Data & Memory Management

CS153: Compilers
Greg Morrisett

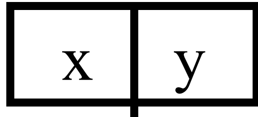
Records in C:

```
struct Point { int x; int y; };

struct Rect { struct Point ll,lr,ul,ur; };

struct Rect mkSquare(struct Point ll, int elen) {
  struct Square res;
  res.lr = res.ul = res.ur = res.ll = ll;
  res.lr.x += elen;
  res.ur.x += elen;
  res.ur.y += elen;
  res.ul.y += elen;
}
```

Representation:

```
struct Point { int x; int y; };
```

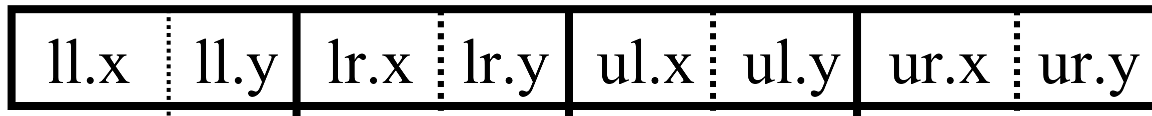
- Two contiguous words. Use base address.



- Alternatively, dedicate two registers?

```
struct Rect { struct Point ll,lr,ul,ur; };
```

- 8 contiguous words.



Member Access

i = rect.ul.y

- Assuming $t holds address of p:
- Calculate offsets of path relative to base:
  - .ul.y = sizeof(struct Point)+sizeof(struct Point) +sizeof(int) = 8 + 8 + 4 = 20
  - So lw $t2, 20($t)

Copy-in/Copy-out

When we do an assignment as in:

```
struct Rect mkSquare(struct Point ll, int elen) {
  struct Square res;
  res.lr = ll;
   ...
```

then we copy all of the elements out of the source and put them in the target. Same as doing word-level opn's:

```
struct Rect mkSquare(struct Point ll, int elen) {
  struct Square res;
  res.lr.x = ll.x;
  res.lr.y = ll.x;
  ...
```

For really large copies, we use something like memcpy.

Procedure Calls:

- Similarly, when we call a procedure, we copy arguments in, and copy results out.
  - Caller sets aside extra space in its frame to store results that are bigger than 2-words.
  - We do the same with scalar values such as integers or doubles.
- Sometimes, this is termed "call-by-value".
  - This is bad terminology.
  - Copy-in/copy-out is more accurate.
- Problem: expensive for large records…

Arrays

```
void foo() {
  char buf[27];

  buf[0] = 'a';
  buf[1] = 'b';
  ...
  buf[25] = 'z';
  buf[26] = 0;
}
```

```
void foo() {
  char buf[27];

  *(buf) = 'a';
  *(buf+1) = 'b';
  ...
  *(buf+25) = 'z';
  *(buf+26) = 0;
}
```

Space is allocated on the stack for buf.

(note, without alloca, need to know size of buf at compile time…)

buf[i] is really just base of array + i * elt_size

Multi-Dimensional Arrays

- In C **`int M[4][3]`** yields an array with 4 rows and 3 columns.
- Laid out in *row-major* order: M[0][0], M[0][1], M[0][2], M[1][0], M[1][1], …
- M[i][j] compiles to?
- In Fortran, arrays are laid out in *column major order*.
- In ML, there are no multi-dimensional arrays -- (int array) array.
- Why is knowing this important?

Strings

- A string constant "foo" is represented as global data:

```
_string42: 102 111 111 0
```

- It's usually placed in the *text* segment so it's read only.
  - allows all copies of the same string to be shared.
- Rookie mistake:

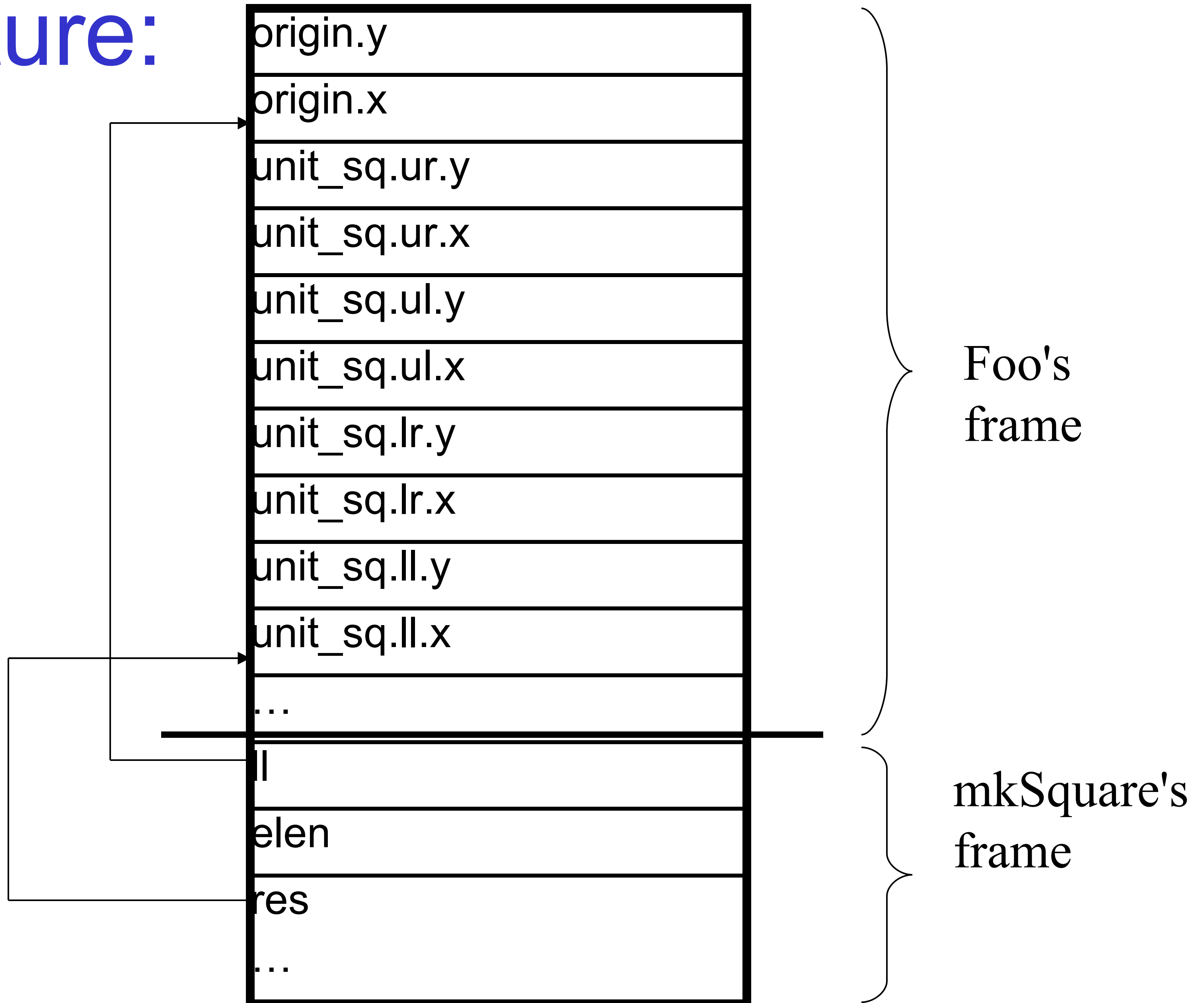
```
char *p = "foo";
p[0] = 'b';
```

Pass-by-Reference:

```
void mkSquare(struct Point *ll, int elen,
              struct Rect *res) {
  res->lr = res->ul = res->ur = res->ll = *ll;
  res->lr.x += elen;
  res->ur.x += elen;
  res->ur.y += elen;
  res->ul.y += elen;
}

void foo() {
  struct Point origin = {0,0};
  struct Square unit_sq;
  mkSquare(&origin, 1, &unit_sq);
}
```

The caller passes in the address of the point and the address of the result (1 word each).

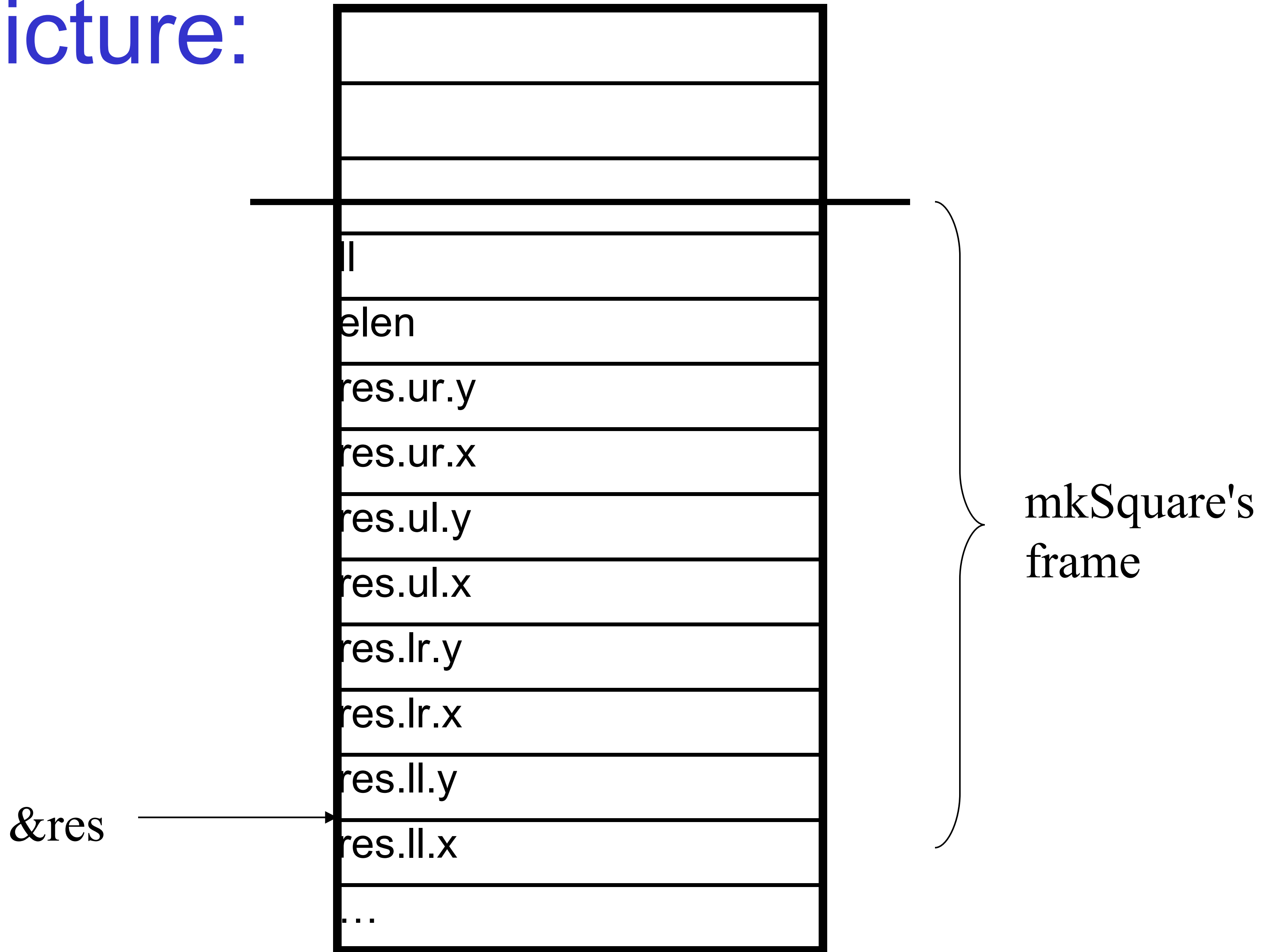
Picture:



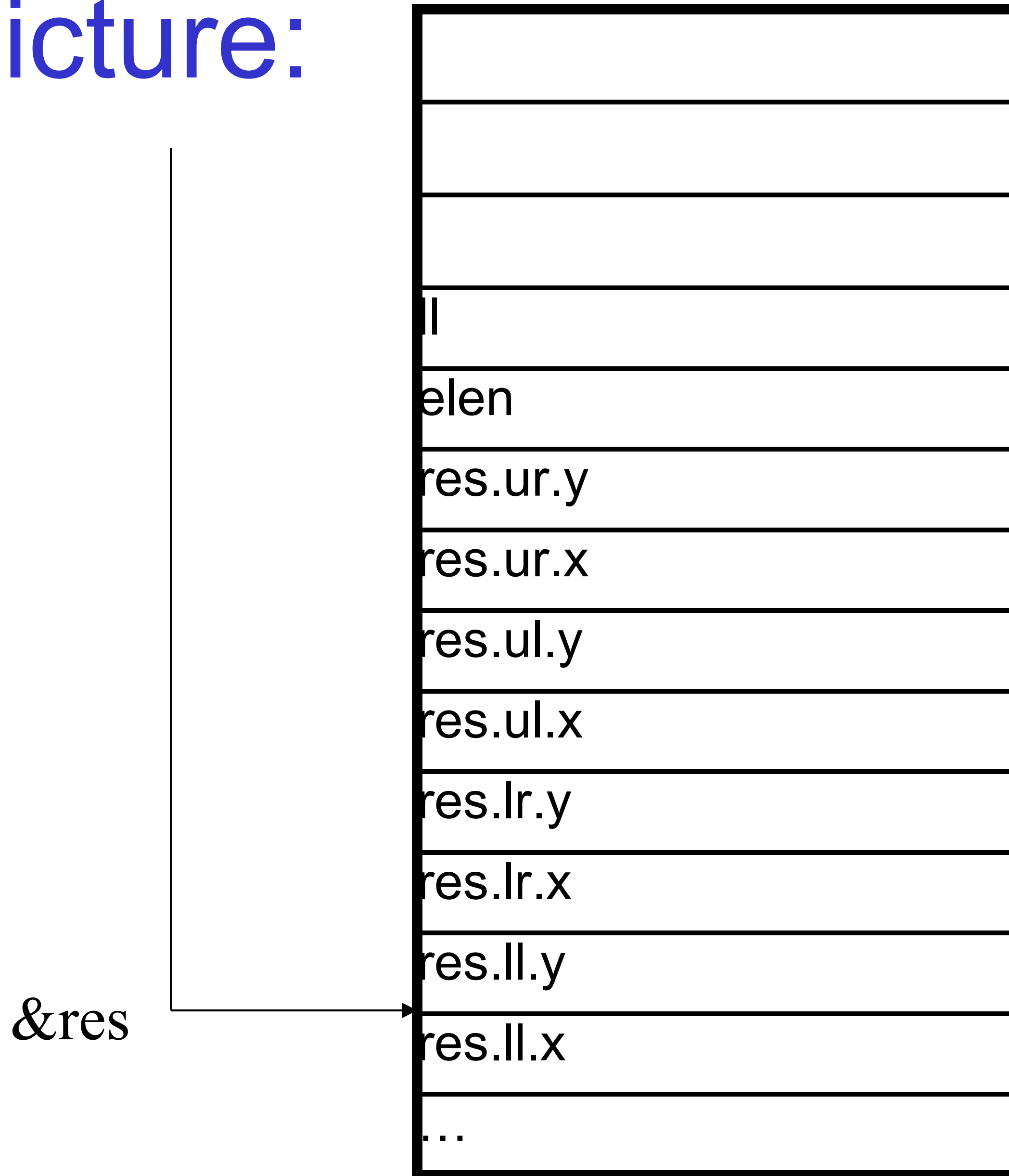
What's wrong with this?

```
struct Rect * mkSquare(struct Point *ll, int elen) {
  struct Rect res;
  res.lr = res.ul = res.ur = res.ll = *ll;
  res.lr.x += elen;
  res.ur.x += elen;
  res.ur.y += elen;
  res.ul.y += elen;
  return &res;
}
```

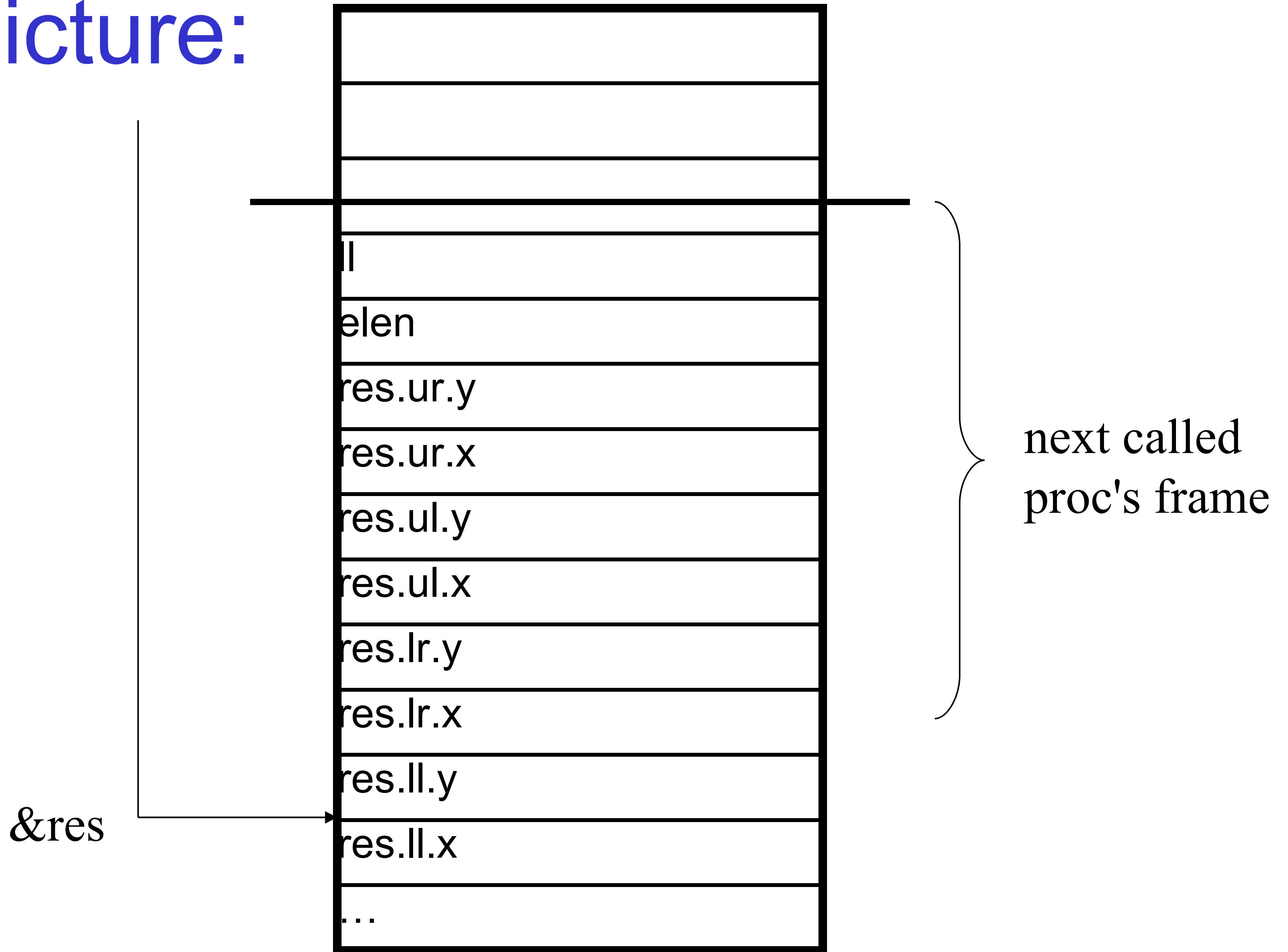
Picture:



Picture:



Picture:



Stack vs. Heap Allocation

- We can only allocate an object on the stack when it is no longer used after the procedure returns.
  - NB: it's possible to exploit bugs like this in C code to hijack the return address. Then an attacker can gain control of the program…
- For other objects, we must use the heap (i.e., malloc).
  - And of course, we must remember to free the object when it is no longer used! Also a big source of bugs in C/C++ code.
  - Java, ML, C#, etc. use a garbage collector instead.

Program Fixed:

```
struct Rect * mkSquare(struct Point *ll, int elen) {
  struct Rect *res = malloc(sizeof(struct Square));
  res->lr = res->ul = res->ur = res->ll = *ll;
  (*res).lr.x += elen;
  res->ur.x += elen;
  res->ur.y += elen;
  (*res).ul.y += elen;
  return res;
}
```

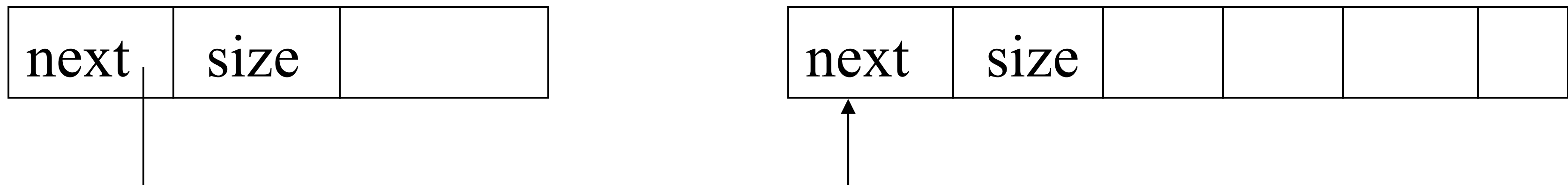
How do malloc/free work?

- Upon malloc(n):
  - Find an unused space of at least size n.
  - (Need to mark space as in use.)
  - Return address of that space.
- Upon free(p):
  - Mark space pointed to by p as free.
  - (Need to keep track of how big object is.)

One Option: Free List

Keep a linked list of contiguous chunks of free memory.

- Each component of list has two words of meta-data.
- 1 word points to the next element in the free list.
- The other word says how big the object is.



Malloc and Free

- To malloc, run down the list until you find a spot that's big enough to satisfy the request.
  - Take left-overs and put them back in the free-list.
  - First-fit vs. Best-fit?
- To free, put the object back in the list.
  - Perhaps keep chunks sorted so that adjacent chunks can be coalesced.
- Pros and Cons?
- What happens if you free something twice or free the middle of an object?

Exponential Scaling:

- Keep an array of free lists:
  - Each list has chunks of the same size.
  - FreeList[i] holds chunks of size 2^i.
  - Round requests up to nearest power of two.
  - When FreeList[i] is empty, take a block from FreeList[i+1] and divide it in half, putting both chunks in FreeList[i].
  - Alternatively, run through FreeList[i-1] and merge contiguous blocks.
- Variations? Issues?

Modern Languages

- Represent all records (tuples, objects, etc.) using pointers.
  - Makes it possible to support *polymorphism*.
  - e.g., ML doesn't care whether we pass an integer, two-tuple, or record to the identity function: they are all represented with 1 word.
  - Price paid: lots of loads/stores…
- By default, allocate records on the heap.
  - Programmer doesn't have to worry about lifetimes.
  - Compiler may determine that it's safe to allocate a record on the stack instead.
  - Uses a garbage collector to safely reclaim data.
  - Because pointers are *abstract*, has the freedom to re-arrange the data in the heap to support compaction.

Allocation in SML/NJ

- Reserve two registers:
  - allocation pointer (like stack pointer)
  - limit pointer
- To allocate a record of size n:
  - checks that limit-alloc > n. If not, invokes garbage collector.
  - Adds n+1 to the alloc pointer, returns old value of alloc pointer as result.
  - Extra word holds meta-data (e.g., size.)
  - Actually, amortizes the limit check across a bunch of allocations (just as we amortize stack pointer adjustment.)
  - Result: 3-5 instructions to allocate a record.

Garbage Collection:

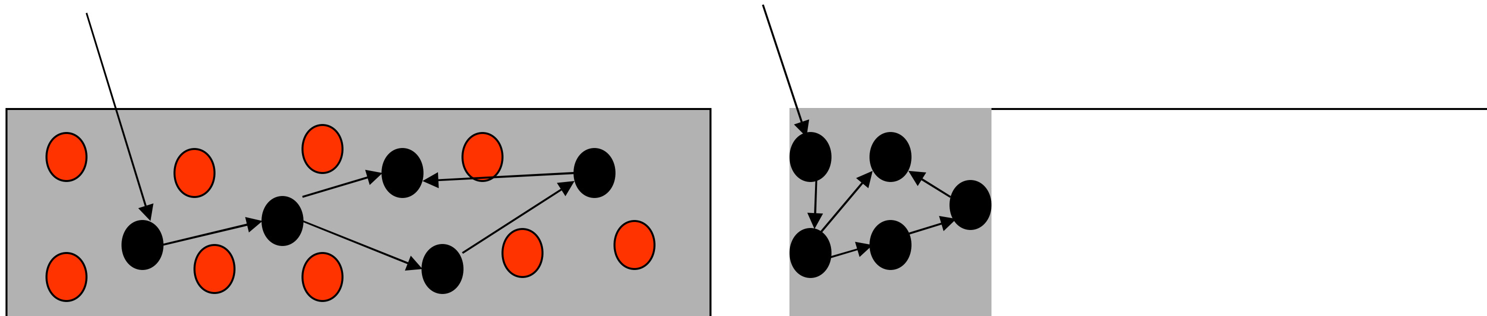
- Starting from stack, registers, & globals (roots), determine which objects in the heap are reachable following pointers.
- Reclaim any object that isn't reachable.
- Requires being able to determine pointer values from other values (e.g., ints).
  - SML/NJ uses the low bit:
    1 it's a scalar, 0 it's a pointer.
  - In Java, we use put the tag bits in the meta-data.
  - For BDW collector, we use heuristics:
    (e.g., the value doesn't point into an allocated object.)

Mark/Sweep Traversal:

- Reserve a mark-bit for each object.
- Starting from roots, mark all accessible objects.
- Stick accessible objects into a queue or stack.
  - queue: breadth-first traversal
  - stack: depth-first traversal
- Loop until queue/stack is empty:
  - remove marked object (say x).
  - if x points to an (unmarked) object y, then mark y and put it in the queue.
- Run through all objects:
  - If they haven't been marked, put them on the free list.
  - If they have been marked, clear the mark bit.

Copying Collection:

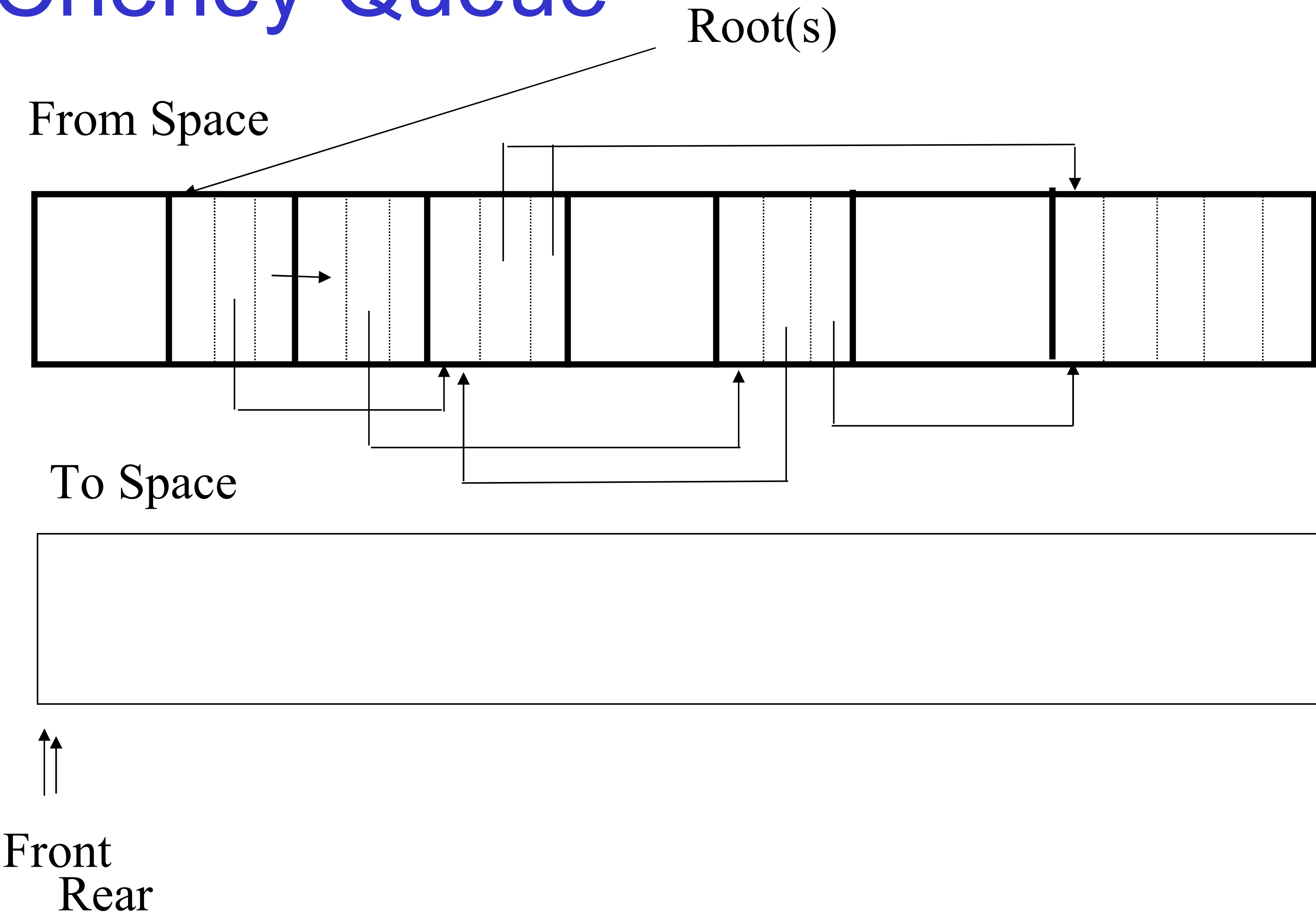
- Split data segment into two pieces.
- Allocate in 1st piece until it fills up.
- Copy the reachable data into the 2nd area, compressing out the holes corresponding to garbage objects.



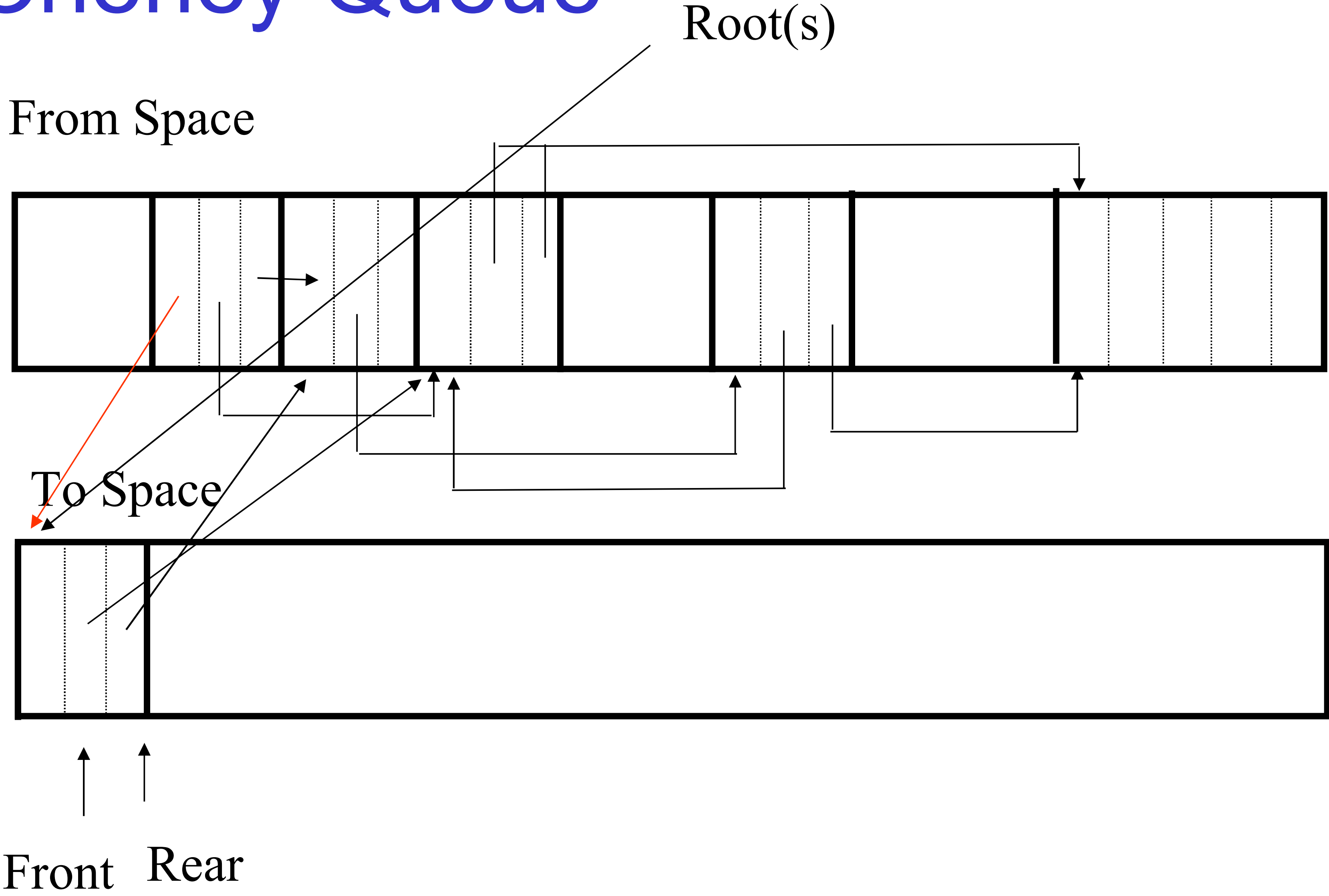
Algorithm: Queue-Based

- Initialize front/rear to beginning of to-space.
  - A trick for representing the queue using the to-space.
- Enqueue the items pointed to by roots.
  - Copy the objects into to-space (bump rear pointer).
  - Place a *forwarding pointer* in the old copy that points to the new copy.
- While queue is not empty:
  - Dequeue a word (i.e., bump front pointer).
  - If the word is a pointer to an unforwarded object, then enqueue the object and set its forwarding pointer.
  - If the word is a pointer to a forwarded object, overwrite the word with the address of the new copy.

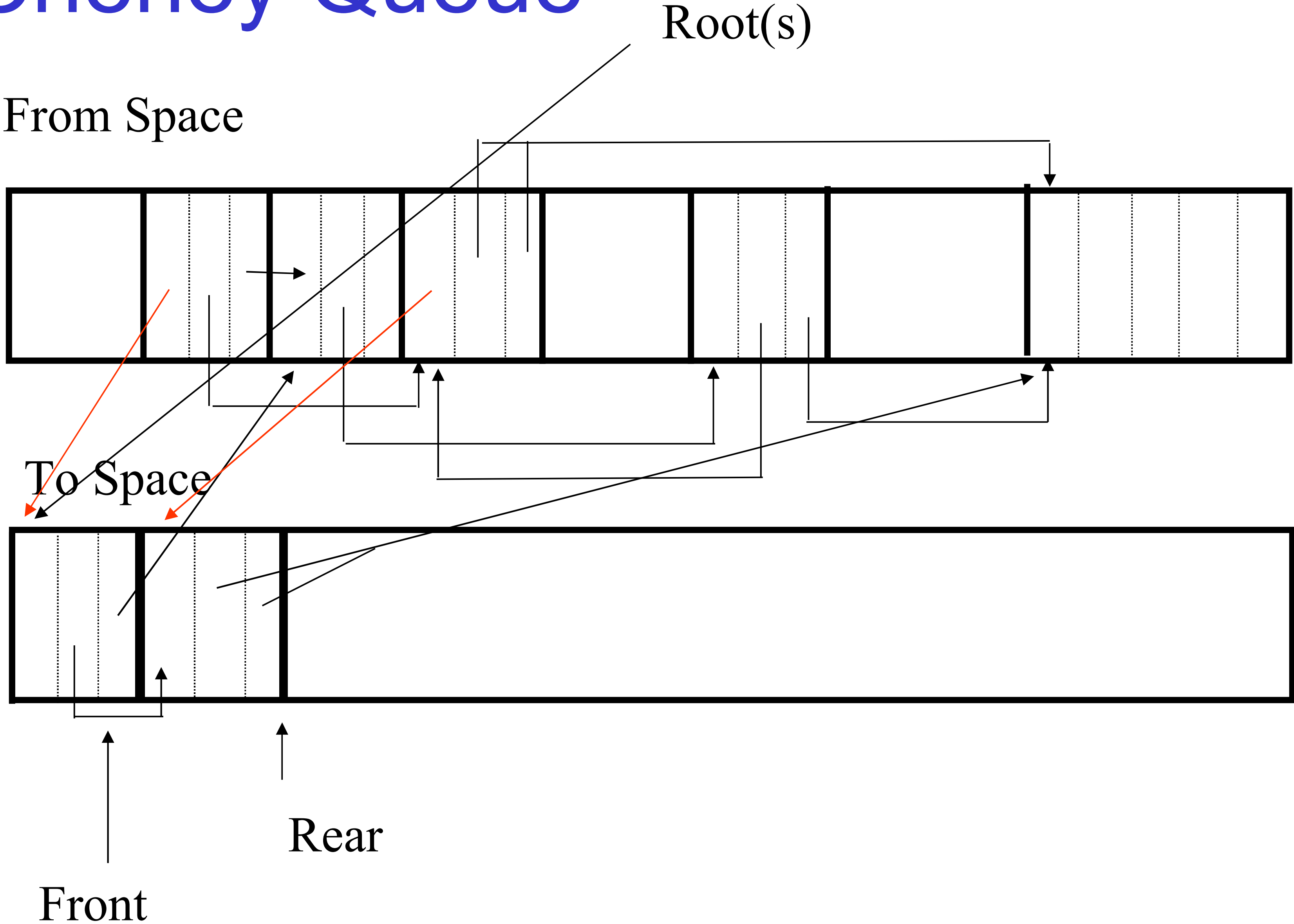
Cheney Queue



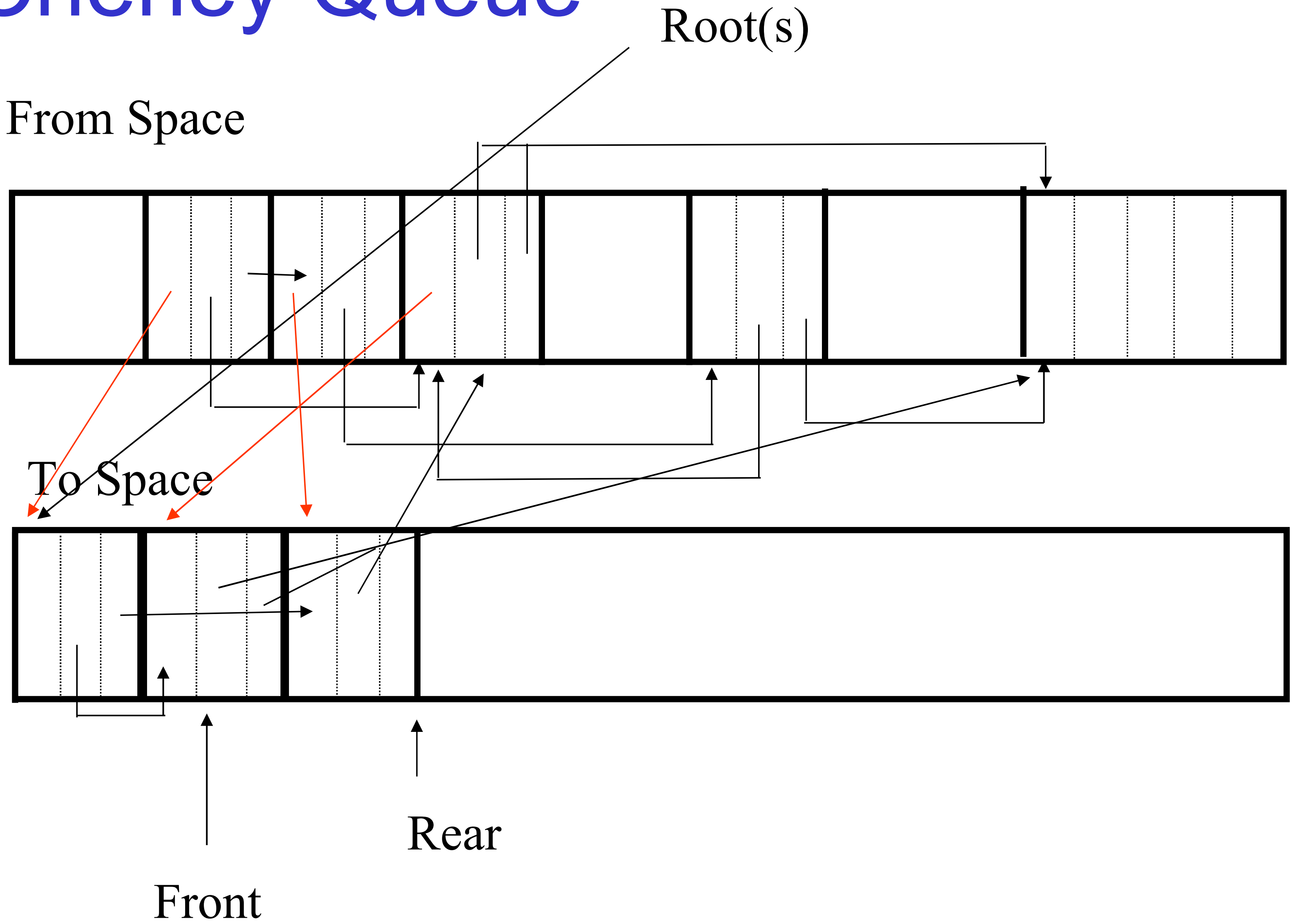
Cheney Queue



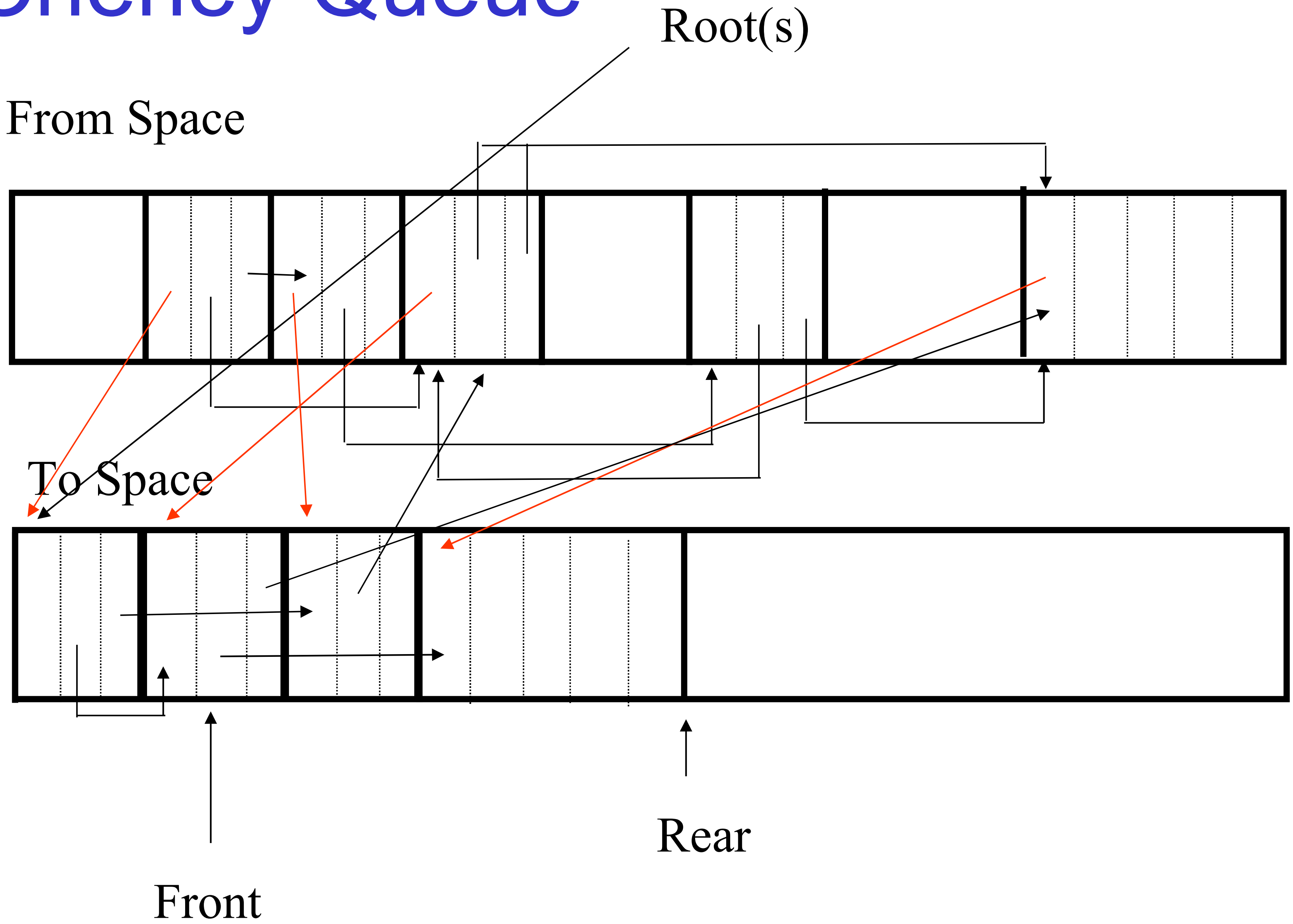
Cheney Queue



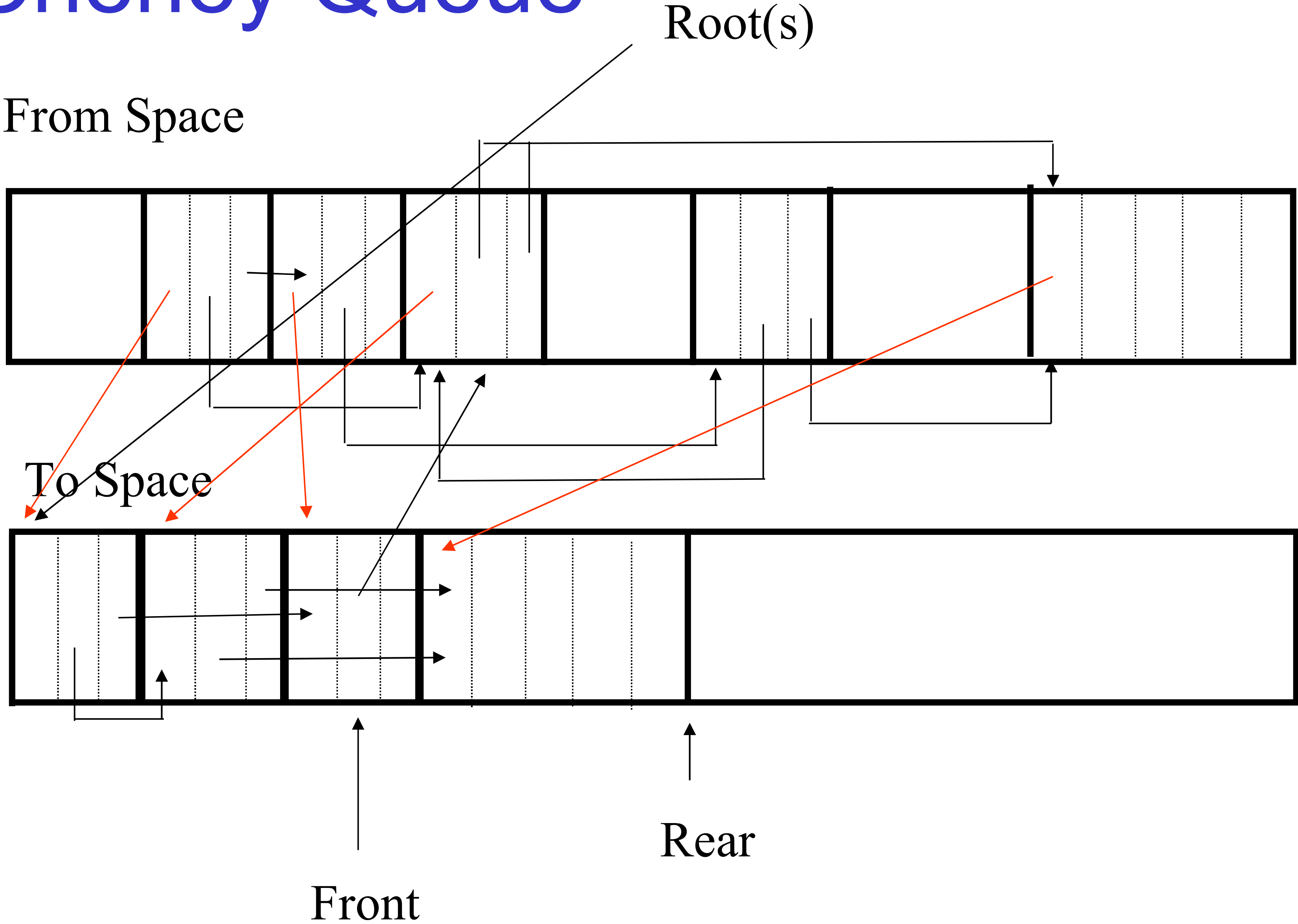
Cheney Queue



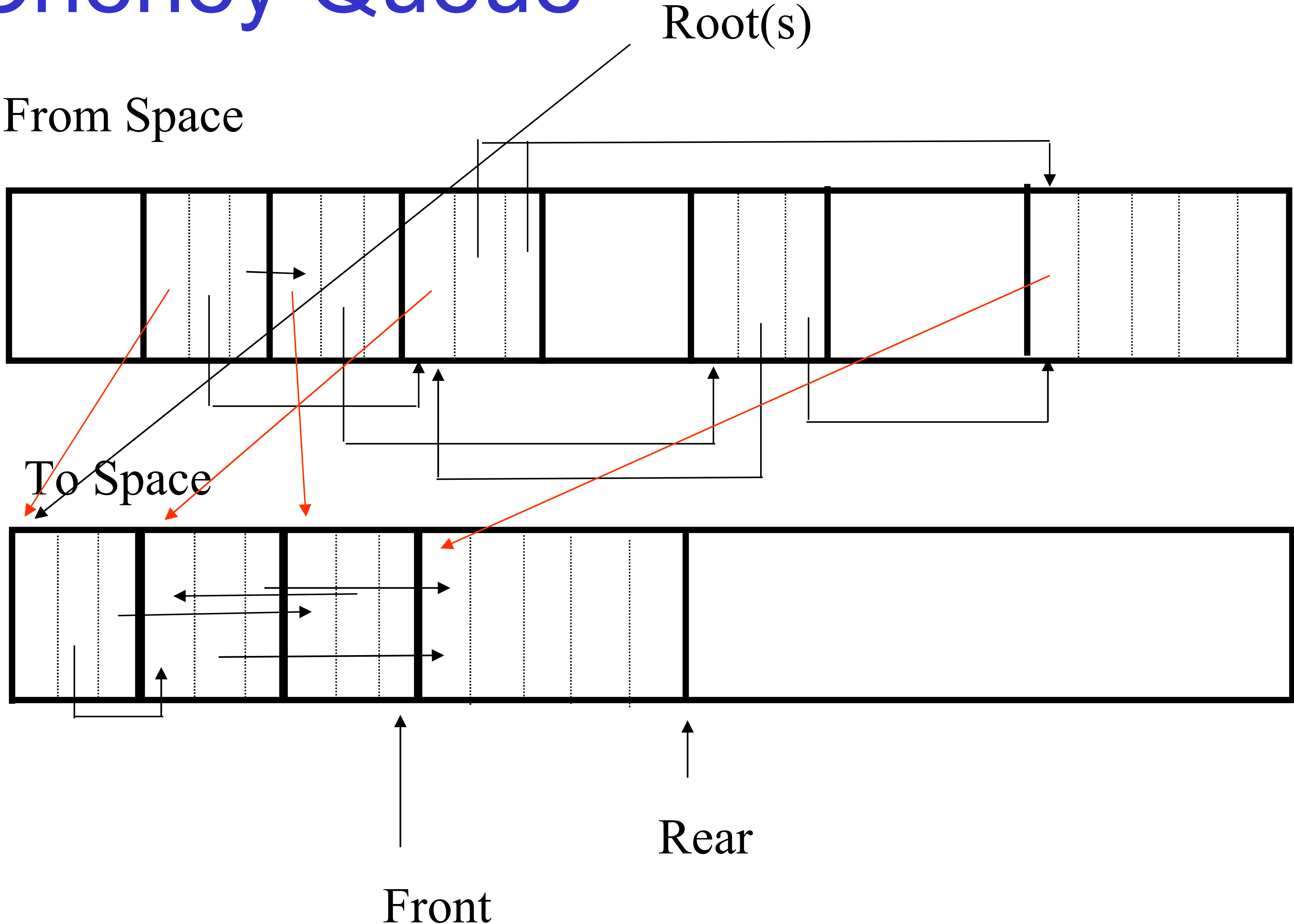
Cheney Queue



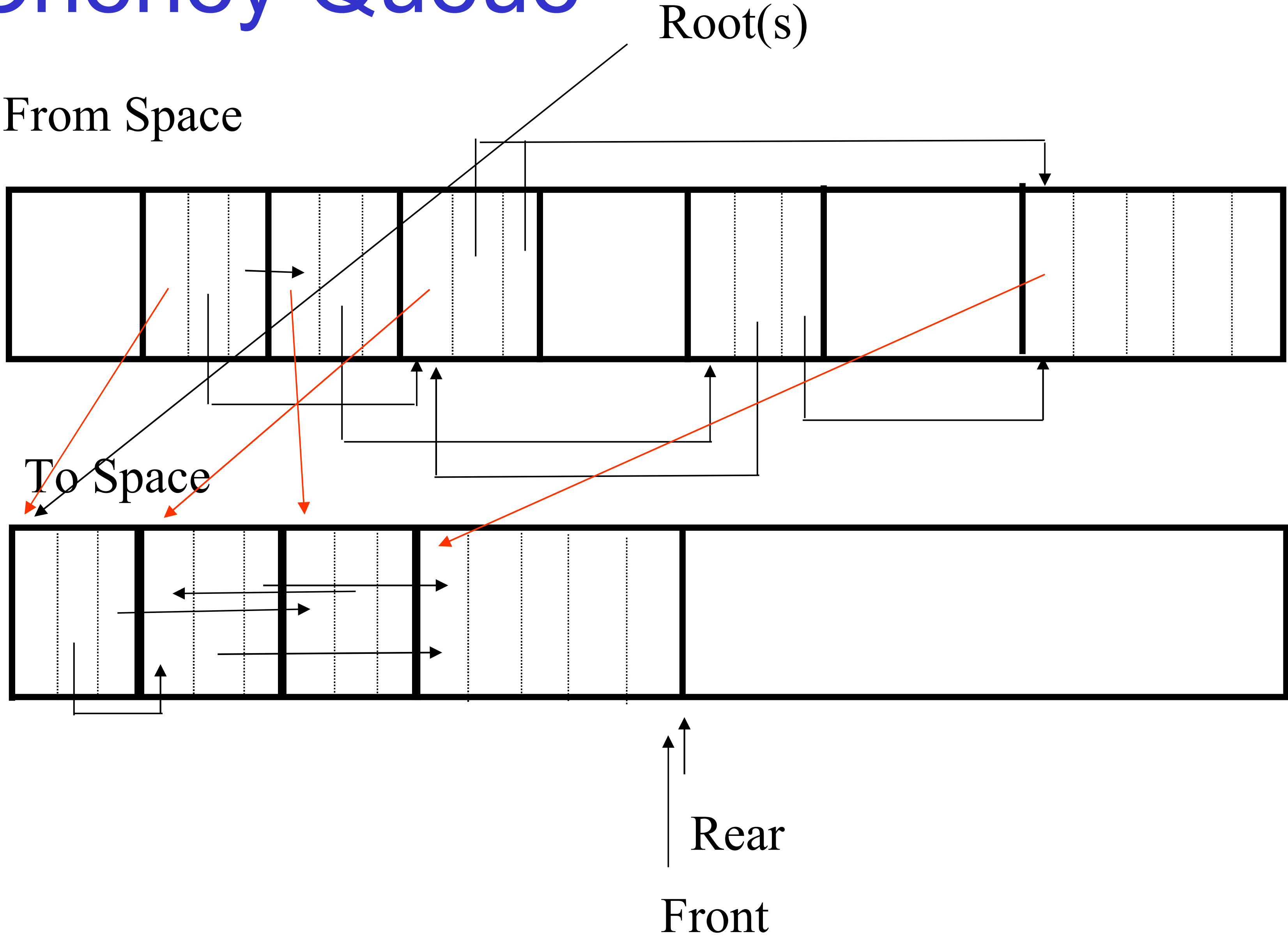
Cheney Queue



Cheney Queue



Cheney Queue

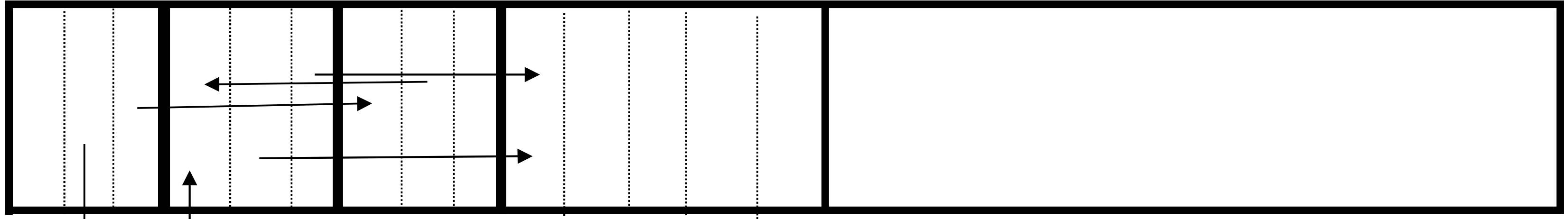


Cheney Queue

Root(s)

From Space

To Space



Rear

Front

Pros and Cons:

- Pros:
  - Fast, bump-pointer allocation.
  - Cost of GC is proportional to live data (not all of memory).
  - Compaction happens for free.
- Cons:
  - Long pauses.
  - Memory cut in half.
  - Lots of memory traffic.

Reality:

- Techniques such as *generational* or *incremental* collection can greatly reduce latency.
  - A few millisecond pause times.
- Large objects (e.g., arrays) can be copied in a "virtual" fashion without doing a physical copy.
- Some systems use a mix of copying collection (young data) and mark/sweep (old data) with support for compaction.
- A real challenge is scaling this to server-scale systems with terabytes of memory…
- Interactions with OS matter a lot: cheaper to do GC than it is to start paging…

Conservative Collectors:

- Work without help from the compiler.
  - e.g., legacy C/C++ code.
  - e.g., your compiler :-)
- Cannot accurately determine which values are pointers.
  - But can rule out some values (e.g., if they don't point into the data segment.)
  - So they must conservatively treat anything that looks like a pointer as such.
  - Two bad things result: leaks, can't move.
  - Further problems if pointers are "hidden".

The BDW Collector

- Based on mark/sweep.
  – performs sweep lazily
- Organizes free lists as we saw earlier.
  – different lists for different sized objects.
  – relatively fast (single-threaded) allocation.
- Most of the cleverness is in finding roots:
  – global variables, stack, registers, etc.
- And determining values aren't pointers:
  – blacklisting, etc.