Simple GC Algorithms

see web pages for links

Reference counting

- Every object keeps a reference count
- Add a pointer to the object: increment count
- Delete a pointer: decrement count
- Count reaches zero: dispose object
-
- Problems: extra space, overhead of each operation, cycles cannot be reclaimed

Mark-Sweep collection

- Two phases:
  - MARK live objects (starting from a root-set)
  - SWEEP non-marked space into free-list(s)
- Problems:
  - Fragmentation
  - Cost proportional to heap-size (SWEEP)
  - Locality of reference

Mark-Compact collection

- Similar MARK phase as before, but instead of SWEEP:
- COMPACT: move live objects “down” until contiguous
- PRO: allocation like a stack (==> cheap)
- CONS: still multiple phases, objects move (i.e. addresses change),
- NOTE: does not actually collect garbage, but extracts “live objects”

Copying GC

- Split into 2 spaces, allocate from one space only until full
- Simple stop-and-copy:
  - Copy all reachable objects from “old” into “new” space,
  - Only one phase: traversal-and-copy in one, but need “forwarding pointers”
- Pro: simpler, fast allocation
- Cons: like mark-and-compact, plusneed to set aside half the memory

Generational GC

- Fact: most objects die young
  - ==> split space into “nursery” and one or more older generations
  - More frequent collections in younger generations
  - Objects surviving one or more collections in one generation are promoted to older generations
- Pro: reduce redundant copying, closer to real-time
- Cons: more complicated, e.g. need to keep track of old-to-new pointers,

Non-copying implicit collection

- Inspired by copying GC: spaces are a special case of sets:
  - Move live objects from set “old” into set “new”
  - Different implementation for this sets where it is cheap to move an object between sets, and switch sets
- Non-copying: doubly-linked list (i.e. 2 additional pointers / object) plus “color”
- “Copy”: unlink from one list, toggle color, link into other list
- CONS: extra space (but..), fragmentation
- PRO: no copying, no pointer updates

Conservative GC

- Simplify work by not detecting all garbage
- e.g. Assume all references from the stack to alive
- Most collectors are
- But some MUST, like addons for C++:
  everything that might be a pointer is treated as one ==> some garbage will be kept indefinitely

GC-related language features

- Weak pointers
- Finalisation
- Multiple differently-managed heaps
- Compile-time GC

Total GC cost

- Lots of old studies with not up-to-date assumptions,
- Estimate: 10 % overhead in runtime, 100% overhead in memory
- Active area of research:
  - Incremental algorithms
  - Alternatives like “region-based allocation”, etc.