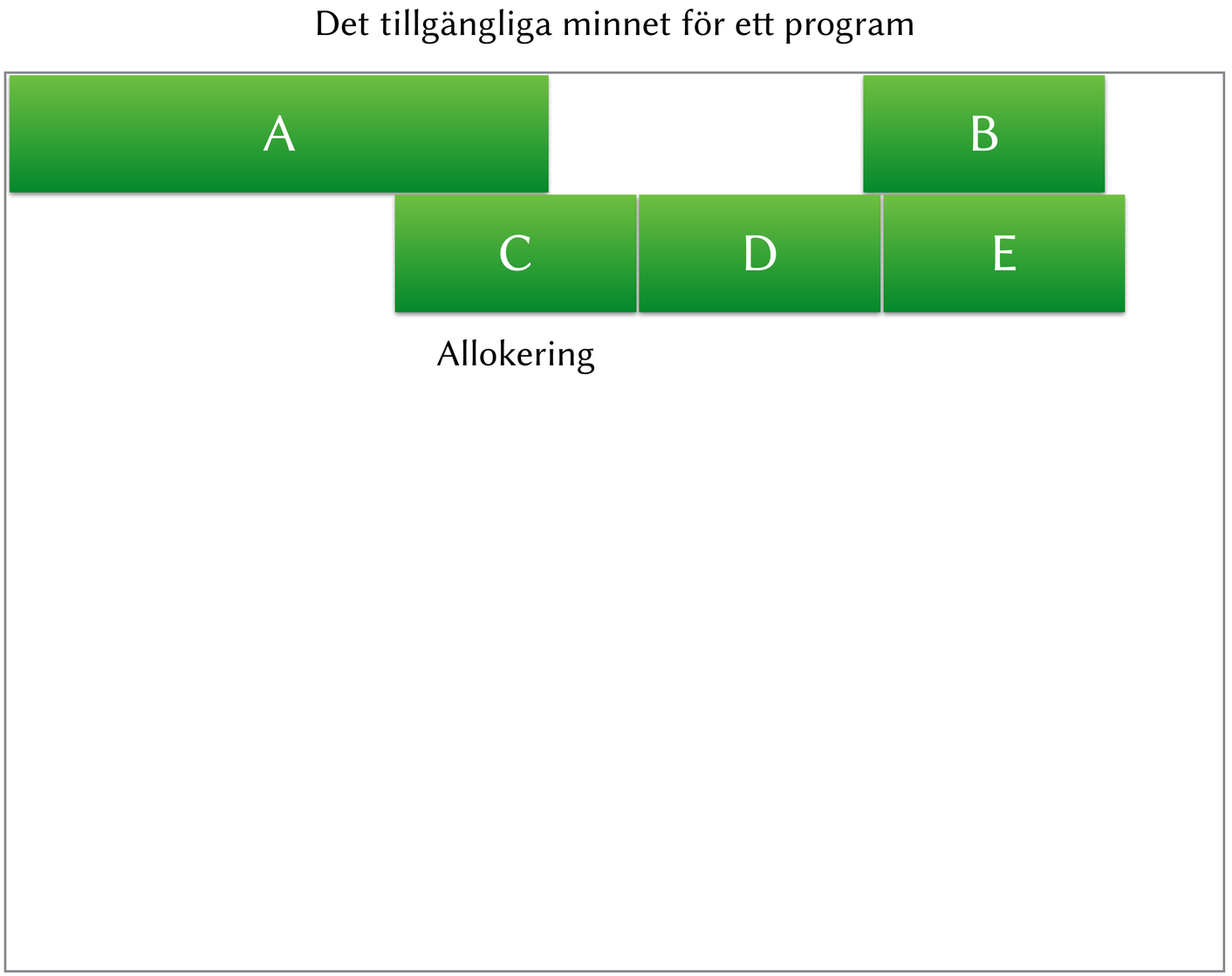
Skräpsamling och minneshantering

Tobias Wrigstad
Uppsala University

IOOPM 2020

Minnesallokering

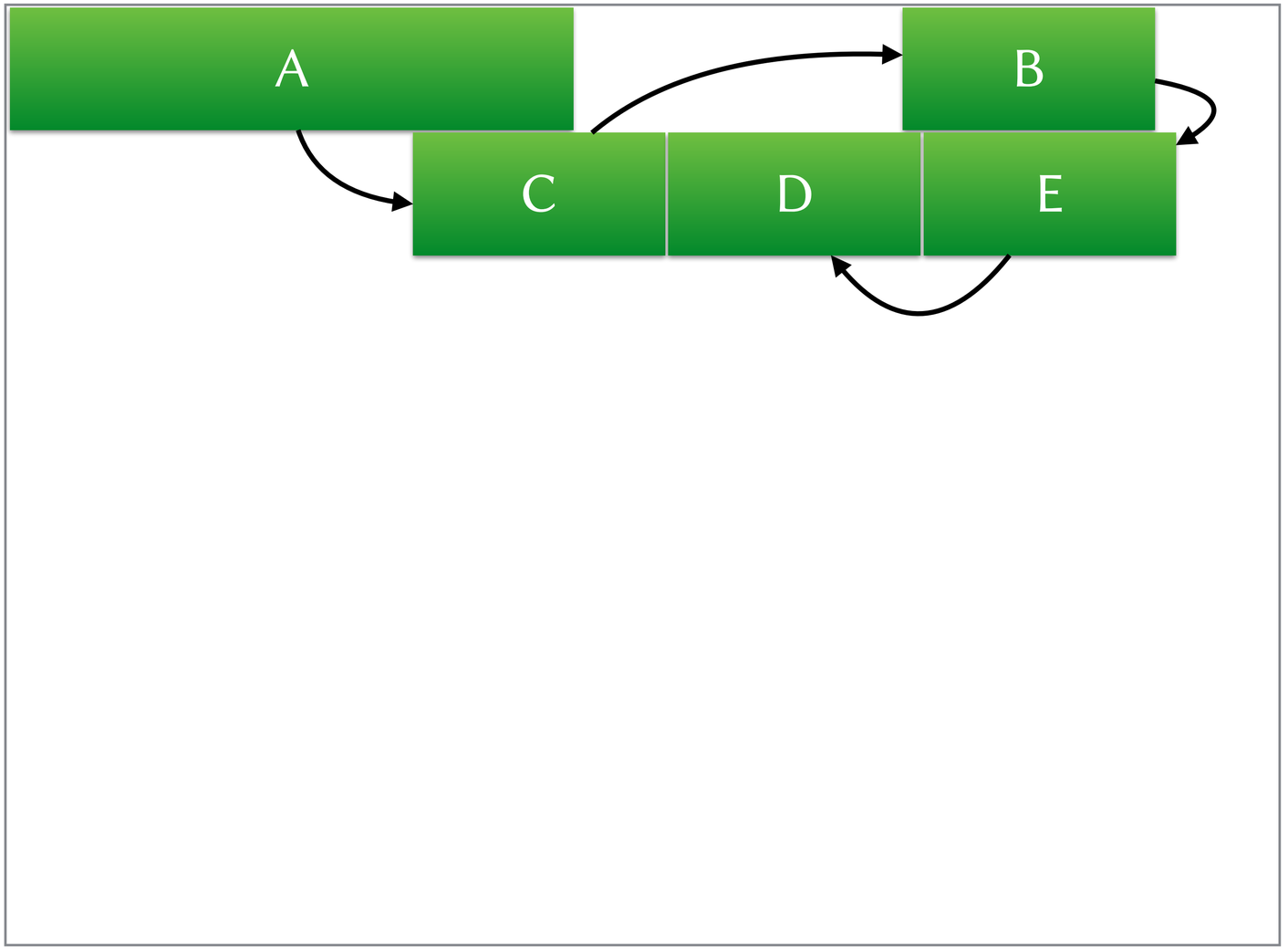
Hur vet malloc var det finns ledigt minne?



Minnesallokering

Hur vet malloc var det finns ledigt minne?

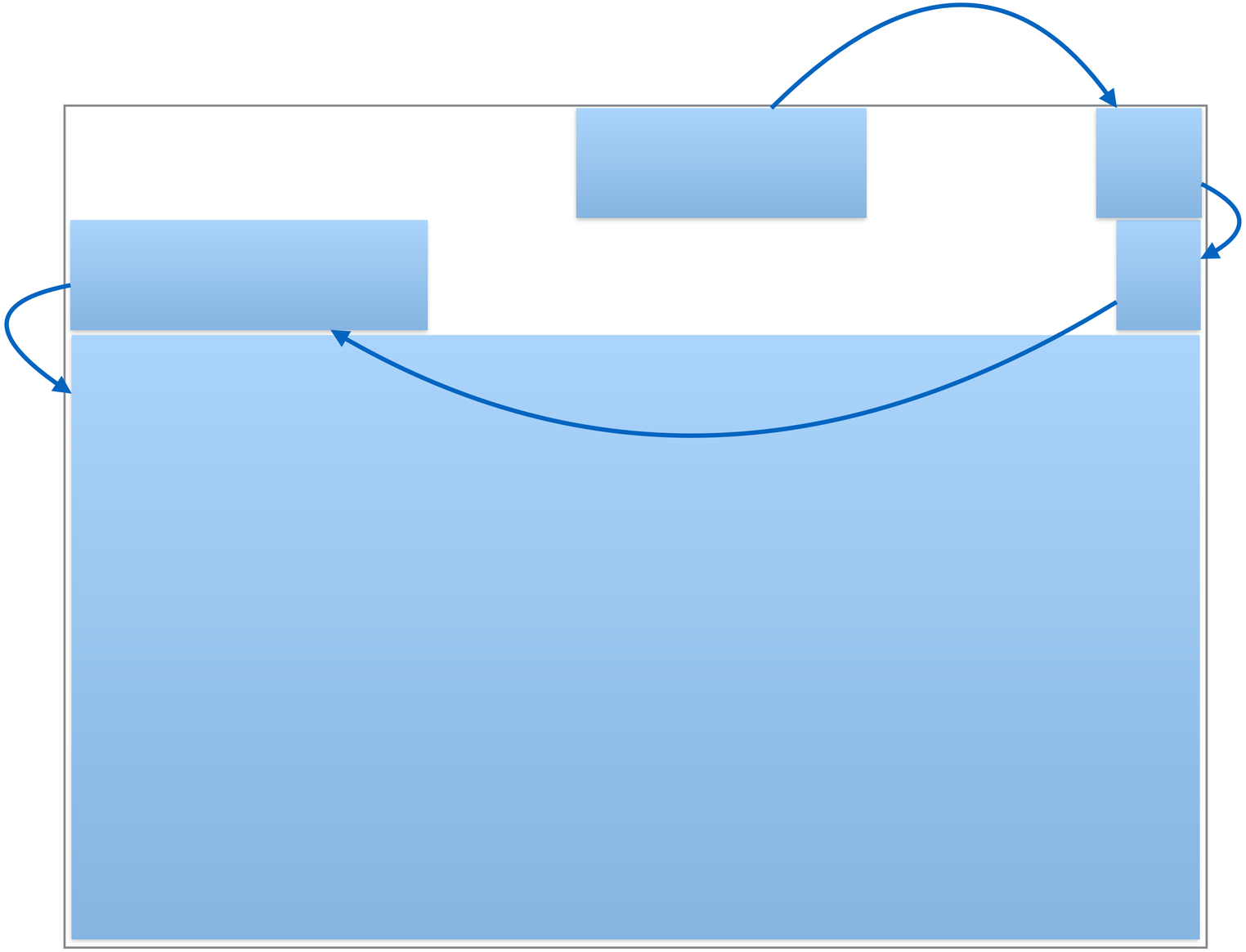
- Länka samman alla allokeringar
- Räcker inte — måste också veta var det finns ledigt utrymme!



Minnesallokering

Hur vet malloc var det finns ledigt minne?

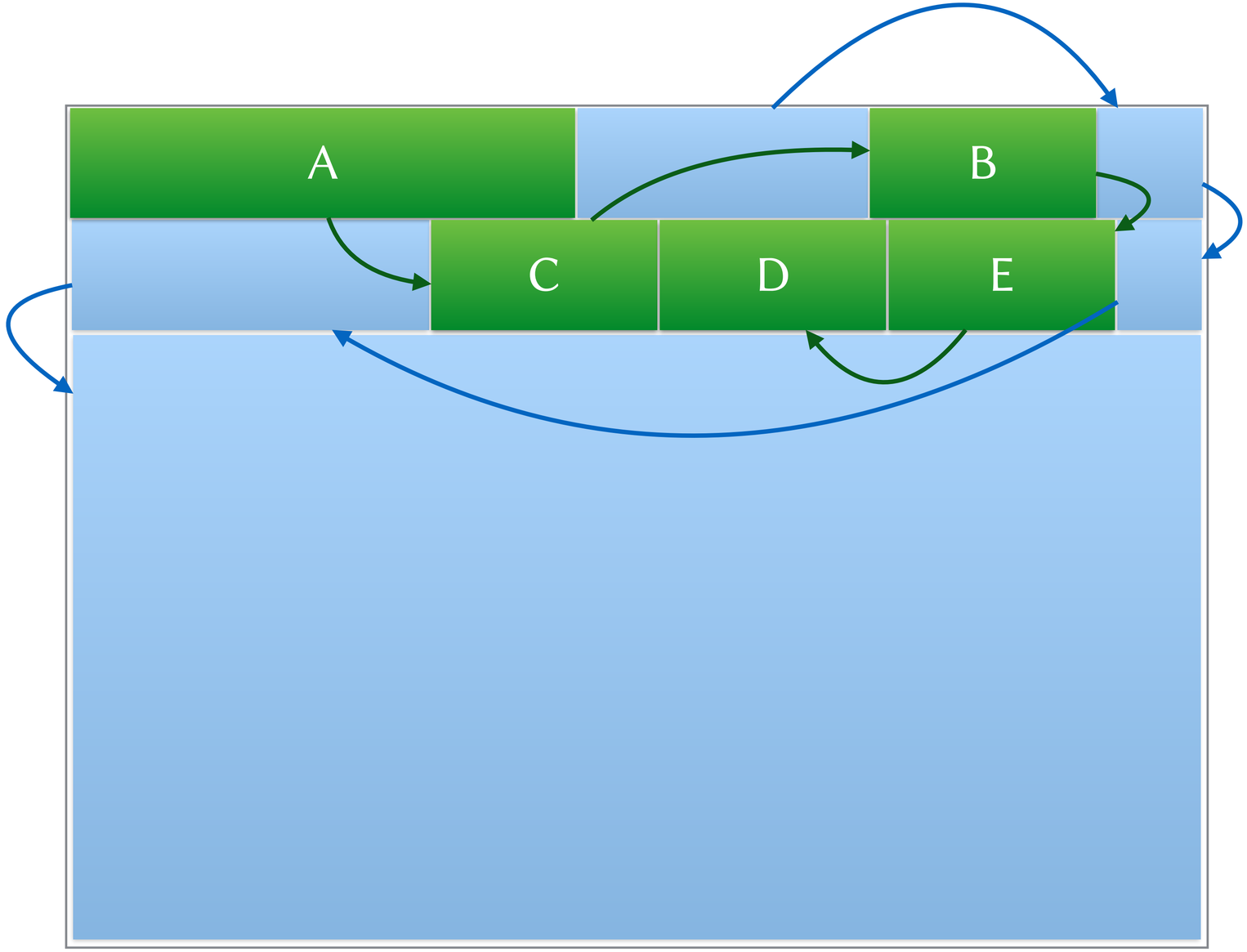
- Länka samman alla allokeringar
- Räcker inte — måste också veta var det finns ledigt utrymme!



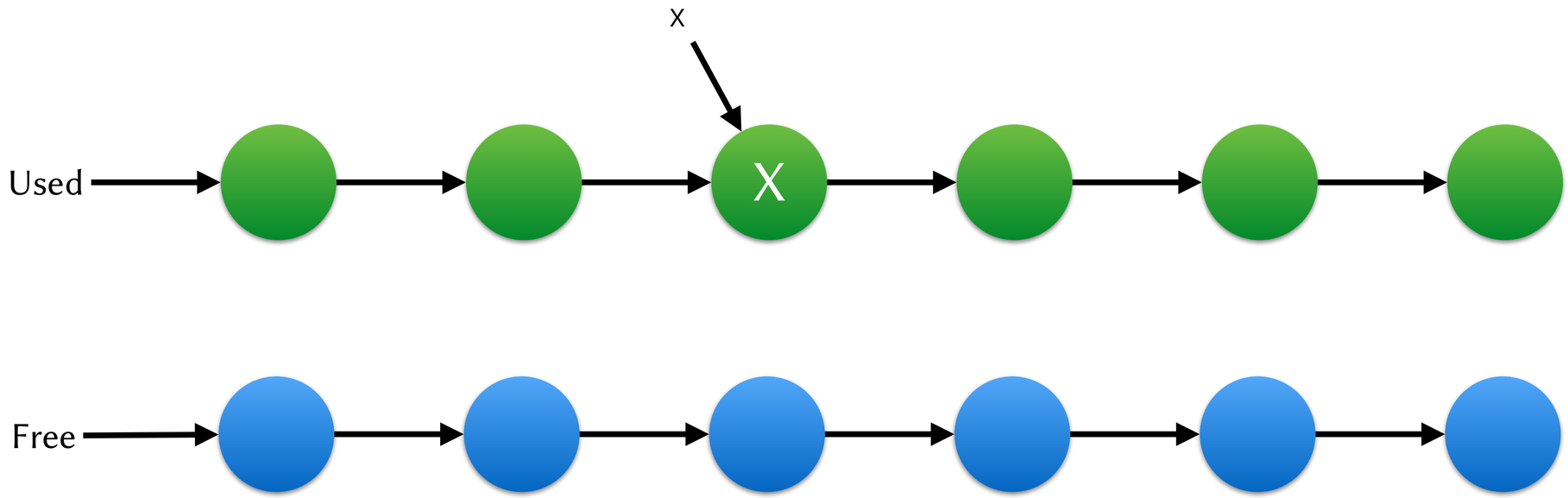
Minnesallokering

Hur vet malloc var det finns ledigt minne?

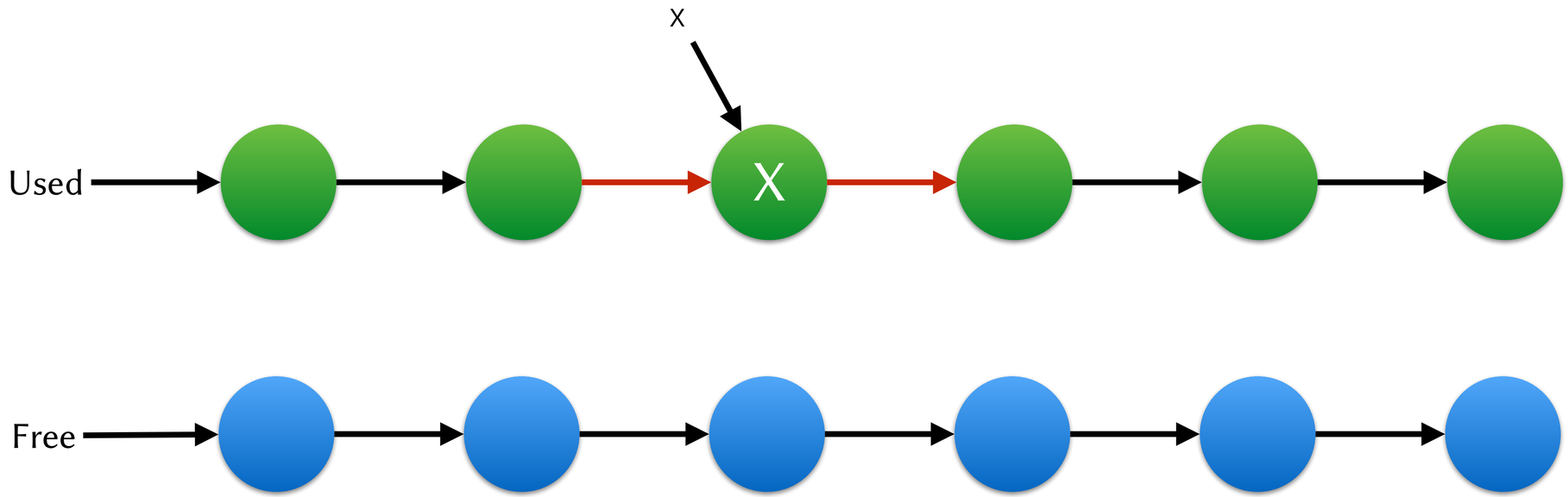
- Länka samman alla allokeringar
- Räcker inte — måste också veta var det finns ledigt utrymme!



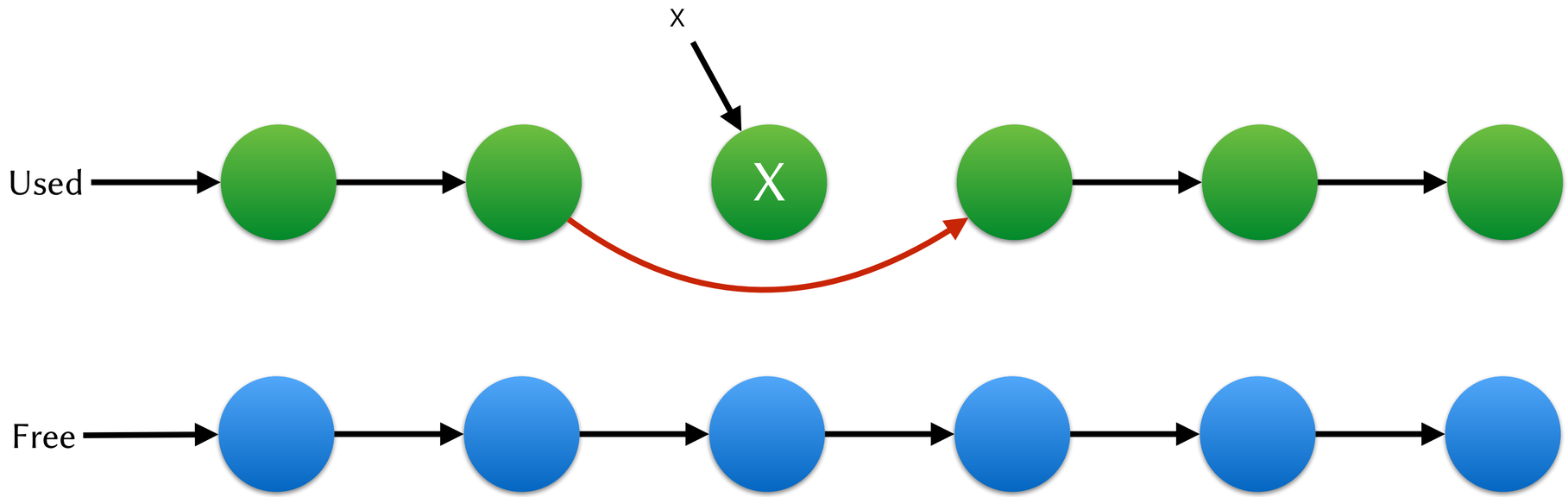
Använt och tillgängligt minne



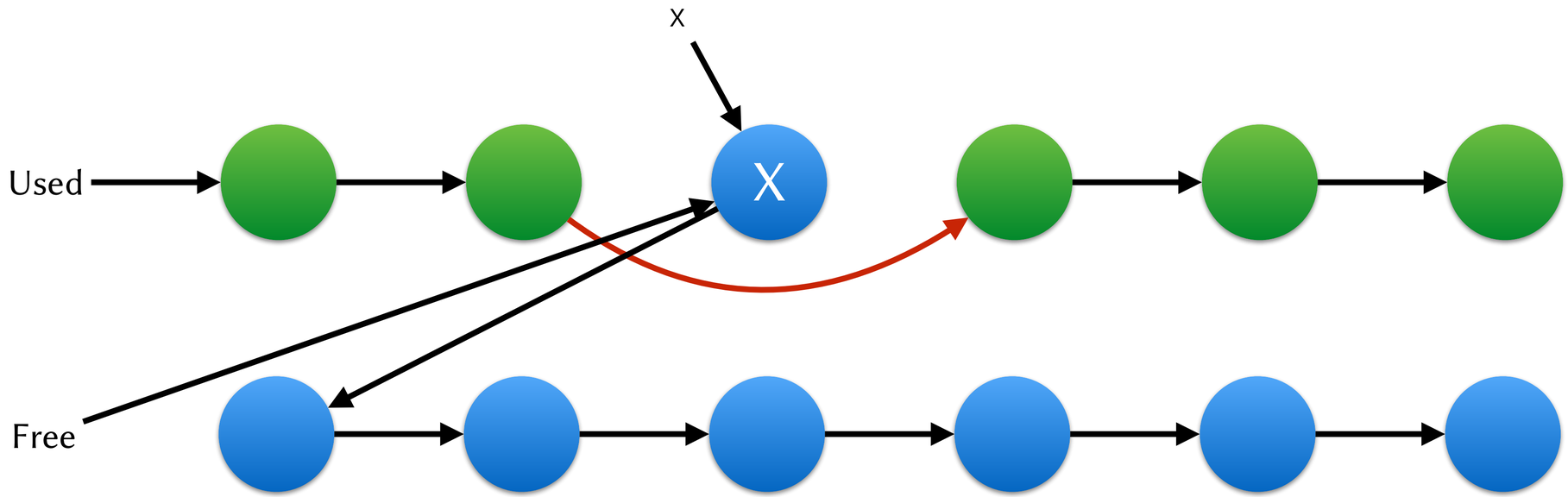
free(x);



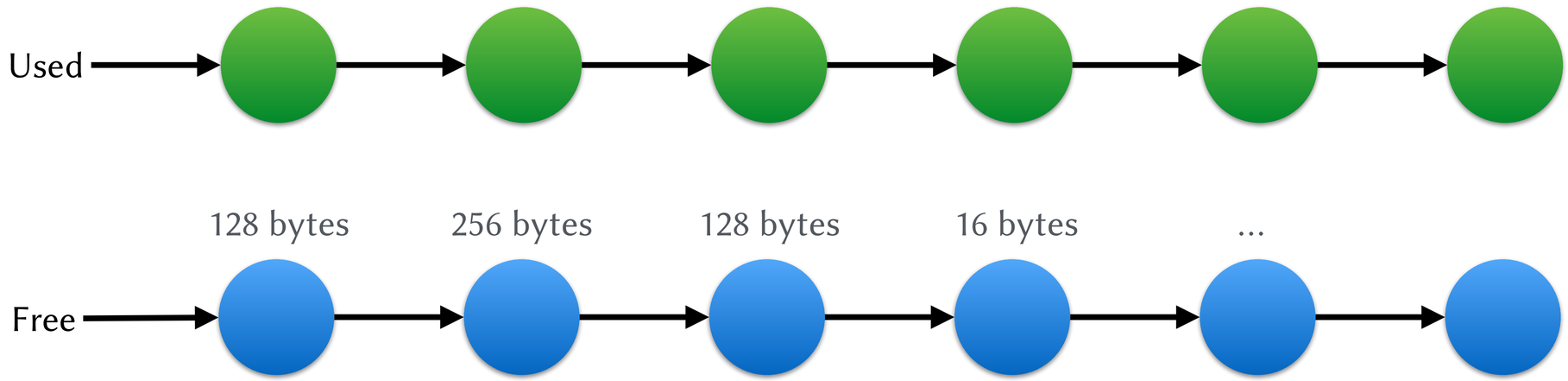
free(x);



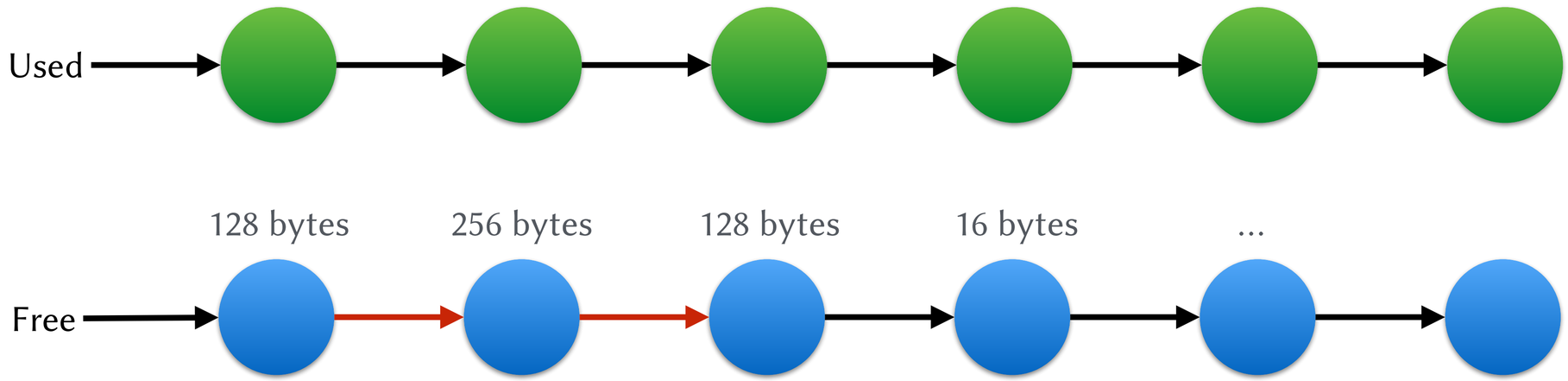
free(x);



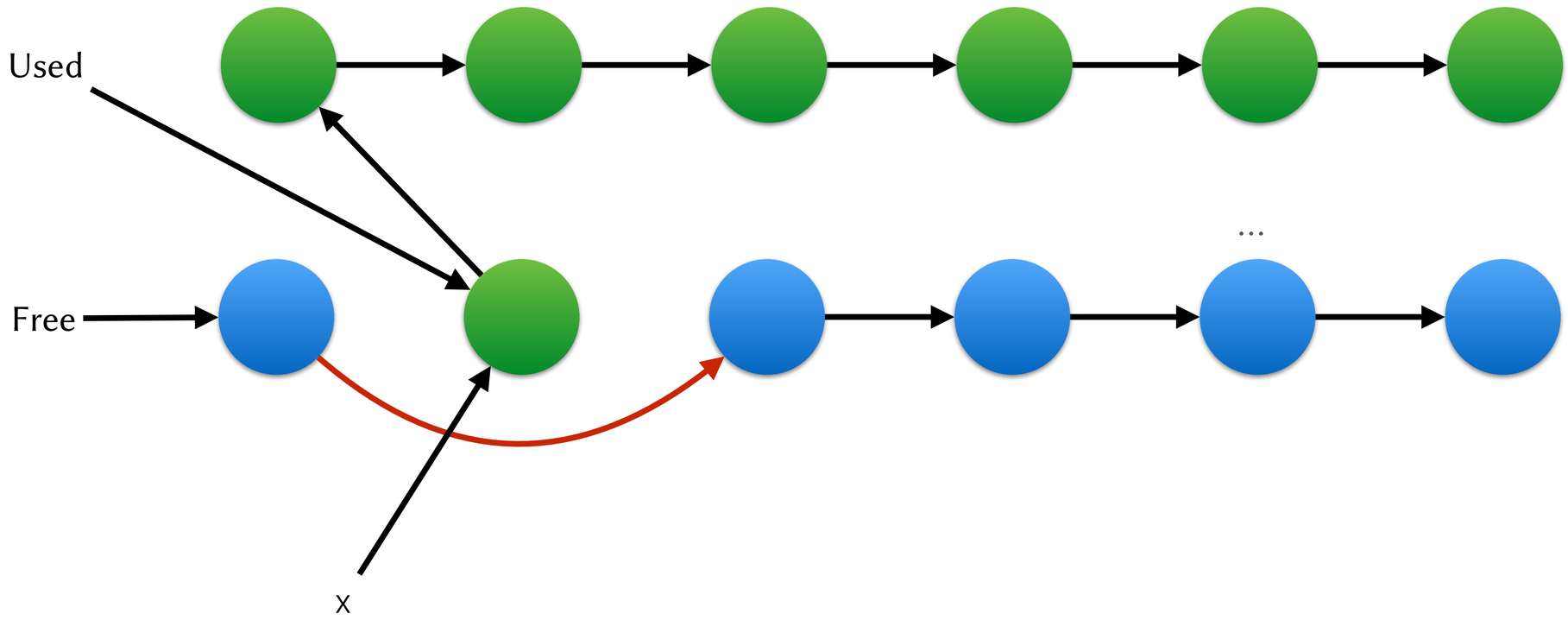
x = malloc(256);



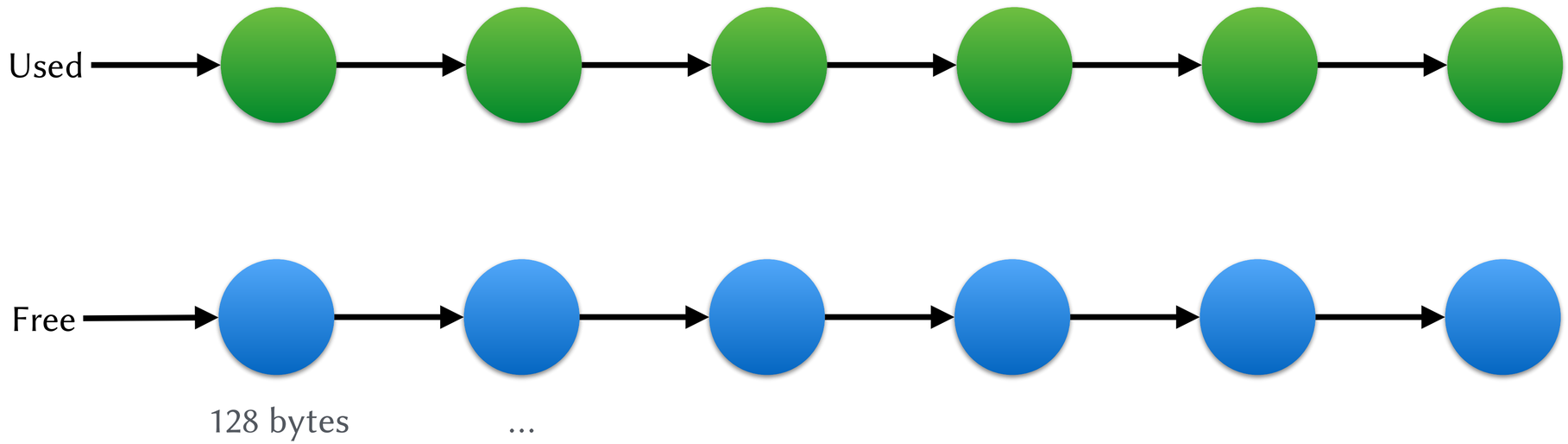
x = malloc(256);



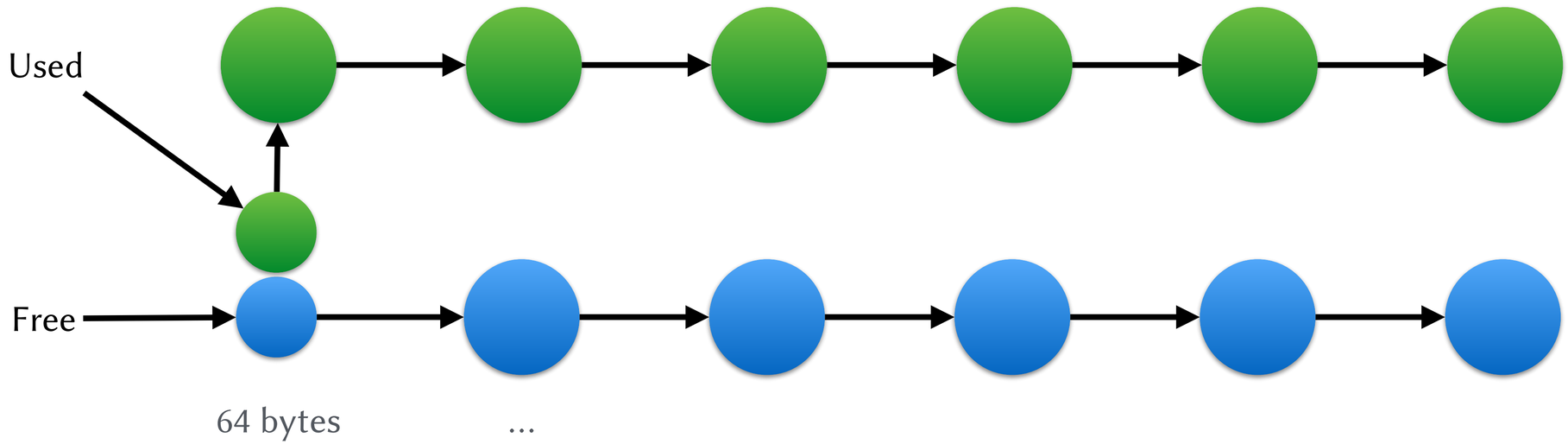
x = malloc(256);



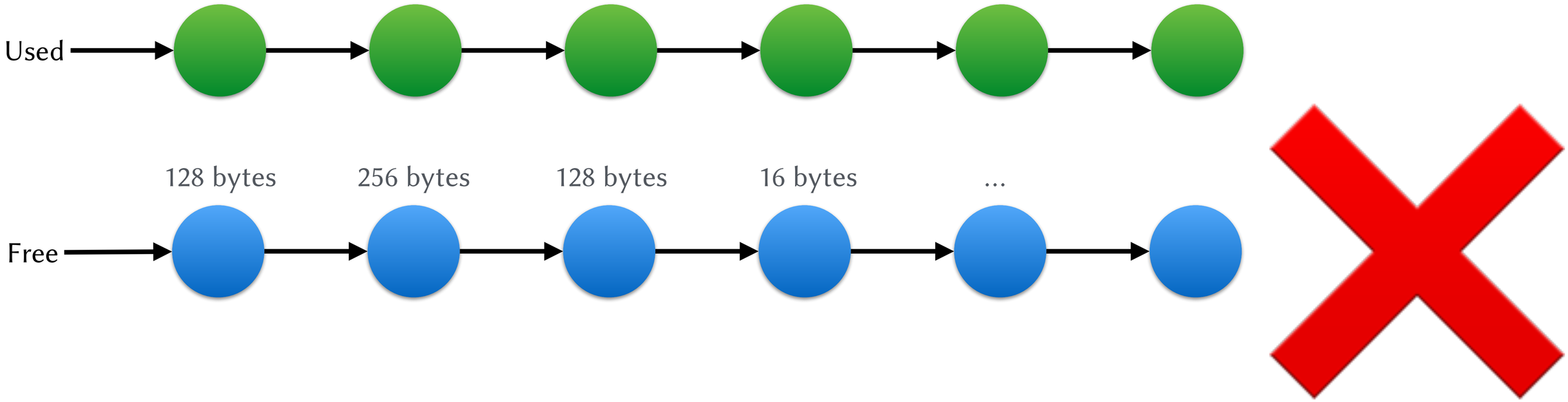
x = malloc(64);



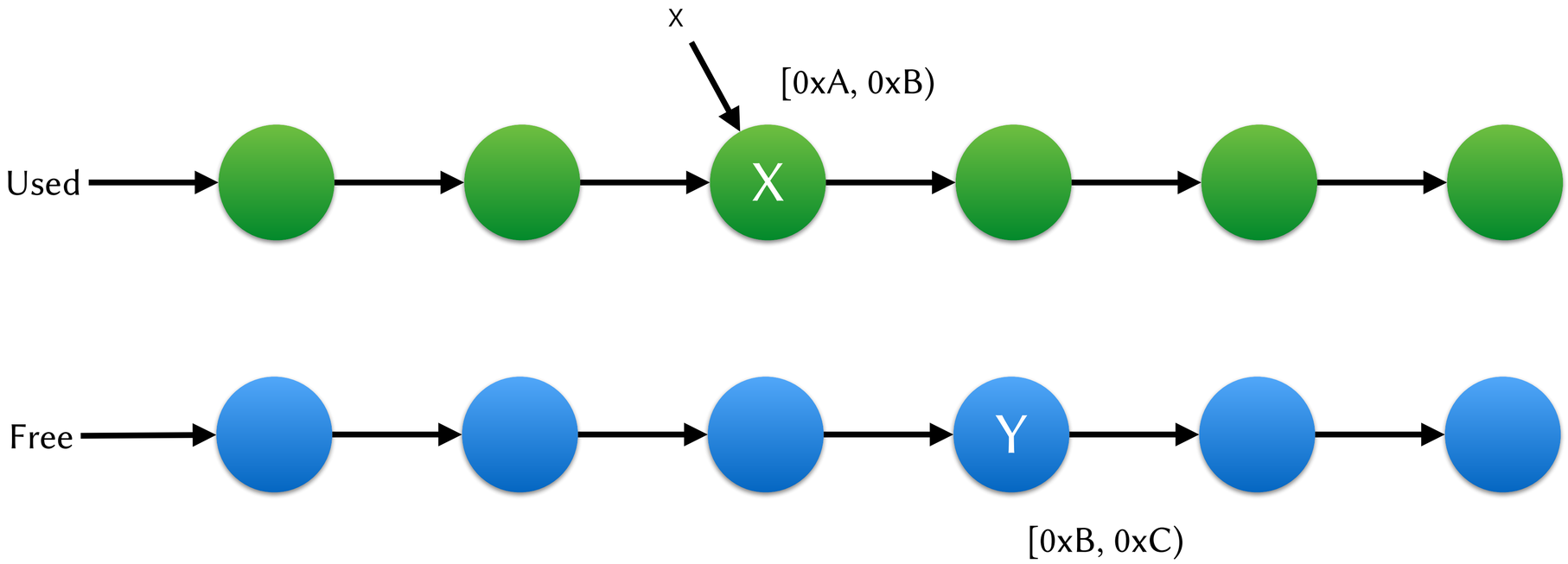
x = malloc(64);



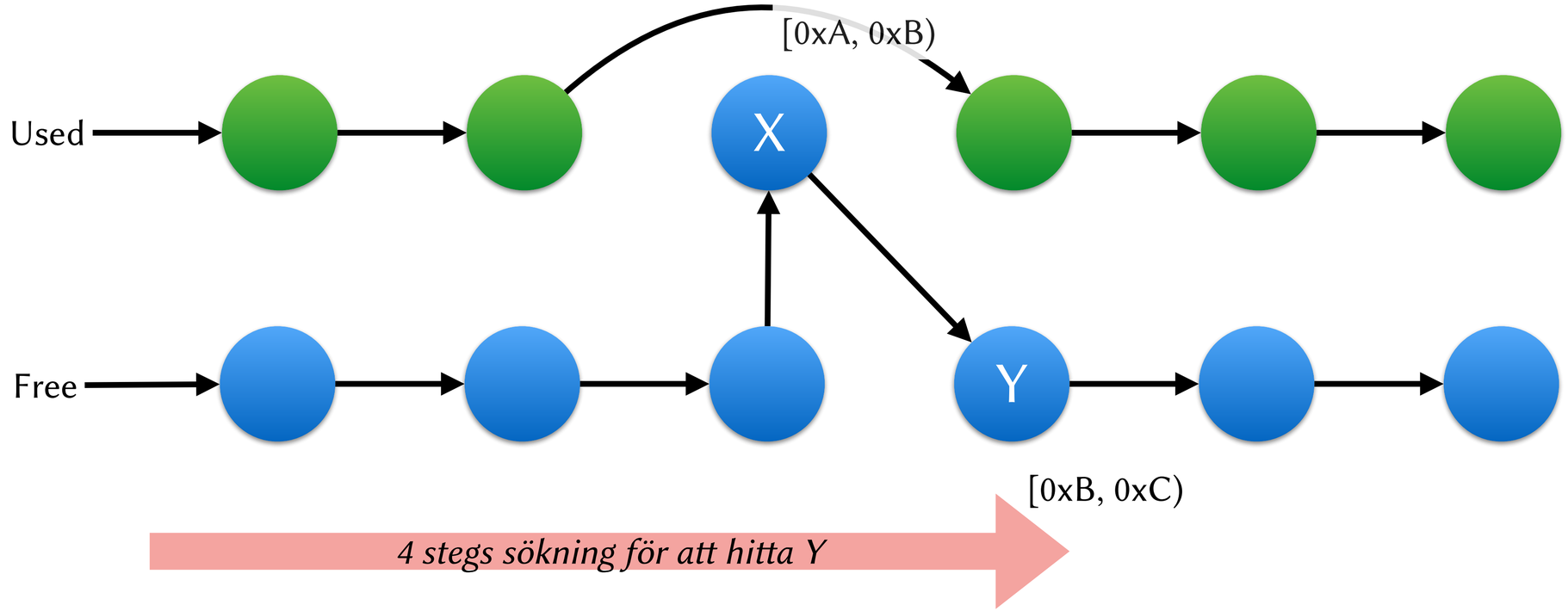
x = malloc(512);



Free måste hantera fragmentering — free(x);

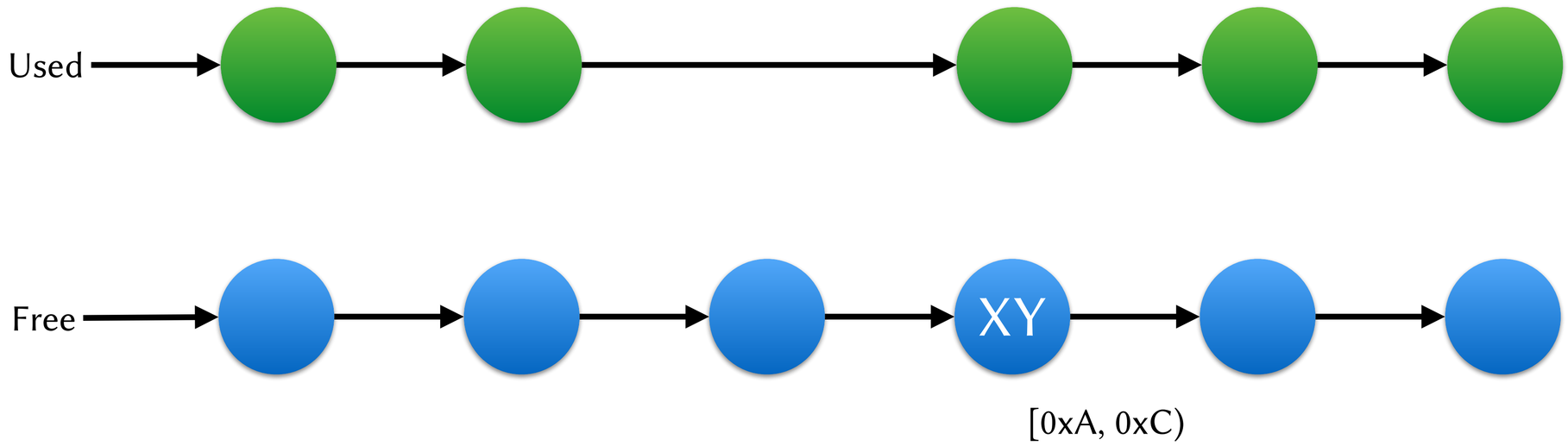


Free måste hantera fragmentering — free(x);



Detta innebär att free(x) — åtminstone logiskt — måste söka upp rätt plats i free-listan för inlänkning

Free måste hantera fragmentering — free(x);



Detta innebär att free(x) — åtminstone logiskt — måste söka upp rätt plats i free-listan för inlänkning

En dubbellänkad lista med blandade block i adressordning

Allokera X bytes i A:

1. Om X < sizeof(A), splitta A till A (x bytes) och B (resten)
    a. Länka samman B och A
2. Gör A grön

Frigör A:

1. Gör A blå
2. Om C är ledig, slå samman A och C
2. Om D är ledig, slå samman A och D

👍 Slå samman i kontant tid
👎 Dubbelt så många pekare

Free-listans sorteringsordning

Sorterad från minsta till största

Sök från minsta till ”best-fit” med minsta möjliga fragmentering

Sorterad i stigande adressordning

Ökad sannolikhet att allokeringar nära i tiden hamnar nära varandra i rummet — dyr sökning efter ledigt utrymme

Sorterad från största till minsta

Allokering i konstant tid men med högre risk för fragmentering

Problemet med manuell minneshantering

Lokalt beslut som kräver globalt resonerande

```
void remove(list_t *list, int index)
{
  link_t *cursor = list->first; // dummy
  for (int i = 0; i < index; ++i)
    {
      cursor = cursor->next;
    }
  link_t *to_be_removed = cursor->next;
  cursor->next = cursor->next->next;

  free(to_be_removed->element); ???
  free(to_be_removed);
}
```

Problemet med manuell minneshantering

Lokalt beslut som kräver globalt resonerande

```
void remove(list_t *list, int index)
{
  link_t *cursor = list->first; // dummy
  for (int i = 0; i < index; ++i)
    {
      cursor = cursor->next;
    }
  link_t *to_be_removed = cursor->next;
  cursor->next = cursor->next->next;

  free(to_be_removed->element); ???
  free(to_be_removed);
}
```

Om `to_be_removed->element` är den **enda** pekaren till objektet i systemet får vi ett minnesläckage om vi **inte** gör free

Problemet med manuell minneshantering

Lokalt beslut som kräver globalt resonerande

```
void remove(list_t *list, int index)
{
  link_t *cursor = list->first; // dummy
  for (int i = 0; i < index; ++i)
    {
      cursor = cursor->next;
    }
  link_t *to_be_removed = cursor->next;
  cursor->next = cursor->next->next;

  free(to_be_removed->element); ???
  free(to_be_removed);
}
```

Om `to_be_removed->element` är den **enda** pekaren till objektet i systemet får vi ett minnesläckage om vi **inte** gör free

Om `to_be_removed->element` **inte** är den enda pekaren till objektet i systemet får vi en dangling pointer om vi **gör** free

Problemet med manuell minneshantering

Lokalt beslut som kräver globalt resonerande

```
void remove(list_t *list, int index)
{
  link_t *cursor = list->first; // dummy
  for (int i = 0; i < index; ++i)
    {
      cursor = cursor->next;
    }
  link_t *to_be_removed = cursor->next;
  cursor->next = cursor->next->next;

  free(to_be_removed->element); ???
  free(to_be_removed);
}
```

Om `to_be_removed->element` är den **enda** pekaren till objektet i systemet får vi ett minnesläckage om vi **inte** gör free

Om `to_be_removed->element` **inte** är den enda pekaren till objektet i systemet får vi en dangling pointer om vi **gör** free

Vem ansvarar för att städa bort data, och hur görs det tydligt i systemet?

Fördelen med automatisk minneshantering

Sker automatiskt — i de flesta fall

```
void remove(List list, int index)
{
  Link cursor = list.first; // dummy
  for (int i = 0; i < index; ++i)
    {
      cursor = cursor.next;
    }
  Link to_be_removed = cursor.next;
  cursor.next = cursor.next.next;
}
```

Fördelen med automatisk minneshantering

Sker automatiskt — i de flesta fall

```
void remove(List list, int index)
{
  Link cursor = list.first; // dummy
  for (int i = 0; i < index; ++i)
    {
      cursor = cursor.next;
    }
  Link to_be_removed = cursor.next;
  cursor.next = cursor.next.next;
}
```

Om `to_be_removed.element` är den **enda** pekaren till objektet i systemet blir objektet skräp och tas därför till slut bort av GC

Fördelen med automatisk minneshantering

Sker automatiskt — i de flesta fall

```
void remove(List list, int index)
{
  Link cursor = list.first; // dummy
  for (int i = 0; i < index; ++i)
    {
      cursor = cursor.next;
    }
  Link to_be_removed = cursor.next;
  cursor.next = cursor.next.next;
}
```

Om `to_be_removed.element` är den **enda** pekaren till objektet i systemet blir objektet skräp och tas därför till slut bort av GC

Om `to_be_removed.element` **inte** är den enda pekaren till objektet i systemet fortsätter det att existera som om inget hade hänt

Fördelen med automatisk minneshantering

Sker automatiskt — i de flesta fall

```
void remove(List list, int index)
{
  Link cursor = list.first; // dummy
  for (int i = 0; i < index; ++i)
    {
      cursor = cursor.next;
    }
  Link to_be_removed = cursor.next;
  cursor.next = cursor.next.next;
}
```

Om `to_be_removed.element` är den **enda** pekaren till objektet i systemet blir objektet skräp och tas därför till slut bort av GC

Om `to_be_removed.element` **inte** är den enda pekaren till objektet i systemet fortsätter det att existera som om inget hade hänt

Hur vet vi om vi släppt alla pekare till ett objekt?

The World is (Mostly) Garbage Collected

Most modern programs are written in managed languages that use garbage collection

Garbage Collection

- Avoids entire classes of bugs
- Speeds up productivity
- Improve code quality
- Incurs performance overhead

Jan 2019

1. JavaScript ✔
2. Java ✔
3. Python ✔
4. PHP ✔
5. C++ ✘
6. C# ✔
7. TypeScript ✔
8. Shell
9. C ✘
10. Ruby ✔

GitHub

Sep 2019	Sep 2018	Programming Language	Ratings
1	1	Java ✔	16.661%
2	2	C ✘	15.205%
3	3	Python ✔	9.874%
4	4	C++ ✘	5.635%
5	6	C# ✔	3.399%
6	5	Visual Basic .NET ✔	3.291%
7	8	JavaScript ✔	2.128%
8	9	SQL ✔	1.944%
9	7	PHP ✔	1.863%
10	10	Objective-C ✔	1.840%

TIOBE Index

Garbage Collection

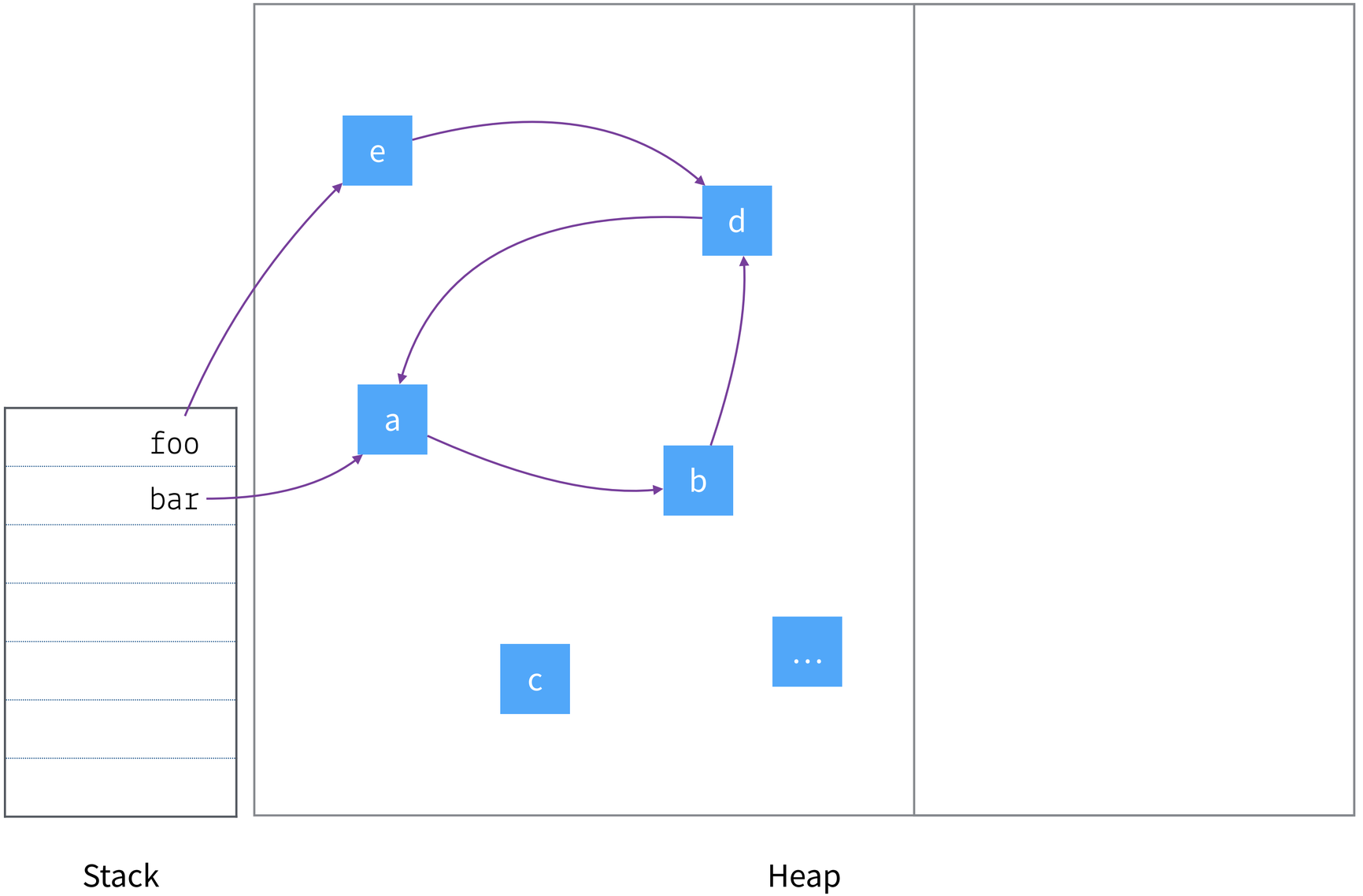
Reference counting

Tracing

Defragmentation

Deallocation

Allocation



Garbage Collection

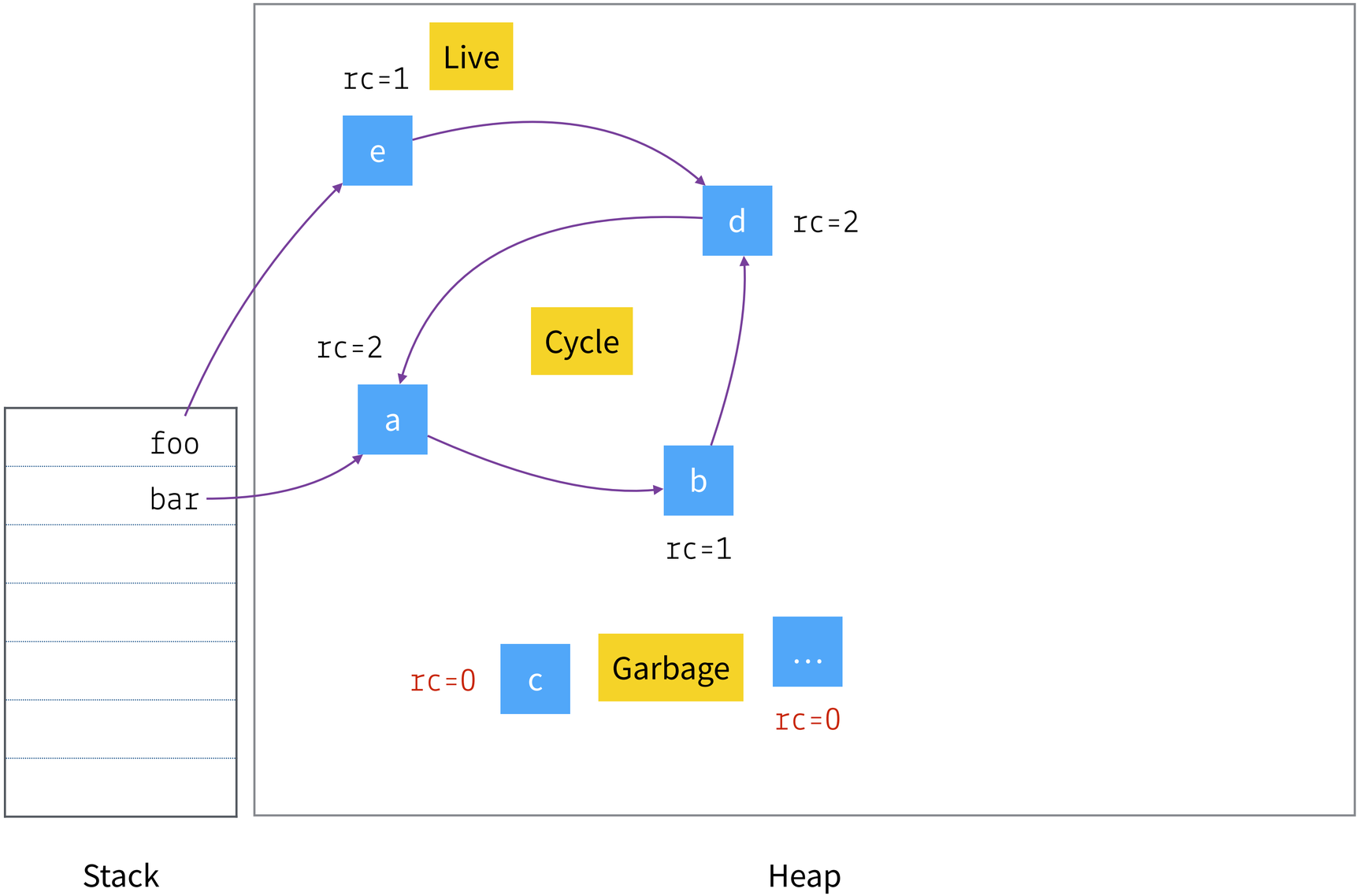
Reference counting

Tracing

Defragmentation

Deallocation

Allocation



Garbage Collection

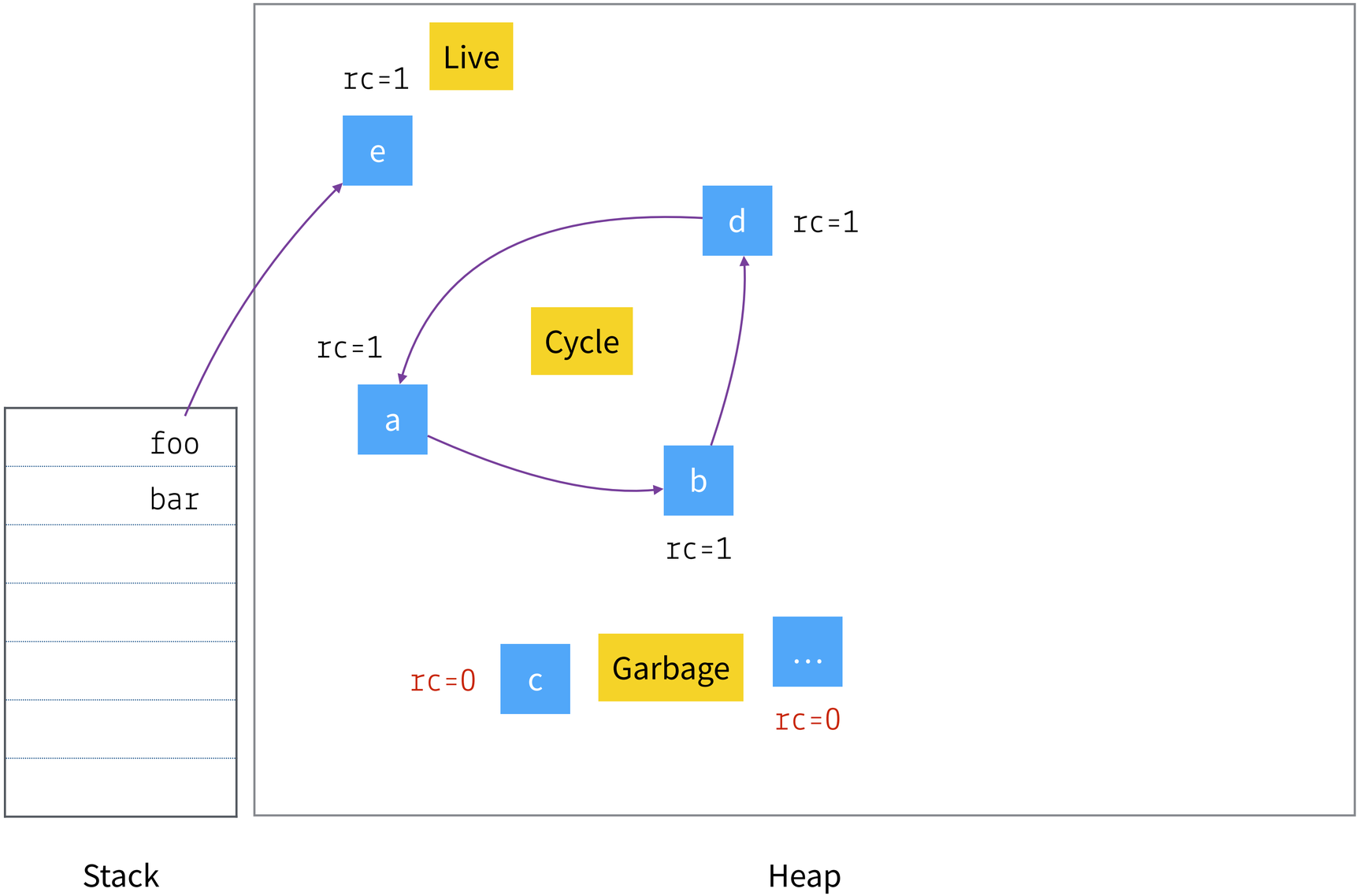
Reference counting

Tracing

Defragmentation

Deallocation

Allocation



Garbage Collection

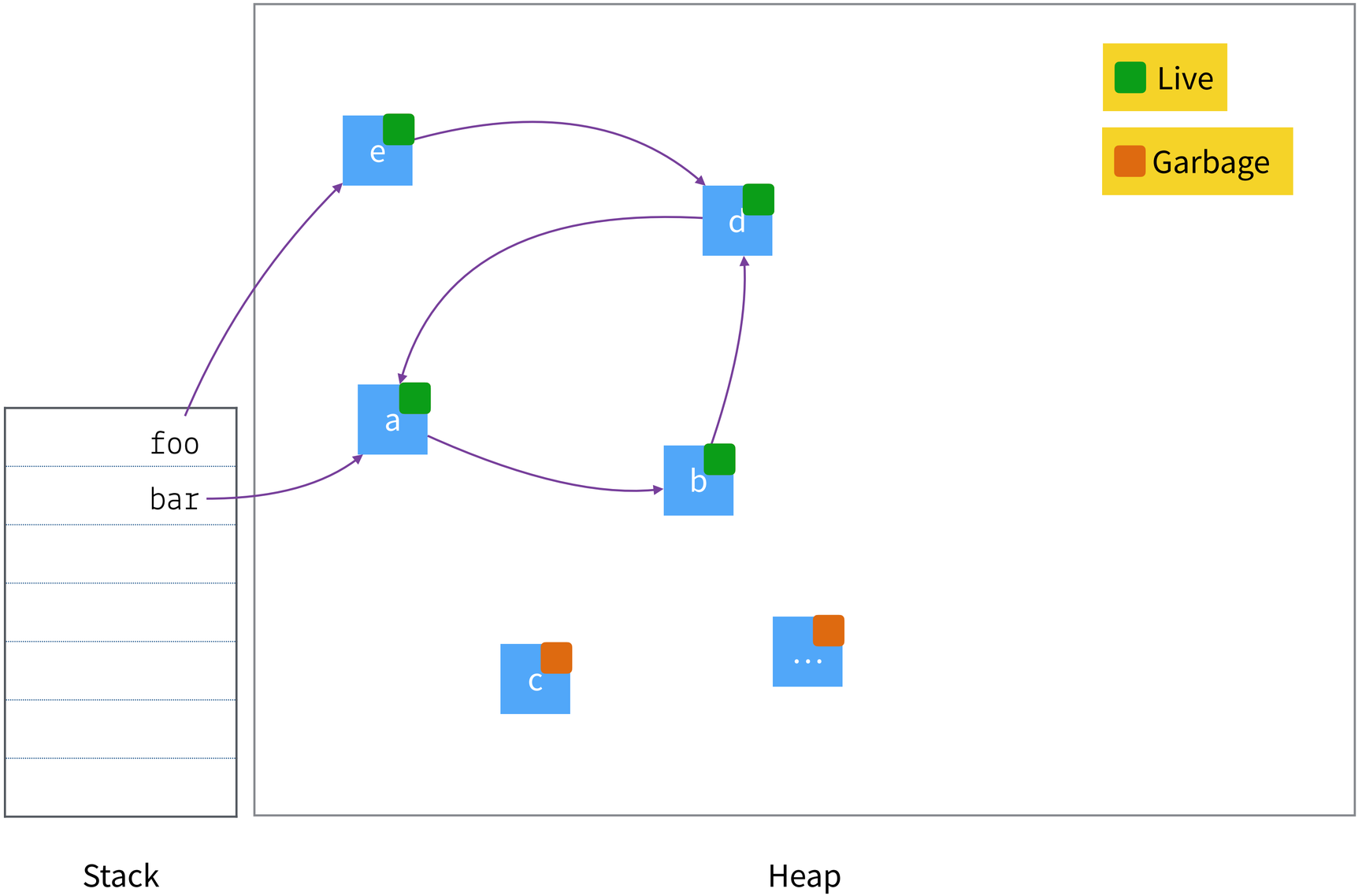
Reference counting

Tracing

Defragmentation

Deallocation

Allocation



Garbage Collection

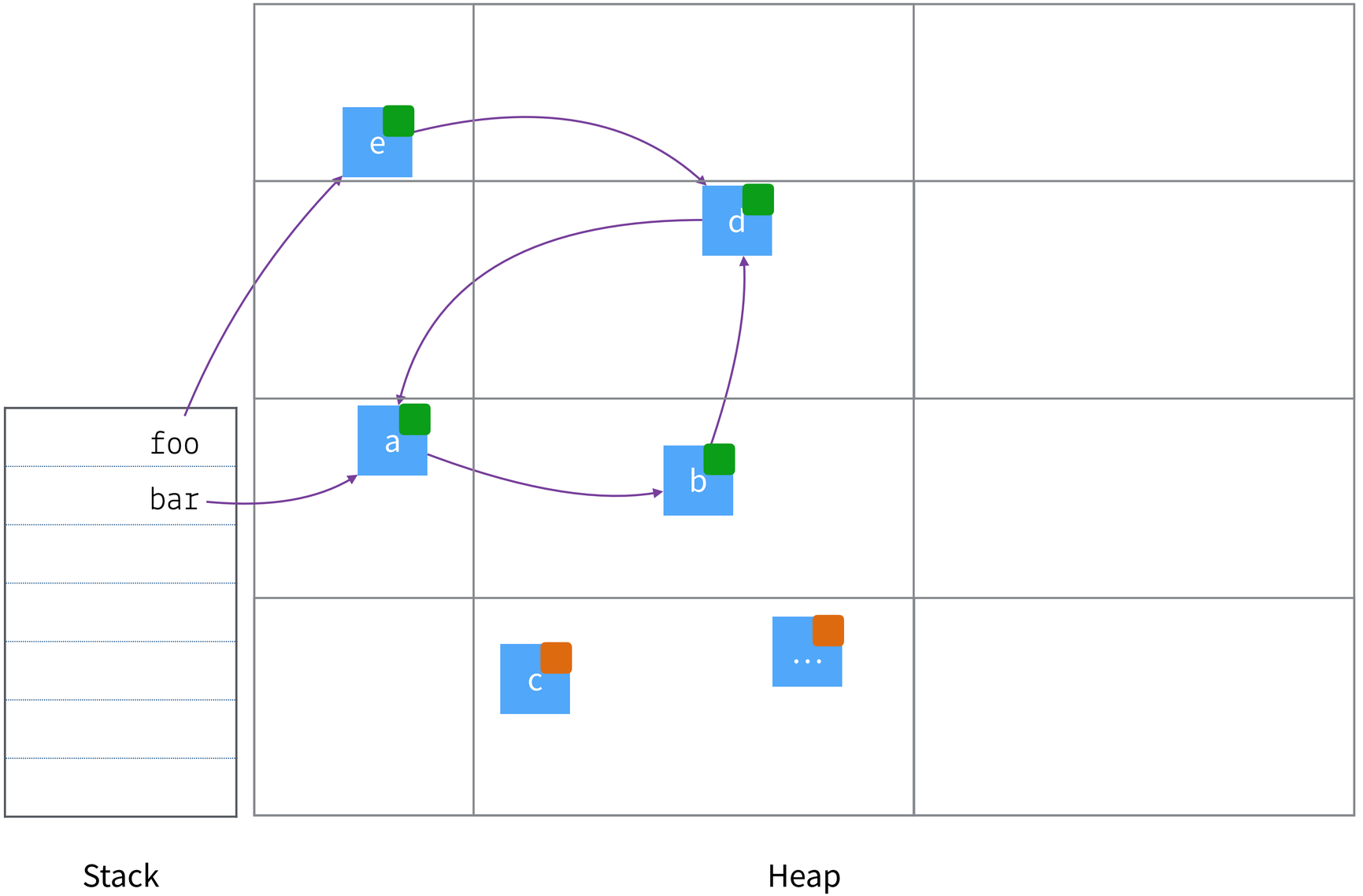
Reference counting

Tracing

Defragmentation

Deallocation

Allocation



Garbage Collection

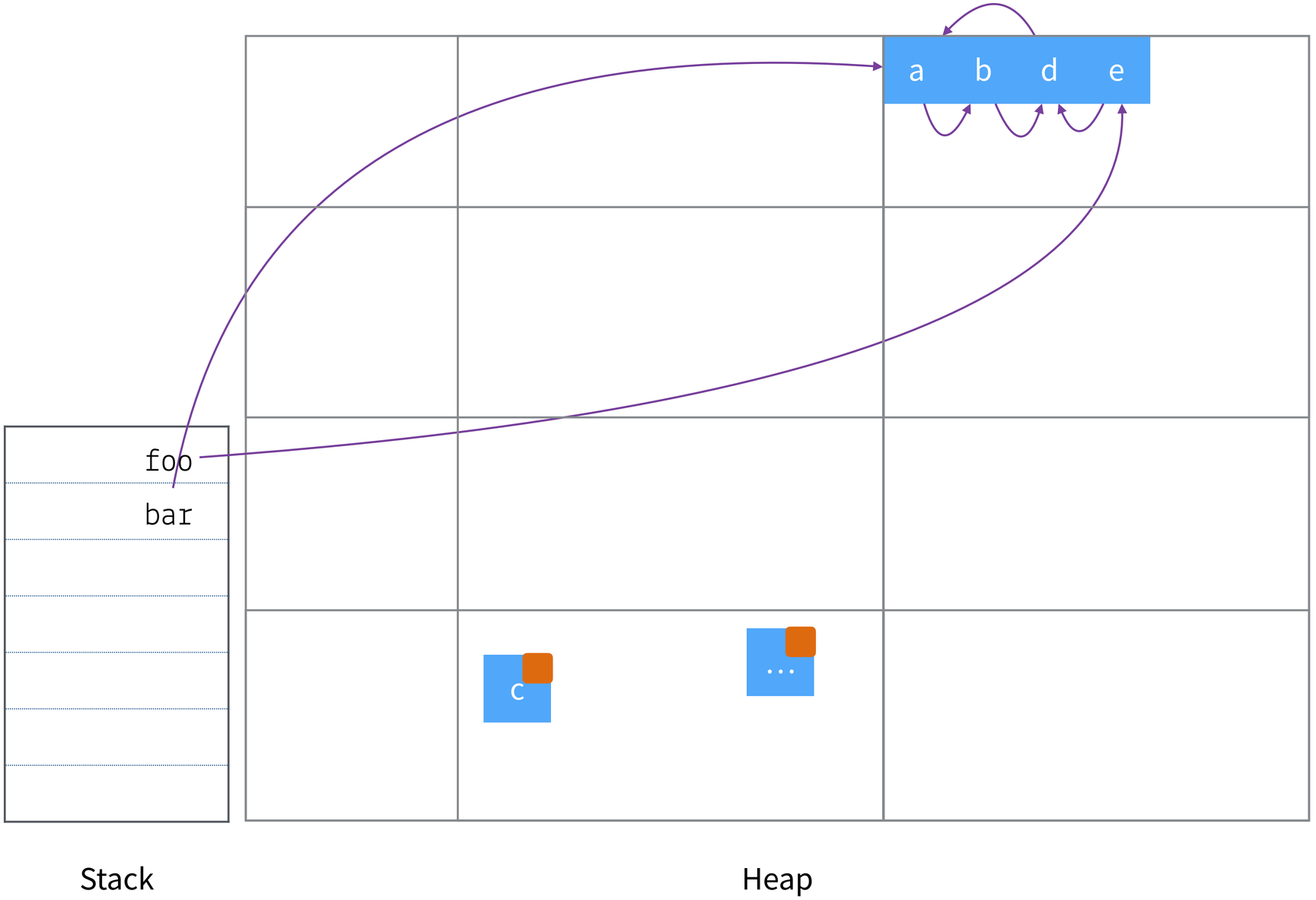
Reference counting

Tracing

Defragmentation

Deallocation

Allocation



Garbage Collection

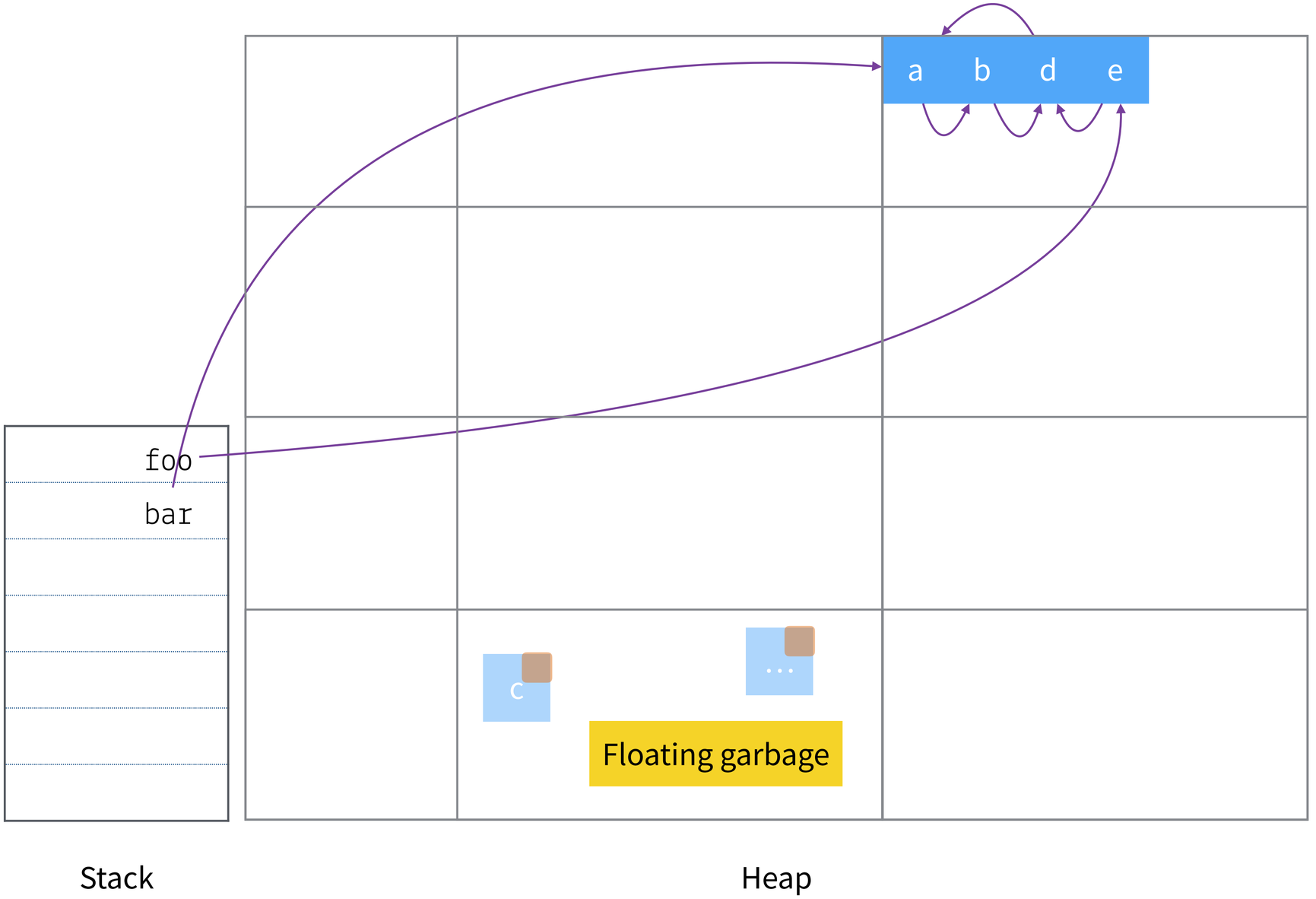
Reference counting

Tracing

Defragmentation

Deallocation

Allocation



Garbage Collection

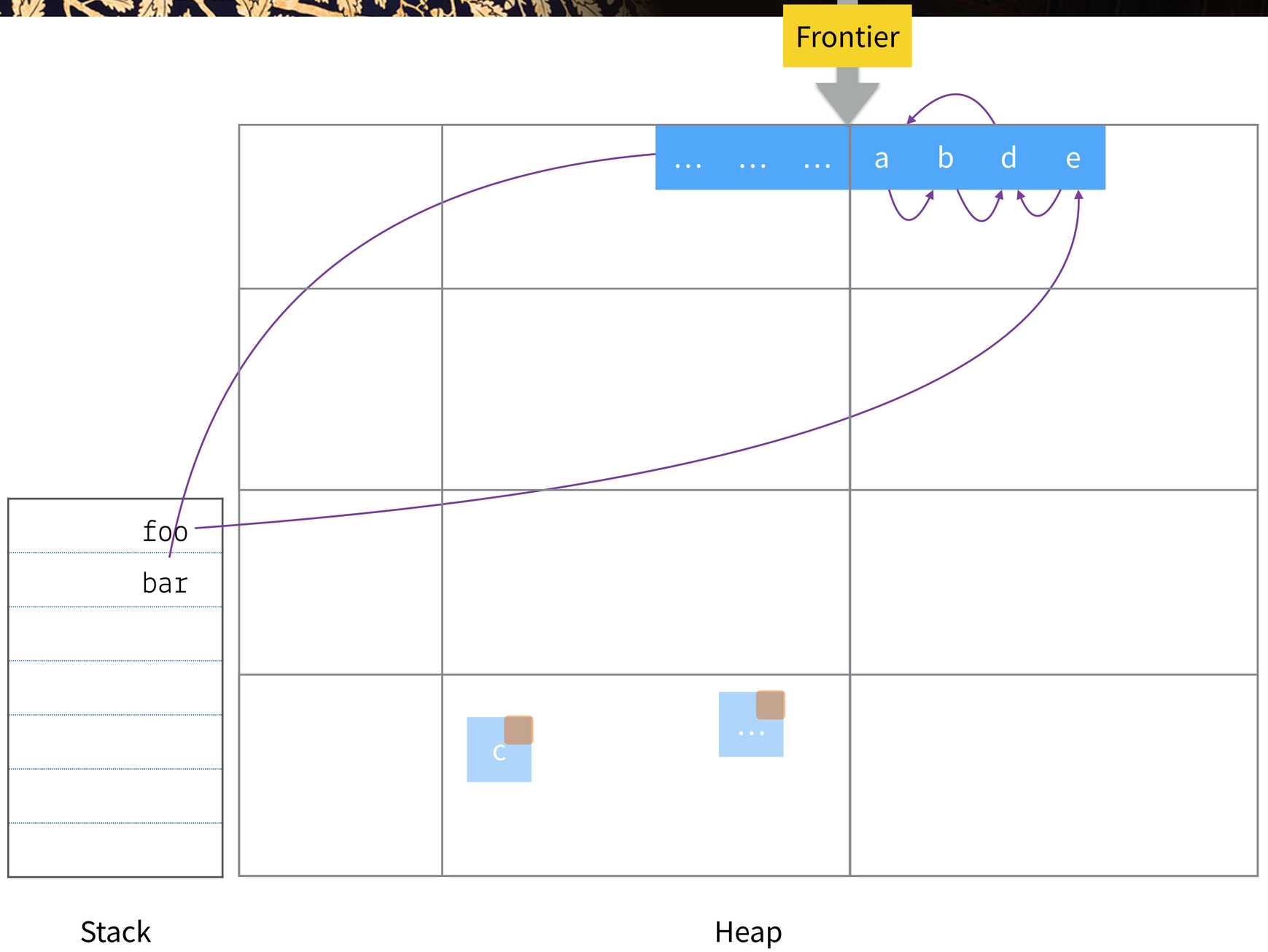
Reference counting

Tracing

Defragmentation

Deallocation

Allocation



Exempel på skräpsamlare i Java

Du kan välja vilken skräpsamlare du vill använda på kommandoraden när du kör ditt program

Parallel GC (`-XX:+UseParallelGC`)

En tracing collector som stannar hela programmet vid behov och tar bort skräp (optimerar för genomströmning)

G1: Garbage First (`-XX:+UseG1GC`, standard I modern Java)

En tracing collector som bara stannar hela programmed vid defragmentering — allt annat GC-arbete sker samtidigt med programmets normala körning

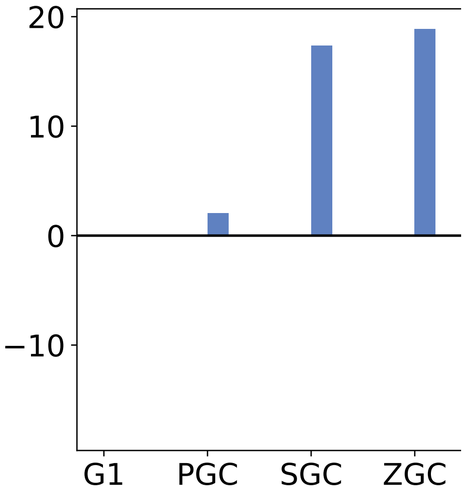
ZGC (`-XX:+UseZGC`, experimental I JDK < 15)

En tracing collector som gör allt sitt arbete parallellt med GC (optimerar för latens på bekostnad av genomströmning)

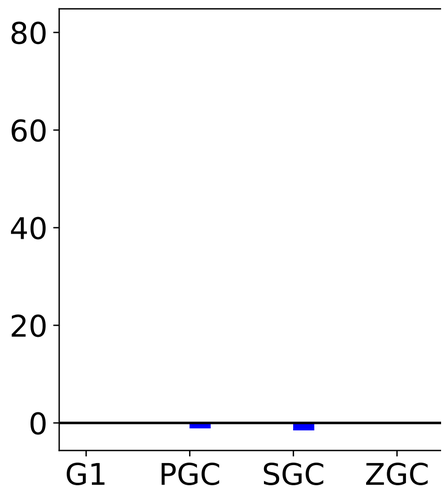
Epsilon (`-XX:+UseEpsilonGC`, experimental I JDK < 15)

Stänger av GC helt och kraschar programmet om minnet tar slut

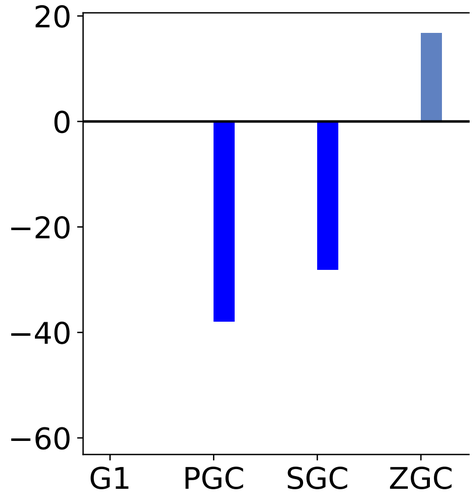
Valet av skräpsamlare kan ha enorm effekt på programmet



Time to run program



CPU Utilisation



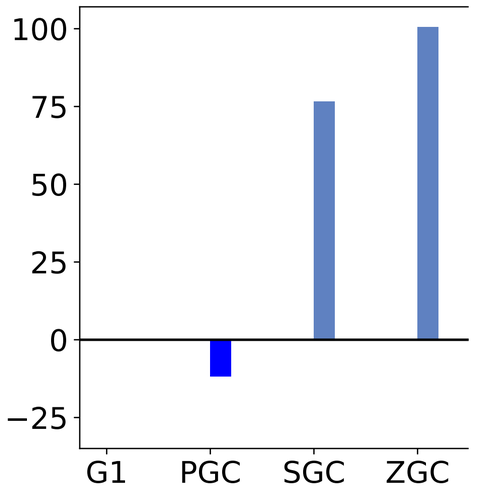
Heap size

Program: DaCapo **h2** huge heap, 4 threads
5 x 25 runs

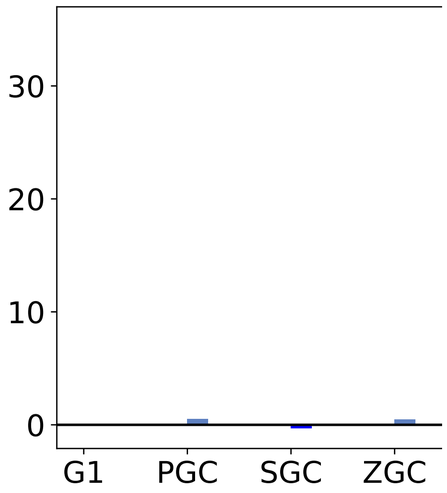
PGC = Parallel GC

SGC = Shenandoah GC

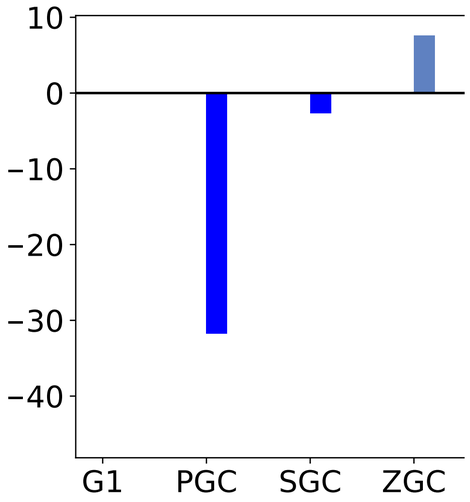
Valet av skräpsamlare kan ha enorm effekt på programmet



Time to run program



CPU Utilisation



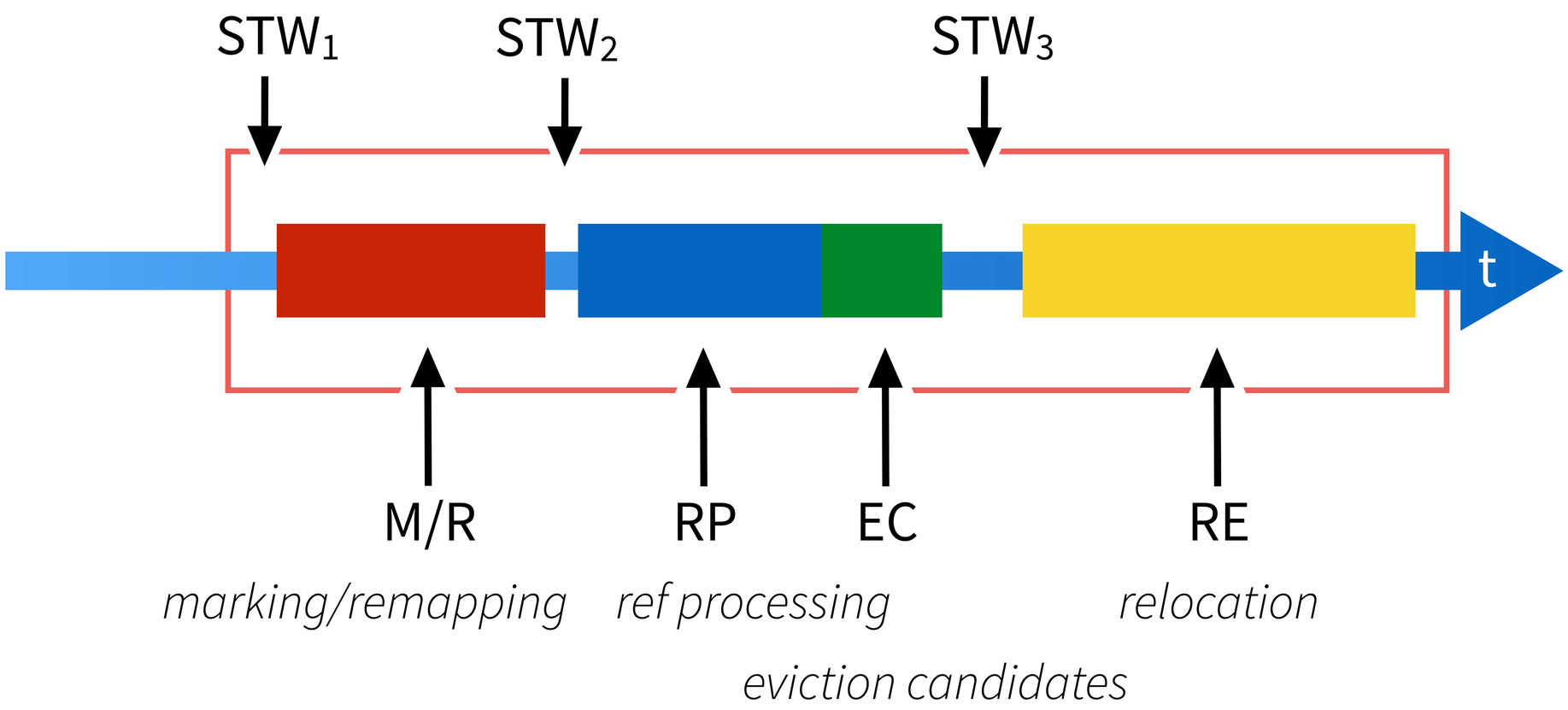
Heap size

Program: jGraphT
5 x 25 runs

PGC = Parallel GC

SGC = Shenandoah GC

The GC Cycle (ZGC)

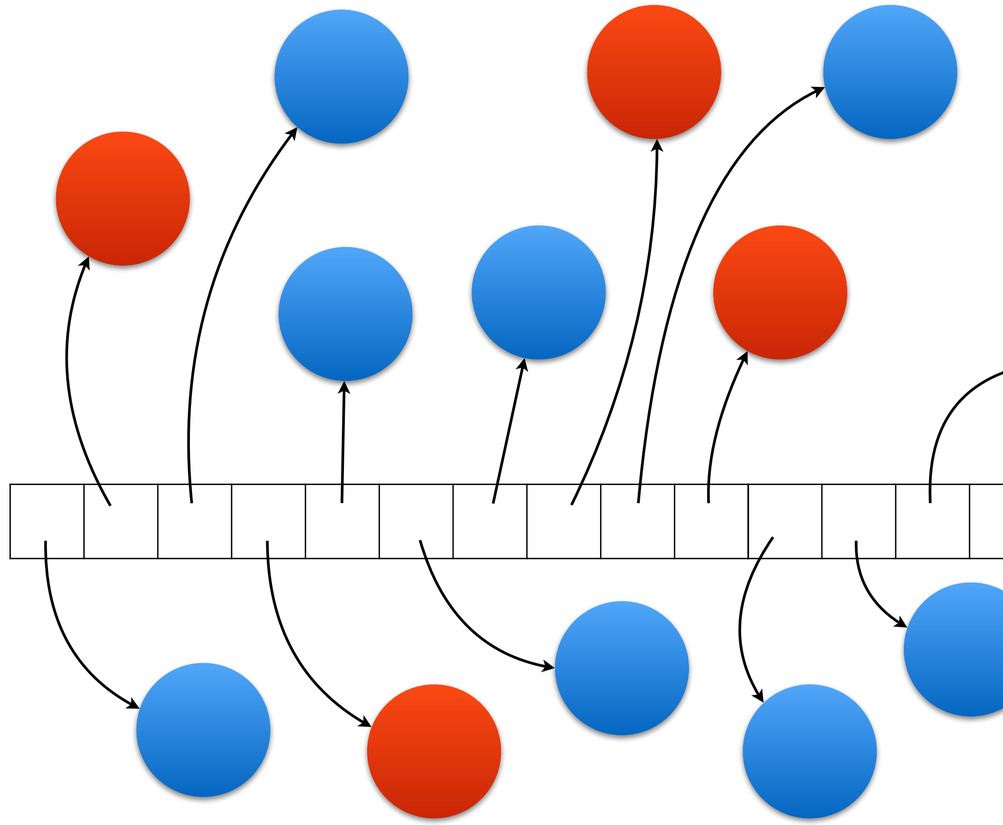


Skräpsamling för förbättrad prestanda

- Om vi ändå flyttar alla levande objekt när vi samlar skräp, kan vi flytta objekten till ”bättre” platser i minnet?

  Se till att objekt som används tillsammans lever tillsammans

  Flytta objekt så att de ligger i accessordning



Skräpsamling för förbättrad prestanda

- Om vi ändå flyttar alla levande objekt när vi samlar skräp, kan vi flytta objekten till ”bättre” platser i minnet?

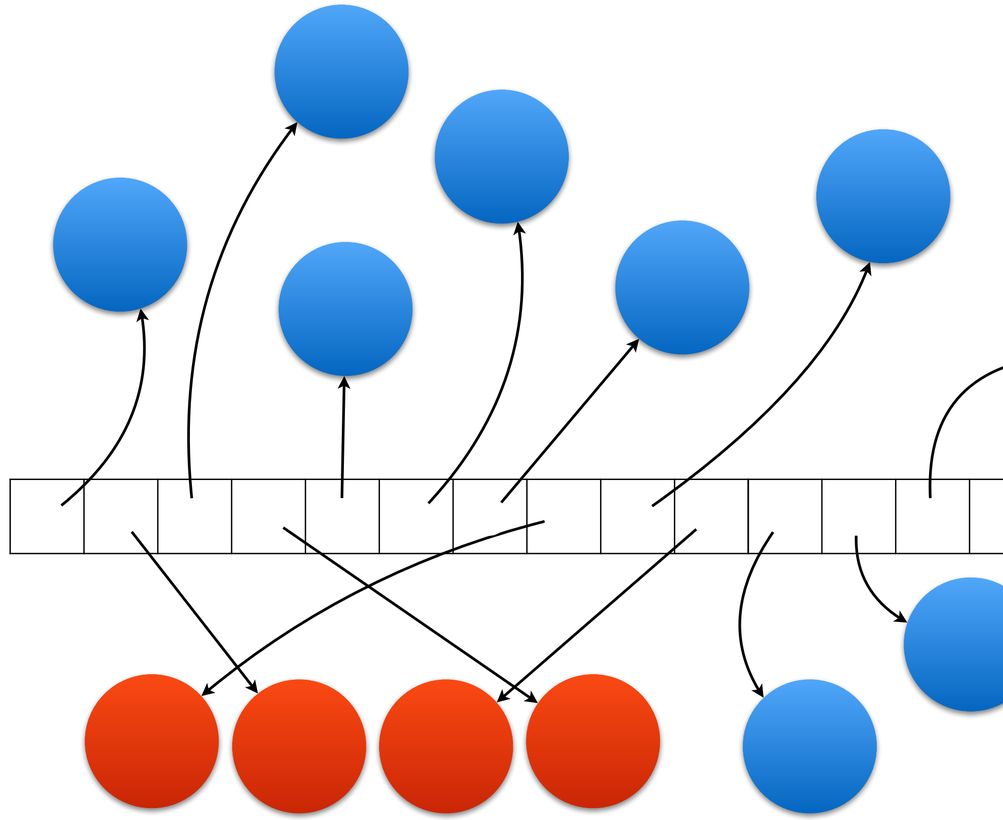
  Se till att objekt som används tillsammans lever tillsammans

  Flytta objekt så att de ligger i accessordning

- Objekt placerade i accessordning

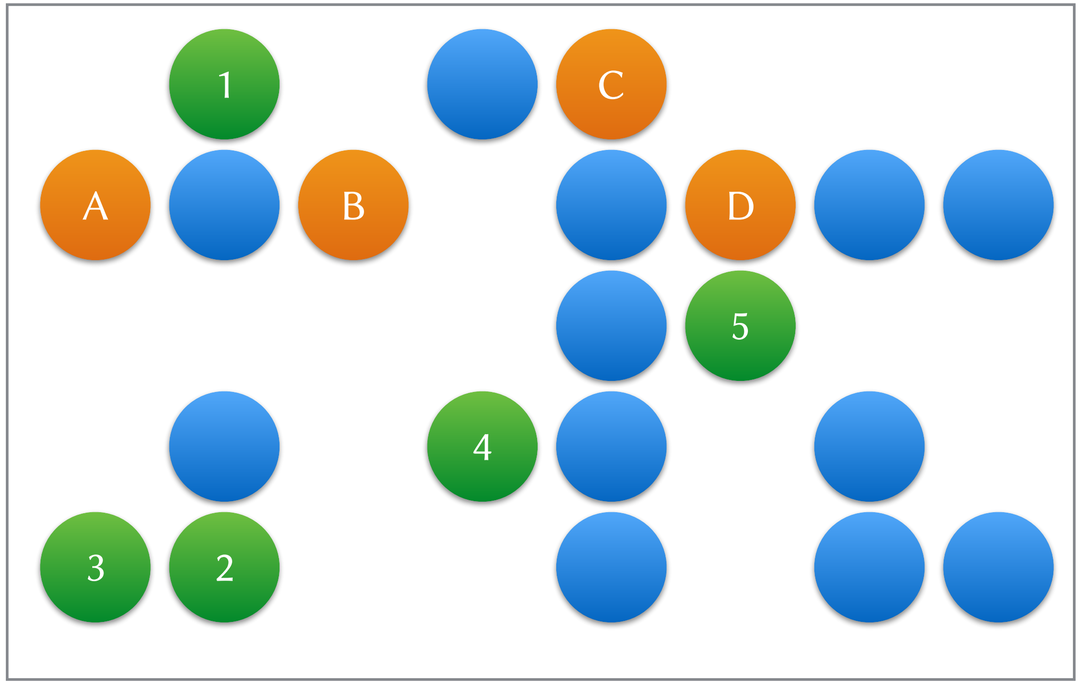
  Varje load har chans att ladda in nästa objekt som behövs

  Chans att mönster uppstår som en prefetcher förstår



HCSGC och M1

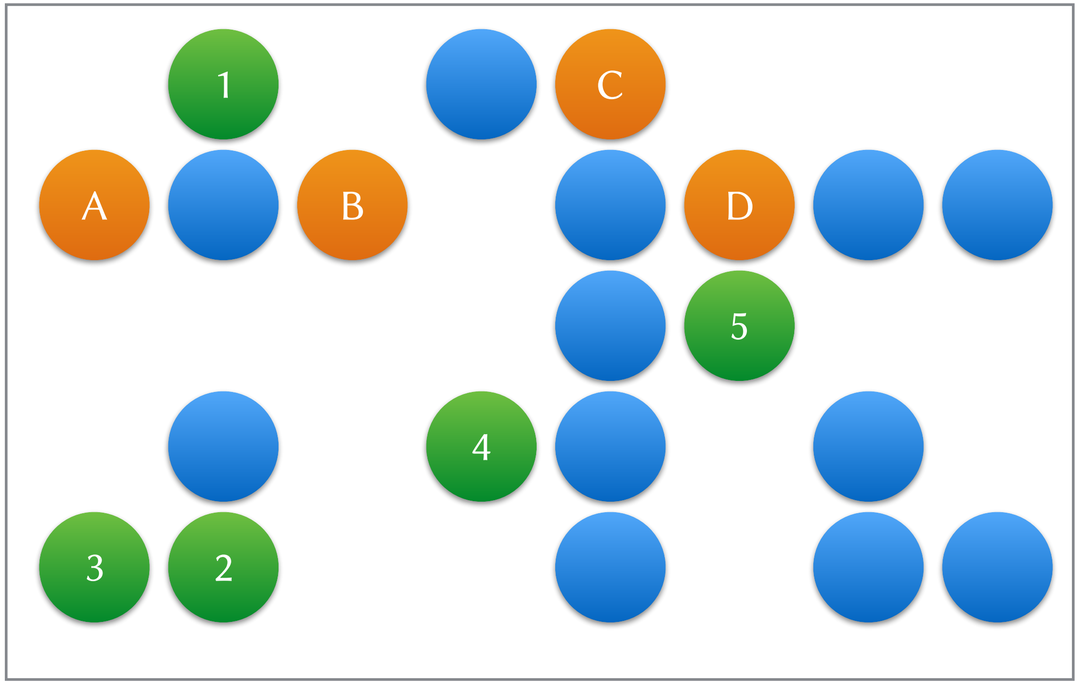
Tråd 1



Tråd 2

HCSGC och M1

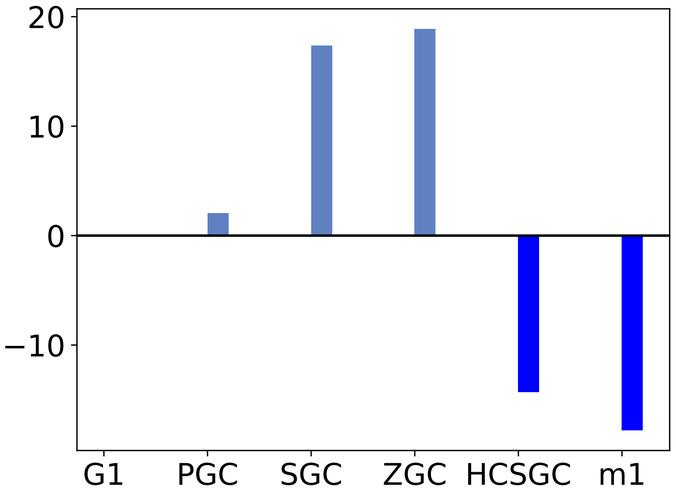
Tråd 1



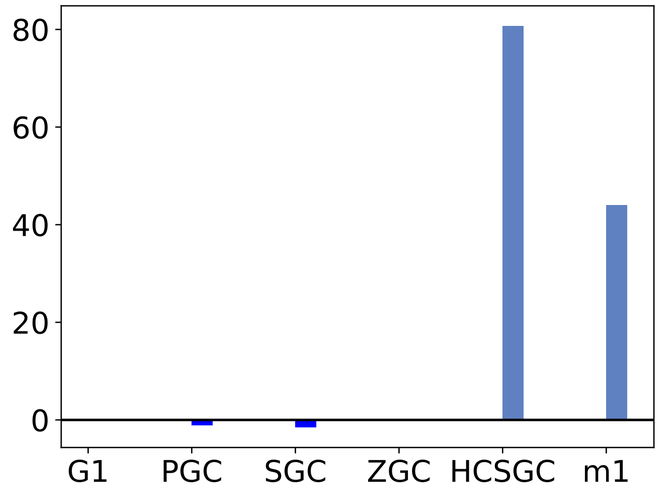
Tråd 2

Kostnad: extra flytt per objekt en gång per ”period” i systemet — risk att efterföljande accesser inte tjänar igen detta

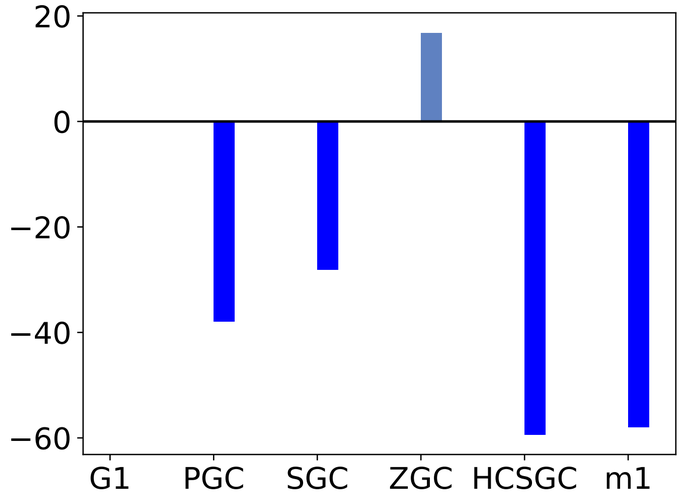
Valet av skräpsamlare kan ha enorm effekt på programmet



Time to run program



CPU Utilisation



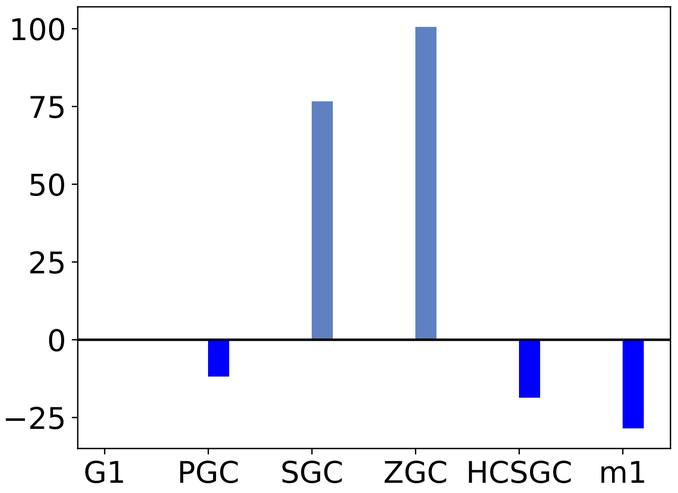
Heap size

Program: DaCapo **h2** huge heap, 4 threads
5 x 25 runs

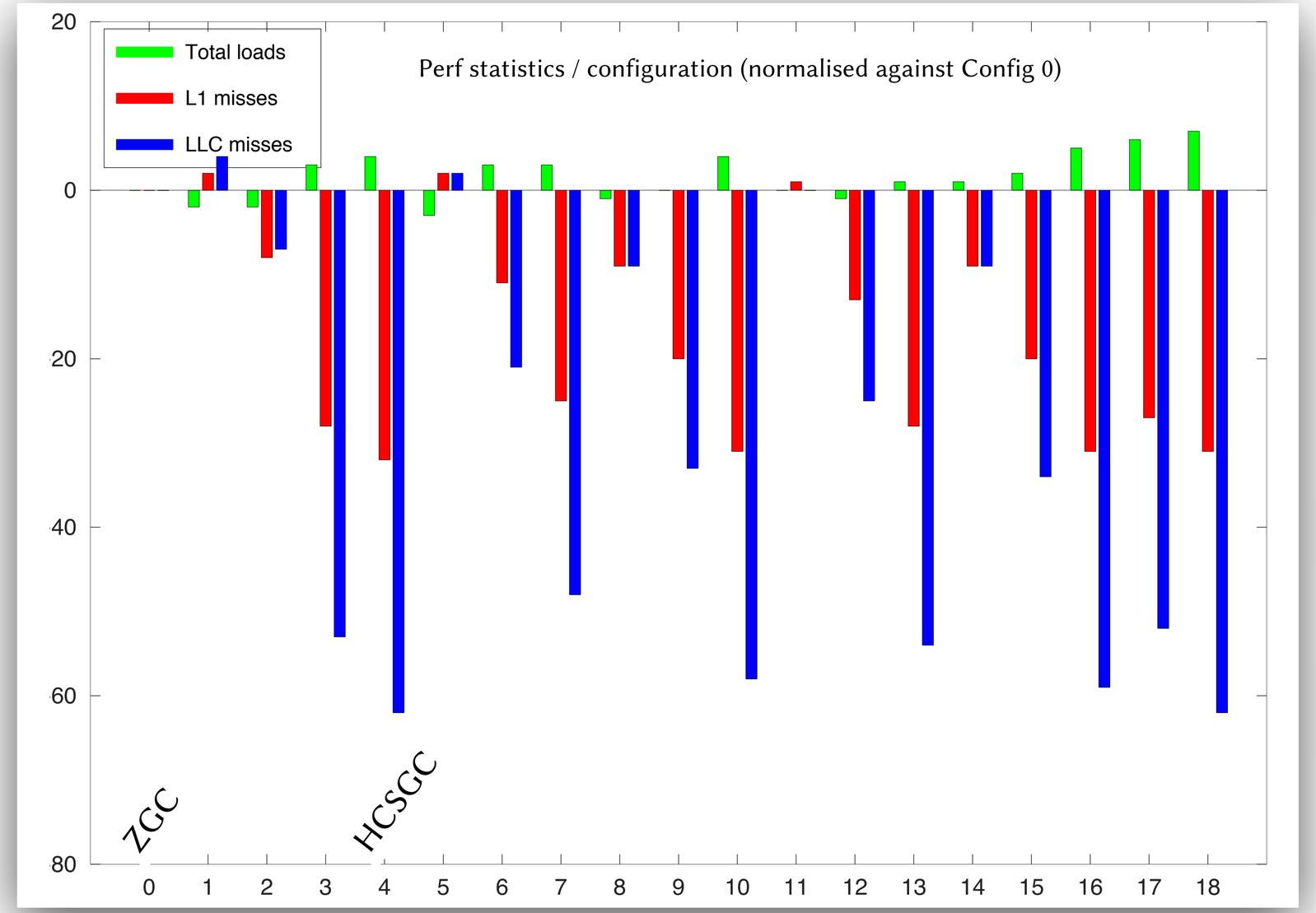
PGC = Parallel GC

SGC = Shenandoah GC

Valet av skräpsamlare kan ha enorm effekt på programmet



Time to run program



Slutsatser

- Skräpsamling underlättar programutveckling

  Men titta t.ex. på Rust de närmsta 10 åren

- Skräpsamling gör det svårare för programmeraren att förutsäga programmets beteende

  Latens-orienterad GC som t.ex. ZGC minskar effekterna av detta

- Skräpsamling kan försämra ett programs prestanda
- Skräpsamling kan förbättra ett programs prestanda
- Skräpsamling hanterar defragmentering av minnet