New Memory Management Structure

[It is assumed the reader is familiar with OakMap data structure. This document speaks only about memory management.]

Hereby please find the class diagram of the new memory management structure for Oak. It shows the default memory allocation schema and allows user to provide its own memory allocator. In case user-defined memory allocator is used, the off-heaping of the underlying ByteBuffers is up to the provided allocator.

Default Oak memory management schema is

- 1. Thread safe
- 2. Allows allocation of any amount of memory for multiple Oak instances (up to OS/HW boundaries)
- 3. Currently works only with ByteBuffers
- 4. Keeps a memory size bound per OakMap, meaning if more than predefined amount of memory (memory capacity) is requested per single OakMap instance the exception will be thrown.

class MemoryBlocksPool singleton Pre-allocates Blocks and gives them to OakNativeMemory- Allocator upon request. Methods: getInstance() getBlock() returnBlock()	1 N BI of BI th	ass Block ByteBuffer from a re-allocated and re lock can allocate a f it), but cannot de lock can be return en reset and reus lethods: reset(given (high) cap eused by the Poo a new ByteBuffer allocate it. Only e ed to the Pool an ed.) allocate(s	acity, I. (part entire id size)	public interface OakMemoryAllocator Pluggable implementation, Oak has its default implementation Methods: ByteBuffer allocate(size) void free(ByteBuffer) void close	÷()
One per Oak instance (OakMap data members)						
Class OakNativeMemoryAllocator Implements OakMemoryAllocator Uses and reuses the Block's memory (allocates small-sized ByteBuffers). If needed and possible, manages multiple Blocks (from Pool). Keeps the size bound of this OakMap and throws an out of memory exception if more memory is requested. Keeps the freeList for previously allocated ByteBuffers now available for reuse. The allocation request either reuses the freeList or allocates new from the Block. Upon close() returns all the Blocks to the Pool Methods: ByteBuffer allocate(size) free(ByteBuffer) close()						
			1			
class OakMemoryManager The only gate for all memory requests of a single OakMap instance, keeps OakMemoryAllocator as a private data field. Manages the garbage collection and thus releaseList , once it is safe to free some ByteBuffer (not accessed by any thread) returns it to the OakMemoryAllocator (free()).						
Methods: startOp	() stopO	p() ByteBuffer	allocate(size)	relea	se(ByteBuffer) close	e()

How to create OakBuilder with user-defined MemoryAllocator or with default OakMemoryAllocator. As a reminder the way to create an OakMap instance is to create an OakBuilder and to use its *build()* method later. Here is the code example:

```
OakMapBuilder<K,V> builder = new OakMapBuilder()
.setKeySerializer(new OakKeySerializerImplementation(...))
.setValueSerializer(new OakValueSerializerImplementation(...))
.setMinKey(...)
.setKeysComparator(new OakKeyComparatorImplementation(...))
.setMemoryCapacity(...);
```

OakMap<K,V> oak = builder.build();

In the above example an user-defined MemoryAllocator wasn't used, thus any OakMap build by this builder will be created with the Oak's native OakMemoryAllocator. In order to create a builder with user-defined MemoryAllocator see the code example below:

OakMapBuilder<K,V> builder = new OakMapBuilder() .setKeySerializer(new OakKeySerializerImplementation(...)) .setValueSerializer(new OakValueSerializerImplementation(...)) .setMinKey(...) .setKeysComparator(new OakKeyComparatorImplementation(...)) .setMemoryCapacity(...) .setMemoryAllocator(new SpecialMemoryAllocator(...));

OakMap<K,V> oak = builder.build();