The prio_tree.c code indexes vmas using 3 different indexes:
* heap_index = vm_pgoff + vm_size_in_pages : end_vm_pgoff
* radix_index = vm_pgoff : start_vm_pgoff
* size_index = vm_size_in_pages

A regular radix-priority-search-tree indexes vmas using only heap_index and radix_index. The conditions for indexing are:
* ->heap_index >= ->left->heap_index &&
  ->heap_index >= ->right->heap_index
* if (->heap_index == ->left->heap_index)
  then ->radix_index < ->left->radix_index;
* if (->heap_index == ->right->heap_index)
  then ->radix_index < ->right->radix_index;
* nodes are hashed to left or right subtree using radix_index
  similar to a pure binary radix tree.

A regular radix-priority-search-tree helps to store and query intervals (vmas). However, a regular radix-priority-search-tree is only suitable for storing vmas with different radix indices (vm_pgoff).

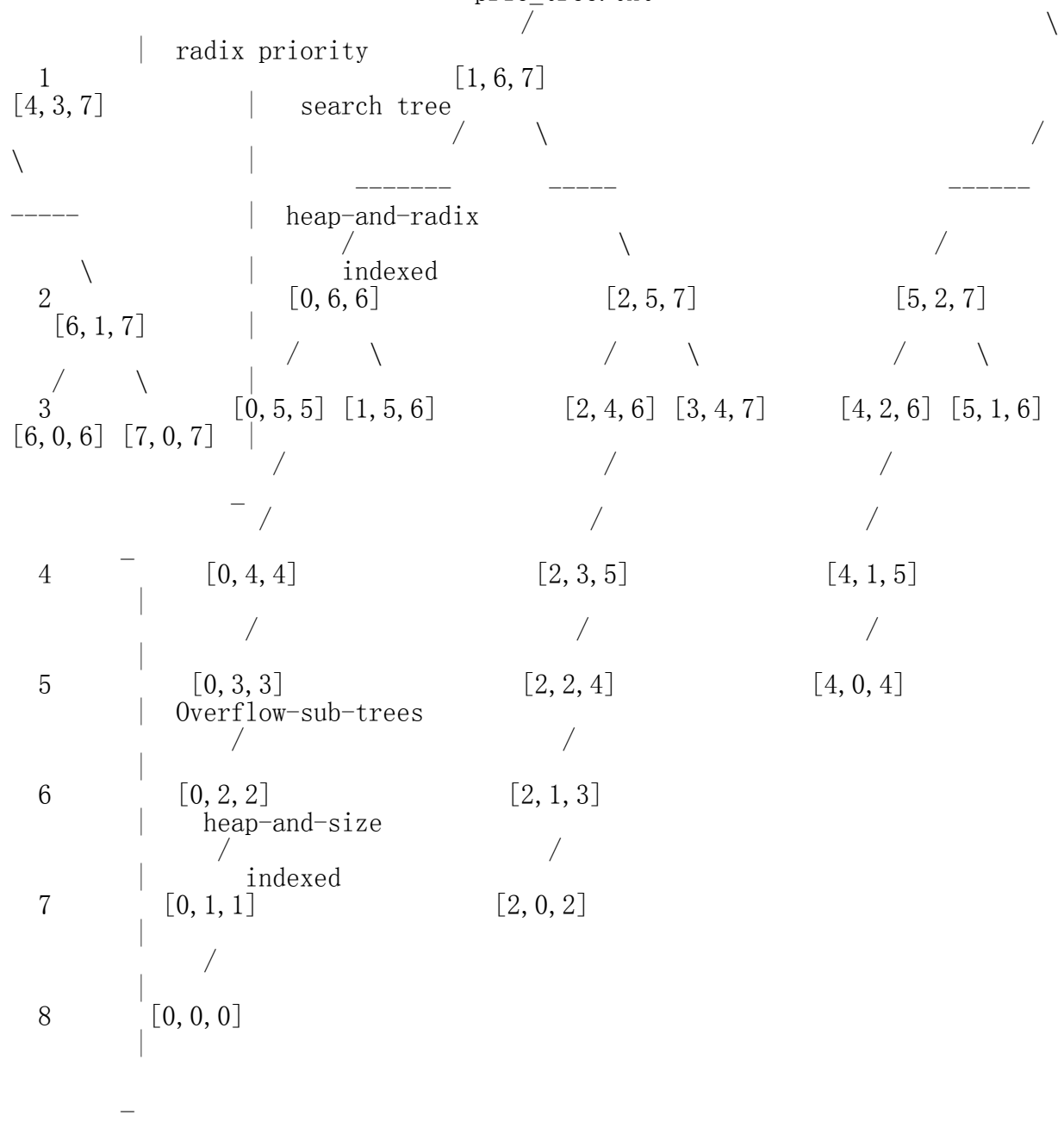
Therefore, the prio_tree.c extends the regular radix-priority-search-tree to handle many vmas with the same vm_pgoff. Such vmas are handled in 2 different ways: 1) All vmas with the same radix _and_ heap indices are linked using vm_set.list, 2) if there are many vmas with the same radix index, but different heap indices and if the regular radix-priority-search tree cannot index them all, we build an overflow-sub-tree that indexes such vmas using heap and size indices instead of heap and radix indices. For example, in the figure below some vmas with vm_pgoff = 0 (zero) are indexed by regular radix-priority-search-tree whereas others are pushed into an overflow-subtree. Note that all vmas in an overflow-sub-tree have the same vm_pgoff (radix_index) and if necessary we build different overflow-sub-trees to handle each possible radix_index. For example, in figure we have 3 overflow-sub-trees corresponding to radix indices 0, 2, and 4.

In the final tree the first few (prio_tree_root->index_bits) levels are indexed using heap and radix indices whereas the overflow-sub-trees below those levels (i.e. levels prio_tree_root->index_bits + 1 and higher) are indexed using heap and size indices. In overflow-sub-trees the size_index is used for hashing the nodes to appropriate places.

Now, an example prio_tree:

vmas are represented [radix_index, size_index, heap_index]
i.e., [start_vm_pgoff, vm_size_in_pages, end_vm_pgoff]

level prio_tree_root->index_bits = 3



Note that we use prio_tree_root->index_bits to optimize the height
of the heap-and-radix indexed tree. Since prio_tree_root->index_bits is
set according to the maximum end_vm_pgoff mapped, we are sure that all
bits (in vm_pgoff) above prio_tree_root->index_bits are 0 (zero). Therefore,
we only use the first prio_tree_root->index_bits as radix_index.
Whenever index_bits is increased in prio_tree_expand, we shuffle the tree
to make sure that the first prio_tree_root->index_bits levels of the tree
is indexed properly using heap and radix indices.

We do not optimize the height of overflow-sub-trees using index_bits.
The reason is: there can be many such overflow-sub-trees and all of
them have to be suffled whenever the index_bits increases. This may involve
walking the whole prio_tree in prio_tree_insert->prio_tree_expand code
path which is not desirable. Hence, we do not optimize the height of the
heap-and-size indexed overflow-sub-trees using prio_tree->index_bits.

Instead the overflow sub-trees are indexed using full BITS_PER_LONG bits of size_index. This may lead to skewed sub-trees because most of the higher significant bits of the size_index are likely to be 0 (zero). In the example above, all 3 overflow-sub-trees are skewed. This may marginally affect the performance. However, processes rarely map many vmas with the same start_vm_pgoff but different end_vm_pgoffs. Therefore, we normally do not require overflow-sub-trees to index all vmas.

From the above discussion it is clear that the maximum height of a prio_tree can be prio_tree_root->index_bits + BITS_PER_LONG. However, in most of the common cases we do not need overflow-sub-trees, so the tree height in the common cases will be prio_tree_root->index_bits.

It is fair to mention here that the prio_tree_root->index_bits is increased on demand, however, the index_bits is not decreased when vmas are removed from the prio_tree. That's tricky to do. Hence, it's left as a home work problem.