HW4 — 810101394 — محمد سینا پرویزی مطلق

سوال ① الف) fun f(a,b,c) = if c(a) then append(a,b) else append(a, f(a-1, b, c));

C در حالت if آمده است پس خروجی آن باید bool باشد

a:u b:v c:ω

$f(a,b,c) \rightarrow Y \Rightarrow f_s(u \times v \times \omega) \rightarrow Y$

c: u → bool

همچنین طبق سوال تابع append یک لیست برمی‌گرداند (a* alist → alist)

از آنجایی که stmt های داخل then ، else ، append هستند پس خروجی آنها یک لیست است و در نهایت خروجی function ما هم یک لیست خواهد بود ⟸ Y: alist

تایپ لیست را باید پیدا کنیم

a-1 : u , int → real×real → real / int × int → int ⟸ چون قابل 1 برابر int است ← (int×int → int) ، (a: int) خواهد بود

(u = int)

(v = intlist)

(ω = int → bool)

(c: int → bool)

حال تایپ خروجی لیست هم بدست می‌آید (int × intlist → intlist) ← (b: int list)

پس تایپ Y هم برابر با intlist خواهد بود

f_s (int × intlist × (int → bool)) → intlist

تعریف f صحیح است. ✓

ب) fun f(a,b,c,d) = if b(Concat(c, a(true))) then d(c) else a(false);

a: u

b: v

c: z

d: ω

طبق گفته سوال برای Concat ⟸ (string×string → string)

b هم چون در شرط if آمده یک bool برمی‌گرداند پس ورودی b که تابع Concat است از آنجایی string و خروجی آن bool است

b: string → bool

c, a(true) نیز string خواهد بود (طبق صورت سوال)

خود a(true) یک تابع است که (bool → string)

⇓

c: string

a: bool → string

از آنجایی که a(false) هم خروجی آن string است پس خروجی d(c) هم string خواهد بود

d: string → string

c: string

در نهایت خروجی تابع f هم string خواهد بود طبق (else, then)

f_s((bool → string) × (string → bool) × string × (string → string)) → string

8) $fun\ Y(f) = (fn\ x \Rightarrow f(xx))(fn\ x \Rightarrow f(xx))$

این عبارت پیاده سازی Y-Combinator در lambda calculus است. Y یک Fixed-point Combinator است به این معنا که برای هر تابعی مثل f عبارت $Y(f)$ یک نقطه ثابت از f را برمی‌گرداند (مثلاً مقدار y مثل $(f(y)=y)$)

برای (xx): برای اینکه xx از نظر type معتبر باشد x باید تابعی باشد که آرگومانی از نوع خودش را بپذیرد.

پس اگر تایپ x را $\alpha \to \beta$ در نظر بگیریم، x به x اعمال می‌شود به این معنی است که x باید از نوع α باشد پس

$$\alpha = \alpha \to \beta$$

برای $f(xx)$: اگر تایپ xx از نوع β باشد پس f باید تابعی باشد که β را به عنوان آرگومان قبول کند. پس اگر تایپ f را $v \to \omega$ در نظر بگیریم، $v = \beta$. بنابراین f باید از نوع $\beta \to \omega$ باشد.

برای $fn\ x \to f(xx)$: تایپ این طبق موارد بالا $\alpha \to \omega$ خواهد بود. با توجه به اینکه $\alpha = \alpha \to \beta$

این عبارت معادل $(\alpha \to \beta) \to \omega$ است

حال برای $(fn\ x \to f(xx))(fn\ x \to f(xx))$ ما تایپ A را $\alpha \to \omega$ در نظر گرفتیم. عبارت ما AA است.

برای اینکه این معتبر باشد نوع آرگومان A اول از نوع A دوم باید یکی باشد پس ما داریم $(\alpha \to \omega) = \alpha$

با جایگذاری $\alpha = \alpha \to \beta$ داریم $(\alpha \to \beta) \to \omega = \alpha \to \beta$

پس $\alpha = \beta$ خواهد بود

بنابراین تایپ $fun\ Y(f)$ از نوع $(\beta \to \beta) \to \beta$ خواهد بود

چون Y بصورت $A\ A$ تعریف شده است

سوال ۲

(a→bool) x alist → alist   fun f(p, x) = List.filter p x

این تابع یک predicate(p) را از نوع a→bool و یک alist (x) را می‌گیرد و عناصری از لیست که شرط p را برقرار کنند را برمی‌گرداند. یعنی p یک تابع است که روی هر عنصر از نوع a در لیست x که از نوع alist است اعمال می‌شود و خروجی آن هم از نوع alist خواهد بود که فیلترشده است طبق p

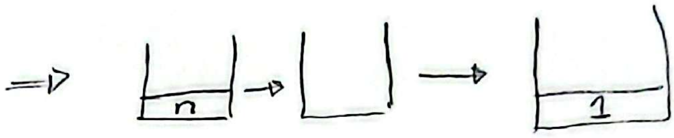
سوال ۳

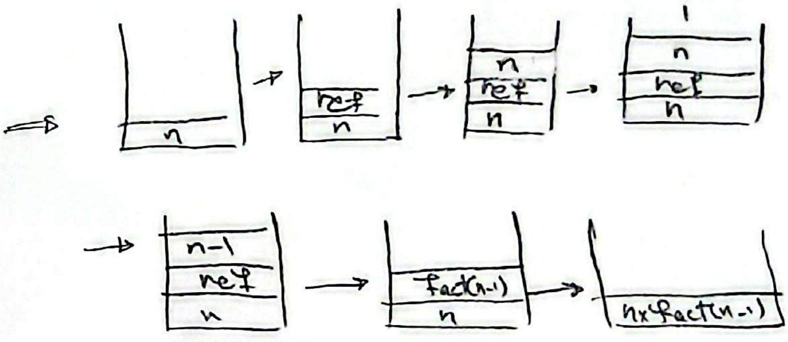
```
int fact(int n) { return n==0 ? 1 : n x fact(n-1) ; }
```

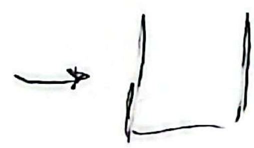
```
.method fact(I) I
.limit stack 4
.limit locals 2

    iload_1
    ifeq t1
    goto f1
t1: iconst_1
    goto exit
f1: iload_1
    aload_0
    iload_1
    iconst_1
    isub
    invokevirtual /fact(I) I
    imult
exit: ireturn
    .end method
```

چون static نیست پس this
به تابع refrence است



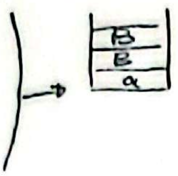




```
.method public B/f(IAC)C
.limit stack 2
.limit locals 2
aload_2
new B
dup
invokespecial B/<init>()V
astore_3
iload_2
invokevirtual A/g(a)I
aload_1
iadd
aload_3
getfield A/i I
pop
.end method
```

سوال 4

با توجه به تعریف چون non-static است باید locals آن 4 باشد.



فضا باید عمق stack

در واقع این تابع که مربوط به B.f است:

public C f(int i, A a, C c)....

متغیر محلی با اندیس 2 یعنی I از نوع int است دستور aload_1 به obj اشاره می‌رود. اینجا خطا داریم و باید از iload_1 استفاده می‌شد.

برای iadd هم دو int باید با هم جمع شوند که در این حالت که خطا داریم با حالت aload_1

در اینجا ref به B اشاره داریم ولی از A می‌خواهیم i را بگیریم که این مورد غلط است. همچنین بعد از pop مقدار int روی استک مانده که return می‌شود که با type برگشتی B.f تطابق ندارد.

همان خروجی iadd

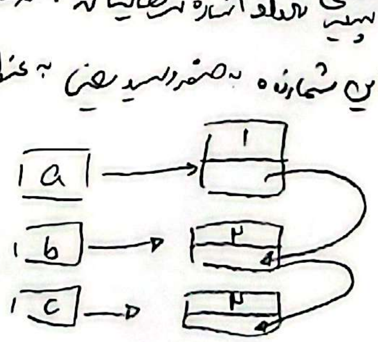
در این تابع که هیچ دستور return نداریم چون باید از تایپ C خروجی می‌دادیم.

سوال 5 — مثلا می‌توان انواع تکنیک‌ها را دسته‌بندی کرد

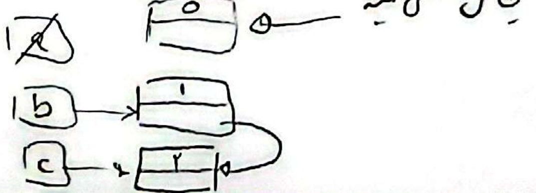
الف)

1. Refrence Counting : شمارش تعداد اشاره‌گرهایی که به هر شی اشاره می‌کنند (در لحظه گرفتن یا دنبال کردن) (به هر obj یک refrence count) و در Garbage Collection اگر این شمارنده به صفر رسید یعنی به عنوان garbage باید در لحظه گرفته شود.

```
Node a = new Node();
Node b = new Node();
Node C = new Node();
```



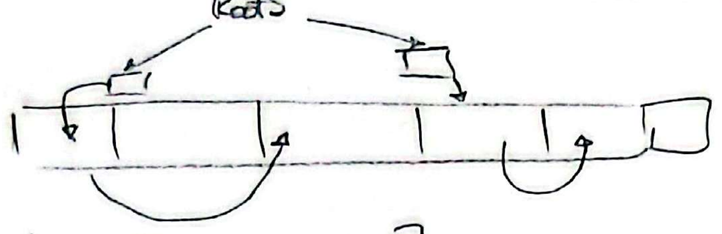

```
a.next = b;
b.next = C;
a = null;
```

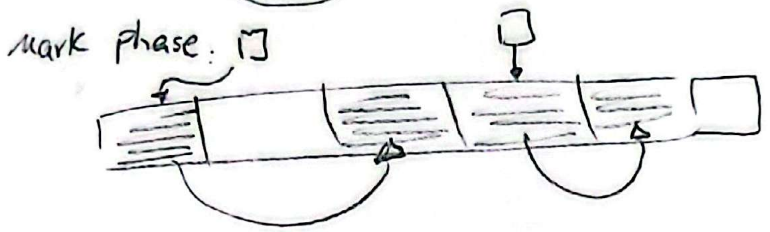


② Trace-Based Garbage Collection: این دسته از الگوریتم‌ها با پیمایش گراف اشیاء از Root ها اشیاء قابل دسترس را شناسایی می‌کنند و اشیاء غیرقابل دسترس را به عنوان garbage آزاد می‌کنند

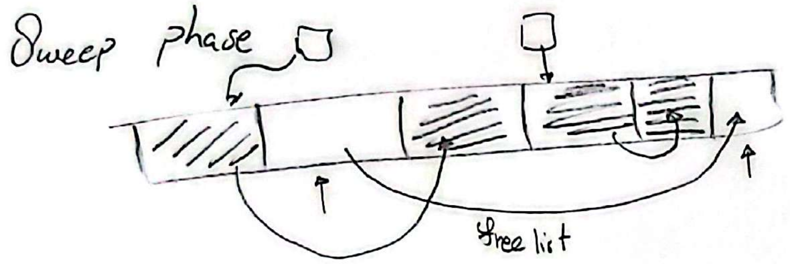
مثل mark-and-Sweep یا Deutsch-Schorr-Waite alg.

مثال





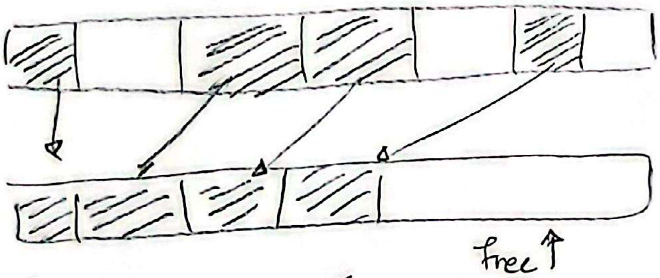
mark reachable objects



traverse the heap exhaustively and add unmarked objects to free list

③ Compacting Garbage Collection: این روش علاوه بر شناسایی و جمع‌آوری garbage ها اشیاء زنده را در حافظه جابه‌جا می‌کند و کنار هم قرار می‌دهد تا فضای خالی و گسسته ایجاد نشود. این روش‌ها به مشکل fragmentation کمک می‌کنند

مثل mark-and-Compact و Cheney مثل Copying alg.



memory after mark-and-Sweep

memory after Compaction

ب) در این روش heap را به ۲ قسمت from space و to space تقسیم می‌کنیم. باید به object ها به کمک BFS از from space به to space برویم. فضای to space در ابتدا خالی است. هنگام شناسایی obj را به to space می‌بریم یک pointer در مکان قدیمی obj در from space جا می‌گذاریم این اشاره‌گر به مکان جدید obj در to space اشاره می‌کند. سپس باید to space را اسکن کنیم و اشاره‌گرهایی که از obj به from space اشاره دارند را به to space کپی می‌کنیم (اگر قبلا نیاورده‌ایم) و اشاره‌گرها را آپدیت می‌کنیم. این کار را ادامه می‌دهیم تا زمانی که تمام obj های درون to-space اسکن شوند و هیچ پوینتری به from space نباشد. از دو پوینتر free و scan در to space استفاده می‌کنیم. در آخر هم نقش‌ها from و to با هم عوض می‌شوند

P a = new P()

P b = new P()

P c = new P()

, d

e ~ ~

, f ~ ~

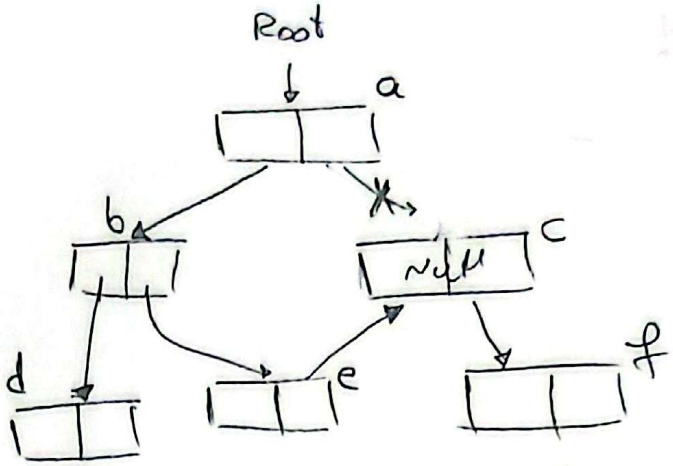
a. Set XY (b, c)

b. set XY (d, e)

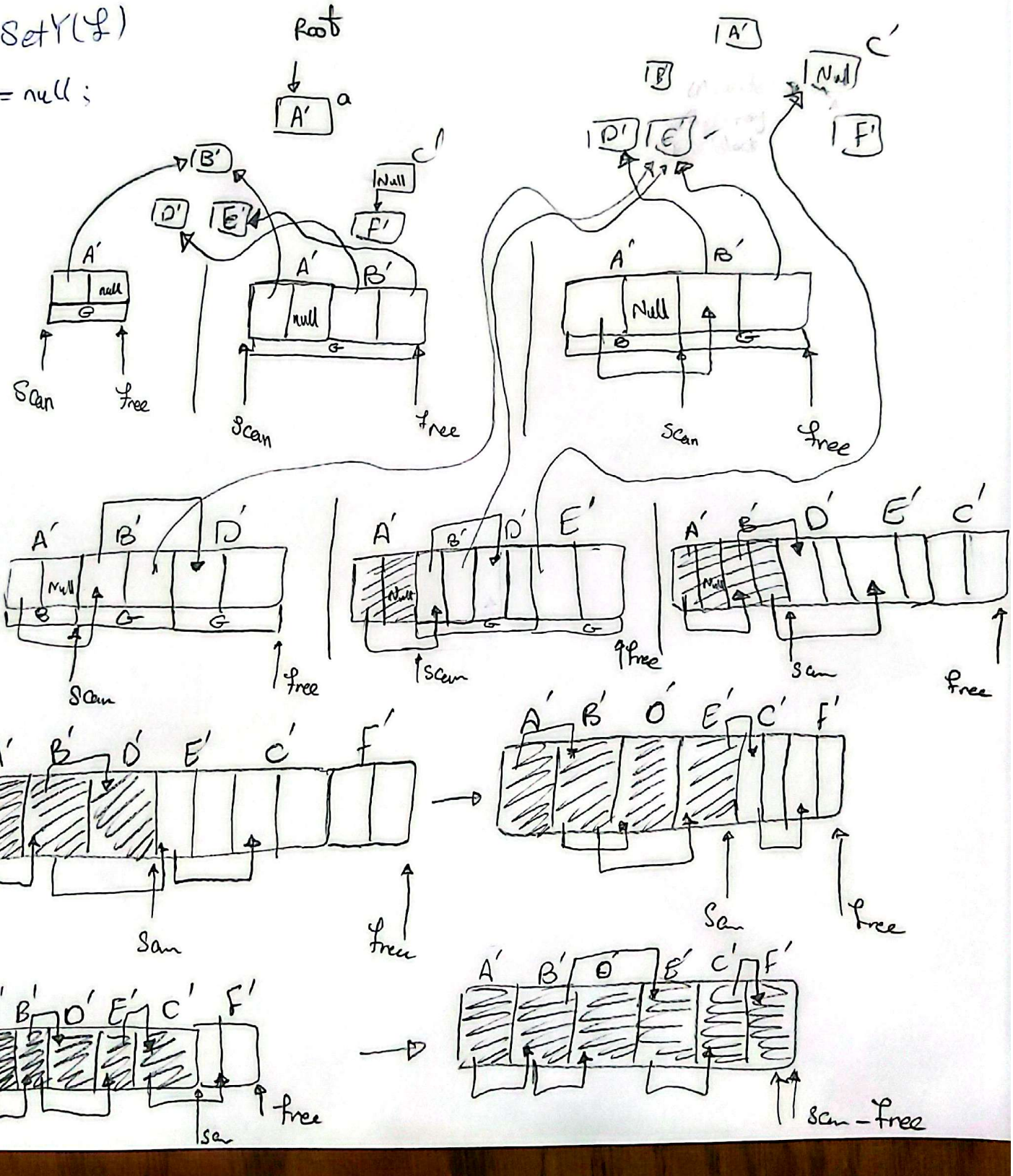
e Set Y (c)

c. Set Y (f)

c = null;



W=white G=Grey B=Black



سوال (۶)

الف) زمان اجرای mark-and-sweep متناسب با کل حجم heap و همچنین تعداد obj های زنده است یعنی O(H)+O(L)

اما زمان اجرای stop and copy متناسب با obj های زنده است یعنی O(L)

پس الگوریتم stop & copy سریع‌تر است. اگر حجم heap بسیار بزرگ باشد اما تعداد obj های زنده کم، تفاوت سرعت قابل توجه می‌شود.

با توجه به تعداد دفعاتی که این الگوریتم‌ها اجرا می‌شوند چون طبق سوال تنها زمانی که حافظه نداشته باشیم الگوریتم‌ها اجرا می‌شوند البته مشکلات به علت fragmentation ناشی از mark&sweep چند دفعه بیشتر از stop & copy اجرا شود.

ب) به عنوان در stop & copy، heap را دو قسمت می‌کنیم حافظه مصرفی تقریباً دو برابر حافظه قابل ملاحظه بیشتر است. در هر زمان فقط از یکی از دو حافظه استفاده می‌شود یعنی همیشه نیمی از heap بلااستفاده است (2xL)

اما فضای مورد نیاز برای mark & sweep برای نگهداری اشیاء زنده است (L)

ج) به وجود cyclic Refrences در ساختارها روزمره بسیار رایج اند. مثلاً لیست‌های پیوندی دو طرفه یا درخت‌ها در گراف‌ها یا بافرهای دایره‌ای را می‌توان اشاره کرد.

د) یک روش برای پیاده‌سازی refrence counting که با cycle-detector ترکیب می‌شود استفاده از الگوریتم‌هایی است که شبیه گراف است. مشابه فاز mark در mark&sweep اما برای تشخیص چرخه‌ها در obj هایی که refrence count صفر نمی‌شوند. در ابتدا obj هایی که refrence count آنها صفر نیست اما هیچ pointer خارجی (از root ها یا خارج cycle) به آنها وجود ندارد شناسایی کنیم اینها کاندیدهایی برای تشکیل دور هستند. یک پیمایش مثل DFS از هر یک از این کاندیدها می‌زنیم. در حین پیمایش اگر به obj ای برسیم که قبلاً در همین مسیر دیده شده (مسیرها را باید در نظر بگیریم) یک دور یا cycle شناسایی شده است. برای تشخیص دقیق cycle ها می‌توان ref.cnt موقت obj های درون یک cycle را کاهش داد و سپس پیمایش کرد. اگر پس از این کاهش موقت ref.cnt یک obj به صفر رسید یعنی obj از بیرون cycle قابل دسترسی نیست آنگاه فقط یک cycle از نوع garbage است. پس از تشخیص، شمارنده‌ها باید به حالت اولیه برگردانده شوند.

یک cycle را می‌توان به عنوان garbage پاک کرد زمانی که هیچ pointer خارجی (از خارج cycle یا از مجموعه root ها) به هیچ یک از اشیاء درون آن چرخه وجود نداشته باشد. یعنی وقتی یک cycle detector چرخه‌ای را شناسایی می‌کند و تایید می‌کند که این چرخه کاملاً طوری است که نمی‌توان از ریشه به آن رسید، آنگاه تمام اشیاء درون آن cycle می‌توانند پاک شوند