PC 04: Graph traversal & garbage collection

Based on the PC notes “Parcours de graphes et glanage des cellules” by François Pottier
Memory allocation as a graph problem

- Garbage collector in Java takes care of memory reallocation when not used
- Variables: roots or graphs connecting to other memory locations with pointers
- Naturally represented by graphs
Q1: how can we map the program heap to a directed graph a tree?

• The heaps can be considered as a directed graph:

  • Nodes:
    – The root variables
    – The objects that are created with the *new* predicate

  • Edges
    – Root variable \( r \), object \( o \): \( r == o \)
    – Objects \( o_1, o_2 \), there is a reference from a field \( f \) of \( o_1 \) to \( o_2 \): \( o_1.f == o_2 \)

• An object \( o \) is *alive* if there is a path from a root variable to \( o \)
Garbage collection by scanning and marking (mark-sweep)

- Assume the program variables are the roots.
- **Object**: unit of storage on the heap.
- **Object/Reference graph**: directional graph of objects in memory
- Nodes are objects in memory and the edges (arrows) are references of one object to another
- Circular references and self referencing is feasible
- **Roots**: set of nodes in the object graph from which the references start.
- **Unreachable object**: nodes with no edge referencing them.
- **These nodes are cleaned by the GC**

Some nodes are not reachable – therefore the relevant memory nodes will be “swept” and given back to the OS.
Marking and sweeping

• Marking: The memory graph is parsed and all reachable objects are marked (i.e. with a bit $= 1$)
• Sweeping: All the non marked objects (i.e. not reachable) are swept and freed for further use by the OS.
• Q2: we can achieve this by graph traversal
Q3: are all the useless objects necessarily dead?

- All dead (i.e. non referenced objects) are useless

- The contrary is not true:
  
  ```java
  Object x = new Object ();
  f();
  System.out.println(x)
  ```

- Object x is useless if f() does not terminate.

- This is generally not captured by a GC algorithm
Garbage collection

• Mark phase:
  – Start from the root variables and traverse the graph
  – Each object visited is marked by a “mark” bit (0/1)
Garbage collection - Q4

• marks are clearly used to transmit information (ie. all living variables) of the marking phase to the scanning phase. However, they play also essential role for the execution of the marking phase. Which one?

• the marks are used to ensure that each edge of the graph is traversed at most once

• the graph is traversed in linear time, even if
  – some objects are shared (i.e. accessible through multiple paths)
  – the graph contains cycles
Q 5

• Why use one only two "colors" (here, 0 and 1) rather than using three (black / gray / white) and in the course?

• The use of gray can detect the presence of a cycle.

• can be useful in some applications, but not here. Cycles between pointers of heap objects are not interesting here

• Use just two colors, which has the advantage of requiring just one bit.
Marking phase - Question 6
depth first search in the heap

• Write the pseudo code for a recursive procedure of marking, DFS (o), which marks the object o and its descendants. Think how to treat the case where o is the null pointer. Then write the pseudo code a procedure MARK (), which implements the mark phase of GC.

--------------------------
procédure DFS(o)
if (o is not null) and (o is not marked)
    mark o
    for each field o.f
        DFS(o.f)
Question 6 - continued

• For the GC to be correct, it is necessary and sufficient that all objects reachable from the roots, and only these, are marked at the end of the course.

• The mark phase of GC is a simple graph traversal from all roots. We assume all objects initially unmarked. The pseudo-code is as follows:

```plaintext
procedure MARK()
  for each root r containing the address of an object o
    DFS(o)
```

• We assume the list of GC roots is known.
Marking - Q7

• Prove that the pseudo code above ends.
• Determine the complexity of the marking phase.

marks remain until the end of the marking phase thus the number of tagged objects will therefore increasing during this phase bounded by the number of existing objects, which is fixed, Thus algorithm terminates!

• Complexity
  – \( r \) : \#roots, \( m \) : \#objects marked.
  – count how many times DFS is called, including itself.
Calls to MARK for each DFS root and recursive call from DFS itself for each field of an object newly marked.
The total number of calls to the DFS for marking phase is \( O(r) + O(m) \),
  - the time required to complete the marking phase is \( O(m) + O(r) \).
Also, note that the parameter \( m \) is only the marked objects, thus objects reachable from the roots (see question 8).

• Non referenced objects are not marked, thus they do not contribute to the complexity.
Q8.

• Prove that only objects reachable from the roots will be marked.
Q 9: Give a non recursive solution to the marking problem

Procedure MARK()
for each root r containing a pointer to object o
push o to the stack.
While stack is not empty
do
    pull pointer o
    if o is not null
    then
        mark o
        for each field o.f push o.f to the stack
Q 9: Give a non recursive solution to the marking problem - variant

To ensure that
• No “null” is marked and
• each node is stacked only once

Procedure consider(o)
if o (is not null) and (not marked) Then
    mark o; push o to stack

Procedure MARK ()
for each root r containing a pointer to object o
    CONSIDER (o)
While stack is not empty do
    pull pointer o
    for each field o.f CONSIDER(o.f)

- Size of the stack can reach \( n \), where \( n \) is the number of objects in the program heap.
- The space needed for the stack is \( \sim 3N \) (each variable has at least 2 fields-pointers)
Q11: Sweeping and maintaining free lists

- distinguish between allocated object and free zones
- In free zones:
  - store in the first word of, a value distinguishing it from the header of an allocated object - for example 0, and to ensure that the header of an object is never 0.
  - The size of each free zone can be stored, in the second word in the zone.
  - Free zones can be maintained by building a singly-linked list of free zones (free list) - a pointer to Free next,
  - the third word of each area can be used to free it.
- The address of the first element of the free list will be stored in a global variable.
- Free list will allow three main operations:
  - IsEmpty,
  - ALLOCATE, searches a free area and turns it into an allocated object
  - FREE, which transforms an object allocated in a free area and (possibly) merges it with or the free areas adjacent.

- Once all allocated objects are marked the sweeping phase consists in traversing linearly the heap and adding the non marked ones to the free list and re initializing for the next marking phase.
Q12: pseudo code for the sweeping phase

// a: address of an allocated object or a free zone
procédure SWEEP()
a = first node in the heap
While a < end of the heap do
    if a is an allocated object then
        if a is marked
            remove marking
        else
            FREE (a)
    a= a+size(a)
(then call allocate to allocate memory from the free list)

Function NEW(object o)

// if there is no free space
If ISEMPTY()
    // Try to recover some
    MARK()
    SWEEP()
    if ISEMPTY()
        return failure
    else
        return ALLOCATE(o)

// If the free list is not empty
else
    // Assume all objects of the same size – this is why no parameters.
    return ALLOCATE(o)
Q14: How do we treat mark-sweep for objects of variable sizes?

• Fragmentation (sequence of allocated and free memory)
• Large object allocation may fail
• Consecutive allocations may allocate memory parts quite far from each other
• Compact free space to put allocated / free memory in contiguous space.
Q15: if the objects fields do not contain only pointers but also arbitrary values how can this be treated by the GC?

- Java GC knows how to distinguish among pointers and integers
- Integers of size 31 bits (instead of 32 bit pointers) use the 1 remaining bit for distinguishing (Caml)
- Approximate distinguishing...