Runtime Environments II

Ronghui Gu

Spring 2019

Columbia University

* Course website: https://www.cs.columbia.edu/ rgu/courses/4115/spring2019

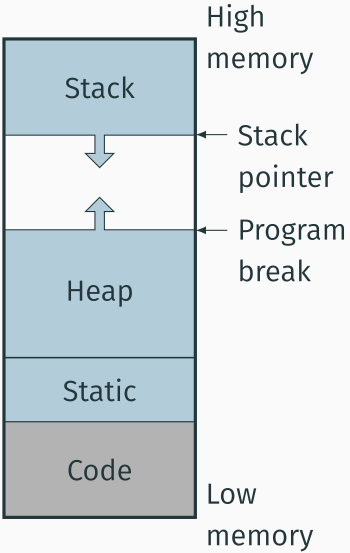
** These slides are borrowed from Prof. Edwards.

Storage Classes and Memory Layout

Stack: objects created/destroyed in last-in, first-out order

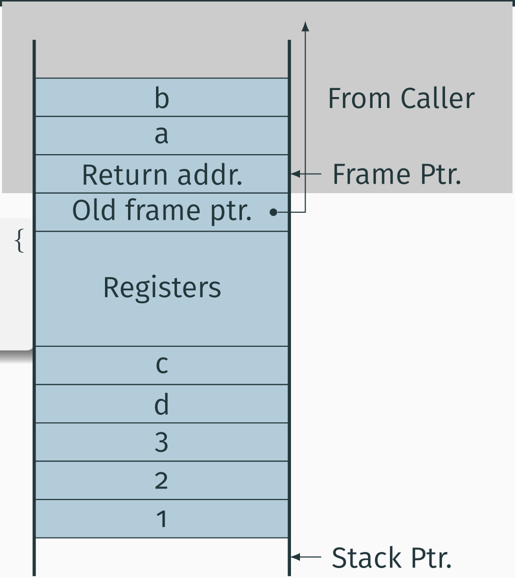
Heap: objects created/destroyed in any order; automatic garbage collection optional

Static: objects allocated at compile time; persist throughout run



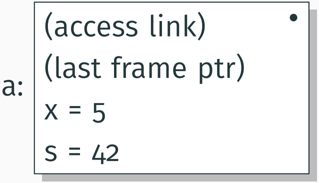
Stack-Allocated Objects

```
int foo(int a, int b) {
  int c, d;
  bar(1, 2, 3);
}
```



Implementing Nested Functions with Access Links

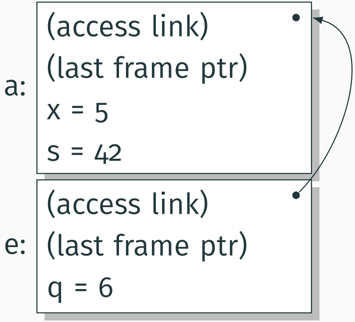
```
let a x s =
  let b y =
    let d w = w + s in
    d (y+1) in (* b *)
  let e q = b (q+1) in
e (x+1) (* a *)
```



What does “a 5 42” give?

Implementing Nested Functions with Access Links

```
let a x s =
  let b y =
    let d w = w + s in
    d (y+1) in (* b *)
  let e q = b (q+1) in
e (x+1) (* a *)
```

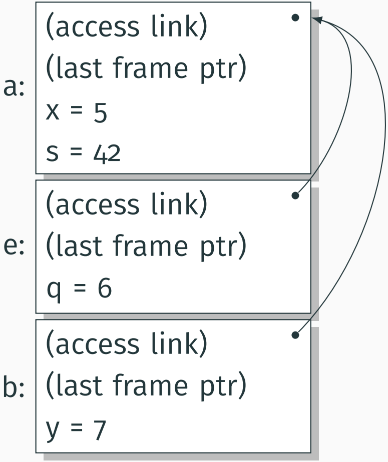


What does “a 5 42” give?

Implementing Nested Functions with Access Links

```
let a x s =
  let b y =
    let d w = w + s in
    d (y+1) in (* b *)
  let e q = b (q+1) in
e (x+1) (* a *)
```

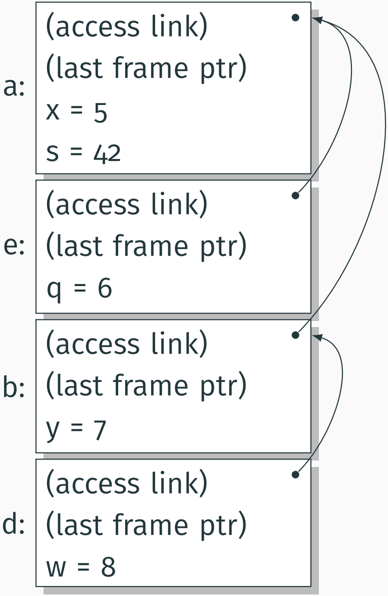
What does “a 5 42” give?



Implementing Nested Functions with Access Links

```
let a x s =
  let b y =
    let d w = w + s in
    d (y+1) in (* b *)
  let e q = b (q+1) in
e (x+1) (* a *)
```

What does “a 5 42” give?

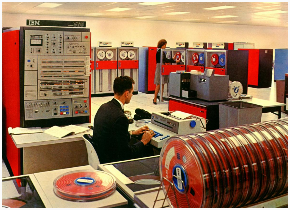


In-Memory Layout Issues

Layout of Records and Unions

Modern processors have byte-addressable memory.

0
1
2
3



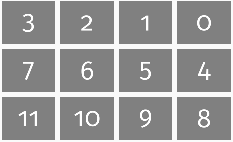
The IBM 360 (c. 1964) helped to popularize byte-addressable memory.

Many data types (integers, addresses, floating-point numbers) are wider than a byte.

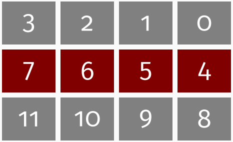
16-bit integer:			1	0
32-bit integer:	3	2	1	0

Layout of Records and Unions

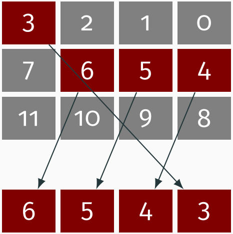
Modern memory systems read data in 32-, 64-, or 128-bit chunks:



Reading an aligned 32-bit value is fast: a single operation.



How about reading an unaligned value?

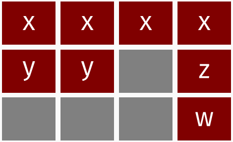


Padding

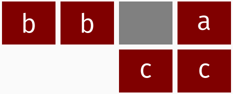
To avoid unaligned accesses, the C compiler pads the layout of unions and records. Rules:

- Each n-byte object must start on a multiple of n bytes (no unaligned accesses).
- Any object containing an n-byte object must be of size mn for some integer m (aligned even when arrayed).

```c
struct padded {
  int x;    /* 4 bytes */
  char z;   /* 1 byte  */
  short y;  /* 2 bytes */
  char w;   /* 1 byte  */
};
```




```c
struct padded {
  char a;   /* 1 byte  */
  short b;  /* 2 bytes */
  short c;  /* 2 bytes */
};
```

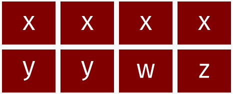


Padding

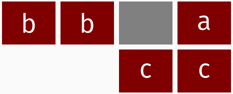
To avoid unaligned accesses, the C compiler pads the layout of unions and records. Rules:

- Each n-byte object must start on a multiple of n bytes (no unaligned accesses).
- Any object containing an n-byte object must be of size mn for some integer m (aligned even when arrayed).

```c
struct padded {
  int x;    /* 4 bytes */
  char z;   /* 1 byte  */
  char w;   /* 1 byte  */
  short y;  /* 2 bytes */
};
```

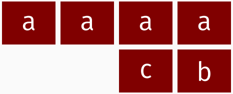



```c
struct padded {
  char a;   /* 1 byte  */
  short b;  /* 2 bytes */
  short c;  /* 2 bytes */
};
```

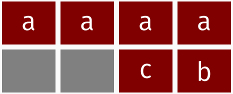


Padding: (1) or (2)?

```c
struct padded {
  int a;   /* 4 bytes  */
  char b; /* 1 byte */
  char c; /* 1 byte */
};
```



(1)



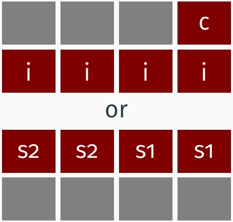
(2)

Unions

A C *union* shares one space among all fields

```c
union intchar {
  int i;     /* 4 bytes */
  char c;    /* 1 byte  */
};
```


```c
union twostructs {
  struct {
    char c;     /* 1 byte */
    int i;      /* 4 bytes */
  } a;
  struct {
    short s1;  /* 2 bytes */
    short s2;  /* 2 bytes */
  } b;
};
```



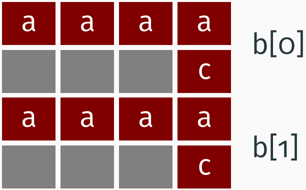
Arrays

Basic policy in C: an array is just one object after another in memory.

```c
int a[10];
```

What if we remove rule 2 of padding?

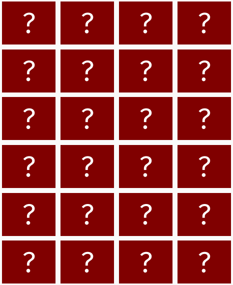
```c
struct {
  int a;
  char c;
} b[2];
```



Arrays and Aggregate types

The largest primitive type dictates the alignment

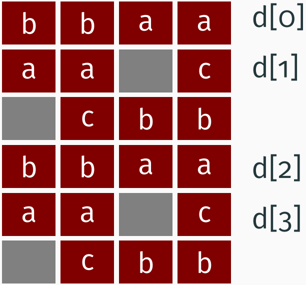
```
struct {
  short a;
  short b;
  char c;
} d[4];
```



Arrays and Aggregate types

The largest primitive type dictates the alignment

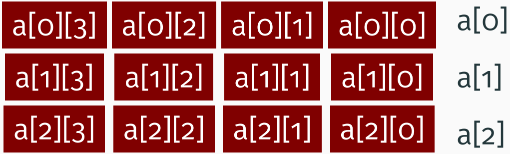
```
struct {
  short a;
  short b;
  char c;
} d[4];
```



Arrays of Arrays

```
char a[4];
```


```
char a[3][4];
```



The Heap

Heap-Allocated Storage

A *heap* is a region of memory where blocks can be dynamically allocated and deallocated in any order.

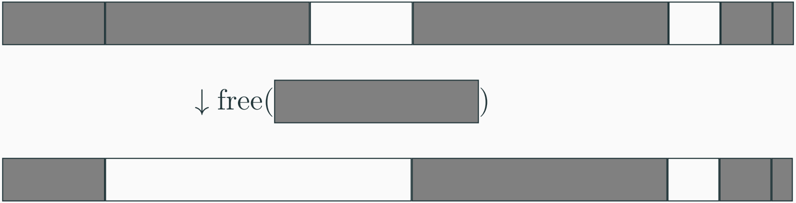
Dynamic Storage Allocation in C

```
struct point {
   int x, y;
};

int play_with_points(int n)
{
  int i;
  struct point *points;

  points = malloc(n * sizeof(struct point));

  for ( i = 0 ; i < n ; i++ ) {
    points[i].x = random();
    points[i].y = random();
  }

  /* do something with the array */

  free(points);
}
```

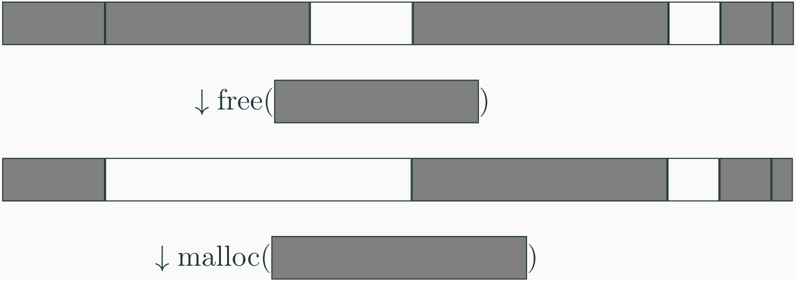
Dynamic Storage Allocation

Dynamic Storage Allocation

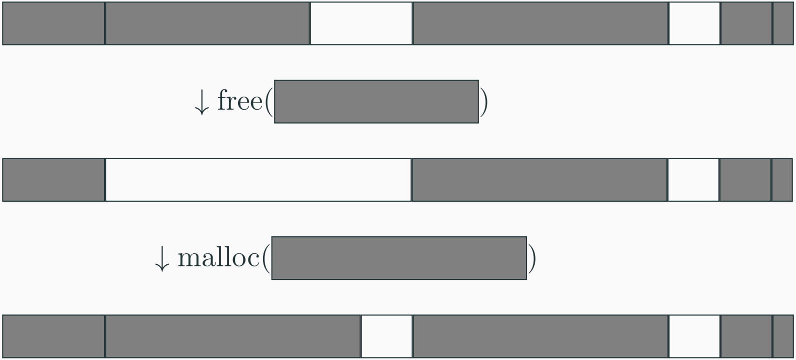
Dynamic Storage Allocation



Dynamic Storage Allocation



Dynamic Storage Allocation



Dynamic Storage Allocation

Rules:

- Each allocated block contiguous
- Blocks stay fixed once allocated

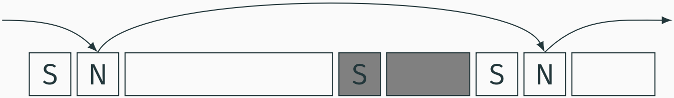
malloc()

free()

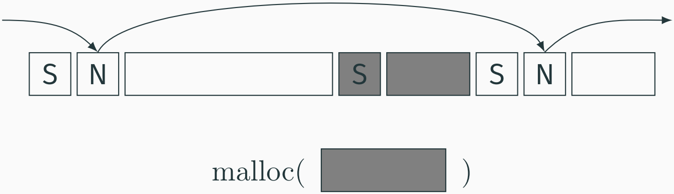
Simple Dynamic Storage Allocation

- Maintaining information about free memory
  - Simplest: Linked list
- The algorithm for locating a suitable block
  - Simplest: First-fit
- The algorithm for freeing an allocated block
  - Simplest: Coalesce adjacent free blocks

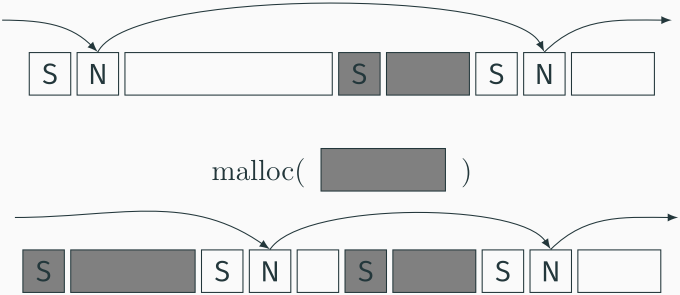
Simple Dynamic Storage Allocation



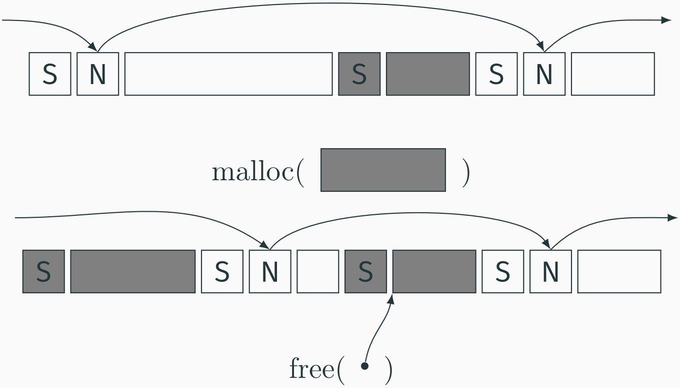
Simple Dynamic Storage Allocation



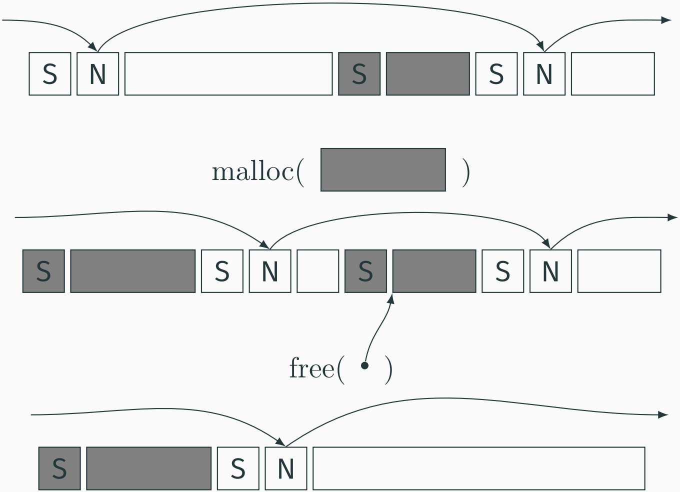
Simple Dynamic Storage Allocation



Simple Dynamic Storage Allocation



Simple Dynamic Storage Allocation



Fragmentation

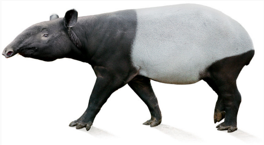
malloc( ) seven times give

free() four times gives

malloc( ) ?

Need more memory; can’t use fragmented memory.

Zebra

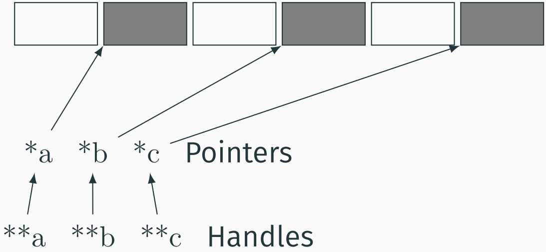


Tapir

Fragmentation and Handles

Standard CS solution: Add another layer of indirection.

Always reference memory through “handles.”

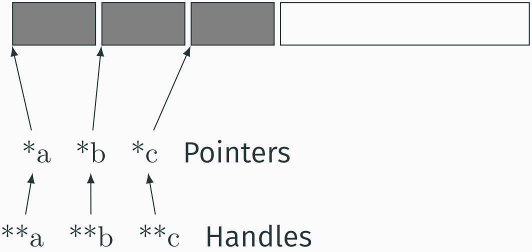


The original Macintosh did this to save memory.

Fragmentation and Handles

Standard CS solution: Add another layer of indirection.

Always reference memory through “handles.”



The original Macintosh did this to save memory.

Automatic Garbage Collection

Automatic Garbage Collection

Entrust the runtime system with freeing heap objects

Now common: Java, C#, Javascript, Python, Ruby, OCaml and most functional languages

Advantages?

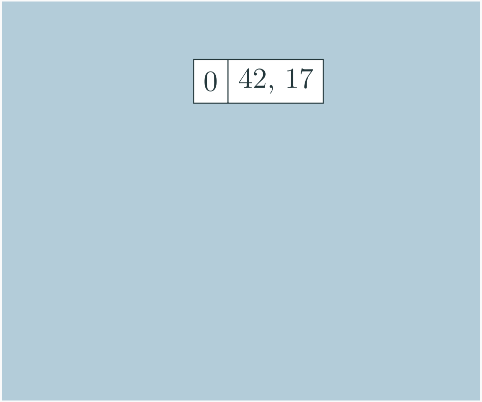
Disadvantages?

Reference Counting

What and when to free?

- Maintain count of references to each object
- Free when count reaches zero

```
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```

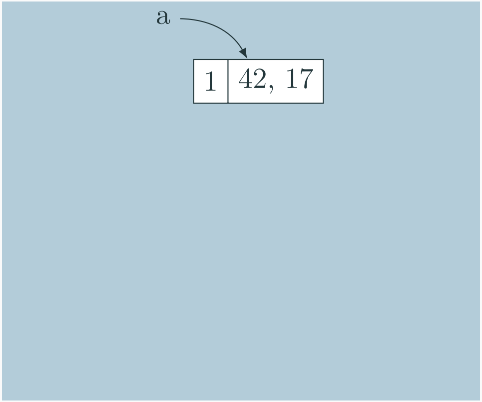


Reference Counting

What and when to free?

- Maintain count of references to each object
- Free when count reaches zero

```
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```

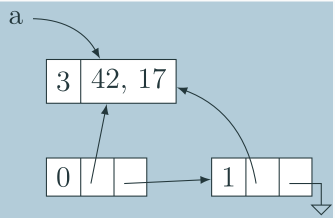


Reference Counting

What and when to free?

- Maintain count of references to each object
- Free when count reaches zero

```
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```

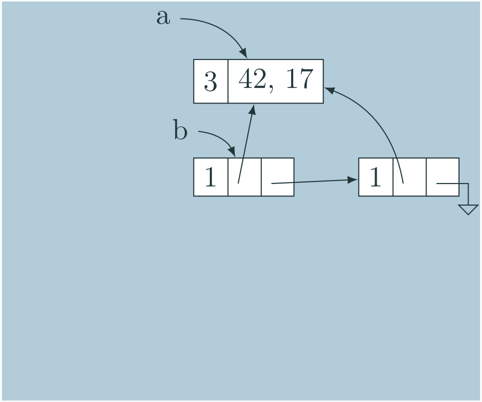


Reference Counting

What and when to free?

- Maintain count of references to each object
- Free when count reaches zero

```
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```

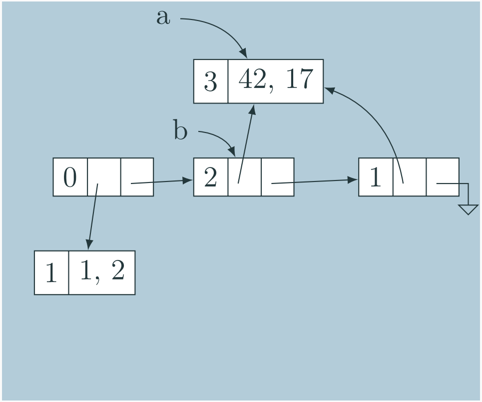


Reference Counting

What and when to free?

- Maintain count of references to each object
- Free when count reaches zero

```
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```

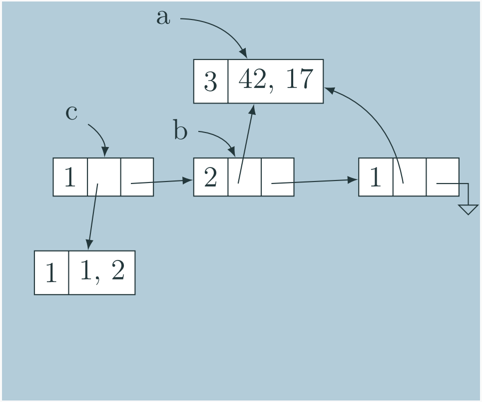


Reference Counting

What and when to free?

- Maintain count of references to each object
- Free when count reaches zero

let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b

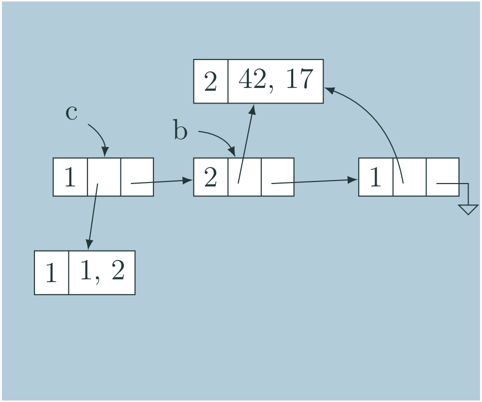


Reference Counting

What and when to free?

- Maintain count of references to each object
- Free when count reaches zero

let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b

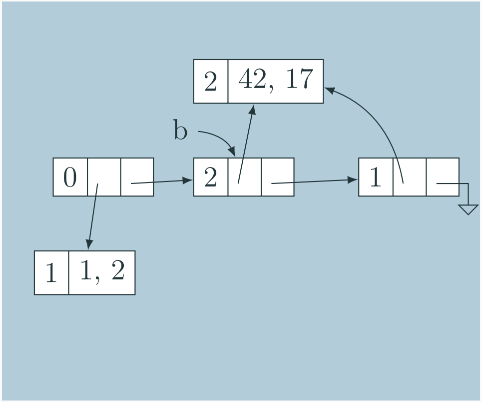


Reference Counting

What and when to free?

- Maintain count of references to each object
- Free when count reaches zero

let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b

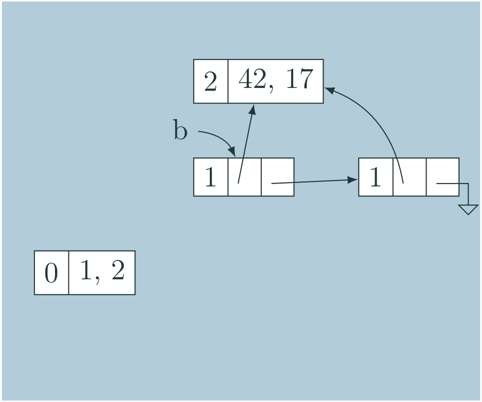


Reference Counting

What and when to free?

- Maintain count of references to each object
- Free when count reaches zero

let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b

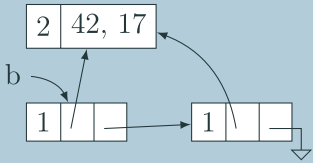


Reference Counting

What and when to free?

- Maintain count of references to each object
- Free when count reaches zero

let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b



Issues with Reference Counting

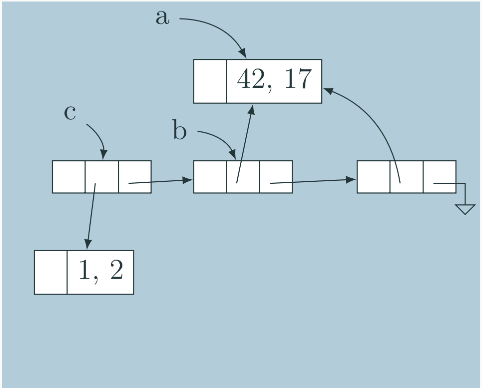
Circular structures defy reference counting?

Mark-and-Sweep

What and when to free?

- Stop-the-world algorithm invoked when memory full
- Breadth-first-search marks all reachable memory
- All unmarked items freed

let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b

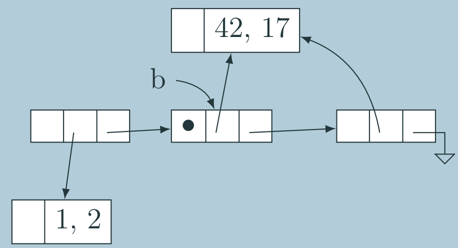


Mark-and-Sweep

What and when to free?

- Stop-the-world algorithm invoked when memory full
- Breadth-first-search marks all reachable memory
- All unmarked items freed

let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b

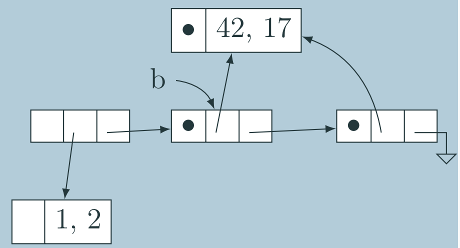


Mark-and-Sweep

What and when to free?

- Stop-the-world algorithm invoked when memory full
- Breadth-first-search marks all reachable memory
- All unmarked items freed

let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b

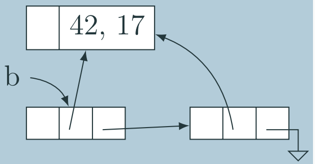


Mark-and-Sweep

What and when to free?

- Stop-the-world algorithm invoked when memory full
- Breadth-first-search marks all reachable memory
- All unmarked items freed

```
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```



Mark-and-Sweep

Mark-and-sweep is faster overall; may induce big pauses

Mark-and-compact variant also moves or copies reachable objects to eliminate fragmentation

Incremental garbage collectors try to avoid doing everything at once

Most objects die young; generational garbage collectors segregate heap objects by age

Parallel garbage collection tricky

Real-time garbage collection tricky