Memory Management

```
char **names;
widget listbox, button;

names = get_students_from_database();

listbox = make_list_control();
set_choices(listbox, names);

add_widget_to_window(directory, listbox);

button = make_button();
set_widget_label(button, "Send Email");
set_widget_callback(button, send_email, names);

....
```

Memory Management

```
char **names;
widget listbox, button;

names = get_students_from_database();

listbox = make_list_control();
set_choices(listbox, names);

add_widget_to_window(directory, listbox);

button = make_button();
set_widget_label(button, "Send Email");
set_widget_callback(button, send_email, names);

....
```

Who/when to free memory for **`names`**, **`listbox`**, ...?

Automatic memory management makes calling **`free`** someone else's problem

Allocation Benchmark

```
struct node {
  struct node *left;
  struct node *right;
};
```

Allocation Benchmark

```
struct node *make_tree(int depth) {
  if (depth == 0)
    return NULL;
  else {
    /* sumulate useful work: */
    destroy_tree(make_tree(depth-1));

    struct node *t = allocate();
    t->left = make_tree(depth-1);
    t->right = make_tree(depth-1);

    return t;
  }
}
```

Allocation Benchmark

```
void destroy_tree(struct node *n) {
  if (n != NULL) {
    destroy_tree(n->left);
    destroy_tree(n->right);
    deallocate(n);
  }
}
```

Allocation Benchmark

```
struct node *make_tree(int depth) {
  if (depth == 0)
    return NULL;
  else {
    /* sumulate useful work: */
    make_tree(depth-1);

    struct node *t = allocate();
    t->left = make_tree(depth-1);
    t->right = make_tree(depth-1);

    return t;
  }
}

void destroy_tree(struct node *n) {
 /* do nothing */
}
```

Automatic Memory Management

Two main flavors:

- ***Reference counting***
  - ✓ Timing of `free` more predictable
  - ✗ Does not handle cyclic references
- ***Garbage collection***
  - ✗ Timing of `free` less predictable
  - ✓ Handles cyclic references

Reference Counting

Reference counting:

a way to know whether an object has other users

- Attach a count to every object, starting at 0
- When saving a pointer to an object, increment the object's count

  saving = setting variable or field in another object
- When removing a pointer to an object, decrement the object's count

  removing = variable done or variable/field updated
- When a count is decremented to 0, decrement counts for other objects referenced by the object, then free

Reference Counting

Reference counting:

a way to know whether an object has other users

Manual reference counting

```
....
listbox = make_list_control();
retain_widget(listbox);
add_widget_to_window(directory, listbox);
release_widget(listbox);
....
```

Reference Counting

Reference counting:

a way to know whether an object has other users

Manual reference counting

```
void add_widget_to_window(window *win, widget *wig) {
  retain_widget(wig);
  win->child = wig;
  ....
}

void remove_window_child(window *win) {
  ....
  release_widget(win->child);
  win->child = NULL;
  ....
}
```

Reference Counting

Reference counting:

a way to know whether an object has other users

Automatic reference counting

```
struct node *make_tree(int depth) {
  if (depth == 0)
    return NULL;
  else {
    /* sumulate useful work: */
    make_tree(depth-1);

    struct node *t = allocate();
    t->left = make_tree(depth-1);
    t->right = make_tree(depth-1);

    return t;
  }
}
```

original

```
struct node *make_tree(int depth) {
  if (depth == 0)
    return NULL;
  else {
    /* sumulate useful work: */
    struct node *n = NULL;
    SET_NODE(n, make_tree(depth-1));
    UNSET_NODE(n);

    struct node *t = NULL;
    SET_NODE(t, allocate());
    SET_NODE(t->left, make_tree(depth-1));
    SET_NODE(t->right, make_tree(depth-1));

    UNSET_RETURN_NODE(t);

    return t;
  }
}
```

converted

Reference Counting

Reference counting:

a way to know whether an object has other users

Automatic reference counting

```
struct node *make_tree(int depth) {
  if (depth == 0)
    return NULL;
  else {
    /* sumulate useful work: */
    make_tree(depth-1);

    struct node *t = allocate();
    t->left = make_tree(depth-1);
    t->right = make_tree(depth-1);

    return t;
  }
}
```

original

```
struct node *make_tree(int depth) {
  if (depth == 0)
    return NULL;
  else {
    /* sumulate useful work: */
    struct node *t = NULL;
    SET_NODE(t, allocate());
    SET_NODE(t->left, make_tree(depth-1));
    SET_NODE(t->right, make_tree(depth-1));

    UNSET_RETURN_NODE(t);

    return t;
  }
}
```

converted

Every allocation must be explicitly received

Reference Counting

Reference counting:

a way to know whether an object has other users

Automatic reference counting

```
struct node *make_tree(int dep
  if (depth == 0)
    return NULL;
  else {
    /* sumulate useful work: */
    make_tree(depth-1);

    struct node *t = allocate();
    t->left = make_tree(depth-1);
    t->right = make_tree(depth-1);

    return t;
  }
}
```

original

Every allocation must be explicitly received

```
struct node *make_tree(int depth) {
    struct no e *n = NULL;
    SET_NODE(n, make_tree(depth-1));
    UNSET_NODE(n);

    struct node *t = NULL;
    SET_NODE(t, allocate());
    SET_NODE(t->left, make_tree(depth-1));
    SET_NODE(t->right, make_tree(depth-1));

    UNSET_RETURN_NODE(t);

    return t;
  }
}
```

converted

Reference Counting

Reference counting:

a way to know whether an object has other users

Automatic reference counting

```
struct node *make_tree(int depth) {
  if (depth == 0)
    return NULL;
  else {
    /* sumulate useful work: */
    make_tree(depth-1);

    struct node *t = allocate();
    t->left = make_tree(depth-1);
    t->right = make_tree(depth-1);

    return t;
  }
}
```

original

```
struct node *make_tree(int depth) {
  if (depth == 0)
    SET_NODE(n, make_tree(depth-1));
    UNSET_NODE(n);

    struct node *t = NULL;
    SET_NODE(t, allocate());
    SET_NODE(t->left, make_tree(depth-1));
    SET_NODE(t->right, make_tree(depth-1));

    UNSET_RETURN_NODE(t);

    return t;
  }
}
```

converted

Make every "forgetting" of a reference explicit

Reference Counting

Reference counting:

a way to know whether an object has other users

Automatic reference counting

```
struct node *make_tree(int depth) {
  if (depth == 0)
    return NULL;
  else {
    /* sumulate useful work: */
    make_tree(depth-1);

    struct node *t = allocate();
    struct node *l = make_tree(depth-
    t->left = l;
    t->right = make_tree(depth-1);

    return t;
  }
}
```

```
struct node *make_tree(int depth) {
  if (depth == 0)
    SET_NODE(n, make_tree(depth-1));
    UNSET_NODE(n);

    struct node *t = NULL;
    SET_NODE(t, allocate());
    struct node *l;
    SET_NODE(l, make_tree(depth-1));
    SET_NODE(t->left, l);
    UNSET_NODE(l);
    SET_NODE(t->right, make_tree(depth-1));

    UNSET_RETURN_NODE(t);

    return t;
```

Make every "forgetting" of a reference explicit

Reference Counting

Reference counting:

a way to know whether an object has other users

Automatic reference counting

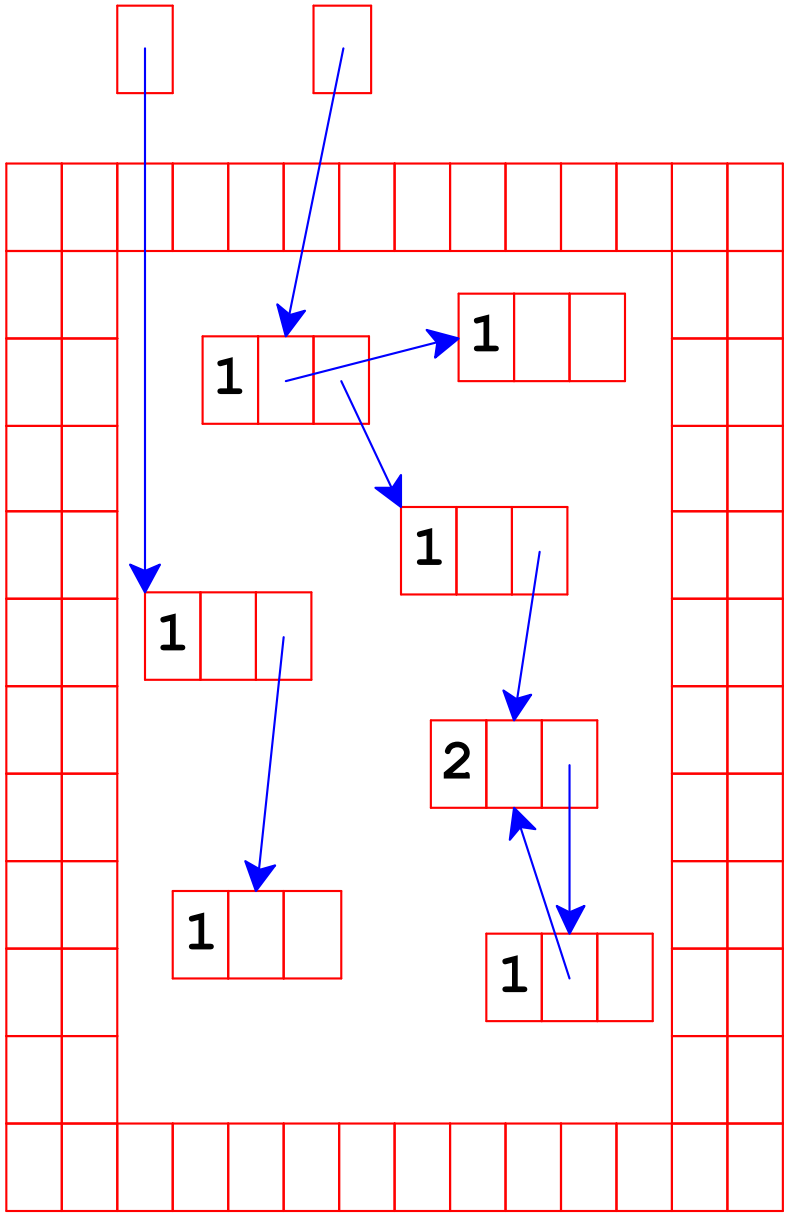
```
struct node *make_tree(int depth) {
  if (depth == 0)
    return NULL;
  else {
    /* sumulate useful work: */
    make_tree(depth-1);

    struct node *t = allocate();
    t->left = make_tree(depth-1);
    t->right = make_tree(depth-1);
    return t;
  }
}
```

Allocation return: a "forgetting" combined with an allocation

```
struct node *make_tree(int depth) {
  if (depth == 0)
    return NULL;
  else {
    /* sumulate useful work: */
    struct node *n = NULL;
    SET_NODE(n, make_tree(depth-1));
    UNSET_NODE(n);

    struct node *t = NULL;
    SET_NODE(t, allocate());
    SET_NODE(t->left, make_tree(depth-1));
    SET_NODE(t->right, make_tree(depth-1));

    UNSET_RETURN_NODE(t);

    return t;
  }
}
```

converted

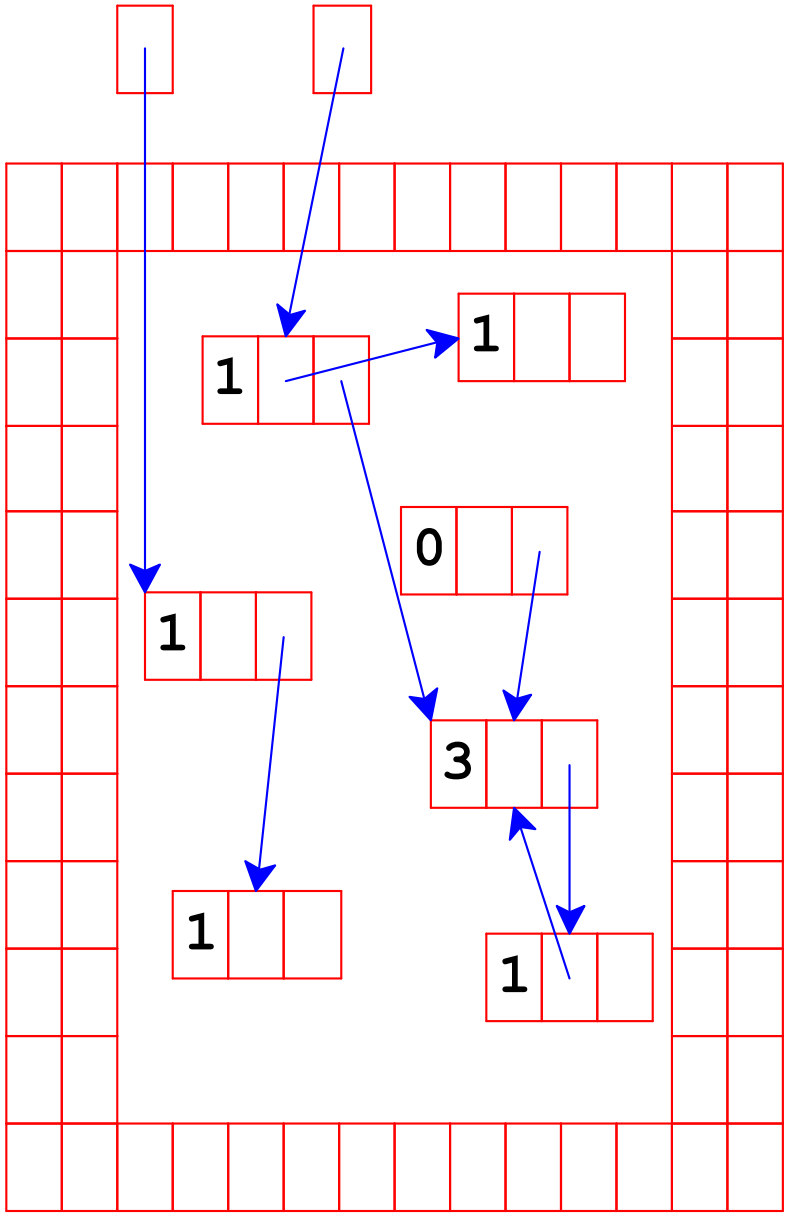
Reference Counting



Top boxes are variables

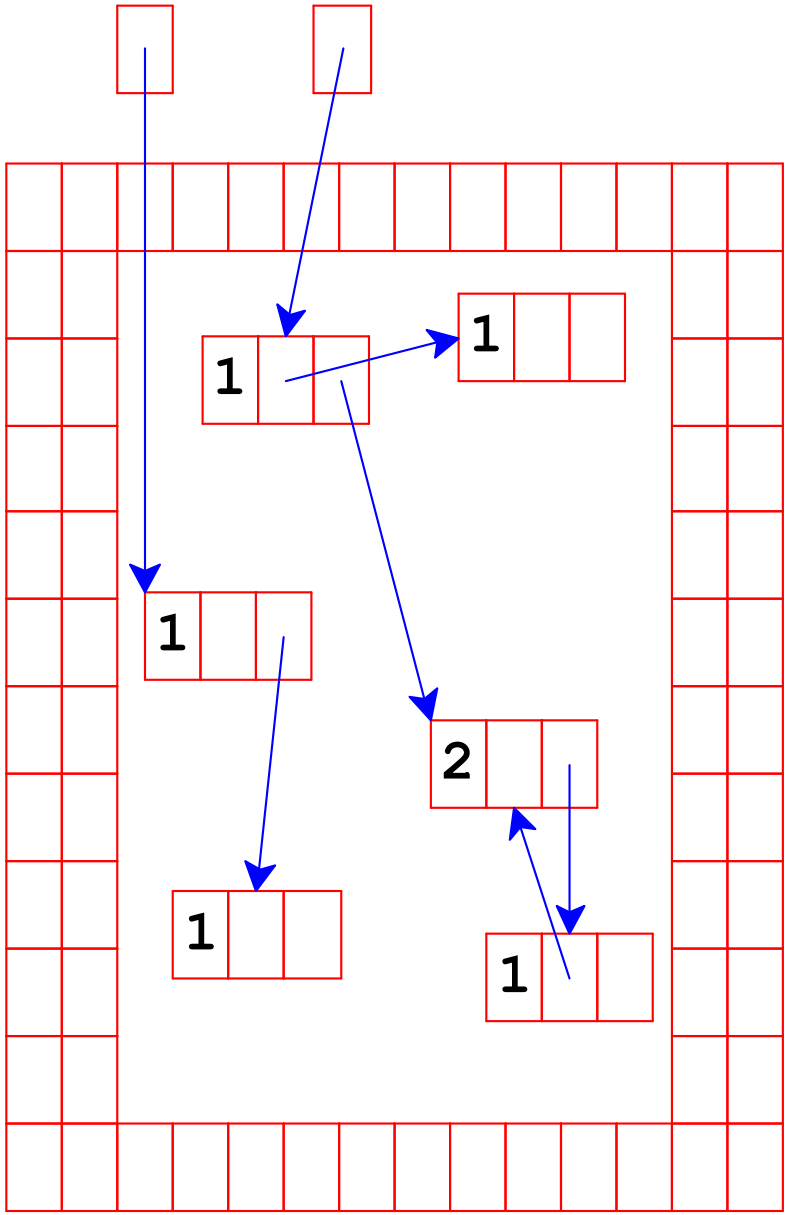
Other boxes are allocated with **`allocate`**

Reference Counting



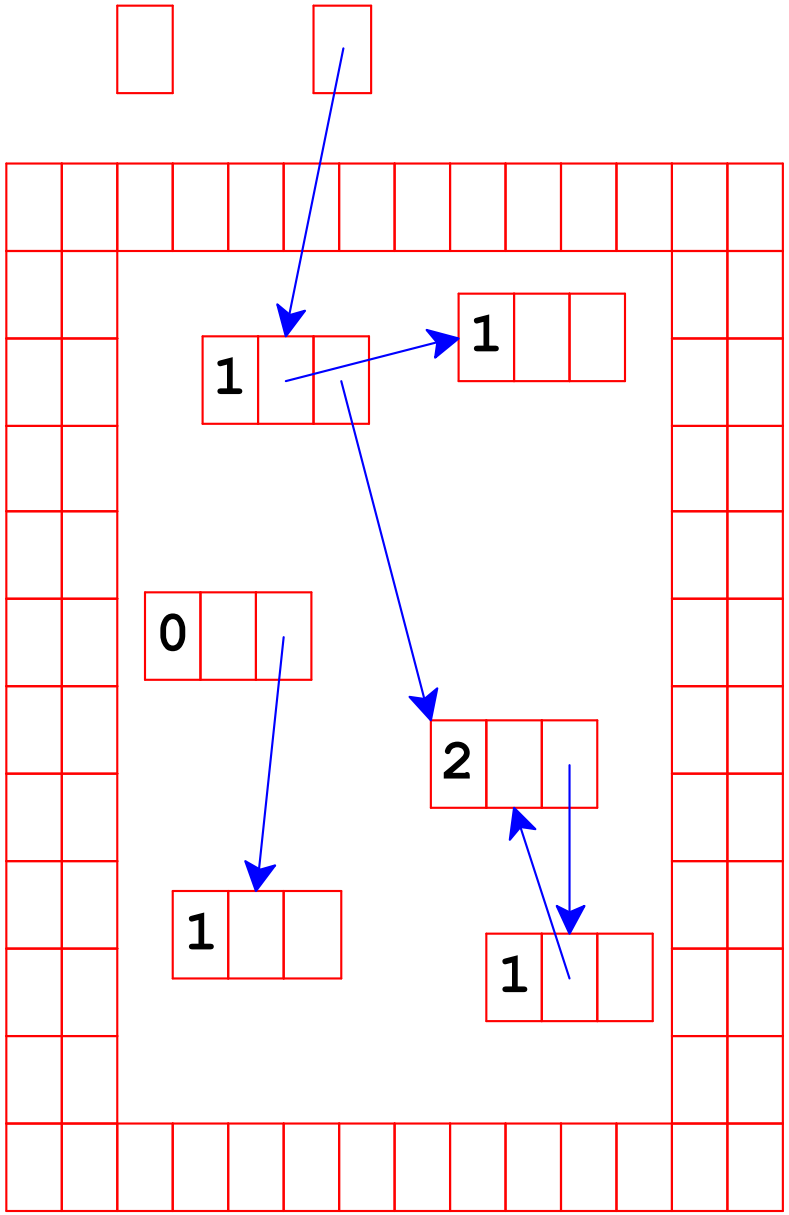
Adjust counts when a pointer is changed...

Reference Counting



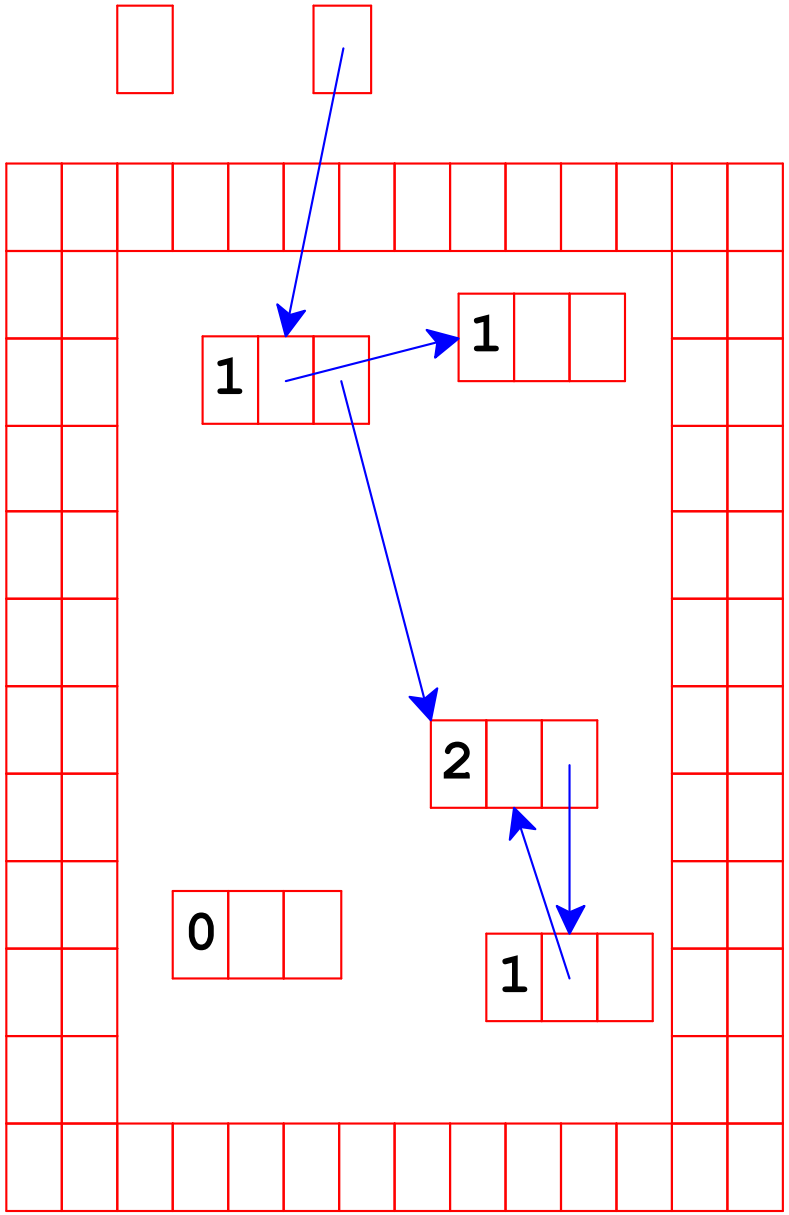
... freeing an object if its count goes to 0

Reference Counting



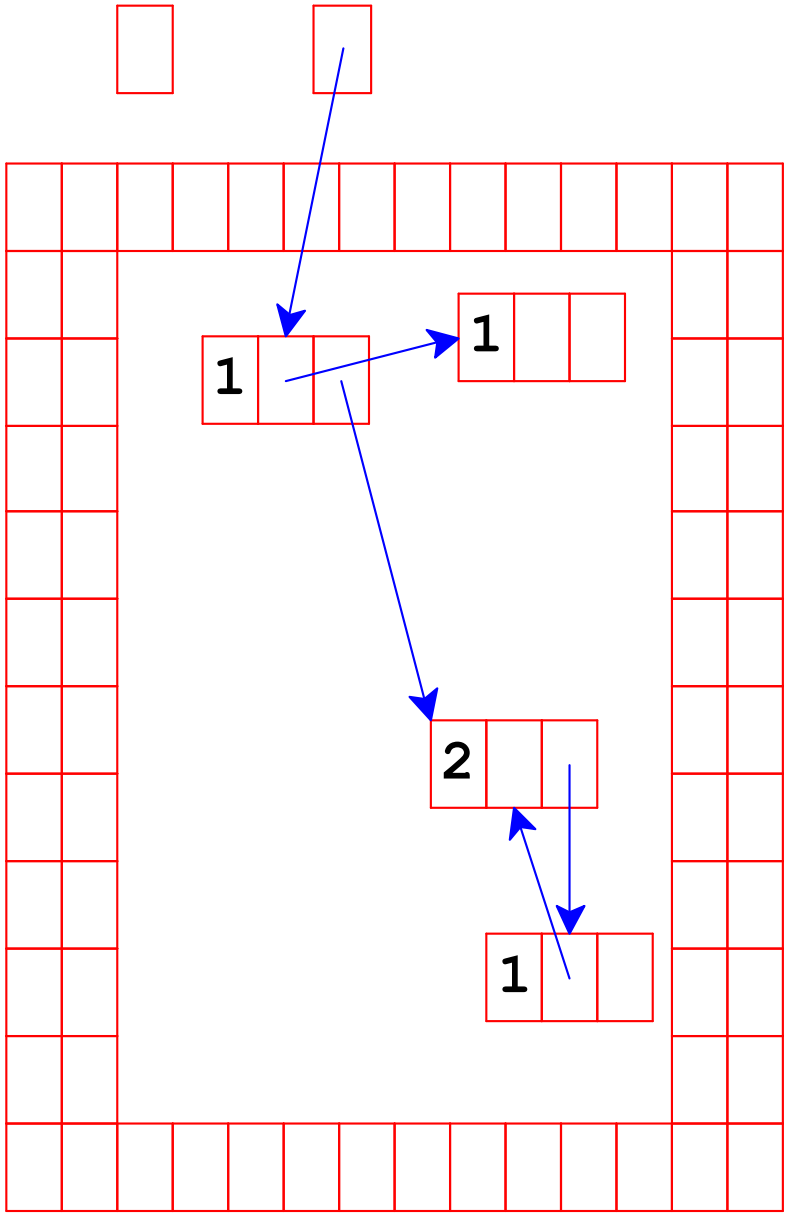
Same if the pointer is in a variable

Reference Counting



Adjust counts after frees, too...

Reference Counting



... which can trigger more frees

Reference Counting Implementation

Allocator needs room for a count

refcount.c

```
struct rc_node {
  int count;
  struct node n;
};

#define NODE_TO_RC(p) ((struct rc_node *)((char *)(p) \
                       - offsetof(struct rc_node, n)))
```

Reference Counting Implementation

refcount.c

```
struct node *allocate() {
  struct rc_node *rn;
  rn = raw_malloc(sizeof(struct rc_node));
  rn->count = 0;
  rn->n.left = NULL;
  rn->n.right = NULL;
  return &rn->n;
}
```

Reference Counting Implementation

refcount/allocate.h

```
void refcount_inc(struct node *p);
void refcount_dec(struct node *p);
void refcount_dec_no_free(struct node *p);

#define SET_NODE(field, val) \
  (refcount_dec(field), refcount_inc(field = val))

#define UNSET_NODE(var) \
  refcount_dec(var)

#define UNSET_RETURN_NODE(var) \
  refcount_dec_no_free(var)
```

Reference Counting Implementation

refcount.c

```
void refcount_inc(struct node *p) {
  if (p)
    NODE_TO_RC(p)->count++;
}
```

Reference Counting Implementation

refcount.c

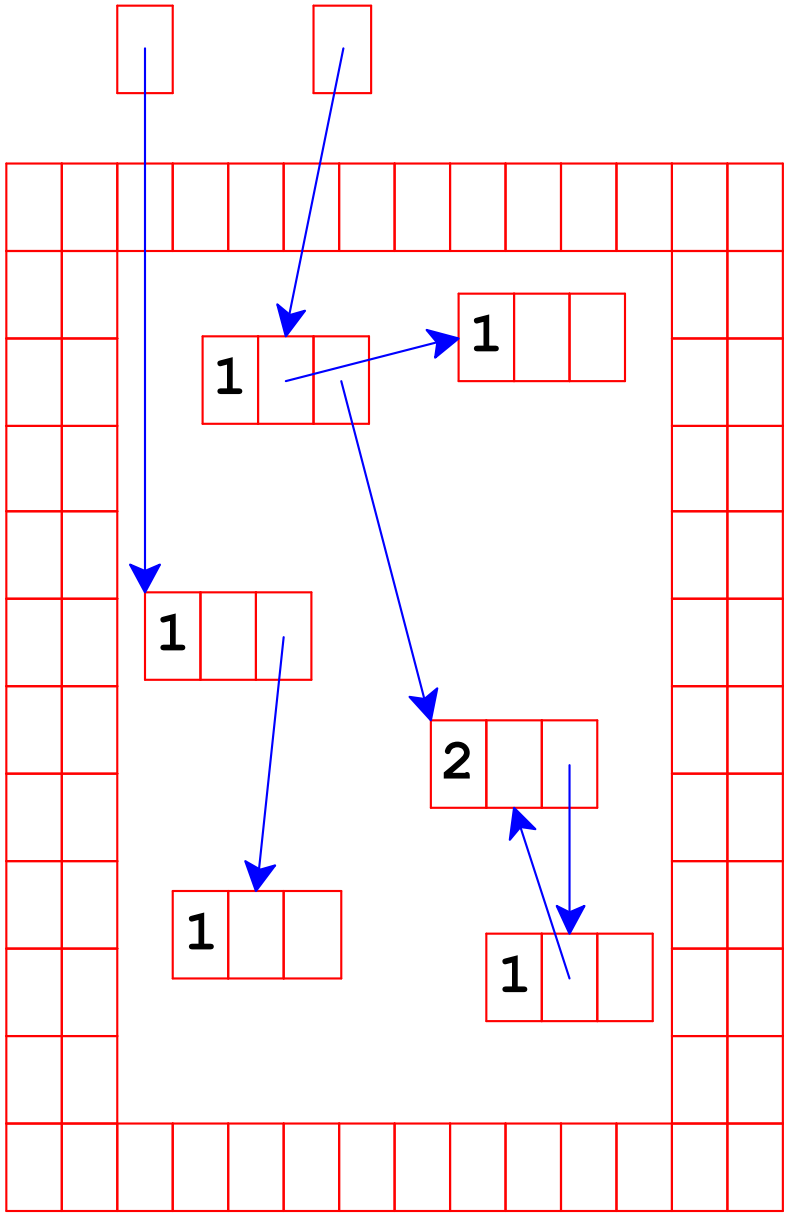
```
void refcount_dec(struct node *p) {
  if (p) {
    struct rc_node *rn = NODE_TO_RC(p);
    if (--rn->count == 0) {
      refcount_dec(rn->n.left);
      refcount_dec(rn->n.right);
      raw_free(rn, sizeof(struct rc_node));
    }
  }
}
```

Reference Counting Implementation

refcount.c

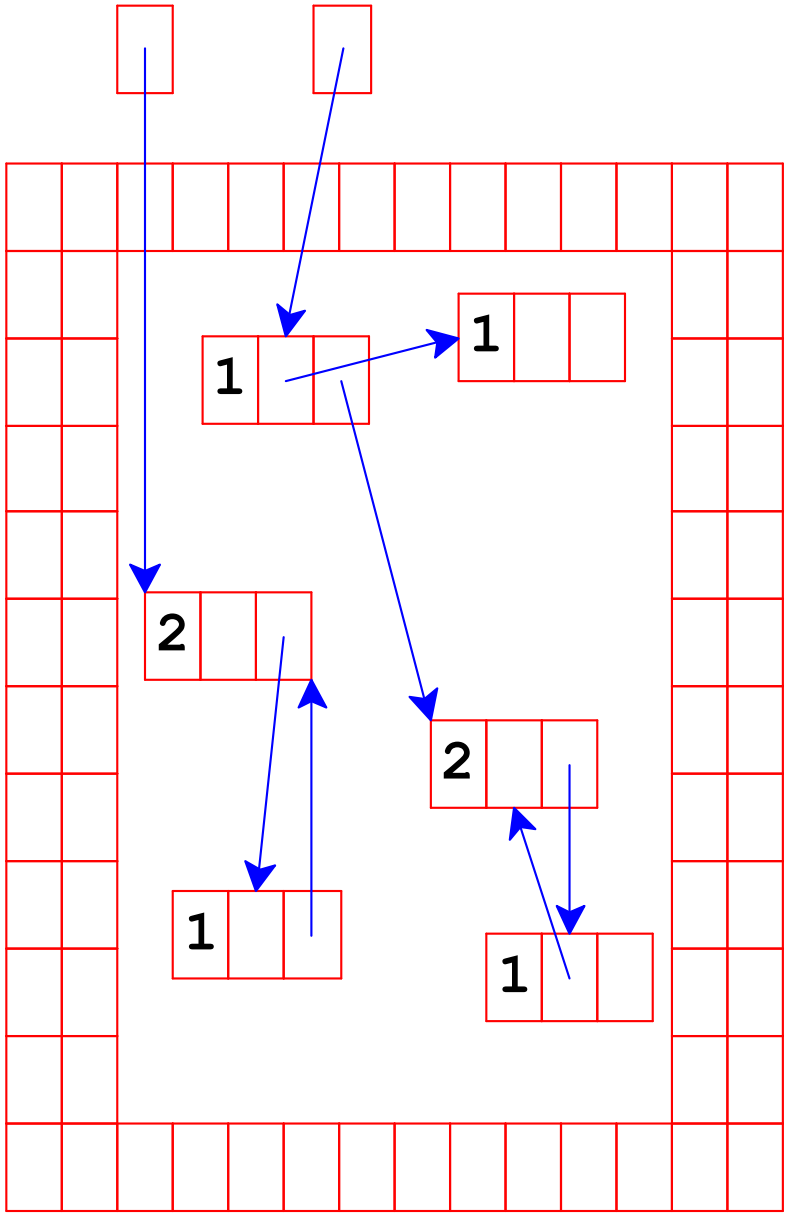
```
void refcount_dec_no_free(struct node *p) {
  if (p)
    --NODE_TO_RC(p)->count;
}
```

Reference Counting And Cycles



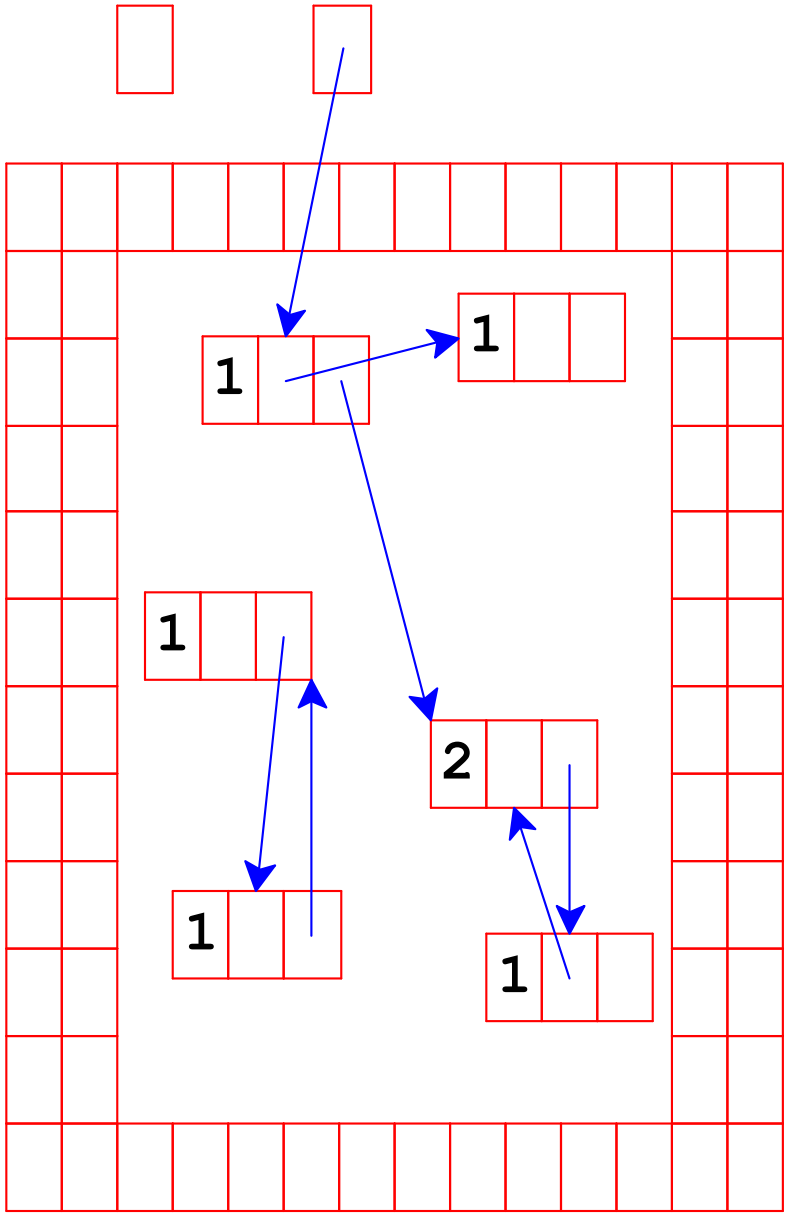
An assignment can create a cycle...

Reference Counting And Cycles



Adding a reference increments a count

Reference Counting And Cycles



Lower-left objects are inaccessible, but not deallocated

Cycles break reference counting

Garbage Collection

Garbage collection:

a way to know whether an object is *accessible*

- An object referenced by a variable is ***live***
- An object referenced by a live object is also live
- Assume that program can only possibly use live objects

no reliable way to get to other objects?

Not allowed:

```
uintptr_t v = (uintptr_t)ptr1 ^ (uintptr_t)ptr2;
....
ptr2 = (void *)(v ^ (uintptr_t)ptr1);
```

Garbage Collection

Garbage collection:

a way to know whether an object is *accessible*

- An object referenced by a variable is ***live***
- An object referenced by a live object is also live
- Assume that program can only possibly use live objects

  no reliable way to get to other objects?

- A garbage collector frees all objects that are not live
- Allocate until we run out of memory, then run a garbage collector to get more space

Garbage Collection

Garbage collection:

a way to know whether an object is *accessible*

```
struct node *make_tree(int depth) {
  if (depth == 0)
    return NULL;
  else {
    /* sumulate useful work: */
    make_tree(depth-1);

    struct node *t = allocate();
    t->left = make_tree(depth-1);
    t->right = make_tree(depth-1);

    return t;
  }
}
```

original

```
struct node *make_tree(int depth) {
    if (depth == 0)
      return NULL;
    else {
      /* sumulate useful work: */
      make_tree(depth-1);

      struct node *t = NULL;
      PUSH_STACK_POINTER(t);

      t = allocate();
      t->left = make_tree(depth-1);
      t->right = make_tree(depth-1);

      POP_STACK_POINTER(t);

      return t;
    }
  }
```

converted

Garbage Collection

Garbage collection:

a way to know whether an object is *accessible*

```
struct node *make_tree(int depth) {
  if (depth == 0)
    return NULL;
  else {
    /* sumulate useful work: */
    make_tree(depth-1);

    struct node *t = allocate();
    struct node *l = make_tree(depth-
    t->left = l;
    t->right = make_tree(depth-1);

    return t;
  }
}
```

```
struct node *make_tree(int depth) {
    if (depth == 0)
      return NULL;
    else {
      /* sumulate useful work: */
      make_tree(depth-1);

      struct node *t = NULL;
      PUSH_STACK_POINTER(t);

      t = allocate();
      struct node *l = NULL;
      PUSH_STACK_POINTER(l);
      l = make_tree(depth-1);
      t->left = l;
      t->right = make_tree(depth-1);

      POP_STACK_POINTER(l);
      POP_STACK_POINTER(t);
```

Garbage Collection

Garbage collection:

a way to know whether an object is *accessible*

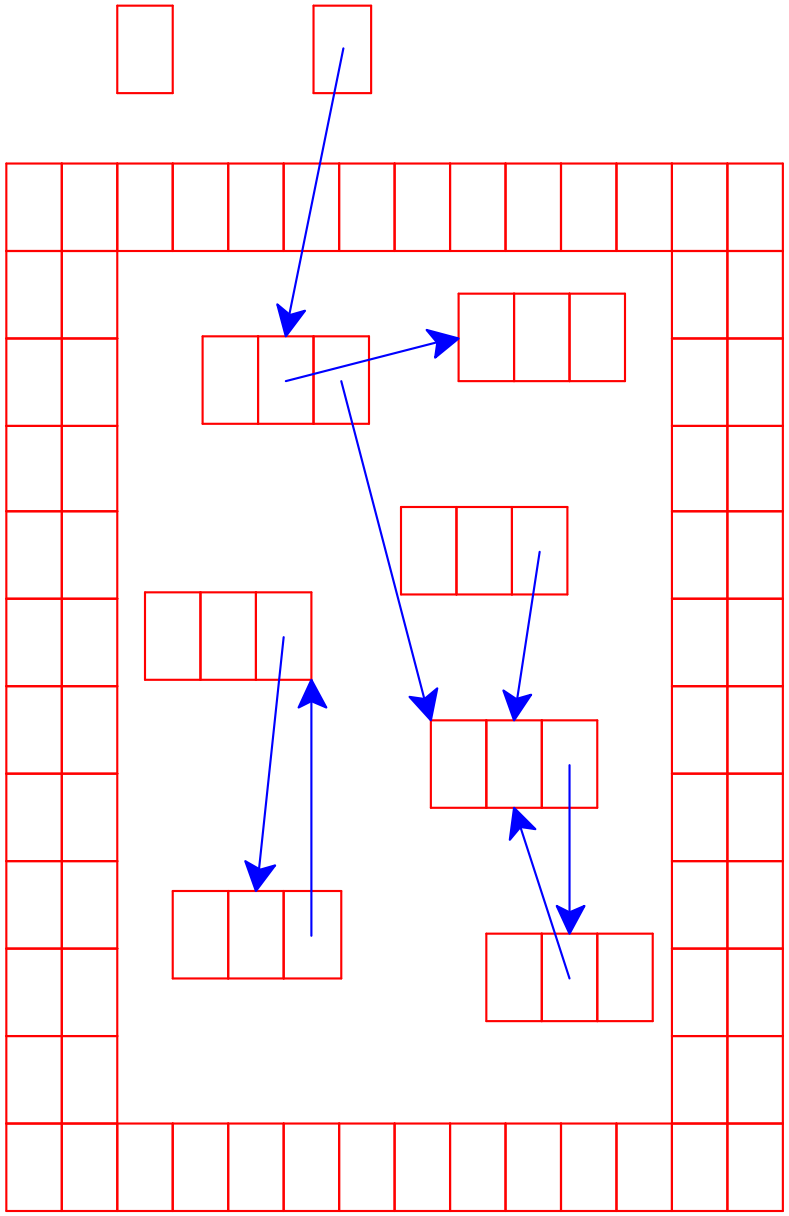
```
struct node *make_tree(int depth) {
  if (depth == 0)
    return NULL;
  else {
    /* sumulate useful work: */
    make_tree(depth-1);

    struct node *t = allocate();
    struct node *l = make_tree(depth-
    t->left = l;
    t->right = make_tree(depth-1);

    return t;
  }
}
```

```
struct node *make_tree(int depth) {
    if (depth == 0)
      return NULL;
    else {
      /* sumulate useful work: */
      make_tree(depth-1);

      struct node *t = NULL;
      PUSH_STACK_POINTER(t);

      t = allocate();
      struct node *l = NULL;
      PUSH_STACK_POINTER(l);
      l = make_tree(depth-1);
      t->left = l;
      l = NULL;
      t->right = make_tree(depth-1);

      POP_STACK_POINTER(l);
      POP_STACK_POINTER(t);
```

Garbage Collection Algorithm

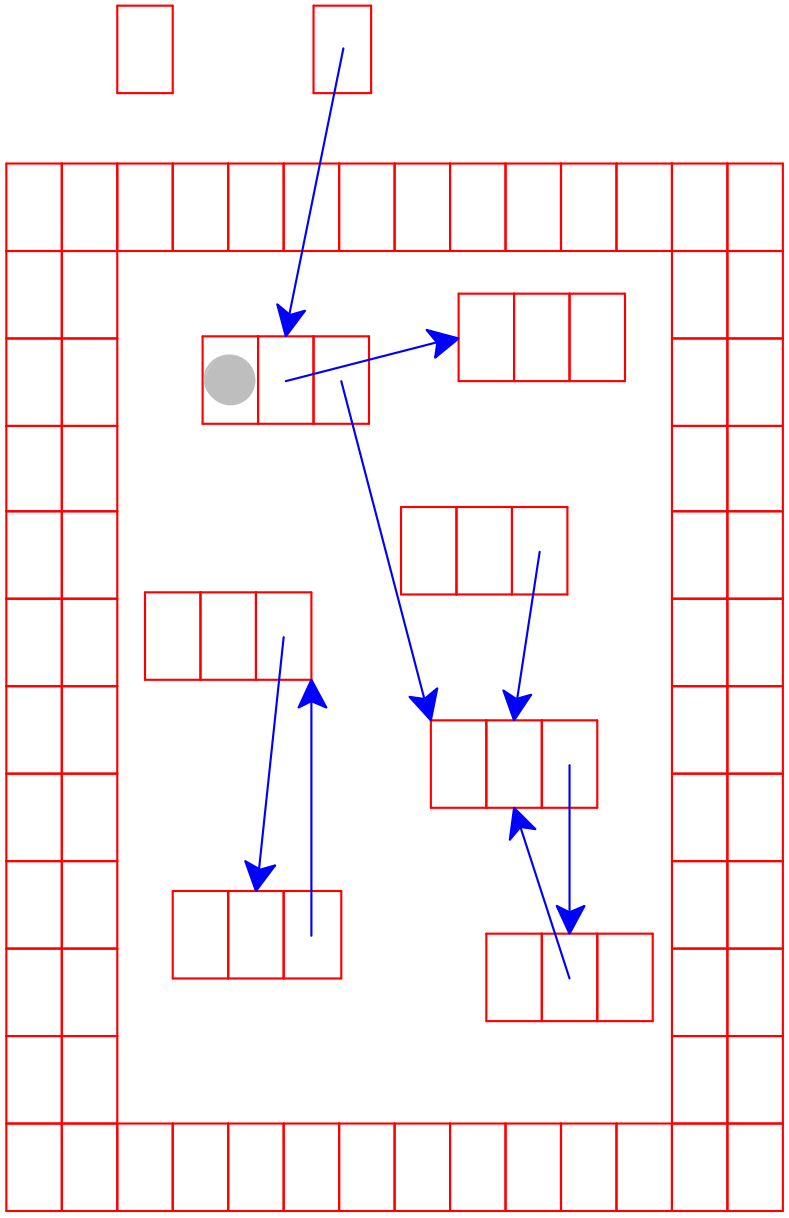
- Color all objects ***white***
- Color objects referenced by variables ***gray***
- Repeat until there are no gray objects:
  - Pick a gray object, *r*
  - For each white object that *r* points to, make it gray
  - Color *r* ***black***
- Deallocate all white objects

Garbage Collection



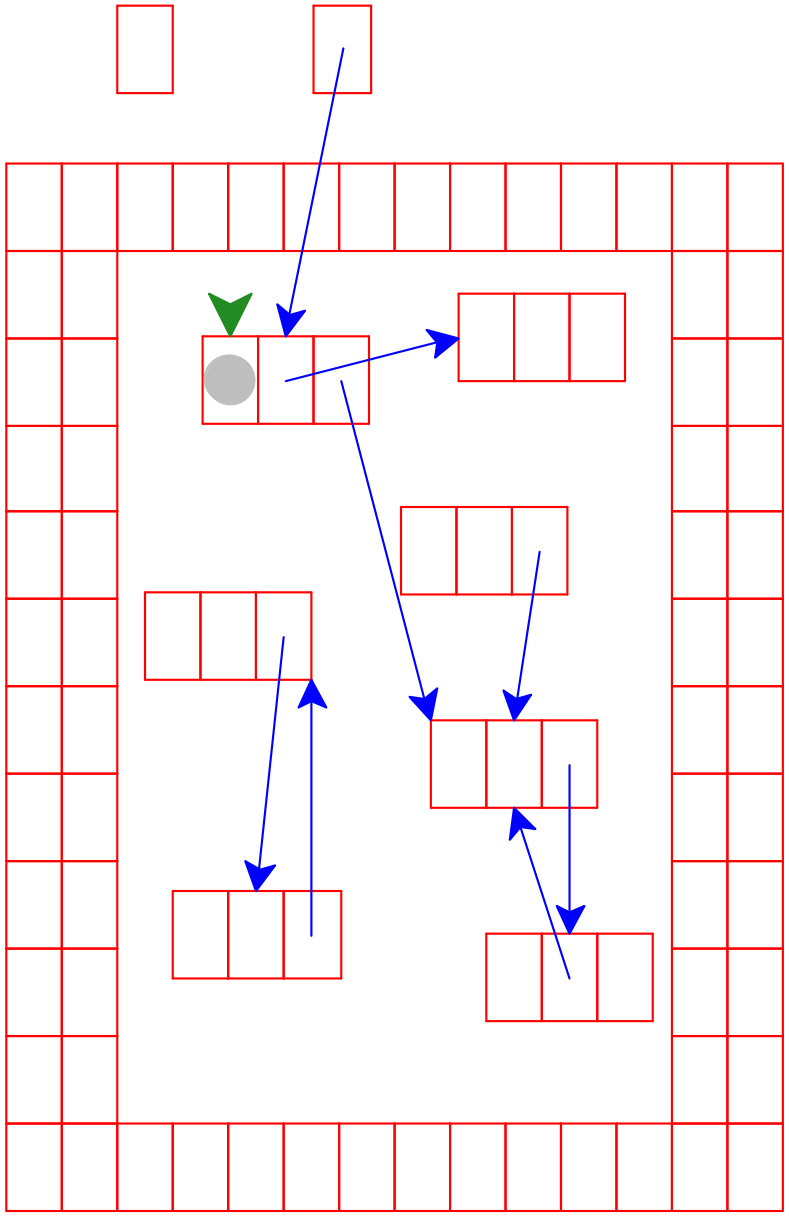
All objects are marked white

Garbage Collection



Mark objects referenced by variables as gray

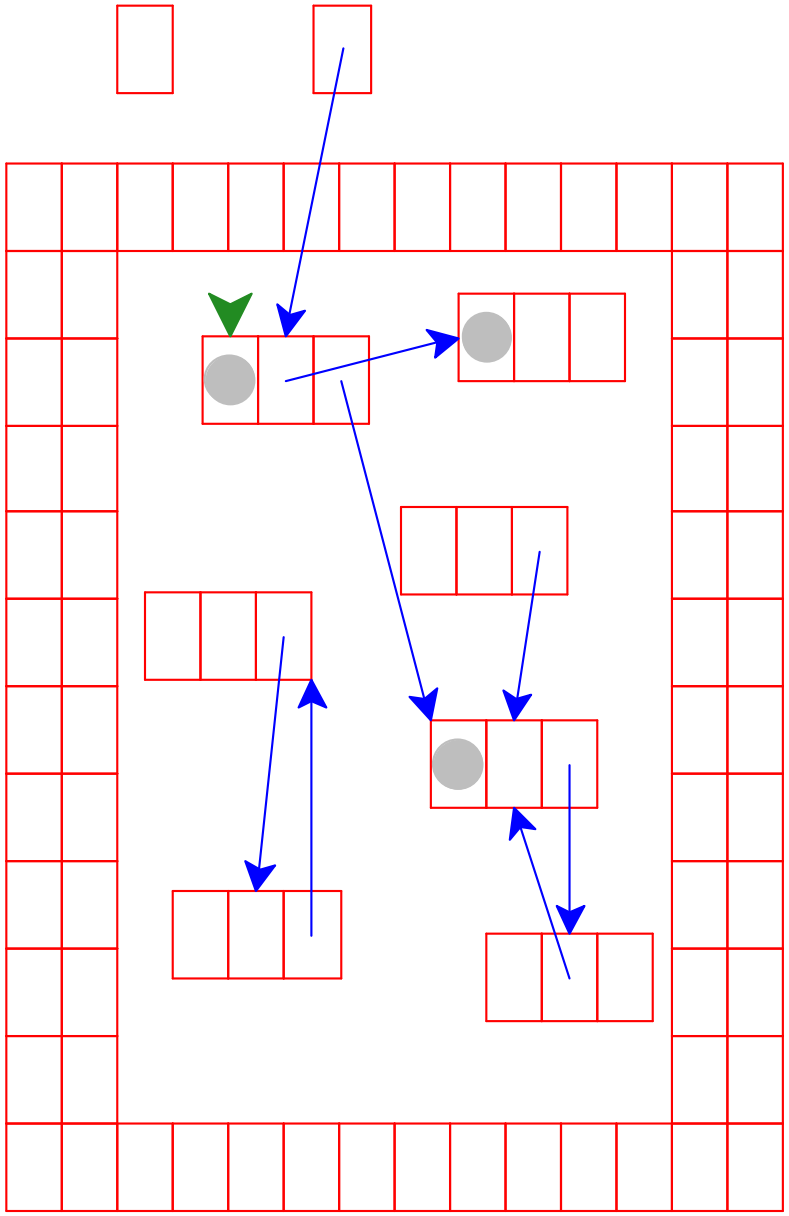
Garbage Collection



Need to pick a gray object

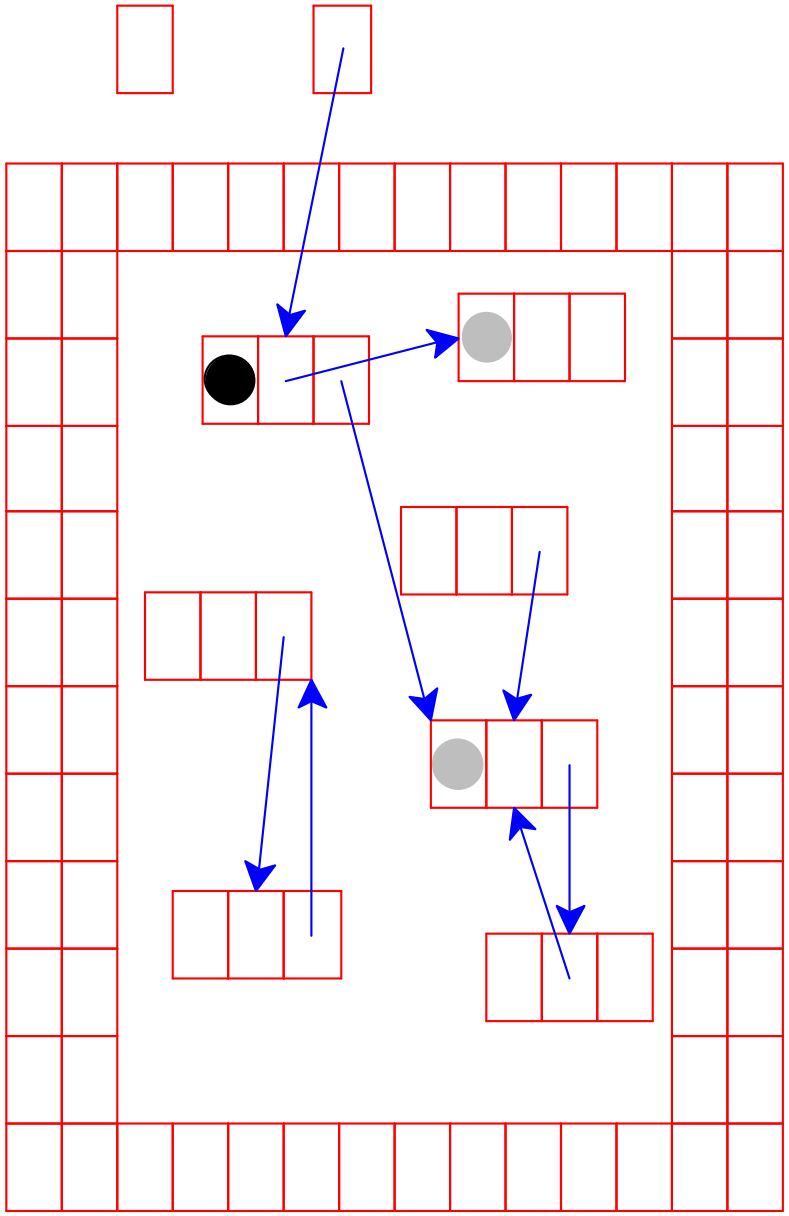
Green arrow indicates the chosen object

Garbage Collection



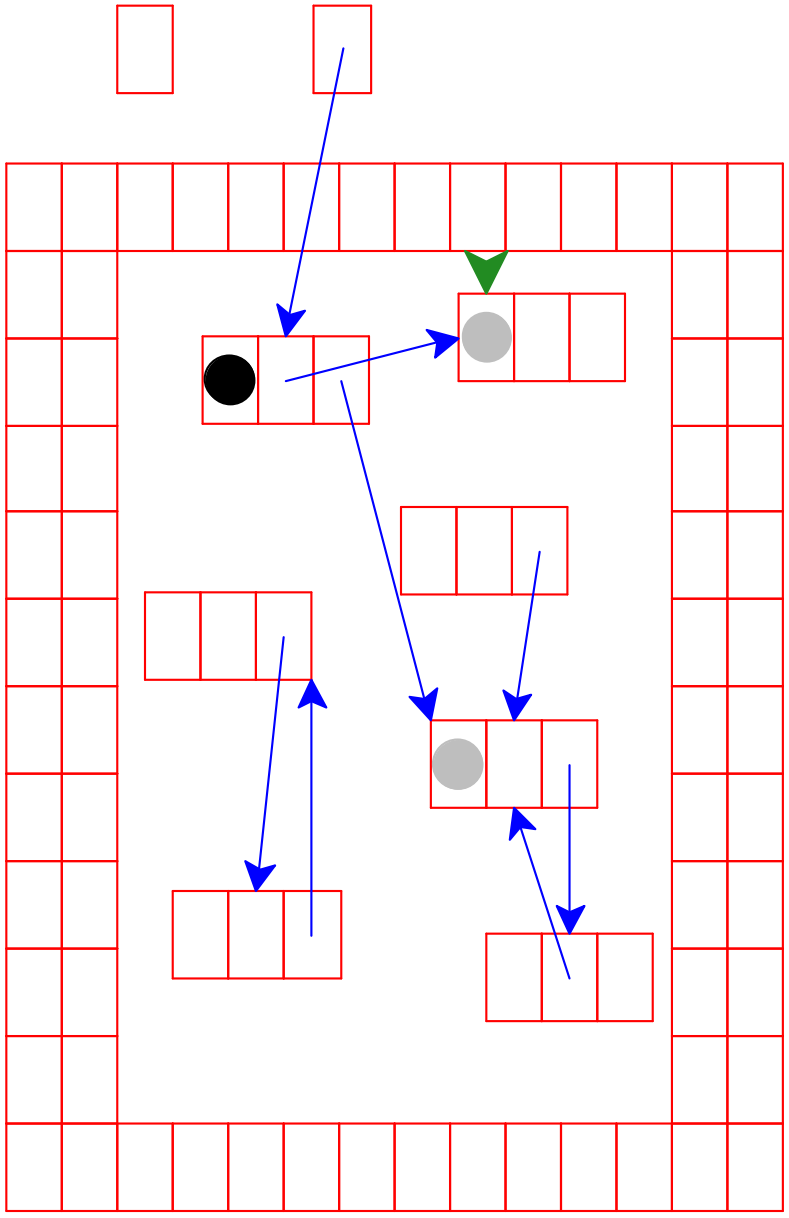
Mark white objects referenced by chosen object as gray

Garbage Collection



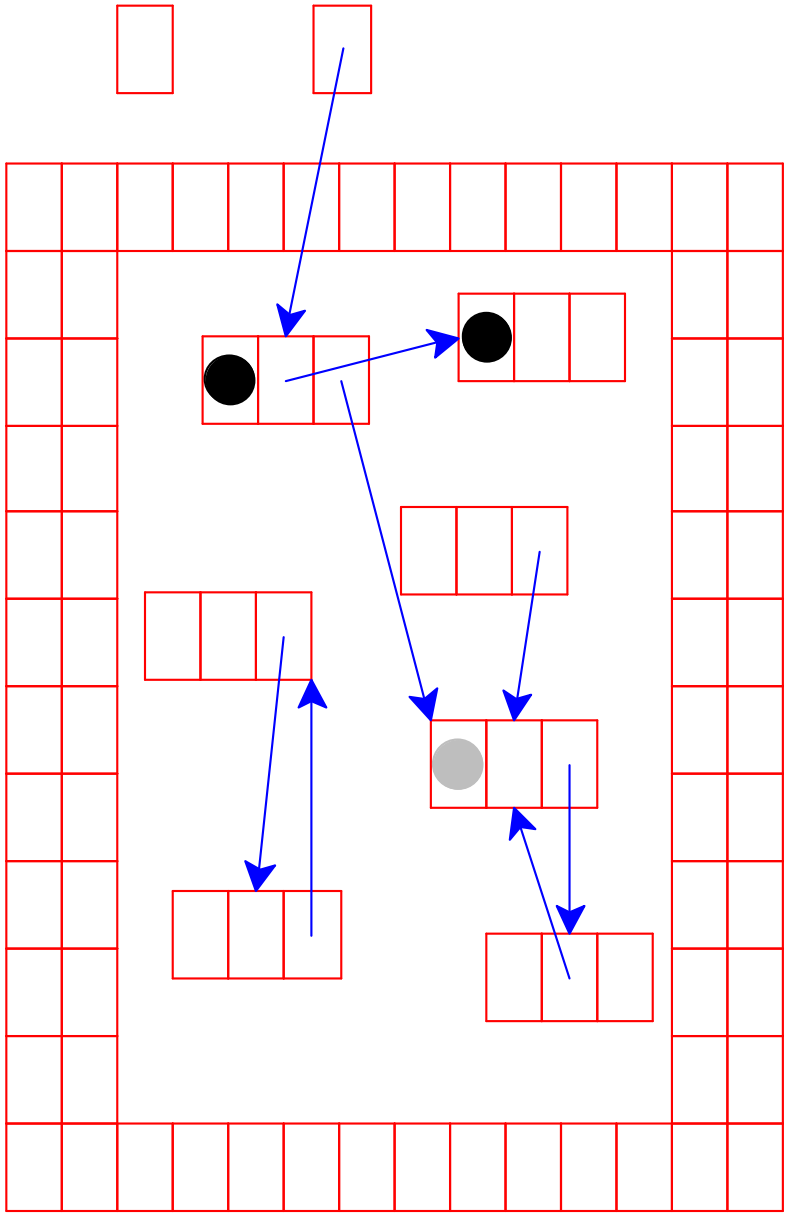
Mark chosen object black

Garbage Collection



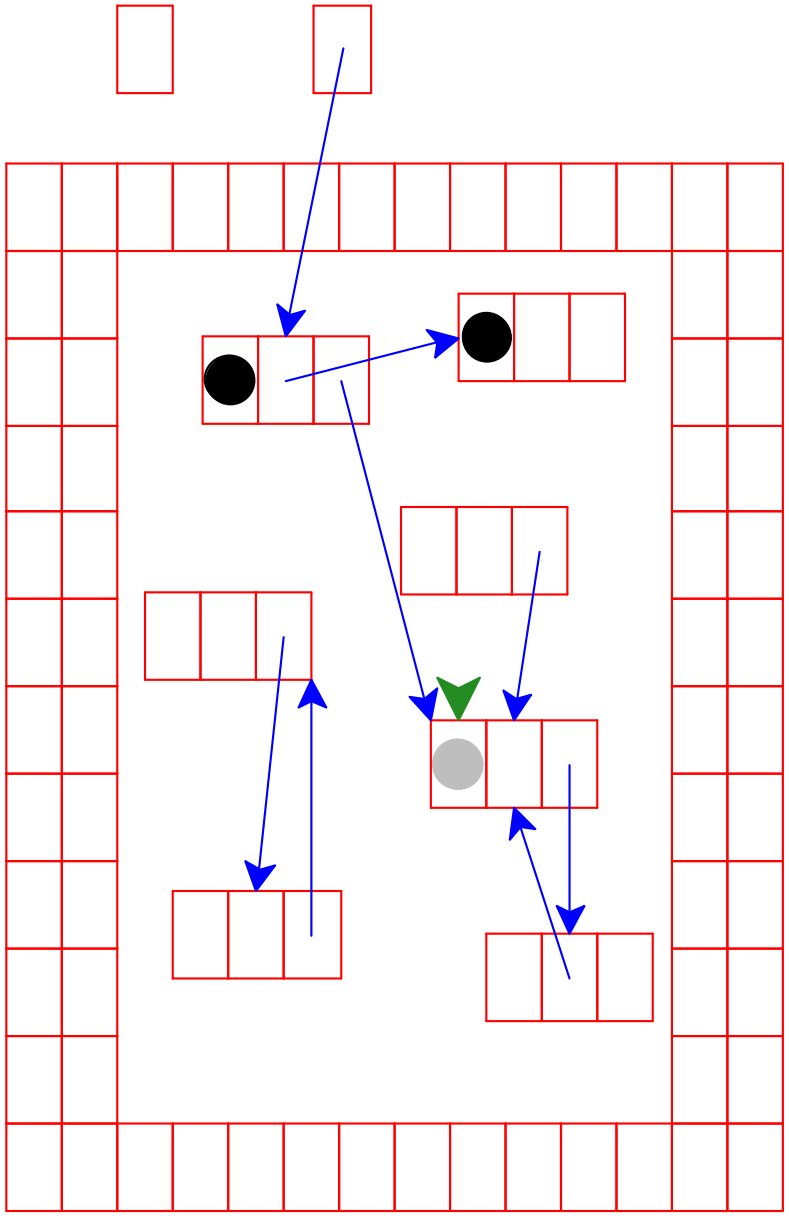
Start again: pick a gray object

Garbage Collection



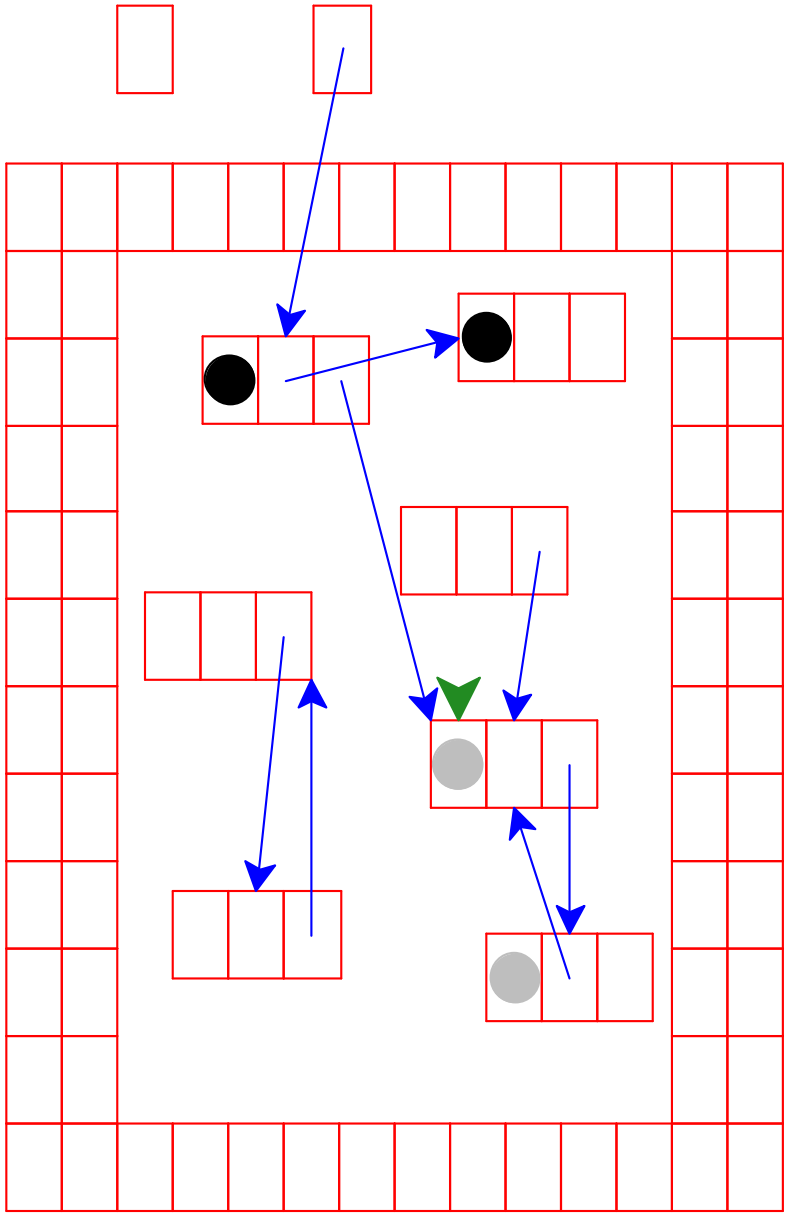
No referenced objects; mark black

Garbage Collection



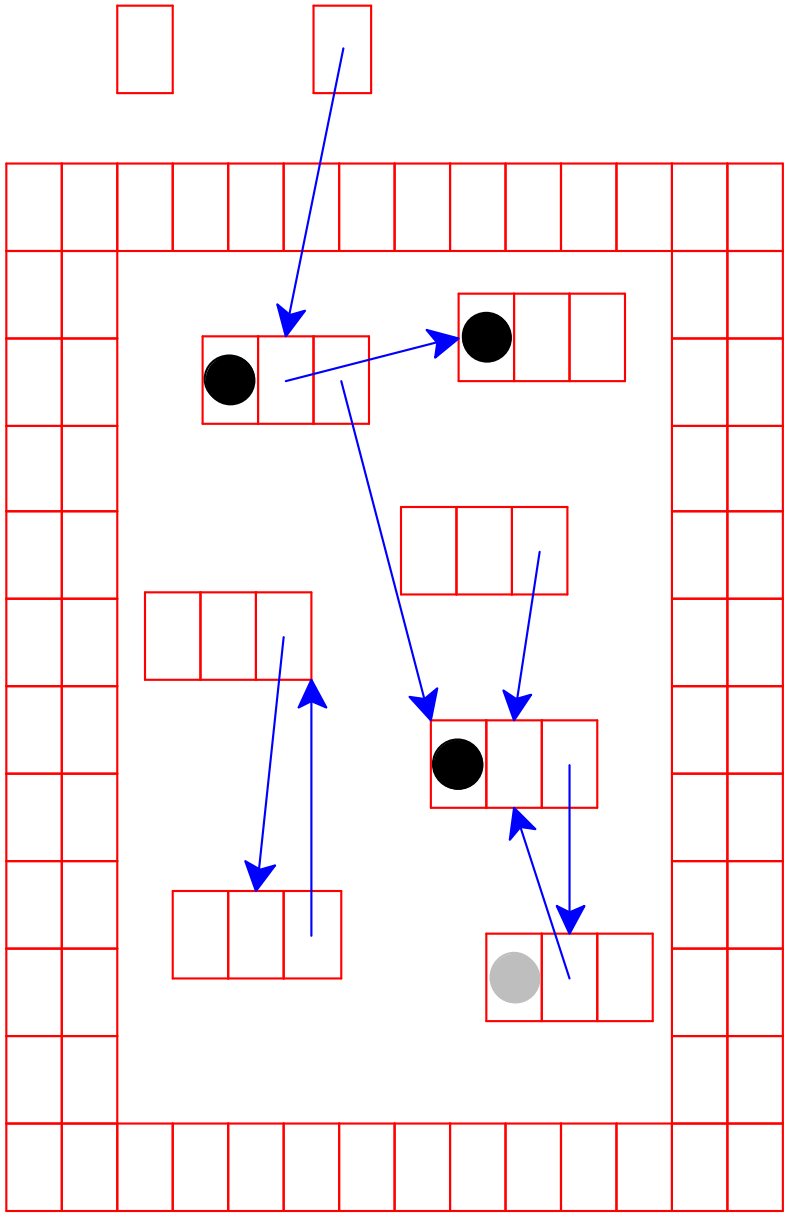
Start again: pick a gray object

Garbage Collection



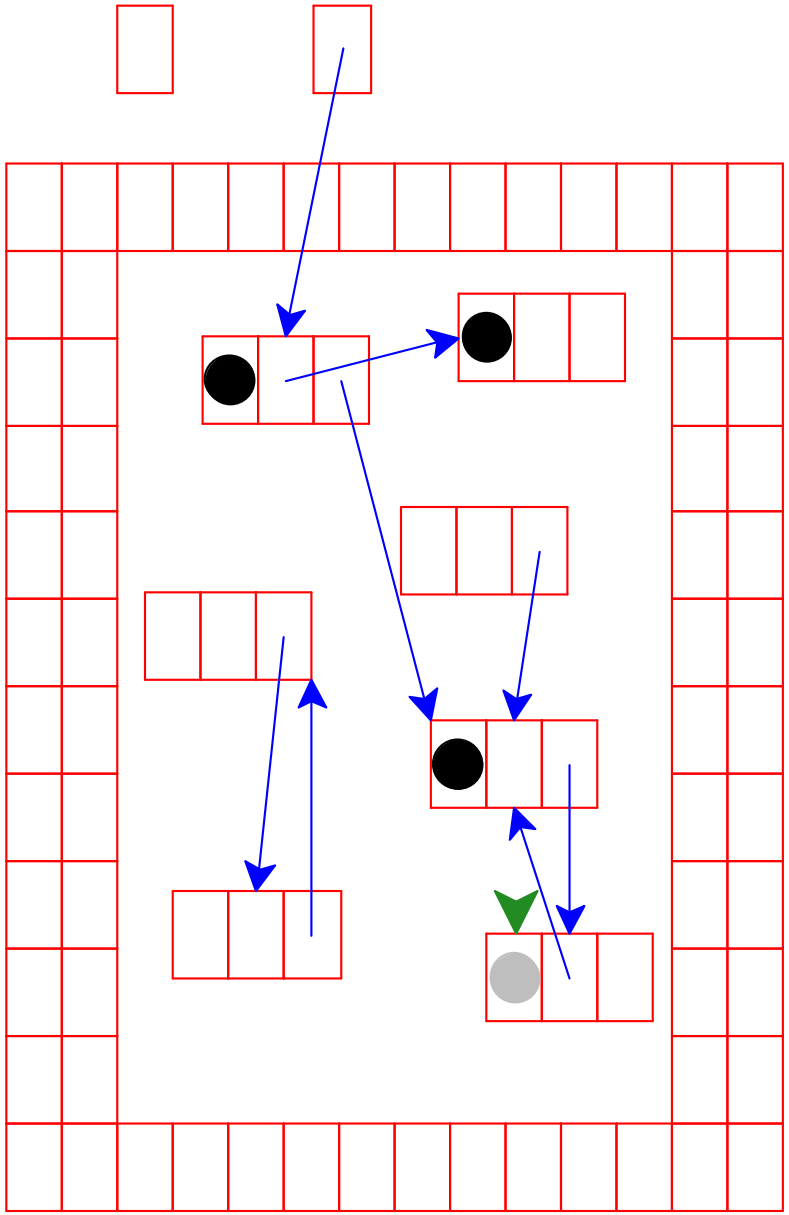
Mark white objects referenced by chosen object as gray

Garbage Collection



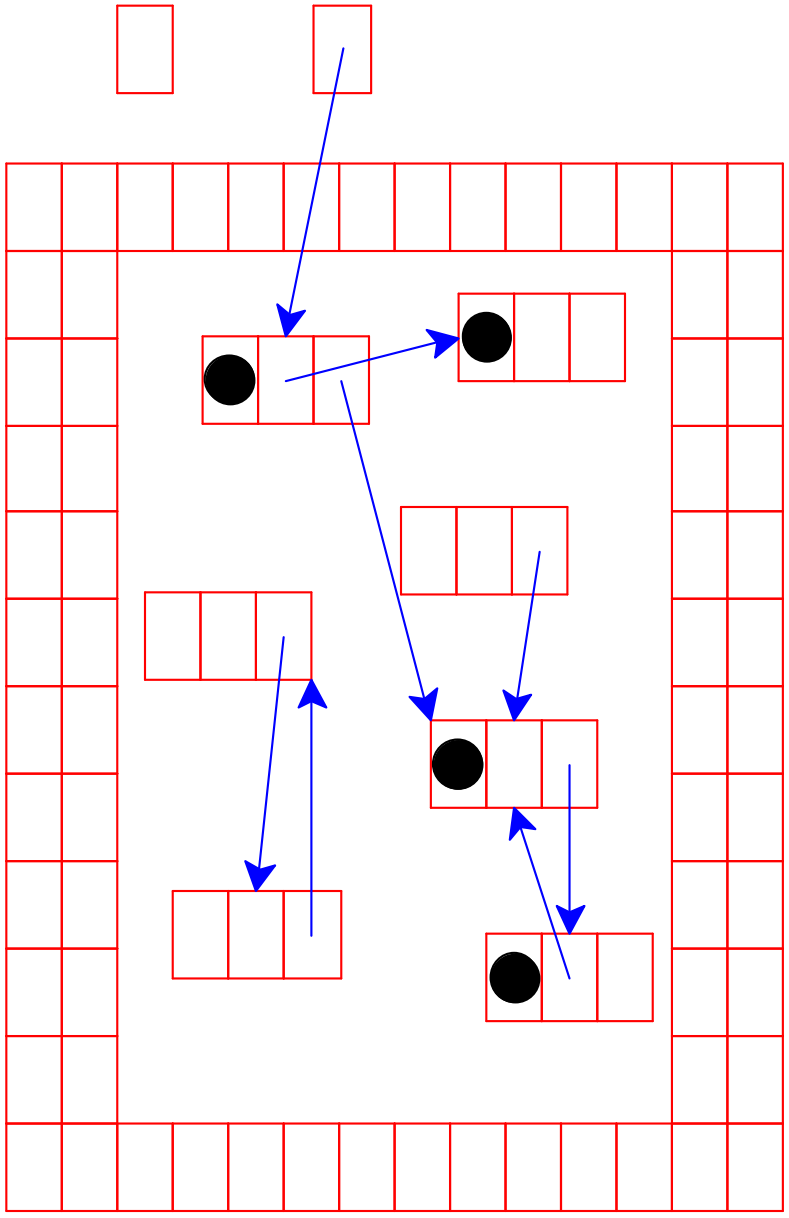
Mark chosen object black

Garbage Collection



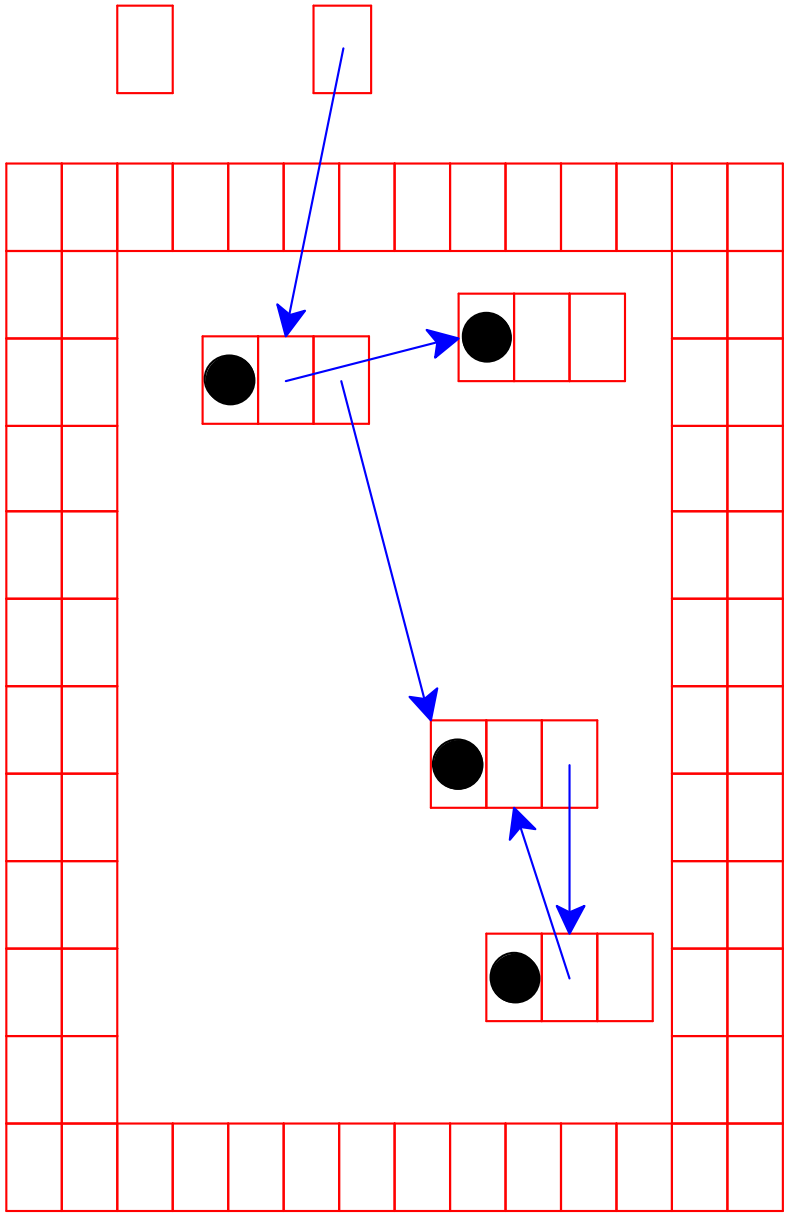
Start again: pick a gray object

Garbage Collection



No referenced white objects; mark black

Garbage Collection



No more gray objects; deallocate white objects

Cycles don't break garbage collection

Garbage Collector Implementation

Collector needs room for mark bits and free list

gc.c

```
struct gc_node {
  union {
    int marked;
    struct gc_node *next_free;
  };
  struct node n;
};

#define NODE_TO_GC(p) ((struct gc_node *)((char *)(p) \
                       - offsetof(struct gc_node, n)))
```

Garbage Collector Implementation

gc.c

```
static void get_more_memory();
struct gc_node *free_list = NULL;

struct node *allocate() {
  struct gc_node *nh;

  if (!free_list) {
    collect_garbage();
    if (!free_list)
      get_more_memory();
  }

  nh = free_list;
  free_list = free_list->next_free;

  return &nh->n;
}
```

Garbage Collector Implementation

gc.c

```
static void get_more_memory()
{
  struct gc_chunk *c = raw_malloc(sizeof(struct gc_chunk));
  ....
  c->mem = raw_malloc(NODES_PER_CHUNK * sizeof(struct gc_node));

  /* Add everything in the new chunk to the free list: */
  for (i = 0; i < NODES_PER_CHUNK; i++) {
    struct gc_node *nh = c->mem + i;
    nh->next_free = free_list;
    free_list = nh;
  }

  ....
}
```

Garbage Collector Implementation

To register stack variables:

allocate.h

```
extern struct node **root_addrs[];
extern int num_roots;

#define PUSH_STACK_POINTER(var) root_addrs[num_roots++] = &var;
#define POP_STACK_POINTER(var)  --num_roots
```

Garbage Collector Implementation

Collect = mark + sweep

gc.c

```
void collect_garbage() {
  reset_all_marks();

  /* mark (i.e., find all referenced) */
  for (int i = 0; i < num_roots; i++)
    mark(*(root_addrs[i]));

  /* sweep (i.e., add unmarked to the free list) */
  sweep();
}
```

Garbage Collector Implementation

Recursive mark strategy — represent "gray" via C continuation

gc.c

```
void mark(struct node *n) {
  if (n != NULL) {
    struct gc_node *nh;
    nh = NODE_TO_GC(n);

    if (!nh->marked) {
      nh->marked = 1;
      mark(nh->n.left);
      mark(nh->n.right);
    }
  }
}
```

Better: use custom stack to avoid overflowing C stack

Garbage Collector Implementation

Sweeping adds all "white" nodes to free list

gc.c

```
void sweep() {
  struct gc_chunk *c;
  int i;

  free_list = NULL;
  for (c = chunks; c; c = c->next) {
    for (i = 0; i < NODES_PER_CHUNK; i++) {
      if (!c->mem[i].marked) {
        c->mem[i].next_free = free_list;
        free_list = &c->mem[i];
      }
    }
  }
}
```

Conservative GC

Conservative garbage collection works without cooperation

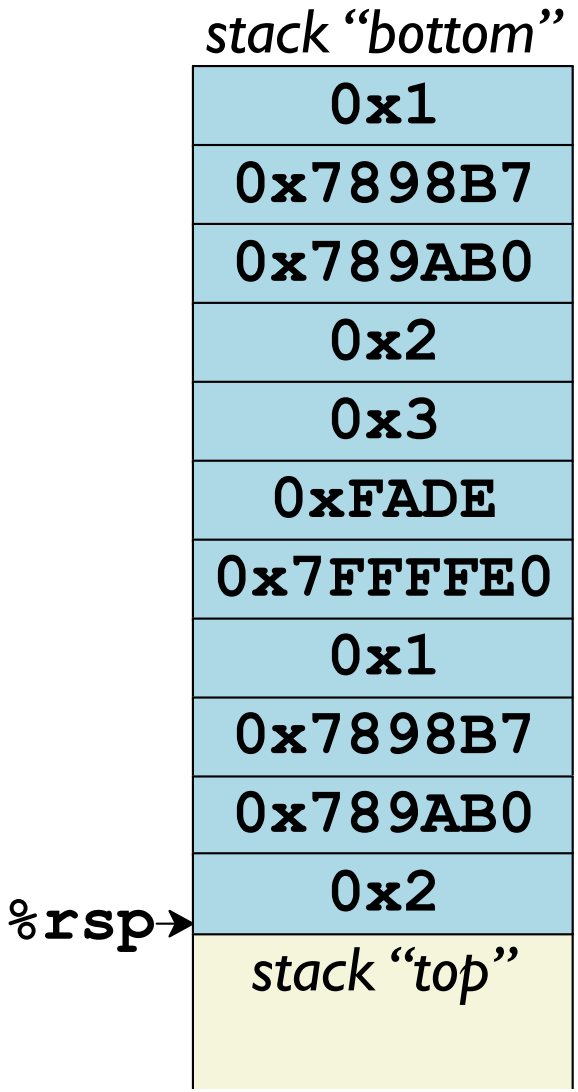
```
struct node *make_tree(int depth) {
  if (depth == 0)
    return NULL;
  else {
    /* sumulate useful work: */
    make_tree(depth-1);

    struct node *t = allocate();
    t->left = make_tree(depth-1);
    t->right = make_tree(depth-1);

    return t;
  }
}
```

depends on and assumes details of the machine, OS, and compiler

Conservative GC

Node allocation range:
`0x789800` to **`0x78A000`**

```
void collect_garbage() {
   .... look at the stack ....
}
```

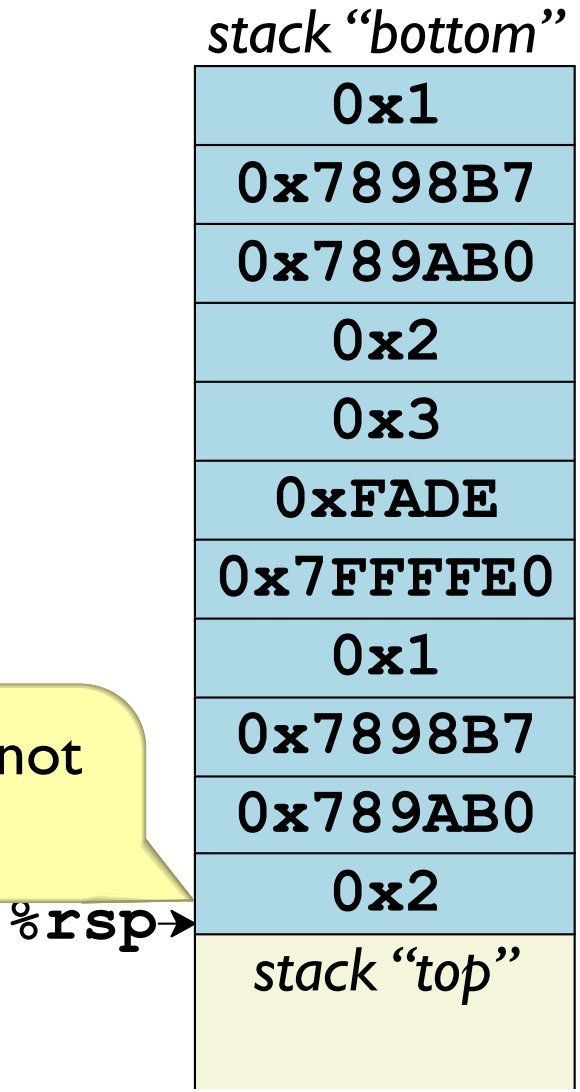


Conservative GC

Node allocation range:
`0x789800` to **`0x78A000`**

```
void collect_garbage() {
   .... look at the stack ....
}
```

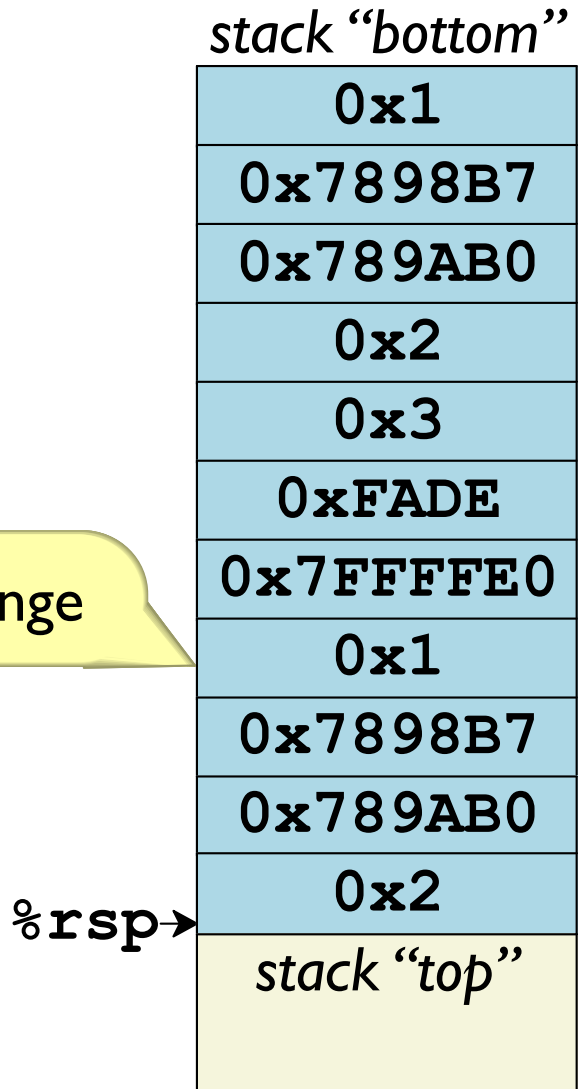
Does not look like a pointer — not in allocation range



Conservative GC

Node allocation range:
`0x789800` to **`0x78A000`**

```
void collect_garbage() {
   .... look at the stack
}
```

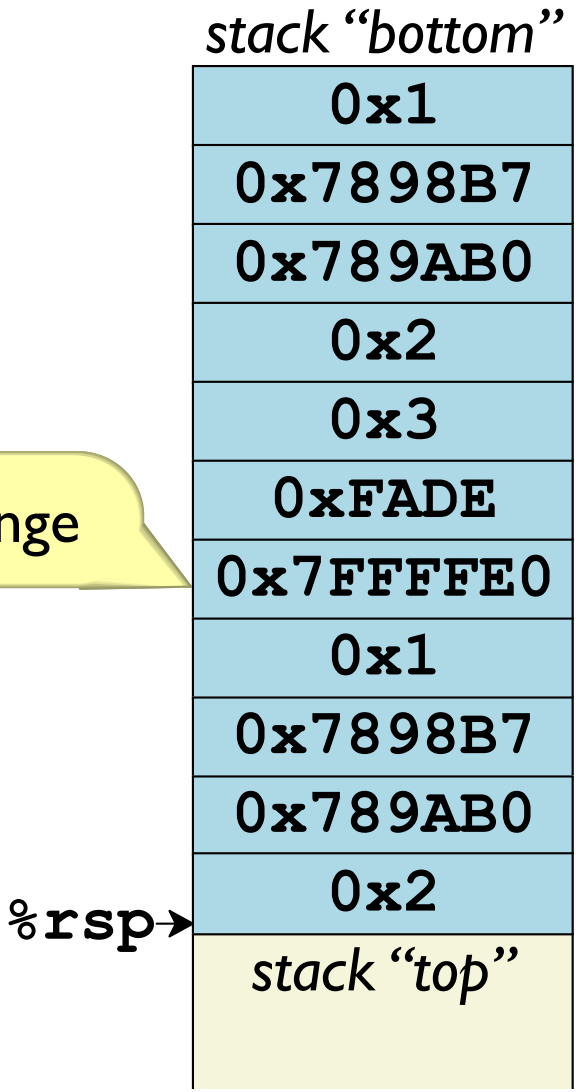


Not in allocation range

Conservative GC

Node allocation range:
`0x789800` to **`0x78A000`**

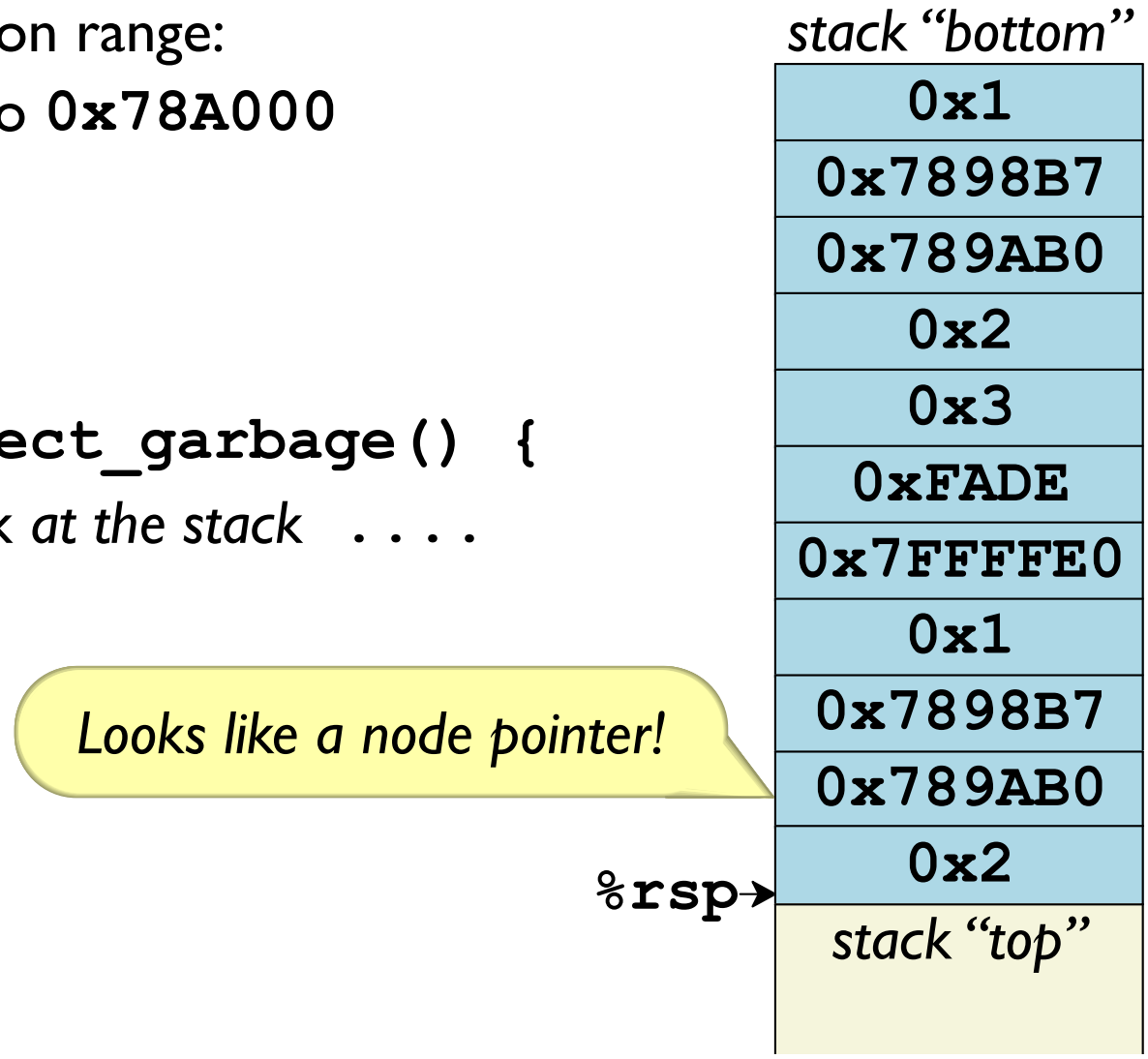
```
void collect_garbage() {
   .... look
}
```



Conservative GC

Node allocation range:
`0x789800` to **`0x78A000`**

```
void collect_garbage() {
   .... look at the stack ....
}
```

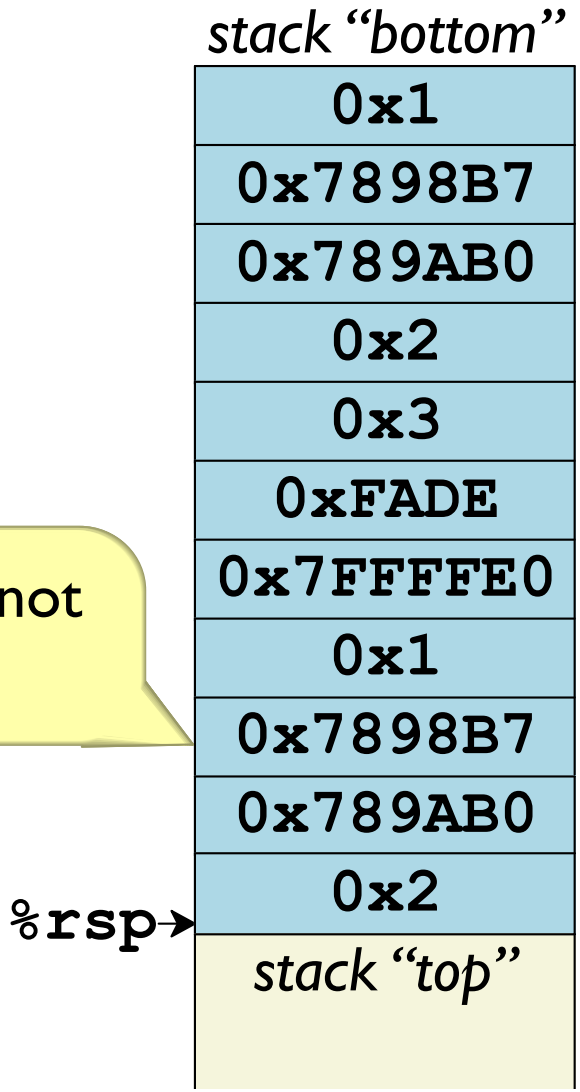


Conservative GC

Node allocation range:
`0x789800` to **`0x78A000`**

```
void collect_garbage() {
   ...
}
```

In node pointer range, but not suitably aligned



Conservative GC

Need to know machine details:

- find stack bounds
- inspect registers
- find global variables

Conservative because some things are guesses

- Pointer or number?

  Rarely a problem in practice

- Stack position or register has live variable?

  More of a problem in practice

Conservative guesses often better than manual attempts

Convervative GC Implementation

Semi-portable stack finding:

allocate.h

```
#define main original_main
```

gc.c

```
static void **stack_init;

int main(int argc, char **argv) {
  /* Record the original stack position */
  stack_init = &argc;

  return original_main(argc, argv);
}

void find_roots() {
   void **stack_now = (void **)&stack_now;
   ....
}
```

Convervative GC Implementation

gc.c

```
void collect_garbage() {
  find_roots();
  mark_and_sweep_from_roots();
}
```

Convervative GC Implementation

gc.c

```
static void find_roots() {
  void **stack_now = (void **)&stack_now;
  num_roots = 0;

  while ((void *)stack_now < stack_init) {
    if (looks_like_allocated(*stack_now))
      root_addrs[num_roots++] = (struct node **)stack_now;
    stack_now++;
  }
}
```

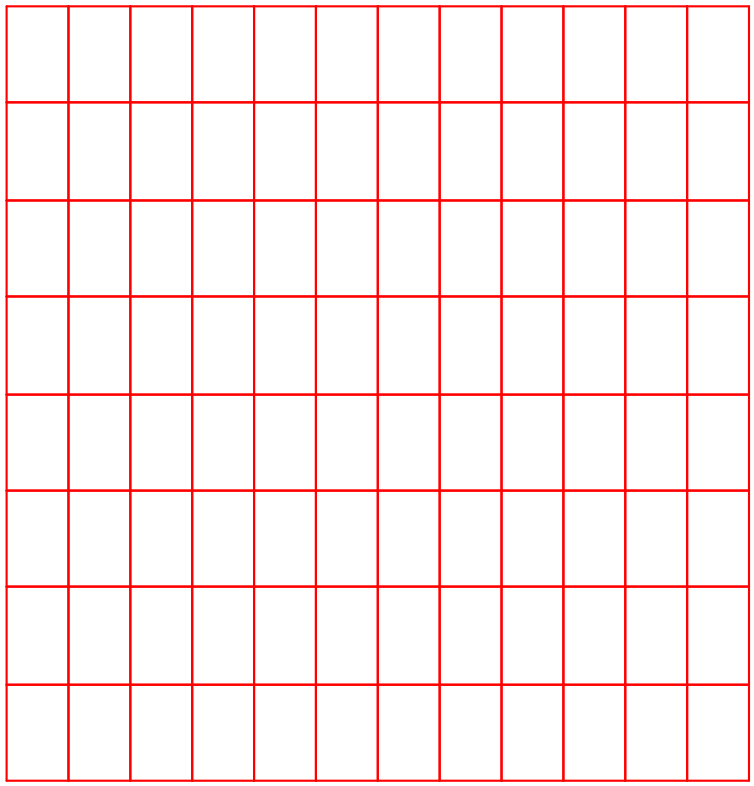
Convervative GC Implementation

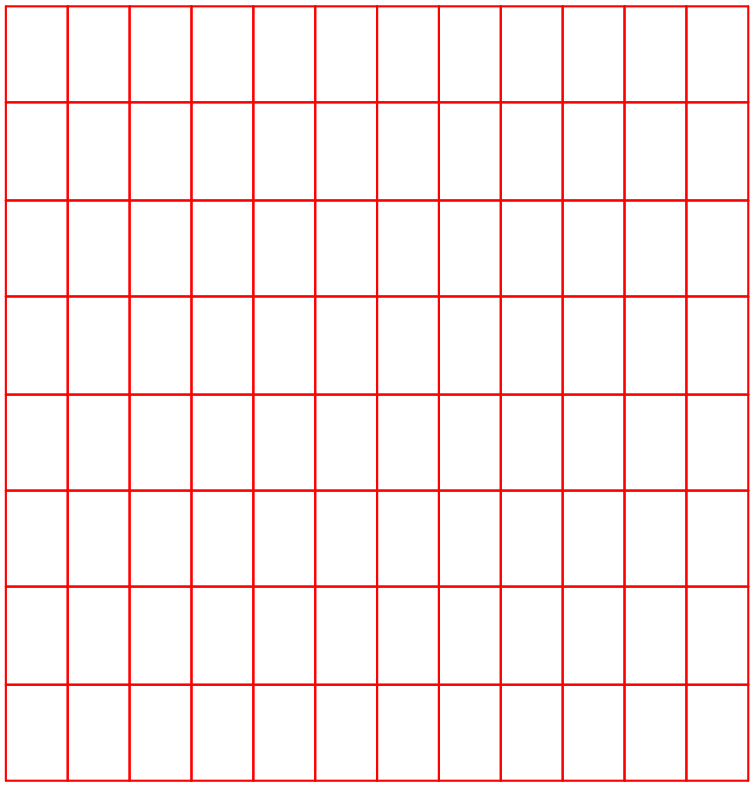
gc.c

```
int looks_like_allocated(struct gc_node *p) {
  struct gc_chunk *c;

  /* Resides in a chunk? */
  for (c = chunks; c; c = c->next) {
    if ((p >= c->mem)
        && (p <= (c->mem + NODES_PER_CHUNK))) {
      /* Properly aligned? */
      uintptr_t delta = ((char *)p - (char *)c->mem);
      if ((delta % sizeof(struct gc_node))
          == offsetof(struct gc_node, n))
        return 1;
    }
  }

  return 0;
}
```

Two-Space Copying Collectors

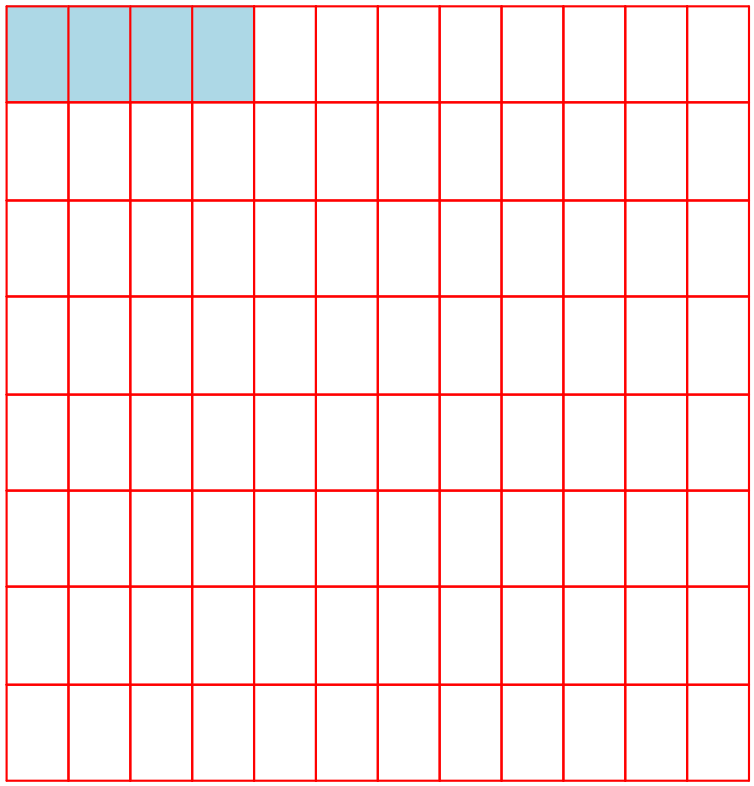
A ***two-space*** copying collector compacts memory as it collects, making allocation easier.

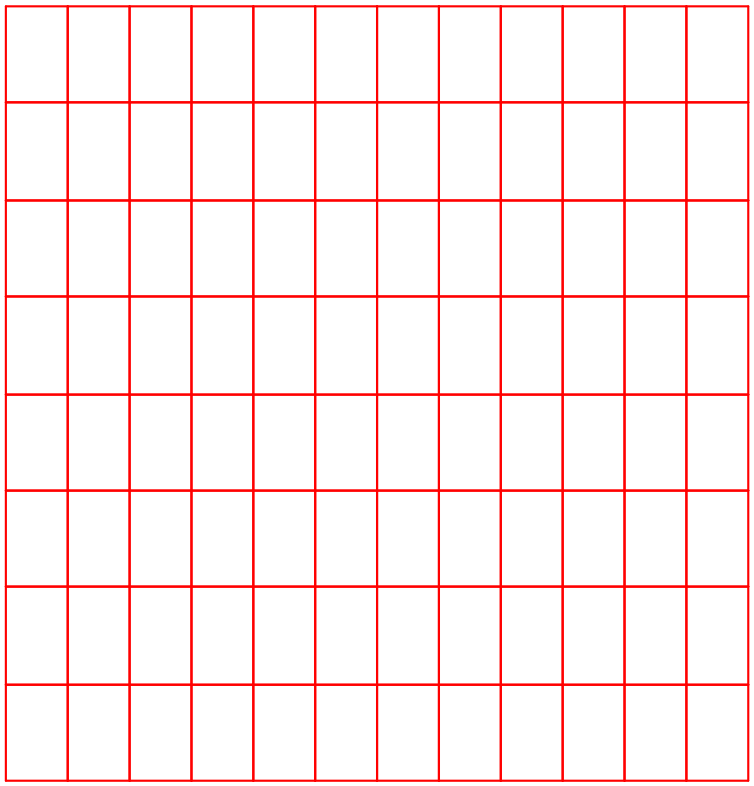




Two-Space Copying Collectors

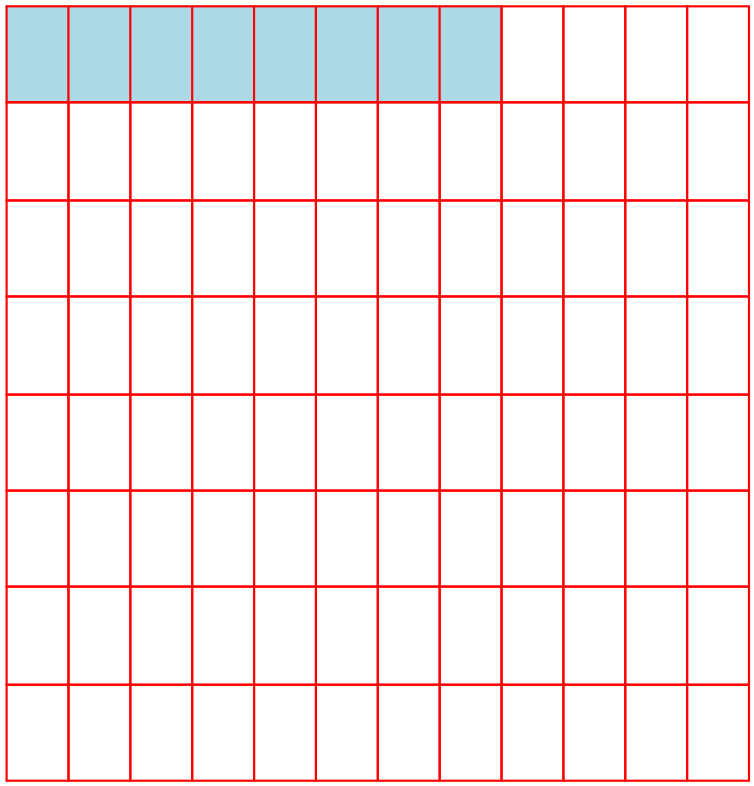
A ***two-space*** copying collector compacts memory as it collects, making allocation easier.

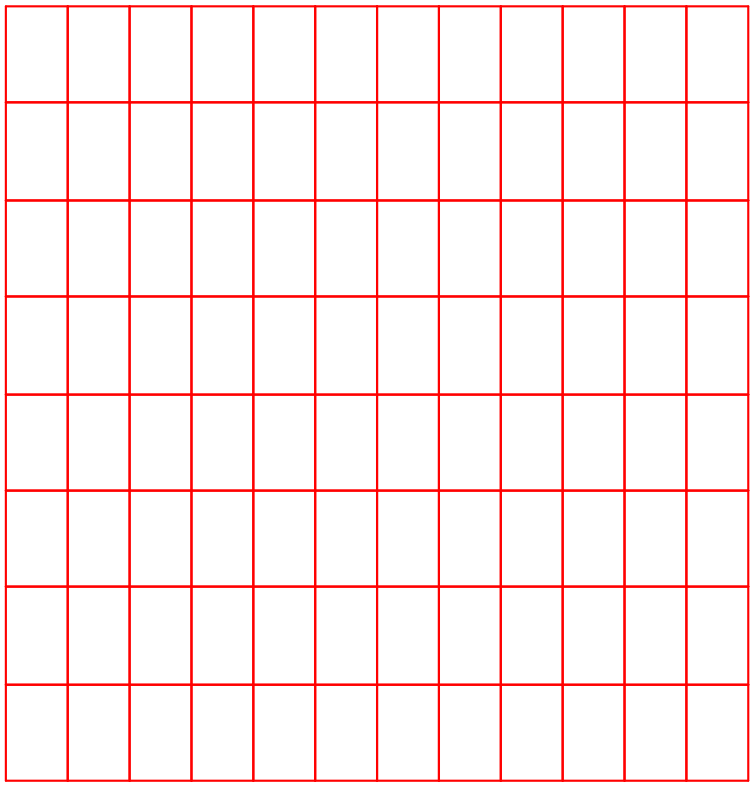




Two-Space Copying Collectors

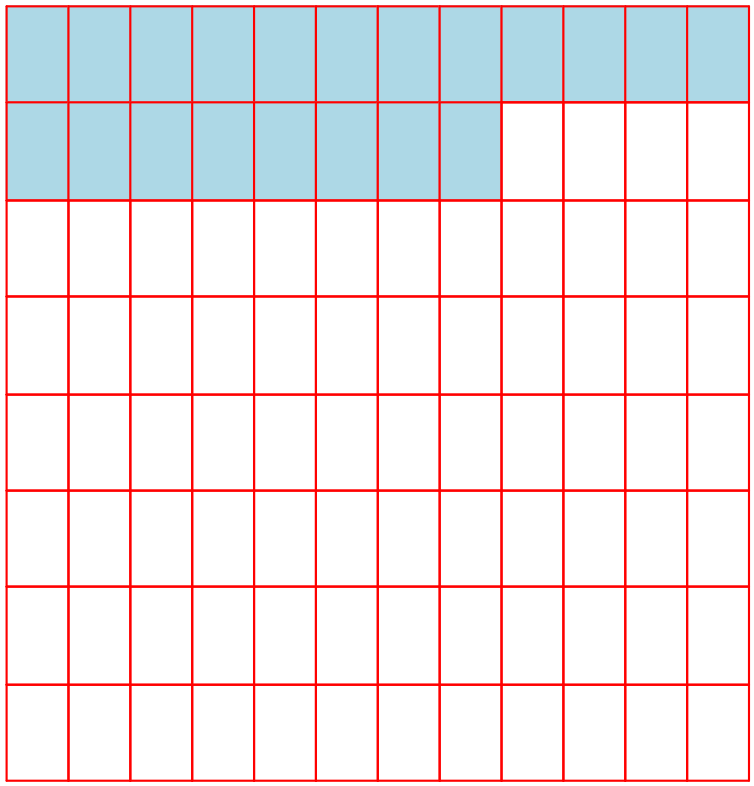
A ***two-space*** copying collector compacts memory as it collects, making allocation easier.

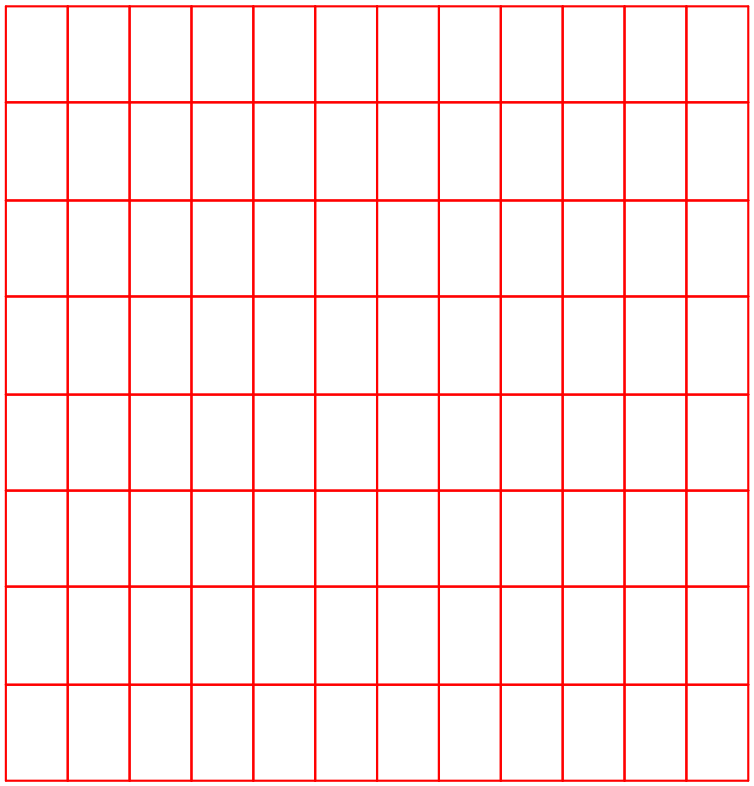




Two-Space Copying Collectors

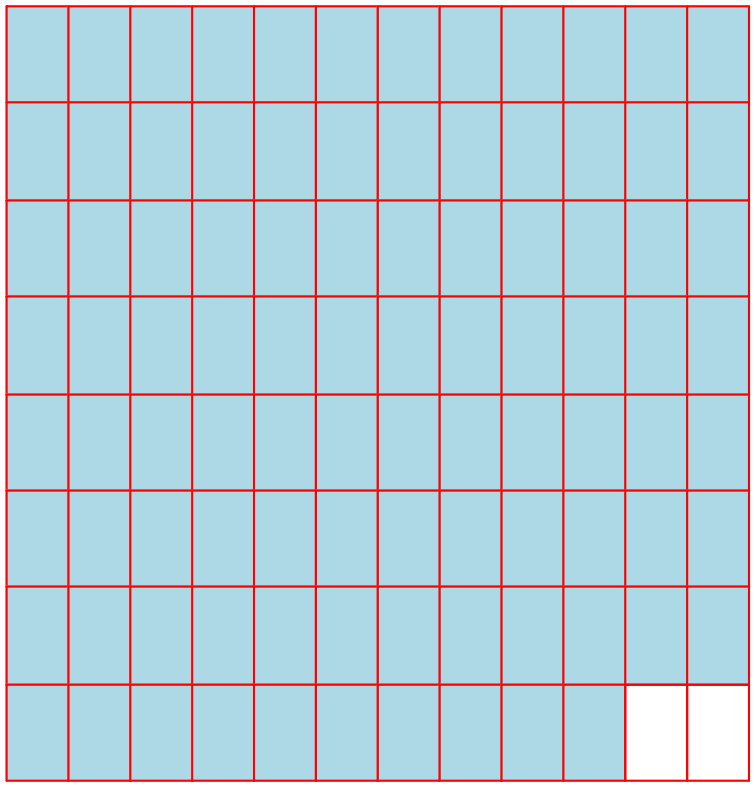
A ***two-space*** copying collector compacts memory as it collects, making allocation easier.

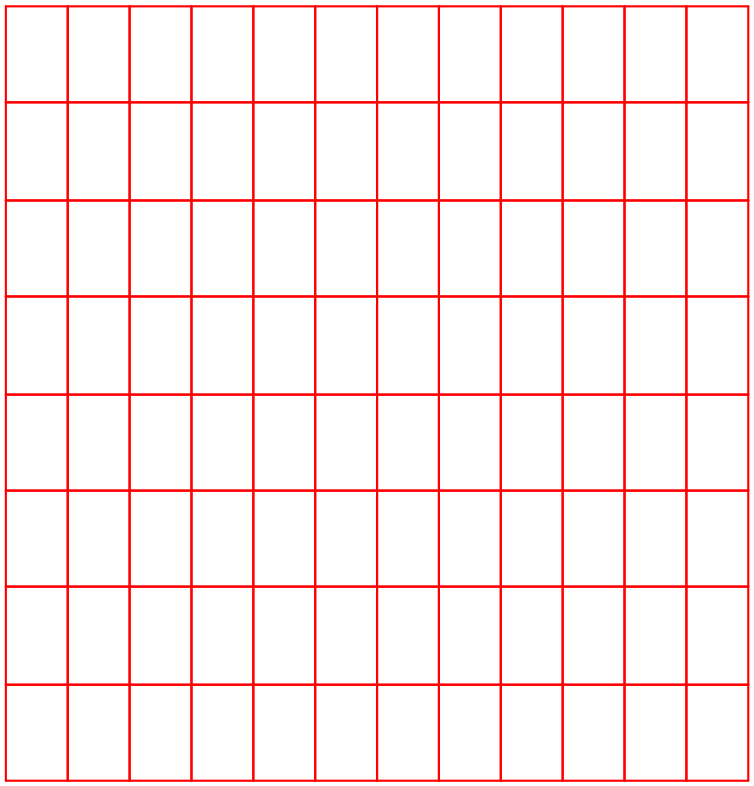




Two-Space Copying Collectors

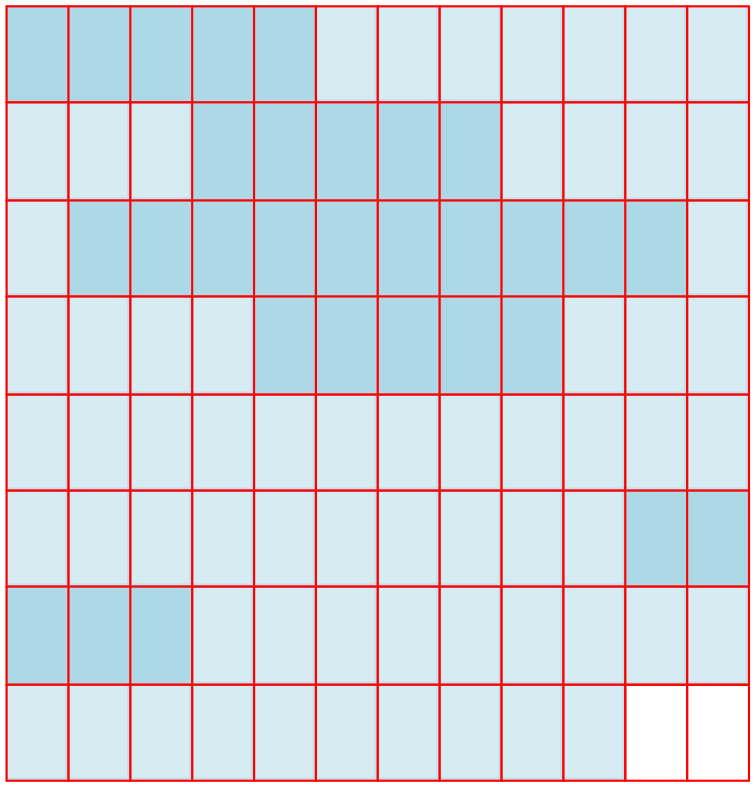
A ***two-space*** copying collector compacts memory as it collects, making allocation easier.

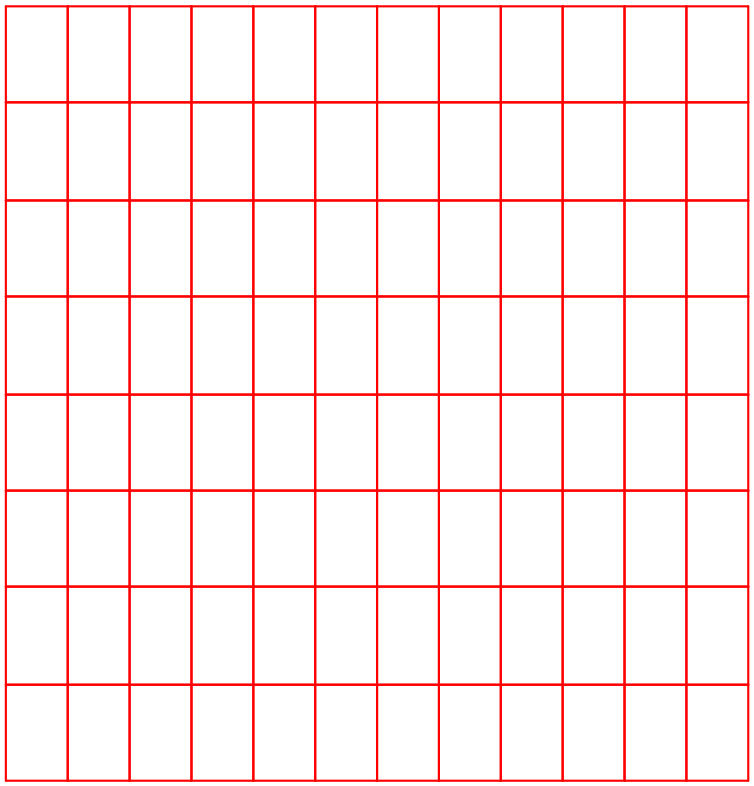




Two-Space Copying Collectors

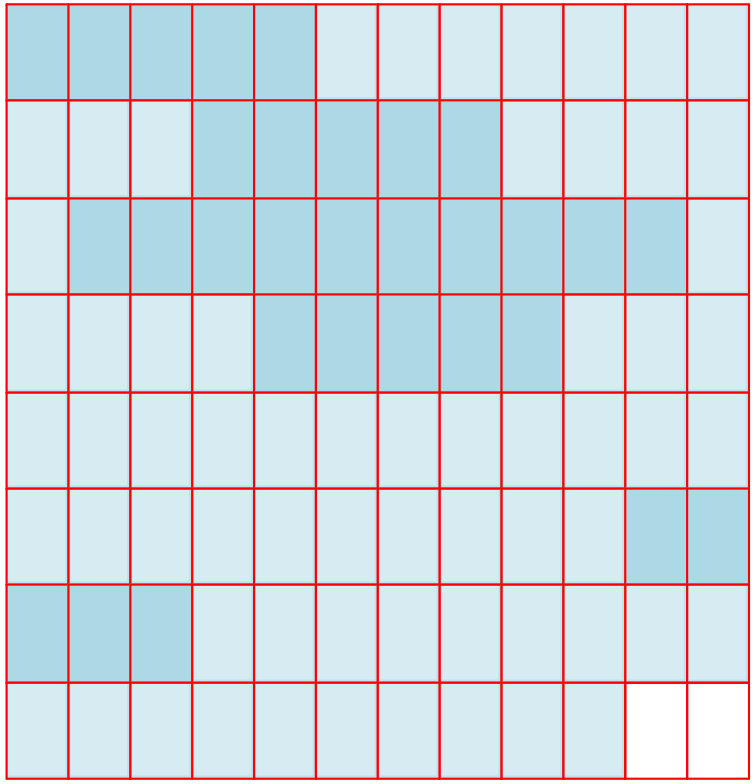
A ***two-space*** copying collector compacts memory as it collects, making allocation easier.

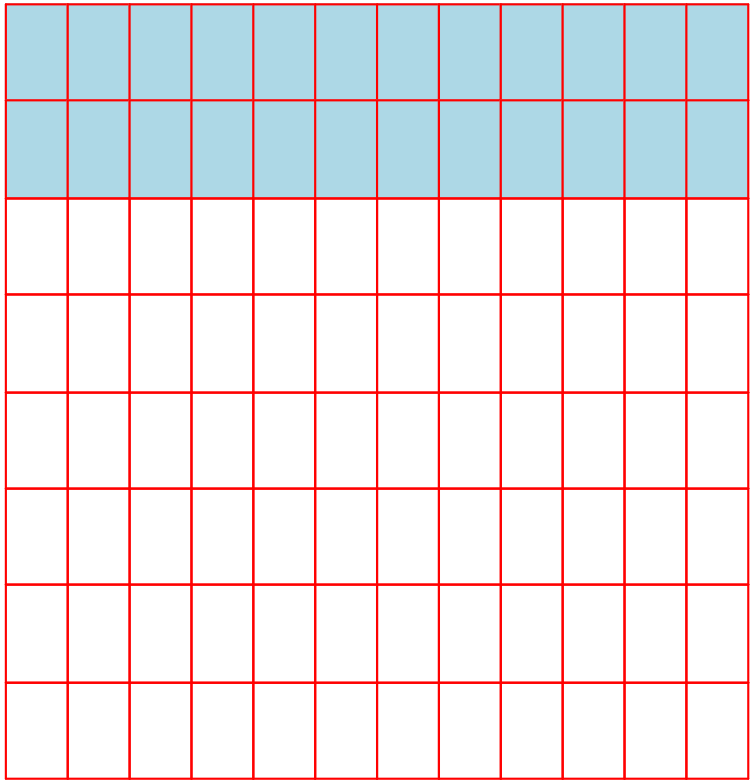




Two-Space Copying Collectors

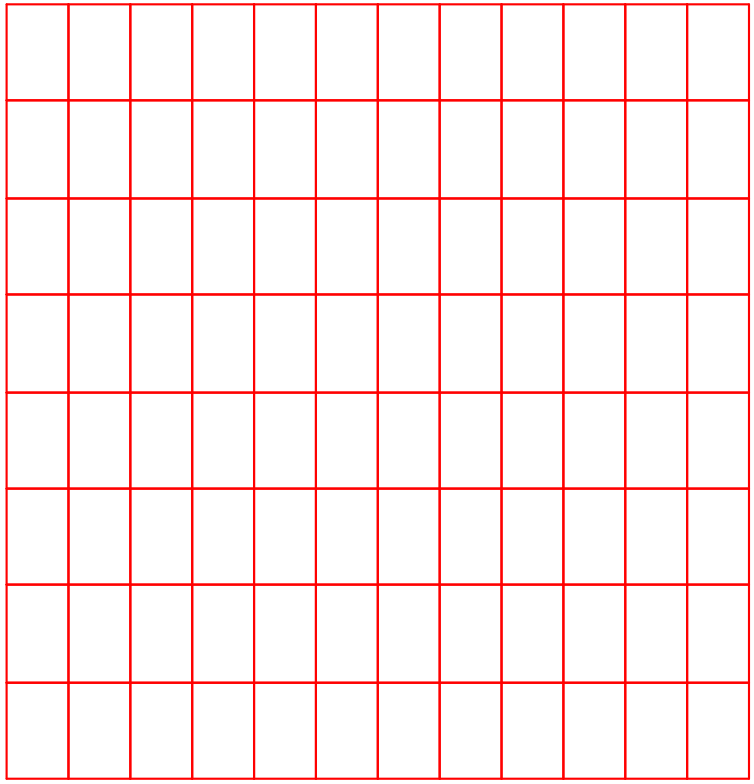
A ***two-space*** copying collector compacts memory as it collects, making allocation easier.

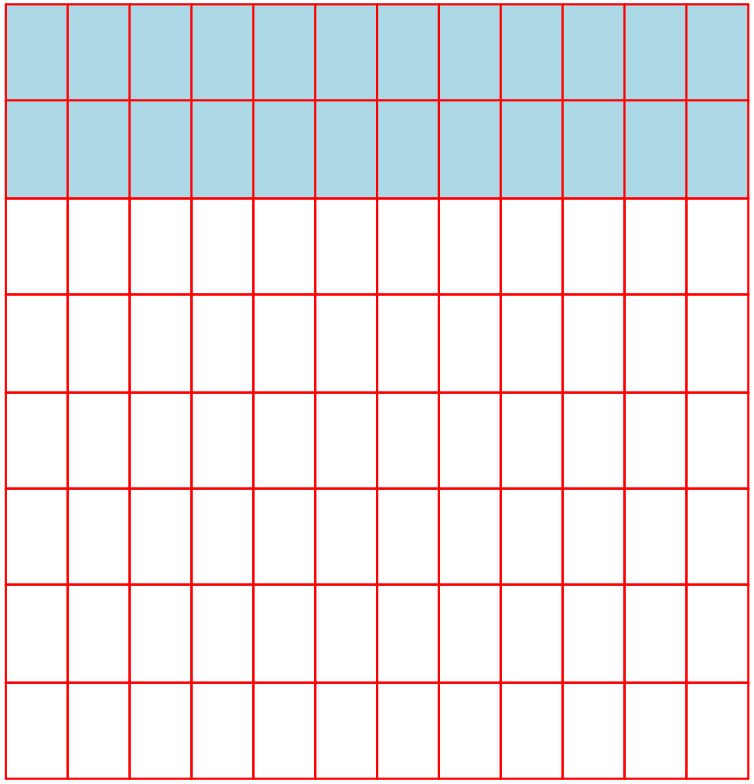




Two-Space Copying Collectors

A ***two-space*** copying collector compacts memory as it collects, making allocation easier.





Two-Space Copying Collectors

A ***two-space*** copying collector compacts memory as it collects, making allocation easier.

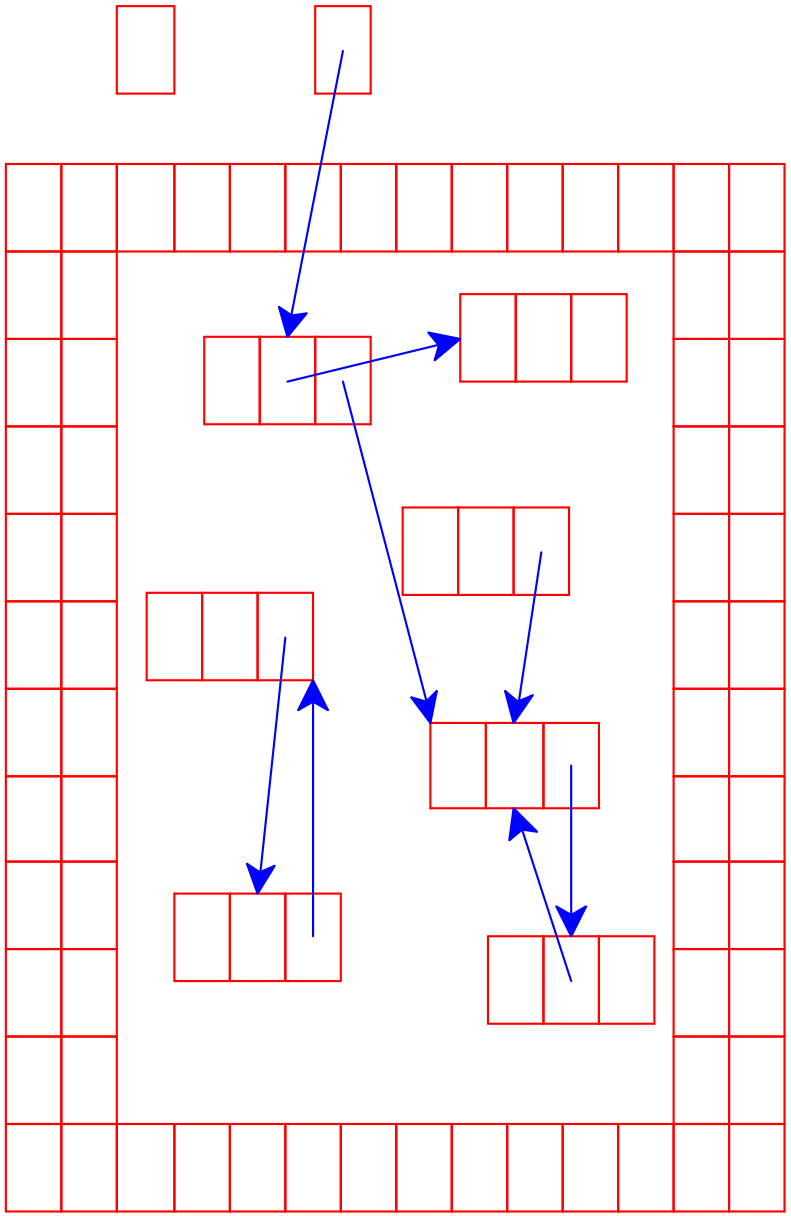
Allocator:

- Partitions memory into ***to-space*** and ***from-space***
- Allocates only in ***to-space***

Collector:

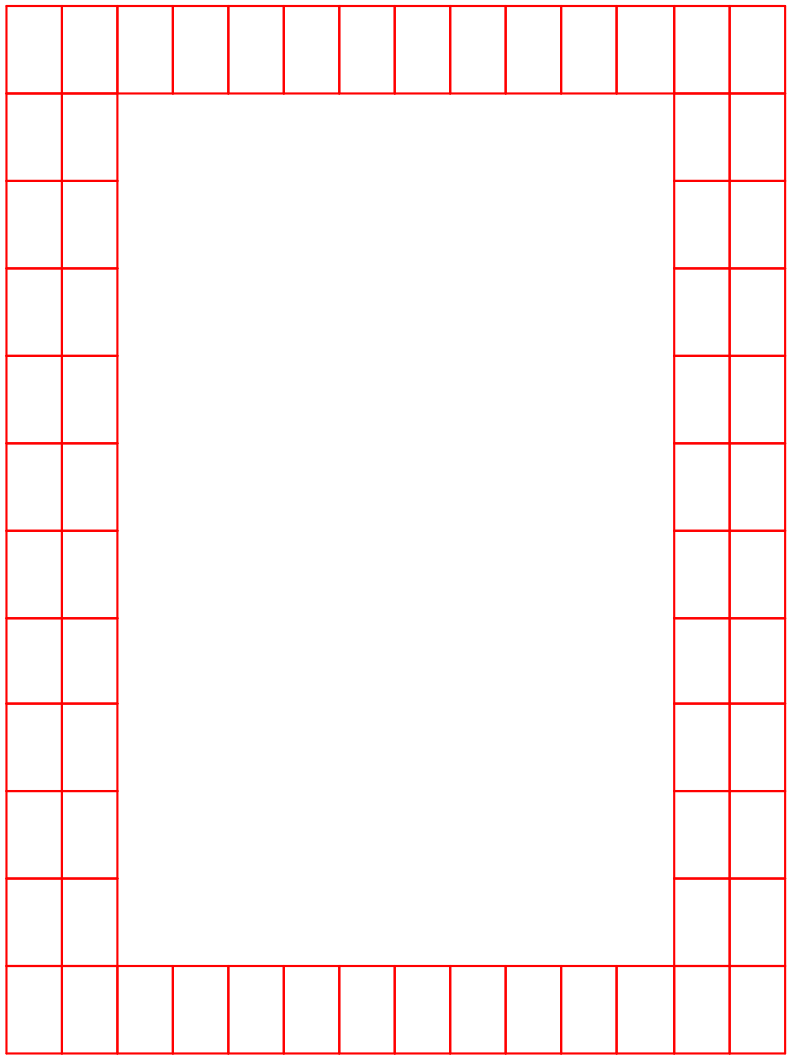
- Starts by swapping ***to-space*** and ***from-space***
- Coloring gray ⇒ copy from ***from-space*** to ***to-space***
- Choosing a gray object ⇒ walk once though the new ***to-space***, update pointers

Two-Space Collection

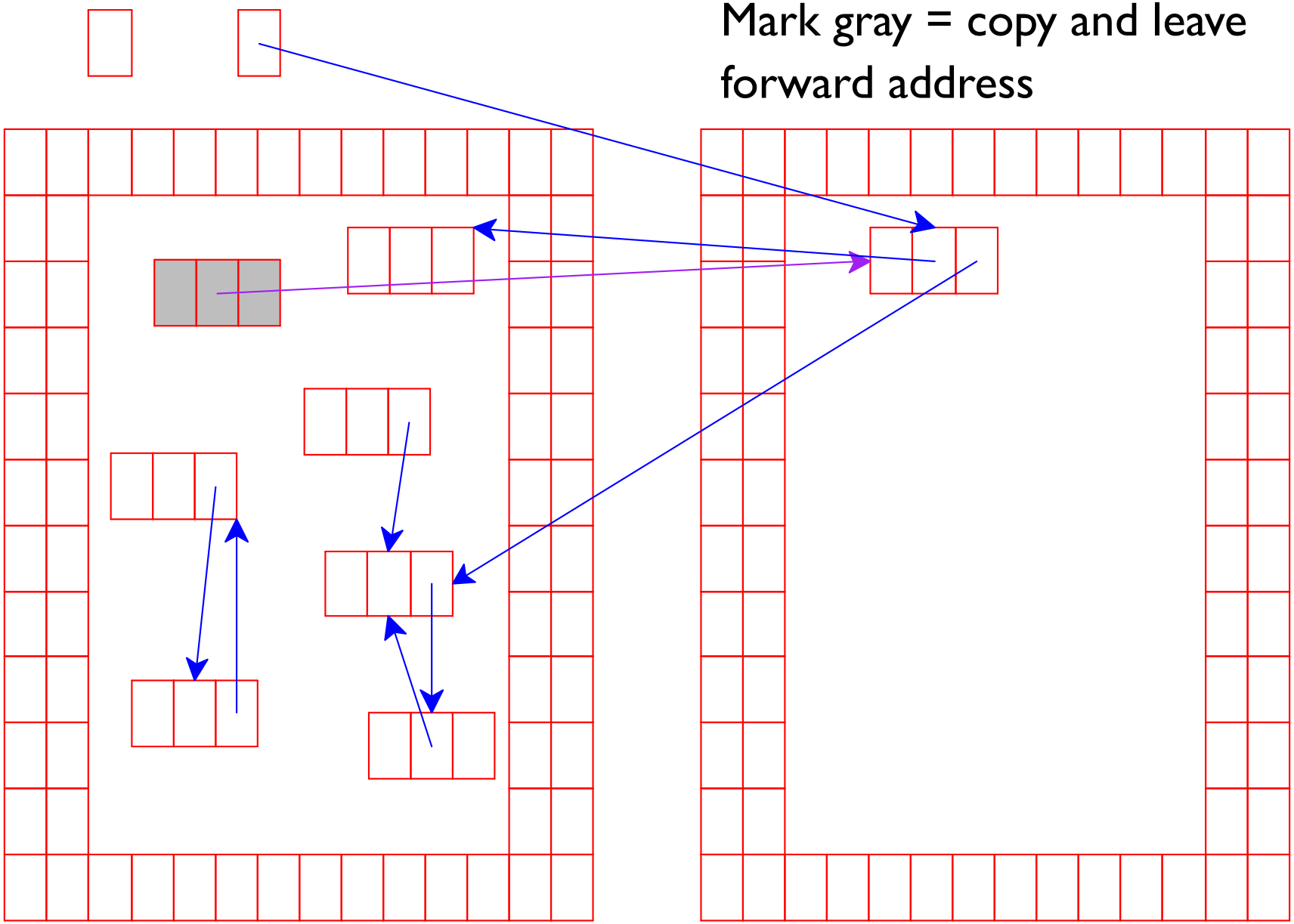


Left = from-space

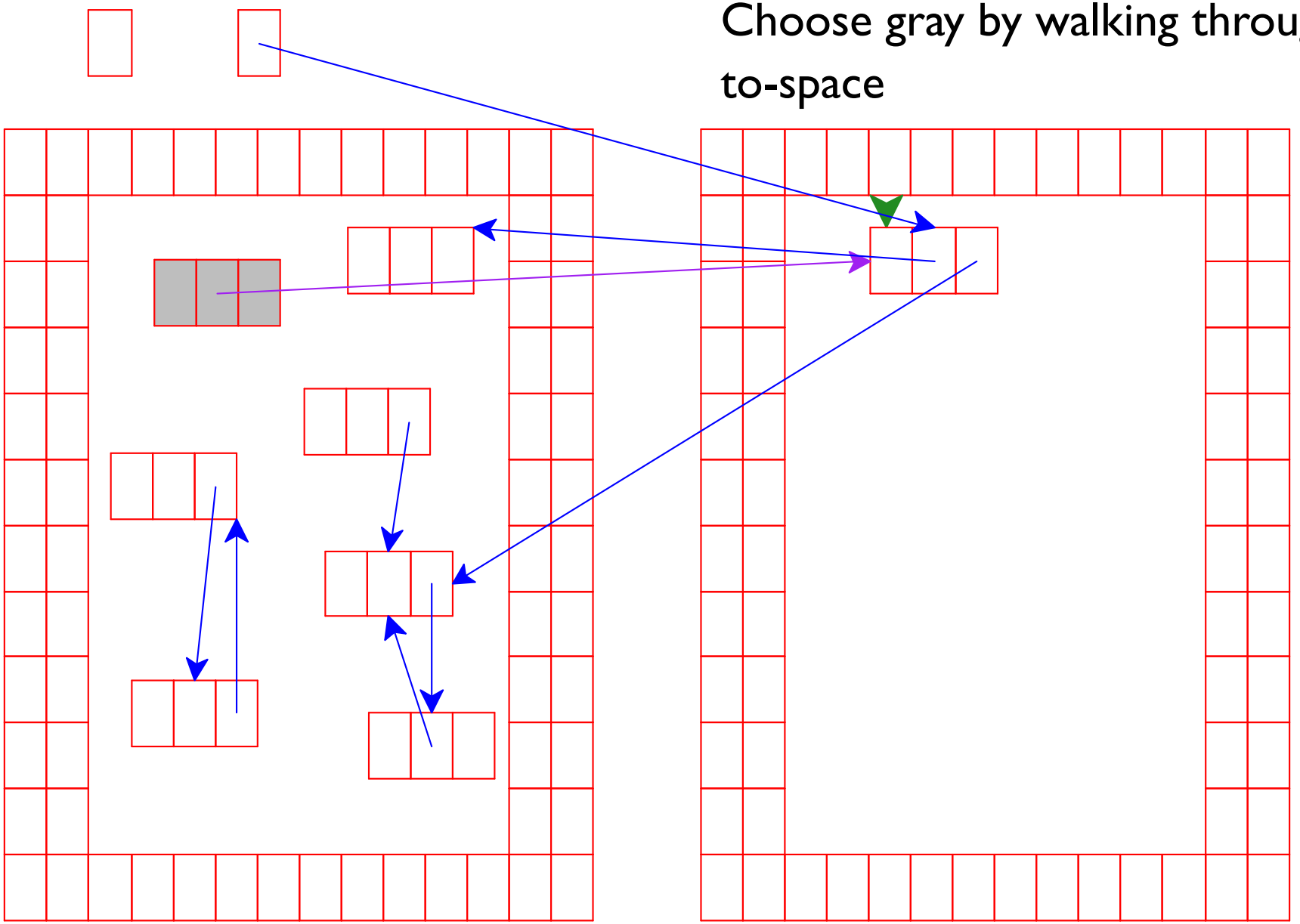
Right = to-space



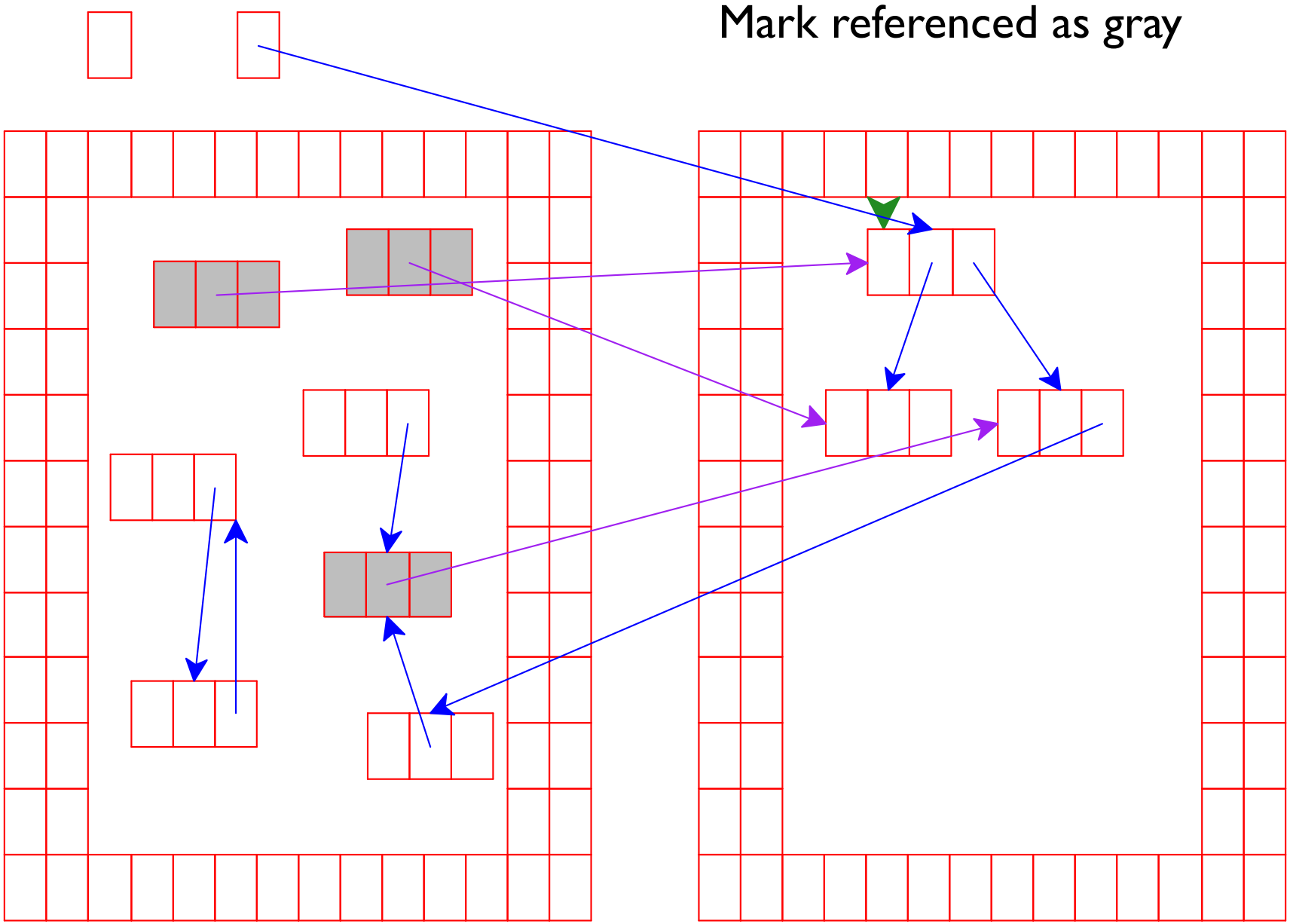
Two-Space Collection



Two-Space Collection

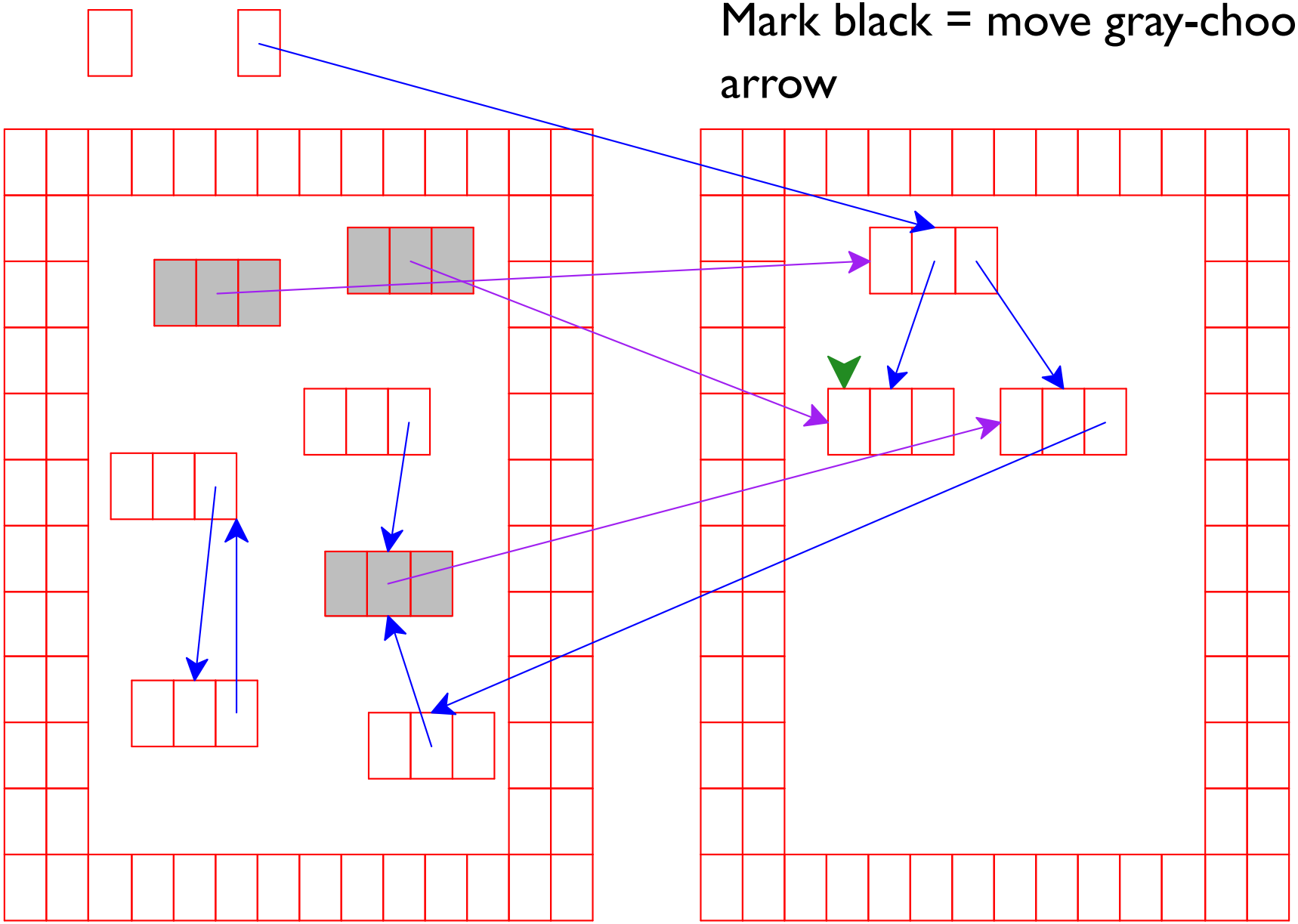


Two-Space Collection

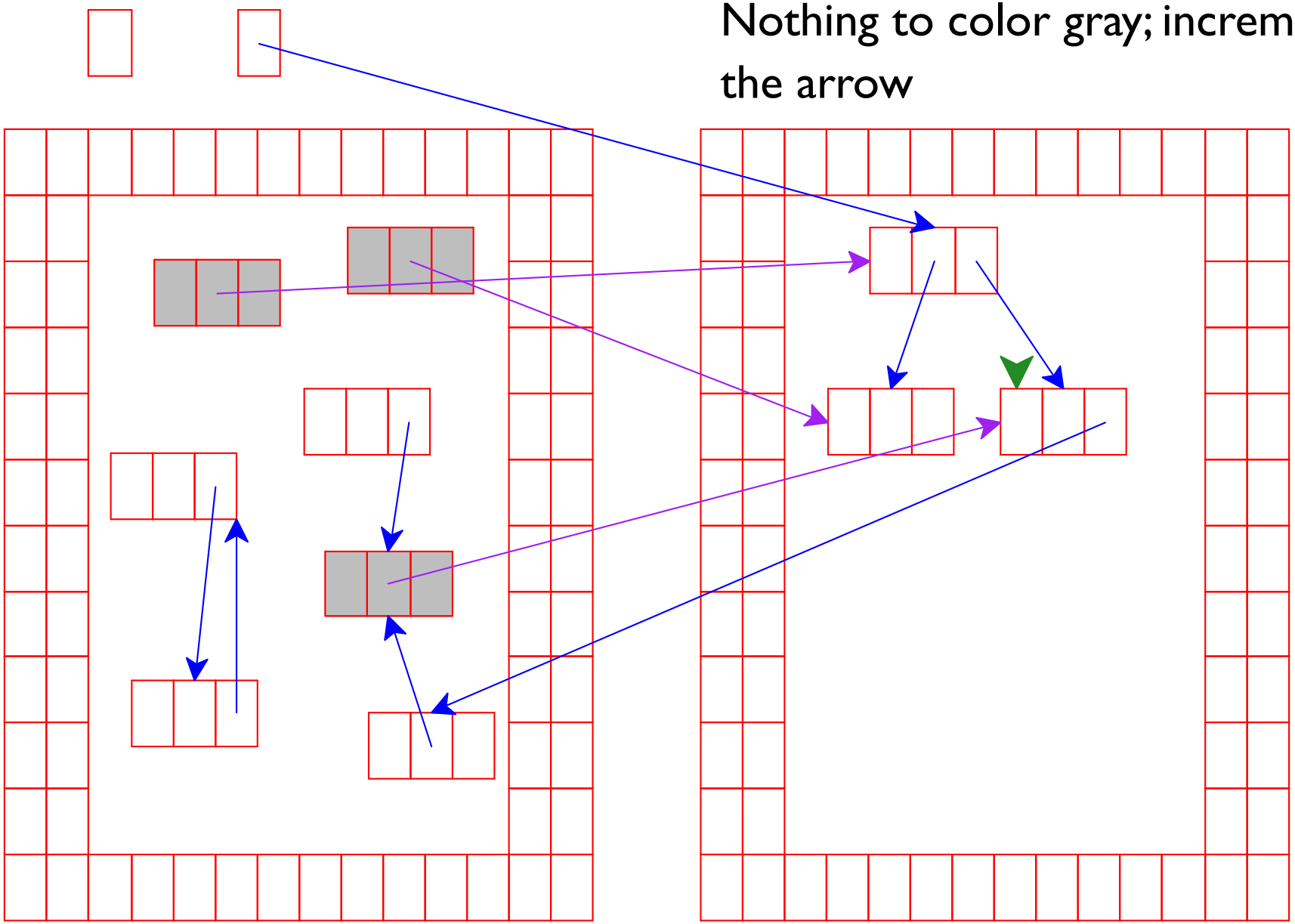


Two-Space Collection

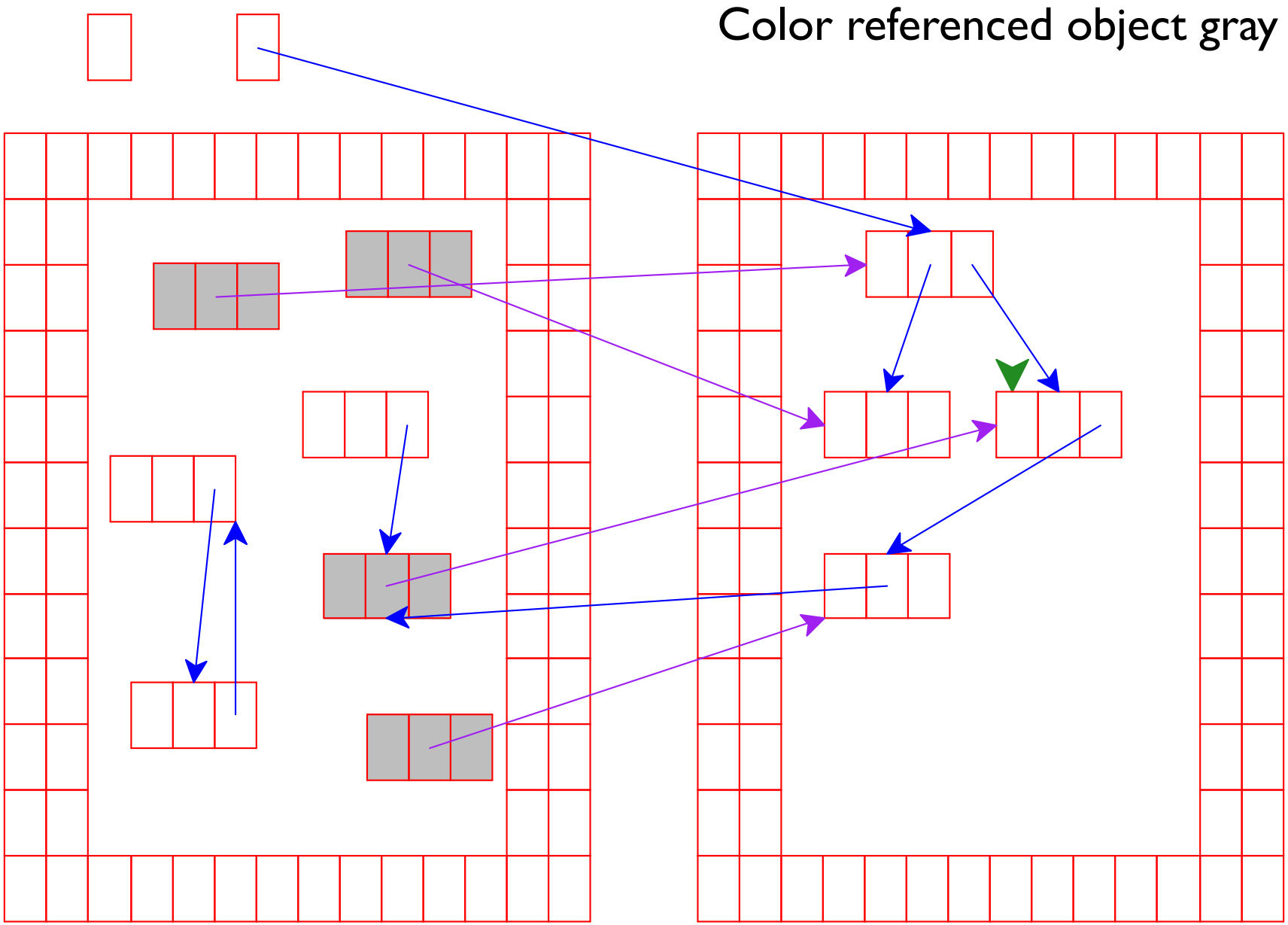
Mark black = move gray-choosing arrow



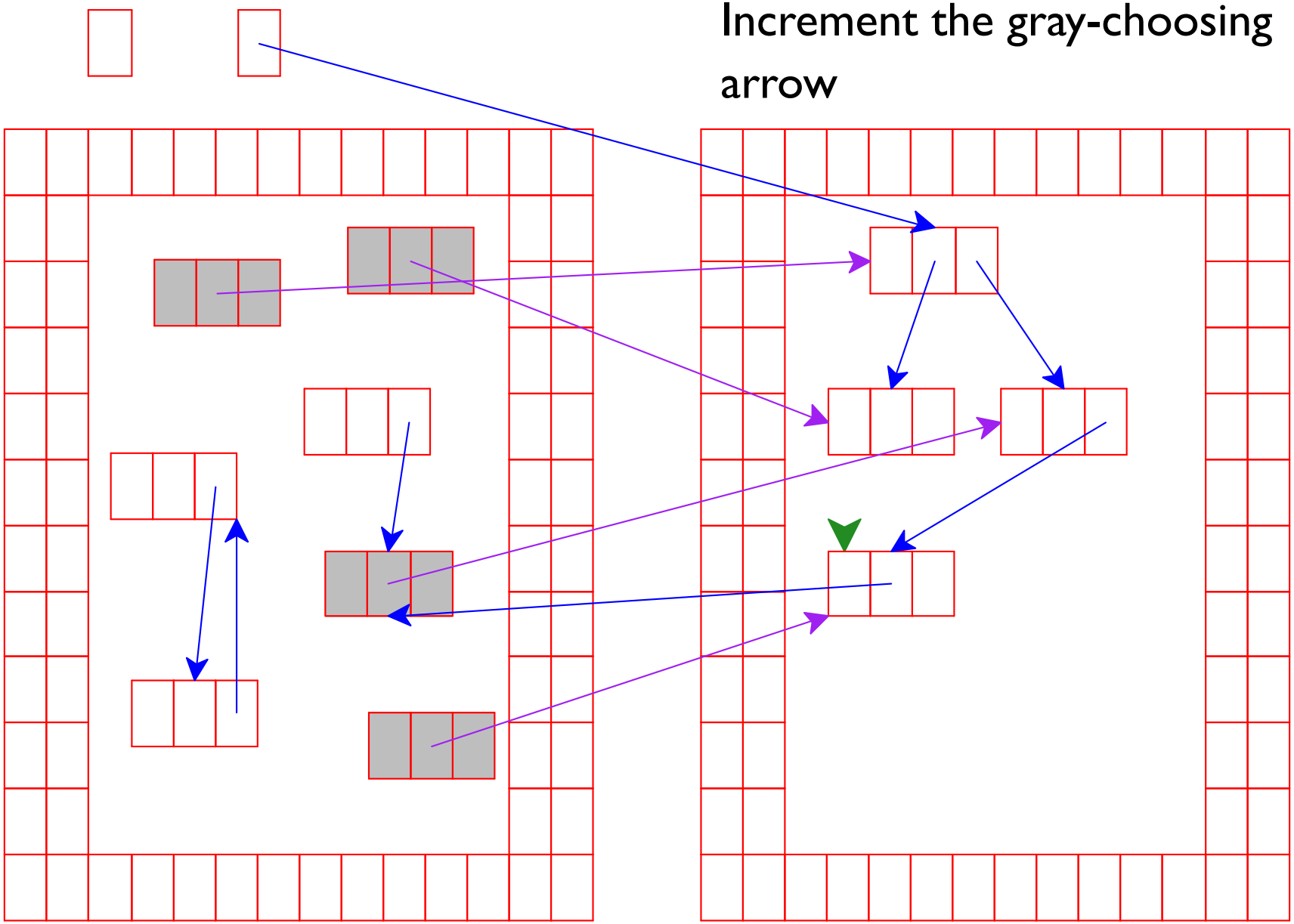
Two-Space Collection



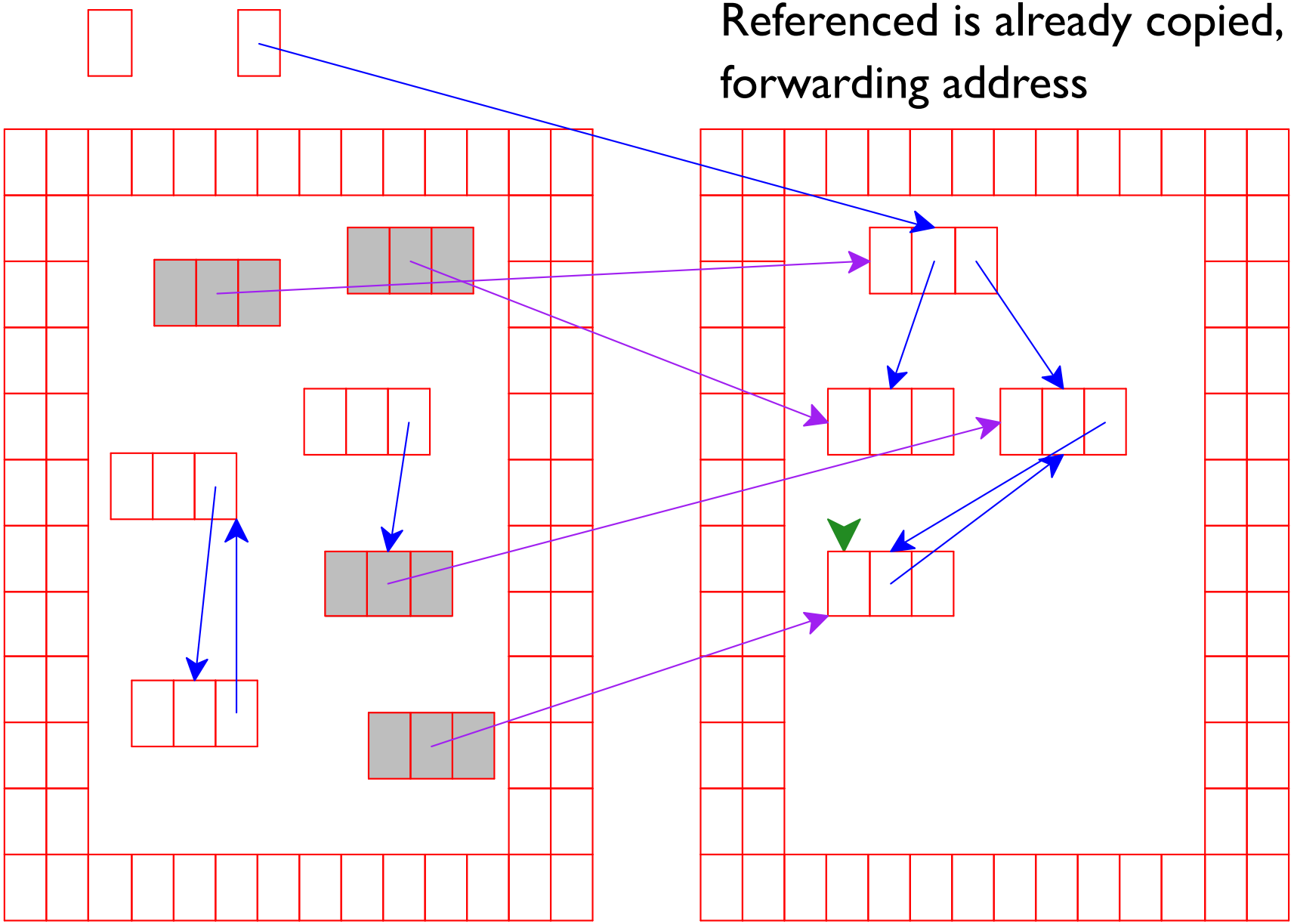
Two-Space Collection



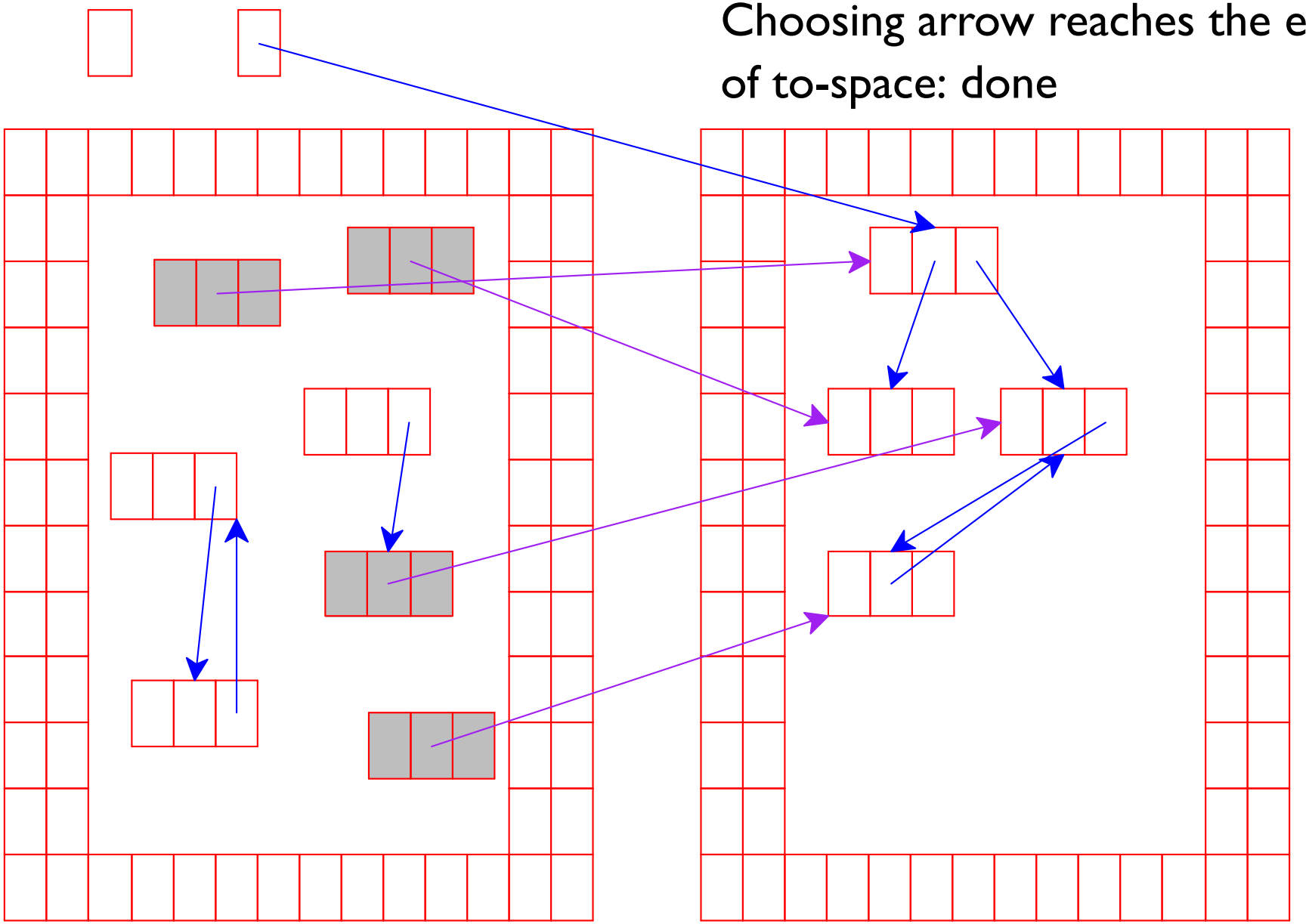
Two-Space Collection



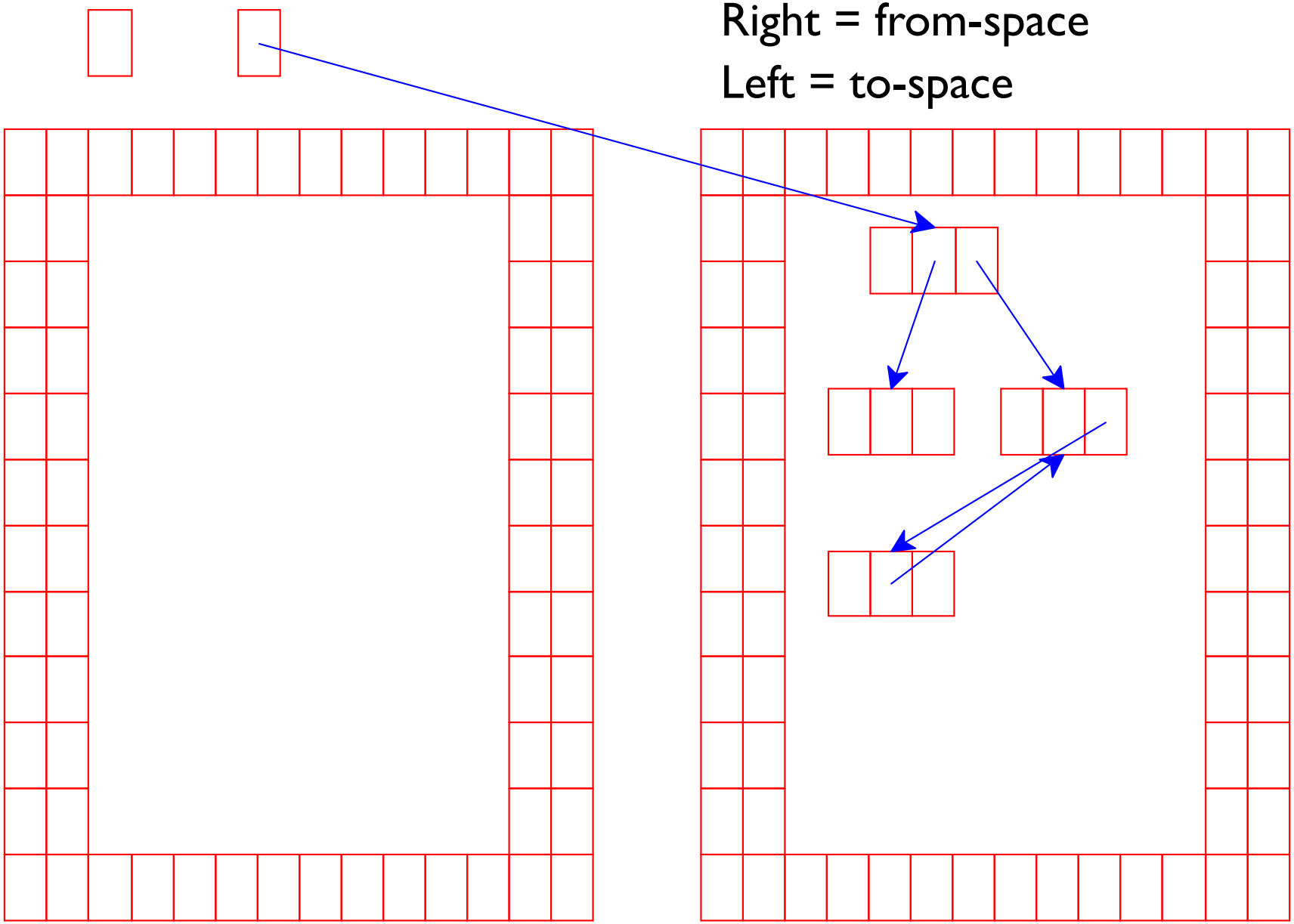
Two-Space Collection



Two-Space Collection



Two-Space Collection



Two-Space Implementation

gc.c

```
/* Current allocation region: */
uintptr_t space_start, space_end;
/* Current allocation pointer: */
uintptr_t space_next;

struct node *allocate() {
  struct gc_node *p;

  if (space_next + sizeof(struct gc_node) > space_end)
    collect_garbage();

  p = (struct gc_node *)space_next;
  space_next += sizeof(struct gc_node);

  p->forwarded = 0;

  return &p->n;
}
```

Two-Space Implementation

gc.c

```
void collect_garbage() {
  ....
  void *new_space = raw_malloc(new_size);
  struct gc_node *alloc_pos = new_space;

  /* Copy into new space */
  copy_from_roots(&alloc_pos);

  /* Clean up old space */
  raw_free((void *)space_start, space_end - space_start);

  /* Continue allocating into new space */
  space_start = (uintptr_t)new_space;
  space_end = space_start + new_size;
  space_next = (uintptr_t)alloc_pos;
  ....
}
```

Two-Space Implementation

gc.c

```
void copy_from_roots(struct gc_node **alloc_pos_ptr) {
  struct gc_node *init_ptr = *alloc_pos_ptr;

  for (int i = 0; i < num_roots; i++)
    copy_or_forward(root_addrs[i], alloc_pos_ptr);

  traverse_copied(init_ptr, alloc_pos_ptr);
}
```

Two-Space Implementation

gc.c

```
void copy_or_forward(struct node **addr,
                     struct gc_node **alloc_pos_ptr) {
  if (*addr) {
    struct gc_node *nh = NODE_TO_GC(*addr);

    if (nh->forwarded)
      *addr = nh->forward_to; /* already copied */
    else {
      struct gc_node *nh2 = *alloc_pos_ptr;
      (*alloc_pos_ptr)++;
      memcpy(nh2, nh, sizeof(struct gc_node));

      nh->forwarded = 1;
      nh->forward_to = &nh2->n;
      *addr = &nh2->n;
    }
  }
}
```

Two-Space Implementation

Iterate through new space to turn "gray" nodes "black"

gc.c

```
void traverse_copied(struct gc_node *traverse_ptr,
                     struct gc_node **alloc_pos_ptr) {
  while (traverse_ptr < *alloc_pos_ptr) {
    copy_or_forward(&traverse_ptr->n.left, alloc_pos_ptr);
    copy_or_forward(&traverse_ptr->n.right, alloc_pos_ptr);
    traverse_ptr++;
  }
}
```

Realistic Garbage Collection

Our collector knows only one shape:

```
mark(nh->n.left);
mark(nh->n.right);
```

Realistic collectors deal with more object shapes

Realistic Garbage Collection

Typically, each object has a ***tag*** to identify its shape

The collector must map a tag to a traversal function

```
tag = ((gc_any *)p)->tag;
switch (tag) {
 case TREE_NODE:
   nh = (gc_node *)p;
   mark(nh->n.left);
   mark(nh->n.right);
   break;
 case ARRAY:
   a = (gc_array *)p;
   for (i = 0; i < a->size; i++)
     mark(a->elems[i]);
   break;
 ....
 }
```

Miniature Two-Space Example with Tags

- 26-byte memory (13 bytes for each space), 2 variables
  - Tag 1: one integer
  - Tag 2: one pointer
  - Tag 3: one integer, then one pointer

Variable 1: 7 Variable 2: 0

Miniature Two-Space Example with Tags

- 26-byte memory (13 bytes for each space), 2 variables
  - Tag 1: one integer
  - Tag 2: one pointer
  - Tag 3: one integer, then one pointer

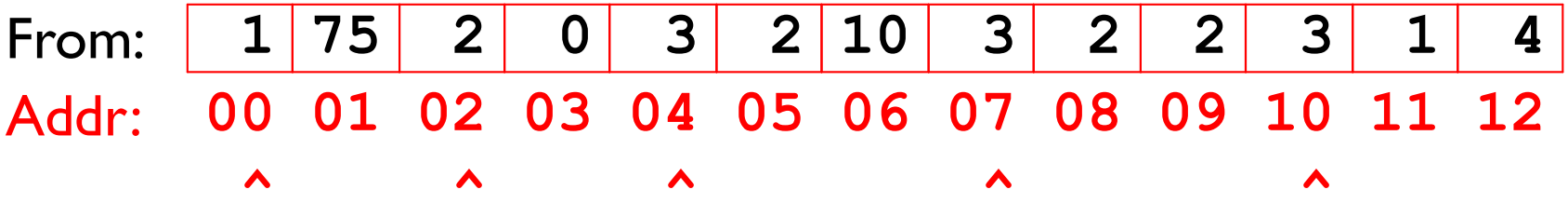
Variable 1: 7 Variable 2: 0

From:	1	75	2	0	3	2	10	3	2	2	3	1	4
Addr:	00	01	02	03	04	05	06	07	08	09	10	11	12

Miniature Two-Space Example with Tags

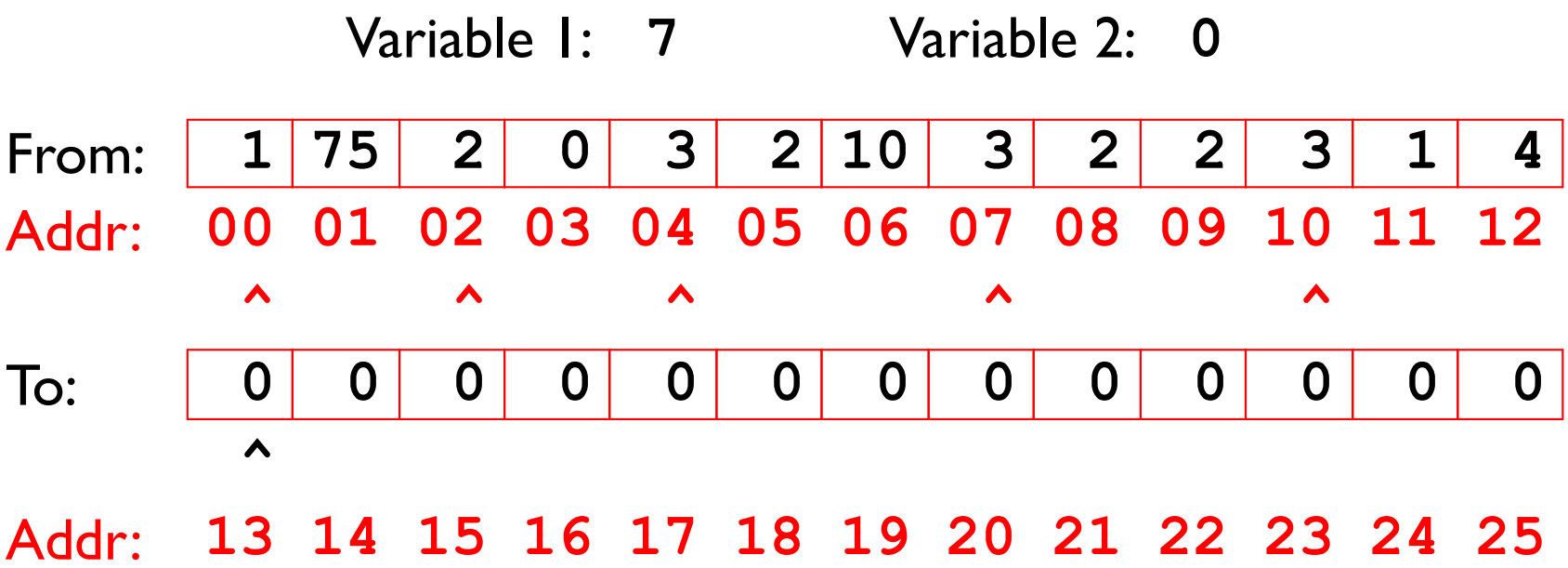
- 26-byte memory (13 bytes for each space), 2 variables
  - Tag 1: one integer
  - Tag 2: one pointer
  - Tag 3: one integer, then one pointer

Variable 1: 7 Variable 2: 0



Miniature Two-Space Example with Tags

- 26-byte memory (13 bytes for each space), 2 variables
  - Tag 1: one integer
  - Tag 2: one pointer
  - Tag 3: one integer, then one pointer



Miniature Two-Space Example with Tags

- 26-byte memory (13 bytes for each space), 2 variables
  - Tag 1: one integer
  - Tag 2: one pointer
  - Tag 3: one integer, then one pointer

Variable 1: 13 Variable 2: 0

From:	1	75	2	0	3	2	10	99	13	2	3	1	4
Addr:	00	01	02	03	04	05	06	07	08	09	10	11	12
	^		^		^			^			^		
To:	3	2	2	0	0	0	0	0	0	0	0	0	0
	^												
Addr:	13	14	15	16	17	18	19	20	21	22	23	24	25

Miniature Two-Space Example with Tags

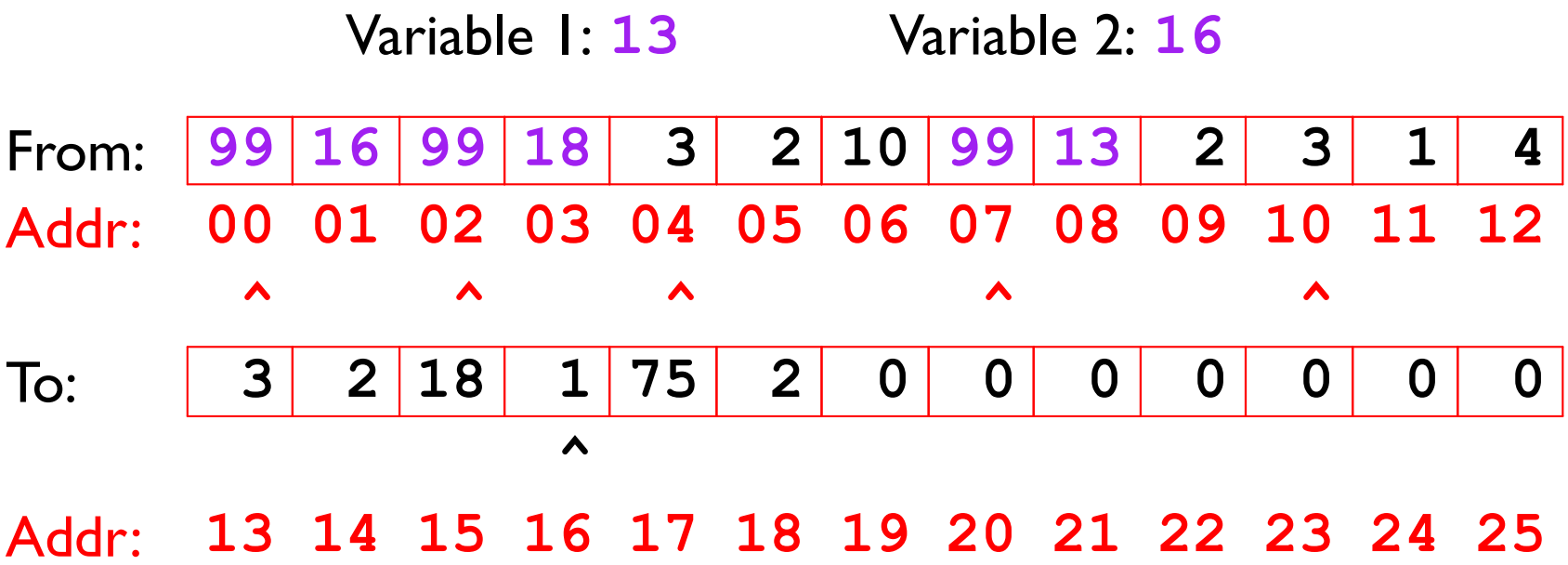
- 26-byte memory (13 bytes for each space), 2 variables
  - Tag 1: one integer
  - Tag 2: one pointer
  - Tag 3: one integer, then one pointer

Variable 1: 13 Variable 2: 16

From:	99	16	2	0	3	2	10	99	13	2	3	1	4
Addr:	00	01	02	03	04	05	06	07	08	09	10	11	12
	^		^		^			^			^		
To:	3	2	2	1	75	0	0	0	0	0	0	0	0
	^												
Addr:	13	14	15	16	17	18	19	20	21	22	23	24	25

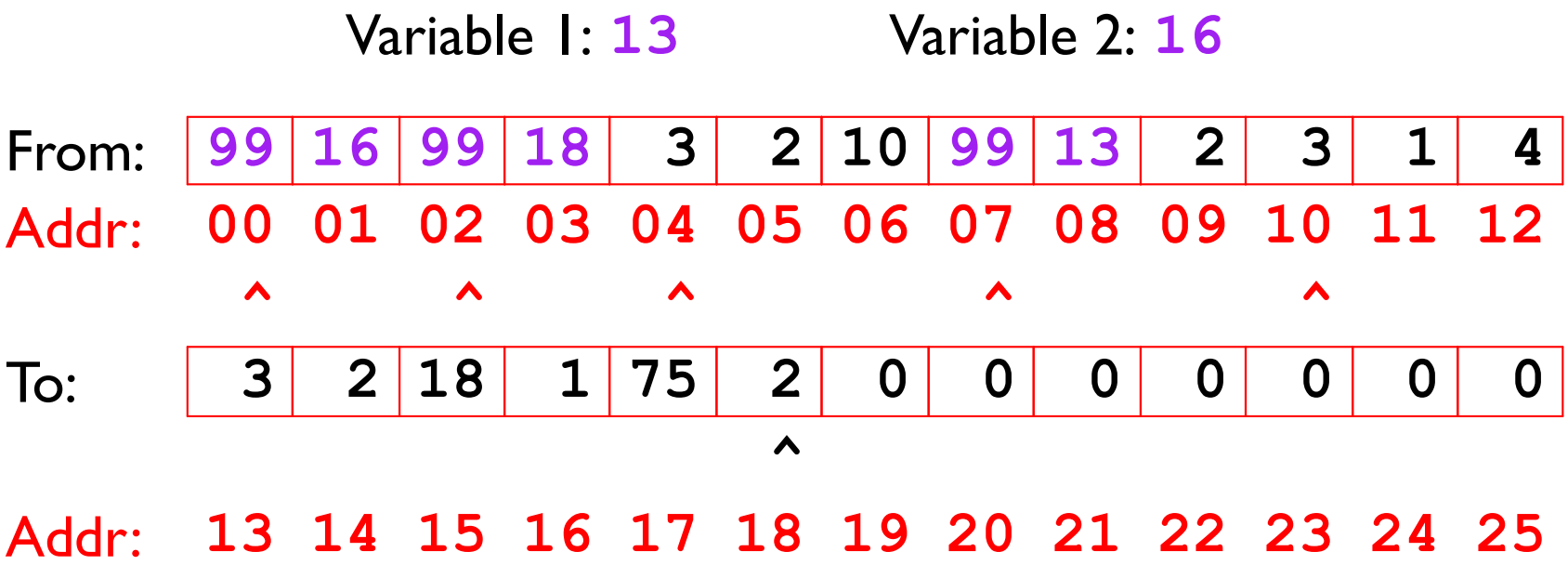
Miniature Two-Space Example with Tags

- 26-byte memory (13 bytes for each space), 2 variables
  - Tag 1: one integer
  - Tag 2: one pointer
  - Tag 3: one integer, then one pointer



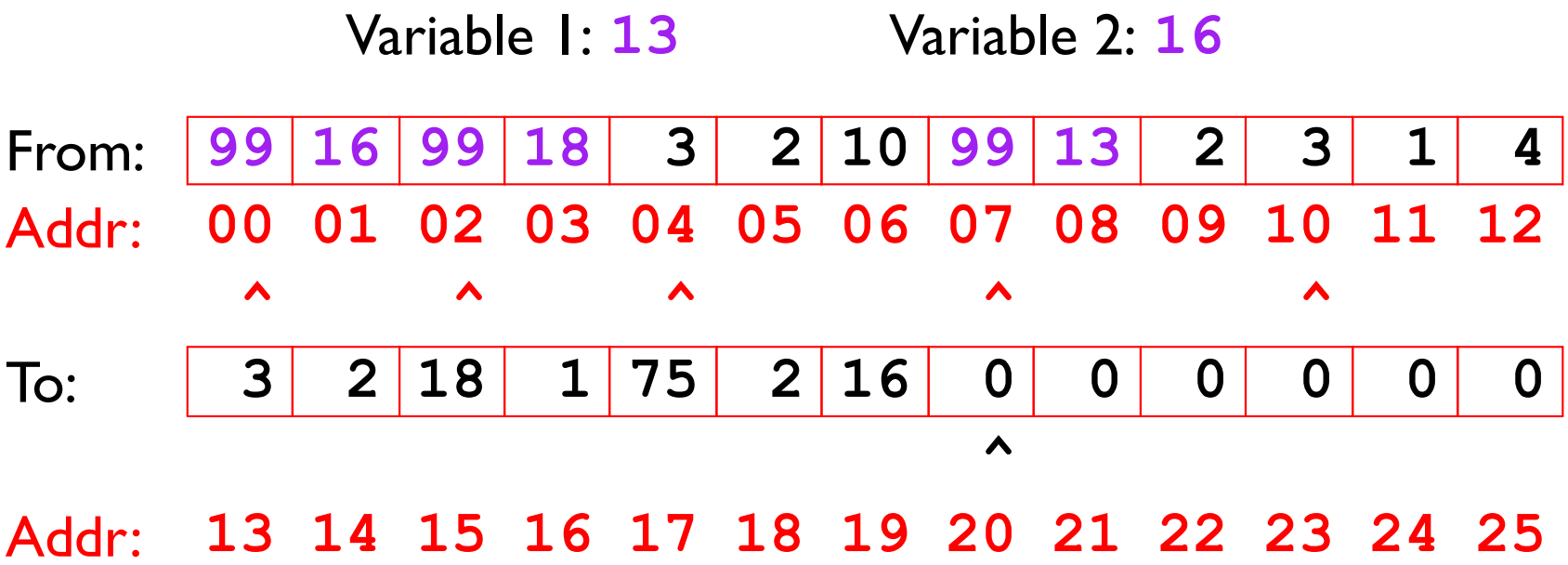
Miniature Two-Space Example with Tags

- 26-byte memory (13 bytes for each space), 2 variables
  - Tag 1: one integer
  - Tag 2: one pointer
  - Tag 3: one integer, then one pointer



Miniature Two-Space Example with Tags

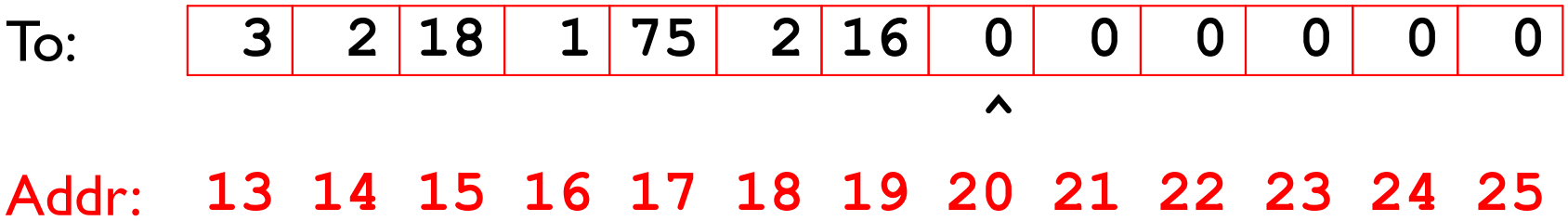
- 26-byte memory (13 bytes for each space), 2 variables
  - Tag 1: one integer
  - Tag 2: one pointer
  - Tag 3: one integer, then one pointer



Miniature Two-Space Example with Tags

- 26-byte memory (13 bytes for each space), 2 variables
  - Tag 1: one integer
  - Tag 2: one pointer
  - Tag 3: one integer, then one pointer

Variable 1: `13` Variable 2: `16`



More GC Topics

Generational collection

mostly try to collect recent allocations

Compacting collection

only copy sometimes — good locality, fewer copies

Hybrid schemes

e.g., copying for nursery, compacting for old generation

Incremental collection

perform some collection work on each allocation