند. لیستی که شامل این نوع گره باشد را لیست پیوندی دو طرفه (Doubly-Linked List) می‌نامند. شکل زیر ساختار گره را نشان می‌دهد.





ایجاد کردن یک گره لیست پیوندی دو طرفه در حافظه:

۱- تعریف ساختار گره:

```
struct binode
{
        int info;
        binode *next;
        binode *back;
};
```

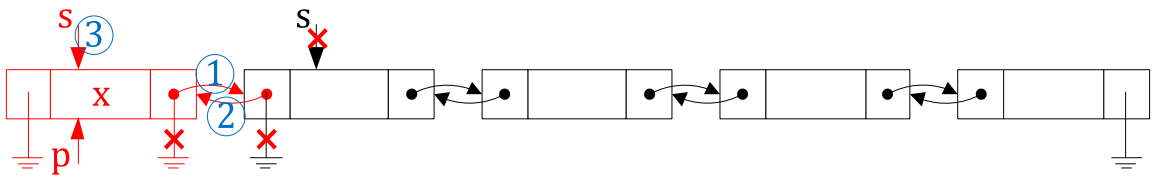
۲- تابع ایجاد گره:

```
binode *makebinode (int x)
{
      binode *p = new binode;
      p → info = x;
      p → next = null;
      p → back = null;
      return p;
}
```

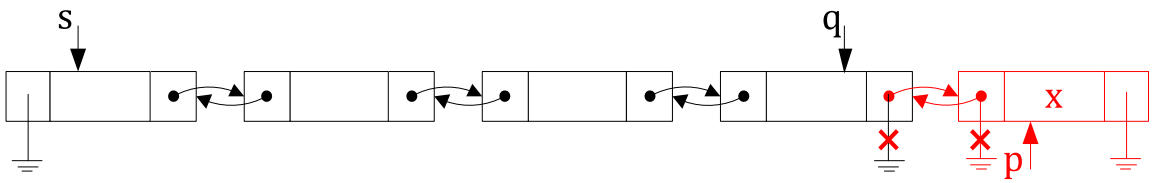
پیمایش روی لیست‌های دو طرفه همانند لیست‌های یک طرفه است و جستجوی خطی لیست‌های دو طرفه نیز همانند لیست‌های یک طرفه می‌باشد.

اضافه کردن به لیست پیوندی: به سه موقعیت از لیست می‌توان گره را اضافه کرد: ۱- به ابتدای لیست پیوندی ۲- به انتهای لیست پیوندی ۳- به وسط لیست پیوندی

۱- اضافه کردن گره به ابتدای لیست:



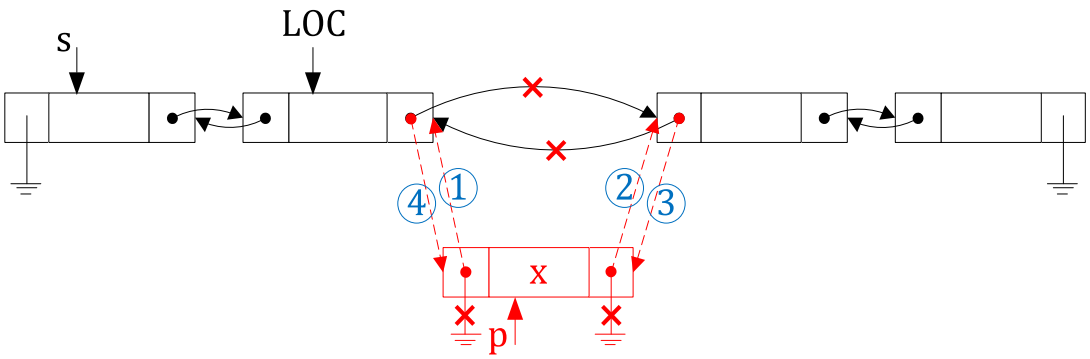
۲- اضافه کردن گره به انتهای لیست:




```
void InsertLast (binode *s , int x)
{
        binode *p = makebinode (x);
        if (s == null)
        {
                s = p;
                return;
        }
        binode *p = s;
        while (q → next != null)
```

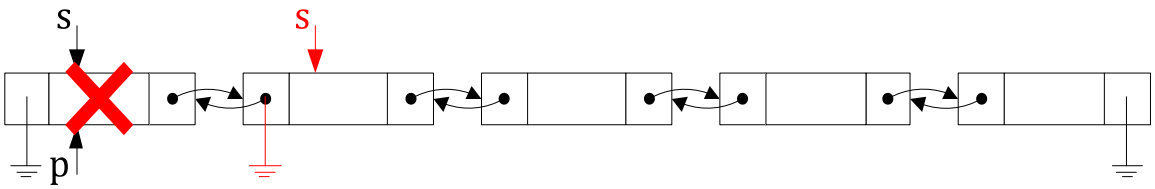
```
            q = q → next;
        q → next = p;
        p → back = q;
}
```

۳- اضافه کردن گره به وسط لیست:



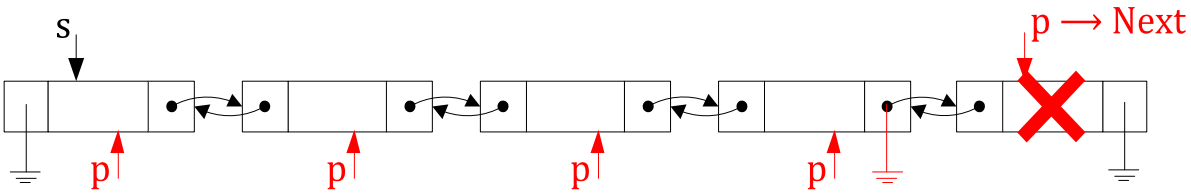
حذف کردن از لیست پیوندی: از سه موقعیت لیست می‌توان گره را حذف کرد: ۱- از ابتدای لیست پیوندی ۲- از انتهای لیست پیوندی ۳- از وسط لیست پیوندی

۱- حذف کردن گره از ابتدای لیست:

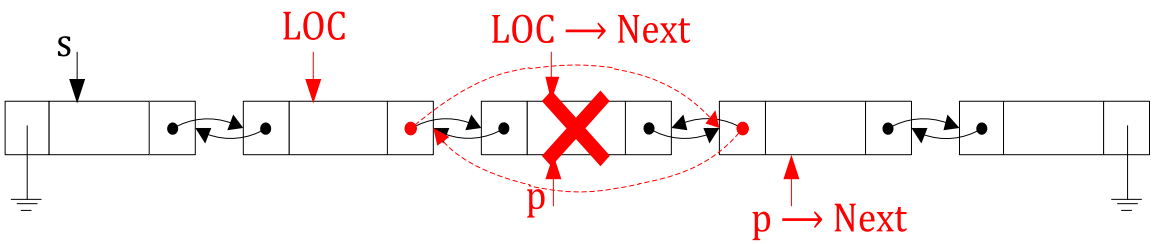


۲- حذف کردن گره از انتهای لیست:

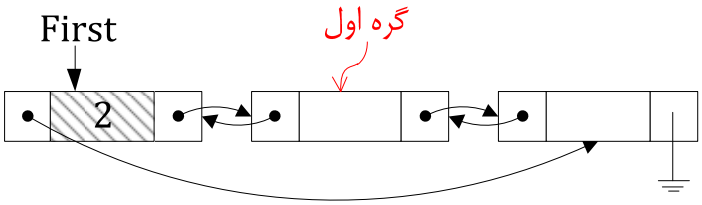
در لیست‌های یک طرفه برای حذف یک گره باید آدرس گره قبل را داشته باشیم.



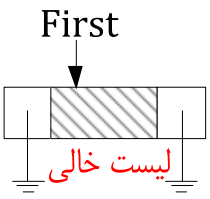
۳- حذف کردن گره از وسط لیست:



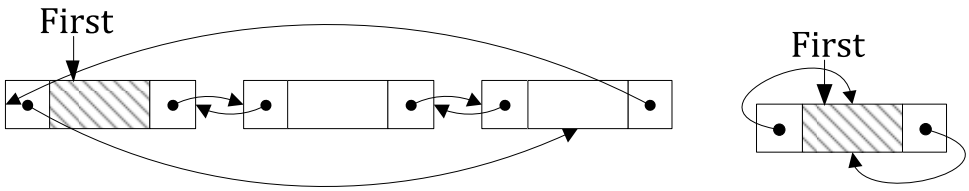
لیست پیوندی دو طرفه با سر لیست:



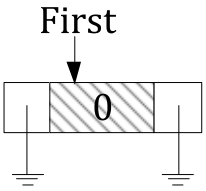
لیست وقتی خالی است که First → Next و First → Back نال باشد. (First → Back == null , First → Next == null)



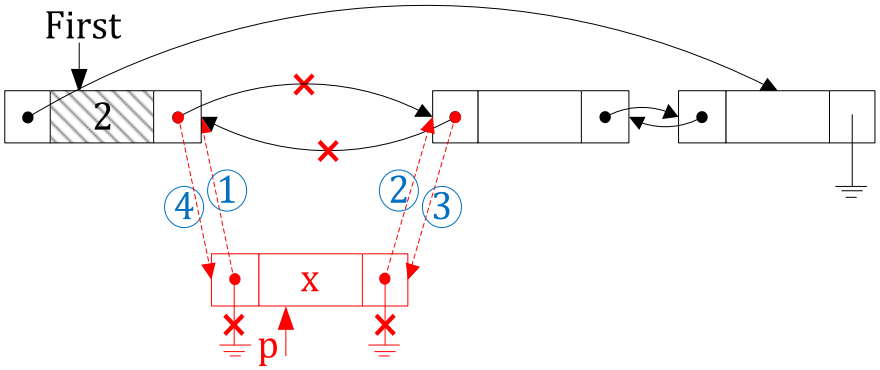
در لیست‌های دو طرفه حلقوی با سر لیست وقتی خالی است که First → Next با First مساوی باشد. (First → Back == First , First → Next == First)



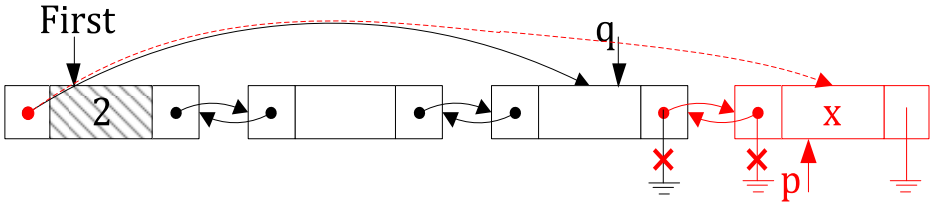

```
node *makeHeader ()
{
      node *p = new node;
      p → next = null;
      p → back = null;
      p → info = 0;
}
```



۱ - اضافه کردن گره به ابتدای لیست:

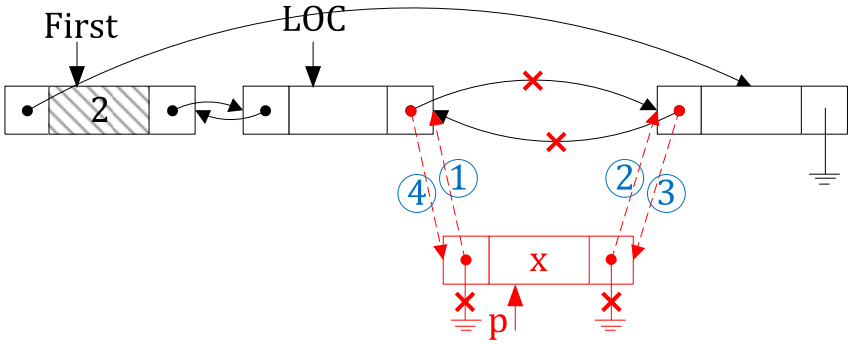


۲ - اضافه کردن گره به انتهای لیست:



```
void InsertLast (binode *first , int x)
{
        binode *p = makebinode (x);
        if (first → next == null)
        {
                first → next = p;
                first → back = p;
                p → back = first;
                first → info ++;
        }
        else
        {
                binode *q = first → next;
                while (q → next != null)
                        q = q → next;
                q → next = p;
                p → back = q;
                first → back = p;
                first → info ++;
        }
}
```

۳- اضافه کردن گره به وسط لیست:



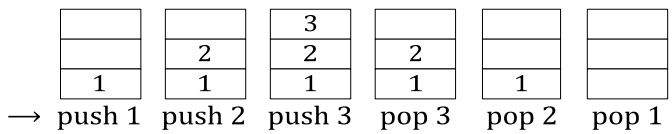
«Stack – پشته»

پشته: ساختمان داده‌ای است که عمل حذف و اضافه از یک انتهای آن به نام بالای پشته (Top) صورت می‌گیرد.

push: قرار دادن اطلاعات در بالای پشته

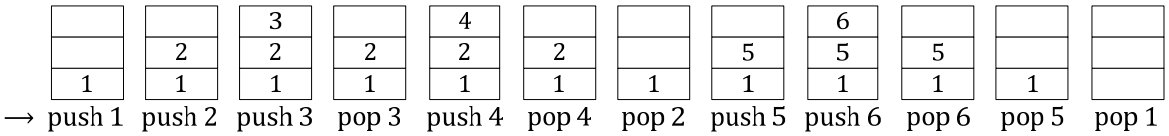
pop: برداشتن اطلاعات از بالای پشته

مثال۱۵) اعداد ۱، ۲ و ۳ را وارد یک پشته می‌کنیم و به ترتیب ۳، ۲ و ۱ آن‌ها را از پشته خارج می‌کنیم.



1 , 2 , 3 ⟶ 3 , 2 , 1

مثال۱۶) اعداد ۱، ۲، ۳، ۴، ۵ و ۶ را وارد یک پشته می‌کنیم و به ترتیب ۳، ۴، ۲، ۶، ۵ و ۱ آن‌ها را از پشته خارج می‌کنیم.



1 , 2 , 3 , 4 , 5 , 6 ⟶ 3 , 4 , 2 , 6 , 5 , 1

پیاده سازی پشته با آرایه:

Top: شامل اندیس خانه پر بالای پشته است.

پشته پر باشد: Top = n-1

پشته خالی باشد: Top = -1

Push کردن داخل پشته:

```
int top = -1;
void push (int stack [] , int n , int x)
{
        if (top == n-1)
        {
                cout << "Stack is Full";
                return;
        }
        top ++;
        stack [top] = x;
}
```

Pop کردن داخل پشته:

```
int pop (int stack [] , int n)
{
        if (top == -1)
        {
                cout << "Stack is Empty";
                c_exit ();
        }
        return stack [top --];
}
```

تمرین۱۶) عمل Push و Pop زمانی که Top به اندیس خالی بالای پشته اشاره می‌کند.

پشته پر باشد: Top = n

پشته خالی باشد: Top = 0

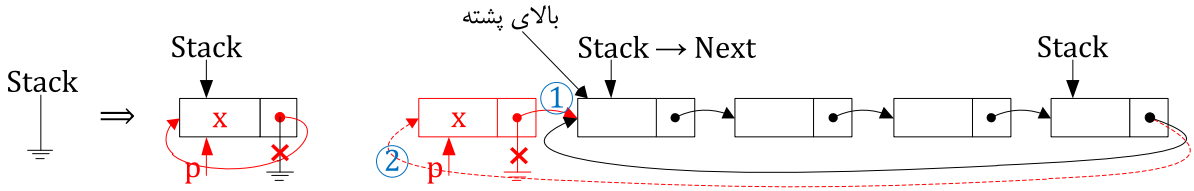
پیاده سازی پشته با لیست‌های پیوندی:

یکی از معایب پیاده‌سازی پشته به کمک آرایه این است که اندازه ثابت آرایه، اندازه پشته را محدود می‌کند. استفاده از لیست پیوندی به جای آرایه جهت نمایش پشته، بدون محدودیت (مگر محدودیت حافظه) رشد می‌کند و بدون هدر دادن حافظه کوچک می‌شود.

می‌دانید که پشته لیستی از عناصر است که فقط از یک طرف به نام بالای پشته قابل دستیابی هستند. بنابراین، می‌توان پشته را با لیست پیوندی پیاده‌سازی کرد، به طوری که فقط اولین گره لیست پیوندی مستقیماً قابل دستیابی است.

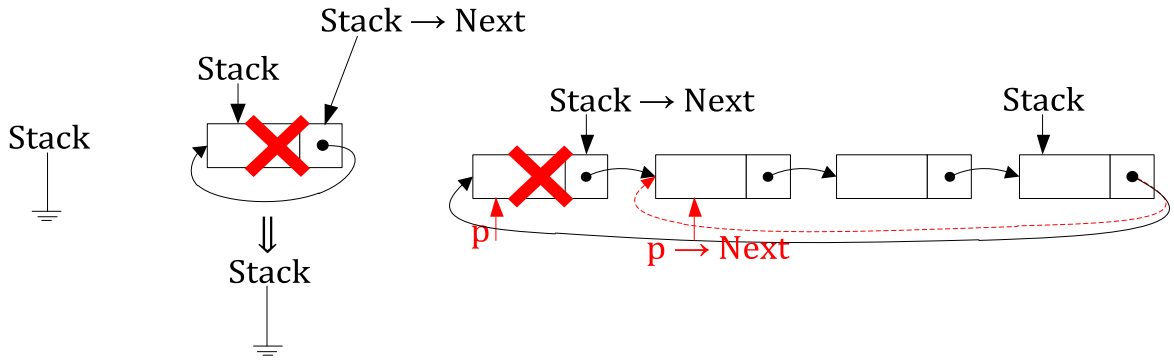
ساختار گره پشته پیوندی همانند گره لیست پیوندی است. برای دستیابی به عناصر پشته پیوندی فقط به یک اشاره‌گر نیاز داریم که به ابتدای لیست پیوندی اشاره نماید.

Push کردن داخل پشته:




```
void push (node *stack , int x)
{
        Node *p = makenode (x);
        if (stack == null)
        {
                stack = p;
                stack → next = stack;
        }
        else
        {
                p → next = stack → next;
                stack → next = p;
        }
}
```

Pop کردن داخل پشته:




```
int pop (node *stack)
{
        if (stack == null)
        {
                cout << "Stack is Empty";
                c_exit ();
        }
        int x = stack → next → info;
        if (stack → next == stack)
        {
                delete (stack);
                stack = null;
```

```
        }
        else
        {
                node *p = stack → next;
                stack → next = p → next;
                delete (p);
        }
        return x;
}
```

کاربرد پشته در ارزیابی عبارات:

یکی از کاربردهای مهم پشته، ارزیابی عبارات می‌باشد. عبارات به سه شکل نوشته می‌شوند. به عنوان مثال، عبارت زیر را برای جمع دو متغیر A و B در نظر بگیرید:

عبارت میانوندی (Infix): عملگر وسط عملوندهای خود قرار دارد. A+B

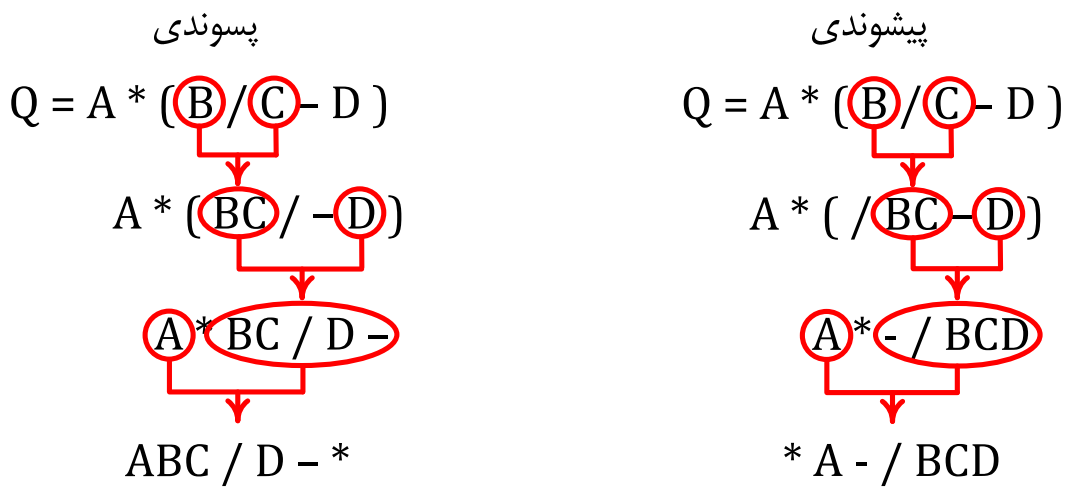
عبارت پسوندی (Postfix): عملوند در سمت راست عملوندهای خود قرار دارد. AB+

عبارت پیشوندی (Prefix): عملوند در سمت چپ عملوندهای خود قرار دارد. +AB

تقدم عملگرها:

۱- ( ) ۲- توان ^ ۳- ٪ / * ۴- + -

مثال۱۷) برای عبارت میانوندی زیر عبارت پسوندی و پیشوندی را بنویسید.



الگوریتم ارزیابی عبارات پسوندی با استفاده از پشته:

۱- انتهای یک عبارت پسوندی یک پرانتز بسته قرار دهید.

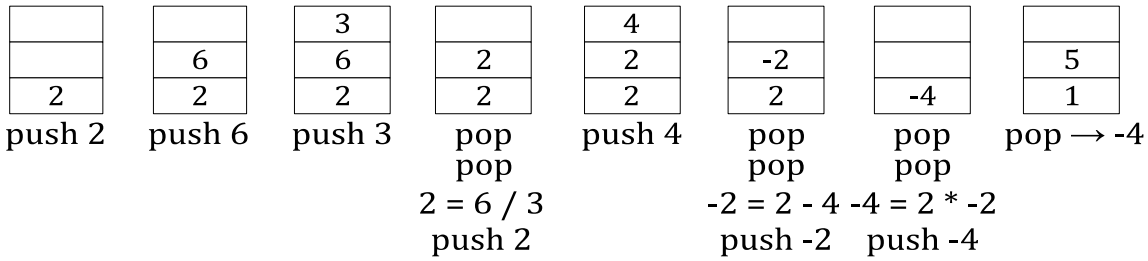
۲- از چپ به راست روی عبارت پسوندی حرکت کنید؛ اگر عملوند دیده شد در پشته push کنید؛ اگر عملگر دیده شد دو بار از پشته pop کنید؛ حاصل عبارت زیر را محاسبه کنید:

x = pop دوم عملگر pop اول

۳- مرحله دوم را تکرار کنید تا اینکه به پرانتز بسته در عبارت پسوندی برسید. در این حالت یک pop انجام دهید، مقدار بدست آمده حاصل عبارت پسوندی است. پشته نیز باید خالی باشد.

مثال۱۸) حاصل عبارت پسوندی زیر را با استفاده از پشته بدست آورید.

P = ABC / D - * P = 263 / 4 - *)



الگوریتم تبدیل یک عبارات میانوندی به پسوندی:

۱- انتهای عبارت میانوندی یک پرانتز بسته قرار دهید و در پشته نیز یک پرانتز باز Push کنید.

۲- از چپ به راست روی عبارت میانوندی حرکت کنید. اگر عملوند دیده شد، در عبارت پسوندی بنویسید؛ اگر پرانتز باز دیده شد، در پشته Push کنید؛ اگر عملگر دیده شد ابتدا بالای پشته را بررسی کنید، در صورتی که عملگر بالای پشته تقدم آن از عملگر مذکور بیشتر بود عملگر بالای پشته را Pop کرده و در عبارت پسوندی بنویسید و سپس عملگر مذکور را Push کنید. (در پشته نباید عملگری با تقدم کمتر مستقیماً روی عملگر با تقدم بالاتر قرار گیرد.) اگر پرانتز بسته دیده شد، از پشته Pop کنید و در عبارت پسوندی بنویسید. این عمل را تکرار کنید تا به پرانتز باز در پشته برسید؛ پرانتز باز را نیز پاک کنید ولی در عبارت پسوندی ننویسید.

۳- مرحله دوم را تکرار کنید تا اینکه پشته خالی شود. در این حالت عبارت پسوندی بدست آمده عبارت مطلوب است.

مثال۱۹) عبارت میانوندی Q = (A + B) * (C / D – E) ^ F را با استفاده از پشته به عبارت پسوندی تبدیل کنید.

Q = (A + B) * (C / D – E) ^ F)

Q = Infix	Stack ⟶ رشد	Postfix
	(	
(	((	
A	((	A
+	((+	A
B	((+	AB
)	(	AB +
*	(*	AB +
(	(* (	AB +
C	(* (	AB + C
/	(* (/	AB + C
D	(* (/	AB + CD
–	(* (–	AB + CD /
E	(* (–	AB + CD / E
)	(*	AB + CD / E –
^	(* ^	AB + CD / E –
F	(* ^	AB + CD / E – F
)	خالی	AB + CD / E – F ^ *

تمرین۱۷) الگوریتم و برنامه‌ای بنویسید که با استفاده از پشته پرانتزهای یک عبارت ریاضی را بررسی کند.

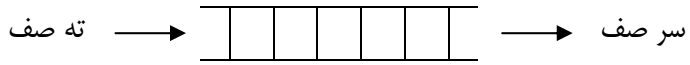
پروژه درسی) برنامه‌ای بنویسید که حاصل یک عبارت ریاضی را محاسبه کند.

عبارت ریاضی ← (الگوریتم تبدیل: میانوندی به پسوندی) ← عبارت پسوندی ← (الگوریتم ارزیابی: عبارت پسوندی) ← حاصل

«صف – Queue»

صف: ساختمان داده‌ای است که عمل حذف از یک انتهای آن به نام سر صف (Front) و عمل اضافه کردن از انتهای دیگر آن به نام ته صف (Rear) صورت می‌گیرد.

اولین عنصری که وارد صف می‌شود اولین عنصری است که از صف خارج می‌شود. بنابراین عناصر به همان ترتیبی که به صف اضافه می‌شوند از آن حذف می‌شوند. به همین دلیل به صف لیست (first in, first out) FIFO نیز گفته می‌شود.



FIFO

First In, First Out

پیاده سازی صف با آرایه:

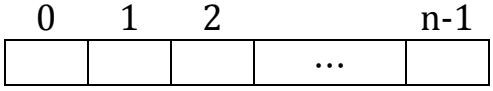
صف را می‌توان توسط یک آرایه یک بعدی پیاده‌سازی کرد. به دو متغیر Front و Rear برای مشخص کردن ابتدا و انتهای صف نیاز است.

هر گاه عنصری به صف اضافه شود Rear یک گام به جلو حرکت می‌کند و هر گاه که عنصری را از صف حذف می‌شود Front یک واحد افزایش می‌یابد.

چون اندازه آرایه از قبل تعریف می‌شود، هنگام اضافه کردن عنصری به صف ابتدا باید اطمینان حاصل کرد که هنوز ظرفیت پذیرش داده را دارد. اگر Rear برابر با ظرفیت کل آرایه شود صف پر در نظر گرفته می‌شود.

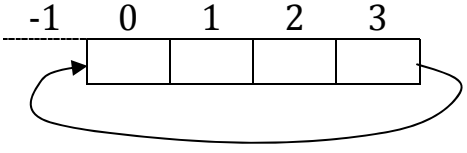
اگر ابتدا و انتهای صف برابر بودند (Front = Rear) یعنی صف خالی است. عمل حذف روی صف خالی انجام نمی‌گیرد.

طول صف یا تعداد عناصر موجود در صف برابر با Rear-Front+1 است.



f (Front): شامل اندیس خانه خالی در صف است.

r (Rear): شامل اندیس خانه پر ته صف است.



صف پر است: f == (r + 1) % n

صف خالی است: f == r

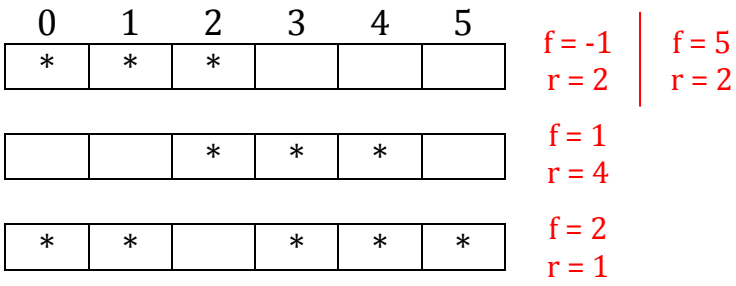
n = تعداد خانه‌های آرایه

برای حذف کردن: f = (f + 1) % n

f	r	
-1	-1	اضافه
-1	0	اضافه
-1	1	حذف
0	1	اضافه
0	2	اضافه
0	3	حذف
1	0	اضافه

برای اضافه کردن: r = (r + 1) % n

مثال ۲۰) حالت‌های f و r را برای صف‌های زیر مشخص کنید.



الگوریتم اضافه کردن به صف:

```
int f = r = -1;
void InsertQueue (int Q[] ; int n ; int x)
{
        if ((f == (r + 1) % n) || (f == -1 && r == n-1))
        {
                cout << "Queue is Full";
                return;
        }
        r = (r + 1) % n;
        Q[r] = x;
}
```

الگوریتم حذف کردن از صف:

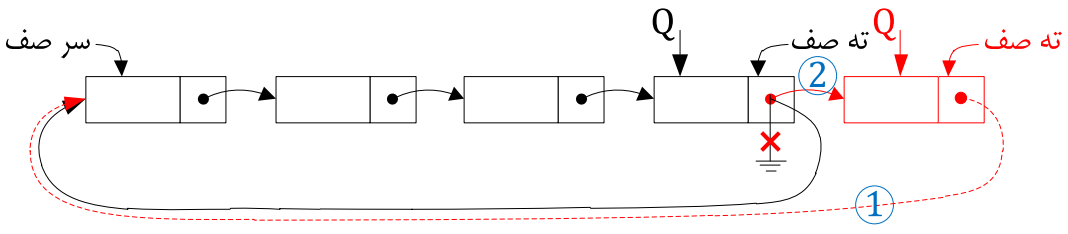
```
int RemoveQueue (int Q[] ; int n)
{
        if (f == r)
        {
                cout << "Queue is Empty";
                c_exit ();
        }
        f = (f + 1) % n;
        return Q[f];
}
```

تمرین۱۸) شرایط را طوری تغییر دهید که حداکثر تعداد آیتم‌ها در صف = n باشد.

پیاده سازی پشته با لیست‌های پیوندی:

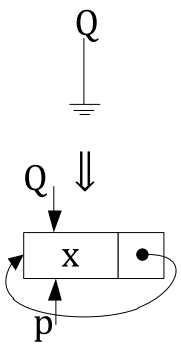
در یک لیست پیوندی اگر درج در انتها و حذف از ابتدای آن انجام گیرد یک صف اجرا شده است. مزیت پیاده‌سازی صف توسط لیست پیوندی در این است که طول صف تنها محدود به حافظه در دسترس است.

الگوریتم اضافه کردن به صف:




```
void InsertQueue (node *Q , int x)
```

```
{
        node *p = makenode (x);
        if (Q == null)
        {
                Q = p;
                Q → next = Q;
        }
        else
        {
                p → next = Q → next;
                Q → next = p;
                Q = p;
        }
}
```



الگوریتم حذف کردن از صف:

الگوریتم حذف از صف همان تابع pop در پشته پیاده‌سازی شده با لیست پیوندی می‌باشد.

```
int DeleteQueue (node *Q)
{
        if (f == r)
        {
                cout << "Queue is Empty";
                c_exit ();
        }
        int x = Q → next → info;
        if (Q → next == Q)
        {
                delete (Q);
                Q = null;
        }
        else
        {
                node *p = Q → next;
                Q → next = p → next;
                delete (p);
        }
        return x;
}
```

کویز ۱) دستوراتی بنویسید که گره kام را با گره k+1ام از یک لیست پیوندی یک طرفه با آدرس شروع s جابجا کند.
(با رسم شکل)

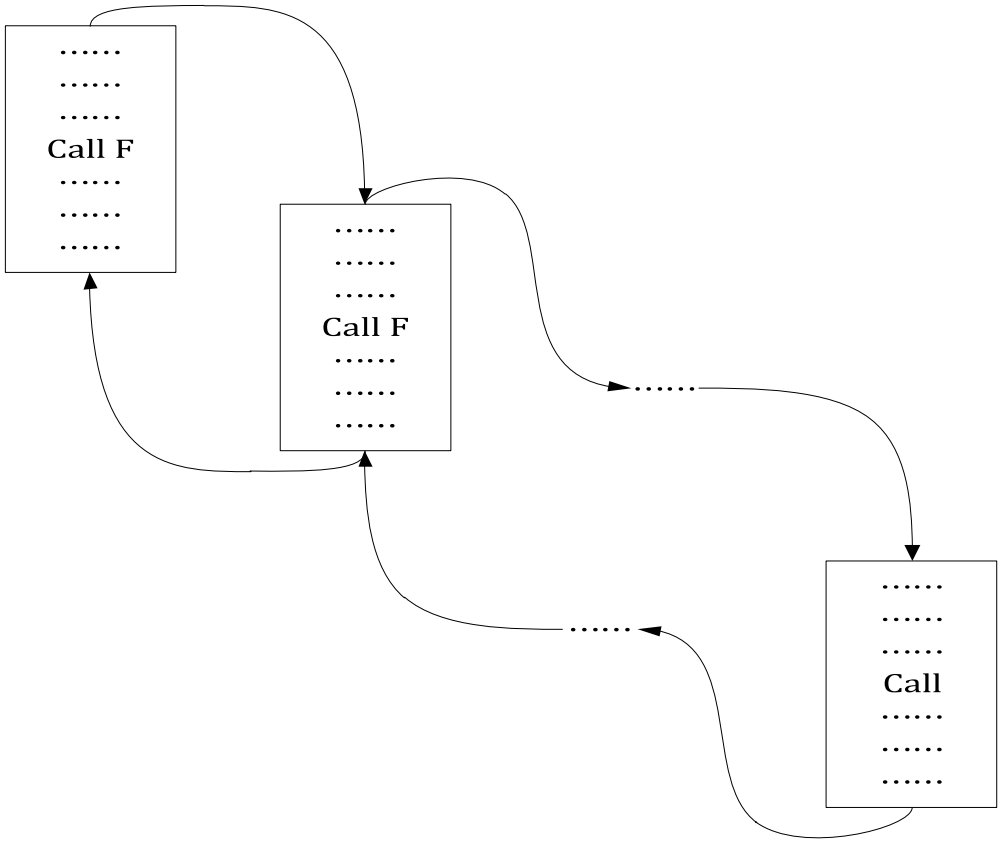
کویز ۲) عبارت پسوندی را برای عبارت زیر بدست آورید.

Q = (A / B) ^ (C – D * E) - F

«بازگشت‌پذیری – Recursive»

بازگشتی: بازگشتی اجازه بیان راه حل یک مسئله را به طور مختصر و مفید می‌دهد. مسئله‌ای که به صورت بازگشتی حل می‌شود باید بتواند به مسائل کوچک‌تر تقسیم بشود و حل مسائل کوچک به همان روش مسئله بزرگ قابل انجام باشد. مسئله کوچک‌تر به مسئله کوچک‌تری شکسته می‌شود تا سرانجام به کوچک‌ترین اندازه مسئله برسد که base case نامیده می‌شود که می‌تواند بدون استفاده از بازگشتی حل شود.

تابع بازگشتی: تابع بازگشتی تابعی است که در بدنه‌اش دستوری دارد که خودش را فراخوانی می‌کند. توابع بازگشتی برای نگهداری حالت قبلی خود از پشته مکرر استفاده می‌کنند.



مزایا و معایب توابع بازگشتی:

مزایا:

تابع کوچک می‌شود.

معایب:

توابع بازگشتی کند هستند.

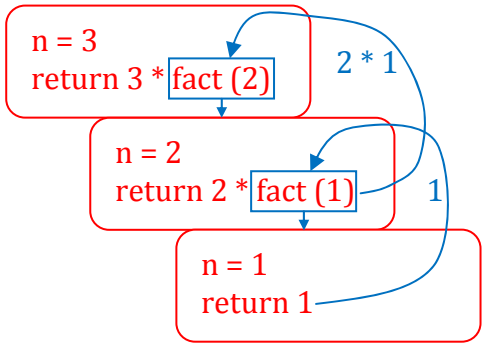
اشکال زدایی توابع بازگشتی مشکل است.

تابع فاکتوریل به صورت معمولی:

```
int fact (int n)
{
        int f = 1;
        for (int i = 1 ; i <= n ; i++)
                f *= i;
        return i;
}
```

تابع فاکتوریل به صورت بازگشتی:

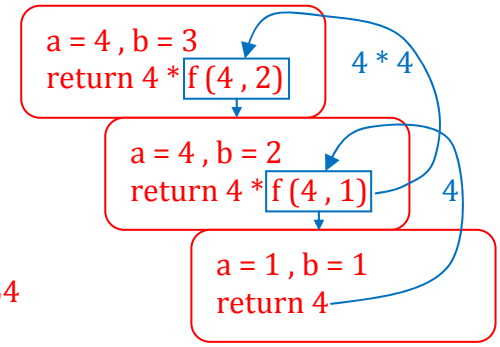
```
int fact (int n)
{
        if (n == 1)
                return 1;
        return n * fact (n-1);
}
```



3 * 2 * 1 = 6

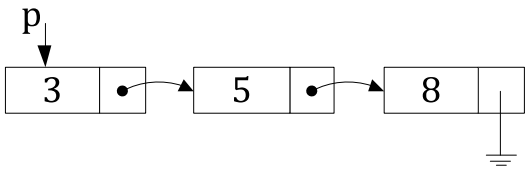
مثال ۲۱) تابع زیر چه عملی را انجام می‌دهد.

```
int f (int a , int b)
{
        if (b == 1)
                return a;
        return a * f (a , b-1);
}
```



4 * 4 * 4 = 64

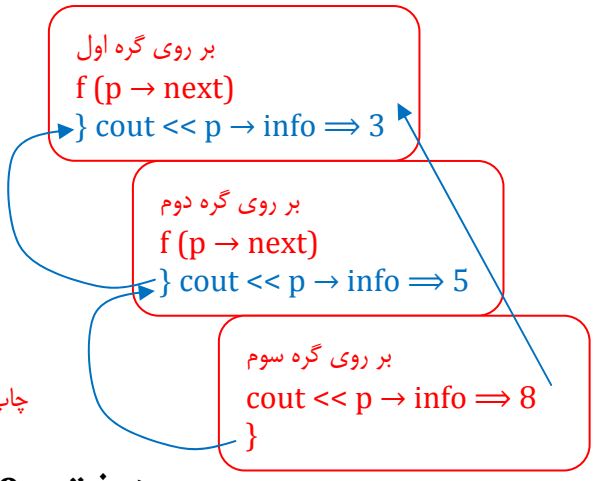
مثال ۲۲) تابع زیر چه عملی را انجام می‌دهد.




```
int f (node *p)
{
        if (p → next != null)
                f (p → next);
        cout << p → info;
}
```

چاپ می‌شوند (از چپ به راست)

8 5 3



«Tree – درخت»

درخت: ساختمان داده درخت برای نمایش داده‌های سلسله مراتبی به کار می‌رود و چون شبیه درخت رسم می‌شود ساختار درختی نام‌گذاری شده است. البته ساختار درختی در مقایسه با درخت واقعی معمولاً به صورت وارونه رسم می‌شود، یعنی ریشه درخت در بالا و برگ‌های آن در پایین قرار می‌گیرند.

به طور کلی یک درخت مجموعه‌ای از گره‌هاست که از طریق پیوندهایی با هم در رابطه هستند. هر گره دارای داده مرتبط و مجموعه‌ای از گره‌های دیگر است.

اصطلاحات مربوط به درخت‌ها:

گره: داده‌ها در درخت در ساختاری به نام گره (node) قرار دارند. هر گره حاوی اطلاعات و پیوندهایی به دیگر گره‌های درخت است.

یال: اتصال دو گره متوالی (اتصال پدر و فرزند) را یال گویند.

مسیر: اتصال چند یال متوالی را مسیر گویند.

شاخه: خطوطی که گره‌ها را در درخت به هم متصل می‌کنند شاخه (branche) نامیده می‌شوند.

والد و فرزند: گره‌ای که بلافاصله زیر یک گره قرار می‌گیرد فرزند (children) آن گره محسوب می‌شود. یک گره والد گره دیگر (parent) است اگر بلافاصله بالاتر از آن نزدیک‌تر به ریشه قرار داشته باشد.

گره‌ای که کلیه گره‌های سطوح پایین را به هم متصل می‌کند جد (ancestor) نامیده می‌شود.

ریشه: هر درخت گره خاصی به نام ریشه (root) دارد که کلیه گره‌های دیگر درخت در پایین آن قرار دارند. گره ریشه والدی ندارد. هر درخت تنها شامل یک گره ریشه است.

گره‌های همزاد: گره‌های همزاد (Sibling) گره‌هایی هستند که والد یکسانی دارند. به عبارت دیگر فرزندان یک گره با هم همزاد هستند.

درجه گره: تعداد فرزندان یک گره درجه (degree) آن گره نامیده می‌شود.

درجه درخت: درجه درخت برابر ماکزیمم درجه گره‌ها در درخت است.

برگ: گره‌های بدون فرزند گره‌های پایانی (end-nodes) یا برگ (leaf) نامیده می‌شوند. درجه گره‌های برگ صفر است.

سطح: مجموعه گره‌هایی طول مسیر آن‌ها تا ریشه یکسان است را سطح درخت (level) می‌نامند. اگر ریشه را در سطح یک فرض کنیم بر حسب اینکه یک گره نسبت به ریشه در چه ردیفی باشد شماره سطح می‌گیرد.

ارتفاع درخت: ارتفاع (height) درخت برابر با بیشترین سطح گره‌ها در درخت یا سطح دورترین برگ است. ارتفاع درختی که تنها گره ریشه را دارد صفر است.

هر درخت خواص زیر را نمایش می‌دهند:

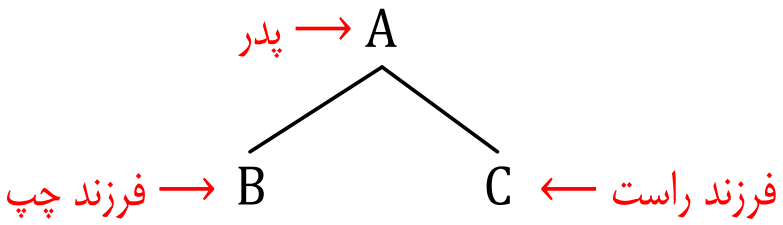
- دقیقاً یک ریشه دارد.
- همه گره‌ها بجز ریشه دقیقاً یک والد دارند.
- تنها یک مسیر از بین هر دو گره وجود دارد.
- دور وجود ندارد یعنی مسیری وجود ندارد که از یک گره شروع شود و به خود آن ختم شود.
- درختی که دارای n گره است n-1 شاخه دارد.

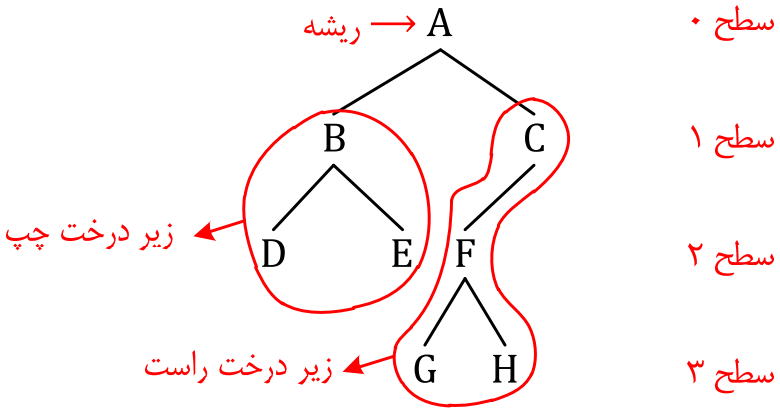
درخت دودویی: درخت دودویی (Binary Tree)، مجموعه محدودی از گره‌ها است که: ***الف)*** خالی است (درخت دودویی خالی) یا ***ب)*** حاوی گره خاصی به نام ریشه است و بقیه گره‌های آن، دو زیر درخت دودویی مجزا به نام‌های زیر درخت چپ و زیر درخت راست را تشکیل می‌دهند. در مورد تفاوت‌های بین درخت دودویی و درخت عمومی به نکات زیر توجه کنید:

۱- درخت دودویی می‌تواند تهی باشد، ولی درخت عمومی نمی‌تواند تهی باشد.

۲- در درخت دودویی، هر گره حداکثر دو فرزند دارد.

۳- در درخت دودویی ترتیب فرزندان هر گره مهم است، در حالی که در درخت عمومی این طور نیست.





درخت‌های دودویی کاربردهای فراوانی در کامپیوتر دارند. بنابراین، درخت دودویی را می‌توان درختی با درجه ۲ در نظر گرفت، اما ترتیب فرزندان در درخت دودویی مهم است. اگر درخت دودویی در سطح i دارای m گره باشد، در سطح $i+1$ حداکثر دارای $2m$ گره است. چون درخت دودویی در سطح صفر حداکثر یک گره دارد (ریشه)، در سطح ۱ حداکثر دارای دو گره است و در سطح ۲ حداکثر دارای ۴ گره است ($۲^۲$) و در سطح i حداکثر 2^i گره دارد.

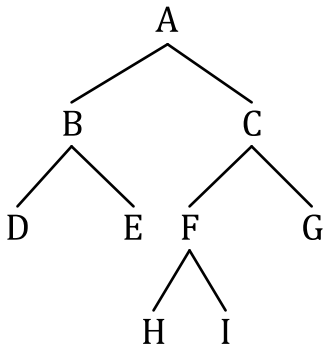
انواع درخت دودویی:

۱- **درخت دودویی پُر (محض) (Full Binary Tree):** یک درخت دودویی پُر، درختی است که هر گره آن صفر یا دو فرزند دارد. در درخت دودویی پر کلیه برگ‌ها در یک سطح هستند. یک درخت دودویی پر به ارتفاع h دارای 2^{h-1} گره است. عمق یک درخت دودویی پر با n گره $\log_2^n+1$ است. تعداد گره‌های سطح iام یک درخت دودویی پر 2^{i-1} است.

۲- **درخت دودویی کامل (Complete Binary Tree):** یک درخت دودویی کامل، درختی است که همه سطوح آن به جز احتمالاً آخرین سطح، حداکثر گره‌ها را دارند و در سطح آخر گره‌ها از سمت چپ ظاهر می‌شوند. یعنی اگر ارتفاع درخت h باشد تعداد کل گره‌ها تا سطح $h-1$ برابر با 2^h-1 است و در سطح آخر اگر گره‌هایی وجود دارند باید از چپ به راست اضافه شوند. (T_n)

۳- **درخت دودویی میزان (Balanced Binary Tree):** درخت دودویی میزان، درختی است که کلیه برگ‌ها حداکثر یک سطح با هم تفاوت دارند.

درخت دودویی کامل:



T_9

ارتفاع درخت T_n:

$$h(T_n)=\lfloor \log_2^n \rfloor +1$$

مثال۲۳) درختی با ۹ گره و دیگری با ۲۵ گره چقدر ارتفاع دارند؟

$$h(T_9)=\lfloor \log_2^9 \rfloor +1 \Rightarrow h(T_9)=4$$

$h(T_{25}) = \lfloor \log_2^{25} \rfloor + 1 \Rightarrow h(T_{25}) = 5$

حداکثر تعداد گره‌ها یک درخت کامل با ارتفاع h برابر با 2^h-1 است.

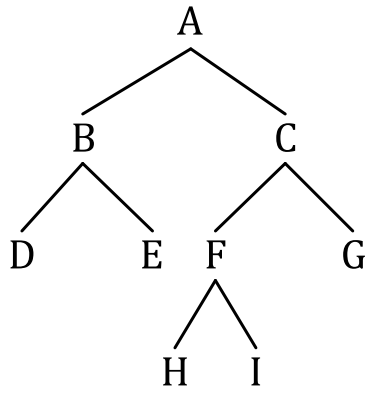
حداقل تعداد گره‌ها یک درخت کامل با ارتفاع h برابر با 2^{h-1} است.

پیاده سازی درخت‌های دودویی با آرایه:

یک درخت دودویی کامل با n گره را می‌توان در یک آرایه یک بعدی ذخیره کرد. برای این کار گره‌های درخت شماره‌گذاری می‌شوند. شماره‌گذاری به ترتیب از بالا به پایین و از چپ به راست انجام می‌شود و به هر گره شماره‌ای تعلق می‌گیرد. سپس گره با شماره i در خانه iام آرایه قرار می‌گیرد.

وقتی درخت دودویی در آرایه نمایش داده می‌شود برای هر گره با اندیس i:

- ریشه درخت در node [1] قرار دارد.
- اگر $i \neq 1$ باشد، والد i در $i / 2$ است و اگر $i = 1$ باشد i ریشه است.
- اگر $2i \leq n$ باشد فرزند چپ i در $2i$ است وگرنه این گره فرزند چپ ندارد.
- اگر $2i+1 \leq n$ باشد، فرزند راست i در $2i+1$ است وگرنه فاقد فرزند راست است.



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	B	C	D	E	F	G					H	I		

روابطی که برای دسترسی به فرزند چپ، فرزند راست و پدر داریم عبارتند از:

اگر اندیس گره n برابر k باشد: در صورتی که اندیس آرایه از صفر شروع شود.

f (m , k)

اندیس شروع آرایه — اندیس گره n

اندیس فرزند چپ n: $2k + 1$

اندیس فرزند راست n: $2k + 2$

اندیس فرزند پدر n: $\left\lfloor \frac{k-1}{2} \right\rfloor$

اگر اندیس گره n برابر k باشد: در صورتی که اندیس آرایه از یک شروع شود.

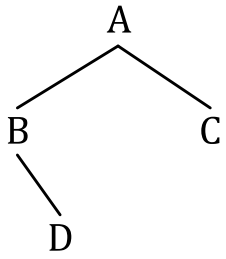
اندیس فرزند چپ n: $2k$

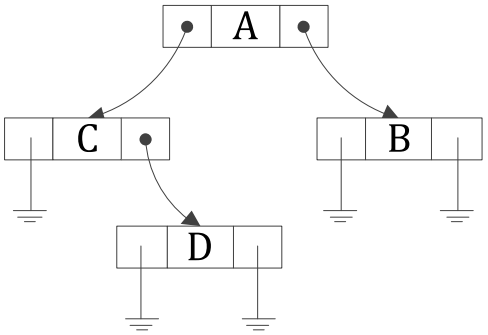
اندیس فرزند راست n: $2k + 1$

اندیس فرزند پدر n: $\left\lfloor \frac{k}{2} \right\rfloor$

پیاده سازی درخت‌های دودویی با لیست‌های پیوندی:

یک درخت دودویی را می‌توان توسط لیست پیوندی نمایش داد. به این صورت که برای هر گره درخت یک گره در لیست در نظر گرفته می‌شود. هر گره یک اشاره‌گر به فرزند چپ و یک اشاره‌گر به فرزند راست دارد.





ایجاد کردن یک گره درخت در حافظه:

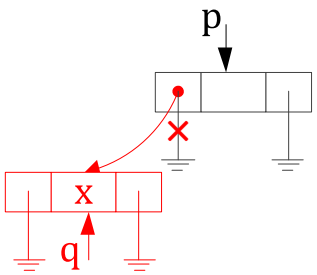
۱- تعریف ساختار گره:

```
struct tree
{
        int info;
        binode *left;
        binode *right;
};
```

۲- تابع ایجاد گره:

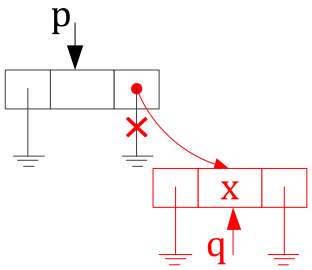
```
binode *maketree (int x)
{
      tree *p = new tree;
      p → info = x;
      p → left = null;
      p → right = null;
      return p;
}
```

اضافه کردن گره به سمت چپ درخت:

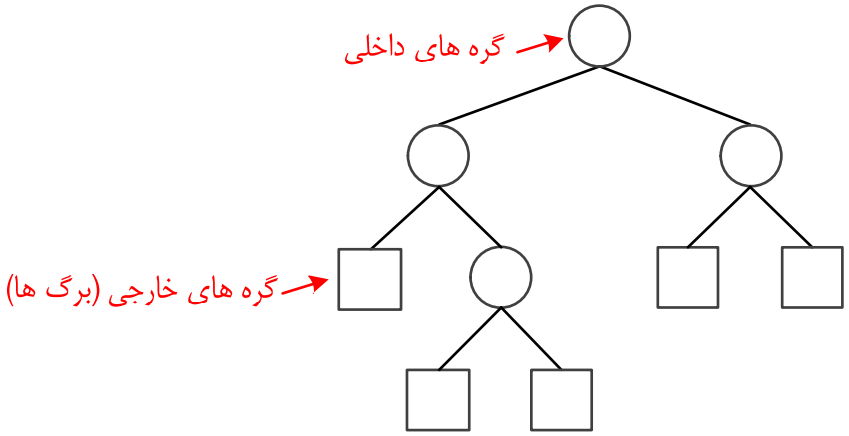



```
void SetLeft (tree *p , int x)
{
        if (p → left != null)
        {
                cout << "Invalid";
                return;
        }
        tree *q = maketree (x);
        p → left = q;
}
```

اضافه کردن گره به سمت راست درخت:



درخت دودویی محض (پُر):



تعداد کل گره‌ها $N = 2 N_I + 1$ تعداد گره‌های داخلی

یا

تعداد کل گره‌ها $N = 2 N_E + 1$ تعداد گره‌های خارجی

پیمایش درخت دودویی:

اغلب می‌خواهیم کلیه گره‌های درخت را بررسی کنیم. چند روش برای پیمایش وجود دارد که در آن‌ها گره‌ها می‌توانند پردازش یا ملاقات شوند. هر روش وقتی روی یک درخت دودویی اجرا می‌شود ویژگی‌های مفیدی را در اختیار می‌گذارد.

سه روش معمول پیمایش درخت‌های دودویی روش‌های Preorder، Postorder و Inorder است که در آن‌ها هر گره و فرزندانش به طور بازگشتی ملاقات می‌شوند. هر سه پیمایش از ریشه درخت شروع می‌شوند. تفاوت آن‌ها در ترتیب ملاقات گره و فرزندانش است. در روش Preorder ابتدا گره ملاقات می‌شود، سپس فرزندانش. در روش Postorder ابتدا فرزندان، سپس خود گره ملاقات می‌شود. در روش Inorder گره مابین فرزندان چپ و راست خود ملاقات می‌شود.

N: ریشه درخت **L:** زیر درخت چپ **R:** زیر درخت راست

L N R	Inorder
N L R	Preorder
L R N	Postorder

روش پیمایش Inorder:

پیمایش Inorder با ملاقات فرزند چپ گره جاری شروع شده سپس خود گره و بعد فرزند راست را ملاقات می‌کند. الگوریتم آن به صورت زیر است:

۱- پیمایش زیر درخت چپ به روش Inorder

۲- پردازش ریشه
۳- پیمایش زیر درخت راست به روش Inorder

تابع زیر با توجه به الگوریتم فوق پیاده‌سازی شده است:

```
void Inorder (tree *R)
{
        if (R ==null)
                Return;
        Inorder (R → left);
        cout << R → info;
        Inorder (R → right);
}
```

روش پیمایش Preorder:

در روش Preorder محتوای گره ریشه قبل از فرزند چپ و راست ملاقات می‌شود. الگوریتم بازگشتی پیمایش Preorder به صورت زیر است:

۱- پردازش ریشه
۲- پیمایش زیر درخت چپ به روش Preorder
۳- پیمایش زیر درخت راست به روش Preorder

تابع زیر با توجه به الگوریتم فوق پیاده‌سازی شده است:

```
void Preorder (tree *R)
{
        if (R ==null)
                Return;
        cout << R → info;
        Preorder (R → left);
        Preorder (R → right);
}
```

روش پیمایش Postorder:

پیمایش Postorder ابتدا محتوای زیر درخت چپ و سپس زیر درخت راست و در نهایت گره ریشه را ملاقات می‌کند. الگوریتم بازگشتی پیمایش Postorder به صورت زیر است:

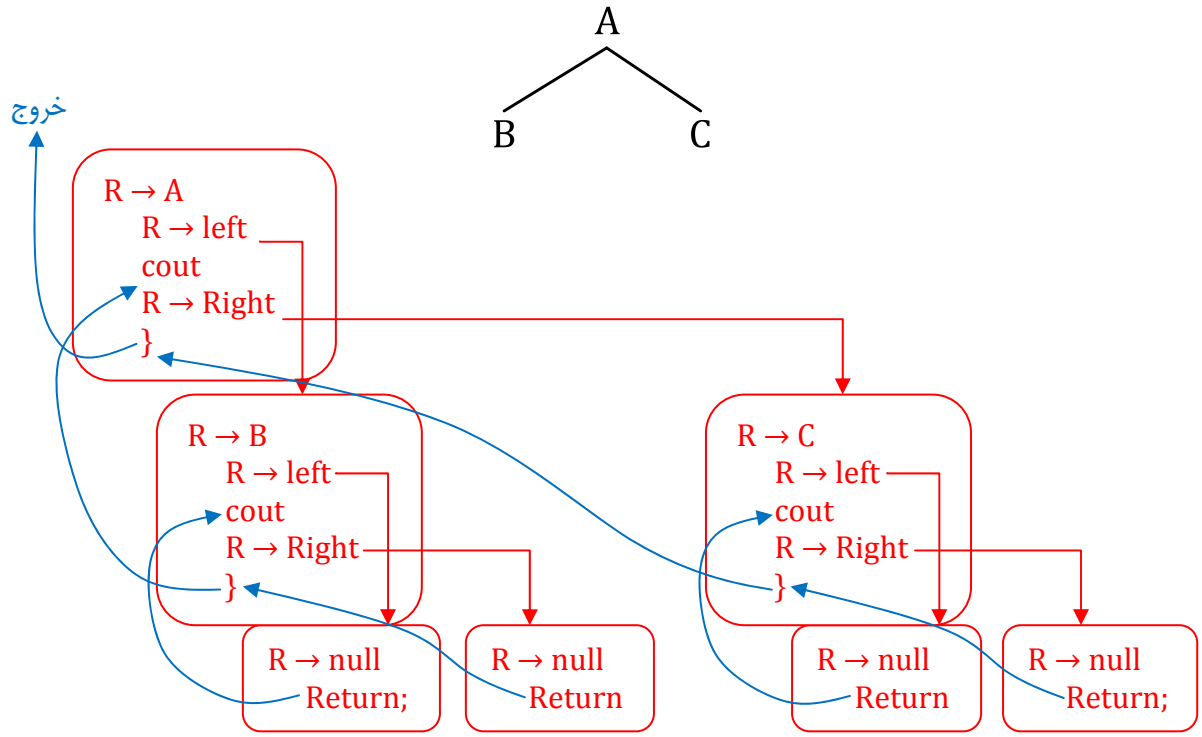
۱- پیمایش زیر درخت چپ به روش Postorder
۲- پیمایش زیر درخت راست به روش Postorder
۳- پردازش ریشه

تابع زیر با توجه به الگوریتم فوق پیاده سازی شده است:

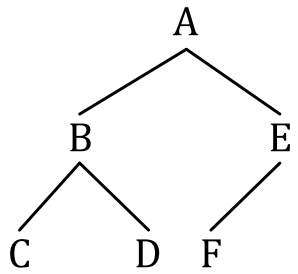
```
void Postorder (tree *R)
{
        if (R ==null)
                Return;
        Postorder (R → left);
```

```
    Postorder (R → right);
    cout << R → info;
}
```

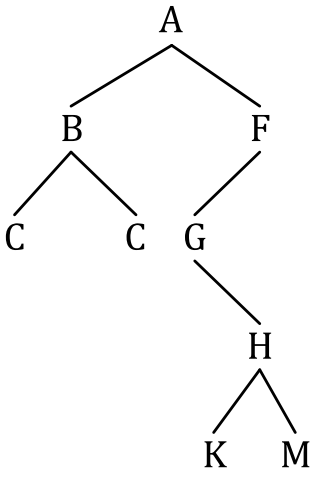
مثال ۲۴) درخت‌های زیر را به سه روش Preorder، Postorder و Inorder پیمایش کنید.



Inorder: BAC Preorder: ABC Postorder: BCA

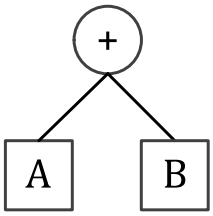


Inorder: CBDAEF Preorder: ABCDEF Postorder: CDBFEA

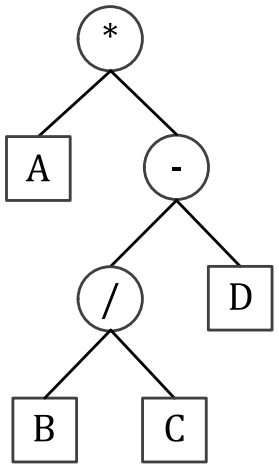


Inorder: BCDEAGKHMF Preorder: ABCDEFGHKM
Postorder: DECBKMHGFA

درخت دودویی محض یک عبارت ریاضی:



Infix: A + B



Infix: A * (B / C – D)

Infix: A * (B / C – D) ≠ Inorder: A * B / C – D

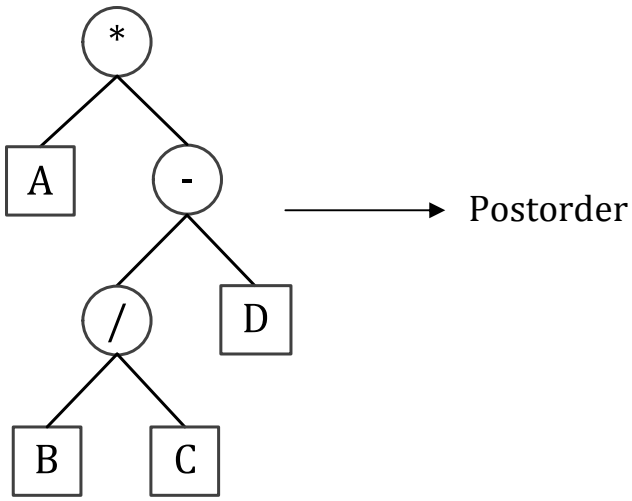
Postfix: ABC / D - * = Postorder: ABC / D - *

Prefix: * A - / BCD = Preorder: * A - / BCD

نتیجه آنکه، عبارت Infix همیشه با عبارت Inorder مساوی نخواهد شد. اگر عبارت Infix ریاضی، بدون پرانتز باشد پیمایش Inorder با عبارت Infix مساوی خواهد شد. اما عبارات Postfix و Prefix با پیمایش‌های Postorder و Preorder همیشه مساوی خواهند بود. به این دلیل که در عبارات Postfix و Prefix و همچنین پیمایش‌های Postorder و Preorder پرانتز دخالت ندارد؛ در نتیجه، عبارات و پیمایش‌های نظیر هم با یکدیگر مساوی خواهند شد.

عبارت پسوندی = ? ⟵ عبارت پیشوندی = * A - / BCD

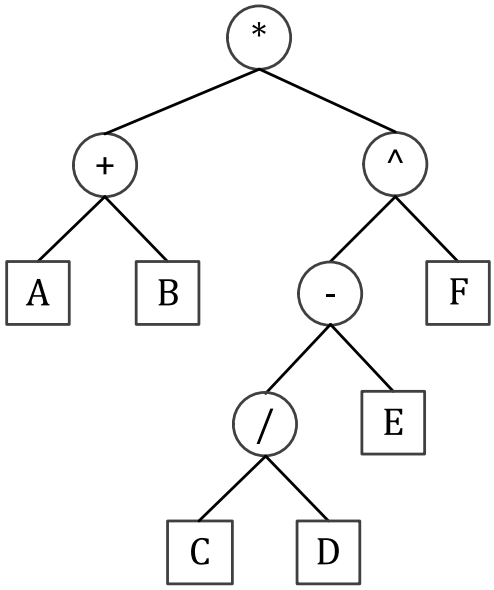
Preorder = NLR



مثال ۲۵) با استفاده از عبارت پیشوندی (Prefix) داده شده عبارت پسوندی (Postfix) را با استفاده از پیمایش Postorder بدست آورید.

Prefix = * + AB ^ - / CDEF ⟶ Postfix = ?

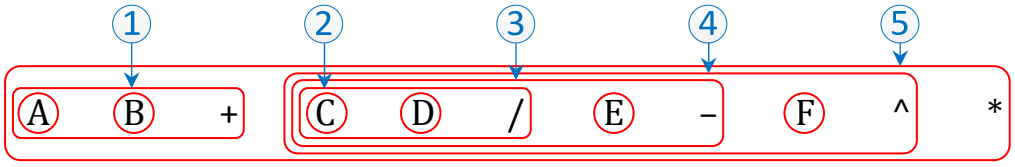
Postfix = AB + CD / E – F ^ * به صورت دستی



Postorder = AB + CD / E – F ^ *

با استفاده از عبارت Postfix می‌توان به صورت دستی عبارت Infix را بدست آورد. که برای این کار به صورت زیر عمل می‌کنیم.

Postfix: AB + CD / E – F ^ *



تبدیل به Infix ↓

Infix: (A + B) * (((C / D) – E) ^ F)

۱- از چپ به راست حرکت می‌کنیم. دو عملوند A و B در سمت راست عملگر + قرار دارند. عملگر + را بین آن‌ها قرار داده و دور آن‌ها پرانتز قرار می‌دهیم. در عبارت Postfix دور دو عملوند و عملگر دایره رسم می‌کنیم تا حکم یک عملوند را داشته باشد.

۲- دو عملوند C و D در سمت راست عملگر / قرار دارند. عملگر / را بین آن‌ها قرار داده و دور آن‌ها پرانتز قرار می‌دهیم. در عبارت Postfix دور دو عملوند و عملگر دایره رسم می‌کنیم تا حکم یک عملوند را داشته باشد.

۳- عملوندی که در مرحله دوم ساخته شد و عملوند E در سمت راست عملگر - قرار دارند. عملگر - را بین آن‌ها قرار داده و دور آن‌ها پرانتز قرار می‌دهیم. در عبارت Postfix دور دو عملوند و عملگر دایره رسم می‌کنیم تا حکم یک عملوند را داشته باشد.

۴- عملوندی که در مرحله سوم ساخته شد و عملوند F در سمت راست عملگر ^ قرار دارند. عملگر ^ را بین آن‌ها قرار داده و دور آن‌ها پرانتز قرار می‌دهیم. در عبارت Postfix دور دو عملوند و عملگر دایره رسم می‌کنیم تا حکم یک عملوند را داشته باشد.

۵- عملوندی که در مرحله چهارم ساخته شد و عملوندی که در مرحله اول ساخته شد، در سمت راست عملگر * قرار دارند. عملگر * را بین آن‌ها قرار داده و دور آن‌ها پرانتز قرار می‌دهیم. در عبارت Postfix دور دو عملوند و عملگر دایره رسم می‌کنیم تا حکم یک عملوند را داشته باشد.

۶- در اینجا عبارت میانوندی (Infix) ما بدست آمده است.

نکته: از روی عبارت پسوندی (Postfix) نمی‌توان درخت دودویی محض یک عبارت ریاضی را بدست آورد.

درخت جستجوی دودویی (Binary Search Tree): وقتی عملیات جستجو، حذف و اضافه مد نظر باشد درخت جستجوی دودویی از تمام ساختارهای دیگر مناسب‌تر است.

یک درخت جستجوی دودویی یا BST نوع خاصی از درخت دودویی است که اگر تهی نباشد خواص زیر را دارا است:

- هر گره یک مقدار منحصر به فرد دارد.
- کلیه مقادیر فرزندان زیر درخت چپ هر گره از مقدار خود گره بزرگ‌تر (بزرگ‌تر مساوی – کوچک‌تر – کوچک‌تر مساوی) هستند.
- کلیه مقادیر فرزندان زیر درخت راست هر گره از مقدار خود گره کوچک‌تر مساوی (کوچک‌تر – بزرگ‌تر مساوی – بزرگ‌تر) هستند.

درخت جستجوی دودویی (BST): مرتبه زمانی جستجو $O(\log_2^n)$ است و حذف و اضافه از درخت نیز به راحتی انجام می‌شود.

جستجوی دودویی آرایه (Array): مرتبه زمانی جستجو $O(\log_2^n)$ است ولی حذف و اضافه از آرایه به دلیل اینکه آرایه به هم می‌ریزد و باید دوباره مرتب گردد، هزینه‌بر است.

جستجوی خطی لیست پیوندی (Linked List): مرتبه زمانی جستجو $O(n)$ است و حذف و اضافه از لیست پیوندی نیز به راحتی انجام می‌شود.

الگوریتم اضافه کردن گره n به BST:

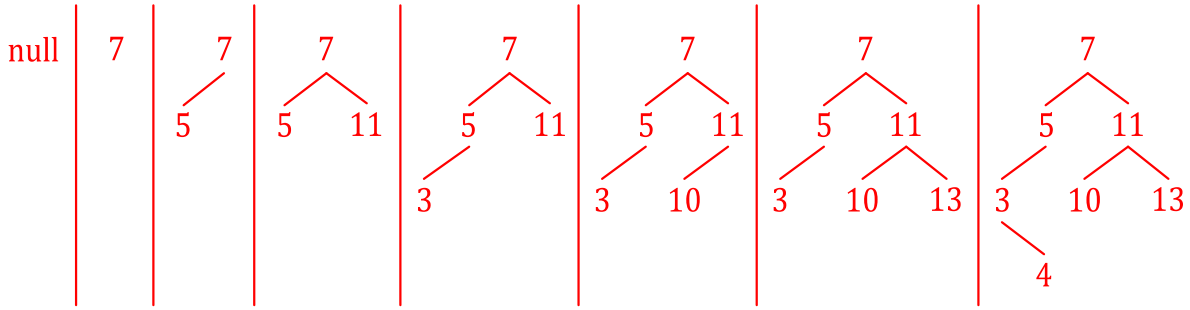
۱- اگر درخت null است، گره n را ریشه درخت قرار بده. در غیر این صورت به مرحله ۲ برو.

۲- گره n را با ریشه درخت مقایسه کن. اگر کوچک‌تر بود به سمت چپ برو در غیر این صورت به سمت راست برو.

۳- مرحله دوم را تکرار کن تا اینکه به null برسید. گره n را جایگزین null کنید.

مثال۲۶) لیست عددی زیر را به درخت BST اضافه کنید.

7 , 5 , 11 , 3 , 10 , 13 , 4



تابع اضافه کردن گره n به BST:

```
void InsertBST (tree *R , int x)
{
        if (R == null)
        {
                R = n;
```

```
                return;
        }
        tree *p = R , *q = null;
        while (p != null)
        {
                int l = 0 , r = 0;
                if (x < p → info)
                {
                        q = p;
                        p = p → left;
                        l = 1;
                }
                else
                {
                        Q = p;
                        p = p → right;
                        r = 1;
                }
        }
        if (l == 1)
                SetLeft (q , x);
        if (r == 1)
                SetRight (q , x);
}
```

تابع جستجوی درخت دودویی:

```
Tree *BST (tree *R , int item)
{
        tree *p = R;
        while (p != null)
        {
                if (p → info == item);
                        return p;   ← جستجوی موفق
                if (p → info > item);
                        p = p → left;
                else
                        p = p → right;
        }
        return p;   ← جستجوی ناموفق
}
```

الگوریتم حذف کردن گره n از BST:

۱- ابتدا گره n را طبق الگوریتم BST پیدا کن.

۲- در صورت پیدا شدن گره n:

الف) گره n هیچ فرزندی ندارد: null را جایگزین گره n کنید تا گره n حذف شود.

ب) گره n یک فرزند دارد: اگر s(n) ریشه بعدی گره n در پیمایش Inorder درخت BST باشد، ابتدا s(n) را با یکی از روش‌های **الف** و **ب** حذف کنید (s(n) هیچ‌گاه دو فرزند ندارد) و سپس آن را جایگزین گره n کنید تا گره n حذف شود.

درخت هرمی (Heap Tree): heap یک ساختمان داده درختی که خواص ویژه‌ای را داراست.

Heap را می‌توان به صورت زیر تعریف کرد:

- یک max meap یک درخت دودویی کامل است که بزرگ هم باشد. در max heap بزرگ‌ترین مقدار همیشه در ریشه قرار می‌گیرد.
- یک min meap یک درخت دودویی کامل است که کوچک هم باشد. در min heap کوچک‌ترین مقدار در ریشه قرار می‌گیرد.

یک درخت بزرگ (max tree) درختی است که مقادیر کلید هر گره بزرگ‌تر از مقادیر فرزندانش باشد. یعنی اگر B فرزند A است آنگاه $key(A) \geq key(B)$ است.

یک درخت کوچک (min tree) درختی است که مقادیر کلید هر گره کوچک‌تر از مقادیر فرزندانش باشد. یعنی اگر B فرزند A است آنگاه $key(A) \leq key(B)$ است.

الگوریتم Heap Sort:

درخت Max Heap: درخت کاملی است که مقدار در هر گره از آن از مقادیر فرزندانش بزرگ‌تر مساوی است. (ماکزیمم در رأس هرم قرار دارد)

درخت Min Heap: درخت کاملی است که مقدار در هر گره از آن از مقادیر فرزندانش کوچک‌تر مساوی است. (مینیمم در رأس هرم قرار دارد)

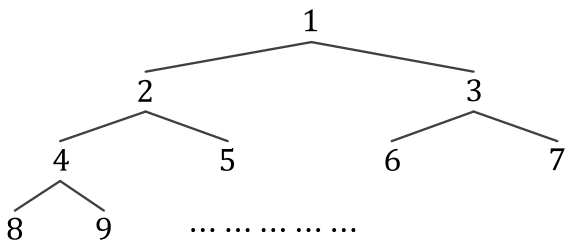
الگوریتم اضافه کردن به درخت Max Heap:

۱- گره n را به انتهای درخت اضافه کنید. درخت کامل است ولی Heap نیست.

۲- بازسازی درخت Heap:

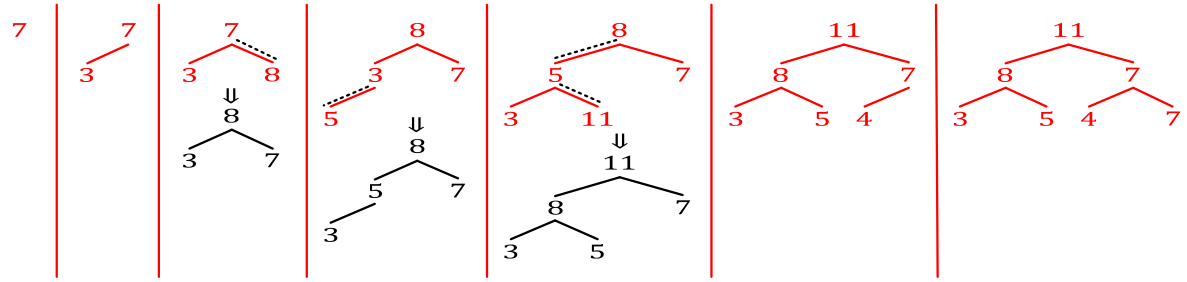
گره n را با ریشه مقایسه کنید. اگر بزرگ‌تر بود جایشان را عوض کنید. این عمل تکرار شود تا اینکه گره n از ریشه کوچک‌تر مساوی شود.

نحوه اضافه کردن به انتهای درخت:



مثال۲۷) لیست عددی زیر را به درخت Max Heap اضافه کنید.

7 , 3 , 8 , 5 , 11 , 4 , 7



$h(T_n)=\lfloor \log_2^n \rfloor +1$

حداکثر تعداد

ساختن درخت Heap

مقایسه‌ها در هر بار بازسازی Heap $= O(\log_2^n)$ $\xrightarrow{\text{تا بازسازی } n \text{ وجود Heap}}$ $= O(n\log_2^n)$

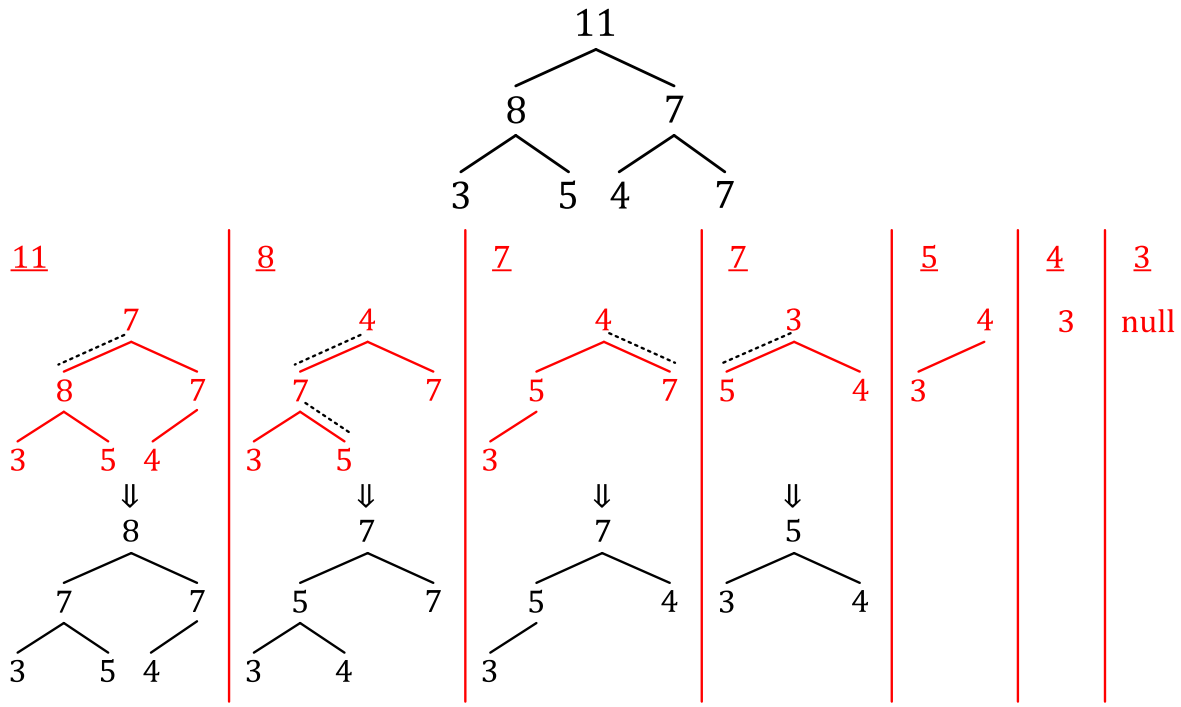
الگوریتم حذف ریشه از درخت Max Heap:

۱- ریشه را حذف کنید و آخرین گره را جایگزین آن کنید. درخت کامل است ولی Heap نیست.

۲- بازسازی درخت Heap:

ریشه را با بزرگ‌ترین فرزندش مقایسه کنید. اگر کوچک‌تر بود، جایشان را عوض کنید. این عمل تکرار شود تا اینکه ریشه از فرزندانش بزرگ‌تر مساوی باشد.

مثال۲۸) برای درخت زیر الگوریتم حذف ریشه از درخت Max Heap را اعمال کنید.



به این درخت که با حذف ریشه اعداد را مرتب می‌کند Heap Sort می‌گویند.

حداکثر تعداد

مقایسه‌ها در هر بار بازسازی Heap $= O(\log_2^n)$ $\xrightarrow{\text{تا بازسازی } n \text{ وجود Heap}}$ $= O(n\log_2^n)+O(n\log_2^n)=O(n\log_2^n)$

ساختن درخت Heap — حذف کردن ریشه‌ها — درخت Heap Sort

کاربردهای Heap: در Max Heap بزرگ‌ترین مقدار همیشه در ریشه است . در Min Heap کوچک‌ترین مقدار همیشه در ریشه قرار می‌گیرد. به خاطر همین ویژگی Heap برای پیاده سازی صف‌های الویت (Priority Queues) بسیار مناسب است. در صف الویت هر عنصر با الویت دلخواه را می‌توان اضافه کرد اما عنصر ماکزیمم یا مینیمم همیشه ابتدا از صف حذف می‌شود.

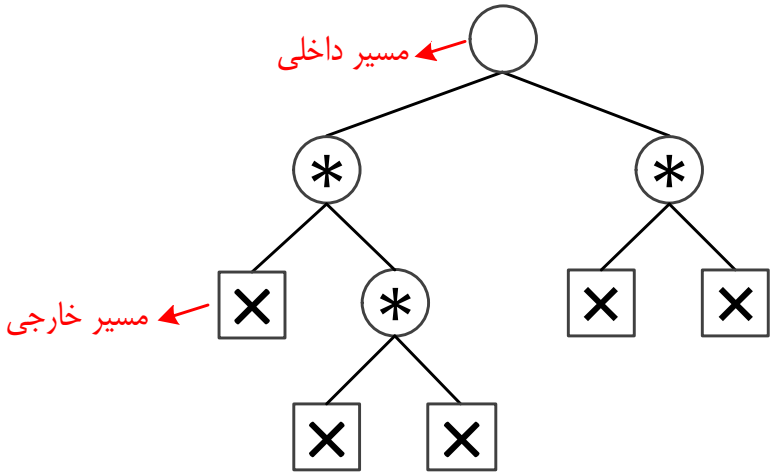
الگوریتم هافمن: الگوریتم فشرده سازی هافمن را دیوید هافمن پروفسور دانشگاه MIT آمریکا اختراع کرد. روش فشرده سازی هافمن الگوریتمی است که برای فشرده سازی متن مناسب می‌باشد.

الگوریتم هافمن جزو خانواده الگوریتم‌هایی است که طول کد متغیری دارند. این به آن معناست که نمادهای مجزا (برای نمونه کاراکترهایی در یک فایل متنی) با رشته بیت‌هایی که طول‌های مختلفی دارند تعویض می‌شود. بنابراین نمادهایی که زیاد در یک فایل تکرار می‌شوند یک رشته بیت کوتاه می‌گیرند در حالی که نمادهای دیگر که به ندرت دیده می‌شوند رشته بیت طولانی‌تری را می‌گیرند.

طول مسیر داخلی (L_I): مجموع طول مسیرهایی که از ریشه شروع شده و به یک گره داخلی ختم می‌شوند.

طول مسیر خارجی (L_E): مجموع طول مسیرهایی که از ریشه شروع شده و به یک گره خارجی ختم می‌شوند.

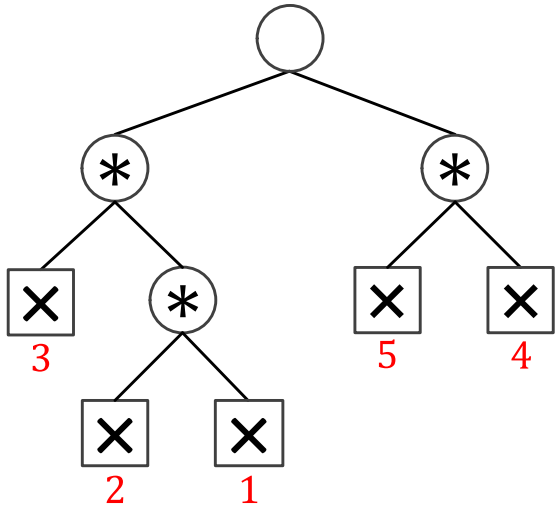
طول مسیر خارجی وزن داده شده ($L_E(w)$): مجموع حاصل‌ضرب طول مسیرهایی که از ریشه شروع شده و به یک گره داخلی ختم می‌شوند در وزن آن گره‌های خارجی.



L_I: 1 + 2 + 1 = 4

L_E: 2 + 3 + 3 + 2 + 2 = 12

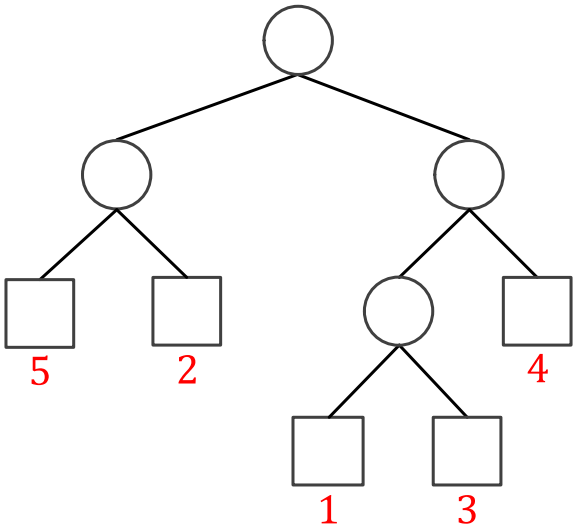
در صورتی که به گره‌های خارجی شکل بالا به صورت زیر وزن داده شود طول مسیر خارجی وزن داده شده به صورت زیر در خواهد آمد.



$L_E(w)$: 2*3 + 3*2 + 3*1 + 2*4 + 2*5 = 33

می‌توان درخت دودویی بالا را با توجه به اوزان ۱، ۲، ۳، ۴ و ۵ به صورت‌های مختلف ترسیم کرد که یکی از حالت‌های آن به صورت شکل زیر می‌باشد ولی طول مسیر خارجی وزن داده شده آن مطابق شکل بالا نخواهد بود. الگوریتم هافمن با روشی که برای رسم درخت دودویی ارائه می‌کند، با توجه به

اوزان مسیرهای خارجی، درختی محض تولید می‌کند که حروفی که بیشترین تکرار را دارند با کمترین کد، کدینگ شوند و حروفی که کمترین تکرار را دارند با بیشترین کد، کدینگ شوند.

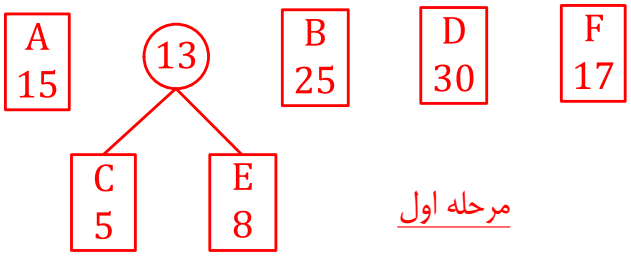


$L_E(w)$: 2*5 + 2*2 + 3*1 + 3*3 + 2*4 = 34

مثال۲۹) الگوریتم هافمن را برای حروف با اوزان داده شده زیر رسم کنید.

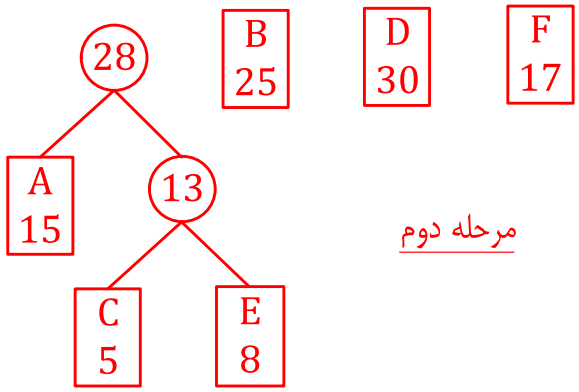
A	B	C	D	E	F
15	25	5	30	8	17

برای این کار در مرحله اول، کمترین اوزان را با یکدیگر جمع کرده در یک گره داخلی (مسیر داخلی) قرار داده و دو حرفی (در مثال حرف C با وزن ۵ و حرف E با وزن ۸) که کمترین وزن (تکرار) را دارند در دو گره خارجی (مسیر خارجی) قرار می‌دهیم. (همانند شکل زیر)



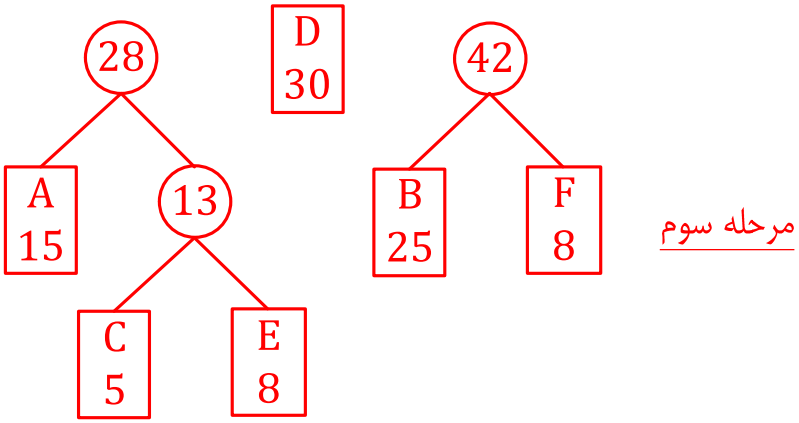
مرحله اول

در مرحله دوم، کمترین اوزان (حتی مسیرهای داخلی) را با یکدیگر جمع کرده در یک گره داخلی (مسیر داخلی) قرار داده و دو حرفی (در مثال جمع مرحله اول با وزن ۱۳ و حرف A با وزن ۱۵) که کمترین وزن (تکرار) را دارند در دو گره خارجی (مسیر خارجی) قرار می‌دهیم. (همانند شکل زیر)

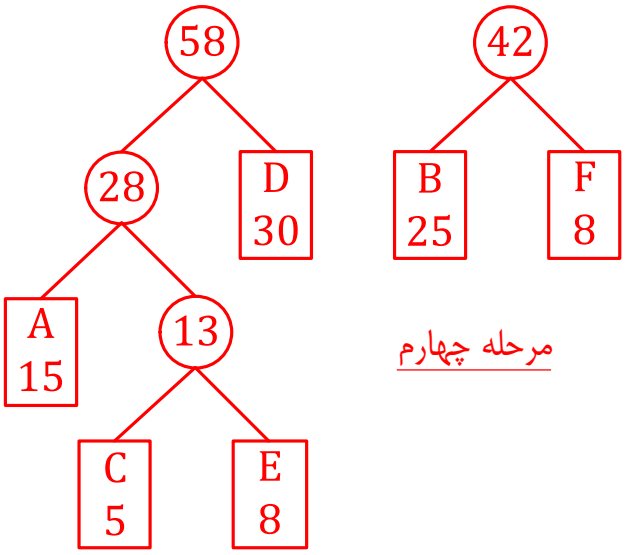


مرحله دوم

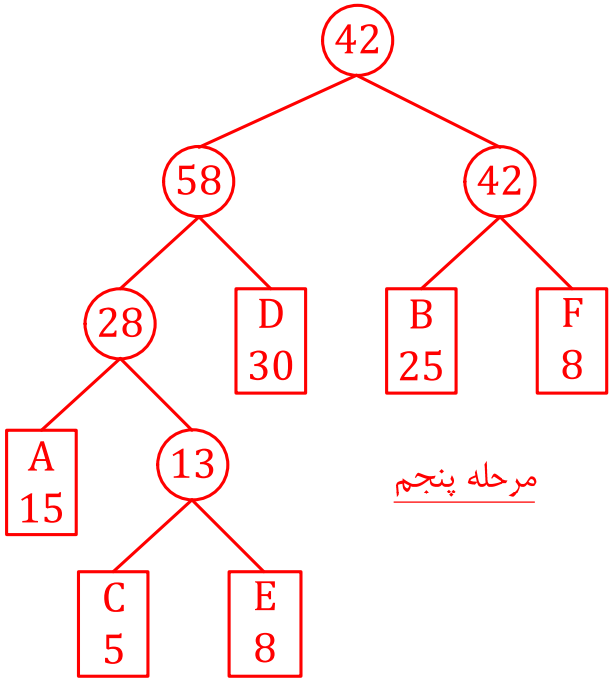
سایر مراحل را تا جایی که کلیه حروف به همراه اوزانشان به عنوان برگ‌های درخت، در درخت قرار بگیرند، پیش می‌بریم در انتها جمع اوزان کلیه حروف در ریشه قرار می‌گیرد. در این درخت محل قرارگیری اوزان مهم نیست. چه وزن کوچک‌تر در سمت راست باشد و وزن بزرگ‌تر در سمت چپ و چه وزن بزرگ‌تر در سمت راست باشد و وزن کوچک‌تر در سمت چپ. (همانند اشکال زیر)



مرحله سوم

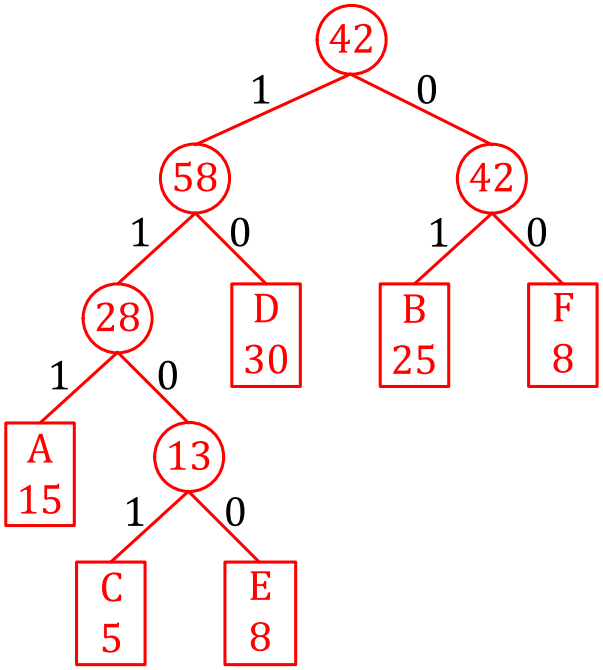


مرحله چهارم



مرحله پنجم

پس از اینکه درخت تشکیل شد، بر روی یال‌های آن صفر یا یک قرار می‌دهیم. اگر یال سمت راست ریشه را یک قرار دادیم باید کلیه یال‌هایی که در سمت راست قرار دارند را یک و کلیه یال‌های سمت چپ را صفر قرار دهیم. در صورتی که یال سمت چپ ریشه را یک قرار دادیم باید کلیه یال‌هایی که در سمت چپ قرار دارند را یک و کلیه یال‌های سمت راست را صفر قرار دهیم. (همانند اشکال زیر که کلیه یال‌های چپ را یک و کلیه یال‌های سمت راست را صفر قرار داده‌ایم)



پس از اینکه یال‌های درخت را با صفر و یک کدگذاری کردیم، بر اساس کدها، طول مسیر هر حرف را همانند زیر در جلوی حرف می‌نویسیم.

A: 111	B: 01	C: 1101
D: 10	E: 1100	F: 00

با توجه به کد مربوط به هر حروف، و محاسبه مجموع حاصل‌ضرب طول مسیرهایی که از ریشه شروع و به هر حرف ختم می‌شود در وزن (تکرار) مربوط به هر حرف ($L_E(w)$) به این نتیجه می‌رسیم که برای یک متن ۱۰۰ کاراکتری از حروف ABCDEF ۲۴۶ بیت فضای حافظه نیاز است؛ که این فضا در حالت معمولی ۸۰۰ بیت خواهد بود.

$L_E(w)$: 3*15 + 4*5 + 4*8 + 2*30 + 2*25 + 2*17 = 246 bit

حالت معمولی: 15*8 + 25*8 + 5*8 + 30*8 + 8*8 + 17*8 = 800 bit

پایان