Garbage Collection

Garbage Collection

- Garbage collection automatically frees storage which is not used by the program any more.
- Has two phases:
  - Garbage detection — finds which objects are alive and which dead;
  - Garbage reclamation — deallocates dead objects.
- Liveness is a global semantic property which is unsolvable in general.
- Garbage collection uses an approximation: an object is alive if it's reachable from the root set; otherwise it's dead.

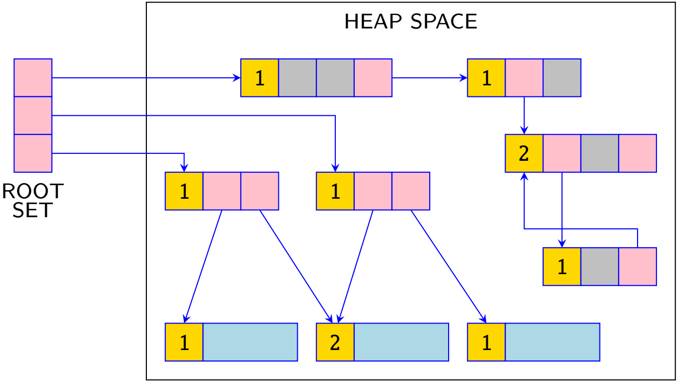
Reference-Counting

Reference-Counting

- Each object has a counter which keeps track the number of references to the object.
- Counter is modified when references to the object are added/deleted:
  - counter is incremented on adding a new reference;
  - counter is decremented on deletion of a reference.
- If counter is zero, then the object is freed:
  - the object is inserted into the free list;
  - all its outgoing pointers are deleted.

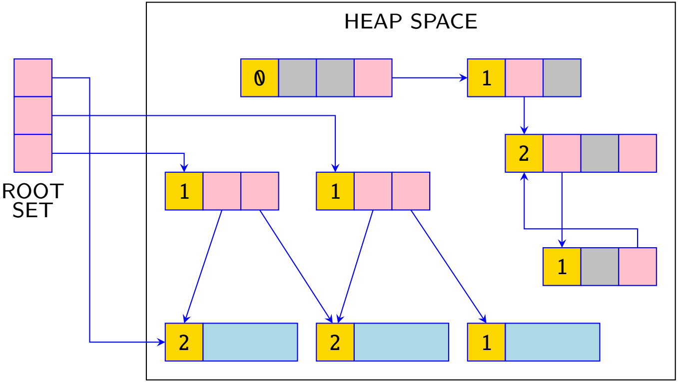
Reference-Counting

Example:



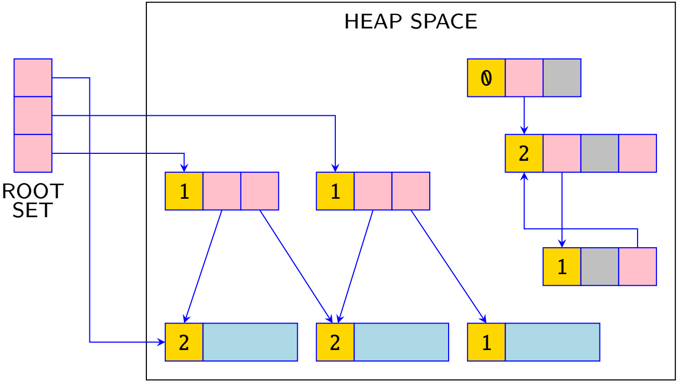
Reference-Counting

Example:



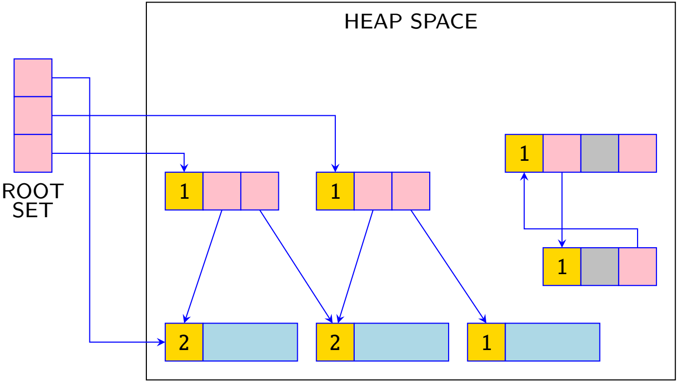
Reference-Counting

Example:



Reference-Counting

Example:



Reference-Counting

Advantages

- ✔ simple to implement;
- ✔ activities related to garbage collection are distributed:
  - relatively easy to make it incremental;
- ✔ good locality:
  - modifies only counters of source and target references;
- ✔ minimal zombie time (time between the object becoming a garbage and its reclamation);
- ✔ allows easy implementation of object finalization.

Reference-Counting

Drawbacks

- ✘ relatively inefficient:
  - must manage counters even when there is no garbage;
- ✘ memory fragmentation:
  - analogous to other free list based methods;
- ✘ if there are many small objects, may require substantial amount of memory for counters;
- ✘ the complexity of recursive deallocation is in worst case bounded by size of the heap;
- ✘ is unable to reclaim all garbage:
  - cyclic data structures.

Mark-Sweep garbage collection

Mark-Sweep

Has two phases:

1. starting from roots, mark all reachable objects;
2. scan over the heap and free all objects which are not marked.

```
void gc () {
  foreach x ∈ Roots do
    mark (x);
  end;
  collect ();
}
```

Mark-Sweep garbage collection

Procedure mark()

- Marks the given node and then recursively marks all nodes reachable from it.
- Recursion stops when the node is already marked or if the node contains only primitive values (no pointers).

```
void mark (ref x) {
  if (x→mark == 0) {
    x→mark = 1;
    foreach y ∈ sons(x) do
      mark (y);
    end;
  }
}
```

Mark-Sweep garbage collection

Procedure collect()

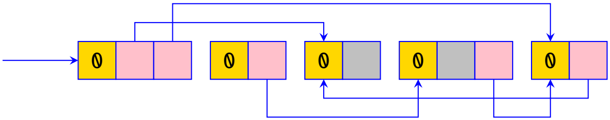
- Performs a full scan over the heap and puts all unmarked objects into the free list.

```
void collect () {
  freelist = NIL;
  foreach x ∈ objects() do
    if (x→mark == 0) {
      x→next = freelist;
      freelist = x;
    }
    else x→mark = 0;
  end;
}
```

Mark-Sweep garbage collection

Example:

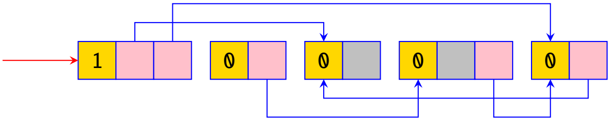
1. Recursive marking:
2. Collecting the garbage:



Mark-Sweep garbage collection

Example:

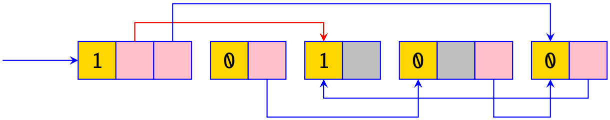
1. Recursive marking:
2. Collecting the garbage:



Mark-Sweep garbage collection

Example:

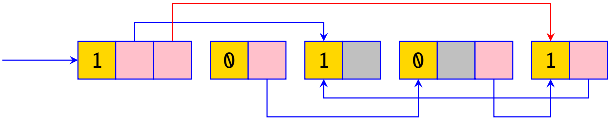
1. Recursive marking:
2. Collecting the garbage:



Mark-Sweep garbage collection

Example:

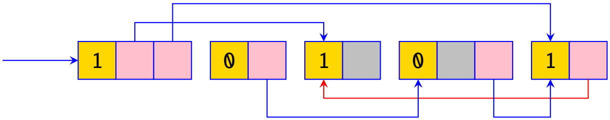
1. Recursive marking:
2. Collecting the garbage:



Mark-Sweep garbage collection

Example:

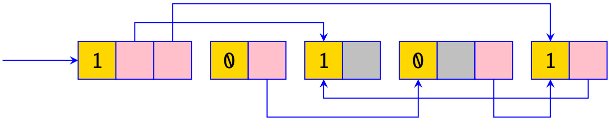
1. Recursive marking:
2. Collecting the garbage:



Mark-Sweep garbage collection

Example:

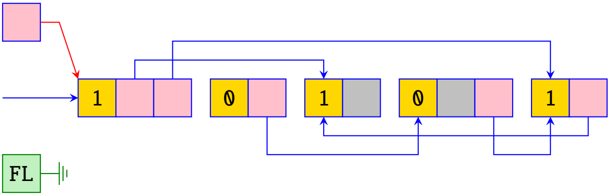
1. Recursive marking:
2. Collecting the garbage:



Mark-Sweep garbage collection

Example:

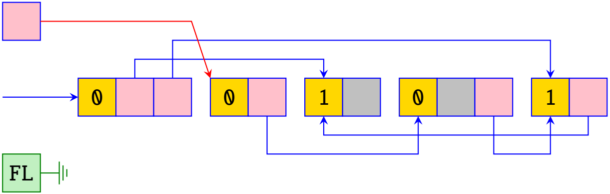
1. Recursive marking:
2. Collecting the garbage:



Mark-Sweep garbage collection

Example:

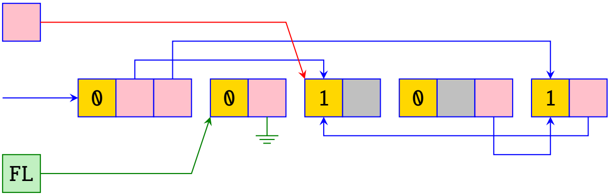
1. Recursive marking:
2. Collecting the garbage:



Mark-Sweep garbage collection

Example:

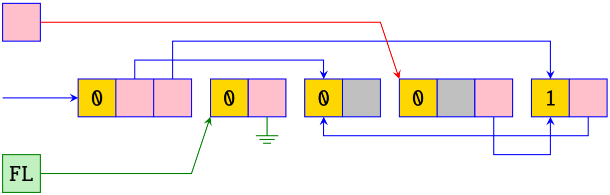
1. Recursive marking:
2. Collecting the garbage:



Mark-Sweep garbage collection

Example:

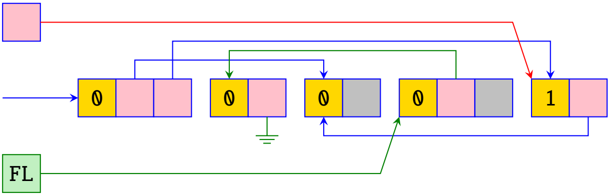
1. Recursive marking:
2. Collecting the garbage:



Mark-Sweep garbage collection

Example:

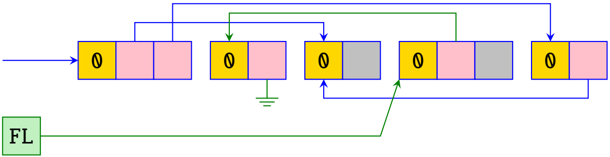
1. Recursive marking:
2. Collecting the garbage:



Mark-Sweep garbage collection

Example:

1. Recursive marking:
2. Collecting the garbage:



Mark-Sweep garbage collection

Drawbacks

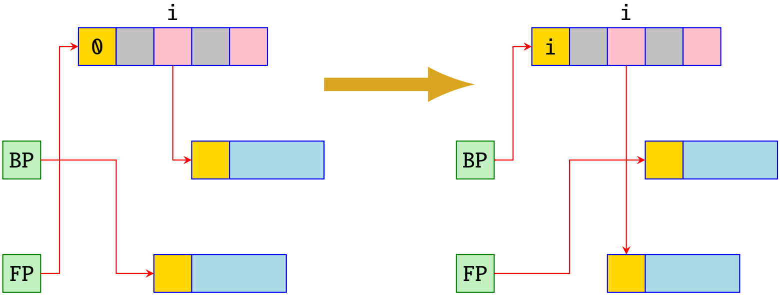
- ✘ Marking is recursive.
  - In the worst case, size of the recursion stack is linear to size of the heap!!
  - Possible solution: Deutsch-Schorr-Waite pointer reversal algorithm.
- ✘ Live objects are mixed with free heap areas.
  - Memory fragmentation.
  - Possible solution: Mark-Compact garbage collection.

Pointer Reversal

Deutsch-Schorr-Waite algorithm

```
void mark (ref x) {
  FP = x; BP = NIL;
  while (FP→mark ≠ -1 || BP ≠ NIL) {
    if (FP→mark == 0) {
      FP→mark = i = nextidx(FP);
      if (i ≠ -1) {
        tmp = FP; FP = tmp[i];
        tmp[i] = BP; BP = tmp;
      }
    } else { // FP→mark ≠ 0
    ...
    }
  }
}
```

Pointer Reversal

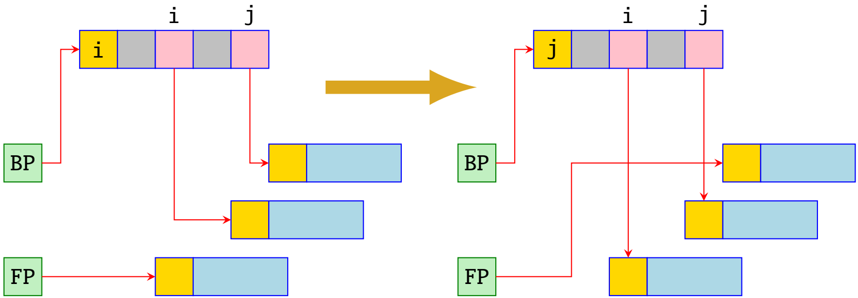


Pointer Reversal

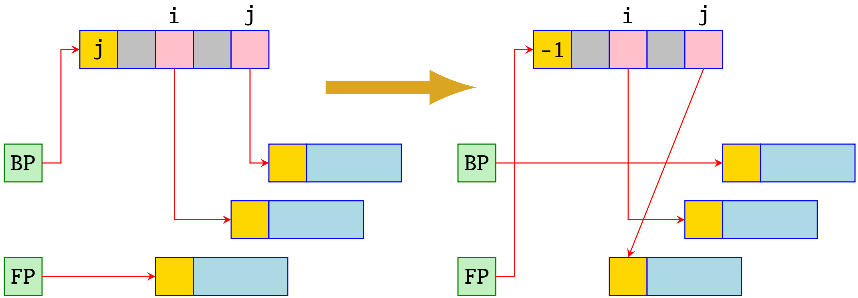
Deutsch-Schorr-Waite algorithm

```
...
} else { // FP→mark ≠ 0
  i = nextidx(BP);
  if (i ≠ -1) {
    tmp = FP; FP = BP[i]; BP[i] = BP[BP→mark];
    BP[BP→mark] = tmp; BP→mark = i;
  } else {
    tmp = FP; FP = BP; BP = FP[FP→mark];
    FP[FP→mark] = tmp; FP→mark = i;
  }
}
...
```

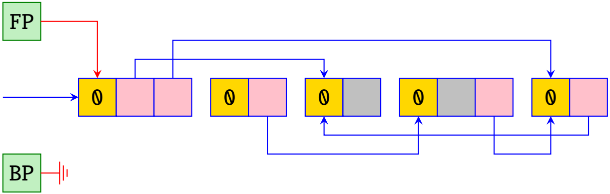
Pointer Reversal



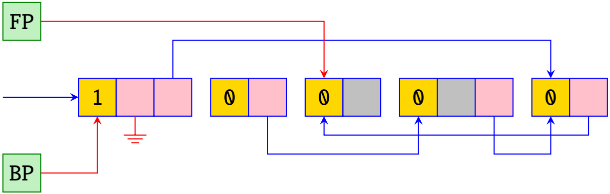
Pointer Reversal



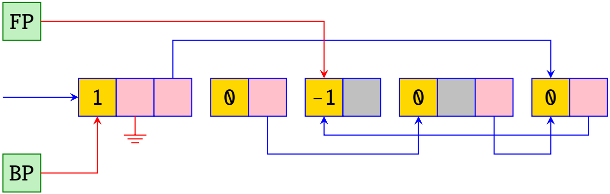
Pointer Reversal



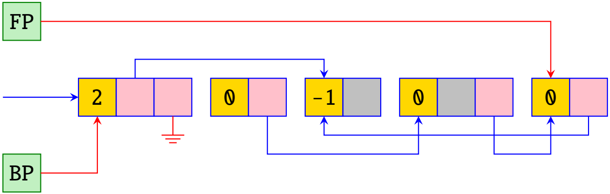
Pointer Reversal



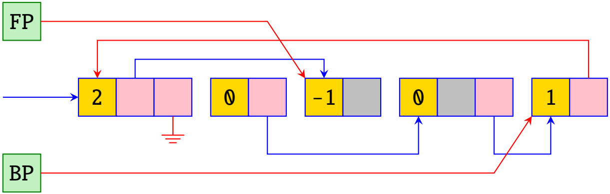
Pointer Reversal



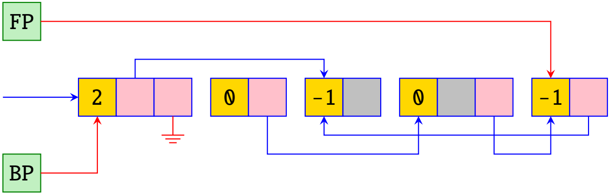
Pointer Reversal



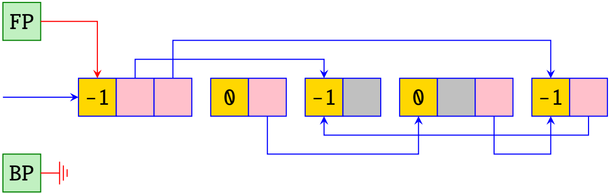
Pointer Reversal



Pointer Reversal



Pointer Reversal



Mark-Compact garbage collection

Mark-Compact

Has three phases:

1. starting from roots, mark all reachable objects (similarly for Mark-Sweep);
2. perform full scan of the heap and compute new addresses for marked objects;
3. move marked objects to their new locations and change pointers accordingly.

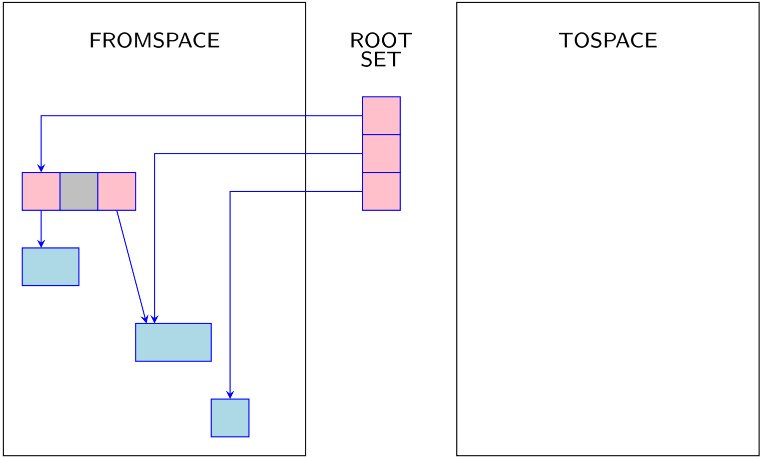
- ✔ At the end of the garbage collection all free memory forms a single compact region in the heap.
- ✘ Relatively inefficient, as it requires several scans over the heap.

Copying garbage collection

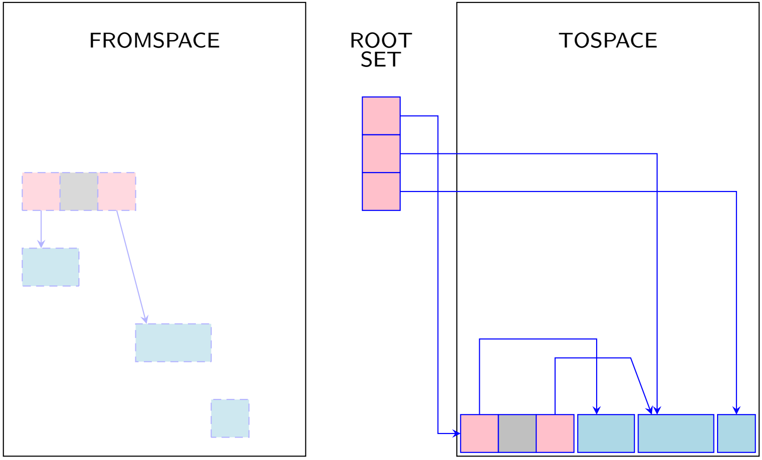
Copying

- The heap is divided into two equal subregions: FromSpace and ToSpace.
- FromSpace is a currently active memory region to where allocated objects are saved.
- Garbage collection is invoked when FromSpace becomes full:
  - live objects are copied from FromSpace to ToSpace;
  - FromSpace and ToSpace flip the roles (ie. former ToSpace becomes FromSpace and vice versa).

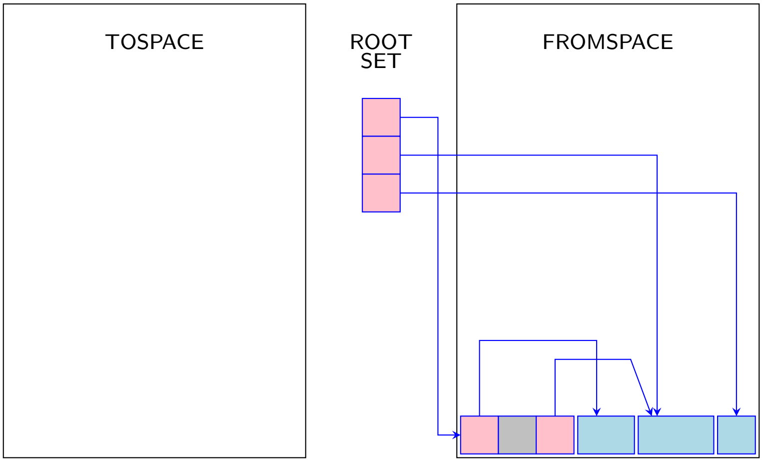
Copying garbage collection



Copying garbage collection



Copying garbage collection



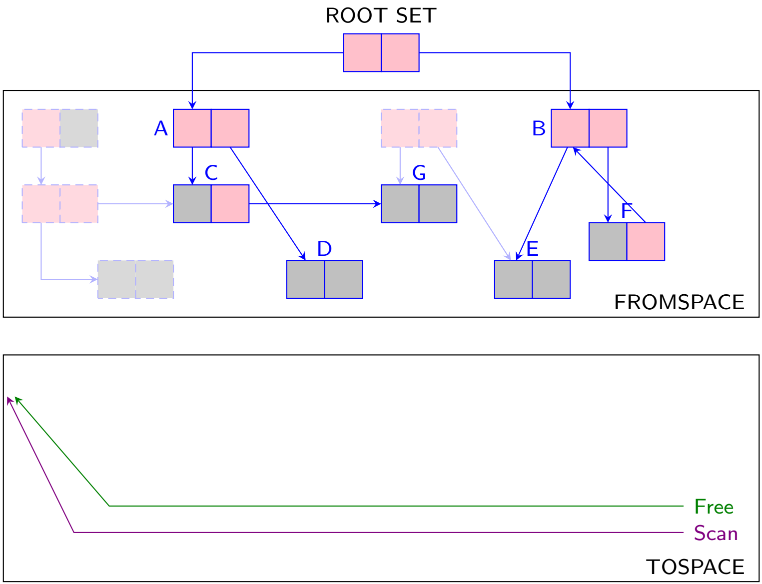
Copying garbage collection

Cheney's algorithm

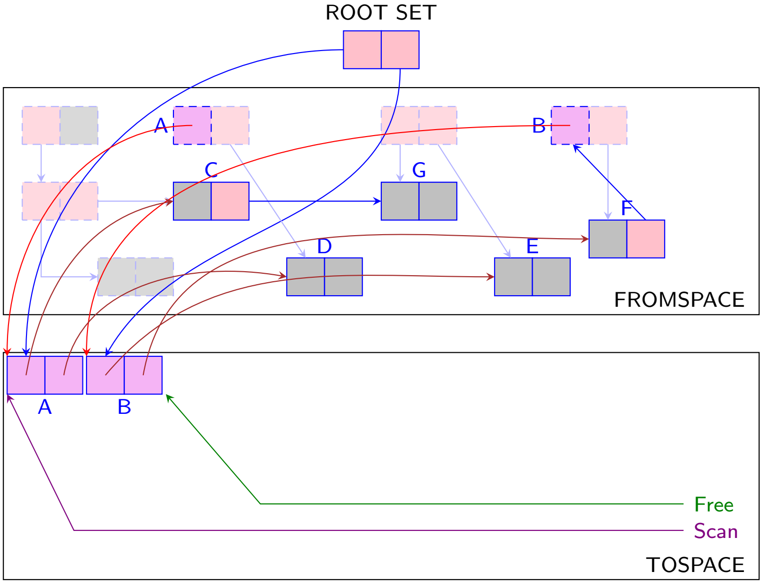
Has two (interchanging) phases:

- the first phase (evacuate) copies all directly reachable objects from FromSpace to ToSpace, replaces used pointers by the ones pointing to the new corresponding objects, and installs forwarding pointers in places of the evacuated objects;
- the second phase (scavenge) linearly scans the objects copied into ToSpace and all objects (in FromSpace) directly reachable from them are evacuated; if the object has already been evacuated before, then it is not copied again but the pointer to it is replaced by the forwarding pointer;
- the process ends, when all objects in ToSpace are scanned.

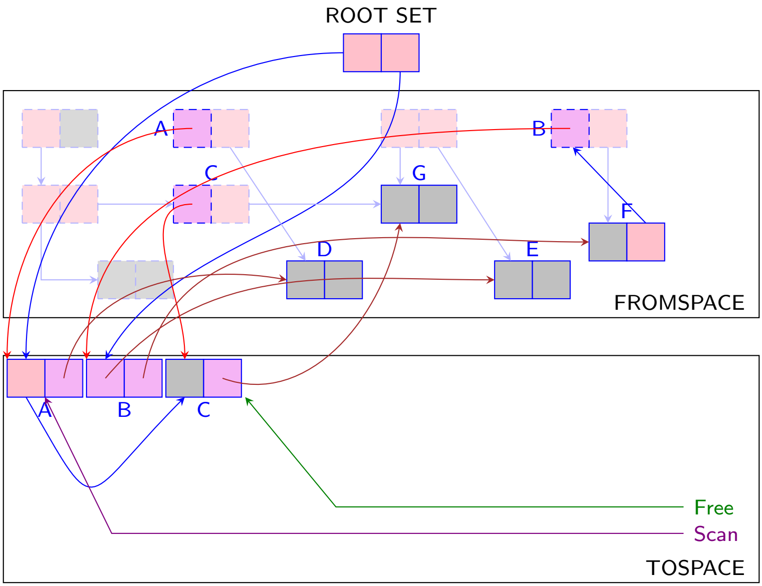
Copying garbage collection



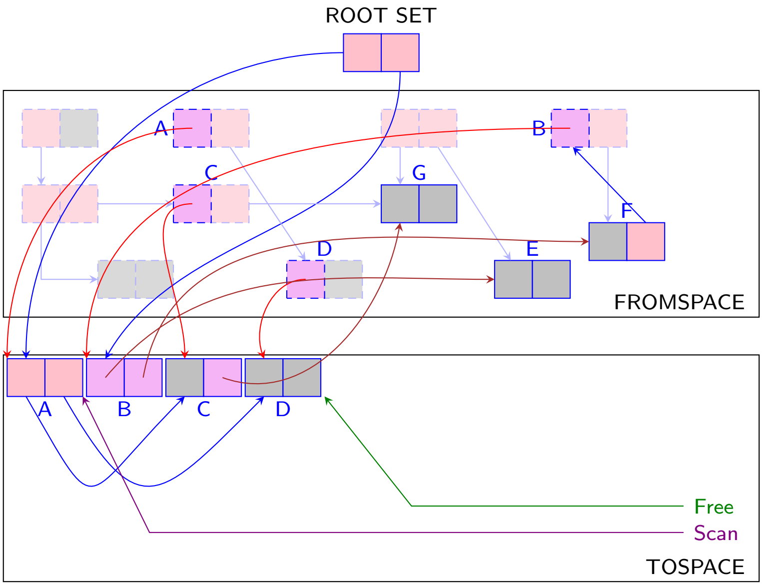
Copying garbage collection



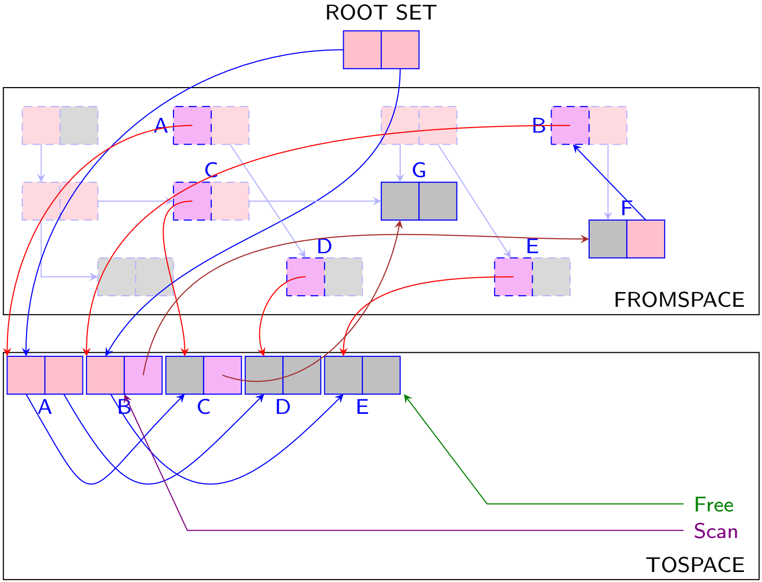
Copying garbage collection



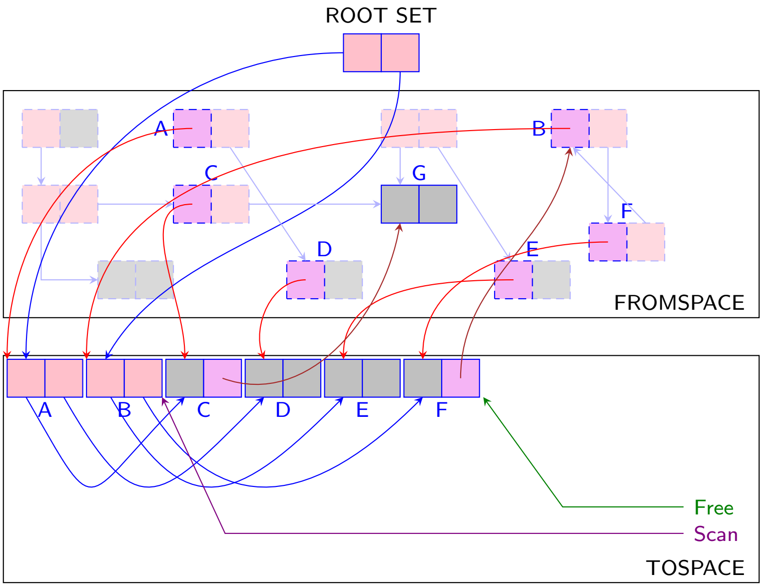
Copying garbage collection



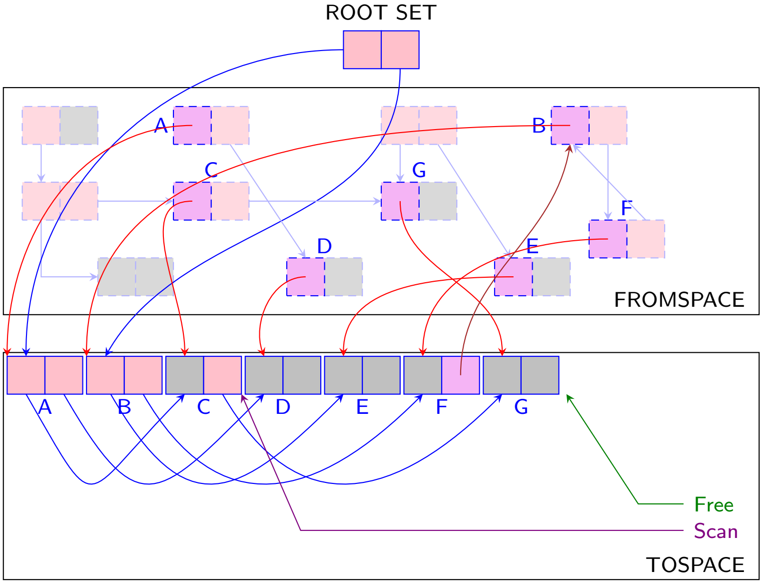
Copying garbage collection



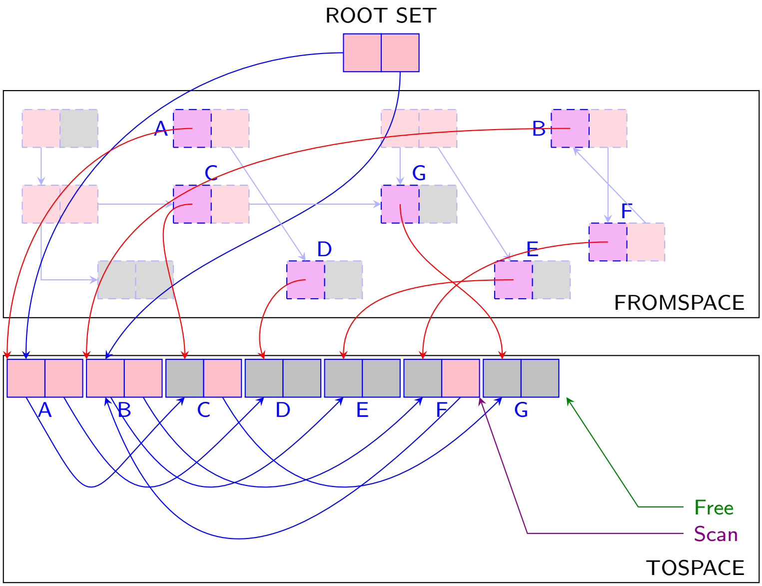
Copying garbage collection



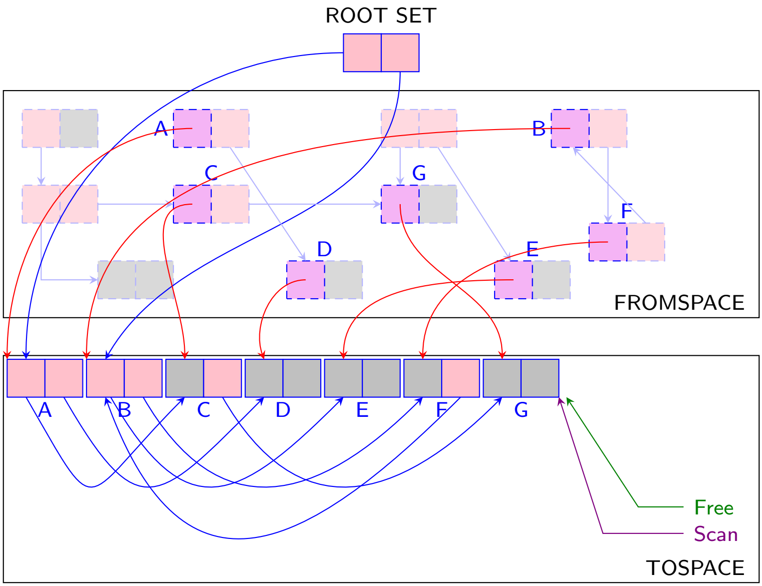
Copying garbage collection



Copying garbage collection



Copying garbage collection



Copying garbage collection

Advantages

- ✔ all free memory is in a single compact region;
- ✔ object creation is very cheap:
  - memory allocation is an incrementation of the heap pointer by the object size;
  - checking of the heap exhaustion is a comparison of two pointers;
- ✔ only live objects are inspected:
  - most objects have relatively short life span;
  - hence, usually there are much less live objects than garbage;
- ✔ theoretical amortized efficiency is very good:
  - on increase of the heap size, the cost of copying will near to zero!

Copying garbage collection

Drawbacks

- ✗ the whole work is concentrated to the garbage collection time:
  - might result for annoying pauses;
- ✗ breath-first traversal may mix locality patterns;
- ✗ all pointers are rearranged:
  - might invalidate some invariants the program is assuming;
- ✗ half of the memory is "useless";
- ✗ objects with long life span are copied over and over again:
  - might be quite costly if "veteran" objects are large.

Generational garbage collection

Empirical facts

- Infant mortality – most objects have very short life span. Usually 80-90% objects die before the next megabyte is used:
  - 60-90% CL and 75-95% Haskell objects die before getting "10 kb old".
  - SML/NJ frees 98% of objects during each garbage collection.
  - 95% of Java objects are "short-lived".
- The older the object, the more probable that it survives the next garbage collection.
- Directionality of reference – usually younger objects point to the older ones.

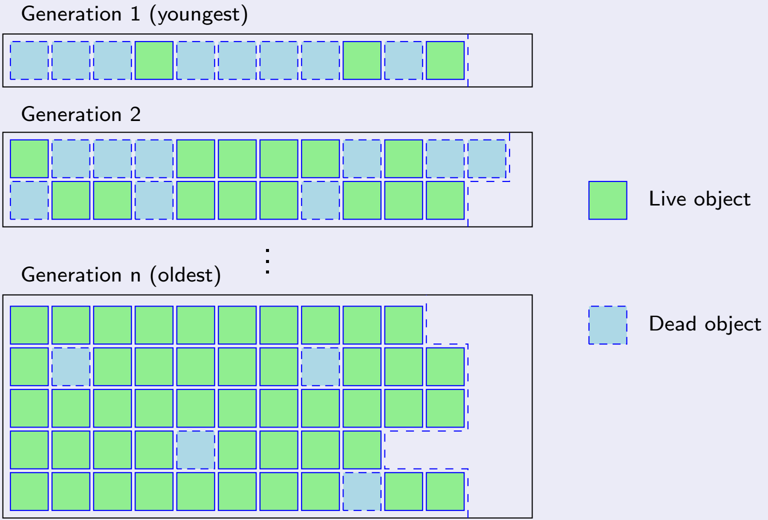
Generational garbage collection

Generational garbage collection

- Memory is divided by the age of objects living there into generations.
- The number and size of different generations is usually fixed beforehand.
- New objects (infants) are created into the youngest generation (nursery).
- When alive objects get older (tenure) they are promoted to the next generation.
- Garbage collections of different generations are done in different frequencies
  - most frequently in the youngest generation.

Generational garbage collection

Memory division into generations



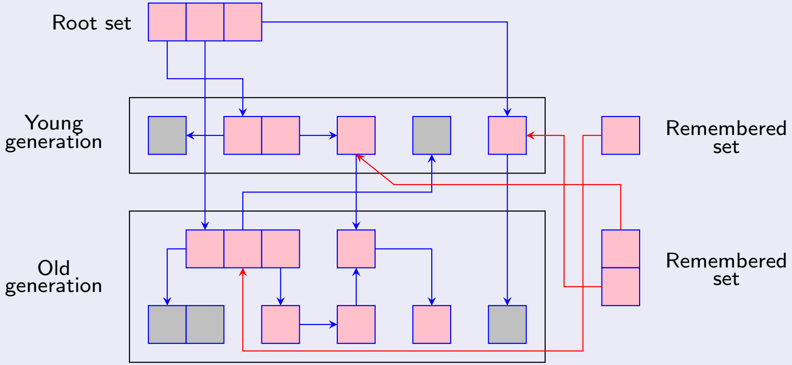
Generational garbage collection

Remembered sets

- In addition to "normal" roots, the given generation may have outside pointers from other generations.
- Their locations can't be determined statically .
- Dynamically searching possible roots from other generations during garbage collection is very costly.
- Hence, each generation has a corresponding remembered set, which contains references from other generations
  - if there is a pointer from one generation to another, then the reference is added into the remembered set of the target generation.

Generational garbage collection

Remembered sets



Generational garbage collection

Problem

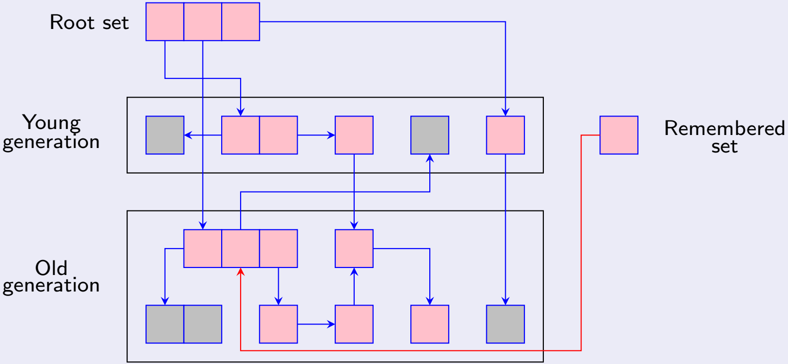
- Remembered sets may require a significant amount of memory
  - all intergeneration dependencies must be recorded.
- Remembered sets must be maintained during the program execution which may be very costly
  - each pointer variable may potentially be intergenerational.

Solution

- Record in remembered sets only references from the older generation to younger ones
  - in case of two generations, only one remembered set (for the nursery) is needed.
- Use approximate remembered sets.

Generational garbage collection

One-way remembered sets



Generational garbage collection

Remembered sets

- Pointers *from an older to a younger generation* are roots for the younger generation:
  - such pointers are relatively infrequent;
  - they may be created only by destructively updating a pointer in a tenure object;
  - such assignments are caught by write barriers.
- Pointers *from a younger to an older generation* are frequent:
  - not a problem, if garbage collection of the older generation always collects also the younger one.

Generational garbage collection

Generational garbage collection

- Usually there are just two generations and the younger one is relatively small.
- Normally, garbage collection performs only a minor collection which:
  - removes garbage only from the nursery;
  - old enough objects are promoted to the tenured space.
- When the tenured space is exhausted, a major collection is performed; ie. garbage is collected from both generations.
- Minor and major collections may use different garbage collection methods (eg. minor uses copying and major uses mark-compact).

Generational garbage collection

Issues

- Minor collections doesn't remove garbage in the tenured space:
  - all young objects pointed by a tenured garbage will remain uncollected (nepotism).
- How old must be an object before promoting?
  - One minor collection is not enough, as objects created just before the collection haven't yet had time to die.
  - Usually, two minor collections is considered to be enough.
- How large should be the nursery?
  - Must fit into the main memory.
  - Too big may result to too long minor collection pauses.
  - Too small doesn't give enough time for young objects to die.

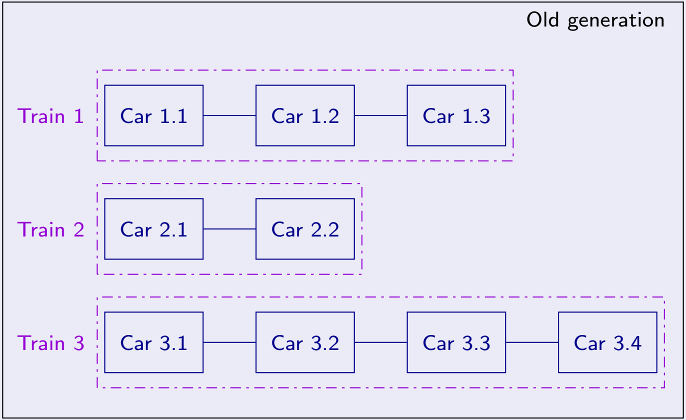
Generational garbage collection

Train algorithm

- Major collection may result to too long pauses for interactive programs.
- Train algorithm by Hudson and Moss uses incremental collection for the old generation.
- The tenured space is divided into cars:
  - each car has its own remembered set;
  - only one care is collected at once.
- As substructures may live in different cars, the cars are grouped into trains:
  - the aim is to accumulate related data structures into one train.

Generational garbage collection

Train algorithm — division of the tenured space



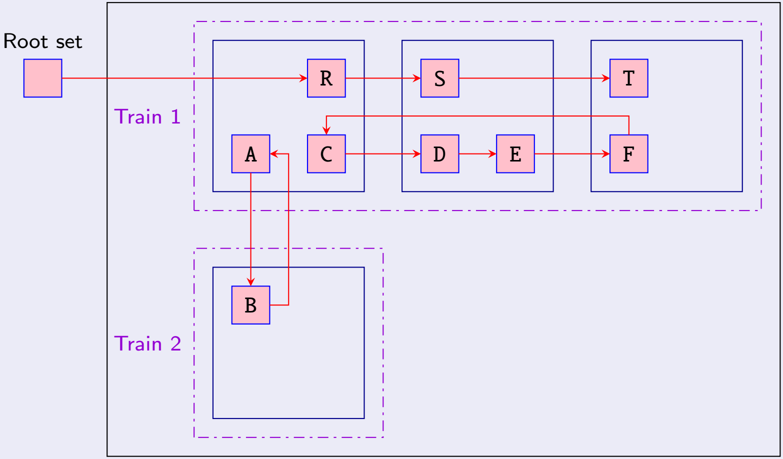
Generational garbage collection

Train algorithm

- Each call of the algorithm frees the first car (FromCar) of the first train (FromTrain).
- If FromTrain doesn't have any outside pointers to it, the whole train will be freed.
- Otherwise, the objects in FromCar pointed from other trains are evacuated into these trains; objects pointed from other generations are evacuated into some other (may be completely new) train.
- Remaining outside pointers of FromCar are from other cars of FromTrain; corresponding objects are evacuated into the last car of FromTrain (creating a new car if necessary), after which FromCar is freed.

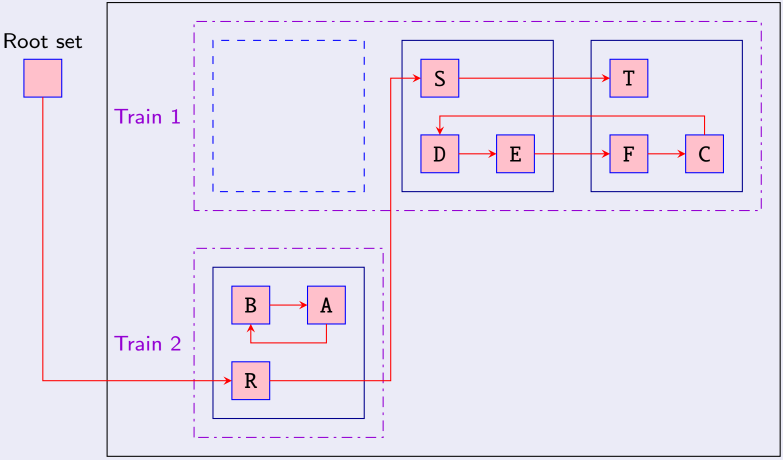
Generational garbage collection

Train algorithm — the initial state



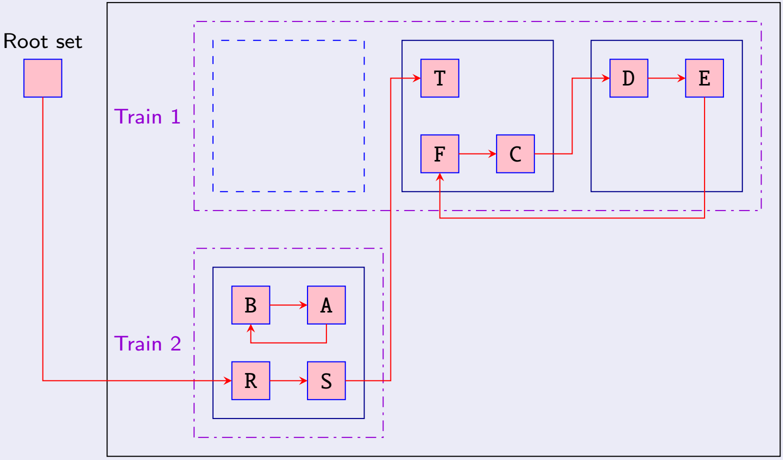
Generational garbage collection

Train algorithm — the state after the first collection



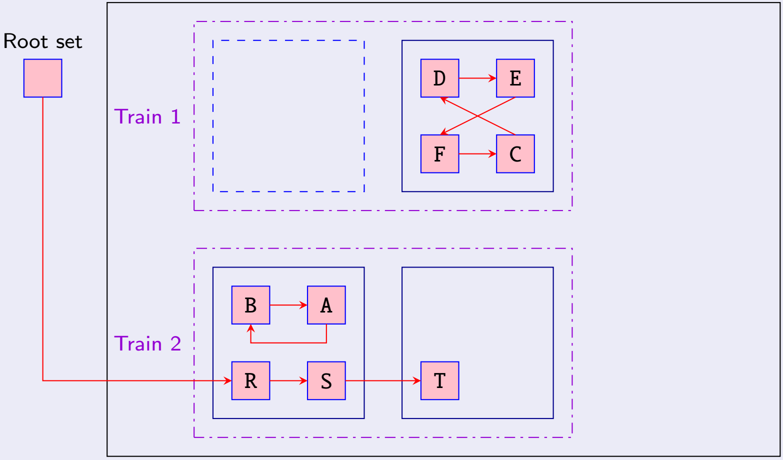
Generational garbage collection

Train algorithm — the state after the second collection



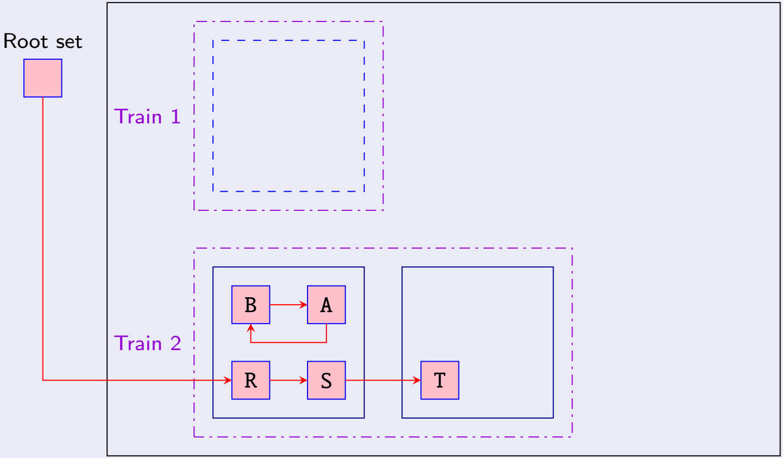
Generational garbage collection

Train algorithm — the state after the third collection



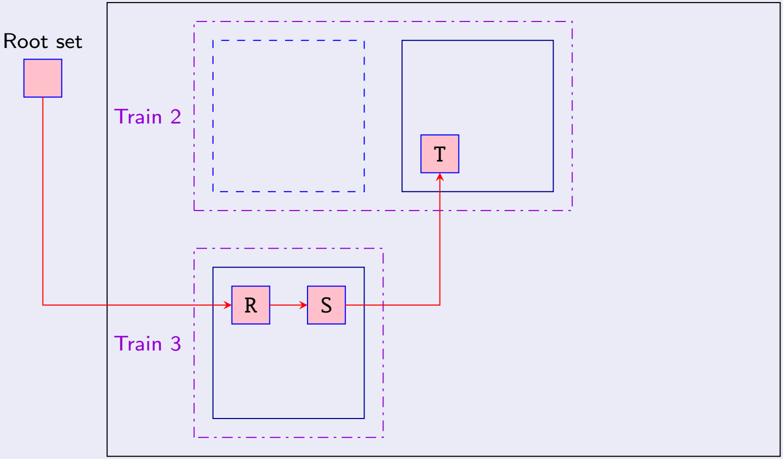
Generational garbage collection

Train algorithm — the state after the fourth collection



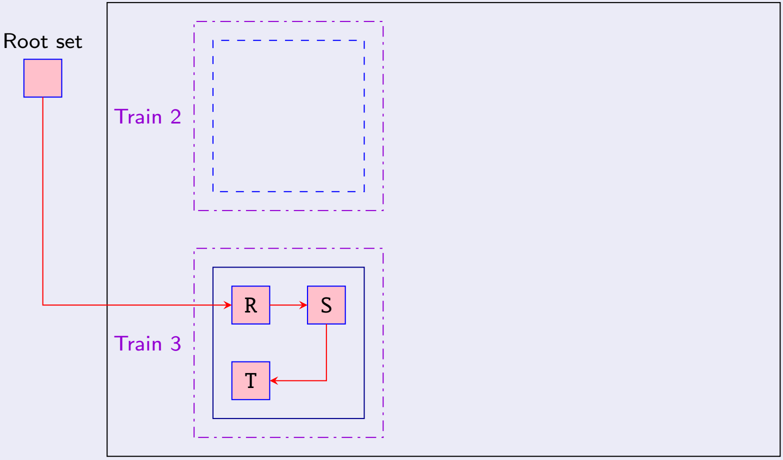
Generational garbage collection

Train algorithm — the state after the fifth collection



Generational garbage collection

Train algorithm — the state after the sixth collection



Generational garbage collection

Train algorithm — conclusion

- ✔ If structures without outside pointers are completely in a single train, they can be freed immediately.
- ✔ In each collection, the number of evacuated objects is bounded by size of a single car.
- ✔ Evacuated objects are compacted into a single train.
- ✘ Relatively complicated.
- ✘ Requires quite a lot of memory for remembered sets.

Generational garbage collection

Advantages of generational garbage collection

- ✔ Very successful for many applications.
- ✔ Often shortens garbage collection pauses into the level tolerable for interactive applications.
- ✔ Has good locality properties.
- ✔ Usually decreases the total garbage collection time.

Drawbacks of generational garbage collection

- ✘ Worst case efficiency is worse than in simpler methods.
- ✘ Objects may not die fast enough.
- ✘ Applications may be ”hindered” by write barriers.
- ✘ Too many old pointers into young objects, or too deep stack, may result longer pauses.