Memory Management

Torben Mogensen

Programming Language Design

Learning Goals for this Lecture

Knowledge of

- Different types of memory management, their properties and limitations
- How different programming languages use memory management and how this has influenced their design
- Methods for implementing manual memory management
- Methods for implementing automatic memory management, including reference counting and tracing garbage collection

Abstractions of Memory

Memory on a modern computer is rather complex: It consists of registers, cache, RAM, and disk/flash storage. By a mixture of hardware and operating system, this is usually abstracted to a large, linear address space, but we might want even higher abstraction levels in programming languages, for example:

- A programmer might not care at what address an object is allocated.
- A programmer may not care exactly how much space is used for an object.
- A programmer may not want to explicitly free the memory used for an object when the object is no longer used, and she may not care exactly when the freeing happens.

Here, "object" is a record, an array, a closure, or similar block of data that resides in memory.

The Most Common Memory Abstractions

Static allocation: The size and location of an object is determined at compile time, and the memory used for the object is not freed until the program ends.

Stack allocation: Objects are allocated and freed by moving a stack pointer, and accessed by addressing relatively to a stack pointer or frame pointer.

Heap allocation: Object allocation and freeing are entirely decoupled. We distinguish different forms of heap allocation by the mechanism for freeing the space used for objects:

- **Manual:** Explicit in program text and (usually) immediate.
- **Automatic:** Implicit and possibly delayed until more space is required.

Static Allocation

Address and size of objects fixed at compile time. Sole allocation mechanism in FORTRAN and COBOL.

- Space is allocated when program starts and not freed until end of execution.
- Statically bounded memory implies language is not Turing complete, unless external storage is used.
- Prevents (unbounded) recursion, especially if return addresses are statically allocated (FORTRAN and COBOL).
- Limited reuse of memory, and mostly explicit (such as by `COMMON` clauses in FORTRAN).

Partial solution to allow variable object size and reuse of memory: Allocate large objects in virtual memory with plenty of space between. Will only use as much physical memory as needed, and will reuse the same physical pages.

Stack Allocation

Objects allocated at top of stack, and freed in opposite order. Sole allocation mechanism in ALGOL 60.

- Size of object need not be known until time of allocation (but it can usually not be extended after allocation).
- Allows unbounded memory (language can be Turing complete).
- Allows unrestricted recursion (until stack memory runs out).
- Reuses memory, usually implicit in program structure (call/return).
- Objects may stay in memory long after last use, as they are not freed until procedure return (LIFO behaviour).
- Freeing is forced at procedure return, so a function should not return pointers to objects allocated inside the function.

Heap Allocation (Dynamic Allocation)

Life times are decoupled from program structure. Introduced in LISP.

- Can allow freeing exactly when object is no longer needed (but freeing can be delayed until any later time).
- Can allow resizing after allocation (requires indirection or redirection of pointers).
- Can allow pointers to be passed around and returned freely.
- Can allow non-stack-like calls and returns (continuations and concurrency).

Two variants: Explicit (manual) and implicit (automatic) freeing. Explicit freeing is used in languages like C and Pascal, implicit freeing is used in LISP, Simula, Java, and most other functional or OO languages. Automatic freeing is often called *garbage collection*.

Before You Continue

Consider the following questions:

- Can you name an object-oriented programming language that does *not* use automatic memory management (garbage collection)?
- Infinite (non-tail) recursion will eventually cause an error, as there is no more stack space available. Have you ever run out of stack space when running a program that uses finite (but deep) recursion? Is so, how did you solve this problem?

Manual Memory Management

Objects are explicitly allocated and explicitly freed. In C by using functions `malloc()` and `free()`, in Pascal by keywords `new` and `dispose`. In C++ also by calling object constructors and destructors.

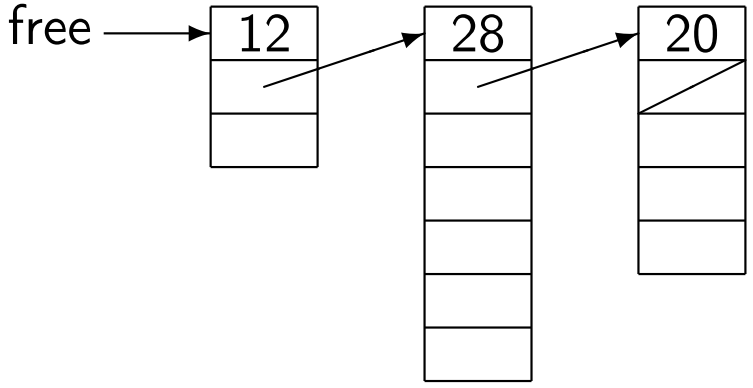
- Memory can be allocated from OS in large chunks (pages), but must be freed in opposite order (stack discipline).
- So a *memory manager* manages a (possible extensible) number of memory pages to allow decoupled allocation/deallocation and smaller object sizes.
- Allocation and freeing usually (slightly) slower than for stack allocation.
- Lifetime of objects independent of lifetime of other objects.
- The programmer must keep track of life times – otherwise space leaks or premature freeing of objects can occur.

Simple Manual MM Using a Free List

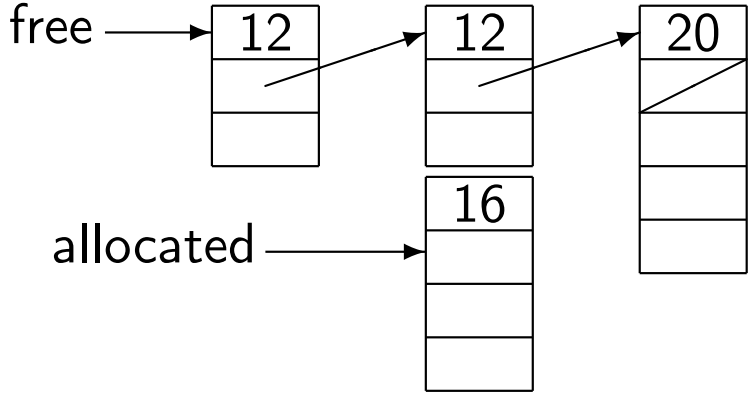
A simple memory manager can:

- Maintain a linked list (called *the free list*) of free blocks, initialised to a single large block.
- Each blocks holds a size field and a pointer to the next free block in the list (not necessarily previous and next by address).
- When allocating, slice off block of sufficient size from any sufficiently large block in the free list, and return remainder of block to free list (if the remainder is large enough to hold a size field and a pointer).
- Allocated blocks need to have a size field, but not a next pointer.
- When freeing a block, add it to the head of the free list.

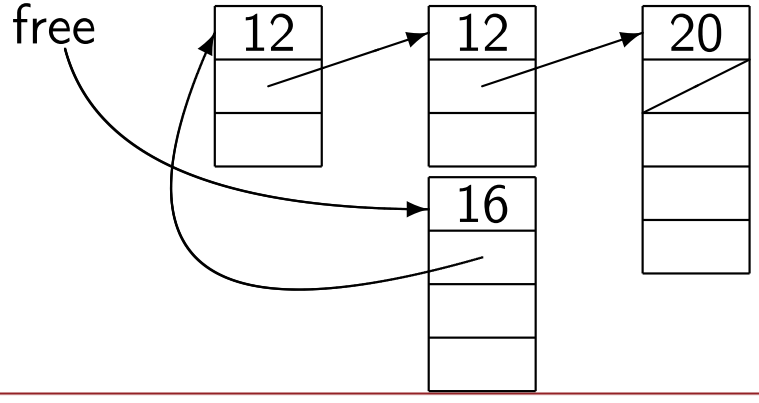
Example (Four-Byte Words)



(a) The initial free list.



(b) After allocating 12 bytes.



(c) After freeing the same 12 bytes.

Issues With This Method

- Have to search for large-enough block, can be O(n).
- Since blocks are never joined, you accumulate a lot of small blocks (fragmentation).
- If object is freed before last use, it can be overwritten. The programmer must make sure this does not happen.
- If object is kept alive long after last use, space can "leak". Again, the programmer must avoid this.

Partial solutions to the first two problems:

- Separate free lists (or sorted tree) for different size blocks – speeds search
- Split larger block only if none found of (near) right size
- Join memory-adjacent blocks when freed – reduces fragmentation

Common strategy (Buddy system, see notes for more details):

- Power-of-two block sizes, aligned to size
- Split only in half
- Join only same-size neighbouring blocks, and only if combined block is aligned

Reduces allocation time from $O(n)$ to $O(\log n)$, but increases freeing time from $O(1)$ to $O(\log n)$. Does not fully prevent fragmentation, but helps.

Before You Continue

- Solve Exercise 5.1 from the notes.
- The notes mention a method for handling very large objects (several memory pages). Who is credited for this method?

Automatic Memory Management

An object is automatically freed *at some time* after the last time it is used (can be much later).

Problem: Last use is undecidable.

Solution: Use *safe approximation* to last use, usually reachability from program variables: If an object can not be reached from any program variable, it can definitely no longer be used.

The converse is not true: Reachability does not imply later use, so reachability is only an approximation of later use.

Limitation: For reachability, we need to know to which object a pointer points, so pointers can not point *into* objects, only to their headers – unless they point *both* to the header and into the object (fat pointers). Alternatively: Keep a global bitmap of object headers / non-headers and search backwards to find header.

How to Detect (Un)Reachability?

Two main methods:

Reference counting: Count number of incoming pointers to object. When zero, object is unreachable and can be freed.

Tracing collection: Make a graph traversal (spanning tree) from *roots*. A root is a variable in the program (stored in a register, on the stack, or statically allocated). All nodes visited by a graph traversal from these roots are reachable, all others can be freed.

Reference Counting

Every heap-allocated object has a field that counts incoming pointers (in addition to size field).

- When object is allocated, counter is set to 1.
- Before a pointer is moved ($\mathtt{p} \leftarrow \mathtt{q}$), the counter in the object that p points to is decreased by 1, and the counter in the object that q points to is increased by 1.
- If counter is decreased to 0, the object is freed (by putting it on free list).
  Note: Pointers *from* freed objects must be followed, and their targets must decrease their pointers. This has two implications:
  - Freeing a large object is no longer constant time.
  - We must be able to distinguish pointer fields and non-pointer fields in freed objects.

Note: Reference counting can not handle cyclic data structures!

Distinguishing Pointers and Non-Pointers

There are several possible ways to do determine if a field is a pointer or not:

- Compile-time type of object determines field types, so if we know type when freeing, we can use this information.
- Header of object contains (pointer to) descriptor, that identifies fields as pointers or non-pointers. Descriptor can be a bit-vector or a *finaliser* method.
- Every field value uses a bit to mark it as pointer or non-pointer. For example by using the least significant bit of every word:
  - Even values are word-aligned heap pointers.
  - Odd values represent non-pointers. Numbers are shifted one bit left and incremented (so 5 is represented as $2 \times 5 + 1 = 11$).

  Alternatively, bit vector outside heap indicates pointer/non-pointer for all words in heap.

Tracing Collection

Introduced with LISP in 1958.

Reachability is determined by a graph traversal from the *root set*, marking the visited nodes (objects) as reachable.

This requires that root pointers must be known. Can use same methods as for identifying fields as pointers/non-pointers.

When tracing is complete, unmarked objects are unreachable and can be freed (e.g., by putting them on free list).

Tracing collection handles cyclic structures just fine (node is marked as reachable before its children are traced).

Tricolour Tracing

Maintain classification of nodes (heap objects) using "tricolour" invariant:

White: The node has not (yet) been visited by the graph traversal.

Grey: The node has been visited, but some children may not have been.

Black: The node and all its children have been visited.

Algorithm:

1. Initially, root set (not on heap) is grey, and all heap nodes are white.
2. Repeat while there are grey nodes:
   - Pick a grey node, colour its white children grey, and colour the node itself black.
3. When there are no more grey nodes, all reachable nodes are black, so white nodes are unreachable and can be freed.

Mark-Sweep Collection

Bit in block header indicates colour: 0 $\sim$ white, 1 $\sim$ grey or black.
Grey nodes are kept on (recursion) stack, black nodes are not.

Mark phase: Depth-first traversal of graph.

```
mark(p):
   if ↑p is white then
      mark ↑p (to grey)
      for all pointer-fields q of p do
         mark(q)
      /* ↑p is now black, so we return */
```

Can be implemented using explicit stack instead of recursion (see notes).

Sweep phase: Add white nodes to free list.

```
sweep():
   for all blocks b in heap do
      if b is white then add b to free list
      else clear mark to white
```

Before You Continue

- Compare the `mark` procedure shown above with the method from the notes that uses an explicit stack. What are the advantages/disadvantages of using an explicit stack for this?

Two-Space Collection

Does not use free list, but allocates objects from contiguous space (like in stack allocation).

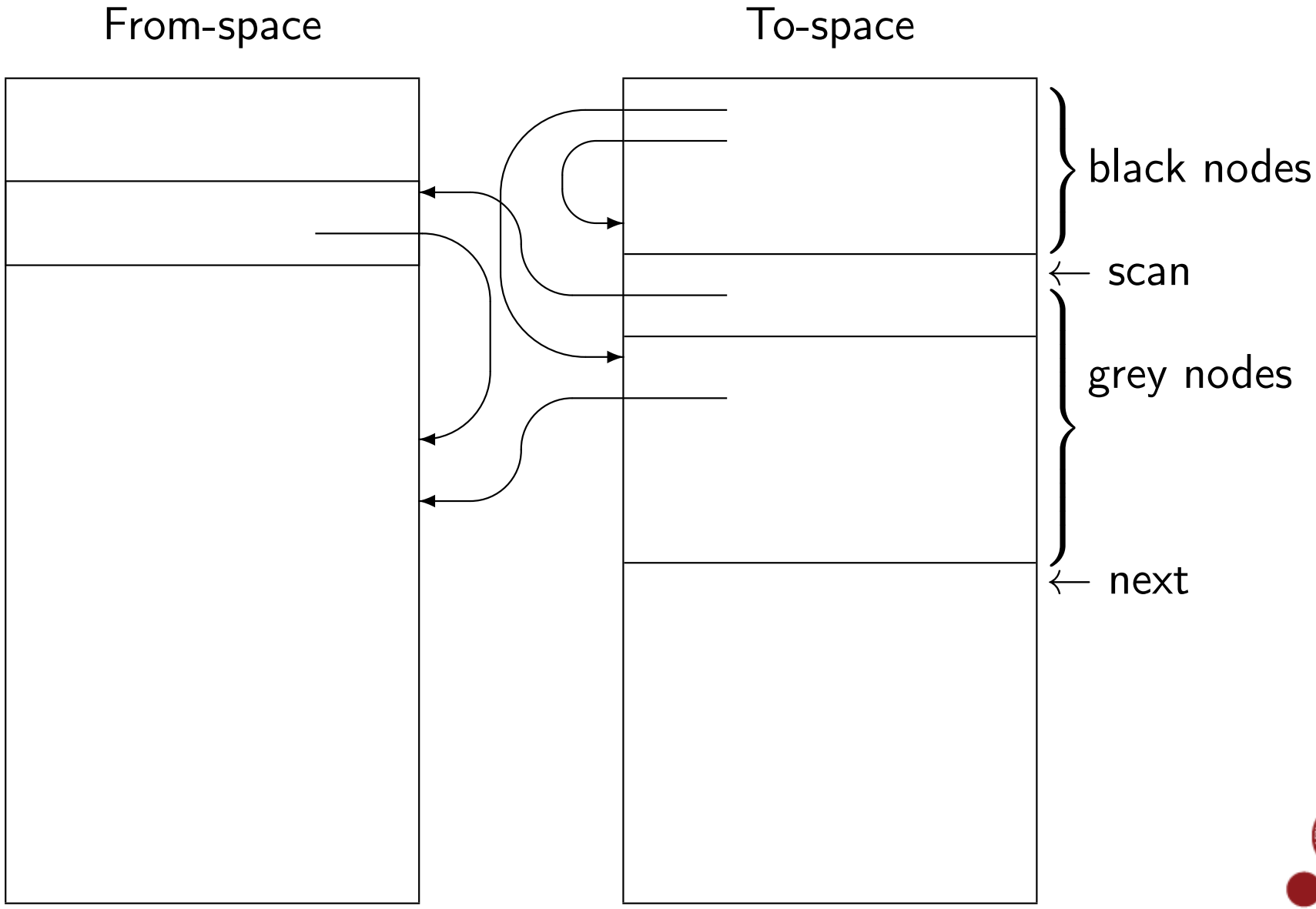
Avoids fragmentation, but needs two heap spaces of same size.

Algorithm: Breath-first copying of reachable nodes from one space (*the from-space*) to the other (*the to-space*).

After collection from-space contains nothing useful. Allocation is now done from to-space.

Next collection switches the two spaces – Analogy: Having two dish washers.

Two-Space Collection Illustrated

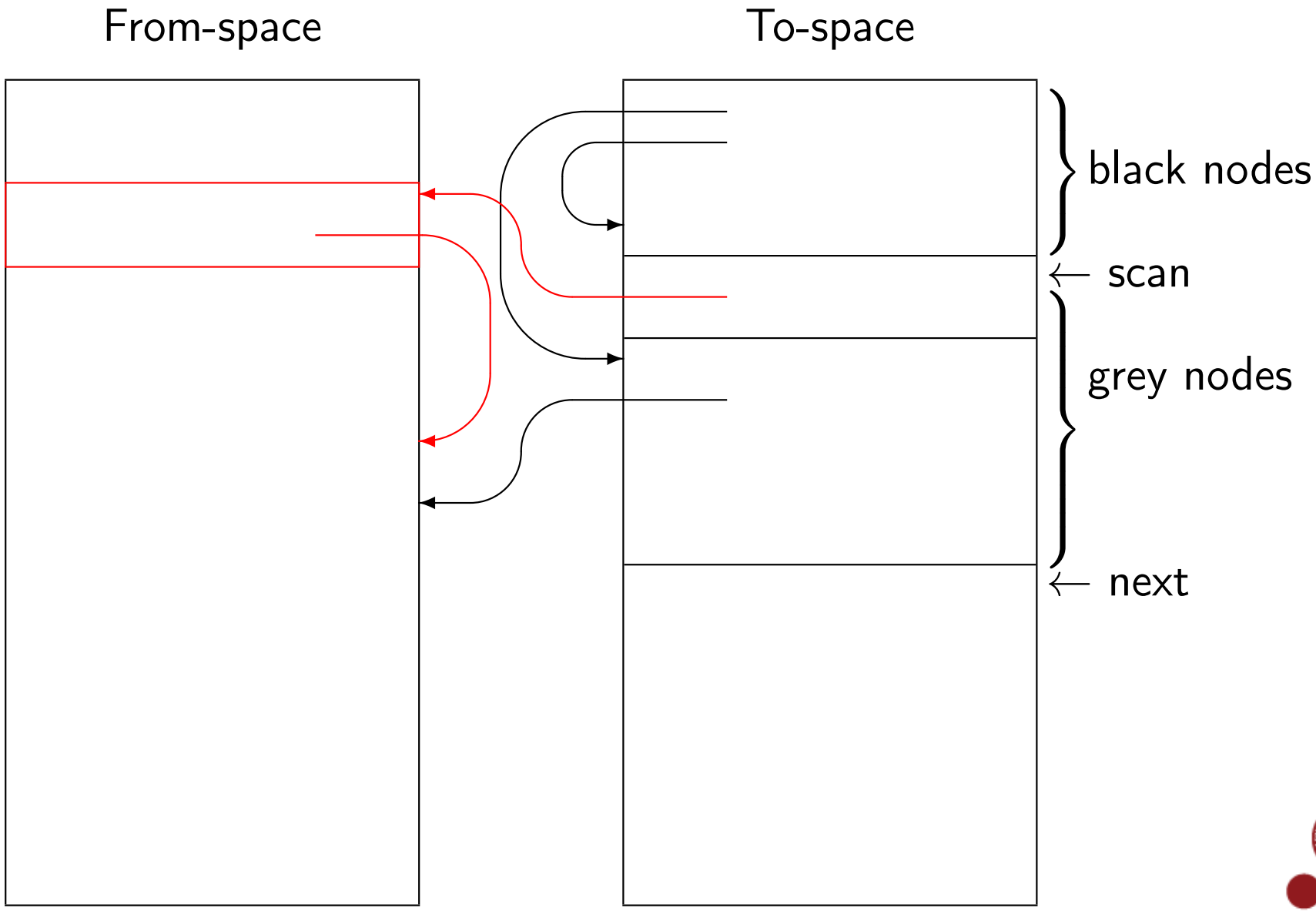


Forwarding Pointers

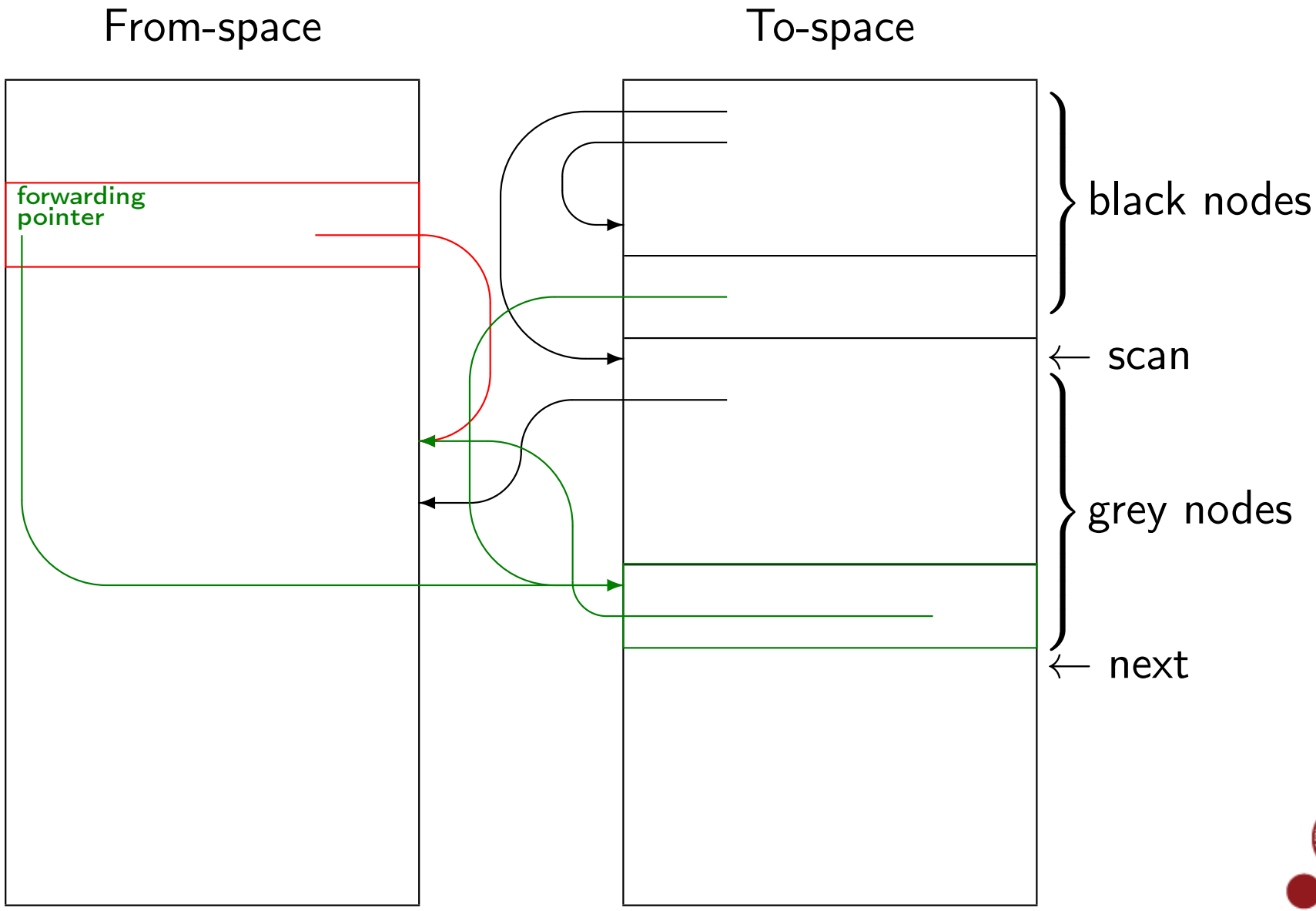
- If an object has already been copied to to-space, the first word of the object (that usually holds the size) is replaced by a pointer to its new location. Size fields and forwarding pointers can be distinguised by using odd machine words for sizes.
- When the scan pointer moves forward, it checks whether the current field f holds a pointer p to from-space. If so, it *forwards* the object that p points to.
  - If the first field of that object is a forwarding pointer to a location q in to-space, store q in f.
  - If not, copy the object to the space after the next pointer, store next in f and the first field af p. Then add the size of the object (now stored in the field at next) to next.

The notes have a very detaied example, we show a simpler example on the next slides.
Garbage collection ends when scan = next.

Forwarding a Node (Before)



Forwarding a Node (After)



Comparing Mark-Sweep and Copying Collection

Mark-sweep	Copying collection
Objects do not move	Objects move at GC
Small space overhead	large space overhead
Both internal and external fragmentation	No fragmentation
Time: O(all objects)	Time: O(live objects)

Mark-sweep best if:

- Few and large objects with long lifetimes.

Copying collection best if:

- Many and small objects with short lifetimes.

Generational Collection

In copying collection, long-lived objects are copied many times.

To avoid this, *generational collection* can be used:

- Several spaces of increasing size are used.
- Objects are copied to next-larger space.
- Allocation (mostly) in smallest space.

Effects:

- Small space(s) collected frequently, larger space(s) less so.
- Long-lived objects copied fewer times, as they are moved to larger (and less frequently collected) areas.
- Complication: Must handle pointers from large spaces to smaller spaces (remembered sets).

Incremental and Concurrent Collection

Program effectively pauses while GC is done, which is bad for reactive programs.

To avoid this, *incremental or concurrent* collection can help:

- Every time an object is allocated, GC does a limited amount of work.
- In best case, collection keeps up with allocation, so no long pauses.

Must preserve tricolour invariant: No black node can point to a white node. Use *barriers:*

- If a pointer is updated, colour source or target object grey, *or*
- If a pointer is fetched, colour target object grey.

These barriers must be inserted by the compiler (or done by the interpreter).

Memory Management and Language Design

- Recursion requires either stack or heap. Deep recursion requires heap.
- Unrestricted closures require heap allocation and automatic memory management. If you restrict closures to be function parameters, but not results, a stack suffices (Pascal).
- Is the GC allowed to change the values (addresses) of pointers? If not, you can't use copying collection.
- Can data structures be cyclic? If so, you can't use reference counting (unless you add *weak pointers*).
- Is very fast reaction to events required? Use manual memory management, reference count (with small modification), or incremental collection. Alternatively, do GC when it is known that no fast reaction is required in the immediate future.

Before You Finish

Consider the following:

- In some languages, all heap-allocated objects are the same size. What does this imply for fragmentation (both internal and external)?