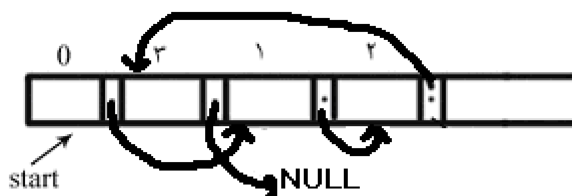
لیست پیوندی:

مجموعه ای از عناصر می باشد که عناصر لزوماً در خانه های پشت سرهم حافظه قرار ندارند. ترتیب عناصر توسط اشاره گرها مشخص می شود. یعنی هر عنصر آدرس بعدی را در خود ذخیره می کند.



لیست تک پیوندی یا لیست یکطرفه

هر عنصر لیست پیوندی که به آن گره (**Node**) می گوییم یک ساختمان است و در این ساختمان یک فیلد اشاره گر به نام **next** وجود دارد که آدرس عنصر بعدی را در خود ذخیره می کند. یک عنصر لیست پیوندی در زبان **C** به صورت زیر تعریف می شود.

```
struct   node
       int    data;
       Struct   node * next;
};
```

Data	**Next**

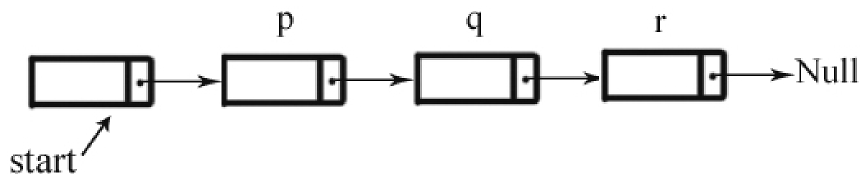
در این ساختار فرض کرده ایم که **data** از نوع **int** باشد ولی می توان آن را از هر نوع دیگری در نظر گرفت.

در لیست پیوندی یک طرفه – هر عنصر فقط ادرس خانه بعدی را ذخیره می کند. همچنین آدرس اولین عنصر در اشاره‌گر **start** ذخیره می شود.

```
struct list {
     struct node * start;
};
```

اگر هیچ عنصری در لیست وجود نداشته باشد یعنی لیست تهی است و مقدار **start** برابر **NULL** می باشد.

نمایش یک لیست پیوندی به صورت زیر است :



p->next	**q**
start->next	**p**
p->next->next	**r**
q->next	**r**
r->next	**NULL**
q->next->next	**NULL**

برای درخواستی از حافظه از تابع `malloc` و برای پس دادن حافظه از تابع `free` استفاده می کنیم.

```
struct   node  * temp ;
temp = ( struct node * ) malloc ( sizeof ( struct  node )
) ;
```

این دستور یک حافظه به اندازه ساختمان **node** اختصاص می دهد و ادرس شروع آن را در اشاره گر **temp** ذخیره می کند. چون خروجی تابع malloc **void *** است بنابراین ، خروجی آن را به صورت

(**struct** node *) تبدیل کردیم.

* اگر **malloc** نتواند حافظه ای را اختصاص دهد ، مقدار **NULL** را بر می گرداند.

از تابع **free** برای پس دادن حافظه می تواند به صورت زیر استفاده شود:

حافظه ای که قبلاً برای **temp** استفاده شد پس می دهد - **free (temp) ;**

بعد از عملیات **free** دیگر نمی توان از اشاره گر استفاده کرد.

مگر اینکه برای اشاره گر یک حافظه جدید اختصاص دهیم.

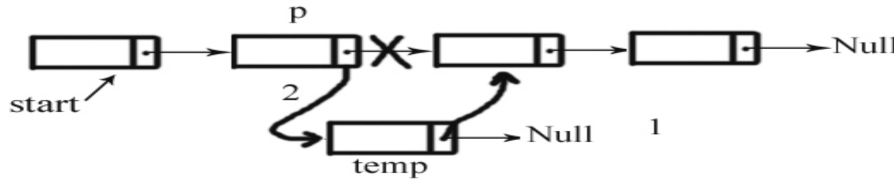
عملیات ایجاد یک گره جدید با داشتن مقدار **data**.

```
struct node * create_node( int data)
{
      struct node * temp;
      temp=(struct node *) malloc( sizeof(struct node) );
      if (temp==NULL) {
            printf("ERROR: out of memory");
            exit(1);
      }
      temp->data = data;
      temp->next=NULL;
      return temp;
}
```

در تابع create_node بایستی به مقدار مورد نیاز برای ذخیره یک گره از فضای حافظه پویا استفاده شود. به این دلیل از تابع malloc استفاده شده است. اگر تابع malloc نتواند میزان حافظه مورد نیاز را تهیه کند در آن صورت مقدار NULL بر میگرداند. و در صورتی که تابع malloc موفق باشد آدرس محل تخصیص داده شده را بر می گرداند.

عملیات درج یک عنصر در لیست پیوندی :

هدف : درج یک عنصر بعد از عنصری که آدرس آن با **P** مشخص شده است.



پیچیدگی زمانی آن O(1) است.

```
temp -> next = p -> next ;
P -> next = temp ;
```

الگوریتم درج یک عنصر در لیست پیوندی :

```
void  insert_after(struct node *p,struct node *temp)
{
      if ( start == NULL )   start = temp;
      else {
            temp->next=p->next;
            p->next =temp;
      }
}
```

الگوریتم درج یک مقدار در لیست پیوندی یکطرفه.

```
void  add_list( struct node *p,int x)
{
      struct node * temp;
      temp = create_node(x);
      insert_after( p, temp );
}
```

یافتن عنصر قبلی در لیست پیوندی یکطرفه:

```
struct node * find_prev( struct node * p)
{
      struct node * temp=start;
      while (temp && temp->next !=p)
            temp=temp->next;
      return temp;
}
```

در تابع frind_prev از اولین عنصر لیست شروع می کنیم و در یک حلقه تکرار لیست را پیمایش می کنیم تا به عنصری برسیم که عنصر بعدی آن p باشد.

اگر p در لیست باشد آنگاه دو وضعیت وجود دارد:

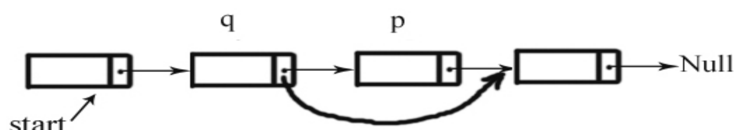
اگر p اولین عنصر لیست باشد خروجی تابع find_prev برابر NULL خواهد بود.

اگر p در لیست باشد و p اولین عنصر لیست نباشد انگاه یک آدرس معتبر به عنوان آدرس عنصر قبلی p وجود دارد.

اگر p در لیست نباشد آنگاه خروجی تابع find_prev مقدار NULL خواهد بود.

حذف یک عنصر از لیست پیوندی :

هدف : حذف عنصری از لیست پیوندی که ادرس آن با **P** مشخص شده است.و فرض می شود که عنصر مشخص شده با **P** در لیست باشد.



ما برای حذف خانه **P** ، باید آدرس خانه قبلی که **q** باشد را پیدا کنیم و به خانه بعد از **P** انتقال دهیم.

```
q-> next = P -> next ;
```

* اگر **P** ، اولین عنصر لیست باشد یعنی **start** باشد ، خانه بعد از **p** بعد از حذف اولین عنصر لیست خواهد بود.

```
if (p==start)    start = start -> next ;
```

- اگر **P** ، در وسط بود ، برای پیدا کردن **q** (که به **p** اشاره می کند)از تابع **find_prev** کمک می گیریم.

الگوریتم حذف یک عنصر از لیست پیوندی :

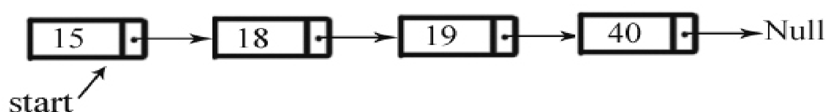
```
void    delete_node ( struct node * P )
{
   if ( P == start ) {
```

```
      start = start -> next;
       free(p);
       return ;
    }
    struct node *q = find_prev(p);
  if ( q==NULL ) {
          printf("Element not found");
    } else {
          q->next = p->next;
          free(p);
    }
}
```

عملیات جستجو در لیست پیوندی :

جستجوی عنصری از لیست پیوندی با داشتن **data** .

می خواهیم **19** را جستجو کنیم.



struct node * search_list(int x);

تابع **search_list** مقدار **x** را در لیست جستجو میکند. اگر **x** پیدا شد آنگاه ادرس عنصری که **data** در ان وجود دارد به عنوان خروجی بر می گرداند در غیر این صورت مقدار **NULL** بر می گرداند.

الگوریتم جستجوی یک عنصر از لیست پیوندی :

```
struct  node * search_list( int x)
{
      struct node * p = start ;
      while ( p! = NULL ) {
            if ( p -> data == x ) break ;
      p = p -> next;
      }
      return p ;
}
```

مرتبه اجرایی الگوریتم فوق :

اگر **n** تعداد عناصر لیست باشد :

1) در بهترین حالت– وقتی **x** در اولین خانه باشد – مرتبه اجرایی $O(1)$ است.

2) در بدترین حالت– وقتی **x** در آخرین خانه باشد یا در هیچ خانه ای نباشد – $O(n)$ است.

3) در حالت متوسط– حالت میانگین را در نظر می گیریم.

x	عنصر اول باشد	1 مرحله
x	عنصر دوم باشد	2 مرحله
...		
...		
x	آخرین عنصر باشد	**n** مرحله

میانگین تعداد تکرار حلقه **while** برابر است با:

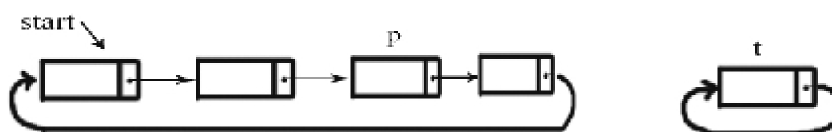
```
(1/n)(1+2+…+n) = (1/n) ( n(n+1)/2 )
= (n+1)/2
```

در حالت کلی مرتبه اجرایی الگوریتم فوق عبارتند از **O(n)**

تمرین : تابع حذف یک عنصر با داشتن **data**.

راهنمایی : از تابع **search_list** و تابع **delet_node** کمک بگیرید.

لیست حلقوی :



،

اگر تابع **create_node** در لیست یک طرفه را باز نویسی نماییم به طوری که یک گره حلقوی به ما تحویل دهد. بنابراین دستور **temp->next=NULL;** در تابع **create_node** را باید تغییر دهیم و قرار می دهیم: **temp->next=temp;** پس تابع **create_node** برای لیست حلقوی به صورت زیر پیاده سازی می شود.

```
struct node * create_node( int data)
{
      struct node * temp;
      temp=(struct node *) malloc( sizeof(struct node) );
      if (temp==NULL) {
            printf("ERROR: out of memory");
            exit(1);
      }
      temp->data = data;
      temp->next=temp;
      return temp;
}
```

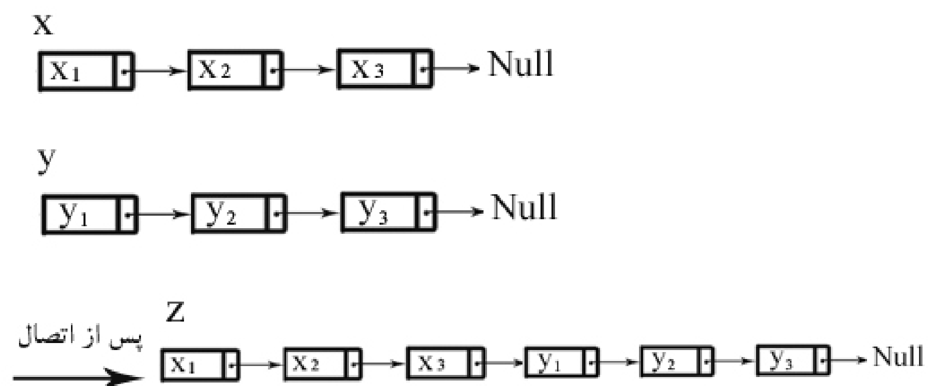
الگوریتم درج به لیست حلقوی :

```
void  insert_after( struct node *p, struct node *q)
{
      if ( start == NULL )     start = t;
      else {
        t -> next = P-> next;
        p – next = t ;
      }
}
```

عملیات حذف در لیست پیوندی حلقوی شبیه عملیات حذف در لیست پیوندی خطی است.

تمرین

1- الگوریتم الحاق دو لیست پیوندی یکطرفه را بنویسید:

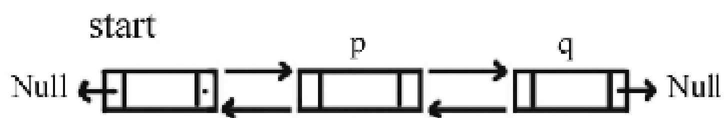


2-الگوریتم الحاق دو لیست پیوندی یکطرفه حلقوی را بنویسید.

3- یک لیست یکطرفه موجود است، الگوریتمی معکوس کردن این لیست پیوندی را بنویسید.

لیست پیوندی دو طرفه :

در لیست پیوندی دو طرفه هر عنصر آدرس عنصر بعدی و آدرس عنصر قبلی را ذخیره می کند. (در صورتیکه در لیست پیوندی یک طرفه هر عنصر فقط آدرس عنصر بعدی را ذخیره می کند). در شکل زیر یک لیست پیوندی دوطرفه با سه گره را مشاهده می کنید.

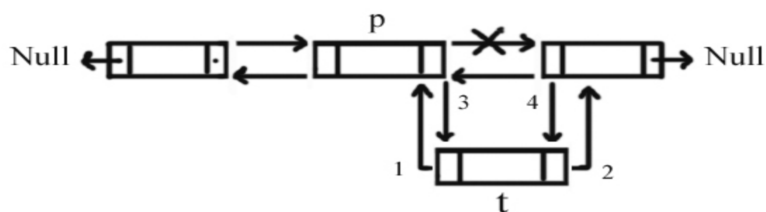


ساختمان داده مربوط به لیست دو طرفه :

```
struct  node {
     int data ;
     struct   node   *  next ;
     struct   node  *  prev ;
};
```

عملیات درج یک عنصر در لیست پیوندی دوطرفه :

الف) عملیات درج یک گره جدید **t** بعد از گره **p** در لیست دو پیوندی:

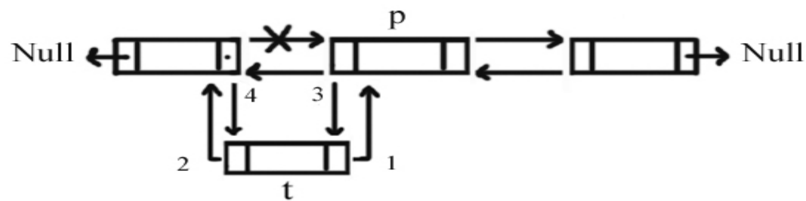


عملیات درج در 4 مرحله انجام می شود که به به شرح زیر است.

```
t->prev = p;
t->next = p->next;
p->next = t;
```

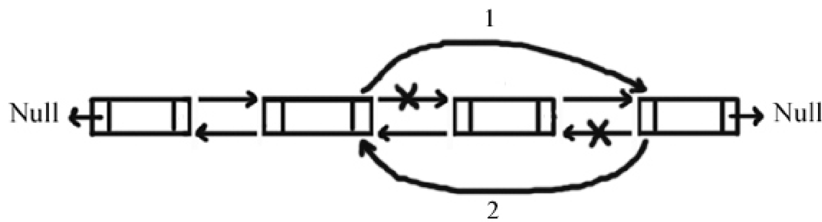
```
if (t->next) t->next->prev= t;
```

ب) عملیات درج یک گره جدید **t** قبل از گره **P** در لیست دوپیوندی.




```
t->next = p;
t->prev = p->prev;
p->prev = t;
if (t->prev) t->prev->next=t;
```

عملیات حذف یک گره از لیست دو پیوندی :




```
1)  if (p->prev)
        p -> prev -> next = p -> next ;
2)  if (p->next)
        p -> next -> prev = p -> prev ;
3) free(p);
```

سوال: چرا در دستورهای 1 و 2 از دستور **if** استفاده شده است؟