داده‌ساختارهای ساده

• لیست‌ها

– لیست‌ها یک‌سویه، دوسویه، حلقه‌ای

– پشته

– صف

– لیست‌های کلی

– کاربردهای لیست‌ها

- درخت‌ها
  - درخت‌های کلی، دودویی
  - درخت عبارت و کار با آن‌ها
  - درخت دودویی جست‌وجو

لیست‌ها

دنباله‌ای از عناصر، که ترتیب آن‌ها مهم است

اعمال متداول

- ایجاد یک لیست تهی
- محاسبه‌ی تعداد عناصر موجود در لیست (اندازه‌ی لیست)
- درج یک عنصر در ابتدای یا انتهای لیست
- درج یک عنصر بعد یا قبل از یک عنصر داده‌شده
- حذف یک عنصر از (ابتدا، انتها، یا عنصر بعدی) لیست

لیست‌ها (ادامه)

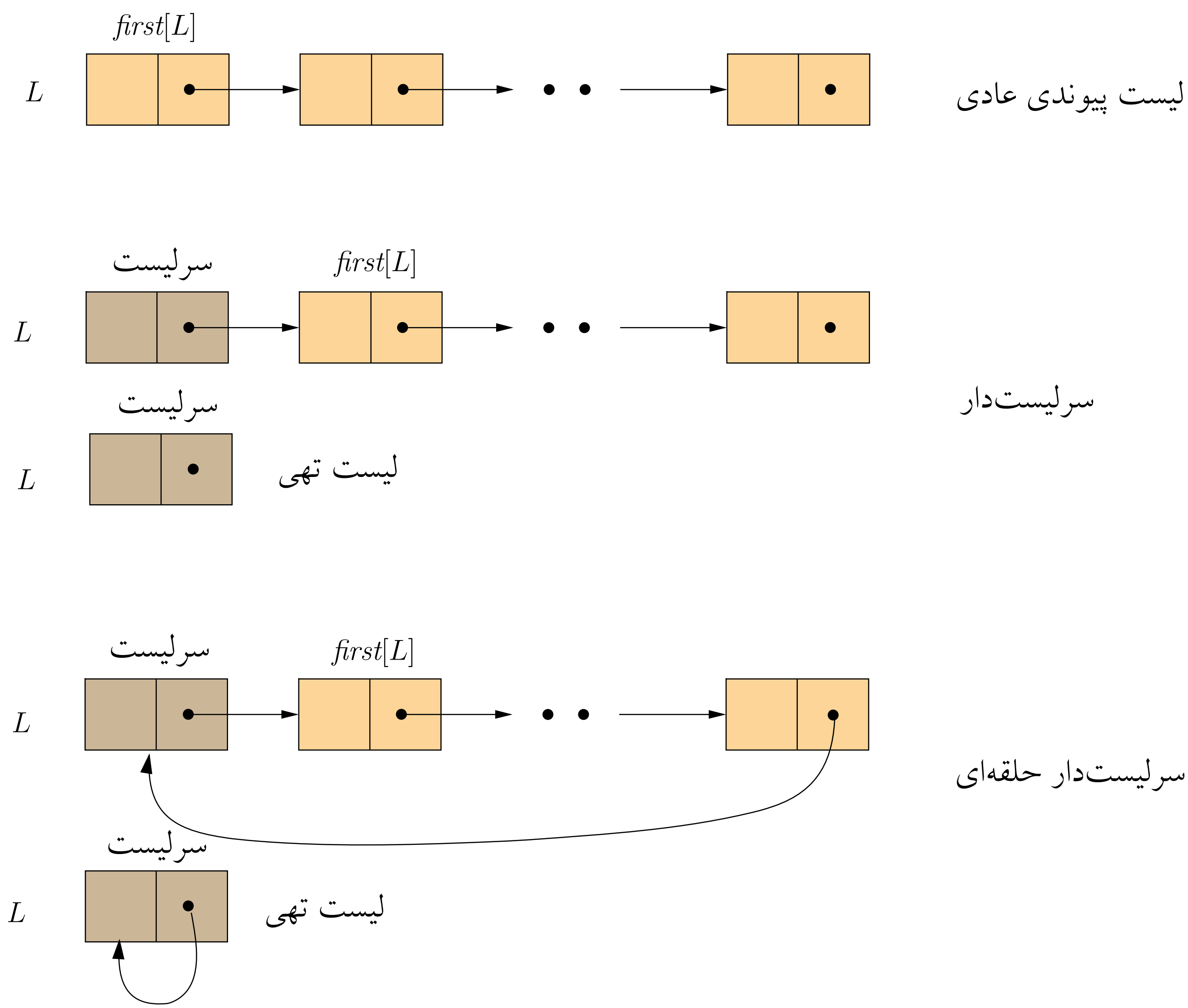
بسته به مکان درج یا حذف یک عنصر، لیست‌ها با اسامی زیر شناخته می‌شوند:

- **پُشته** (stack)**: درج و حذف فقط در یک طرف لیست**
  Last-In-First-Out (LIFO) **یا** First-In-Last-Out (FILO)
- **صف** (queue)**: درج فقط در انتها و حذف از ابتدای**
  First-In-First-Out (FIFO)

انواع لیست‌های پیوندی: (خطی) یک‌سویه، دوسویه، حلقه‌ای، سلسله‌مراتبی و لیست‌های کلی

لیست‌های پیوندی یک‌سویه

```
class Node {
    private Object element;
    private Node next;
    // constructors
    Node(){
       this(null,null);
    }
    public Node(Object e, Node n){
       element = e
       next = n;
    }
    void setElement(Object newElem){ element = newElem;}
    void setNext(Node newNext){ next = newNext;}
    Object getElement(){ return element;}
    Node getNext() {return next;}
}
```



first[L]
L
لیست پیوندی عادی
سرلیست
first[L]
L
سرلیست
L
لیست تهی
سرلیست‌دار
سرلیست
first[L]
L
سرلیست‌دار حلقه‌ای
سرلیست
L
لیست تهی

پیاده‌سازی با CLRS

مولفه‌ها: $next$، $element$ و $size[L]$

جاوا (Java)	شبه‌کد CLRS
`x = new Node()`	$x \leftarrow \textsc{Allocate-Node}()$
`x = new Node(element e, next n)`	$x \leftarrow \textsc{Allocate-Node}(e, n)$
`null x`	$\textsc{Free-Node}(x)$
`x.getNext()`	$next[x]$
`x.setNext(n)`	$next[x] \leftarrow n$

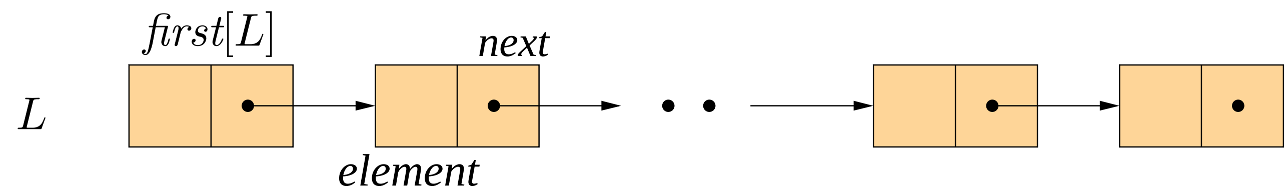
اعمال اصلی بر روی یک لیست خطی

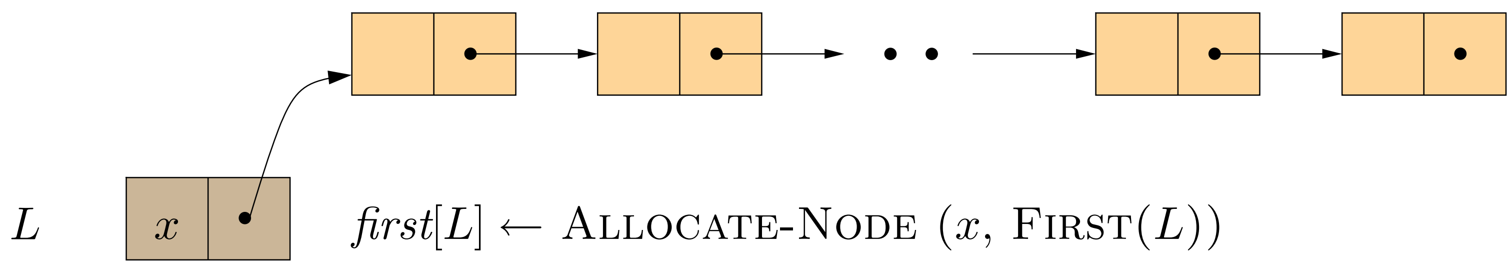
- $\textsc{Create-List}(L)$: ایجاد یک لیست تهی L
- $\textsc{Size}(L)$: تعداد عناصر لیست را بر می‌گرداند
- $\textsc{First}(L)$: عنصر اول را برمی‌گرداند
- $\textsc{isEmpty}(L)$: مشخص می‌کند که آیا لیست خالی
- $\textsc{Insert-First}(L, x)$: درج عنصری با مقدار x در ابتدای L
- $\textsc{Insert-After}(L, x, n)$: درج عنصری با مقدار x پس از عنصر n در L
- $\textsc{Delete-First}(L)$: عنصر اول لیست L را حذف می‌کند
- $\textsc{Delete-After}(L, n)$: عنصر پس از عنصر n در L را حذف می‌کند

پیاده‌سازی

```
CREATE(L)
    size[L] ← 0

SIZE(L)
    return size[L]

FIRST(L)
    if SIZE(L) ≠ 0
      then return first[L]
      else error list is empty

isEMPTY(L)
    return SIZE(L) = 0
```

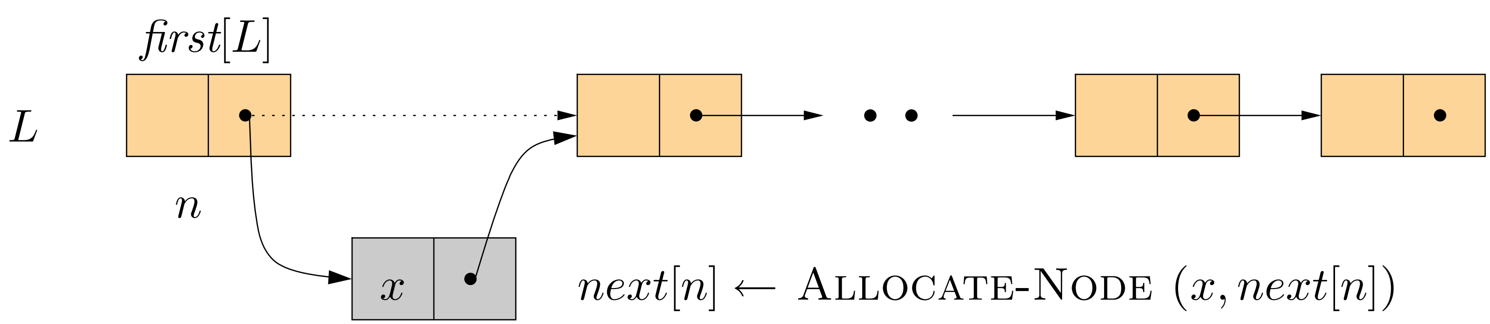
```
INSERT-FIRST (L, x)
    first[L] ← Allocate-Node (x, First(L))
    size[L] ← size[L] + 1

INSERT-AFTER (L, x, n)
    if n = null
      then error element is empty
    next[n] ← ALLOCATE-NODE (x, next[n])
    size[L] ← size[L] + 1
```





$first[L] \leftarrow \textsc{Allocate-Node}\ (x,\ \textsc{First}(L))$

$\textsc{Insert-First}(L, x)$

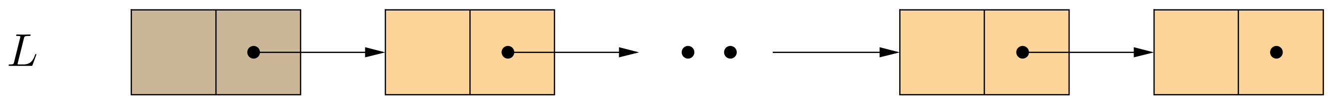


$next[n] \leftarrow \textsc{Allocate-Node}\ (x, next[n])$

$\textsc{Insert-After}(L, x, n)$

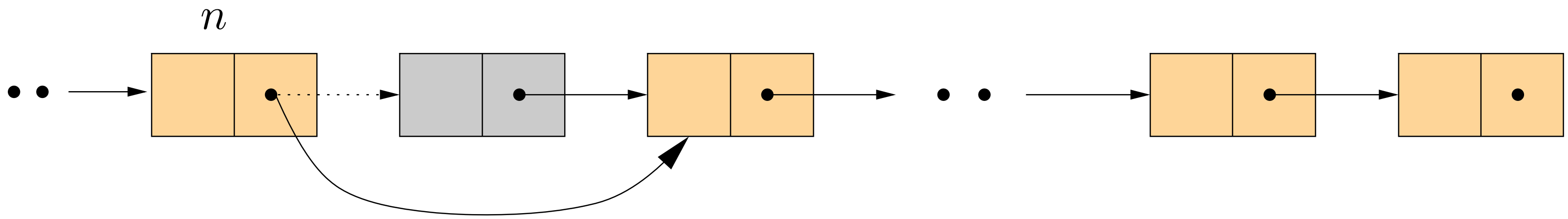
```
Delete-First(L)
if isEmpty (L)
   then error list is empty
n ← First(L)
first[L] ← next[n]
Free-Node(n)
size[L] ← size[L] − 1
```

```
DELETE-AFTER (L, n)
if isEmpty (L)  or n = null  or next[n] = null
  then  error  element does not exist
r ← next[n]
next[n] ← next[r]
FREE-NODE(r)
size[L] ← size[L] − 1
```



$$first[L] \leftarrow next[\,first[L]]$$

$$\textsc{Delete-First}(L)$$



$$next[n] \leftarrow next[next[n]]$$

$$\textsc{Delete-After}(L, n)$$

روشن است که هر یک از این اعمال در $\mathcal{O}(۱)$ قابل انجام است.

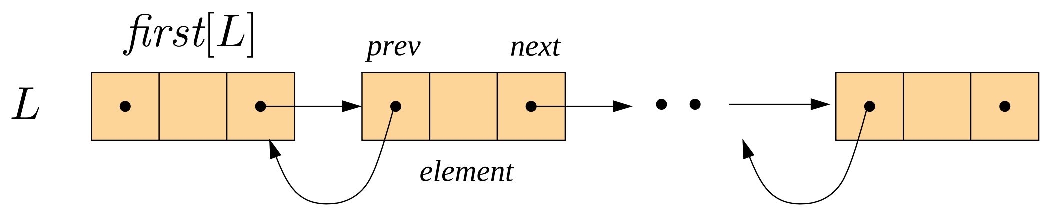
درج و حذف در لیست دوسویه‌ی خطی

```
INSERT-AFTER (L, x, n)
      ▷ عنصری با محتوای x را پس از عنصر n در لیست دوسویه‌ی L درج می‌کند
   if isEMPTY(L)  or n = null
      then  error  element n does not exist
   r ← next[n]
   next[n] ← ALLOCATE-NODE (x, n, r)
   prev[r] ← next[n]
   size[L] ← size[L] + 1
```

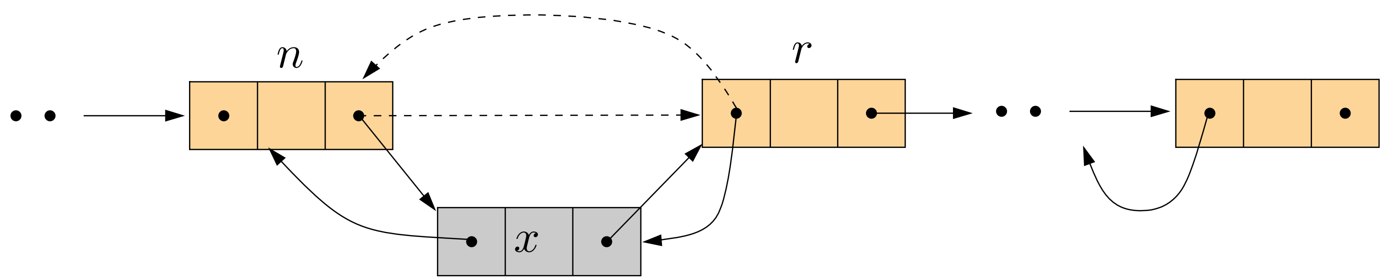
<u>DELETE-AFTER</u> (L, n)

▷ عنصر بعدی n را در لیست دو سویه‌ی L حذف می‌کند

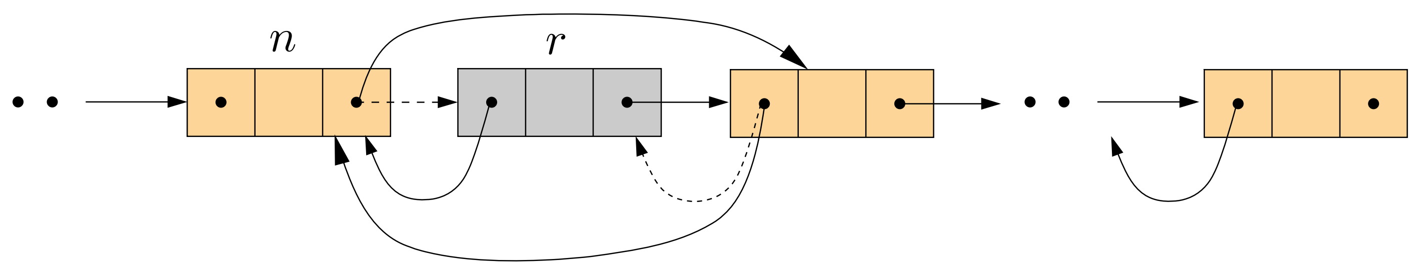
if isEmpty(L) **or** $n =$ **null** **or** $next[n] =$ **null**
  then **error** element does not exist
$r \leftarrow next[n]$
if $next[r] \neq$ **null**
  then $prev[next[r]] \leftarrow n$
$next[n] \leftarrow next[r]$
Free-Node(r)
$size[L] \leftarrow size[L] - 1$



لیست خطی دوسویه‌ی

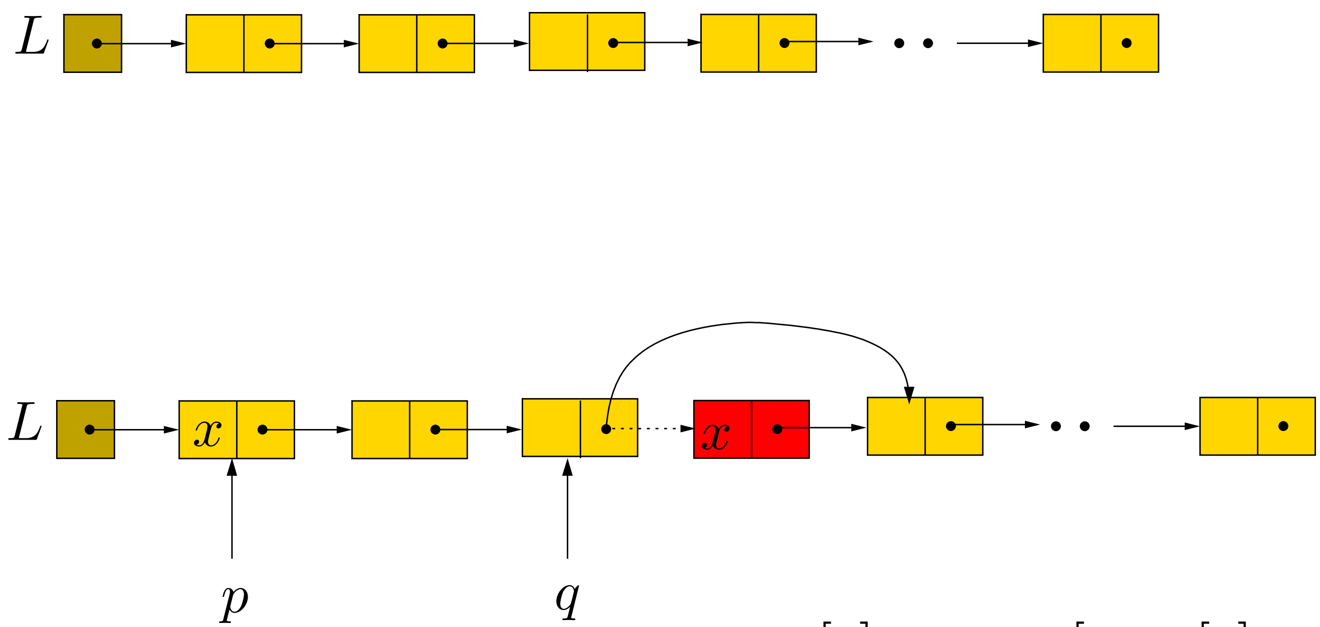


INSERT-AFTER(L, x, n)



DELETE-AFTER(L, n)

عملیات دیگر بر روی لیست‌ها: حذف عناصر تکراری در یک لیست

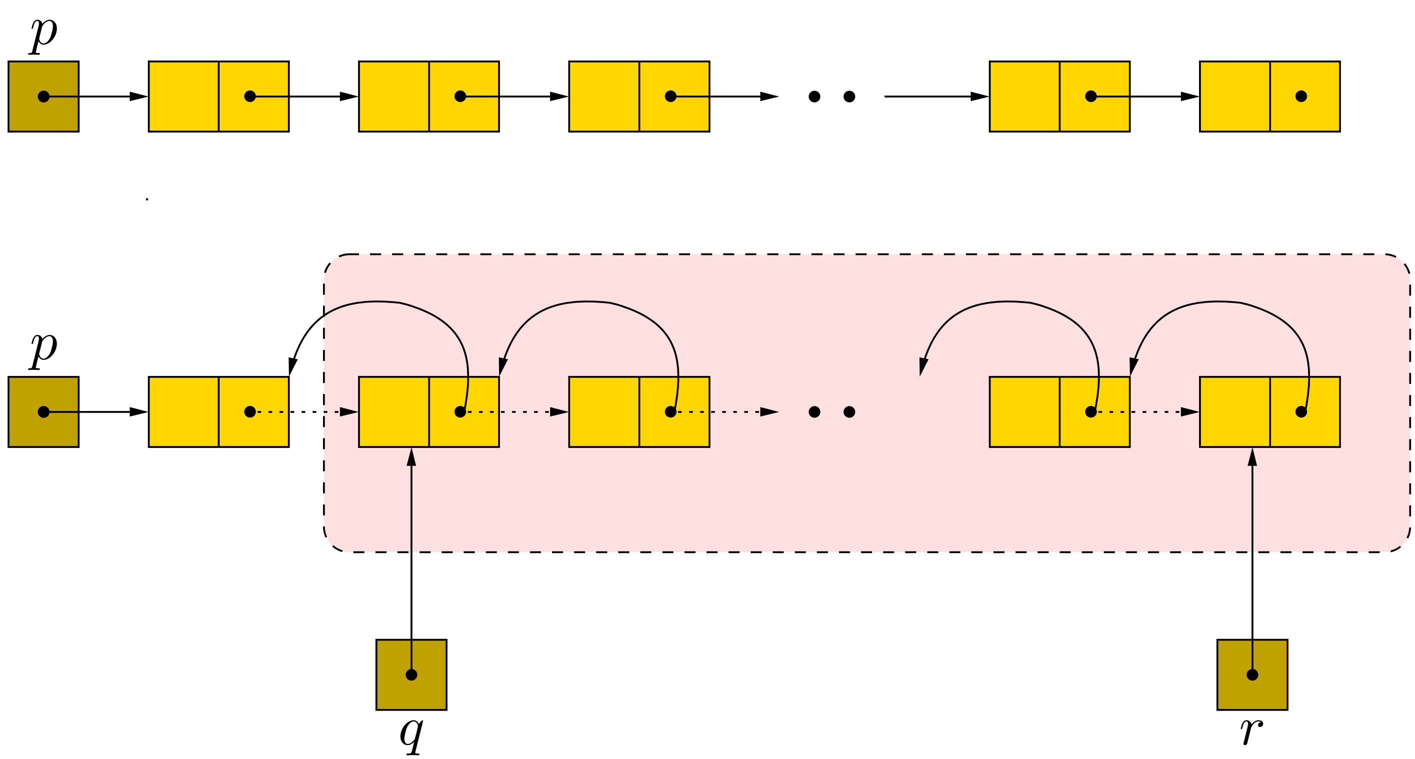


$$next[q] \leftarrow next[next[q]]$$

```
PurgeList(L)
    ▷ همه‌ی عناصر یکسان را جز یکی حذف می‌کند
p ← First(L)
while p <> null
    do q ← p
       while next[q] <> null
           do if element[p] = element[next[q]]
                then Delete-After(L, q)
                else q ← next[q]
       p ← next[p]
```

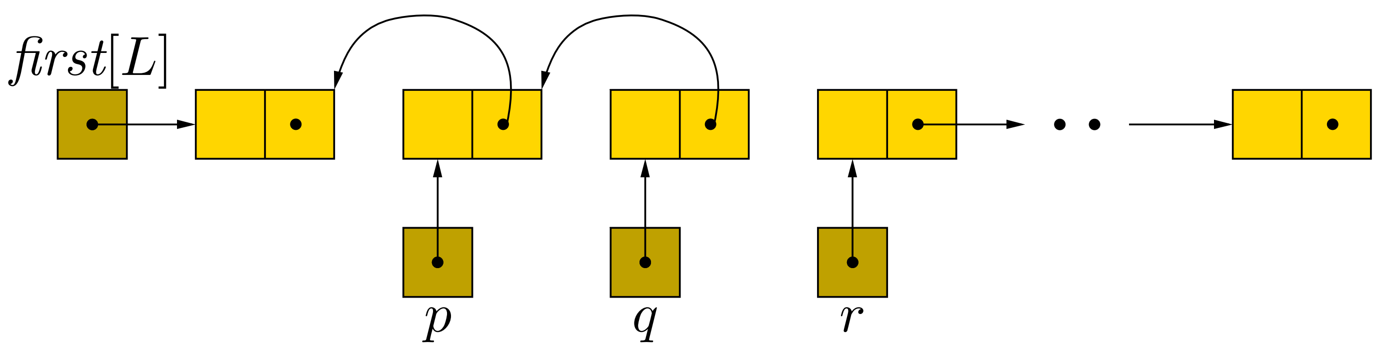
آیا می‌توان این کار را در $\mathcal{O}(n \lg n)$ انجام داد؟

وارون کردن یک لیست با تغییر اشاره‌گرها



```
RECURSIVE-REVERSE(L, p)
    ▷ لیست L را از عنصر p به بعد وارون می‌کند و حاصل را برمی‌گرداند.
if p = null  or next[p] = null
  then return p
q ← next[p]
r ← RECURSIVE-REVERSE (L, q)
next[q] ← p
next[p] ← null
return r
```

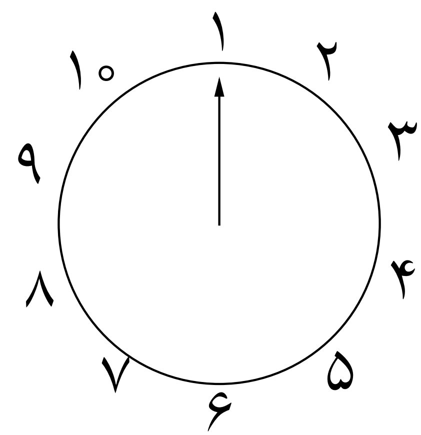
وارون کردن یک لیست به‌صورت غیربازگشتی



```
NR-Reverse(L)
if Size(L) ≤ 1
  then return first[L]
p ← null
q ← First(L)
r ← next[q]
while r ≠ null
      do next[q] ← p
         p ← q
         q ← r
         r ← next[r]
next[q] ← p
return q
```

مسئله‌ی ژوزفوس

اگر n نفر با شماره‌های ۱ تا n دور دایره‌ای قرار بگیرند و با شروع از شماره‌ی ۱ و در جهت ساعت‌گرد هر بار دومین (یا k امین) نفر خودش را بکشد، آخرین نفر چه شماره‌ای دارد؟



مسئله‌ی ژوزفوس با ۱۰ نفر.

برای $n = ۱۰$ به‌ترتیب افراد ۲، ۴، ۶، ۸، ۱۰، ۳، ۷، ۱، ۹ خودکشی می‌کنند و ۵ زنده می‌ماند.

جواب این مسئله $J(n)$ به‌صورت ریاضی قابل محاسبه است و می‌توان جواب را از رابطه‌ی بازگشتی زیر به‌دست آورد.

$$\begin{array}{lcl} J(۱) & = & ۱ \\ J(۲n) & = & ۲J(n) - ۱, \ \text{for } n \geq ۱, \\ J(۲n+۱) & = & ۲J(n) + ۱ \quad \text{for } n \geq ۱. \end{array}$$

اگر n را به‌صورت عدد دودویی بنویسیم و آن‌را یک بیت شیفت چپ دورانی دهیم $J(n)$ به‌دست می‌آید.

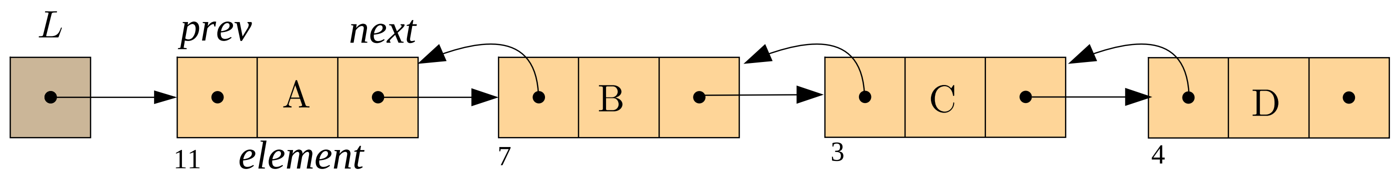
مثلاً برای $(۱۱۰۰۱۰۰)_۲ = ۱۰۰ = n$، جواب $۷۳ = (۱۰۰۱۰۰۱)_۲ = J(n)$ است.

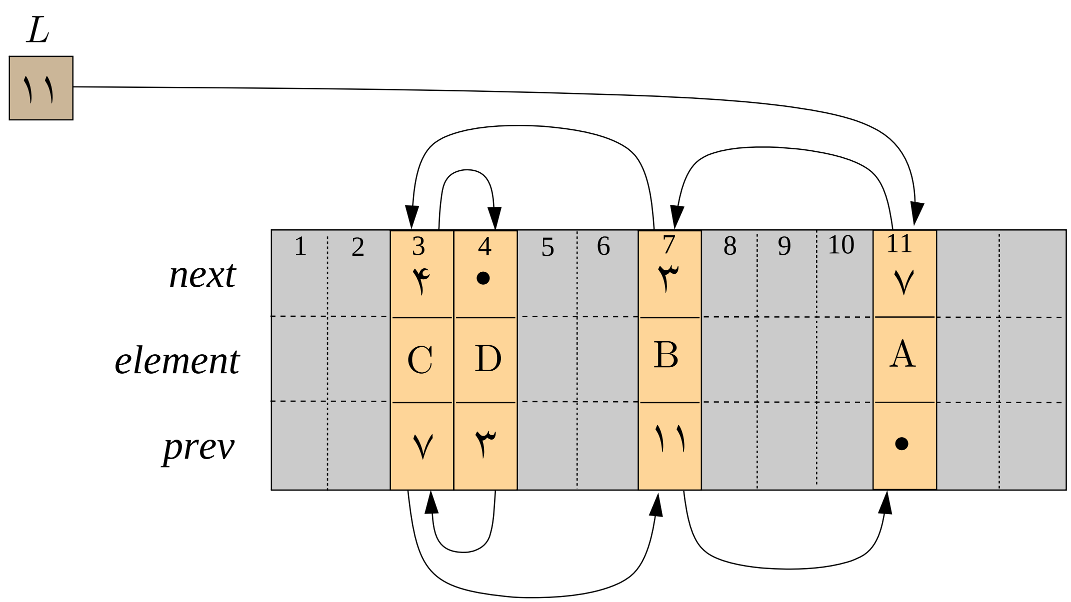
حل مسئله‌ی ژوزفوس با لیست پیوندی حلقه‌ای

```
JOESEPHOUS (n)
    ▷ در ابتدا یک لیست پیوندی حلقوی با n گره ایجاد می‌کند
CREATE(L)
first[L] ← q ← ALLOCATE-NODE(1, null )
p ← q
for i ← 2 to n
    do next[p] ← ALLOCATE-NODE(i, next[p])
       p ← next[p]
next[p] ← q;   size[L] ← n
    ▷ و حالا راه‌حل کُند مسئله‌ی ژوزفوس
p ← FIRST(L)
while next[p] ≠ p
       do DELETE-AFTER(L, p)
          p ← next[p]
PRINT element[p]
```

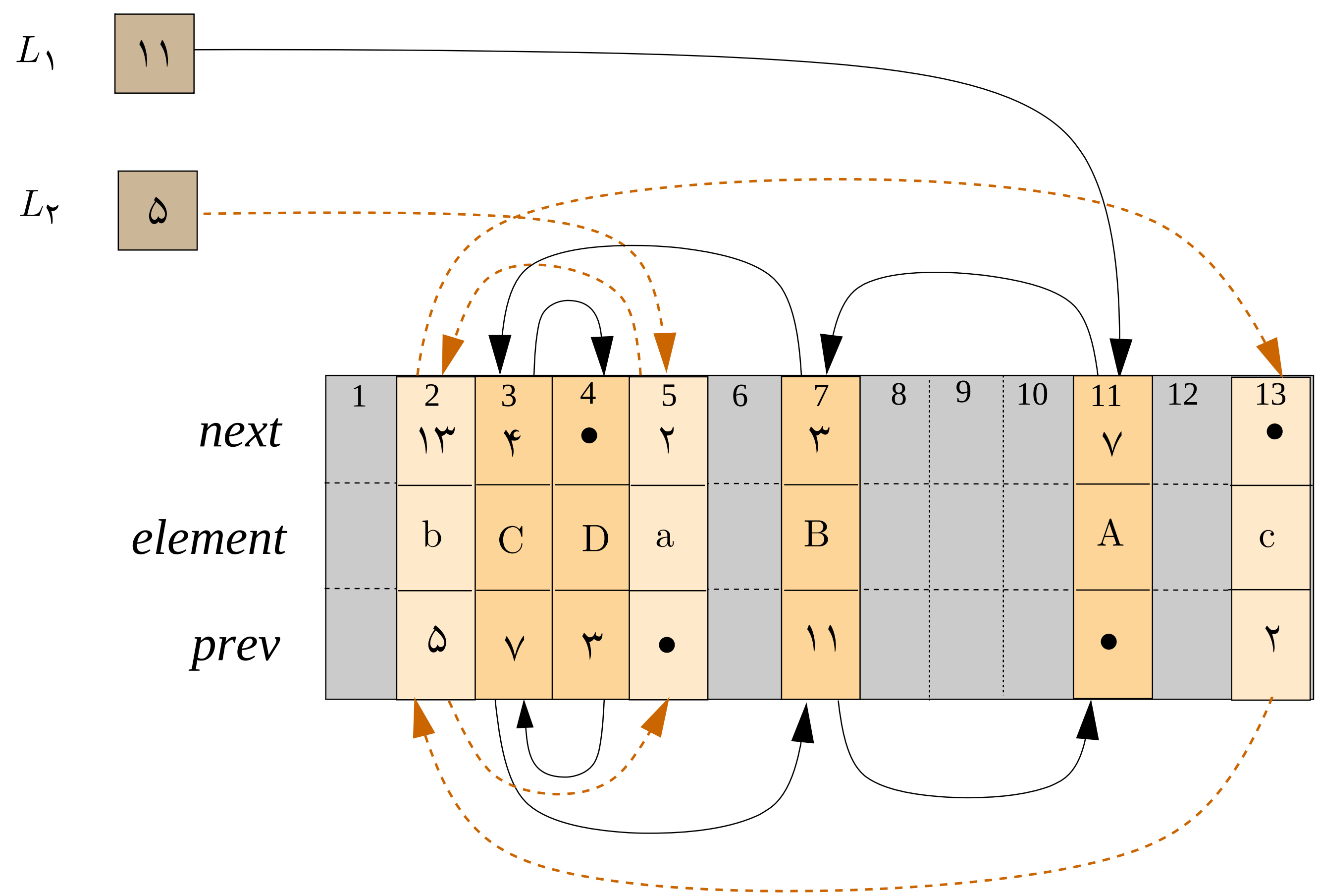
پیاده‌سازی لیست‌ها با اشاره‌گرهای اندیسی

یا زبان فرترن لیست‌ها را چه‌گونه پیاده‌سازی می‌کنیم؟

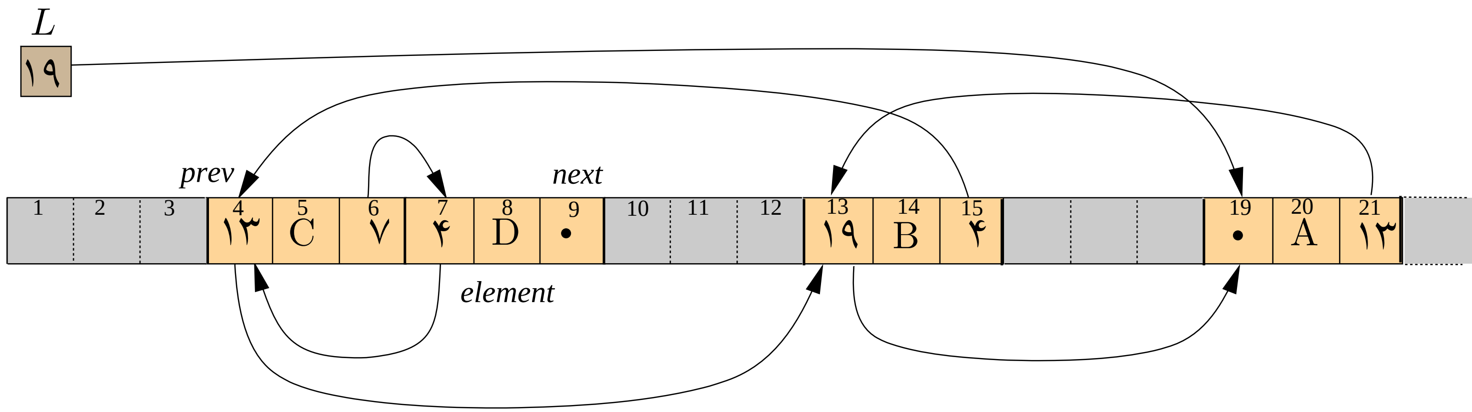




سه آرایه‌ی *next*، *element* و *prev*. اشاره‌گرها اندیس هستند.



بیش از یک لیست (مثلاً $L_۱$ و $L_۲$) را می‌توان در همان حافظه پیاده‌سازی کرد.

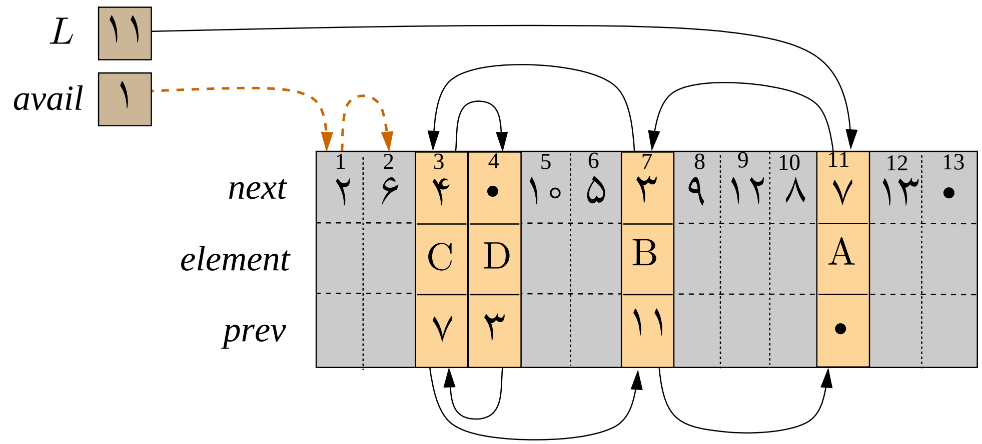


می‌توان فقط از یک آرایه برای پیاده‌سازی یک لیست استفاده کرد.

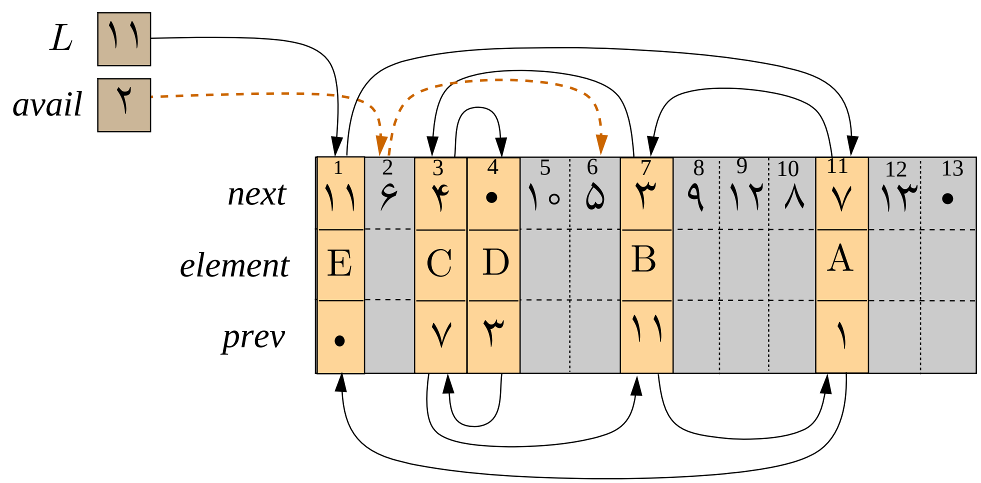
مدیریت فضای آزاد

- همه‌ی عناصر آزاد در لیستی به‌نام *avail* قرار می‌گیرند.
- در ابتدا همه‌ی عناصر آرایه‌ها آزادند.
- (عمل Allocate-Object) برای درج، اولین عنصر لیست آزاد از *avail* حذف شده در اختیار قرار می‌گیرد
- (عمل Free-Object) عنصر با درج در ابتدای لیست *avail* آزاد می‌شود.

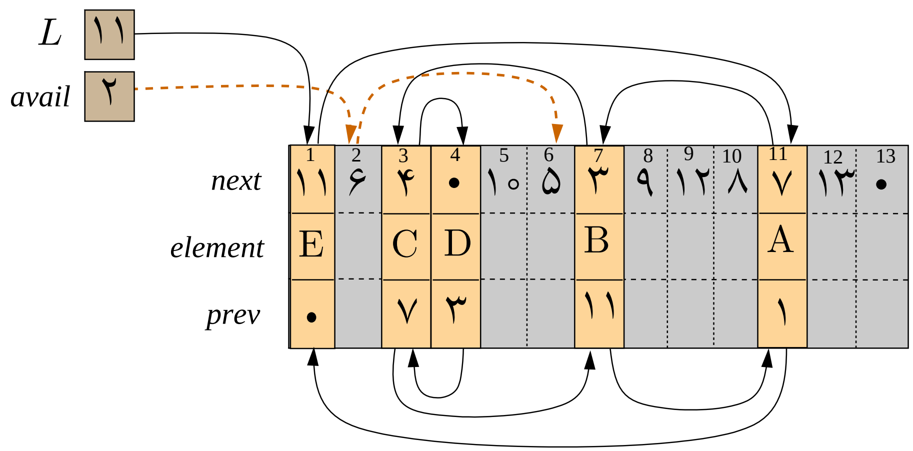
مثالی از درج و حذف در یک لیست با اشاره‌گر اندیسی



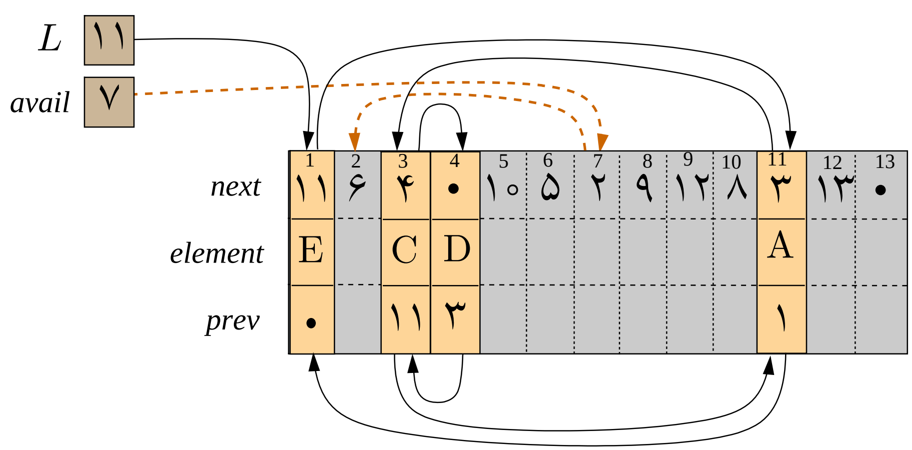
(الف) لیست اصلی



(ب) پس از درج **E** به‌عنوان عنصر اول لیست (الف)



(ب) پس از درج **E** به‌عنوان عنصر اول لیست (الف)



(پ) پس از حذف عنصر **B** در لیست (ب)

پیاده‌سازی

```
INITIALIZE()
null ← 0
avail ← 1
for i ← 1 to M − 1
    do next[i] ← i + 1
next[M] ← null
```

پیاده‌سازی (ادامه)

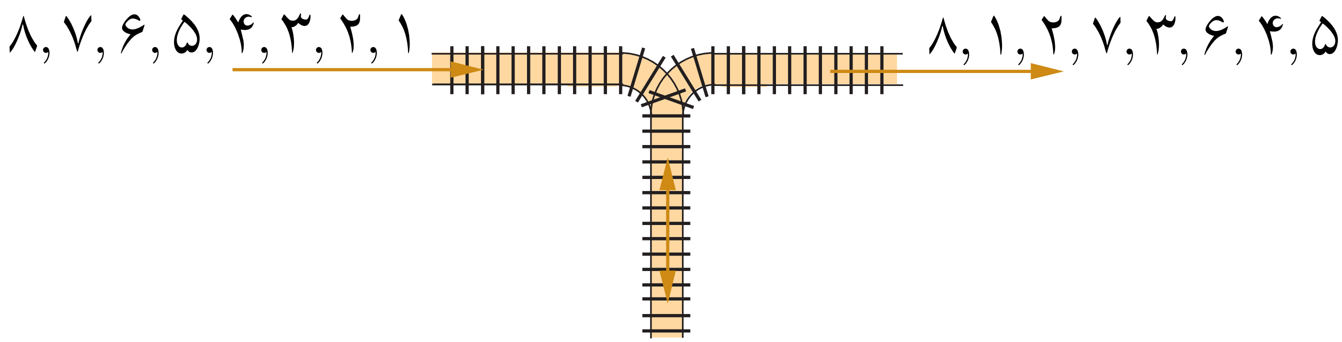
```
ALLOCATE-OBJECT()
1  if avail = null
2     then error out of space
3  x ← avail
4  avail ← next[avail]
5  return x
```

```
FREE-OBJECT(x)
1  next[x] ← avail
2  avail ← x
```

زباله‌روبی (Garbage Collection)

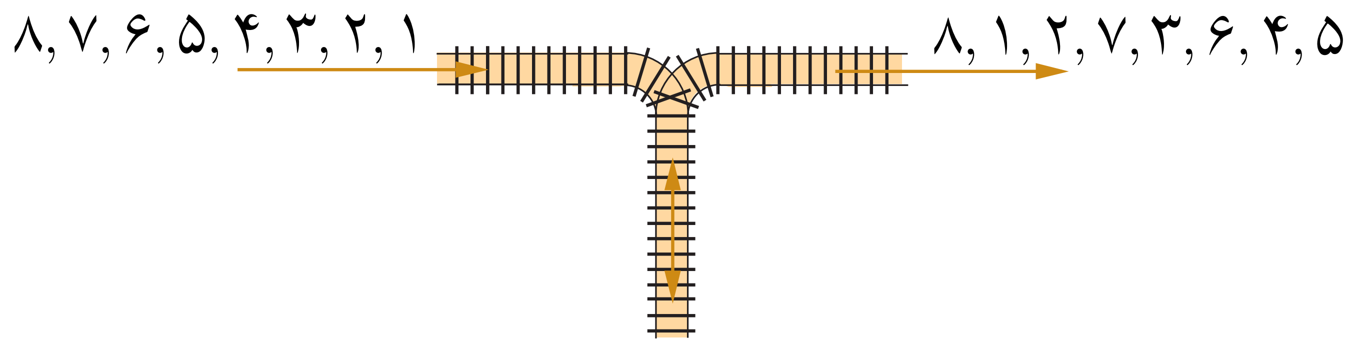
پشته‌ها

- $\textsc{Push}(S, x)$: **درج یک عنصر x در بالای پشته‌ی S**
  ورودی: عنصر (شیئ)، خروجی: هیچ
- $\textsc{Pop}(S)$: **حذف و بازگرداندن عنصر بالای پشته**
  ورودی: هیچ، خروجی: عنصر (شیئ)، خطا: اگر پشته خالی باشد
- $\textsc{Size}(S)$: **تعداد عناصر موجود در پشته**
  ورودی: هیچ، خروجی: یک عدد صحیح
- $\textsc{isEmpty}(S)$: **مشخص می‌کند که آیا پشته خالی است**
  ورودی: هیچ، خروجی: درست یا نادرست
- $\textsc{Top}(S)$: **عنصر بالای پشته را برمی‌گرداند**
  ورودی: هیچ، خروجی: عنصر (شیئ)، خطا: اگر پشته خالی باشد



پشته‌ی قطارها.

⟨۸, ۱, ۲, ۷, ۳, ۶, ۴, ۵⟩ **قابل تولید است. چه‌طور؟**



پشته‌ی قطارها.

⟨۸, ۱, ۲, ۷, ۳, ۶, ۴, ۵⟩ قابل تولید است. چه‌طور؟

Push, Push, Push, Push, Push, Pop, Pop, Push, Pop, Pop, Push, Pop, Pop, Pop, Push, Pop

⟨۱, ۸, ۳, ۶, ۲, ۷, ۴, ۵⟩ چه‌طور؟

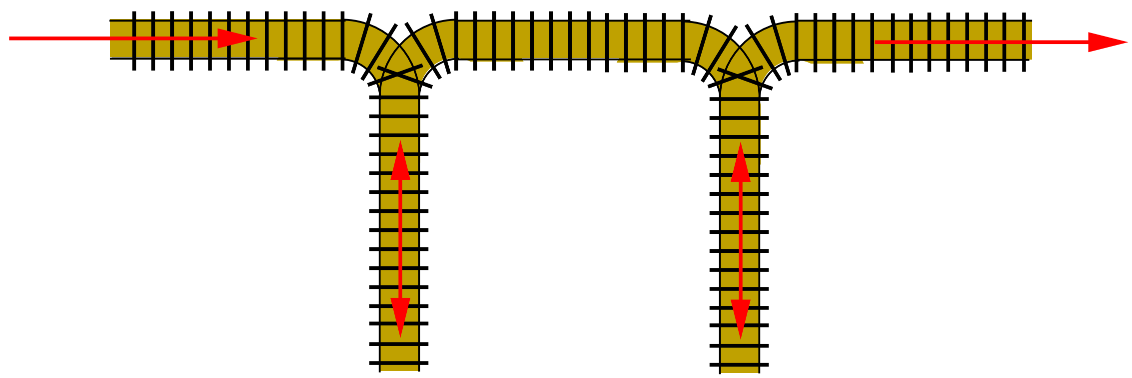
چند مسئله

۱) شرط لازم و کافی برای یک دنباله که قابل تولید باشد چیست؟ آن‌را اثبات کنید.

۲) الگوریتمی از $\mathcal{O}(n)$ ارائه دهید تا قابل تولید بودن دنباله‌ای را تشخیص دهد.

۳) فرض کنید یک ریل مستقیم هم بین قطارهای ورودی و خروجی وجود دارد؛ یعنی اولین قطار ورودی می‌تواند یا به داخل پشته رود و یا مستقیماً به ریل خروجی منتقل شود، و یا برعکس از خروجی به ورودی. در این صورت الگوریتم تشخیص یک دنباله‌ی قابل تولید را ارائه دهید.

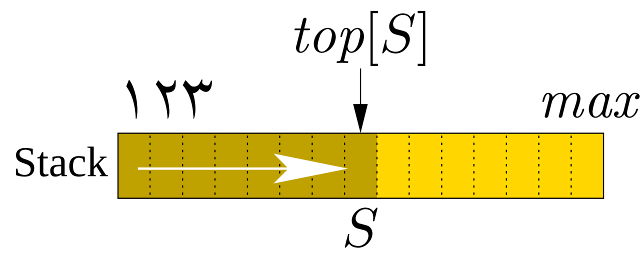
۴) مسئله‌ی اصلی را برای سیستم دو پشته‌ای مطابق شکل حل کنید.



پیاده‌سازی پشته با آرایه

آرایه‌ی S با اندازه‌ی حداکثر max و مؤلفه‌ی $top[S]$ که اندیس بالاترین عنصر موجود در S است.

```
public class ArrayStack implements Stack {
    public static final int CAPACITY=1000;
    private int capacity;
    private object S[];
    private int top = -1;
    public ArrayStack(){
       this(CAPACITY);
    }
    public ArrayStack(int cap){
       capacity = cap;
       S = new object[capacity];
    }
```



پیاده‌سازی پشته با آرایه.

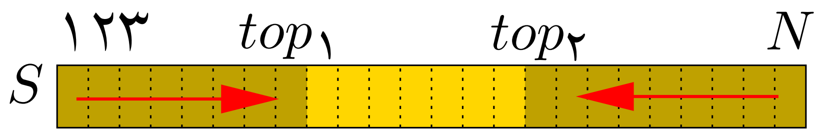
```
Size(S)
    return top[S]    ▷ assuming that initially top[S] = 0

isEmpty(S)
    return Size(S) = 0

Top(S)
    if isEmpty(S)
      then error ("stack is empty")
    return S[top[S]]
```

<u>PUSH</u> (S, x)

if SIZE$(S) = max$
  then **error** ("stack is full")
$top[S] \leftarrow top[S] + 1$
$S[top[S]] \leftarrow x$

<u>POP</u> (S)

if isEmpty()
  then **error** ("stack is empty")
$e \leftarrow S[top[S]]$
$top[S] \leftarrow top[S] - 1$
return e



پیاده‌سازی چند پشته با آرایه.

تحلیل

همه‌ی اعمال در $O(۱)$ انجام می‌شوند.

پیاده‌سازی پشته با لیست پیوندی

```
SIZE (S)
    return  size[S]

ISEMPTY (S)
    return  (size[S] = 0)

TOP (S)
    if isEmpty(S)
      then  error ("stack is empty")
    return  top[S]
```

```
PUSH (S, x)
  top[S] ← ALLOCATE-NODE (x, top[S])
  size[S] ← size[S] + 1
```

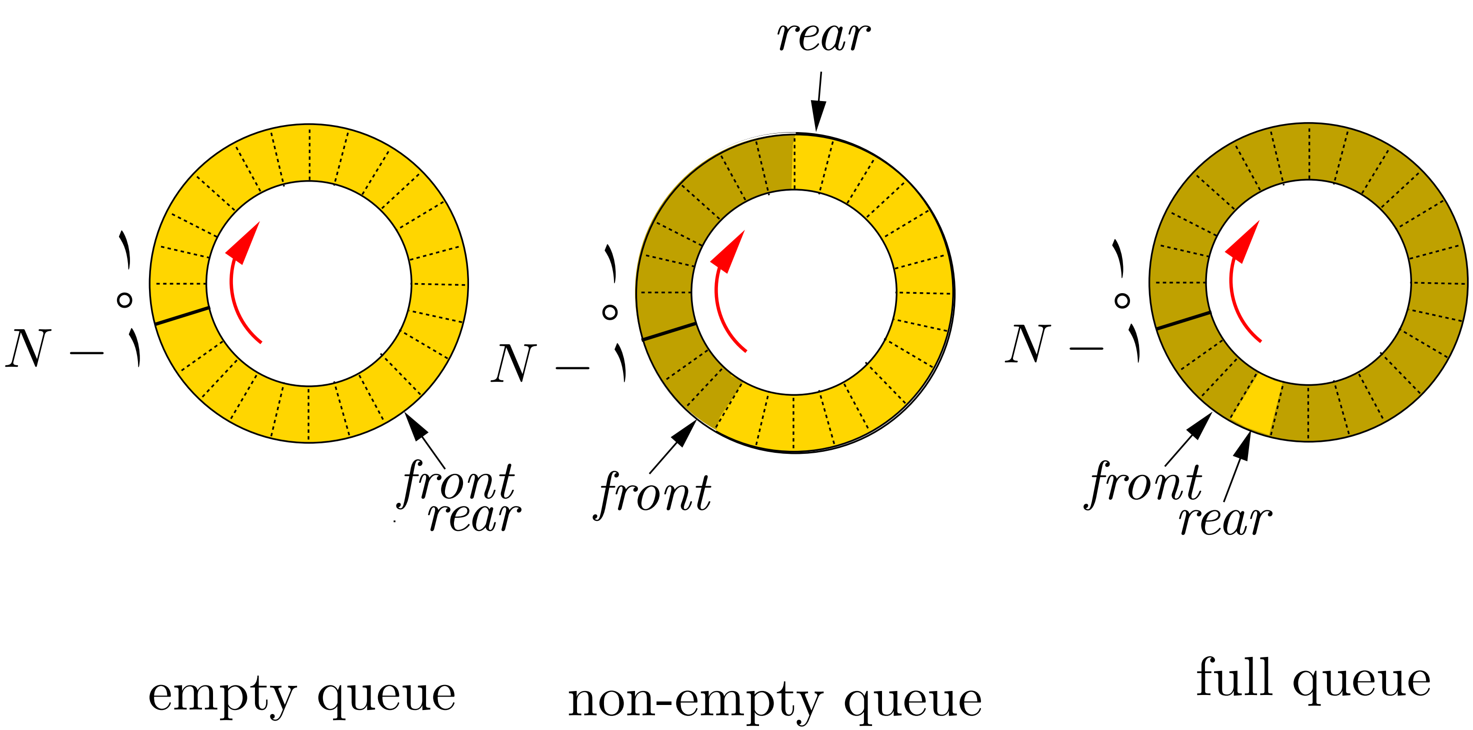
```
POP (S)
  if ISEMPTY(S)
    then error ("STACK IS EMPTY")
  n ← top[S]
  temp ← element[n]
  top[S] ← next[n]
  size[S] ← size[S] − 1
  FREE-OBJECT(n)
  return temp
```

صف

درج در انتها و حذف در ابتدای لیست

- $\textsc{Enqueue}(Q, x)$: درج یک عنصر در انتهای صف
ورودی: عنصر (شیئ)، خروجی: هیچ
- $\textsc{Dequeue}(Q)$: حذف عنصر از ابتدای صف
ورودی: هیچ، خروجی: عنصر (شیئ)، خطا: اگر خالی باشد
- $\textsc{Size}(Q)$: تعداد عناصر موجود در صف
ورودی: هیچ، خروجی: یک عدد صحیح
- $\textsc{isEmpty}(Q)$: مشخص می‌کند که آیا صف خالی است
ورودی: هیچ، خروجی: درست یا نادرست
- $\textsc{Front-Element}(Q)$: عنصر ابتدای صف را برمی‌گرداند
ورودی: هیچ، خروجی: عنصر، خطا: اگر خالی باشد

پیاده‌سازی با آرایه‌ی دوار



یک صف Q

- **یک آرایه با اندازه‌ی max و اندیس‌های ۰ تا $max - ۱$**
- **عناصر به‌صورت دوار و در جهت ساعت‌گرد ذخیره می‌شوند.**
- $Q[(i + ۱) \bmod max]$ **عنصر بعدیِ** $Q[i]$ **است.**
- **مولفه‌ی** $front[Q]$ **اندیس عنصر ابتدایی صف**
- $rear[Q]$ **اندیس عنصر بعدی آخرین عنصر صف.**
- **بنابراین حداکثر تعداد عناصر $max - ۱$ است.**
- **می‌خواهیم دو حالت «کاملاً پر» و «کاملاً خالی» را بتوانیم از هم تمیز دهیم.**

حالت‌های مختلف صف

- در شروع $front[Q] = rear[Q] = \circ$
- تعداد عناصر همیشه برابر $(max - front[Q] + rear[Q]) \bmod max$.
- اگر صف کاملاً خالی باشد داریم $front[Q] = rear[Q]$.
- اگر کاملاً پر باشد داریم $(max - front[Q] + rear[Q]) \bmod max = max - ۱$.

Size(Q)

return $(max - front[Q] + rear[Q]) \bmod max$

isEmpty(Q)

return $(front[Q] = rear[Q])$

Front-Element(Q)

if isEmpty(Q)
  then **error** "Queue is empty"
return $Q[front[Q]]$

<u>ENQUEUE</u> (Q, x)

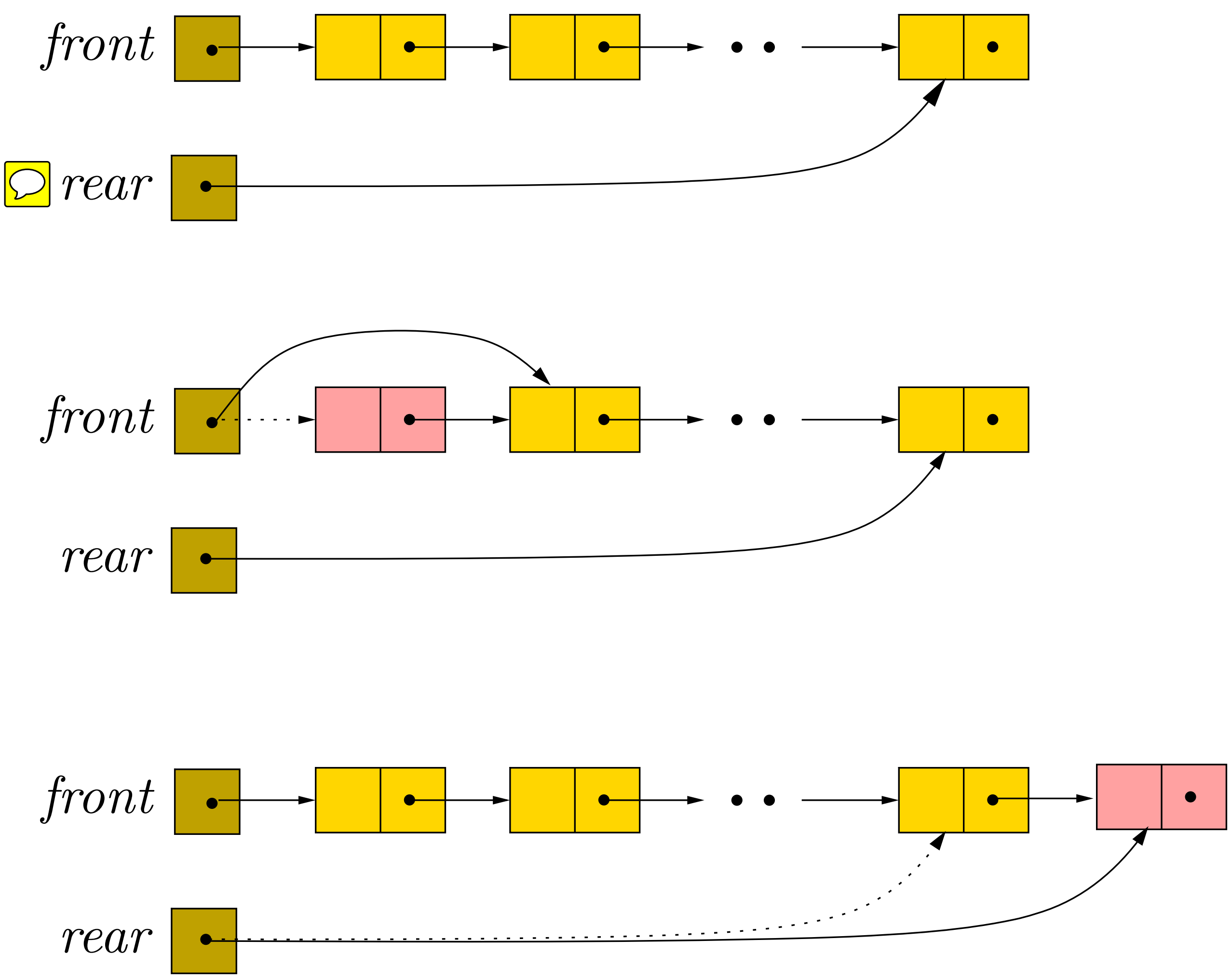
if Size$(Q) = max - 1$
  then **error** "Queue is full"
$Q[rear[Q]] \leftarrow x$
$rear[Q] \leftarrow (rear[Q] + 1)$ mod max

<u>DEQUEUE</u> (Q)

if isEmpty()
  then error "Queue is empty"
$temp \leftarrow Q[front[Q]]$
$front[Q] \leftarrow (front[Q] + 1)$ mod max
return $temp$

این اعمال همه از $O(۱)$ هستند.

پیاده‌سازی صف با لیست پیوندی


front
rear
front
rear
front
rear

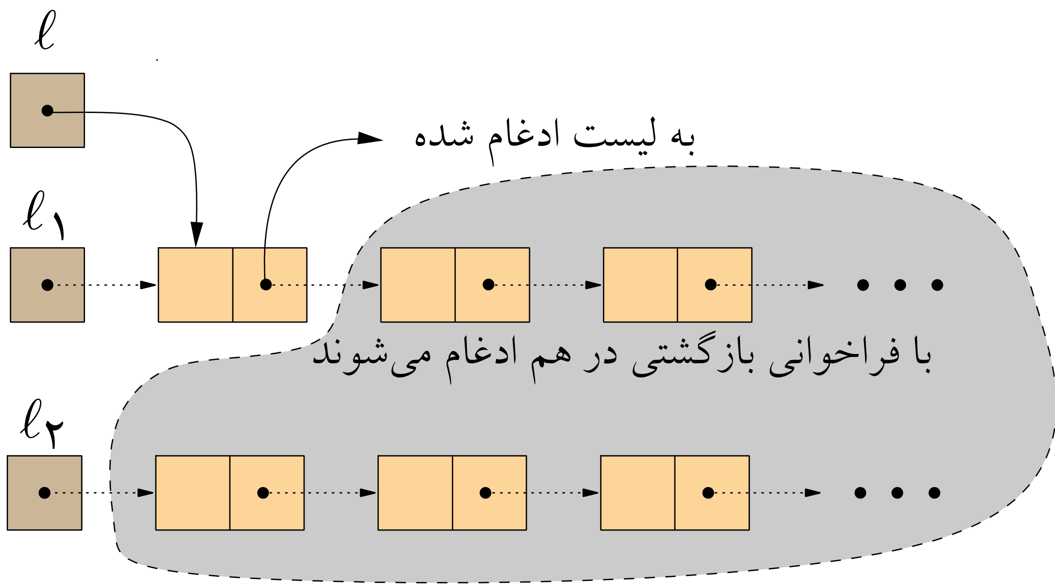
```
isEmpty(Q)
    return size[Q] = 0
```

```
Enqueue(Q, x)
    next[rear[Q]] ← Allocate-Node(x, null)
    rear[Q] ← next[rear[Q]]
    size[Q] ← size[Q] + 1
```

```
DEQUEUE(Q)
if ISEMPTY(Q)
  then error 'QUEUE IS EMPTY'
n ← front[Q]
x ← element[n]
front[Q] ← next[front[Q]]
FREE-NODE(n)
size[Q] ← size[Q] − 1
return x
```

مرتب‌سازی ادغامی با لیست

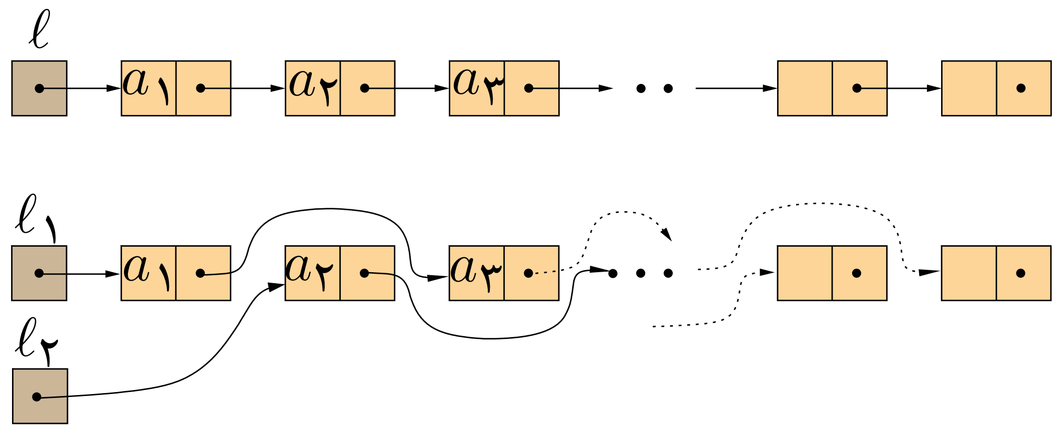
```
MergeSort (L)
      ▷ یک لیست پیوندی را فقط با تغییر اشاره‌گرها مرتب می‌کند
if  Size(L) > 1
  then Create(L1);  Create(L2)
         first[L1], first[L2] ← Split(first[L])
         MergeSort (L1)
         MergeSort (L2)
         first[L] ← Merge(first[L1], first[L2])
         return L
```



ادغام دو لیست مرتب با عناصر اول ℓ_1 و ℓ_2 و تولید یک لیست مرتب با عنصر اول ℓ.

$\underline{\textsc{Merge}}\,(\ell_1, \ell_2)$

if $\ell_1 =$ **null**
    then return ℓ_2
if $\ell_2 =$ **null**
    then return ℓ_1
if $key[\ell_1] \leq key[\ell_2]$
    then $next[\ell_1] \leftarrow \textsc{Merge}(next[\ell_1], \ell_2)$
            return ℓ_1
    else $next[\ell_2] \leftarrow \textsc{Merge}(\ell_1, next[\ell_2])$
            return ℓ_2



تقسیم یک لیست n عضوی به دو لیست $\lceil \frac{n}{2} \rceil$ و $\lfloor \frac{n}{2} \rfloor$ عضوی.

$\underline{\textsc{Split}}\,(\ell)$

```
if ℓ = null
  then return null , null
if next[ℓ] = null
  then return null , ℓ
ℓ1 ← ℓ
ℓ2 ← next[ℓ]
next[ℓ], next[ℓ2] ← SPLIT(next[ℓ2])
return ℓ1, ℓ2
```

لیست‌های کلی

- یک چندجمله‌ای در حالت کلی
- جمع عبارت‌های از نوع $cx^{e_x}y^{e_y}z^{e_z}\ldots$ است که
- c ضریب این عبارت و e_x، e_y، e_z، ... به‌ترتیب ضرایب توان متغیرهای x، y، z و ... هستند. مثلاً

$$P = x^{10}y^{3}z^{2} + 2x^{8}y^{3}z^{2} + 3x^{8}y^{2}z^{2} + x^{4}y^{4}z + 6x^{3}y^{4}z + 6x^{3}y^{4}z + 2yz \quad (1)$$

هدف طراحی داده‌ساختار مناسب با اعمال زیر:

-- چاپ عبارت

-- تعیین بیش‌ترین عمق آن

-- کپی کردن یک عبارت

-- جمع یا تفریق دو عبارت

-- مشتق‌گیری از عبارت برحسب یکی از متغیرها

روش اول: یک لیست با عناصر زیر

coef	*expx*	*expy*
expz	*link*	

لیست کلی با ساختار بازگشتی

اگر $P(z, y, x, n_z, n_y, n_x)$ یک چند جمله‌ای برحسب z, y, x, n_z, n_y, n_x،

- متغیرها به‌ترتیب z، y و x و
- درجه‌ی آن‌ها به‌ترتیب برابر n_z، n_y و n_x باشند،

آن‌را به‌صورت زیر تعریف می‌کنیم:

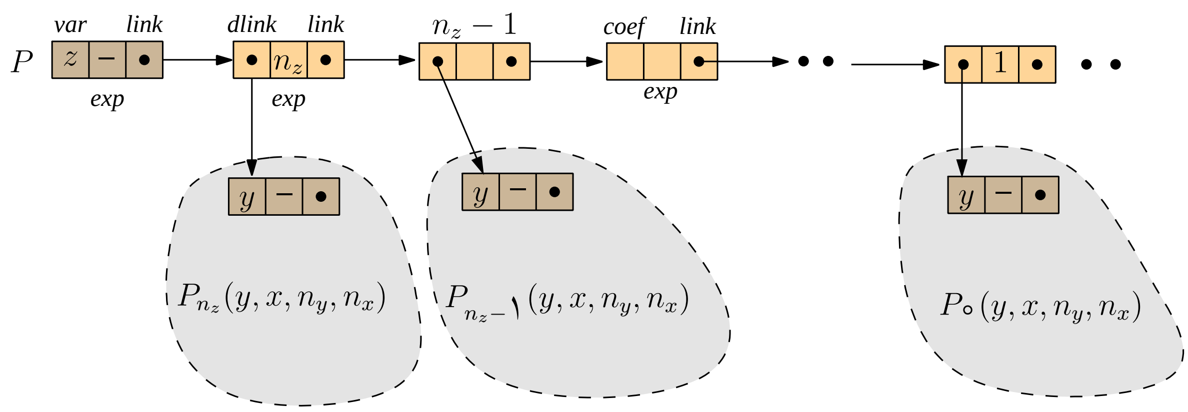
$$P(z, y, x, n_z, n_y, n_x) =$$

$$c_{n_z}(P_{n_z}(y, x, n_y, n_x)z^{n_z} + c_{n_z-1}(P_{n_z-1}(y, x, n_y, n_x)z^{n_z-1} +$$

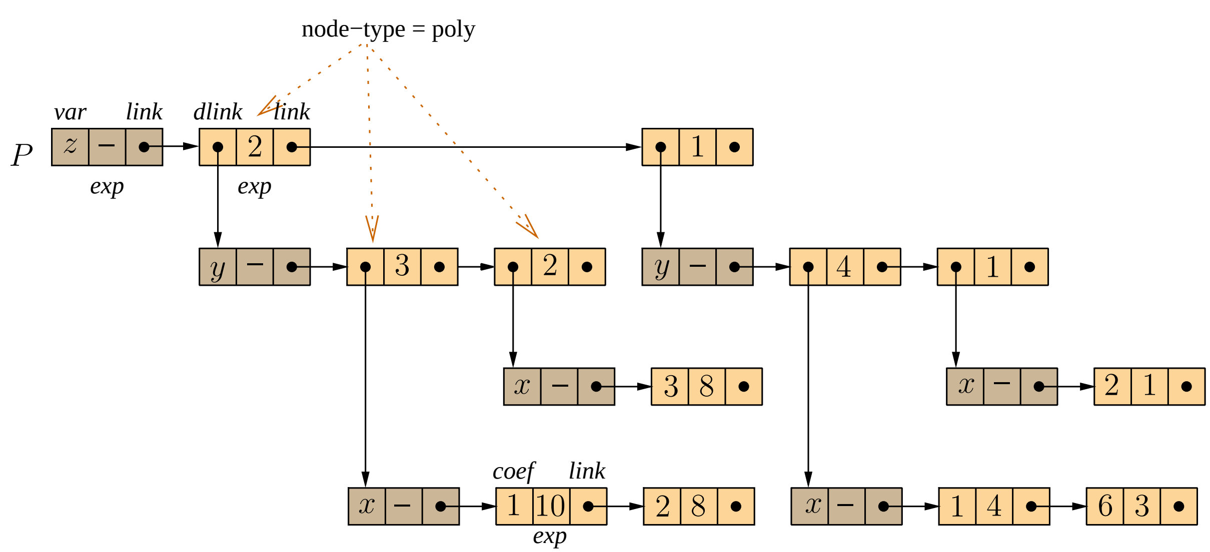
$$\ldots + c_0(P_0(y, x, n_y, n_x))$$

مثال

$$(((x^{10} + 2x^{8})y^{3} + 3x^{8}y^{2})z^{2} + ((x^{4} + 6x^{3})y^{4} + 2y)z)$$



$$P(z, y, x, n_z, n_y, n_x) = P_{n_z}(y, x, n_y, n_x)z^{n_z} + P_{n_z-1}(y, x, n_y, n_x)z^{n_z-1} + \cdots + P_0(y, x, n_y, n_x)$$



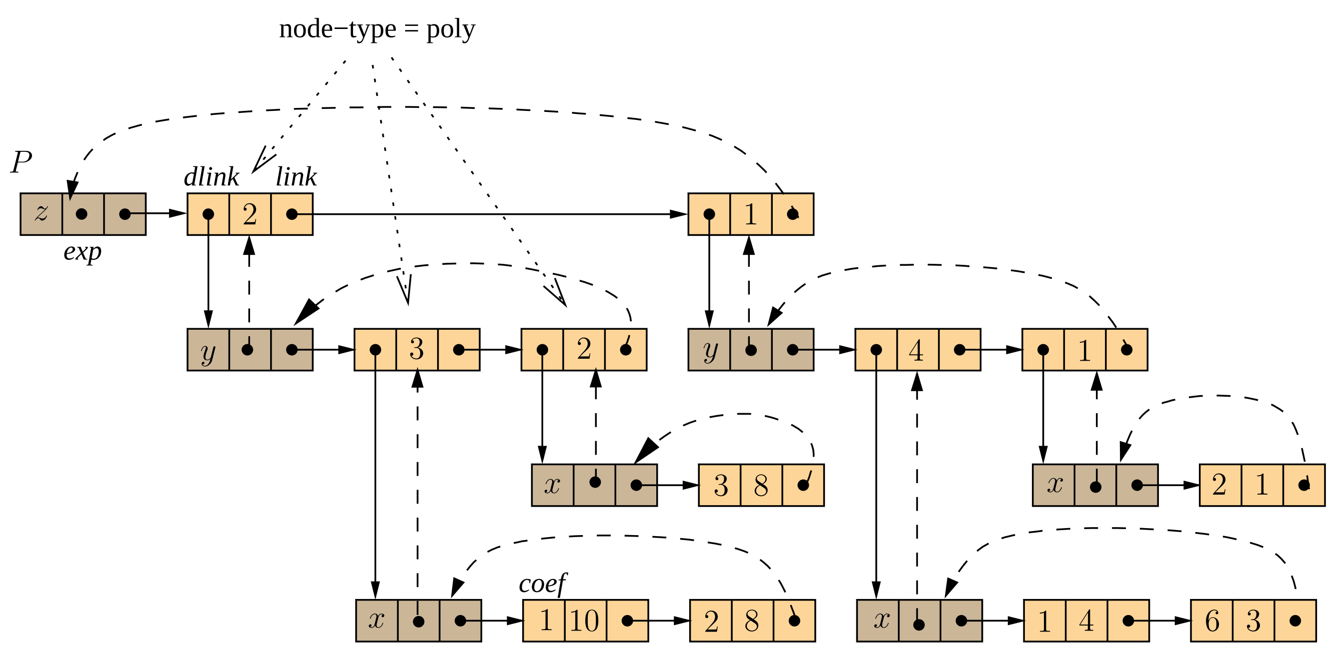
$$((x^{10} + 2x^{8})y^{3} + 3x^{8}y^{2})z^{2} + ((x^{4} + 6x^{3})y^{4} + 2y)z.$$

اعمال

```
PRINT-PLIST (P)
      ▷ فرض می‌کنیم که لیست درست ساخته شده است
p ← P
X = var[p] ▷ P متغیر اصلی چندجمله‌ای
while p ≠ null
      do if NODE-TYPE(p) = poly
            then PRINT '('
                  PRINT-PLIST (dlink[p])
                  PRINT ') ^ exp[p]'
            else
                  ▷ با فرض درستی داده‌گونه‌ها
                  PRINT '+coef[p] X^exp[p]'
         p ← link[p]
```

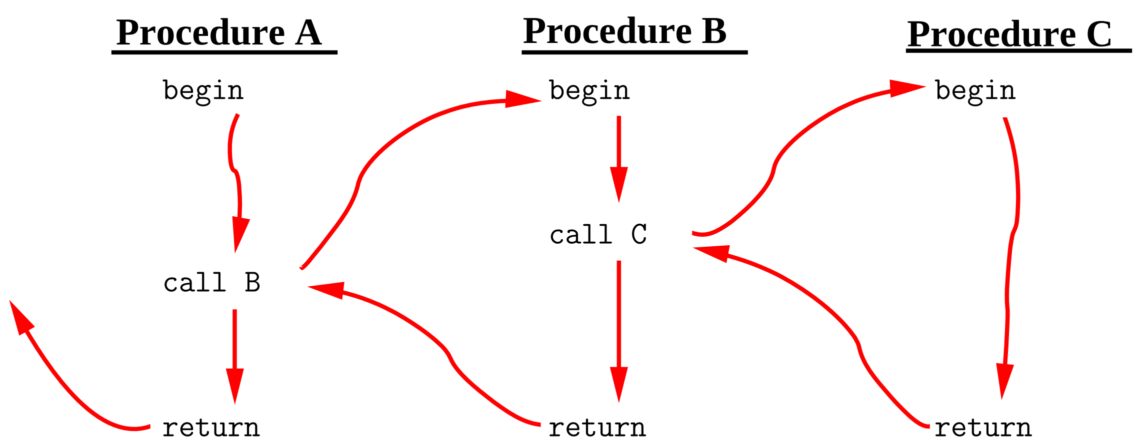
```
DEPTH-PLIST (P)
    ▷ فرض می‌کنیم که لیست درست ساخته شده است
p ← P
depth ← 0
while p ≠ null
      do if NODE-TYPE(p) = poly
            then dp ← DEPTH-PLIST (dlink[p]) + 1
                 depth ← max{depth, dp}
         p ← link[p]
return DEPTH
```

نخ (thread) و اشاره‌گر



```
DEPTH-THREADED-PLIST(P)
p ← P
depth ← 0
maxdepth ← 0
while true
      do if Node-Type[p] = poly
            then depth ← depth + 1
                 p ← dlink[p]
            else while link[p] is a thread
                        do p ← link[p]
                           maxdepth = max{maxdepth, depth}
                           if exp[p] ≠ null
                             then p ← exp[p]
                                  depth ← depth − 1
                             else return maxdepth
            if link[p] is pointer
              then p ← link[p]
```

تبدیل الگوریتم‌های بازگشتی به غیربازگشتی



انتقال کنترل برنامه در فراخوانی و بازگشت

مراحل

۱) عمل فراخوانی

۲) بازگشت از یک فراخوانی

هر فراخوانی (Call)

۱) ذخیره‌ی کلیه‌ی متغیرهای محلی (در حالت کلی کلیه متغیرهای دسترس‌پذیر) و مقدارهایشان در پشته‌ی سیستم (Push).

۲) آدرس بازگشت به پشته منتقل می شود (Push).

۳) عمل انتقال پارامترها (Parameter Passing) صورت می‌گیرد. پارامترها ممکن است از نوع ارزشی (Val) یا آدرسی (Variable) باشند.

۴) کنترل برنامه (ثبات شمارنده‌ی برنامه، Program Counter) به ابتدای رویه‌ی جدید اشاره می‌کند.

عمل بازگشت (Return)

عکس عملیات فوق

۱) مقدارهای متغیرهای محلی را از رکورد بالای پشته برداشته و در خودشان قرار می‌دهیم.

۲) آدرس بازگشت را از بالای پشته به دست می‌آوریم.

۳) آخرین رکورد را از پشته برمی‌داریم (Pop).

۴) کنترل برنامه را از آدرس بازگشت (بند ۲) ادامه می‌دهیم.

مثال

<u>HONOI</u> (n, f, t, h)

▷ moving n coins from leg f to leg t with the help of leg h

if $n = 1$

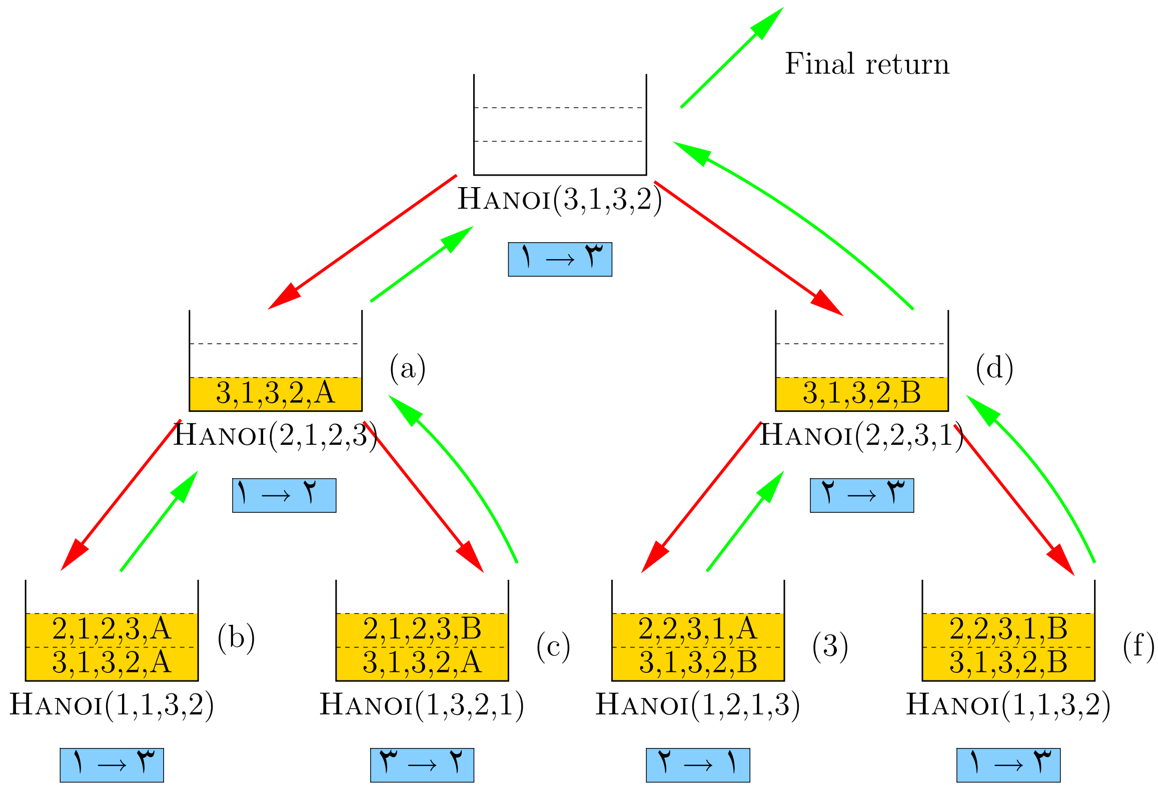
  then Move the coin from leg f to leg t

  else HONOI$(n-1, f, h, t)$

    A: Move the coin from leg f to leg t

      HONOI$(n-1, h, t, f)$

    B:



مراحل مختلف فراخوانی‌های بازگشتی و مقادیر پشته‌ی سیستم در HONOI(3,1,3,2).

```
NonRecursive-Honoi (n, f, t)
  Create-Stack(S) ▷ پشته‌ی S شامل آدرس بازگشت و مقادیر همه‌ی متغیرهای محلی است
  h ← the other peg
  Rec-Call: ▷ (آغاز یک فراخوانی بازگشتی) if n = 1
    then Print f ⟶ t ▷ سکه‌ی بالایی را از میله‌ی f به میله‌ی t منتقل کن
          goto Return-Label
    else Push (S, StackRec(n, f, t, h, 'A'))
          n, f, t, h ← n − 1, f, h, t ▷ انتقال پارامترها با فرض ارزشی بودن
          goto Rec-Call
  Return-Label: if not isEmpty(S)
    then ▷ از این دستور عمل بازگشت شبیه‌سازی می‌شود
          n, f, t, h, return-addrs ← Pop(S)
          switch
                case return-addrs = 'A'
                      do Print f ⟶ t; Push(S, SRec(n, f, t, h, 'B'))
                          n, f, t, h ← n − 1, h, t, f ▷ انتقال پارامترها
                          goto Rec-Call
                case return-addrs = 'B'
                      do goto Return-Label
```

حذف آخرین بازگشت (Tail Recursion)

آخرین فراخوانی بازگشتی که بعد از آن در هیچ شرایطی دستوری که از مقدارهای متغیرها استفاده کند، اجرا نشود را آخرین بازگشت میگوییم.

این بازگشت را می‌توان بدون استفاده از پشته حذف کرد.

```
RecursiveProc(...)
    ...
    ...
    A: RecursiveProc(...)      ▷ this is the last line
    x:
```

در بازگشت از این فراخوانی (A) متغیرهای محلی مقدارهایشان تغییر می‌کند واجرای برنامه از نقطه‌ی (x) دنبال می‌شود. ولی (x) تنها یک بازگشت است.

مثال

```
HONOI(n, f, t, h)
      ▷ moving n coins from leg f to leg t with the help of leg h
  if n = 1
    then Move the coin from leg f to leg t
    else HONOI(n − 1, f, h, t)
      A: Move the coin from leg f to leg t
         HONOI(n − 1, f, h, t)
      B:
```

حذف آخرین فراخوان

```
TOWER-OF-HONOI2 (n, f, t, h)
      ▷ eliminating the last recursion
1  if n = 1
2    then Move the coin from leg f to leg t
3    else TOWER-OF-HONOI(n − 1, f, h, t)
4         Move the coin from leg f to leg t
5         n, f, h ← n − 1, h, f      ▷ parameter passing
6          goto 1
```

```
NonRecursive-Honoi2 (n, f, t)
 1  h ← the other peg
 2  ▷ make recursive call
 3  if n = 1
 4    then
 5         Move the top coin from leg f to leg t
 6          goto 10
 7    else Push(S, StackRec(n, f, t, h))
 8         n, f, t, h ← n − 1, f, h, t      ▷ parameter passing
 9          goto 3
10  ▷ end recursive call
11  if  not isEmpty(S)
12    then n, f, t, h ← Pop(S)
13         Move the top coin from leg f to leg t
14         n, f, t, h ← n − 1, h, t, f      ▷ parameter passing
15          goto 3
```