Memory Management

Variable Storage

- Storage binding - binds the address attribute of a variable to physical storage
  - Disregarding object attributes (fields) for now
- Allocation of space
  - **Static (compile-time or load time)**
  - **Stack (runtime) – aka user/runtime/system stack**
  - Heap (runtime)

Variable Storage and Lifetime

- Time a variable is bound to a particular memory location
- 3 categories of primitive variables (given different lifetimes)
  - Globals (static storage)
    - Variables declared outside of any function or class (outermost scope)
    - Scope: accessible to all statements in all functions in the file
    - Lifetime: from start of program (loading) to end (unloading)
    - Good practice: use sparingly, make constant as often as possible
    - Stored in read-only or read-write segments of the process virtual memory space – allocated/fixed before program starts
      - Read-only segment holds translated/native code as well if any

Variable Storage and Lifetime

- Time a variable is bound to a particular memory location
- 3 categories of primitive variables (given different lifetimes)
  - Globals (static storage)
    - Variables declared outside of any function or class (outermost scope)
    - Scope: accessible to all statements in all functions in the file
    - Lifetime: from start of program (loading) to end (unloading)
    - Good practice: use sparingly, make constant as often as possible
    - Stored in read-only or read-write segments of the process virtual memory space – allocated/fixed before program starts
      - Read-only segment holds translated/native code as well if any
  - Locals (stack storage)
    - **Parameters and variables** declared within a function
    - Scope: accessible to all statements in the function they are defined
    - Lifetime: from start to end of the function invocation
    - Stored in User/Runtime stack in process virtual memory space
      - Allocated/deallocated with function invocations and returns

Variable Storage and Lifetime

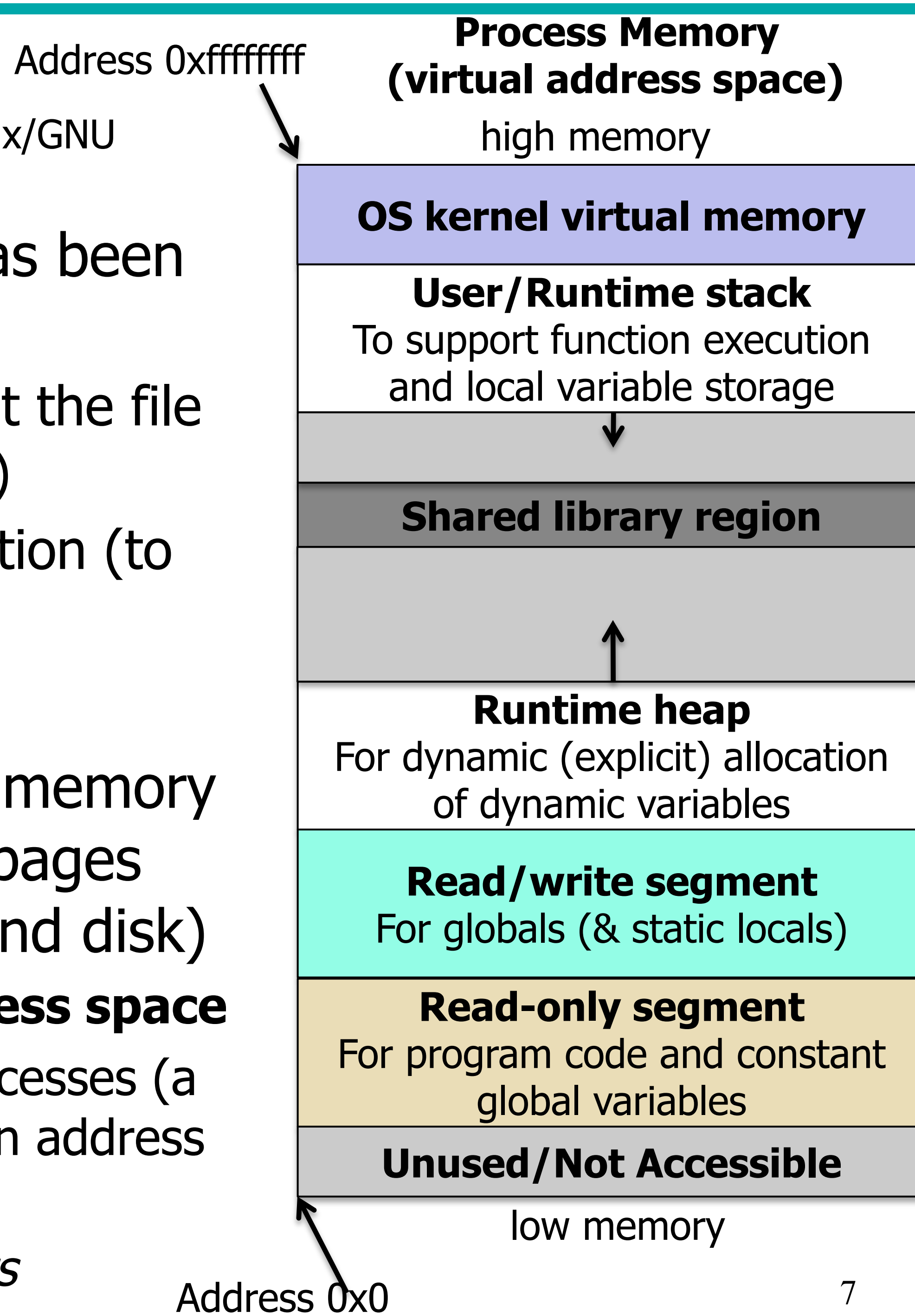
- Time a variable is bound to a particular memory location
- 3 categories of primitive variables (given different lifetimes)
  - Globals (static storage)
  - Locals (stack storage)
  - Dynamic variables, aka pointer variables (heap storage)
    - Pointer variables that point to variables that are allocated ***explicitly***
    - Scope: global or local depending on where they are declared
    - Lifetime: from program point at which they are allocated with **new** to the one at which they are deallocated with **delete**

    - Pointer variables (the address) are either globals or locals
    - The data they point to is stored in the **heap** segment of the process' virtual memory space

An OS Process

Executable & Linkable Format (ELF): Linux/GNU

- An executeable file that has been loaded into memory
  - The OS has been told that the file is ready (**exec** command)
  - OS schedules it for execution (to get a turn using the CPU)
- Since we are using virtual memory (paging physical memory pages between virtual memory and disk)
  - A process has its own **address space**
    - Provides isolation of processes (a process cannot access an address in another process)
    - Broken up into *segments*



Heap Allocation and Deallocation

- Explicit allocation and deletion
  - New, malloc, delete, free
  - Programmer controls all
    - Delete an object following the last use of it
- Implicit
  - Programmers do nothing, its all automatic
  - Non-heap objects are implictly allocated and deallocated
    - Local variables, deallocated with simple SP re-assignment
    - Globals, never deallocated, cleaned up with program at end
  - Implicit deallocation of heap objects
    - Garbage collection
    - May **not** remove an object from system immediately after its last use
    - Stack variables (locals and params), static variables (globals) use implicit allocation and deallocation

Failures in Explicitly Deallocated Memory

- Memory leaks
- Dangling pointers
- Out of memory errors
- Errors may not be repeatable (system dependent)
- Dynamic memory management in complex programs is very difficult to implement **correctly**
  - Even for simple data structures
- Multi-threading/multi-processing complicates matters

- Debugging is very difficult (requires other tools)
  - Purify
  - But these only work on a running program (particular input and set of paths taken are the only ones checked)

Garbage Collection

- Solves explicit deallocation problems through automation
- Introduces runtime processing (overhead) to do the work
- Not the solution to every problem in any language
  - However it is REQUIRED for managed languages
    - For which programs can be **sandboxed** to protect the host system

- But it will
  - Reduce the number of bugs and hard to find programming errors
  - Reduce program development/debugging cycle
- However, it should be an integrated part of the system
  - Not an afterthought or hack

- May even improve performance! … How?

Terminology

- Collector
  - Part of the runtime that implements memory management
- Mutator
  - User program - change (mutate) program data structures
- Stop-the-world collector - all mutators stop during GC
- Values that a program can manipulate directly
  - In processor registers
  - On the program stack (includes locals/temporaries)
  - In global variables (e.g., array of statics)
- **Root set** of the computation
  - References to heap data held in these locations
  - Dynamically allocated data **only** accessible via roots
  - A program should not access random locations in heap

Roots, Liveness, and Reachability

- Individually allocated pieces of data in the heap are
  - Nodes, cells, objects (interchangeably)
  - Commonly have header that indicates the type (and thus can be used to identify any references within the object)
    - AKA boxed

- Live objects in the heap
  - **Graph of objects that can be "reached" from roots**
    - Objects that cannot be reached are garbage
      - For languages without GC, what is this called?

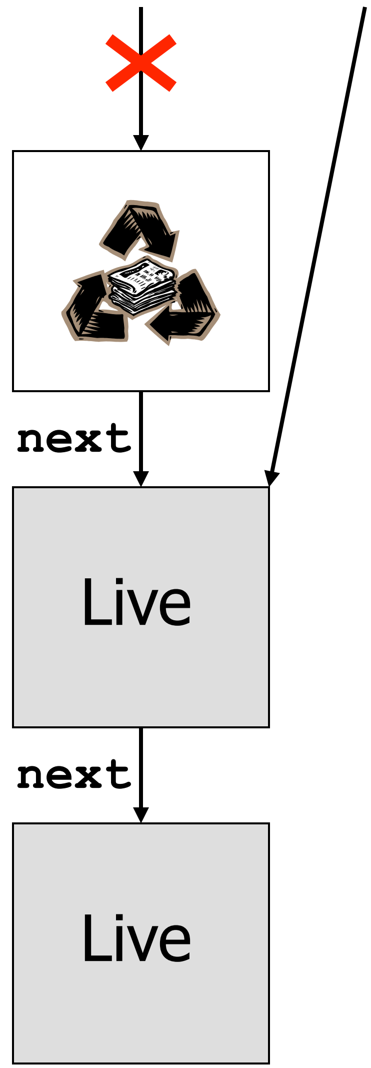
  - An object in the heap is **live** if
    - Its address is held in a root, or
    - There is a pointer to it held in another live heap object

GC Example

mutator

```
static MyList listEle;
void foo() {
listEle = new MyList();
listEle.next = new MyList();
listEle.next.next = new
MyList();
MyList localEle =
listEle.next;

listEle = null;
Object o = new Object();
```

Root Set: statics, stack vars, registers



GC Cycle

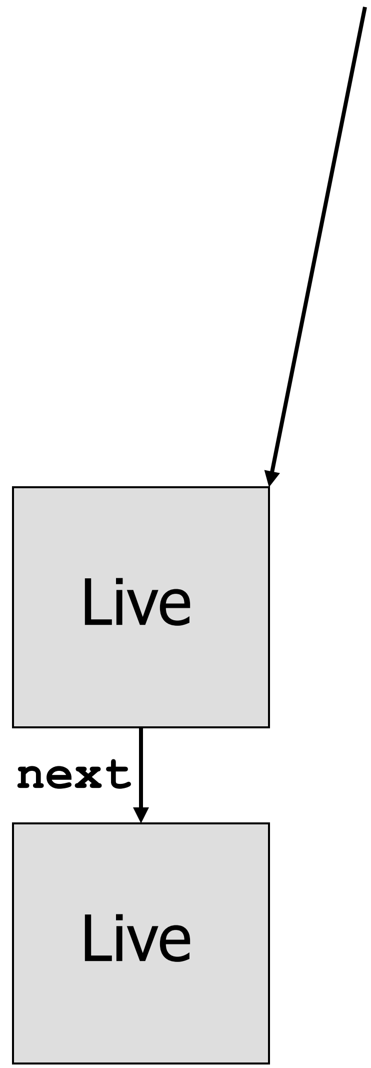
1. Detection
2. Reclamation

GC Example

mutator

```
static MyList listEle;
void foo() {
listEle = new MyList();
listEle.next = new MyList();
listEle.next.next = new
MyList();
MyList localEle =
listEle.next;


listEle = null;

Object o = new Object();
```

Root Set: statics, stack vars, registers



GC Cycle

1. Detection
2. Reclamation

Restart mutators

Liveness of Allocated Objects

- Determined indirectly **or** directly
- Indirectly
  - Most common method: **tracing**
  - Regenerate the set of live nodes whenever a request by the user program for more memory fails
  - Start from each root and visit all reachable nodes (via pointers)
  - Any node not visited is reclaimed

Liveness of Allocated Objects

- Determined indirectly **or** directly
- Directly
  - A record is associated with each node in the heap and all references to that node from other heap nodes or roots
  - Most common method: **reference counting**
    - Store a count of the number of pointers to this cell in the cell itself
  - Alternate example: Distributed systems where processors share memory
    - Keep a list of the processors that contain references to each object
  - Must be kept up to date as the mutator alters the connectivity of the heap graph

Today's Paper

- GC required for *truly modular* programming/programs
- Liveness -- an object that is live (global property)
- When does GC occur?
  - Incremental garbage collection
  - Stop the world
- What are the two abstract phases of GC?
- Memory leak/dangling pointer -- how does GC avoid them?
- Tracing versus reference counting
  - Root set for tracing
    - Mark-sweep versus copying
  - Limitations of reference counting

Terminology

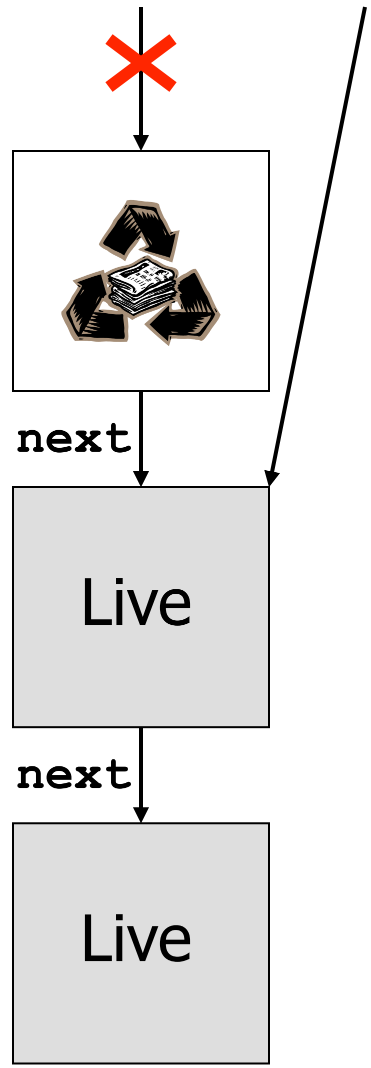
- Collector
  - Memory manager – should not allocate memory!
- Mutator
  - User program - change (mutate) program data structures
- Stop-the-world collector - all mutators stop during GC
- Values that a program can manipulate directly
  - In processor registers, on program stack (includes locals/ temporaries), globals (e.g., data in statics table)
- **Root set** of the computation
  - References to heap data held in these locations
  - Dynamically allocated data **only** accessible via roots
    - A program should not access random locations in heap
  - "**Live**" objects are those reachable by the roots (all else is garbage)

GC Example

mutator

```
static MyList listEle;

void foo() {

listEle = new MyList();

listEle.next = new MyList();

listEle.next.next = new
MyList();

MyList localEle =
listEle.next;

listEle = null;

Object o = new Object();
```

Root Set: statics, stack vars, registers



GC Cycle

1. Detection

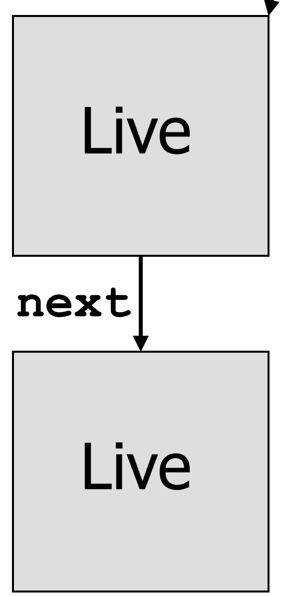
2. Reclamation

GC Example

mutator

```
static MyList listEle;
void foo() {
listEle = new MyList();
listEle.next = new MyList();
listEle.next.next = new
MyList();
MyList localEle =
listEle.next;


listEle = null;

Object o = new Object();
```

Root Set: statics, stack vars, registers



GC Cycle

1. Detection

2. Reclamation

Restart mutators

Liveness of Allocated Objects

- Determined indirectly **or** directly
- Indirectly
  - Most common method: **tracing**
  - Regenerate the set of live nodes whenever a request by the user program for more memory fails
  - Start from each root and visit all reachable nodes (via pointers)
  - Any node not visited is reclaimed

Liveness of Allocated Objects

- Determined indirectly **or** directly
- Directly
  - A record is associated with each node in the heap and all references to that node from other heap nodes or roots
  - Most common method: **reference counting**
    - Store a count of the number of pointers to this cell in the cell itself
  - Alternate example: Distributed systems where processors share memory
    - Keep a list of the processors that contain references to each object
  - Must be kept up to date as the mutator alters the connectivity of the heap graph

Three Classic Garbage Collection Algorithms

- Reference counting
- Mark & Sweep
- Copying

Three Classic Garbage Collection Algorithms

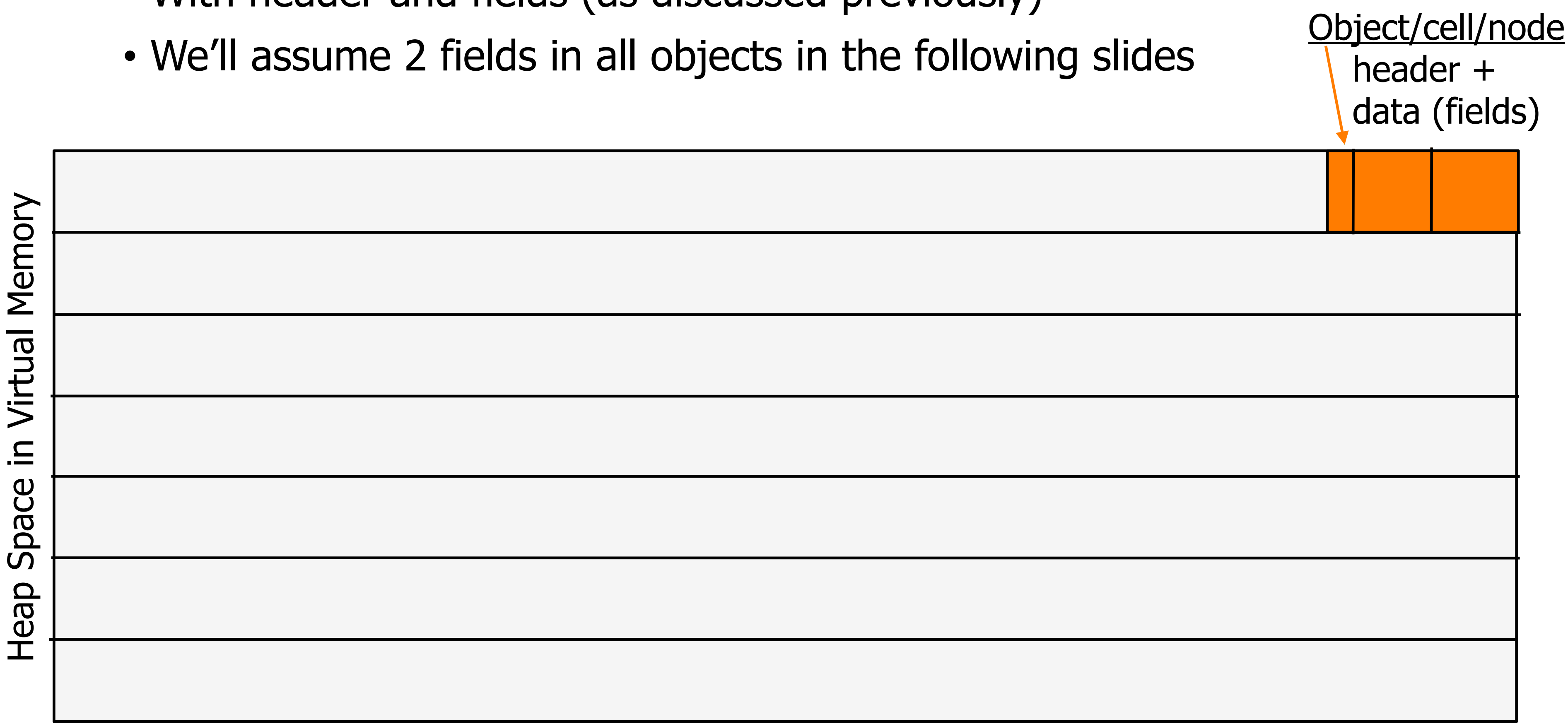
- Reference counting
- Mark & Sweep
- Copying

Free List Allocation: keep 1+ lists of free chunks that we then fill or break off pieces of to allocate an object

The Free List: Internal VM/runtime data structure – linked list of free blocks

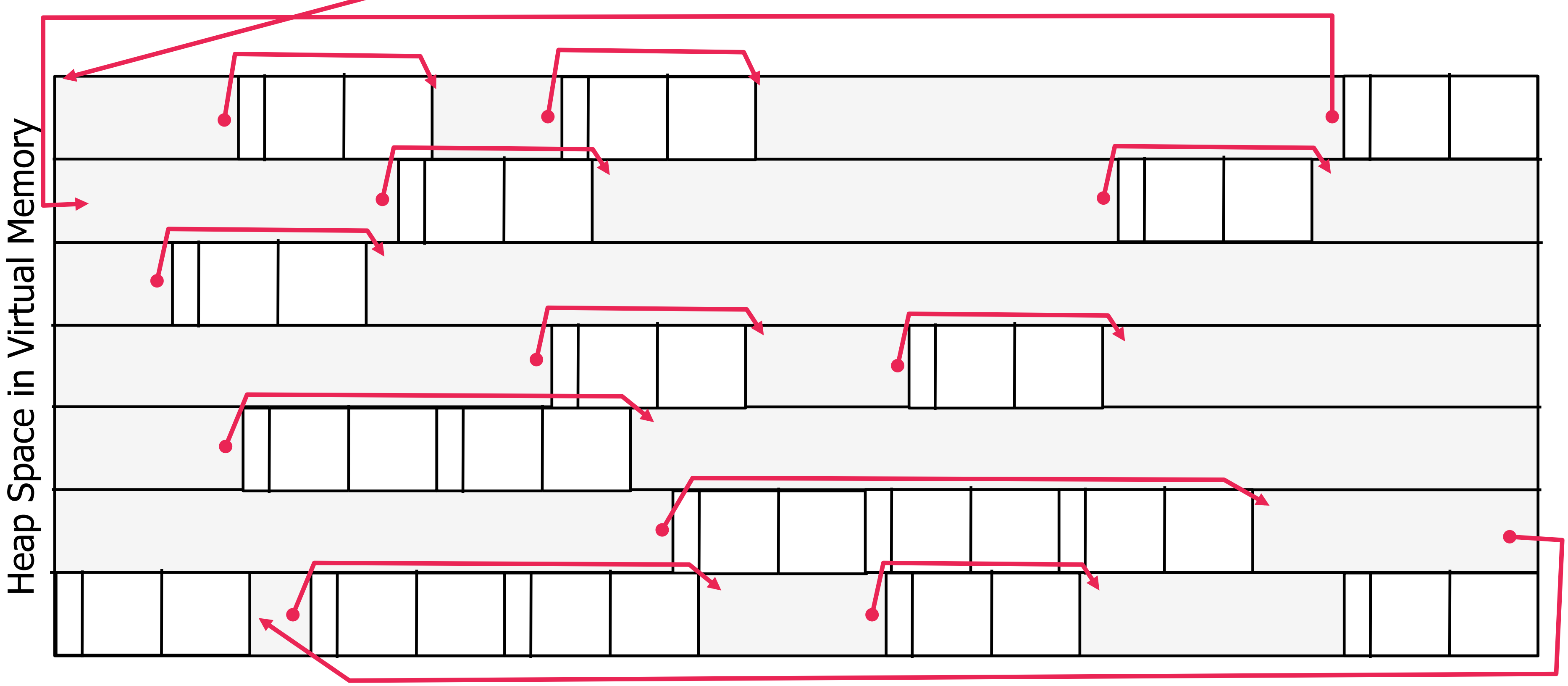
- Memory is **one big contiguous array**
  - In the virtual address space of the processor: Heap area
- Typically word-aligned addresses

- Objects = data allocated in memory
  - With header and fields (as discussed previously)
  - We'll assume 2 fields in all objects in the following slides



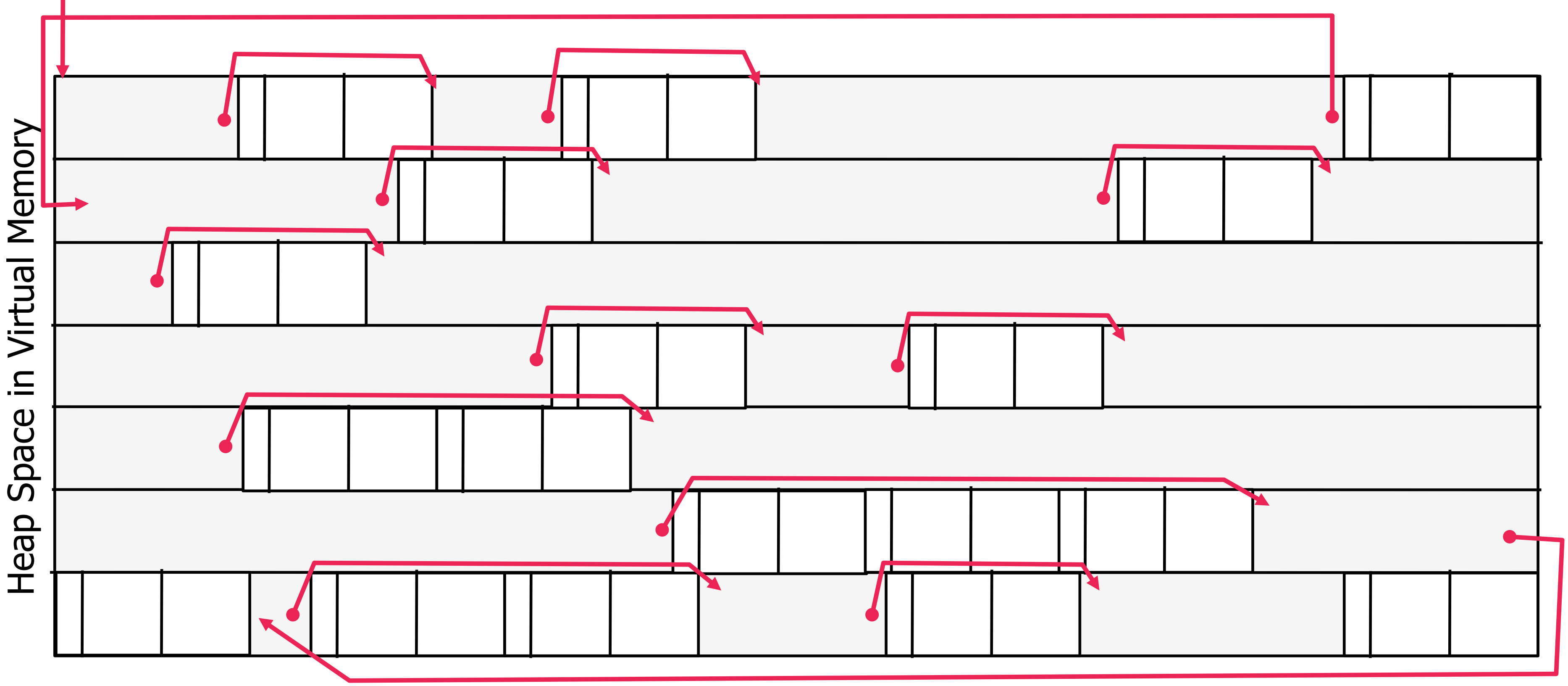
The Free List: Internal VM/runtime data structure – linked list of free blocks

- Memory (virtual/heap) is one big contiguous array
- Typically multiple lists (each with different sized blocks – e.g powers of 2)
  - Linked together in a linked list (hidden next pointer + list_head)

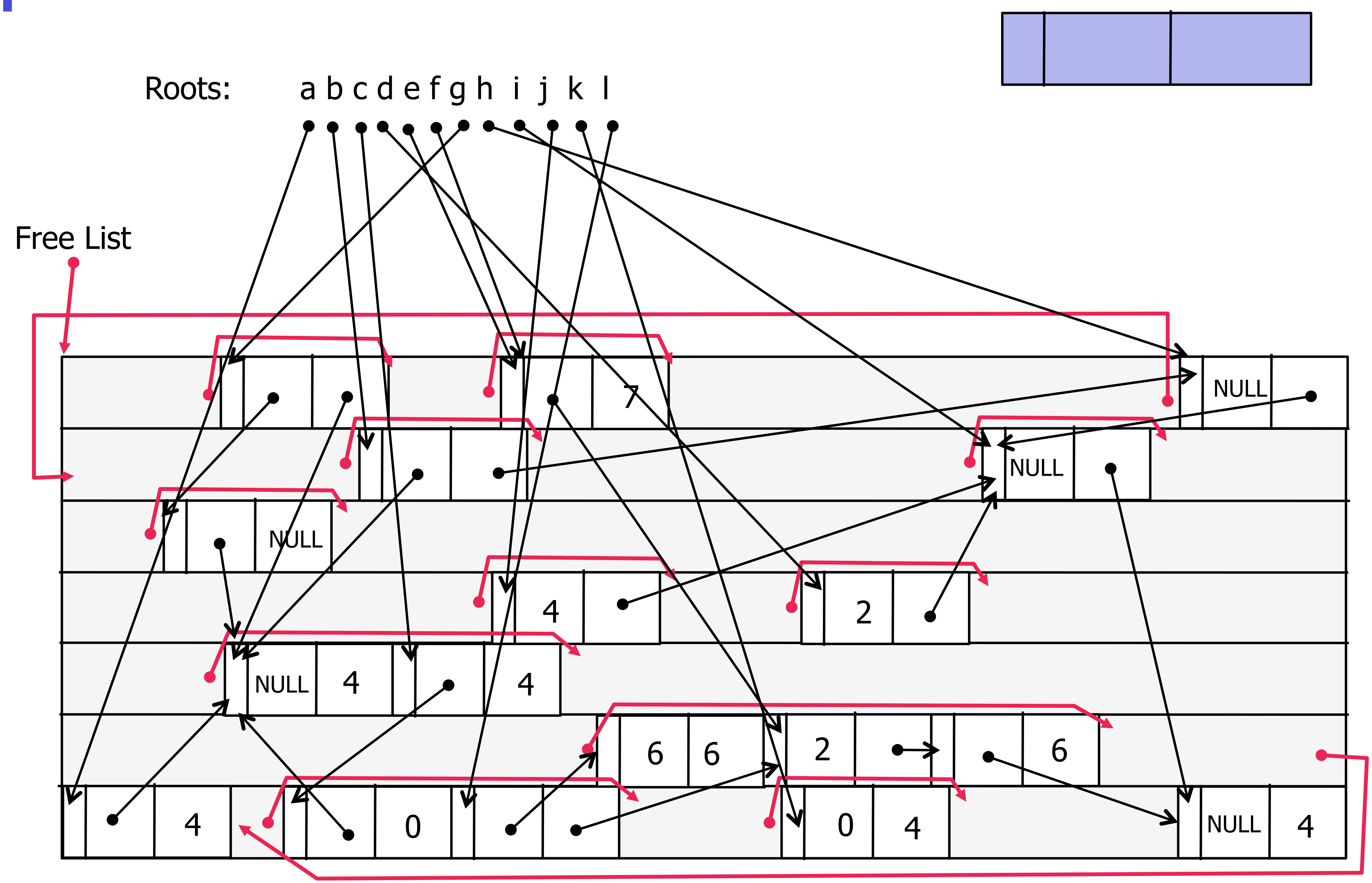


The Free List: Internal VM/runtime data structure – linked list of free blocks

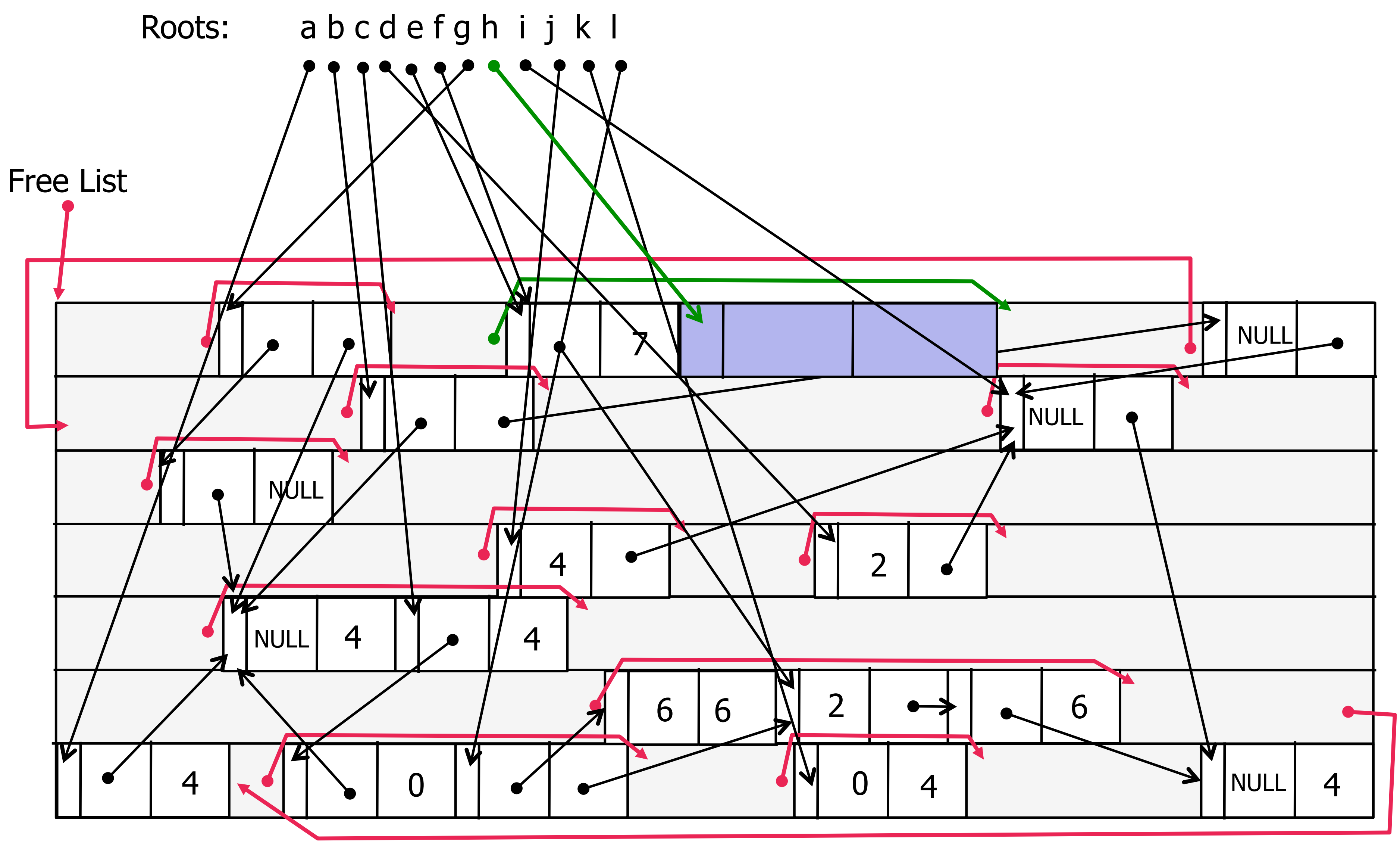
- Memory (virtual/heap) is one big contiguous array
- Typically multiple linked lists (of different sizes)
    - Allocation takes the chuck of list that is ≥ the size needed
    - Deallocation/free puts them on the front for reuse (in cache)
- When a partial block is used, the remainder gets put back on a list (acc. to size)
- When two blocks are next to each other, they can be combined
- Multiple allocations and frees can/will cause fragmentation



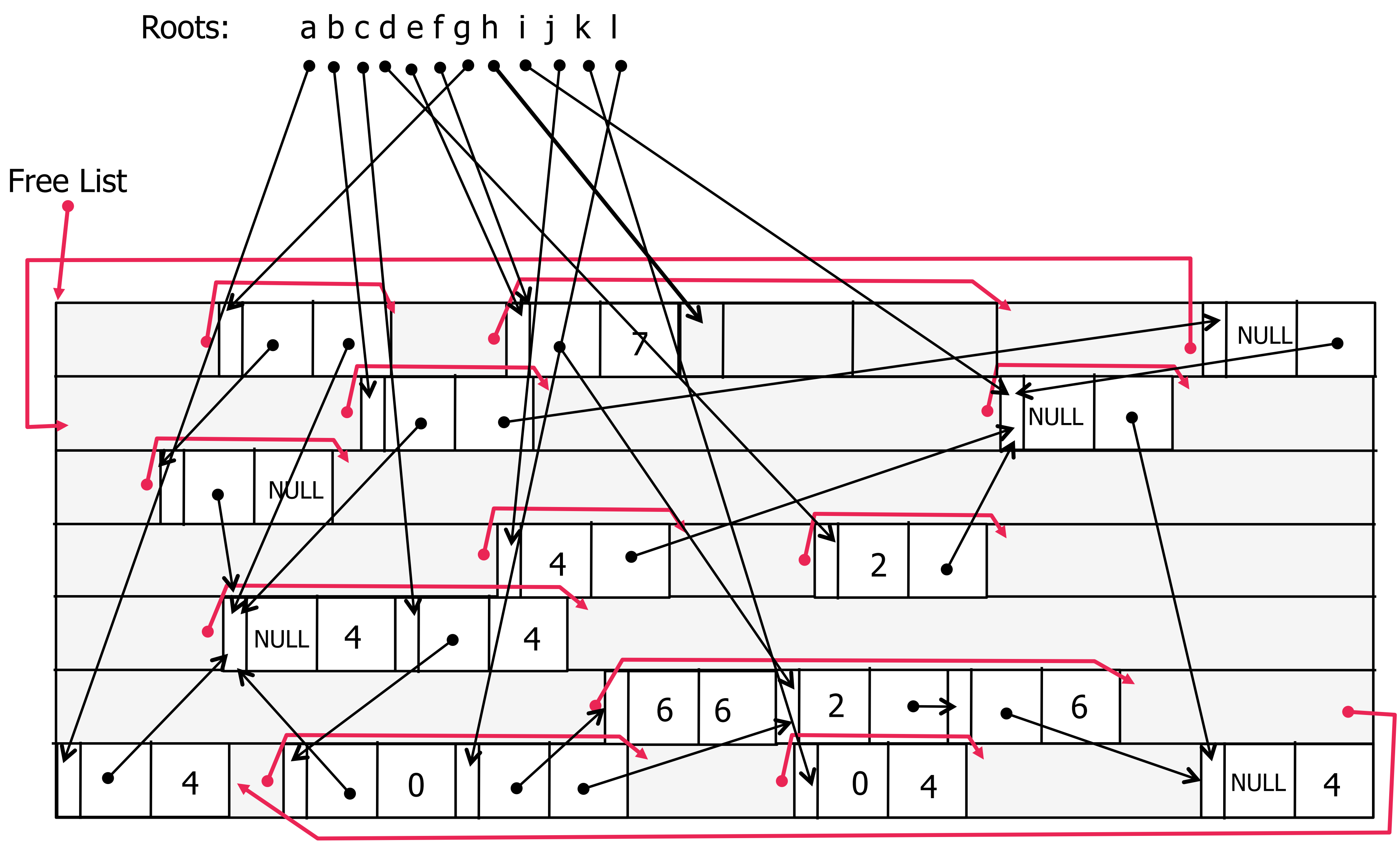
h = new LARGE_OBJECT()



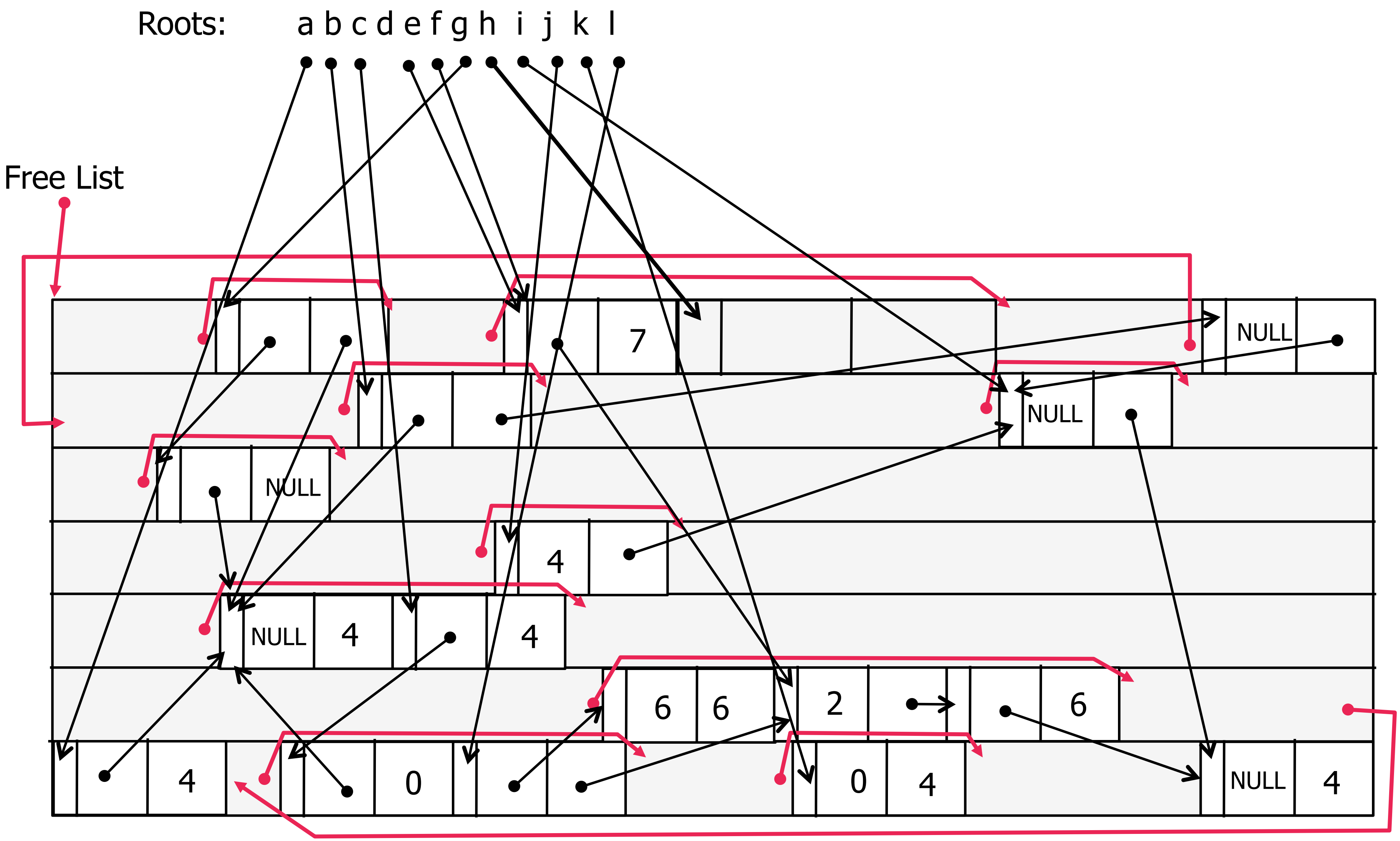
h = new LARGE_OBJECT()



h = new LARGE_OBJECT()
d = NULL (before)



h = new LARGE_OBJECT()
d = NULL (after)



Reference Counting GC Algorithm

- Each object has an additional atomic field in header
  - **Reference count**
    - Holds number of pointers to that cell from roots or other objects
- All cells placed in free list initially with count of 0
- **Free_list** points to the head of the free list

- Each time a **pointer is set** to refer to this cell, the count is incremented
- Each time a reference is removed, count is decremented
  - If the count goes to 0
    - There is no way for the program to access this cell
  - The cell is returned to the free list

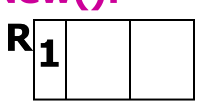
Reference Counting GC Algorithm

- When a new cell is allocated
  - Reference count is set to 1
  - Removed from free list
    - Assume, for now, that all cells are the same size and each has 2 fields left and right which are references

R->left = S

```
Allocate() {
   newcell = free_list
   free_list = free_list->next
   return newcell
}


New() {
   if (free_list) == NULL
      abort("Out of Memory")
  newcell = allocate()
  newcell->RC = 1
  return newcell
}
```

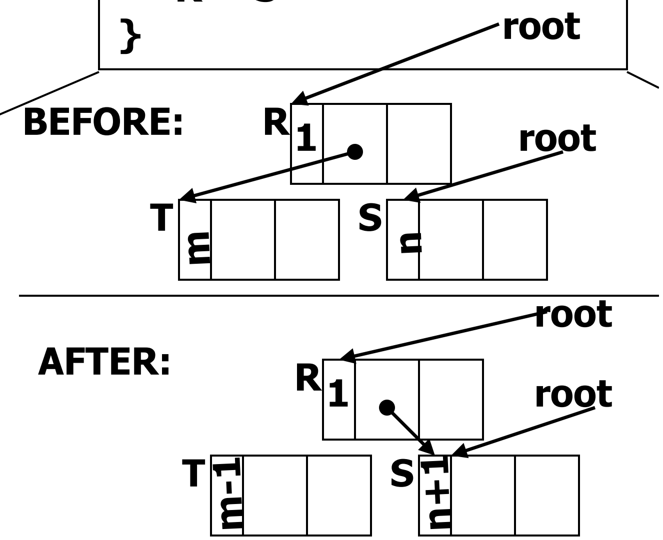
R = New():




```
Free(N) {
  N->next = free_list
  free_list = N
}


Delete(T) {
  T->RC--
  if (T->RC ==0) {
     for U in Children(T)
        Delete (*U)
     Free(T)
  }
}
```

Update(R->left,S)

```
Update(R,S) {
   // we assume R,S are
   // pointers

   S->RC++
   Delete(*R)
  *R = S
}
```



Reference Counting GC Algorithm: Update

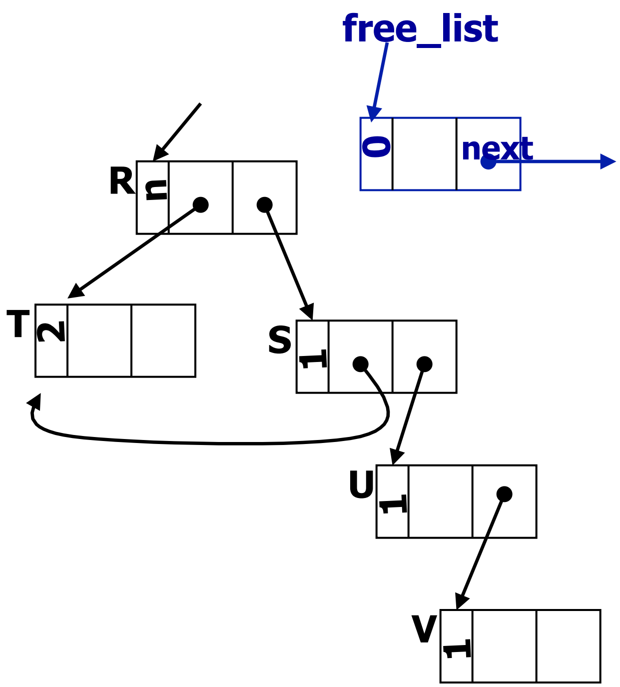
- Assume, for now, that all cells are the same size and each has 2 fields left and right which are references

```
Free(N) {
  N->next = free_list
  free_list = N
}
```

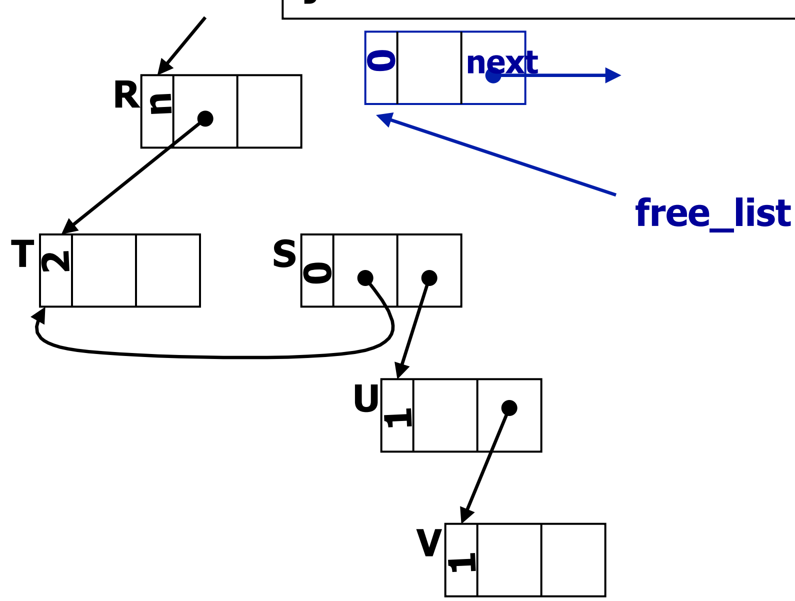
```
Delete(T) {
  T->RC--
  if (T->RC == 0) {
     for U in Children(T)
        Delete(*U)
      Free(T)
  }
}
```

```
Update(R,S) {
    // we assume R,S are
    // pointers and nulls
    // are handled correctly

    S->RC++
    Delete(*R)
   *R = S
}
```

R->right = NULL



Before Update(R->right,NULL)



Before if in Delete(*(R->right))

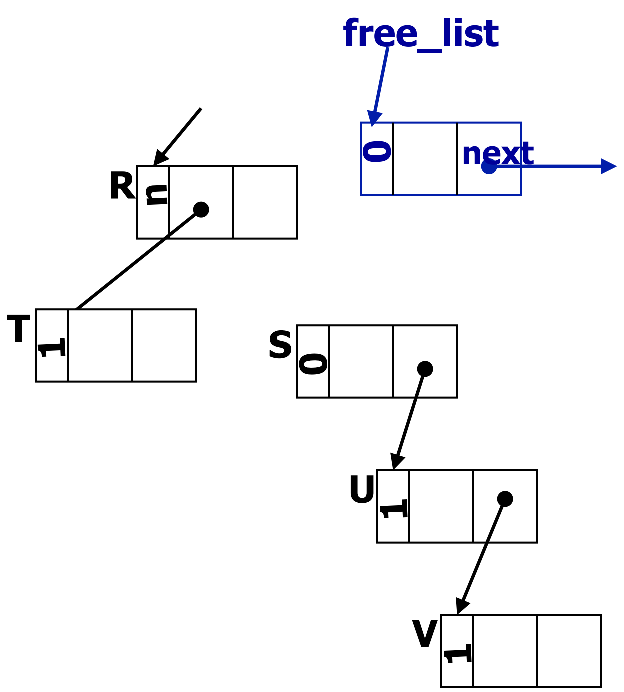
Reference Counting GC Algorithm: Update

- Assume, for now, that all cells are the same size and each has 2 fields left and right which are references

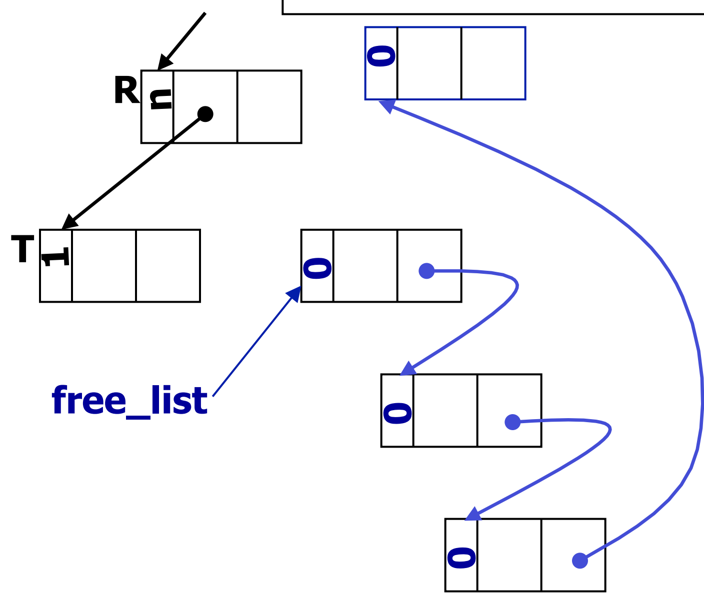
```
Free(N) {
  N->next = free_list
  free_list = N
}
```

```
Delete(T) {
  T->RC--
  if (T->RC == 0) {
    for U in Children(T)
      Delete(*U)
    Free(T)
  }
}
```

```
Update(R,S) {
   // we assume R,S are
   // pointers and nulls
   // are handled correctly

   S->RC++
   Delete(*R)
  *R = S
}
```



Delete(S->left)



After: Update(R->right,NULL)

Reference Counting GC

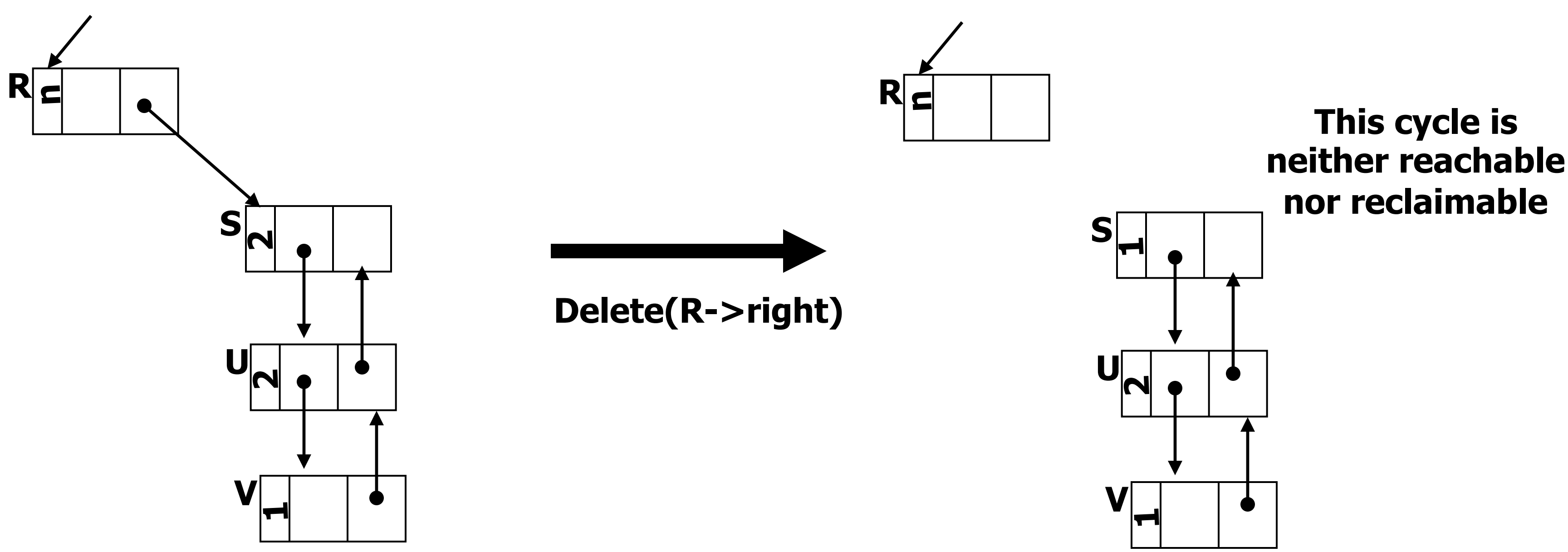
- Strengths
  - Memory management overheads are distributed throughout the computation
    - Management of active and garbage cells is interleaved with execution
    - **Incremental**
    - Smoother response time
  - Locality of reference
    - Things related are accessed together (for mem.hierarchy perf.)
    - No worse than program itself
  - Short-lived cells can be reused as soon as they are reclaimed
    - We don’t have to wait until memory is exhausted to free cells
    - Immediate reuse generates fewer page faults for virtual memory
    - Update in place is possible

Reference Counting GC

- Weaknesses
  - High processing cost for each pointer update
    - When a pointer is overwritten the reference count for **both** the old and new target cells must be adjusted
    - May cause poor memory performance
    - Hence, it is not used much in real systems
  - Fragile
    - Make sure to get all increments/decrements right
    - Increment for each call in which a pointer is passed as a parameter
    - Hard to maintain
  - Extra space in each cell to store count
    - Size = the number of pointers in the heap = sizeof(int)
    - Alternative: smaller size + overflow handling

Reference Counting GC

- Weaknesses
  - Cyclic data structures can't be reclaimed
    - Doubly linked lists
    - Solution: reference counting + something else (tracing)



Three Classic Garbage Collection Algorithms

- Reference counting
- Mark & Sweep
- Copying

Free List Allocation: keep 1+ lists of free chunks that we then fill or break off pieces of to allocate an object

Mark & Sweep GC Algorithm

- Tracing collector
  - Mark-sweep, Mark-scan
  - Use reachability (indirection) to find live objects
- Objects are **not reclaimed** immediately when they become garbage
  - Remain unreachable and undetected until storage is exhausted
- When reclamation happens the program is paused
  - **Sweep** all currently unused cells back into the free_list
  - GC performs a global traversal of all live objects to determine which cells are reachable **(live or active)**
    - Trace, starting from roots, marking them as reachable
    - Free all unmarked cells

Mark & Sweep GC Algorithm

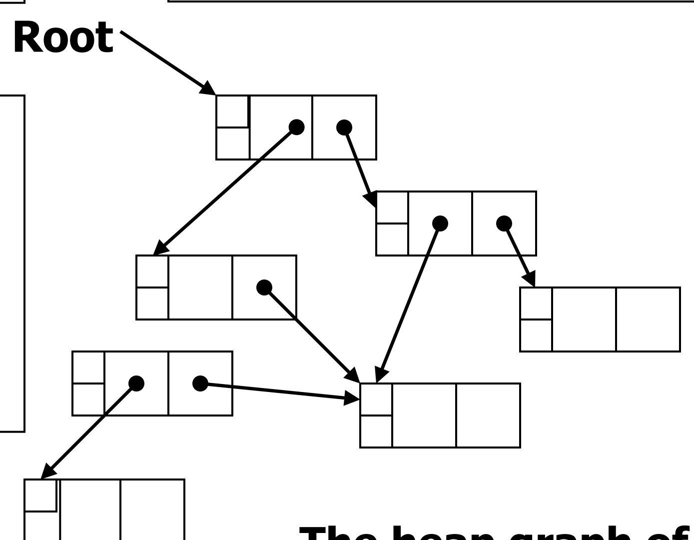
- Each cell contains 1 bit (mark_bit) of extra information
- Cells in free_list have mark_bits set to 0
- No Update(...) routine necessary

```
New() {
  if free_list->isEmpty()
    mark_sweep
  newcell = allocate()
  return newcell
}
```

```
mark(N) {
  if N->mark_bit == 0
    N->mark_bit = 1
    for M in Children(N)
      mark(M)
}
```

```
sweep() {
  N = heap_start
  while (N < heap_end) {
    if N->mark_bit == 0
      free(N)
    else N->mark_bit = 0
    N+=sizeof(N)
  }
}
```

```
mark_sweep() {
  for R in Roots
    mark (R)
  sweep()
  if free_list->isEmpty()
   abort("OutOfMemory")
}
```



The heap graph of objects

Mark & Sweep GC Algorithm

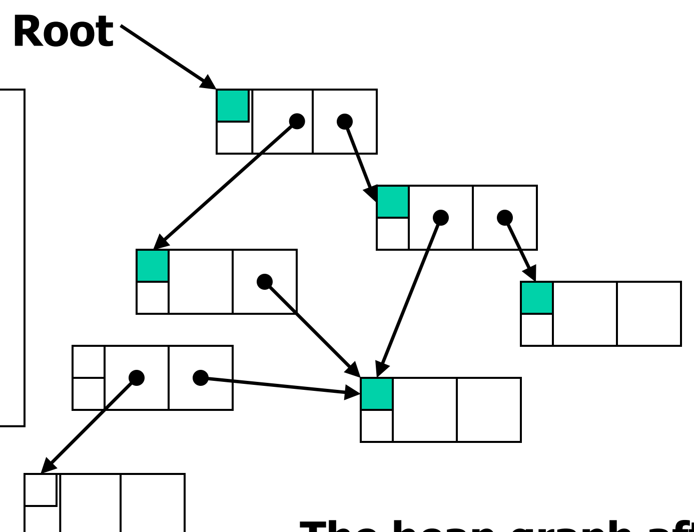
- Each cell contains 1 bit (mark_bit) of extra information
- Cells in free_list have mark_bits set to 0
- No Update(…) routine necessary

```
New() {
  if free_list->isEmpty()
    mark_sweep
  newcell = allocate()
  return newcell
}
```

```
mark(N) {
  if N->mark_bit == 0
    N->mark_bit = 1
    for M in Children(N)
      mark(M)
}
```

```
sweep() {
  N = heap_start
  while (N < heap_end) {
    if N->mark_bit == 0
      free(N)
    else N->mark_bit = 0
    N+=sizeof(N)
  }
}
```

```
mark_sweep() {
  for R in Roots
    mark (R)
  sweep()
  if free_list->isEmpty()
   abort("OutOfMemory")
}
```



The heap graph after the marking phase, all unmarked cells are garbage

Mark & Sweep GC Algorithm

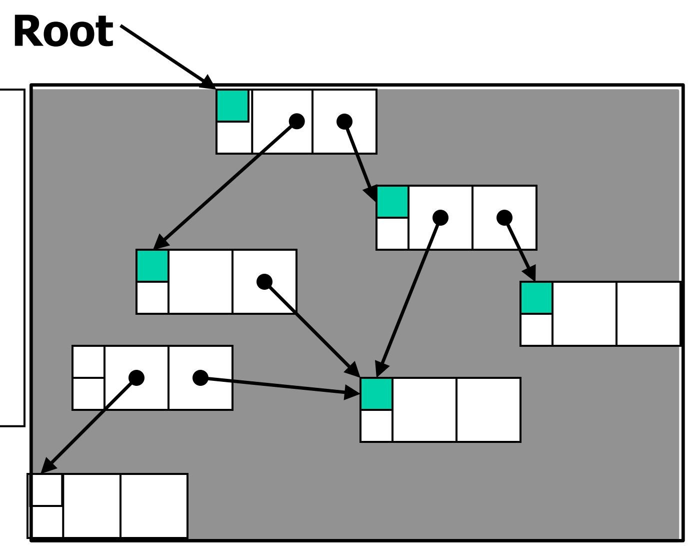
- Each cell contains 1 bit (mark_bit) of extra information
- Cells in free_list have mark_bits set to 0
- No Update(...) routine necessary

```
New() {
  if free_list->isEmpty()
    mark_sweep
  newcell = allocate()
  return newcell
}
```

```
mark(N) {
  if N->mark_bit == 0
    N->mark_bit = 1
    for M in Children(N)
      mark(M)
}
```

```
sweep() {
  N = heap_start
  while (N < heap_end) {
    if N->mark_bit == 0
      free(N)
    else N->mark_bit = 0
    N+=sizeof(N)
  }
}
```

```
mark_sweep() {
  for R in Roots
    mark (R)
  sweep()
  if free_list->isEmpty()
   abort("OutOfMemory")
}
```



The heap graph after the marking phase, all unmarked cells are garbage

All of the gray areas are skipped (but considered) during sweeping

Mark & Sweep GC Algorithm

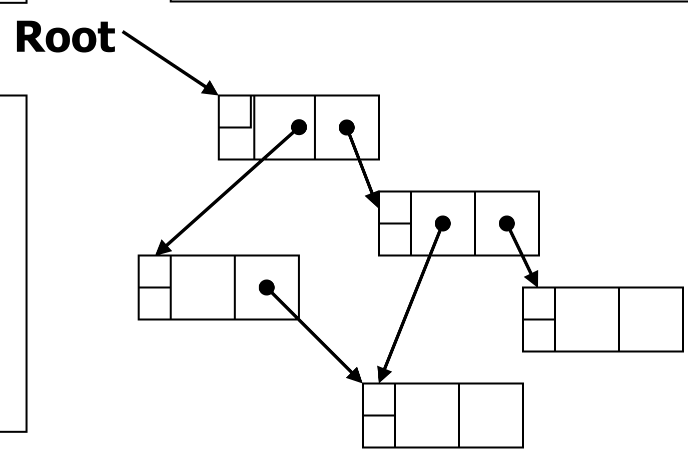
- Each cell contains 1 bit (mark_bit) of extra information
- Cells in free_list have mark_bits set to 0
- No Update(...) routine necessary

```
New() {
  if free_list->isEmpty()
    mark_sweep
  newcell = allocate()
  return newcell
}
```

```
mark(N) {
  if N->mark_bit == 0
    N->mark_bit = 1
    for M in Children(N)
      mark(M)
}
```

```
sweep() {
  N = heap_start
  while (N < heap_end) {
    if N->mark_bit == 0
      free(N)
    else N->mark_bit = 0
    N+=sizeof(N)
  }
}
```

```
mark_sweep() {
  for R in Roots
    mark (R)
  sweep()
  if free_list->isEmpty()
   abort("OutOfMemory")
}
```



The heap graph after the sweeping phase, all unmarked cells are live

Mark & Sweep GC Algorithm

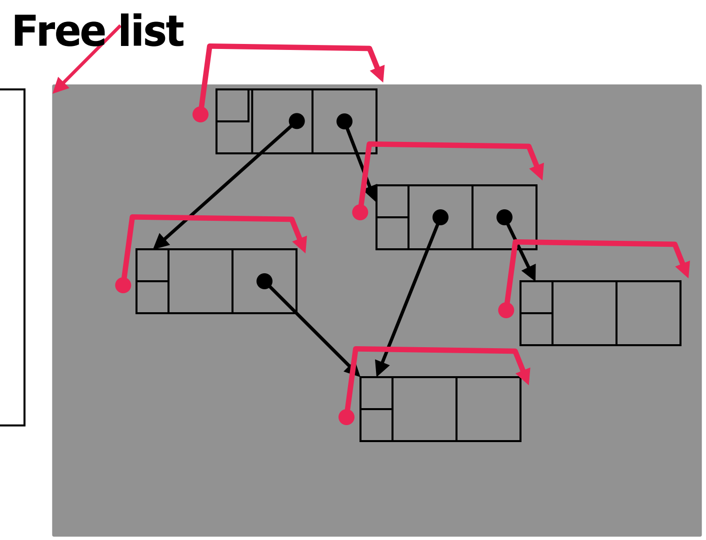
- Each cell contains 1 bit (mark_bit) of extra information
- Cells in free_list have mark_bits set to 0
- No Update(...) routine necessary

```
New() {
  if free_list->isEmpty()
    mark_sweep
  newcell = allocate()
  return newcell
}
```

```
mark(N) {
  if N->mark_bit == 0
    N->mark_bit = 1
    for M in Children(N)
      mark(M)
}
```

```
sweep() {
  N = heap_start
  while (N < heap_end) {
    if N->mark_bit == 0
      free(N)
    else N->mark_bit = 0
    N+=sizeof(N)
  }
}
```

```
mark_sweep() {
  for R in Roots
    mark (R)
  sweep()
  if free_list->isEmpty()
   abort("OutOfMemory")
}
```



The heap graph after the sweeping phase, all unmarked cells are live

Mark & Sweep GC Algorithm

- Strengths
  - Cycles are handled quite normally
  - No overhead placed on pointer manipulations
  - Better than (incremental) reference counting
- Weaknesses
  - Start-stop algorithm (aka stop-the-world)
    - Computation is halted while GC happens
    - Not practical for real-time systems
  - Asymptotic complexity is proportional to the size of the heap not just the live objects
    - For sweep

Mark & Sweep GC Algorithm

- Weaknesses (continued)
  - Fragments memory (scatters free cells across memory)
    - Loss of memory performance (caching/paging)
    - Allocation is complicated (need to find a set of cells for the right size)
  - **Residency** - heap occupancy
    - As this increases, the need for garbage collection will become more frequent
    - Taking processing cycles away from the application
    - Allocation and program erformance degrades as residency increases

Mark-Compact

- Mark-sweep with compaction
- Compact live data during reclamation
- Advantages
  - Zero fragmentation
  - Fast allocation – Increment a pointer into free space
  - Improved locality
- Disadvantages
  - At least two passes required during compaction

Three Classic Garbage Collection Algorithms

- Reference counting
- Mark & Sweep
- Copying

Free List Allocation: keep 1+ lists of free chunks that we then fill or break off pieces of to allocate an object

Three Classic Garbage Collection Algorithms

- Reference counting
- Mark & Sweep
- Copying

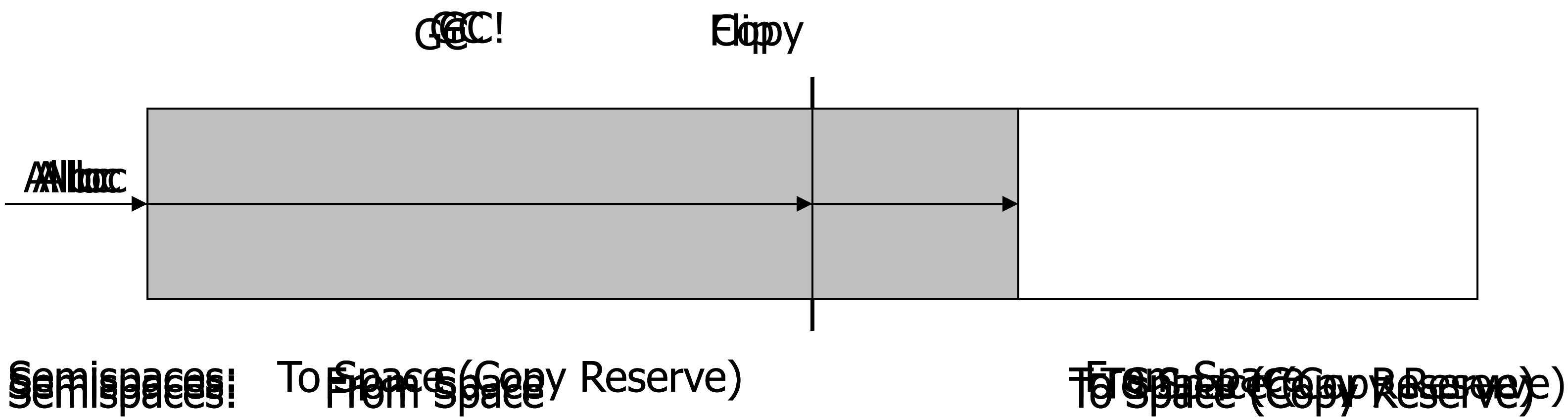
Free List Allocation: keep 1+ lists of free chunks that we then fill or break off pieces of to allocate an object

Bump-Pointer Allocation: increment a pointer to get the next chunk of memory for an object being allocated

Copying Collector

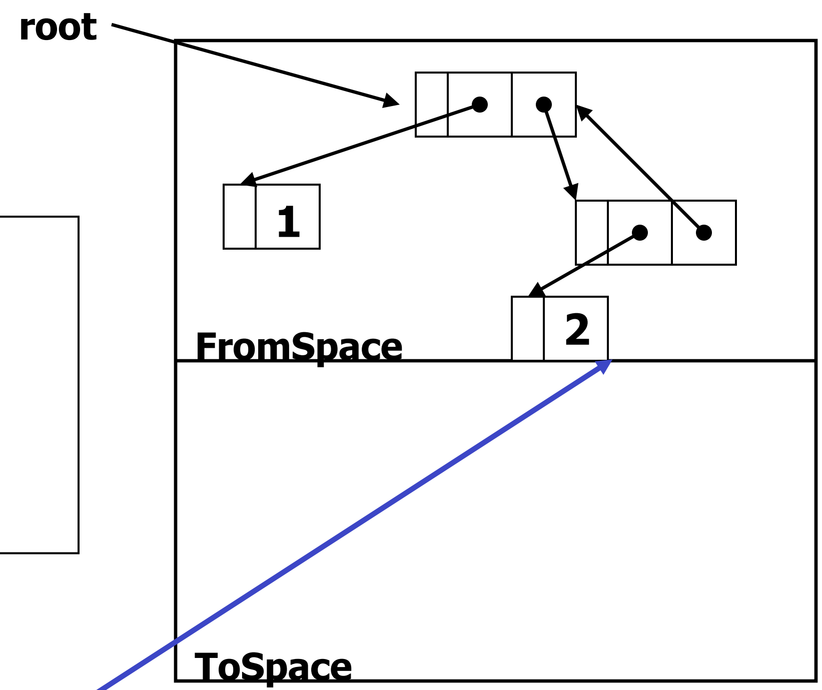
- Tracing, stop-the-world collector
  - Divide the heap into two **semispaces**
    - One with current data
    - The other with obsolete data
  - The roles of the two semispaces is continuously **flipped**
  - Collector copies live data from the old semispace
    - FromSpace
    - To the new semispace (ToSpace) when visited
    - Pointers to objects in ToSpace are updated
    - Program is restarted
  - **Scavengers**
    - FromSpace is not reclaimed, just abandoned

Copying Collection



- Advantages
  - Fast allocation – Increment a pointer into free space
    - Bump pointer allocation
  - No fragmentation
- Disadvantages
  - Available heap space is **halved**
  - Large copying cost
  - **Locality** not always improved

Copying Collector




```
InitGC() {
   ToSpace = heap_start
   space_size = heap_size/2
   top_of_space = ToSpace+space_size
   FromSpace = top_of_space+1
   freeptr = toSpace
}
```

```
New(n) {
   if freeptr+n > top_of_space
      flip()
   if freeptr+n > top_of_space
      abort("OutOfMemory")
   newcell = freeptr
   freeptr = freeptr+n
   return newcell
}
```

```
flip() {
   FromSpace, ToSpace = ToSpace, FromSpace
   top_of_space = ToSpace+space_size
   freeptr = ToSpace
   for R in Roots
      R=copy(R)
}
```

AKA: the bump pointer

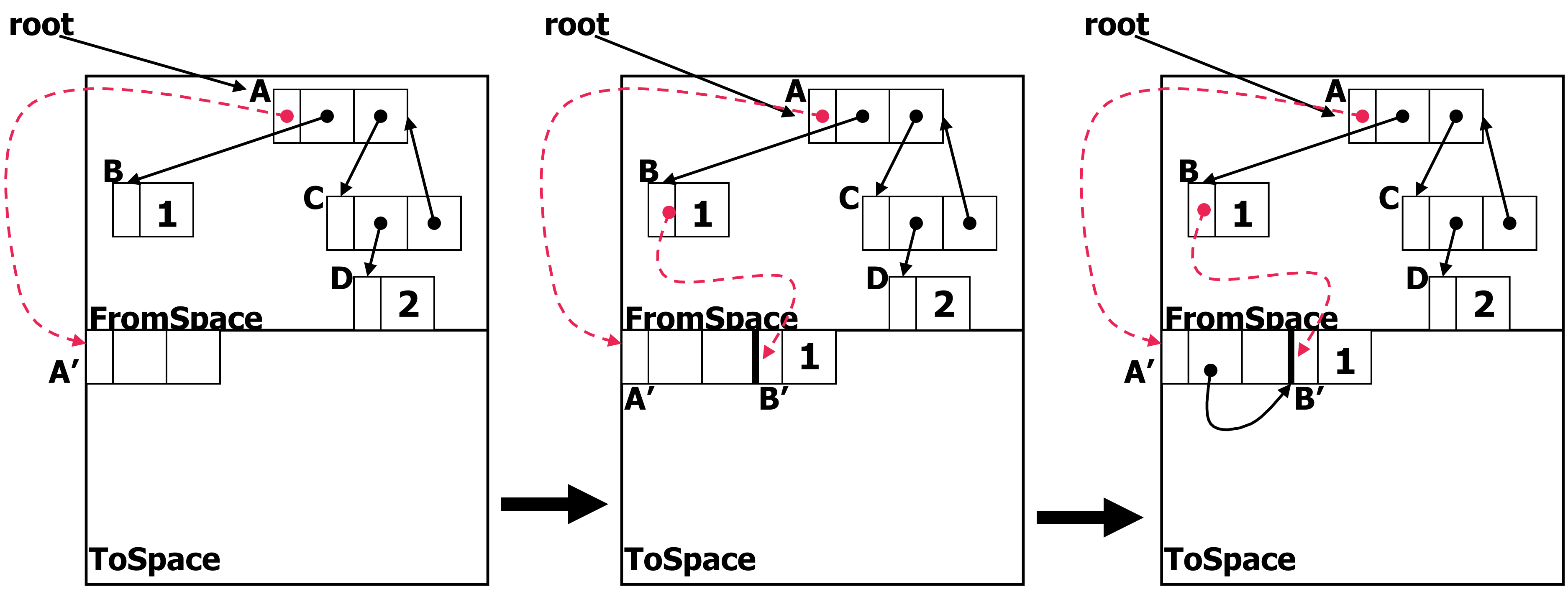
Copying Collector

```
flip() {
   FromSpace, ToSpace = ToSpace, FromSpace
   top_of_space = ToSpace+space_size
   freeptr = ToSpace
   for R in Roots
      R=copy(R)
}
```

```
copy(P)
   if P==NULL || P->is_atomic
      return P
   if !forwarded(P) {
      n = size(P)
      P' = freeptr
      freeptr = freeptr+n
     fowarding_address(P) = P'
      for (i = 0; i<n; i++)
         P'[i] = copy(P[i]);
   }
   return fowarding_address
}
```

→ pointer

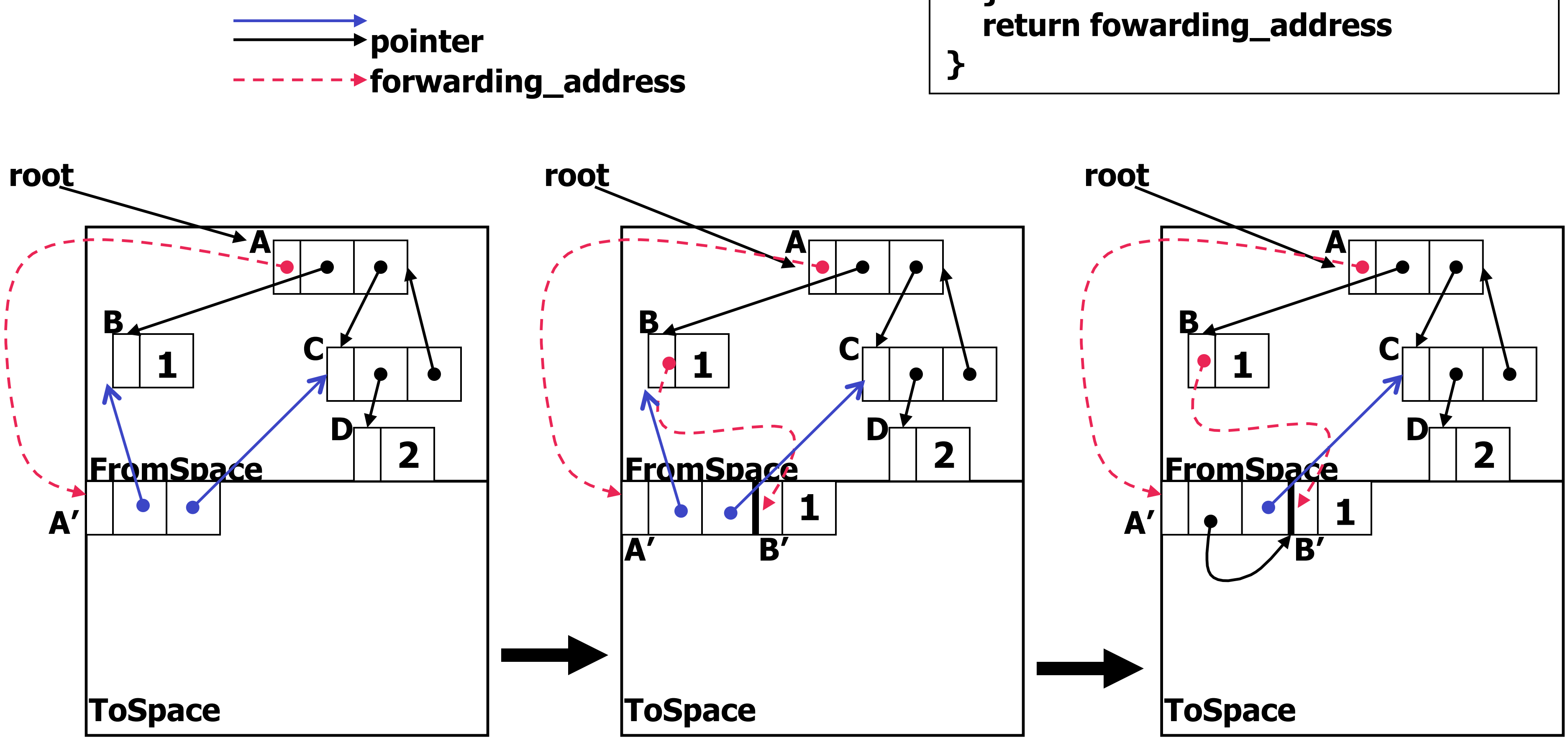
⇢ forwarding_address



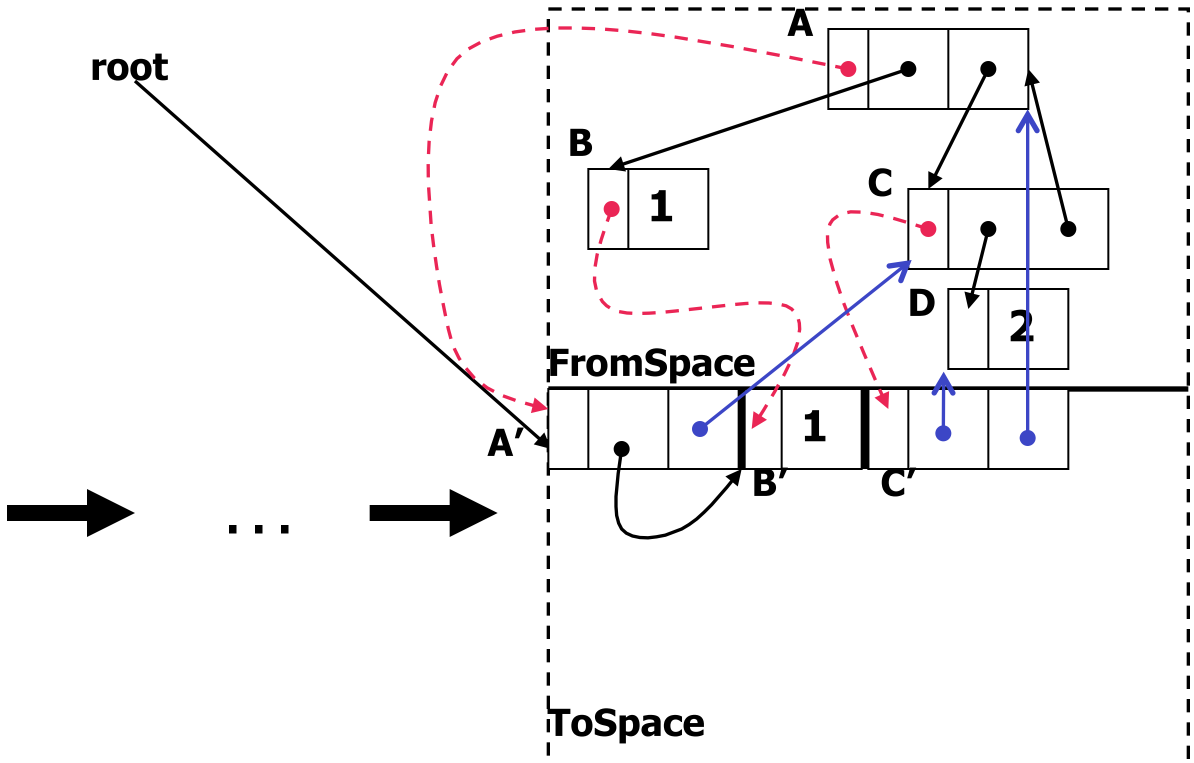
Copying Collector

```
flip() {
    FromSpace, ToSpace = ToSpace, FromSpace
    top_of_space = ToSpace+space_size
    freeptr = ToSpace
    for R in Roots
        R=copy(R)
}
```

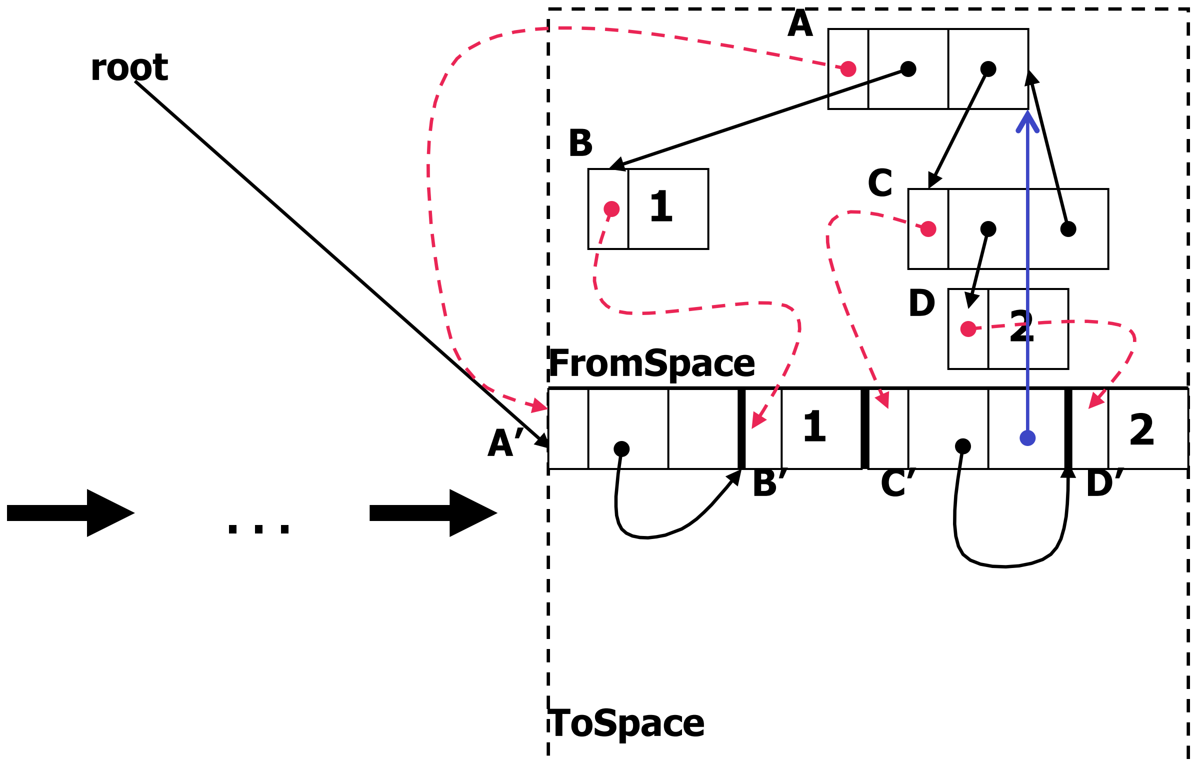
```
copy(P)
    if P==NULL || P->is_atomic
        return P
    if !forwarded(P) {
        n = size(P)
        P' = freeptr
        freeptr = freeptr+n
       fowarding_address(P) = P'
        for (i = 0; i<n; i++)
            P'[i] = copy(P[i]);
    }
    return fowarding_address
}
```



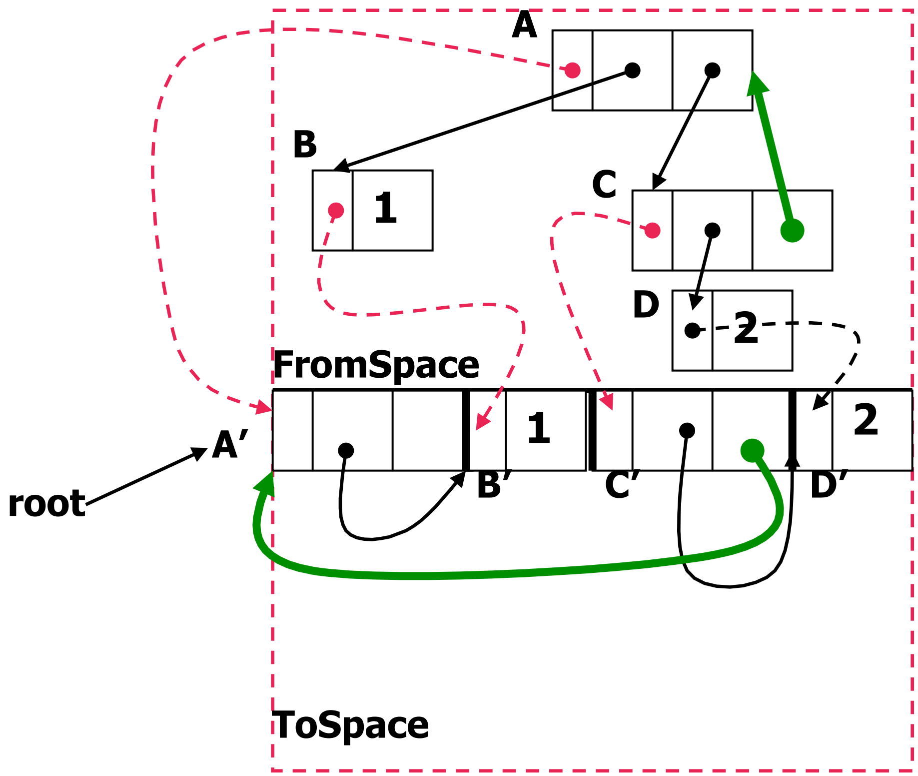
Copying (Semispace) Collector



Copying (Semispace) Collector

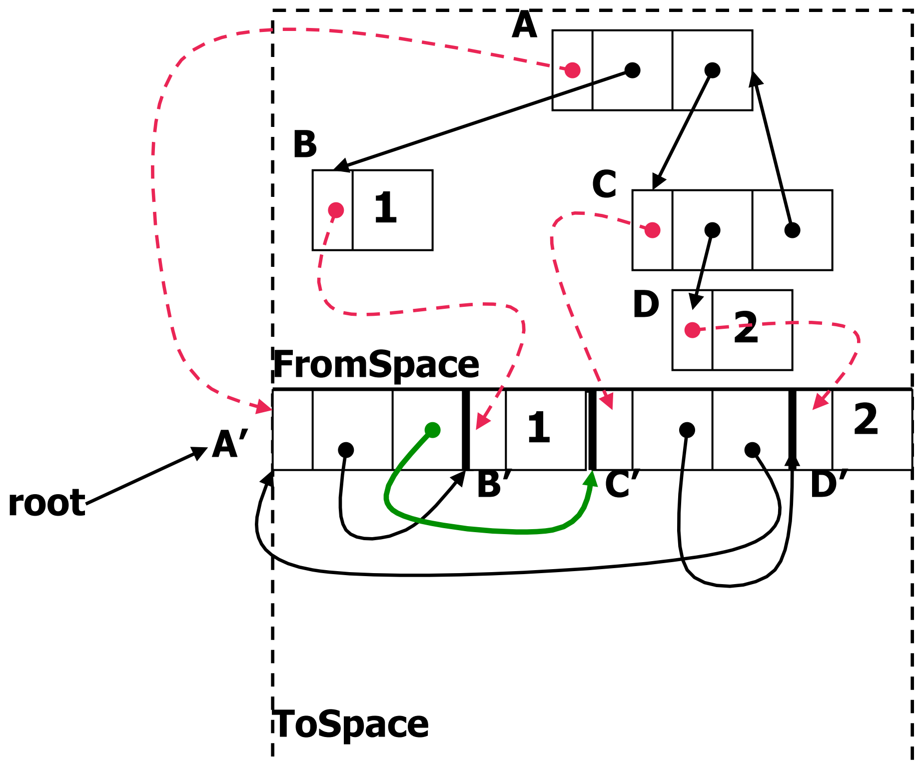


Copying (Semispace) Collector



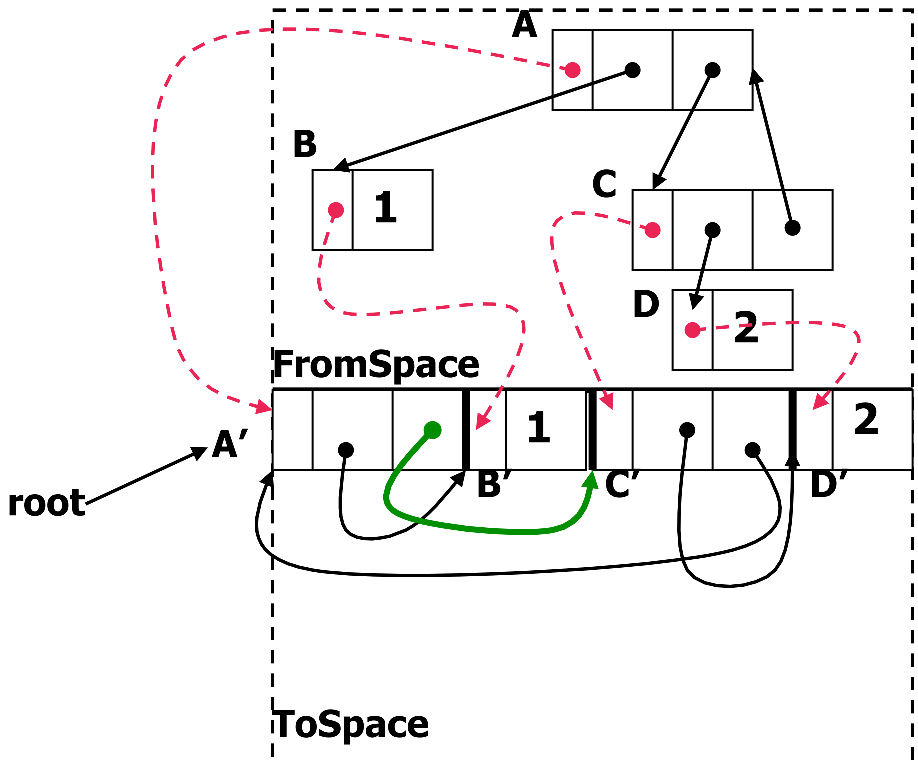

```
copy(P)
   if P==NULL || P->is_atomic
      return P
   if !forwarded(P) {
      n = size(P)
      P' = freeptr
      freeptr = freeptr+n
      fowarding_address(P) = P'
      for (i = 0; i<n; i++)
         P'[i] = copy(P[i]);
   }
   return fowarding_address
}
```

Copying (Semispace) Collector

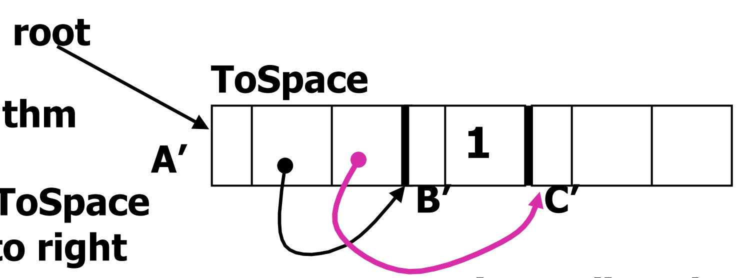



```
copy(P)
   if P==NULL || P->is_atomic
      return P
   if !forwarded(P) {
      n = size(P)
      P′ = freeptr
      freeptr = freeptr+n
      fowarding_address(P) = P′
      for (i = 0; i<n; i++)
         P′[i] = copy(P[i]);
   }
   return fowarding_address
}
```

Copying (Semispace) Collector




```
copy(P)
   if P==NULL || P->is_atomic
      return P
   if !forwarded(P) {
      n = size(P)
      P′ = freeptr
      freeptr = freeptr+n
      fowarding_address(P) = P′
      for (i = 0; i<n; i++)
         P′[i] = copy(P[i]);
   }
   return fowarding_address
}
```



Cheney's Algorithm
(Breadth-First)
Move roots, to ToSpace
Scan from left to right
moving objs in FromSpace to ToSpace (at end) and updating pointers

Copying Collector

- Strengths
  - Have lead to its widespread adoption
  - Active data is compact (not fragmented as in mark-sweep)
    - More efficient allocation, just grab the next group of cells that fits
    - The check for space remaining is simply a pointer comparison
  - Handles variable-sized objects naturally
  - No overhead on pointer updates
  - Allocation is a simple free-space pointer increment
  - Fragmentation is eliminated
    - Compaction offers improved memory hierarchy performance of the user program

Copying Collector

- Weaknesses
  - Required address space is doubled compared with non-copying collectors
    - Primary drawback is the need to divide memory into two
    - Performance degrades as residency increases (twice as quickly as mark&sweep b/c half the space)
  - Touches every page (VM) of the heap regardless of residency of the user program
    - Unless both semispaces can be held in memory simultaneously

Other Things You Should Know

- Conservative collectors
  - Non-copying only
  - Imprecise / Not type-accurate
    - Data values may or may not be pointers
    - Some optimizing compilers make it very difficult to distinguish between the two
  - Any thing that looks like a pointer is one
    - Don't collect it - just in case

- Non-copying also good for programs with modules written in different languages (and opt'd by different compilers)
  - Pointers that escape across these boundaries are not collected

The Principle of Locality

- A good GC should not only reclaim memory but improve the locality of the system on the whole
  - Principle of locality - programs access a relatively small portion of their address space at any particular time
    - What are the two types of locality
  - GC should ensure that locality is exploited to improve performance wherever possible
  - Memory hierarchy was developed to exploit the natural principle of locality in programs
    - Different levels of memory each with different speeds/sizes/cost
    - Registers, cache, memory, virtual memory

Other Popular GCs

- Observations with previous GCs
  - Long-lived objects are hard to deal with
  - Young objects (recently allocated) die young
    - Most are young (80-90%) = **weak-generational hypothesis**
  - Large heaps (that can't be held in memory) degrade perf.
- Goal: Make large heaps more efficient by concentrating effort where the greatest payoff is
- Solution: **Generational GC**
  - Exploit the lifetime of objects to make GC and the program's use of the memory hierarchy more efficient

Generational GC

- Segregate objects by age into two or more heap regions
  - Generations
    - Keep the young generation separate
  - Collected at different frequencies
    - The younger the more often
    - The oldest, possibly never

- Can be implemented as an incremental scheme or as a stop-the-world scheme
  - Using different algorithms on the different regions

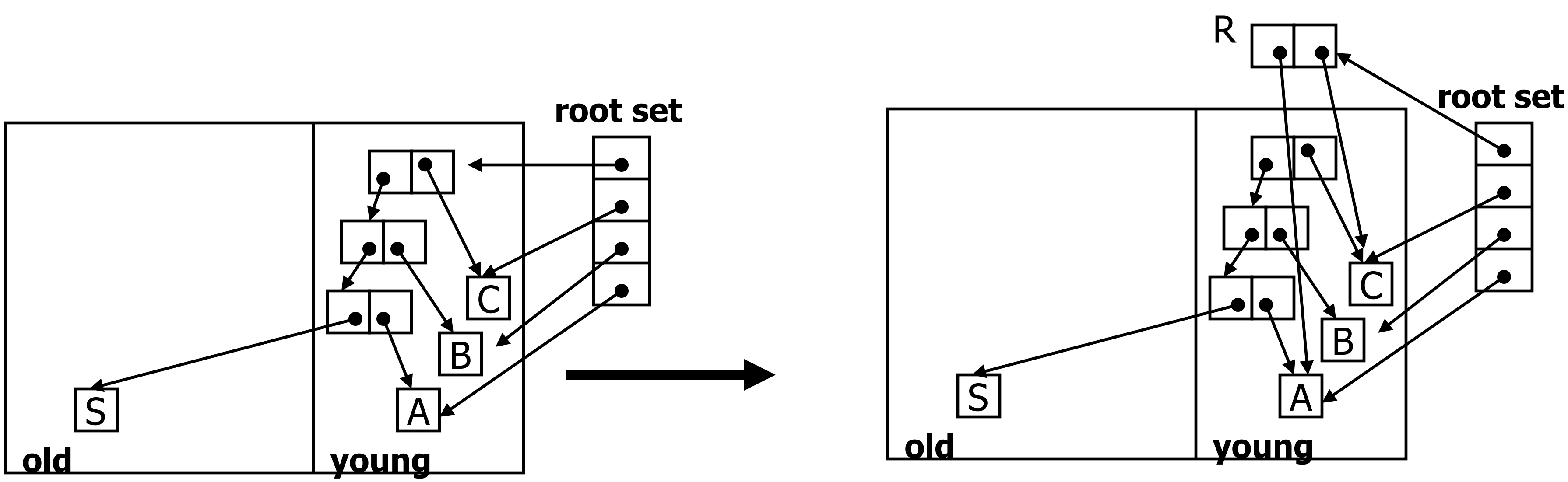
- Ok - so how do we measure life times?

Measuring Object Lifetimes

- Time?
  - Machine dependent - depend on the speed of the machine
  - Alternative 1: Number of instructions executed - also dependent across instruction set architectures
  - Alternative 2: Number of bytes allocated in the heap
    - Machine dependent
    - But gives a good measure of the demands made on the memory hierarchy
    - Closely related to the frequency of collection
    - Problems
      - In interactive systems, this can be dependent upon user behavior
      - Language and VM dependent

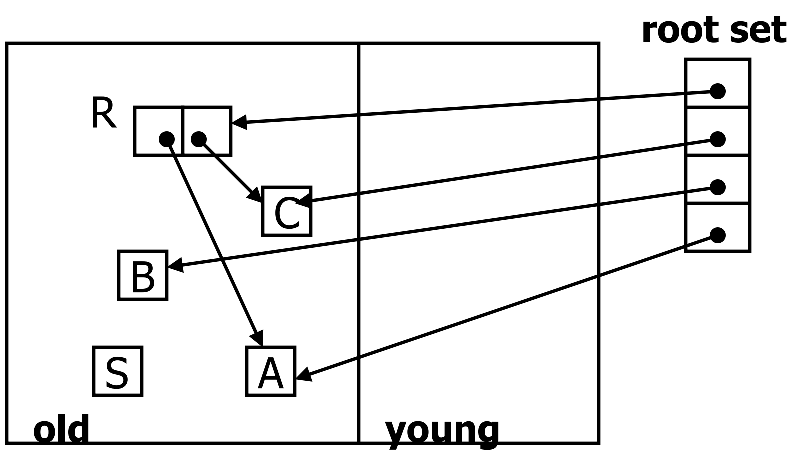
Generational GC

- Promotion
  - Move object to older generation if its survives long enough
- Concentrate on youngest generation for reclamation
  - This is where most of the recyclable space will be found
  - Make this region small so that its collection can be more frequent but with shorter interruption



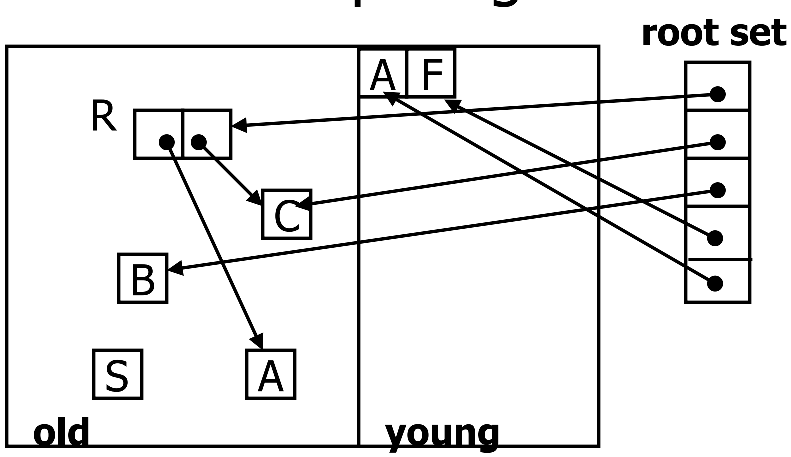
Generational GC

- A younger generation can be collected without collecting an older generation
- The pause time to collect a younger gen. is shorter than if a collection of the heap is performed
- Young objs that survive **minor** collections are **promoted**
  - Minor collections reclaim shortlived objects
- **Tenured garbage** - garbage in older generations

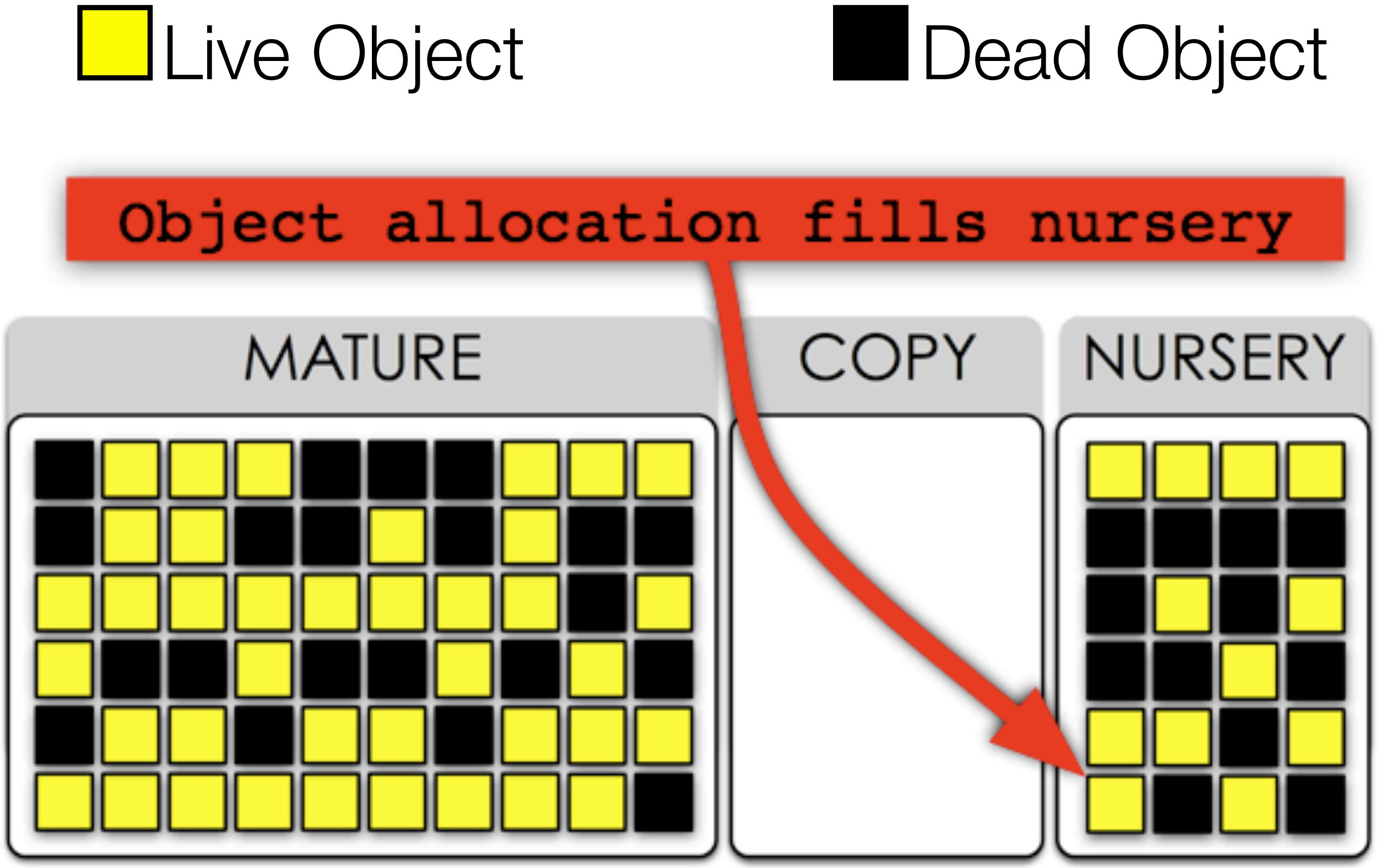


Generational GC (Review)

- Allocation always from minor
  - Except perhaps for large or known-to-be-old objects
- Minor frequent, Major very infrequent
- Major/minor collections can be any type
  - Mark/sweep, copying, mark/compact, **hybrid**
  - Promotion is copying
- Can have more than 2 generations
  - Each requiring collection of those lower/younger

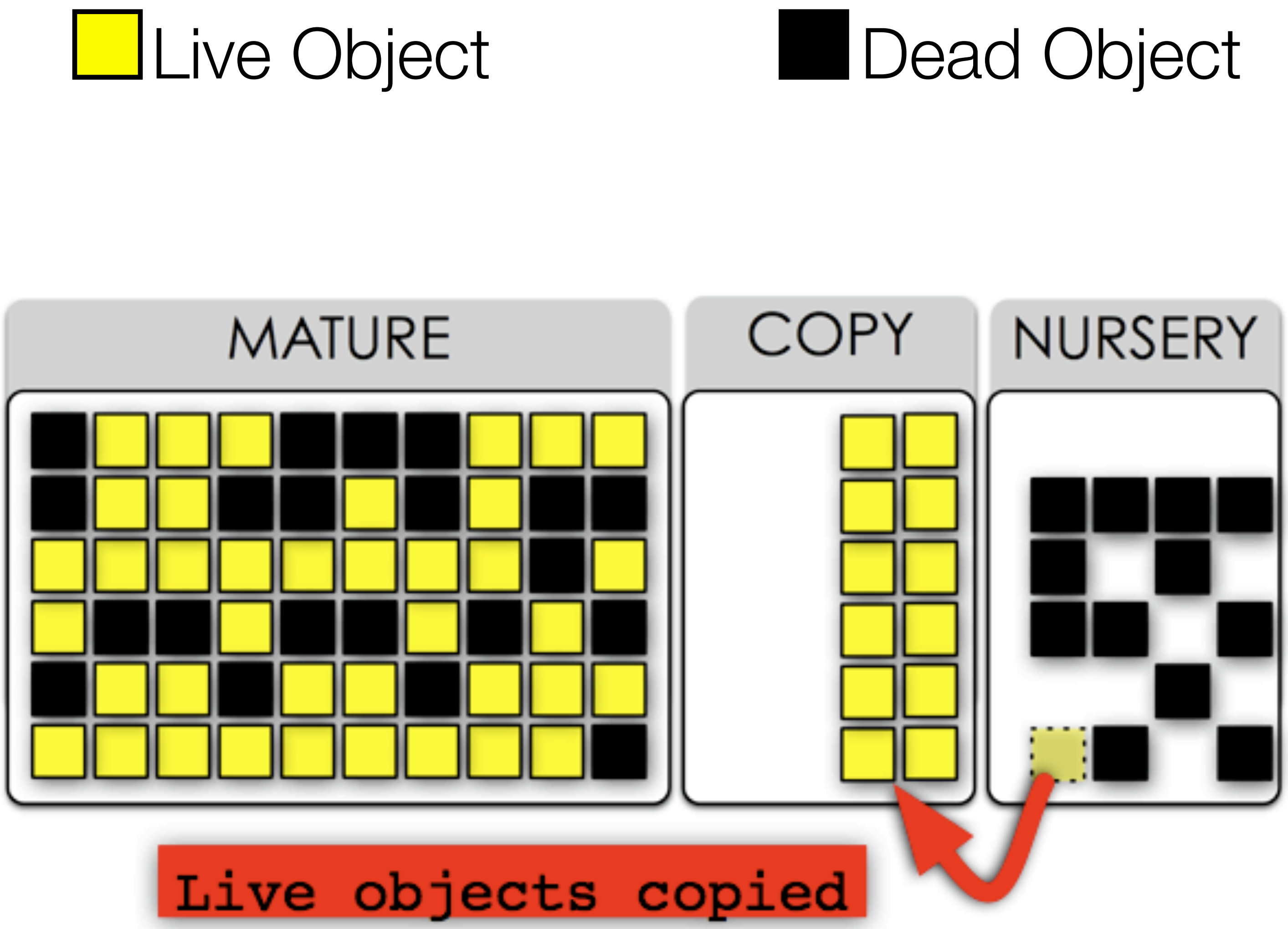


Nursery GC



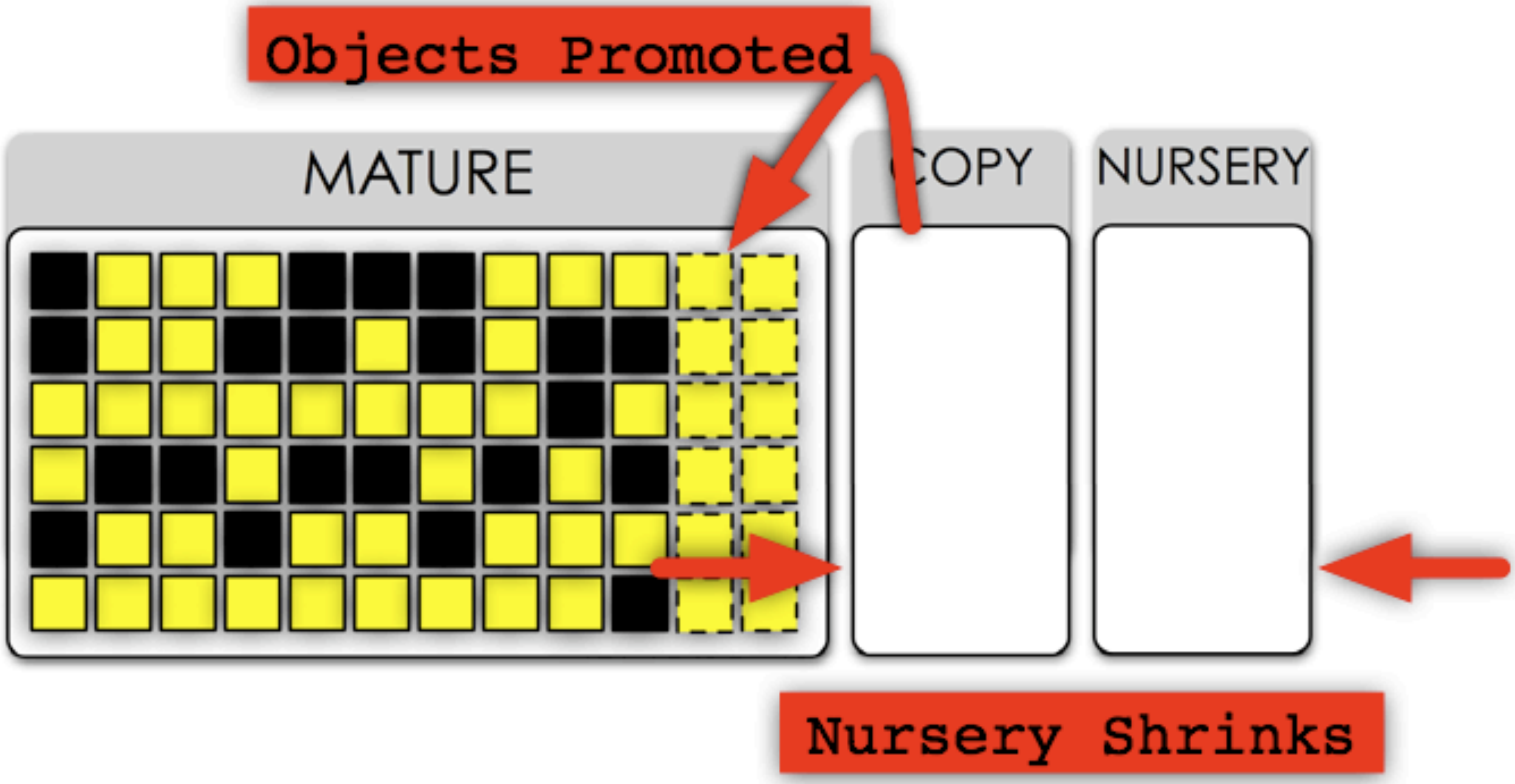
Live Object
Dead Object
Object allocation fills nursery
MATURE
COPY
NURSERY

Nursery GC: Copy

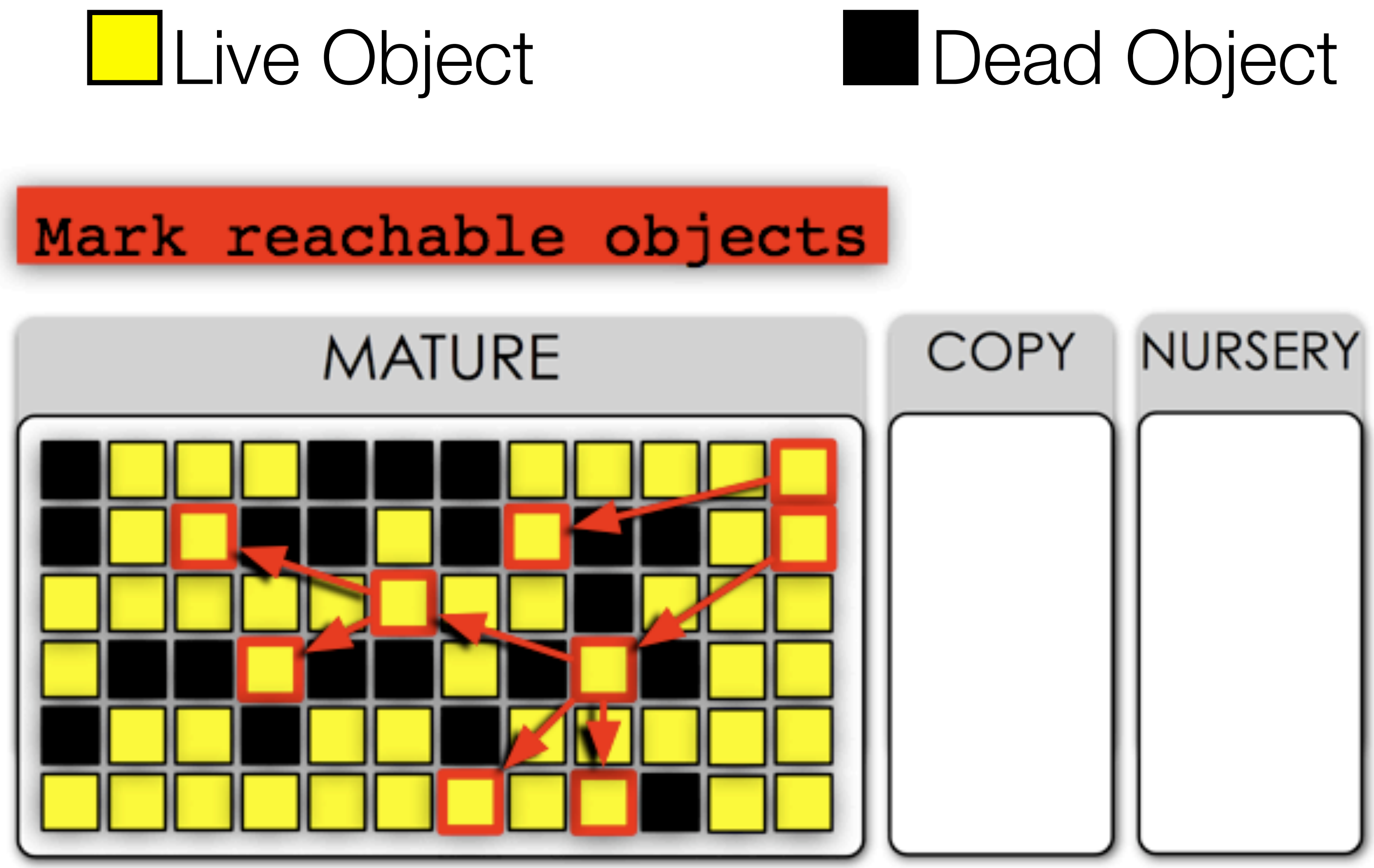


Nursery GC: Promotion

Live Object Dead Object

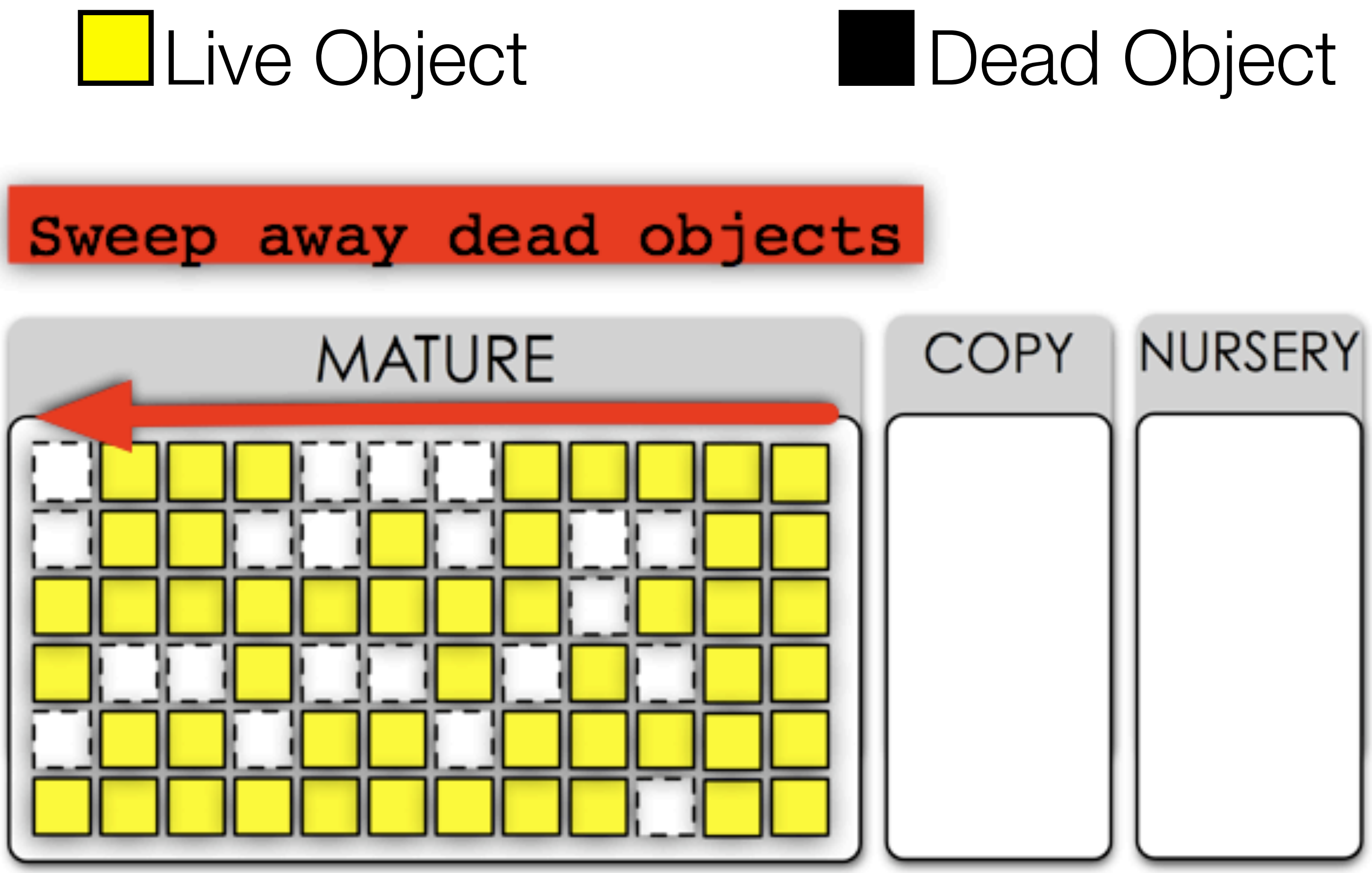


Full GC: Mark



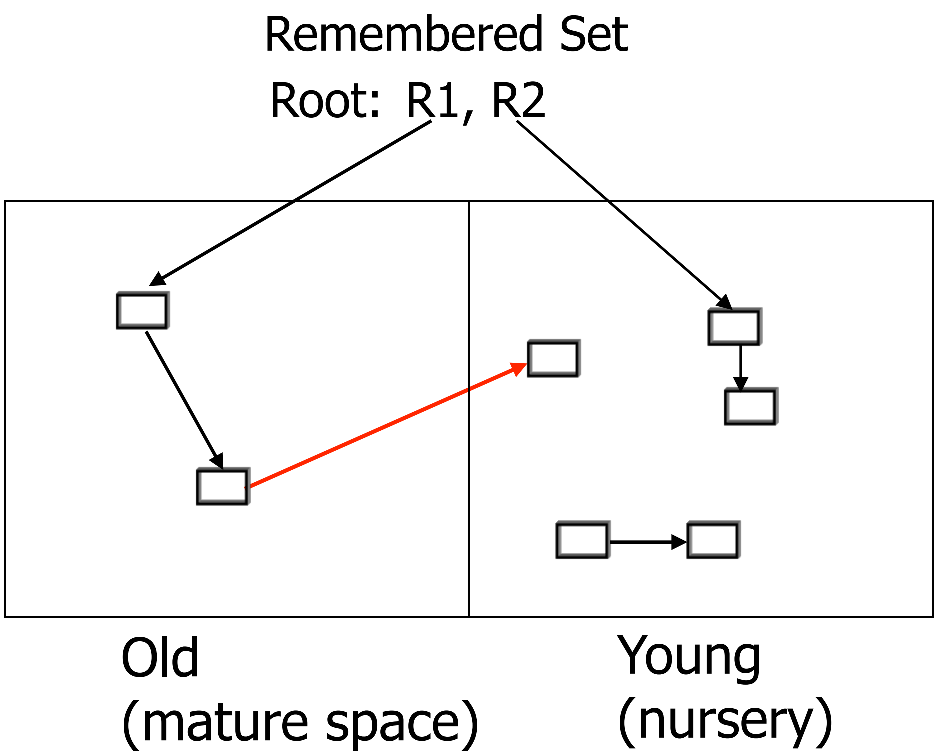
Live Object
Dead Object
Mark reachable objects
MATURE
COPY
NURSERY

Full GC: Sweep



Generational Collection

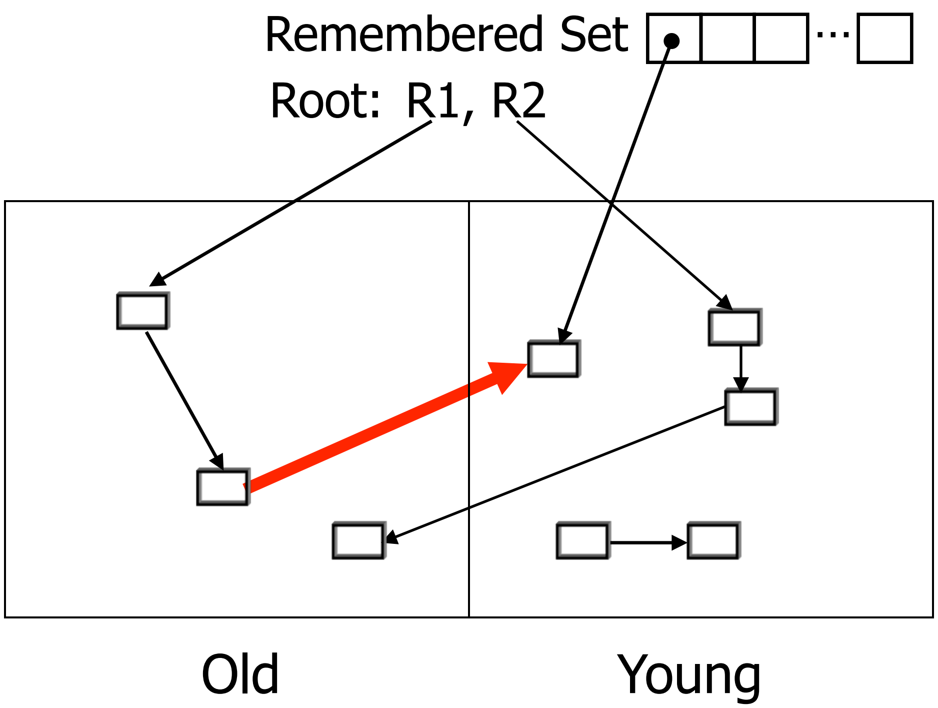
- Minor Collection must be **independent** of major
  - Need to remember old-to-young references
  - Usually not too many – mutations to old objects are infrequent



- Write Barrier
  - Check pointer stores
  - Remember source object
  - Source object is root for minor GC

Generational Collection

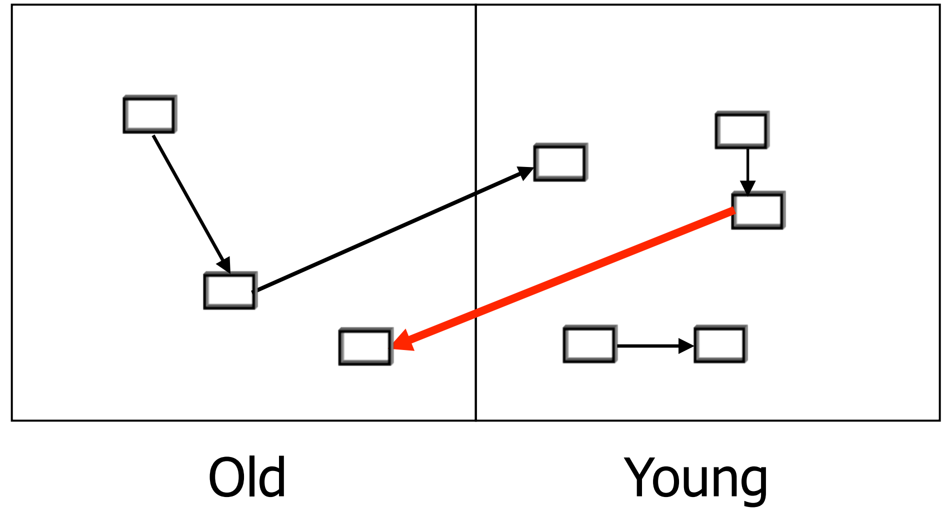
- Minor Collection must be independent of major
  - Need to remember old-to-young references
  - Usually not too many – mutations to old objects are infrequent



- Write Barrier
  - Check pointer stores
  - Remember source object
  - Source object is root for minor GC

Generational GC

- What about young-to-old?



Generational GC

- What about young-to-old?
  - We don’t need to worry about them if we always collect the young each time we collect the old (**major collection**)

- Write barriers
  - Catching old-to-young pointers
  - Code that puts old-generation object into a remembered set
    - Traversed as part of root set
    - All field assignments aka POINTER UPDATES IN YOUR CODE!
- Alternative to write barriers
  - Check all old objects to see if they point to a nursery object
  - Will negate any benefit we get from generational GC

Generational GC Considerations

- Mutator cost is added
  - Proportional to the number of pointer stores

- When does an object become old?
  - Make old early - too much garbage sitting in OldSpace
  - Make old late - spend too much time copying objects back and forth thinking that its about to die
  - **Pig in the snake problem**
- How should each space be collected?
  - Nursery objects copied upon first minor collection
  - Mature space(s)
    - Mark/sweep
    - Copying

Generational Copying Collector

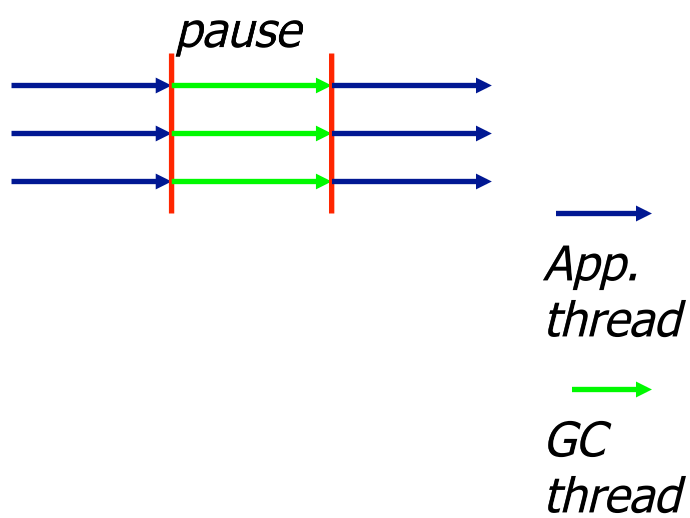
- Cost proportional to root set size + size of live objects
  - You should be able to state the cost of each type of GC
- Pig in the snake problem
  - Relatively longlived objects (together make up a large portion of the heap) become garbage all at once
  - Will be copied repeatedly & until space for it is found
  - Increases traversal cost at every generation
  - Favors fast advancement of large object clusters
- Questions
  - More generations?
  - How big should they be?
  - How can we make things more efficient?

Advanced GC Topics

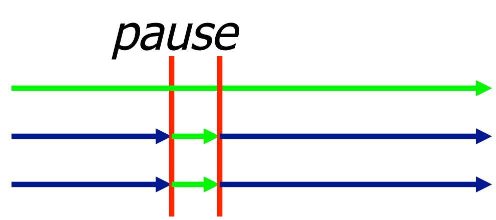
- Parallel collection
- Concurrent collection

Parallel/Concurrent Garbage Collection

- **Parallel** – multi-threaded collection (scalability on SMP/multi-core)
  - GC still stop-the-world



- **Concurrent** – unlike **stop-the-world** (STW), background collection (short pauses through resource over-provisioning)



Advanced GC Topics

- Parallel collection
- Concurrent collection
- Sun HotSpot (OpenJDK) GC
  - Generational mark-sweep/compact
  - Eden: where objects are allocated via bump pointer
    - When full, live objects copied to To space
  - To: half of nursery;   From: half of nursery
    - Flip spaces here 2-3 times (parameter setting)
  - Mature space: Mark-sweep with region-based compaction
- Advanced GC
  - Hybrid (region-based) collection: Immix
  - Partner with operating system: Mapping Collector