.NET GC Internals

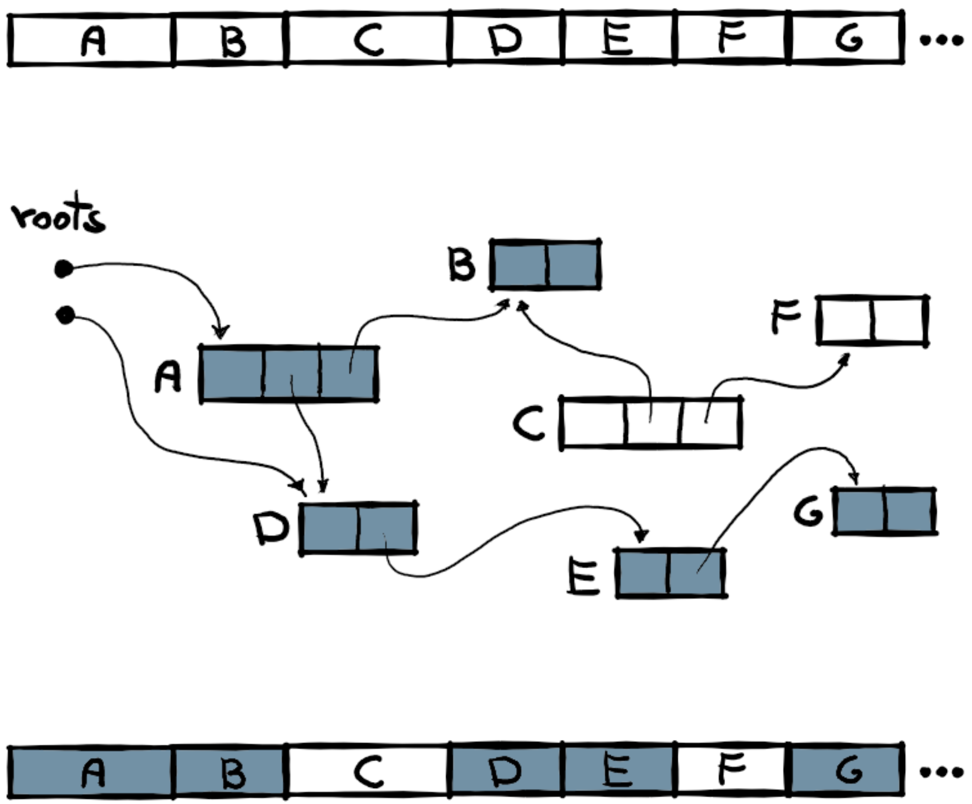
Plan phase

@konradkokosa / @dotnetosorg

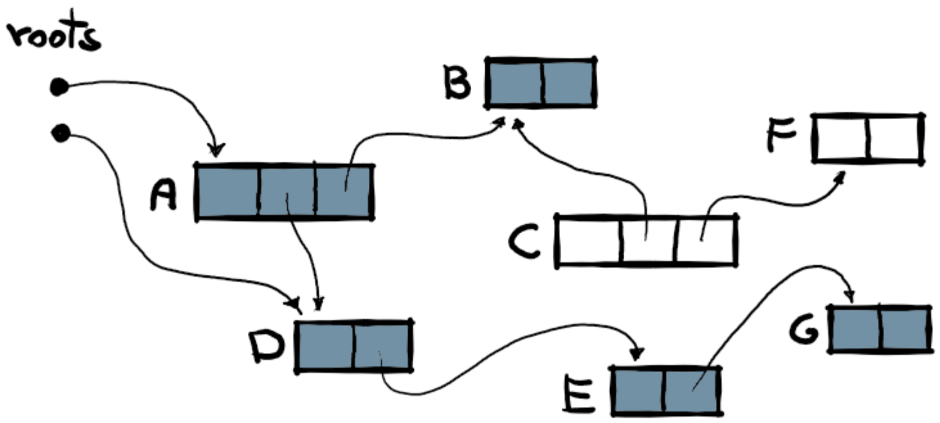
.NET GC Internals Agenda

- Introduction - roadmap and fundamentals, source code, ...
- **Mark** phase - roots, object graph traversal, *mark stack*, mark/pinned flag, *mark list*, ...
- **Concurrent Mark** phase - *mark array/mark word*, concurrent visiting, *floating garbage*, *write watch list*, ...
- **Plan** phase - *gap*, *plug*, *plug tree*, *brick table*, *pinned plug*, *pre/post plug*, ...
- **Sweep** phase - *free list threading*, concurrent sweep, ...
- **Compact** phase - *relocate* references, compact, ...
- **Generations** - physical organization, *card tables*, ...
- **Allocations** - *bump pointer allocator*, free list allocator, *allocation context*, ...
- **Roots internals** - stack roots, *GCInfo*, *partially/full interruptible methods*, statics, Thread-local Statics (TLS), ...
- **Q&A** - "but why can't I manually delete an object?", ...

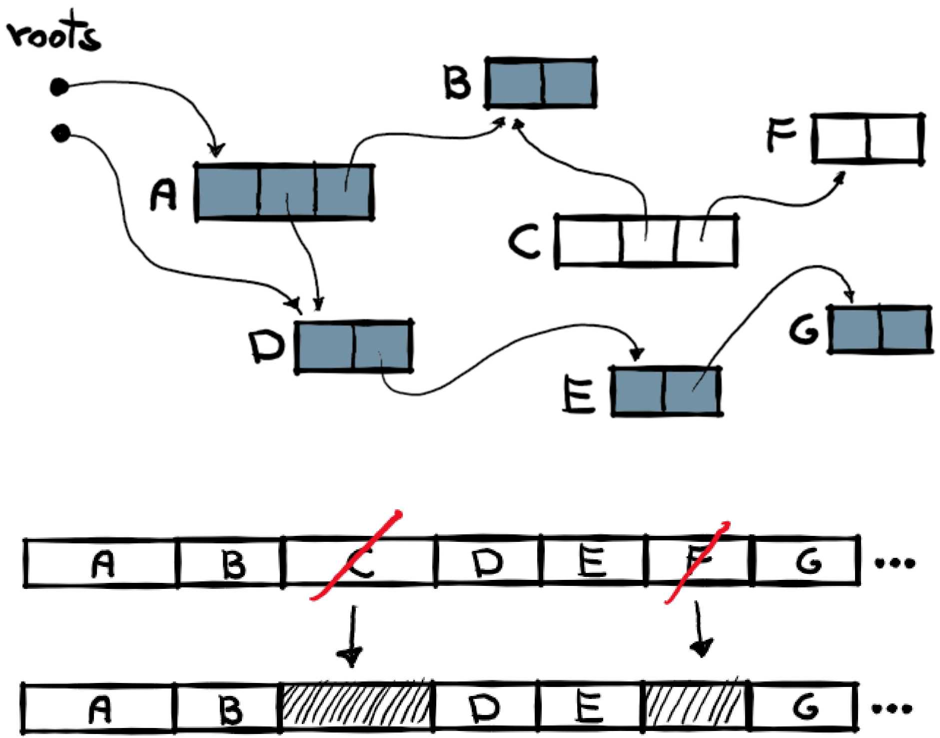
Mark phase...



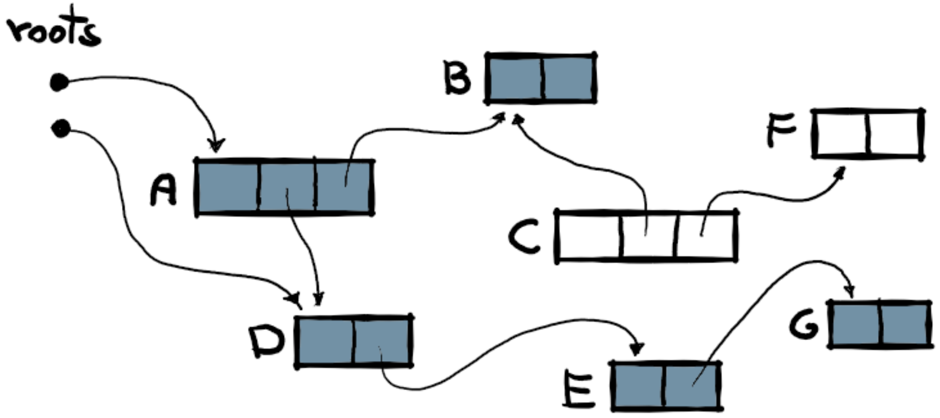
Sweep



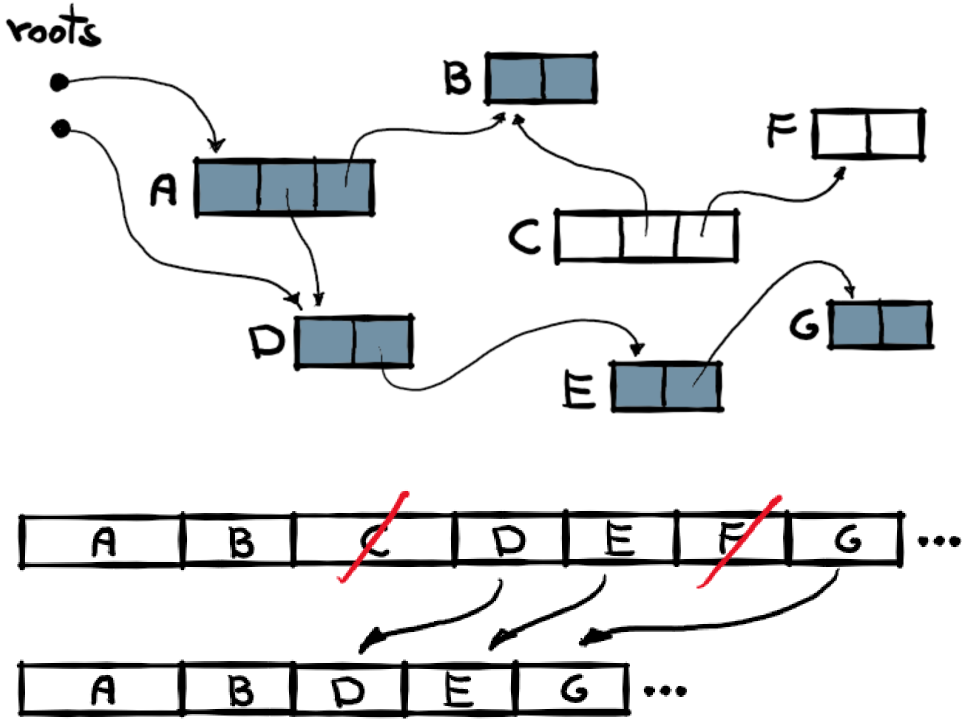
Sweep



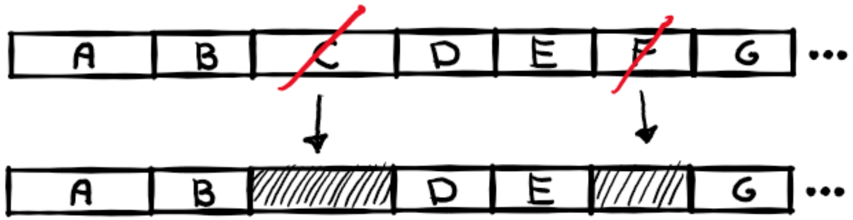
Compact - in-place

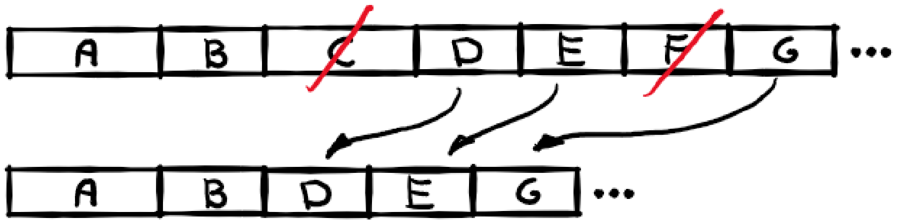


Compact - in-place



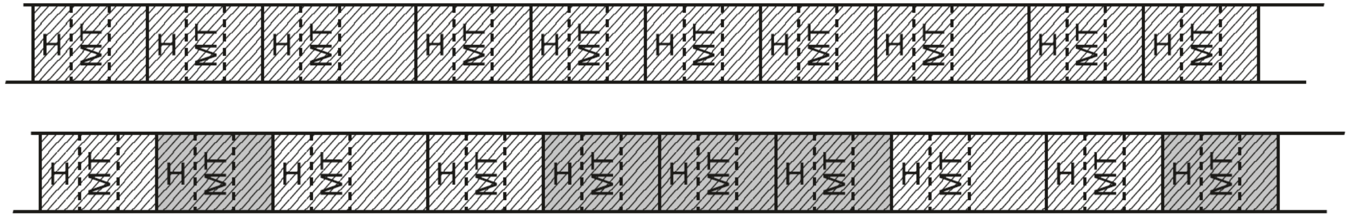
Plan - Sweep/Compact



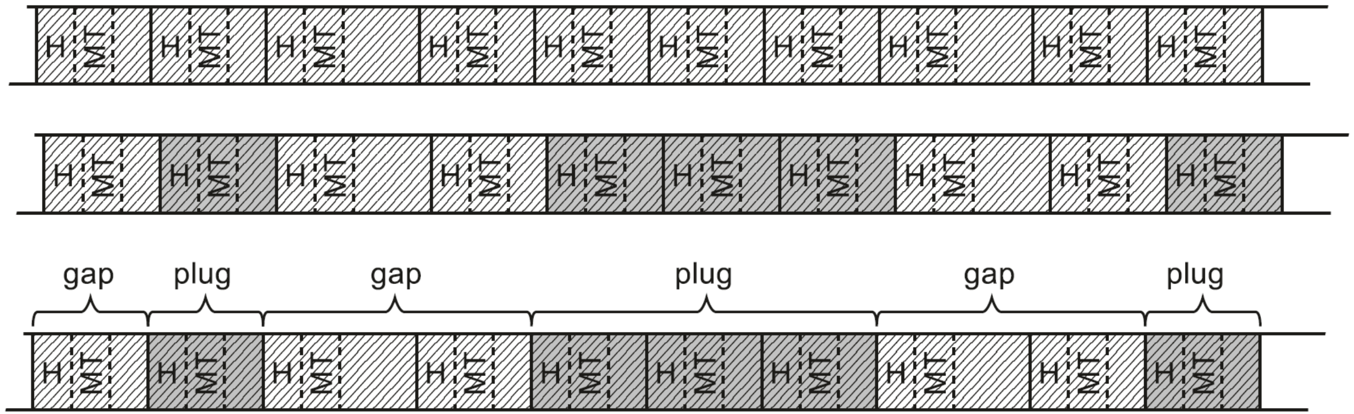


- how to make a decision whether to *sweep* or to *compact*...?! 🤔
- we need an aditional, decision-making *Plan phase*

Plan



Plan

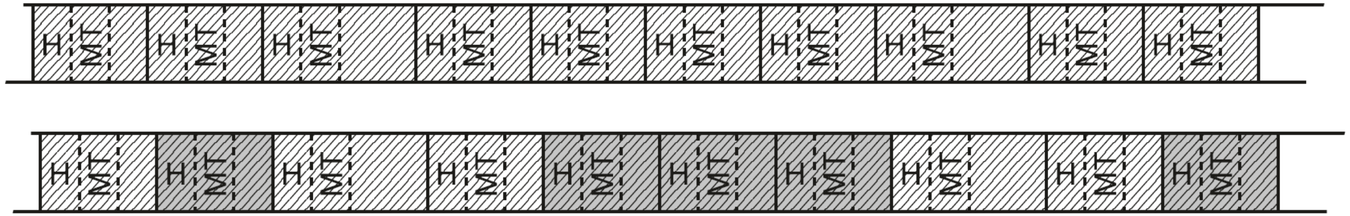


Mark phase implementation - recap

Sequentially for every root type (like stack, finalization, ...):

1. Collect the roots into the "to visit list" (**the mark stack**)
2. For each given target address `addr` from the mark stack:
    - translate it to the proper address of a managed object*
    - set **pinning flag** (in the `Header`) - if the runtime says so
    - start traversal:
        - skip already visited object
        - **mark** an object (in the `MT`)
        - add its address to **the mark list** (if not overflowed)
        - add outgoing references to the **mark stack**

Plan - *mark list* usage



If no *"mark list overflow"*, after finishing given *plug*, we don't scan object by object but use **sorted** mark list to quickly jump to the next marked object:

```
#ifdef MARK_LIST
        if (use_mark_list)
        {
           while ((mark_list_next < mark_list_index) &&
                  (*mark_list_next <= x))
           {
               mark_list_next++;
           }
           if ((mark_list_next < mark_list_index) ...)
*                 x = *mark_list_next;
...
            }
            else
#endif //MARK_LIST
```

Plan - *mark list* usage

Only for ephemeral GCs (gen 0/1):

```
// dont use the mark list for full gc because multiple segments are more complex to handle
// and the list is likely to overflow
```

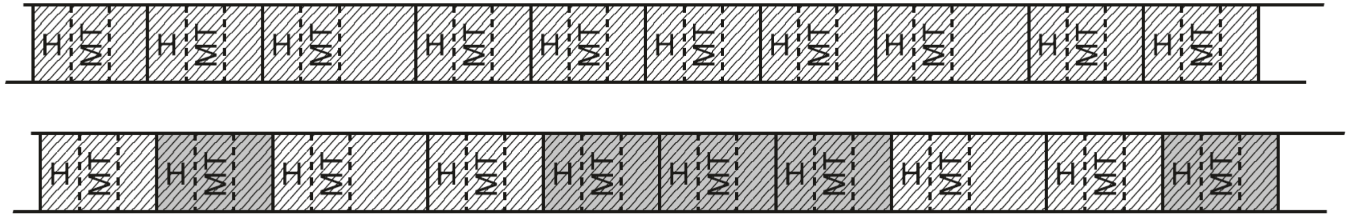
And if not background GC:

```
// we are not going to use the mark list if background GC is running so let's not waste time sorting it
```

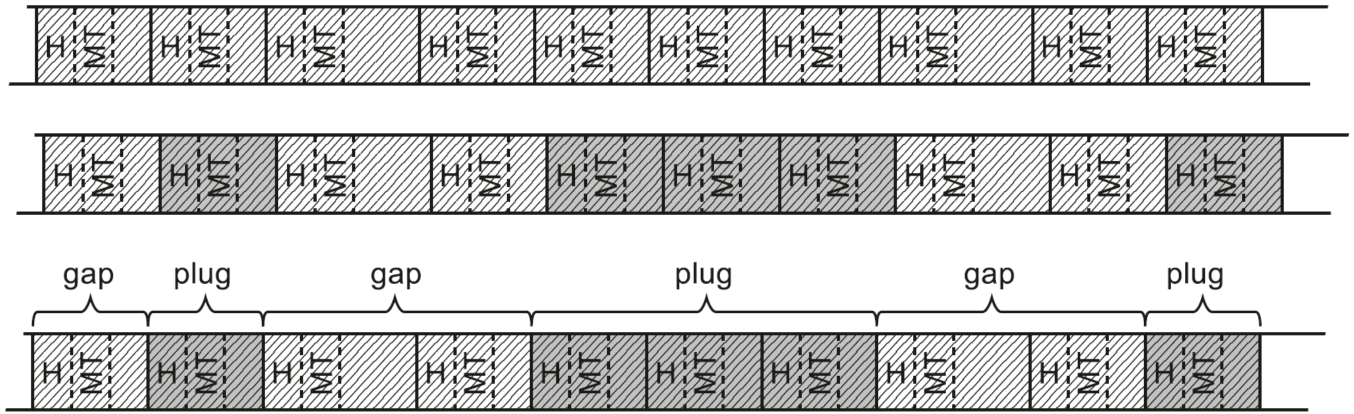
So, *"mark list sorting story"* of vectorizing mark list sorting (AVX2/AVX512F) is beneficial becasue it is:

- faster sorting -> shorter pauses
- bigger mark list sizes -> less objects scanning -> shorter pauses

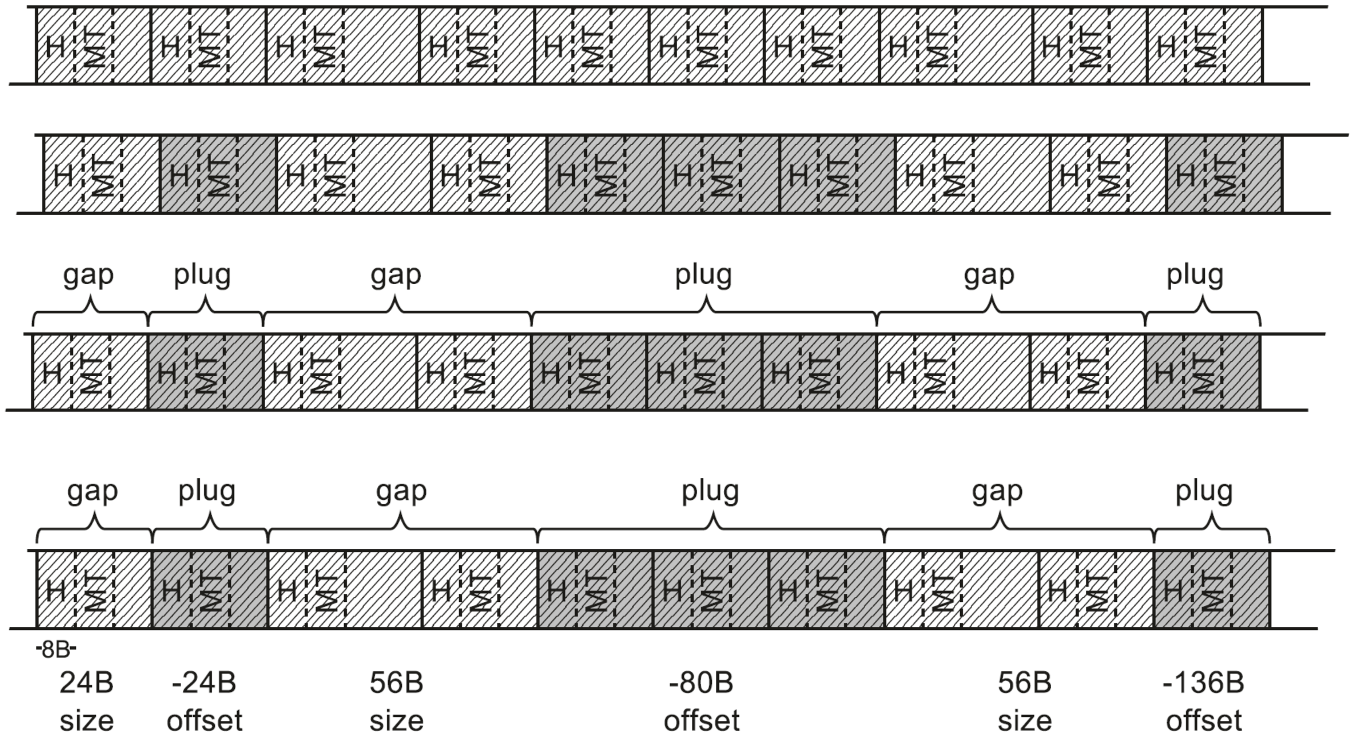
Plan


H
MT
H
MT
H
MT
H
MT
H
MT
H
MT
H
MT
H
MT
H
MT
H
MT
H
MT
H
MT
H
MT
H
MT
H
MT
H
MT
H
MT
H
MT
H
MT
H
MT

Plan



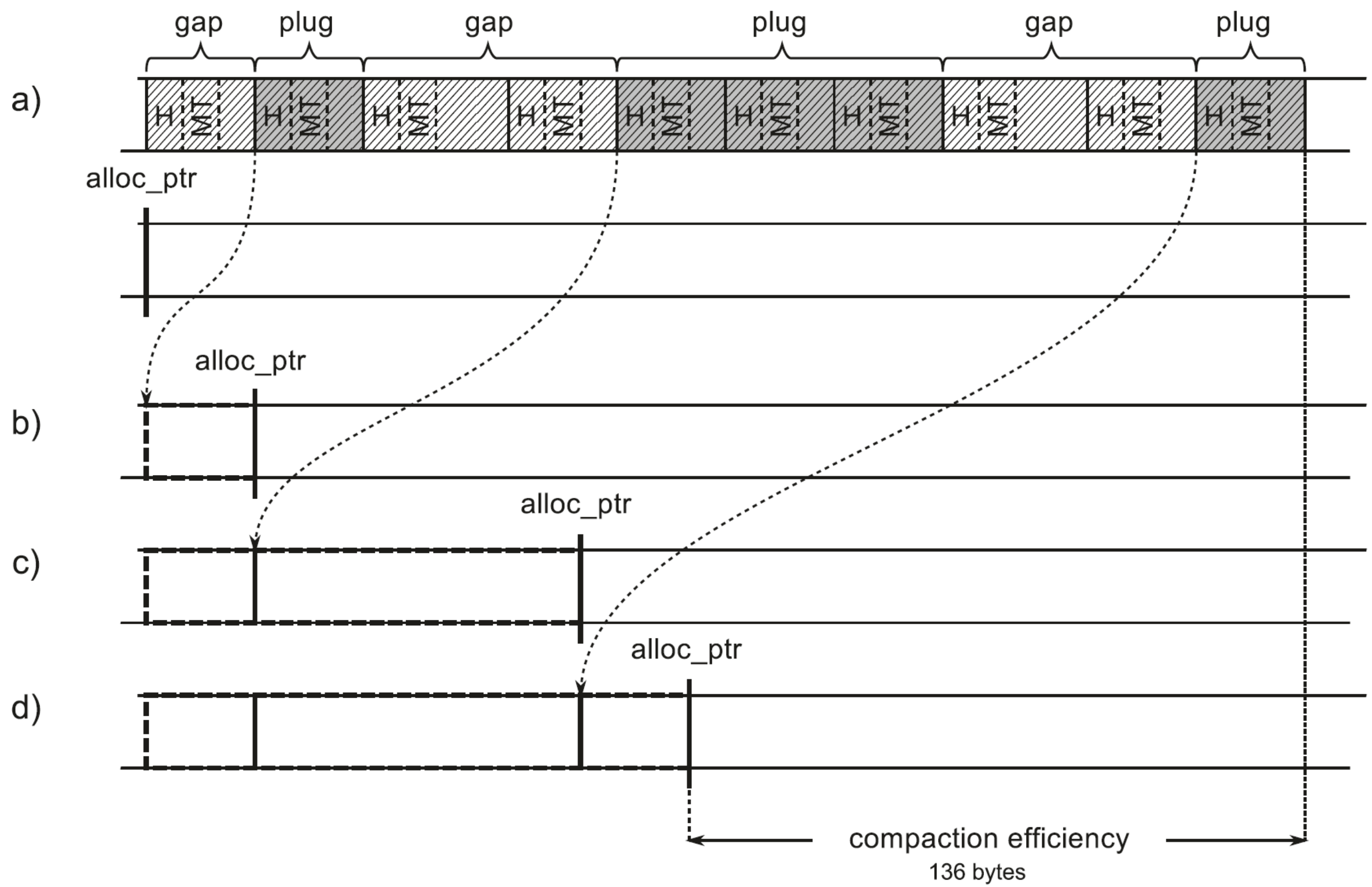
Plan



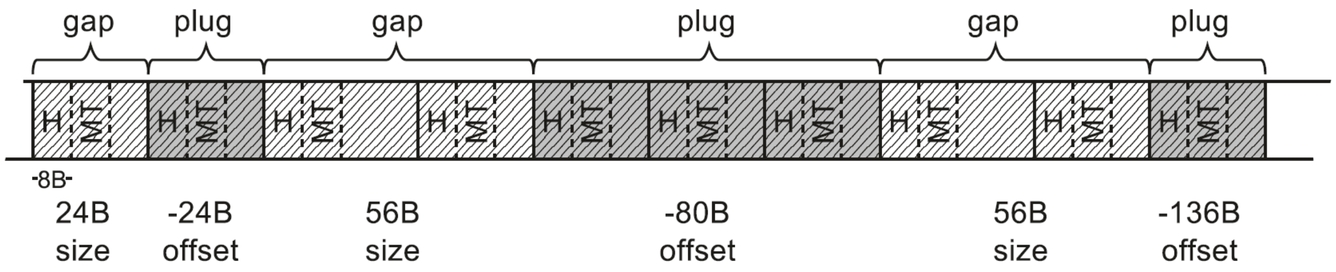
Relocation offset for each plug, *size* for each gap (in fact, gap-plug pairs).

Plan

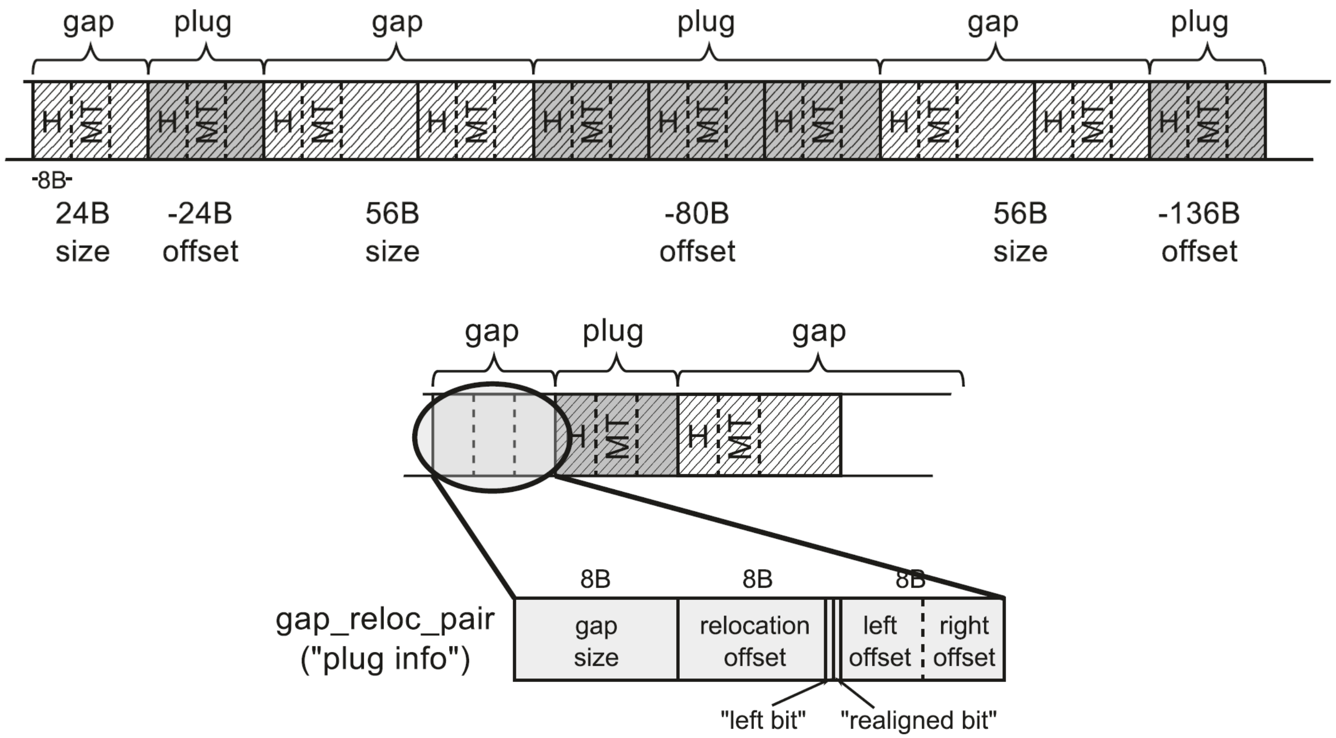
"Allocator" calculates the reallocation offsets of every plug:



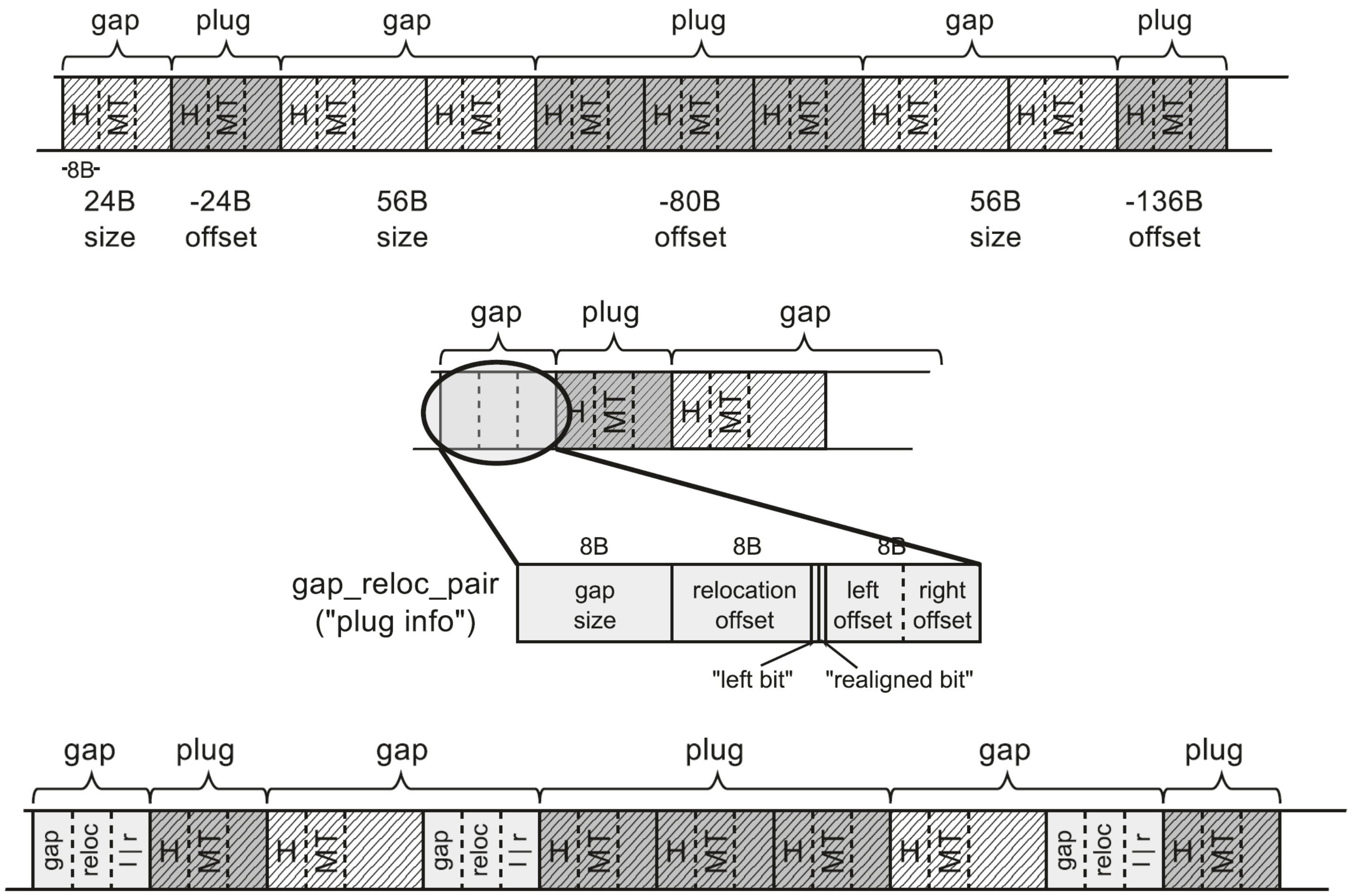
Plan - *gap-plug pairs* storage



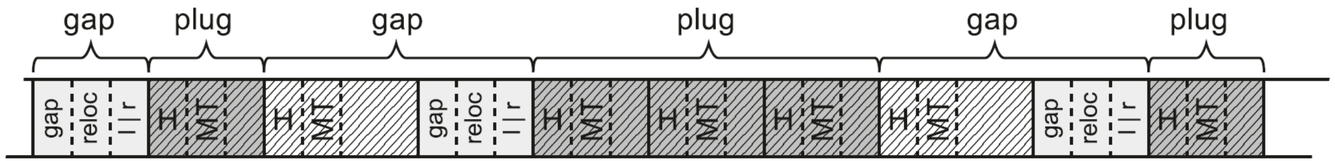
Plan - *gap-plug pairs* storage



Plan - *gap-plug pairs* storage



Plan - *gap-plug pairs* storage

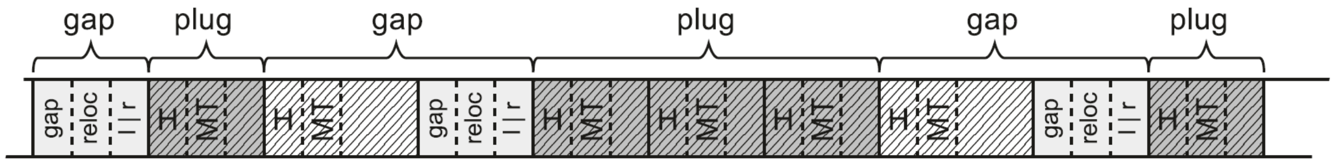


Storing gap-plug info on the Managed Heap just before a plug is the main reason why **even an empty object must be 24-bytes in size** (in case of 64-bit runtime).

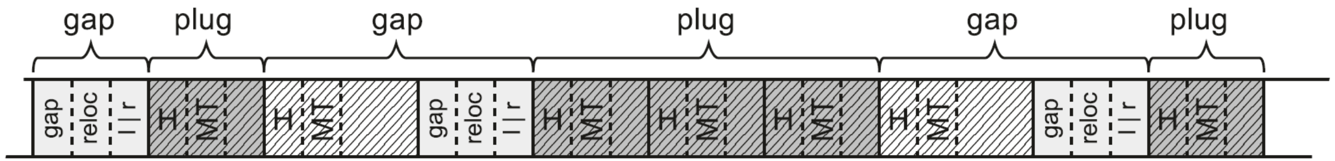
Plan - sidenote

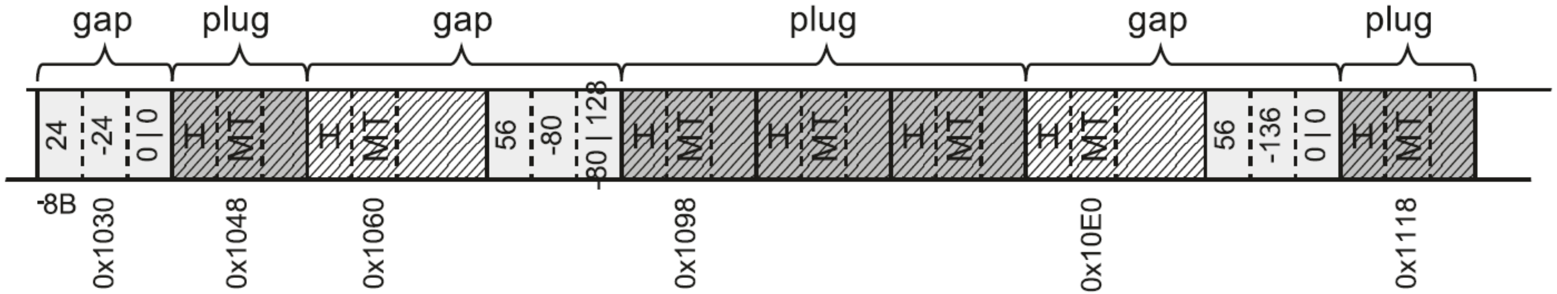
```
> !dumpheap -stat
The garbage collector data structures are not in a valid state for traversal.
It is either in the "plan phase," where objects are being moved around, or
we are at the initialization or shutdown of the gc heap. Commands related to
displaying, finding or traversing objects as well as gc heap segments may not
work properly. !dumpheap and !verifyheap may incorrectly complain of heap
consistency errors.
```

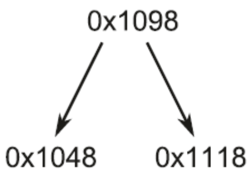
Plan - *gap-plug pairs* storage



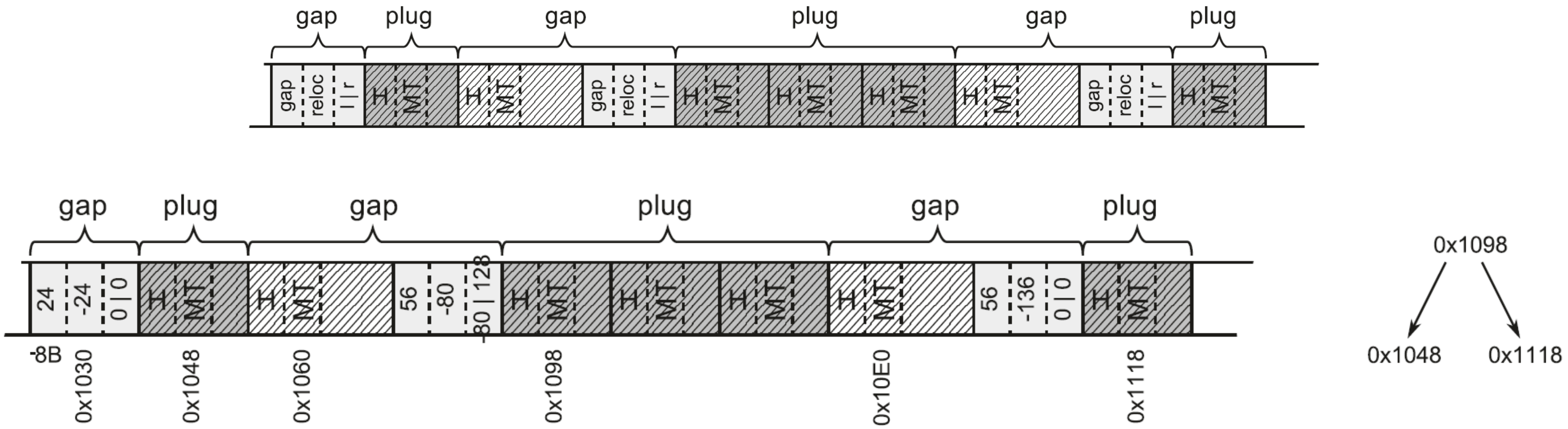
Plan - *gap-plug pairs* storage





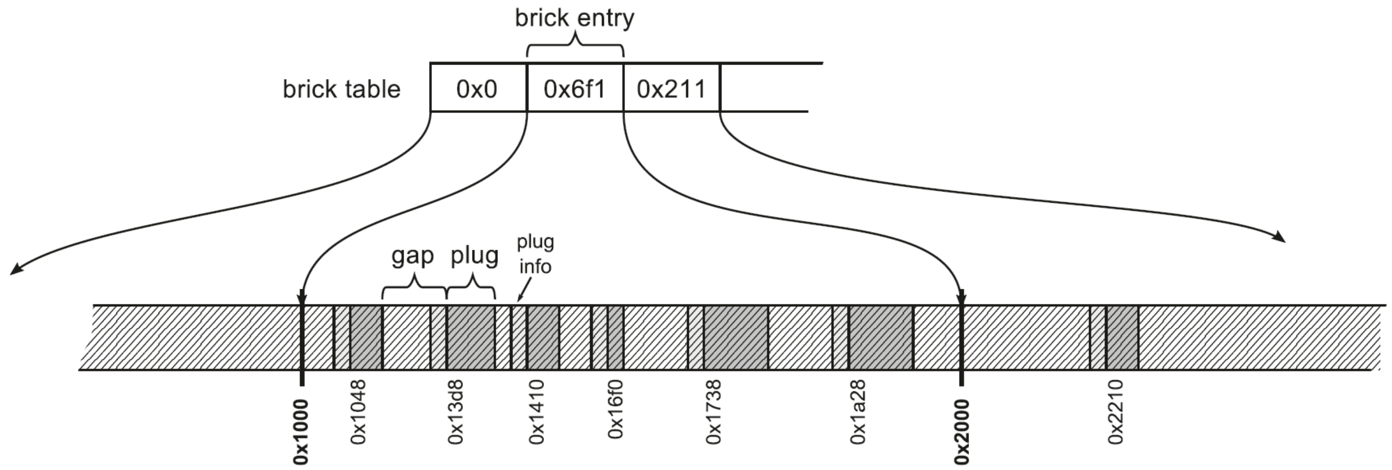


Plan - *gap-plug pairs* storage



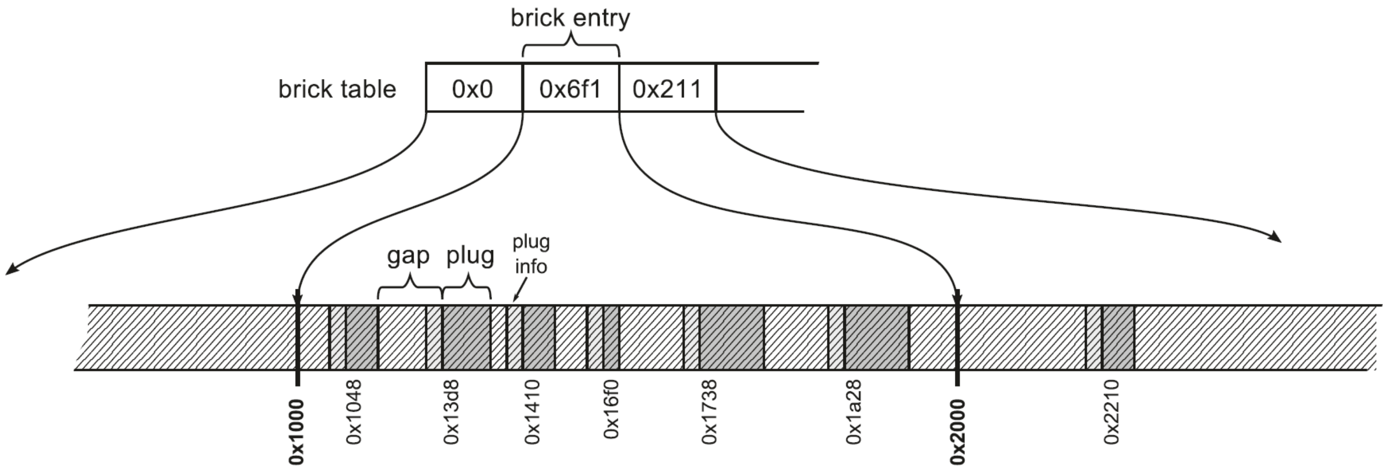
Left/right offsets build a binary tree of plugs info.

Plan - bricktable

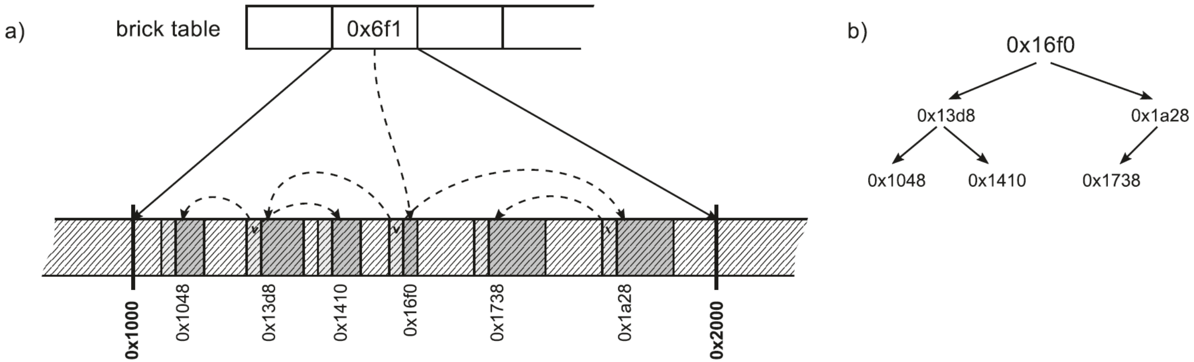


Brick size is 2,048 B for 32bit and 4,096 B for 64-bit runtimes

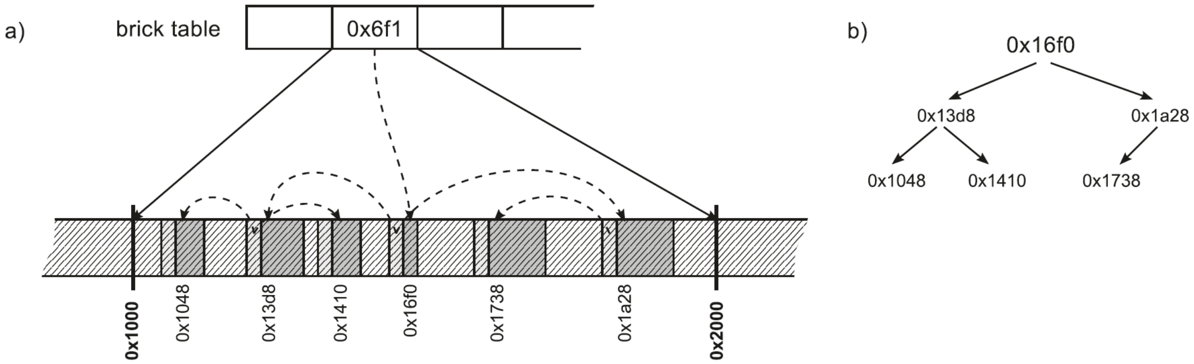
Plan - bricktable



Brick size is 2,048 B for 32bit and 4,096 B for 64-bit runtimes



Plan - bricktable

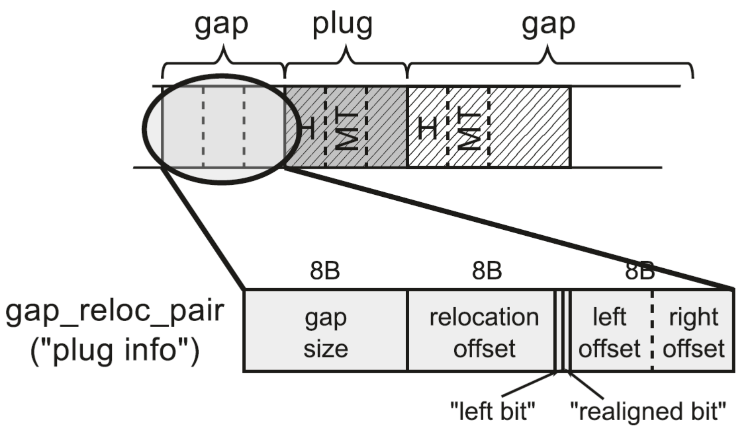


What will be the **new address** of the object at address X?

- calculate the brick table entry based on address X - by simply dividing it by a brick size
- if brick table entry is <0 - jump into proper brick table entry and repeat.
- if brick table entry is >0 - start to traverse plug tree to find proper plug.
  - get relocation offset from the plug and subtract it from X.

Also used for translating interior pointers (**after** Plan phase).

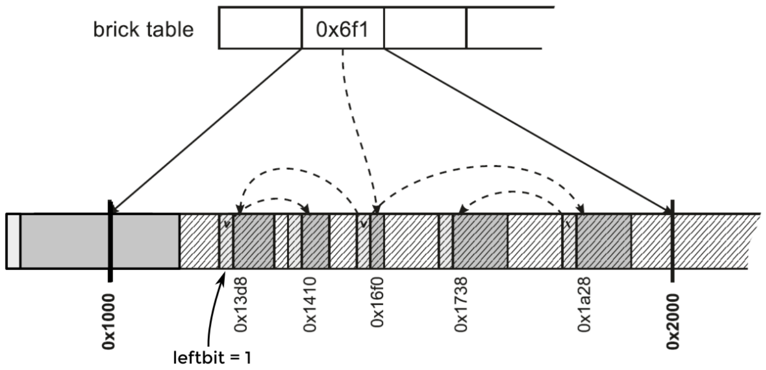
Plan - a little bits...




```
inline ptrdiff_t node_relocation_distance (uint8_t* node)
{
    return (((plug_and_reloc*)(node))[-1].reloc & ~3);
}
```

Plan - left bit

If we know we will relocate a plug right next **to the last plug in the previous brick** - relocation can be taken from the last plug.




```
if ((node <= old_loc))
    new_address = (old_address + node_relocation_distance (node));
else {
    if (node_left_p (node))
        new_address = (old_address + (node_relocation_distance (node) + node_gap_size (node)));
    else {
        brick = brick - 1;
        brick_entry =  brick_table [ brick ];
        goto retry;
    }
}
```

Plan - realigned bit

Information that we need to **pad** (add some additional space) a plug to **align** it based on some requirements - like preserving *double*s alignment in 32-bit runtime.

Plan - realigned bit

Information that we need to **pad** (add some additional space) a plug to **align** it based on some requirements - like preserving *double*s alignment in 32-bit runtime.

```
if (node_realigned (plug))
{
    unused_arr_size += switch_alignment_size (already_padded_p);
}
```

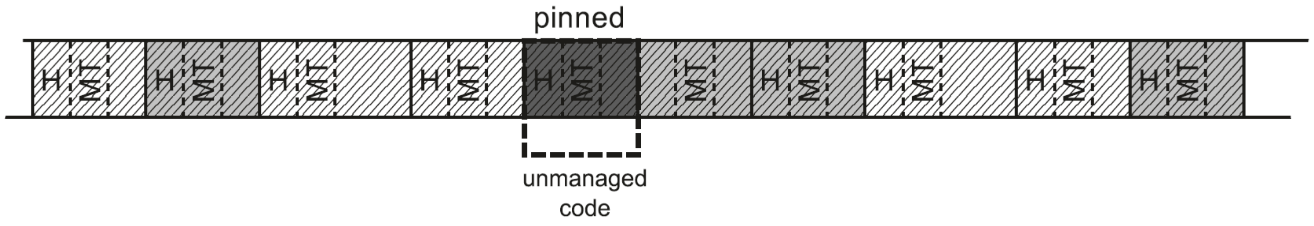
Plan - realigned bit

Information that we need to **pad** (add some additional space) a plug to **align** it based on some requirements - like preserving *double*s alignment in 32-bit runtime.

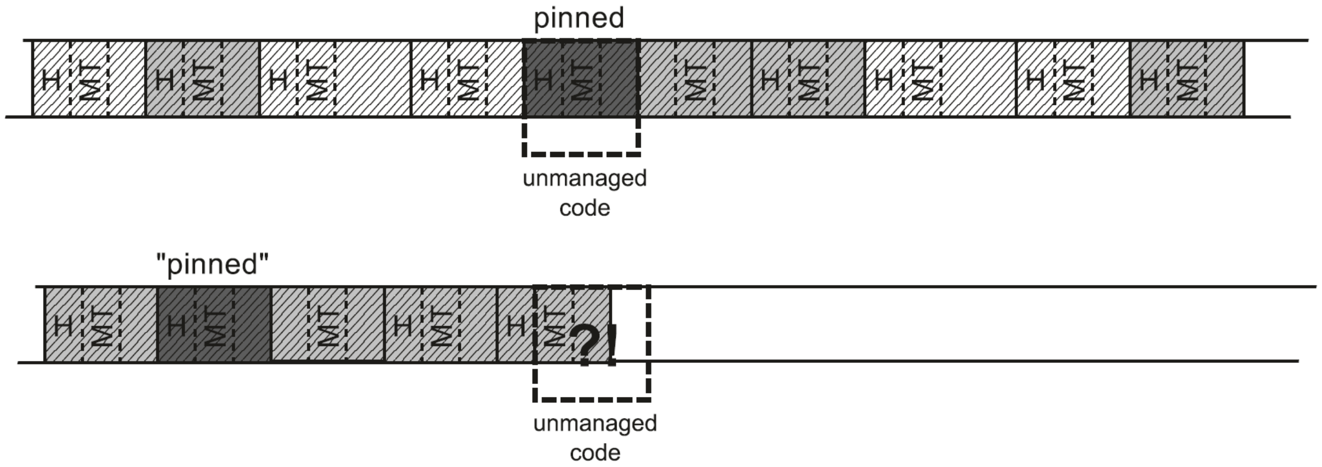
```
if (node_realigned (plug))
{
    unused_arr_size += switch_alignment_size (already_padded_p);
}
```

We will see *"large (double than normal) alignment"* here and there.

Plan - pinning



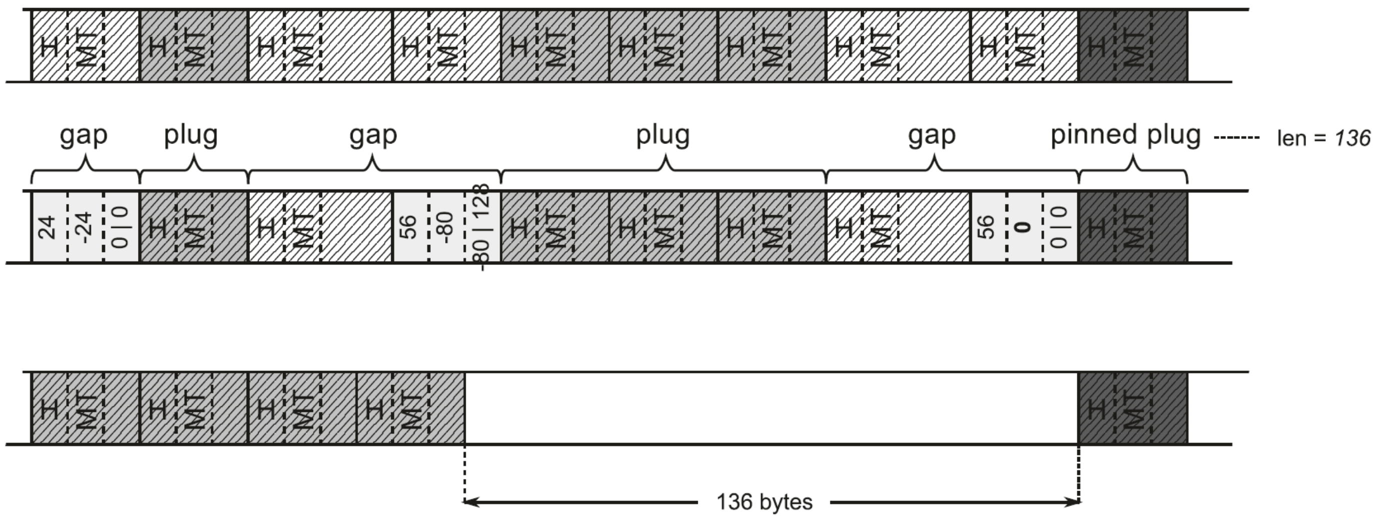
Plan - pinning



Plan - pinning

Pinned plug - special marked plug

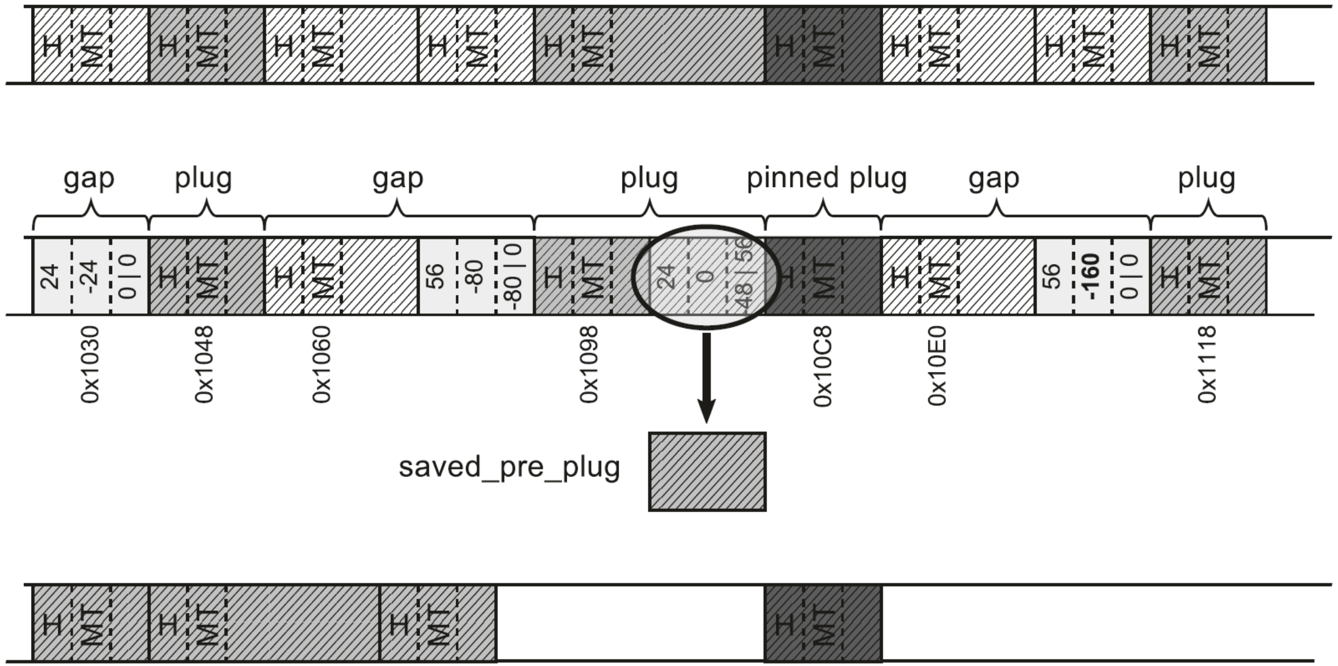
- zeroed relocation offset
- additional info about free space before it (in case of compaction)



Easy case because there was a gap before it.

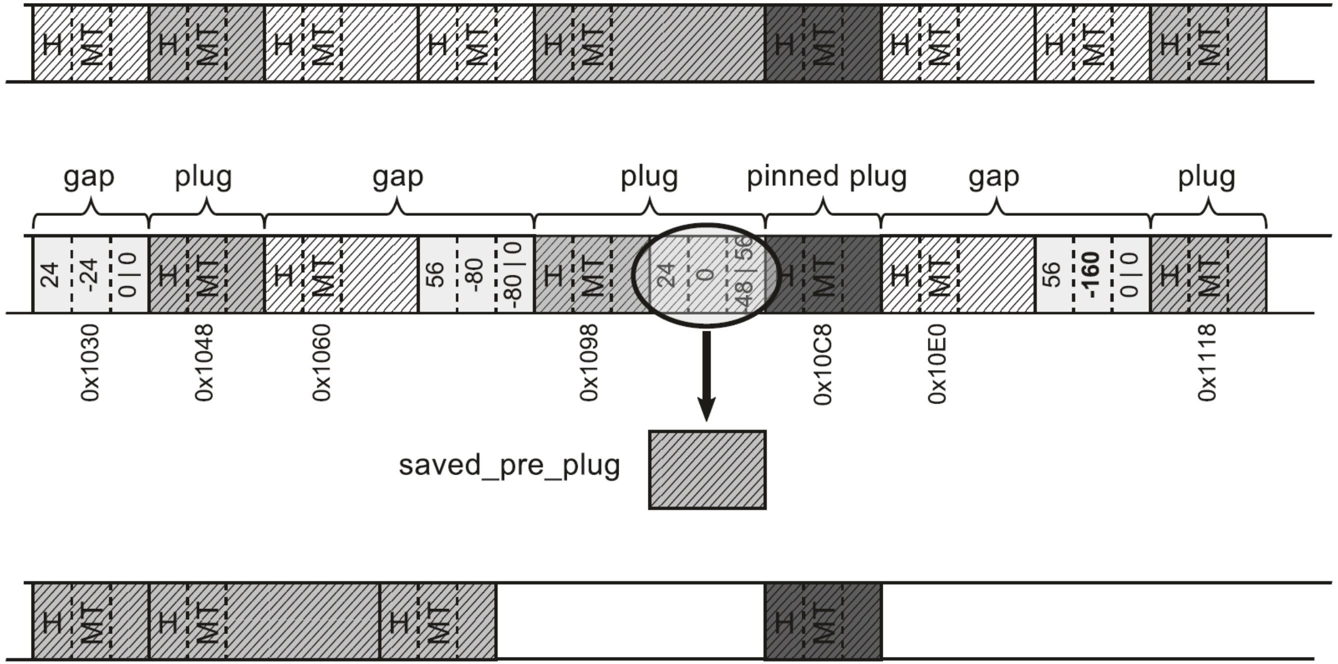
Plan - pinning

Pinned plug after normal plug



Plan - pinning

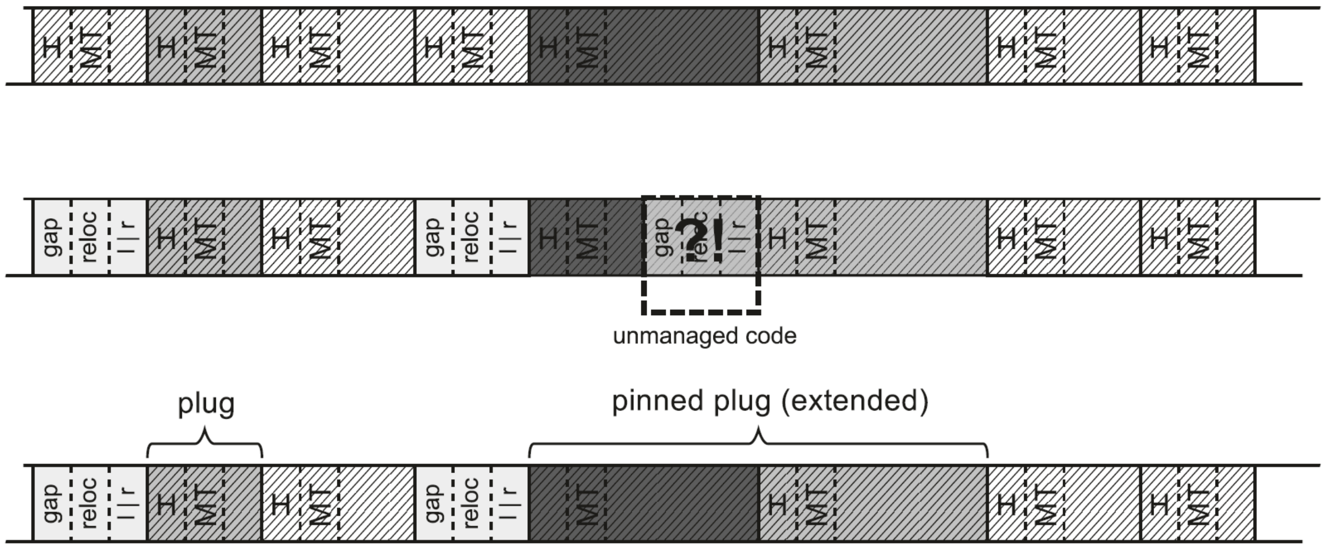
Pinned plug after normal plug



Pinned plug queue is used to save this info (and we reuse *mark stack* - `mark_stack_array` here 👀)

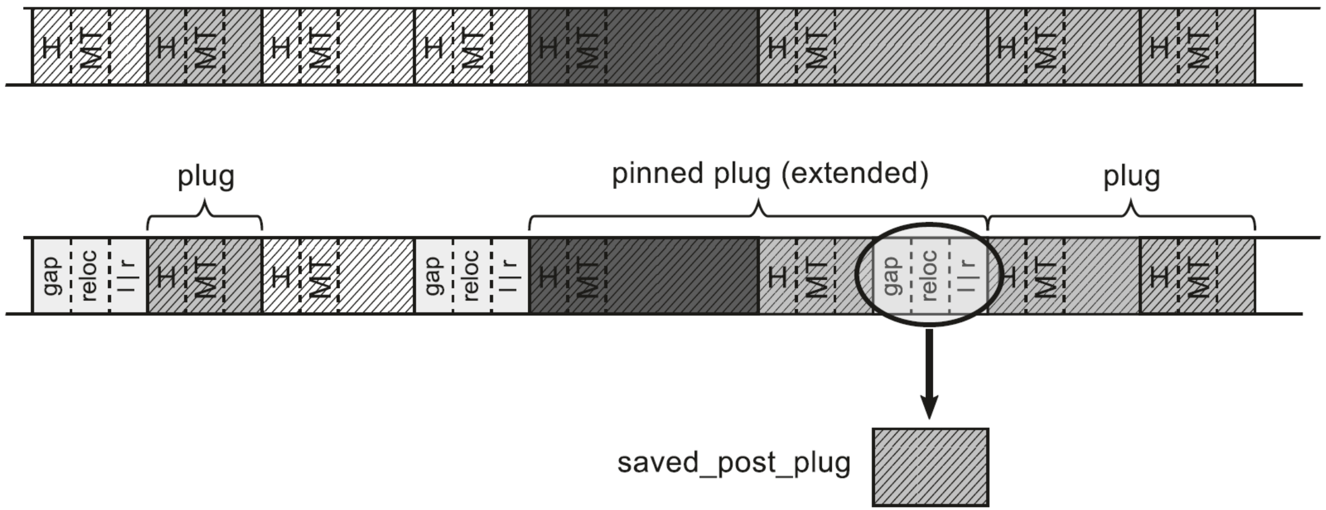
Plan - pinning

Pinned plug before normal plug



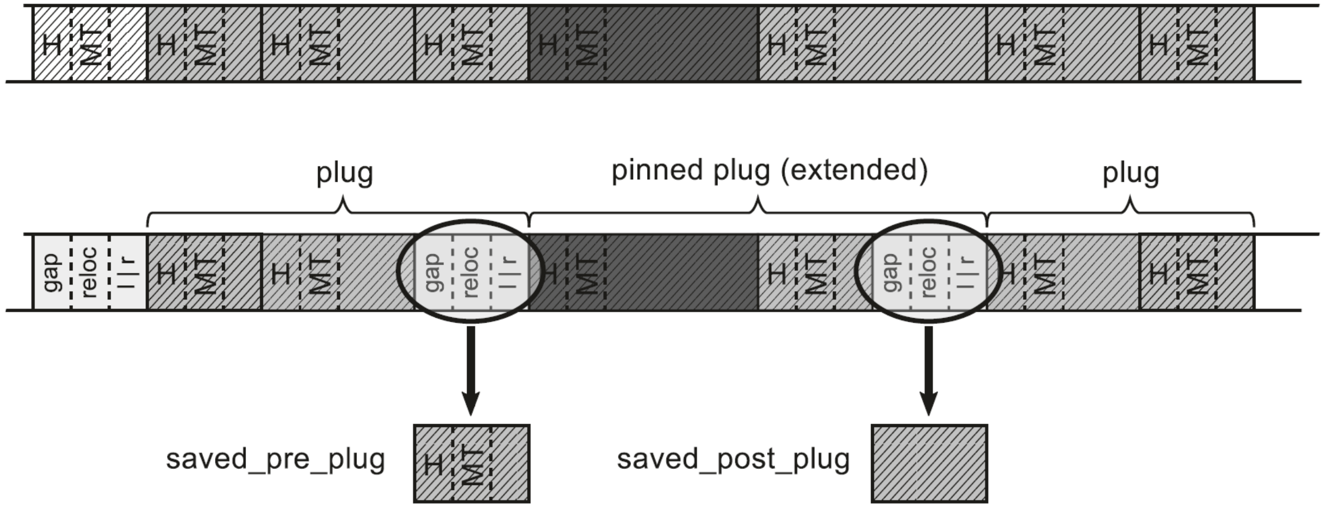
Plan - pinning

Pinned plug is located before at least two marked objects - we could create HUGE pinned plug this way...



Plan - pinning

Pinned plug is located inside larger block of marked objects



Plan - pinning

Pinning consequences summary:

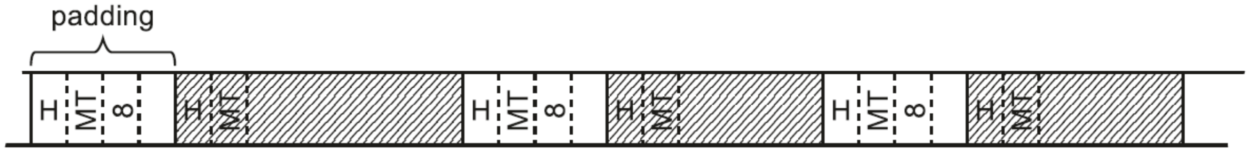
- copying pre and post plugs introduces memory traffic
- pinned plug can be extended by a single object so more memory is being pinned than it could be
- during Plan phase some objects on the Managed Heap are "destroyed" making it not "walkable" in a normal way

Plan - Large Object Heap

- all this was described for Small Object Heap - where we expect many small objects and occassional compacting
- LOH is for not-so-many large objects (almost) never compacted, thus it is much simpler and **optional**:
  - no grouping into plugs/gaps - every object is a "plug"/"gap"
  - no bricktables
  - small padding (of type *Free*) between all objects - to have space for plug info

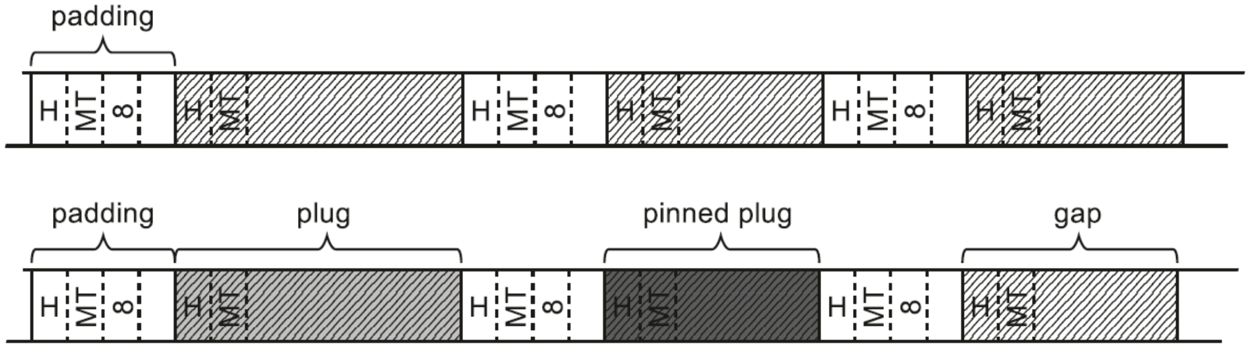
Plan - Large Object Heap

- all this was described for Small Object Heap - where we expect many small objects and occassional compacting
- LOH is for not-so-many large objects (almost) never compacted, thus it is much simpler and **optional**:
  - no grouping into plugs/gaps - every object is a "plug"/"gap"
  - no bricktables
  - small padding (of type *Free*) between all objects - to have space for plug info



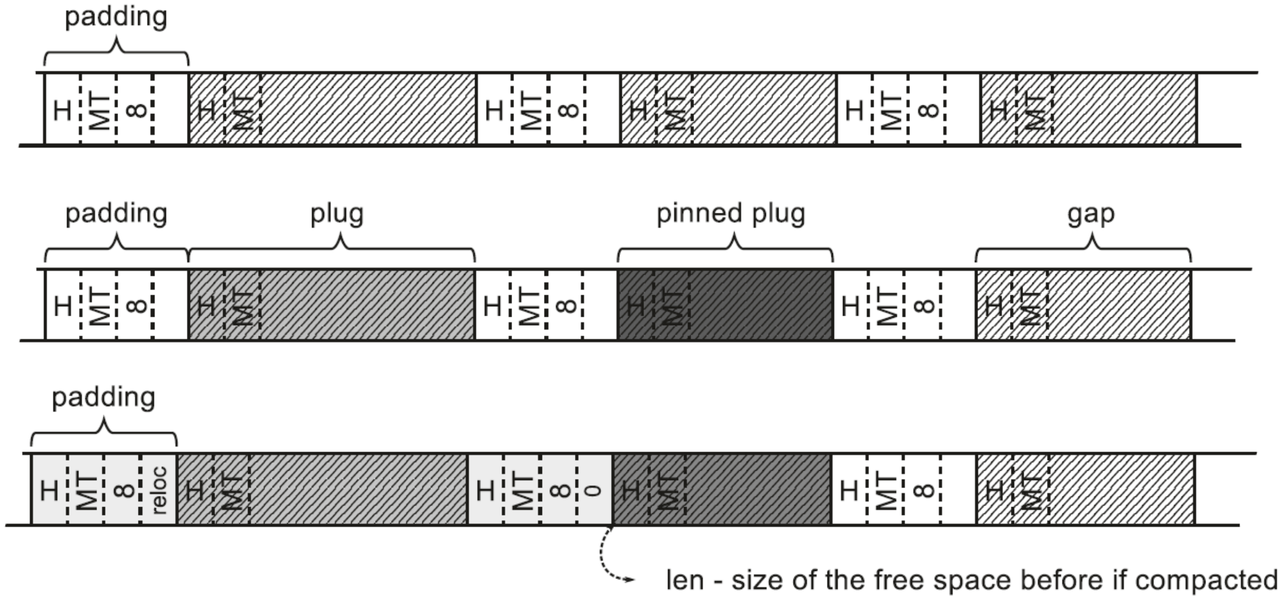
Plan - Large Object Heap

- all this was described for Small Object Heap - where we expect many small objects and occassional compacting
- LOH is for not-so-many large objects (almost) never compacted, thus it is much simpler and **optional**:
  - no grouping into plugs/gaps - every object is a "plug"/"gap"
  - no bricktables
  - small padding (of type *Free*) between all objects - to have space for plug info



Plan - Large Object Heap

- all this was described for Small Object Heap - where we expect many small objects and occassional compacting
- LOH is for not-so-many large objects (almost) never compacted, thus it is much simpler and **optional**:
  - no grouping into plugs/gaps - every object is a "plug"/"gap"
  - no bricktables
  - small padding (of type *Free*) between all objects - to have space for plug info



Plan phase - inside code

All this happens in `gc_heap::plan_phase` method. The new location of each plug is calculated by calling `allocate_in_condemned_generations` or `allocate_in_older_generations`. Optional `gc_heap::plan_loh` may be also called. In the end, to make a decision `decide_on_compacting` is called.