Manual Memory Management

Allocation:

```
s = (snake *)malloc(sizeof(snake));
```

Deallocation:

```
free(s);
```

When to Deallocate

```
int main() {
  snake *s;
  s = (snake *)malloc(sizeof(snake));
  ....
}
```

don't deallocate **s**, since **exit** releases all allocation

When to Deallocate

```
int simulate_zoo() {
  snake *s;
  s = (snake *)malloc(sizeof(snake));
  ....
  free(s);
}

int main() {
  while(...) { ... simulate_zoo(); ... }
}
```

do deallocate **s**, since **simulate_zoo** might be called many times

From the HW11 Solution

```
res_listing* make_res_listing(char* last,
                              char* first,
                              char* num) {
  res_listing* r = malloc(sizeof(res_listing));
  r->l.type = res_type;
  r->l.number = strdup(num);
  r->last = strdup(last);
  r->first = strdup(first);
  return r;
}
```

+ caller doesn't have to worry about string lifetimes

– explicit **`free_res_listing`** is needed

List Deallocator

```
void free_listing(res_listing* r) {
  free(r->l.number);
  free(r->last);
  free(r->first);
  free(r);
}
```

From the HW11 Solution

```
char* read_another_line(FILE* f) {
  char buffer[256];
  if (!fgets(buffer, 256, f))
    buffer[0] = 0;
  strip_newline(buffer);
  return strdup(buffer);
}
```

+ caller doesn't have to supply a buffer
+ code could be improved to handle longer lines
− caller is responsible for **`free`**

From the HW11 Solution

```
last = read_another_line(f);
first = read_another_line(f);
number = read_another_line(f);
l = make_res_listing(last, first, number);
```

Should add

```
free(last);
free(first);
free(number);
```

From the HW11 Solution

```
last = read_another_line(f);
first = read_another_line(f);
number = read_another_line(f);
l = make_res_listing(last, first, number);
```

Should add

```
free(last);
free(first);
free(number);
```

Containers

```
typedef struct node {
  void* val;
  int height;
  struct node *left;
  struct node *right;
} node;

struct avl_tree {
  node* root;
  compare_proc compare;
};

avl_tree* make_avl_tree(compare_proc compare);
```

\+ function pointers never need to be deallocated

+/− **`free_avl`** should free all internal nodes

? values in the container?

Containers

```
void free_avl_tree(avl_tree* t) {
  free_node(t->root);
  free(t);
}

void free_node(node* n) {
  if (n) {
    free_node(n->left);
    free_node(n->right);
  }
}
```

Options for values:

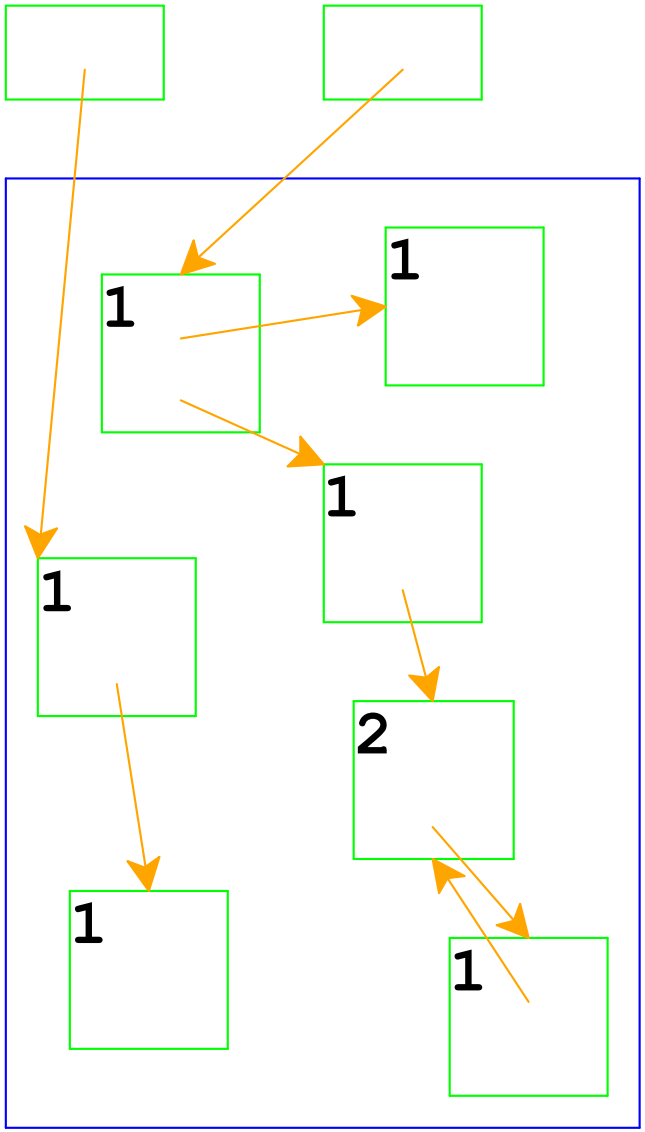
- Ignore, so client is responsible
- Accept a **`free_proc`** along with **`compare_proc`**
- Require that a particular deallocator works

Reference Counting

Reference counting: a way to know whether a record has other users

- Attatch a count to every record, starting at 0
- When installing a pointer to a record (into a register or another record), increment its count
- When replacing a pointer to a record, decrement its count
- When a count is decremented to 0, decrement counts for other records referenced by the record, then free it

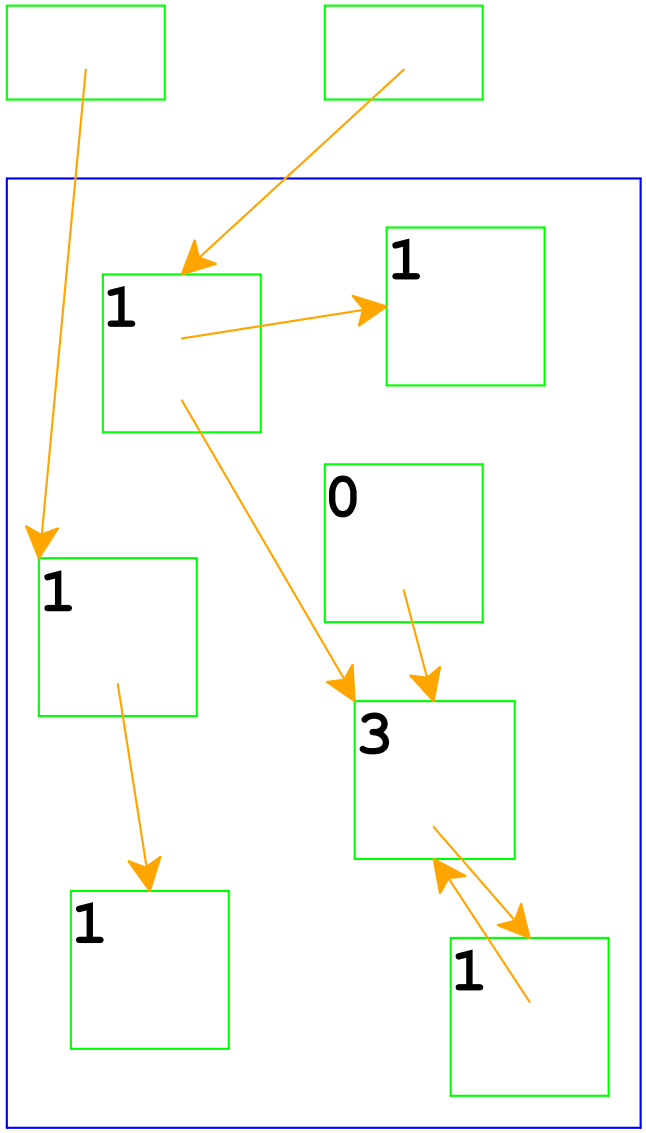
Reference Counting



Top boxes are the registers **`expr`**, **`todos`**, etc.

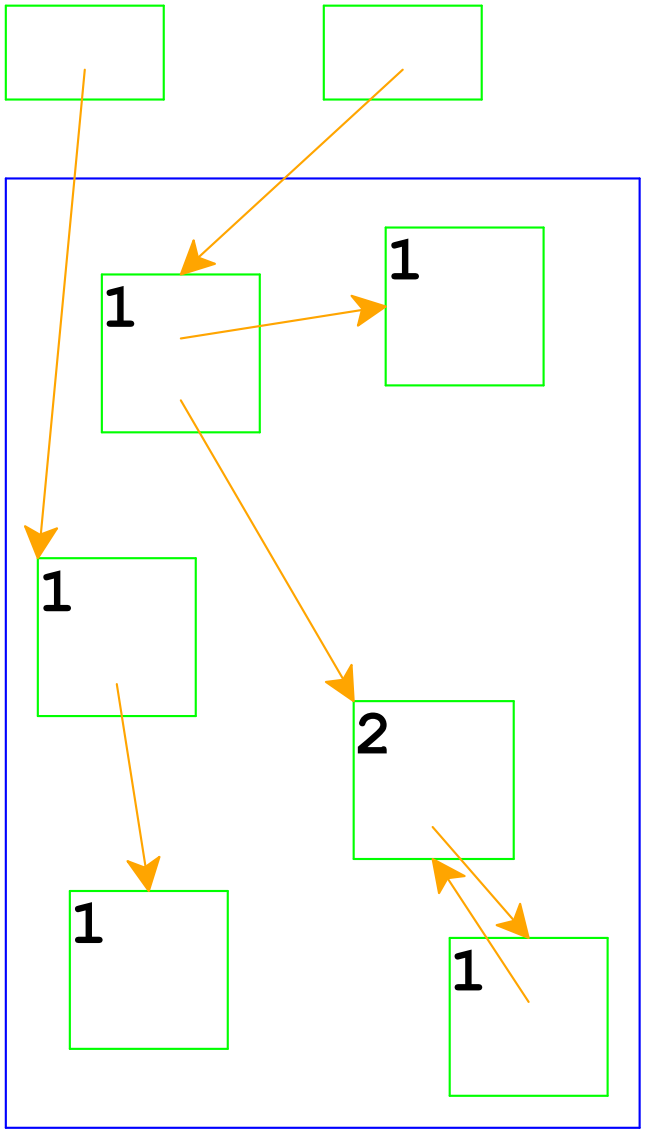
Boxes in the blue area are allocated with **`malloc`**

Reference Counting



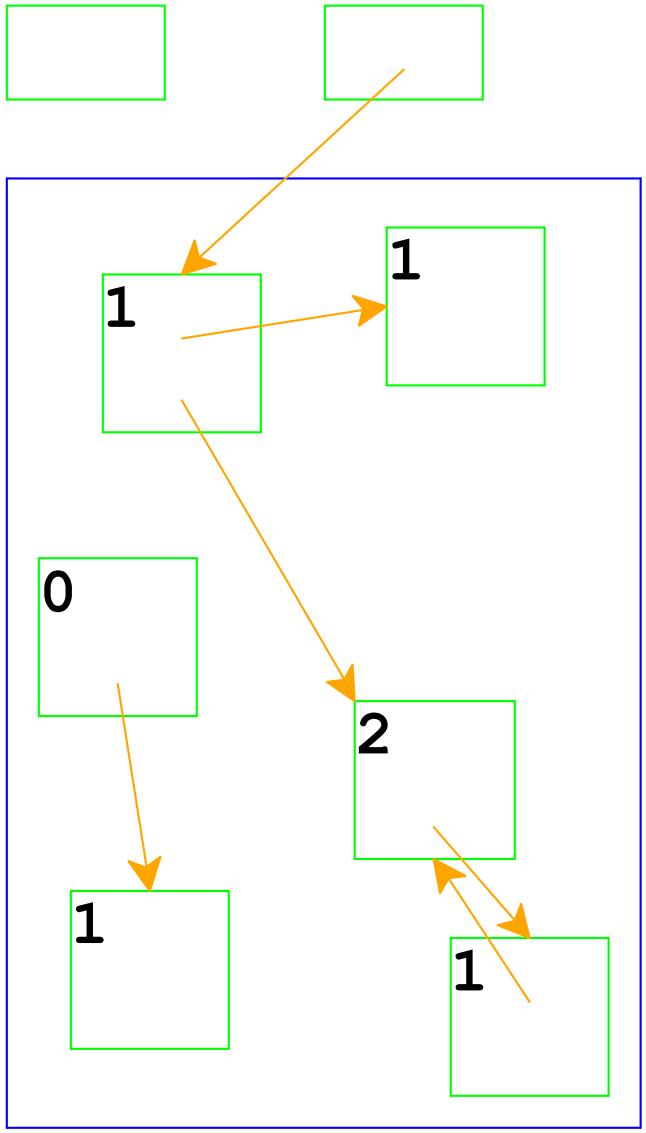
Adjust counts when a pointer is changed...

Reference Counting



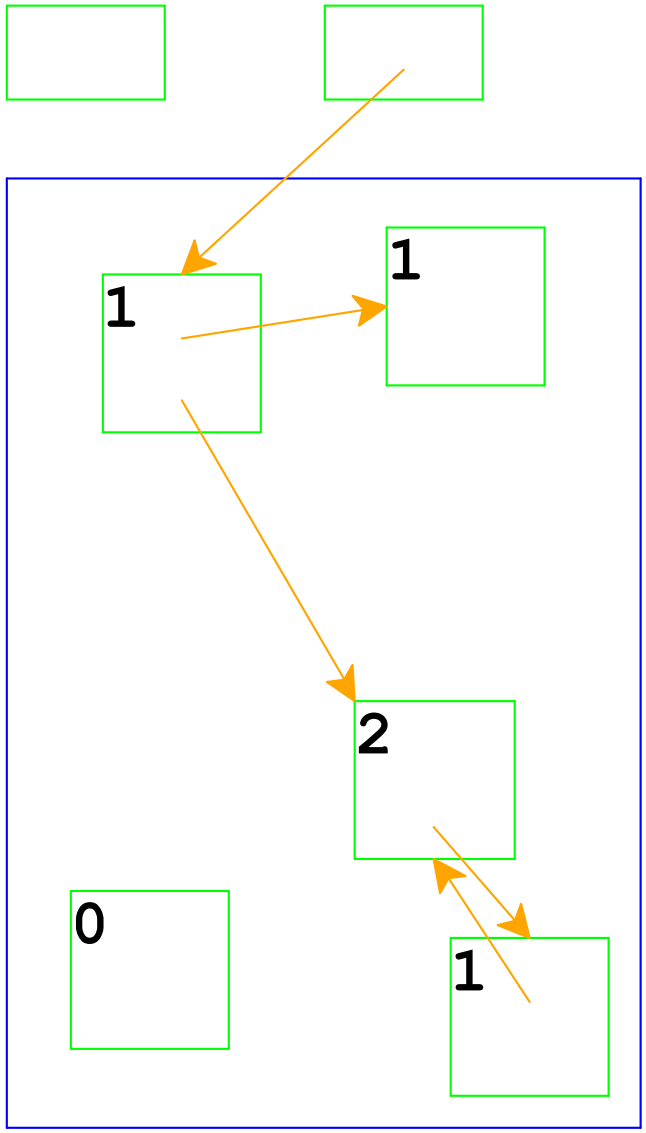
... freeing a record if its count goes to 0

Reference Counting



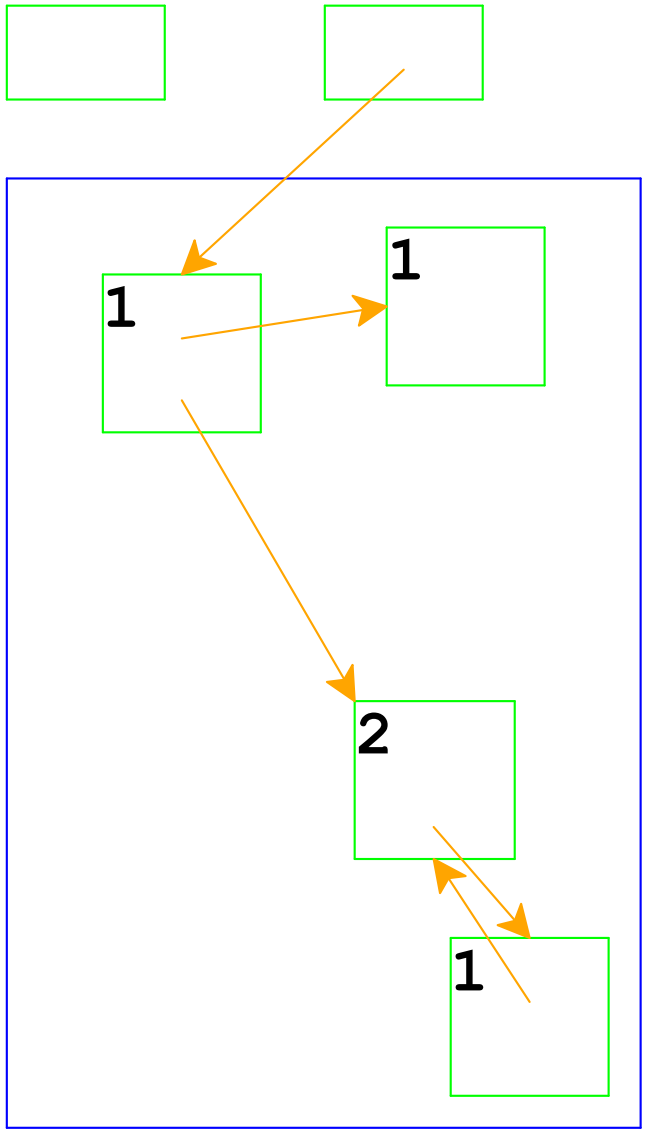
Same if the pointer is in a register

Reference Counting



Adjust counts after frees, too...

Reference Counting



... which can trigger more frees

Using Reference Counting

see **`miniracket5`**

Reference-Counting Issues

- Any missing **`release`** or **`retain`**?
- All **`release`** and **`retain`** ordered correctly?
- Initial reference count?
- Stack overflow via **`release`** cascade?
- Cycles?