UNIVERSITETET I AGDER

FAKULTET FOR ØKONOMI OG SAMFUNNSVITENSKAP

E K S A M E N

Emnekode:	**IS-207**
Emnenavn:	**Algorithms and Data Structures**
Dato:	**22 May 2015**
Varighet:	**0900-1300**
Antall sider inkl. forside:	8
Målform:	**English**
Tillatte hjelpemidler:	**Any books or notes**

Merknad [TM1]: Does not update automatically for me...

NOTE:

You will be asked to write a number of methods that shall work within an existing class. However, you may add fields and methods to the existing classes, and even new classes if you need them. Whichever you choose, make clear what code belongs to which class and document all code as far as needed. Of course, be brief and do not document self-explanatory code!

Take time to read the provided information before you start writing code. If you are unable to write Java code, do not leave the question unanswered. If you can give code fragments, write pseudocode, or provide some other form of design, it is much better than nothing.

Background – Garbage Collection

The setting for this exam is garbage collection. When a program uses a part of the computer memory, for example to store an object, that part of the memory cannot be used for anything else before it has been marked as free for reuse. If programs did not release unused memory, the computer would run out of memory very soon.

Many programming languages require the programmer to explicitly allocate and release memory. This gives the programmer detailed control over memory usage, but introduces the risk of creating some very hard to find bugs (*memory leaks*). Programming languages with automatic memory allocation, like Java, are more convenient for a programmer. They do depend, however, on an efficient implementation of the algorithms we are about to explore.

Objects are allocated from an area of memory that is traditionally called the *heap*. (This is not the same thing as the `Heap` data structure you have learned to use as a priority queue.) The memory heap will usually contain a mix of objects and unused blocks of memory. The manager must keep the unused blocks in a data structure, often called the *free list* – but it doesn't have to be a list. When the memory manager is unable to find memory for a new object, it will start a process called *garbage collection*, which finds objects that the program cannot use anymore, and returns them to the free list.

Mark-and-sweep Garbage collection

Mark-and-sweep garbage collectors are the easiest to implement. They consist of two phases, called *mark* and *sweep*. The purpose of the mark phase is to find all objects that the program may possibly use in the future, and mark them. Then all other objects can be recycled in the sweep phase.

Unused memory is kept in a *free list*, which is a single linked list of unused memory blocks. There is a pointer to the first free block, and each block has a pointer to the next free block. When the program tries to create a new object, the free list is searched for a block of memory that is large enough to hold the object. If one is found, this block is allocated to the object, and taken out of the free list. If the block is too large, it is split into one block allocated to the object, and a free block which remains on the free list. This is illustrated in Figure 1.

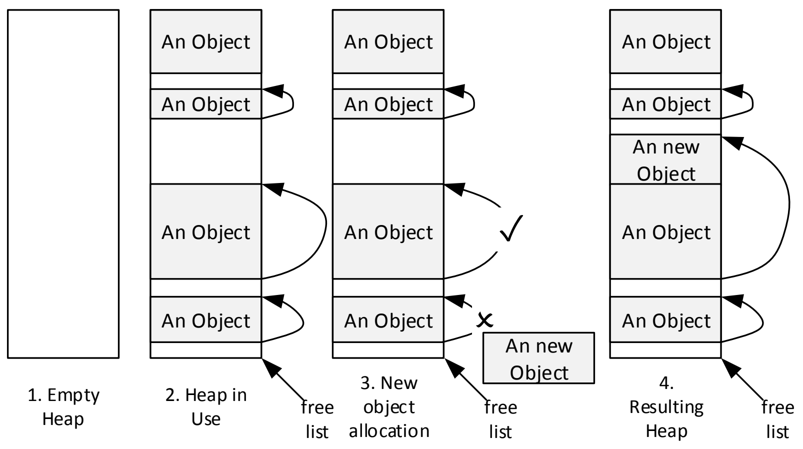

Figure 1: Using a free list to allocate memory

If the free list is empty, or none of the memory blocks on the free list are large enough for the object, garbage collection is initiated, and the mark phase begins.

The mark phase
The purpose of the mark phase is to find and mark all objects that can be reached directly or indirectly from a set of root variables. The *root set* typically includes at least all local variables on the call stack, and any static variables. The mark algorithm then becomes:

- Mark all objects as garbage (with a little help from the memory allocation algorithm this step can be skipped).
- Mark the object pointed to by a variable in the root set as usable.
- Mark the objects that can be reached from the root objects as usable, and then the objects that can be reached from them, and so on.
- Usable objects should only be marked once, there may be more than one path from root to an object, and there may be loop structures.

The sweep phase
When the mark phase is complete, we can start reclaiming unused memory. This is done by visiting all memory blocks, and adding the ones that are marked as garbage to the free list. We can assume that the memory blocks are contiguous: Each block starts immediately after the previous one, so we can get the address of the next block by adding the size of the current block to its address.

Simplified memory model

Writing a “real” garbage collector for a computer can be complicated, so we will make some simplifications.

- The heap is represented by an integer array.
- Pointers (references) to objects are indices into the heap array. In other words a variable that references an object contains the index of the start of the object.
- The root set contains a single variable: root.
- All objects contain exactly two pointers. (This is equivalent to limiting Java objects to have only two fields that point to other objects.) The pointers have fixed offsets to make them easy to find.

- The remaining cells occupied by an object do not have any other purpose than to make it possible to create objects of different sizes.

The objects layout in our simplified memory is:

Field	**Offset**	**Description**
size	0	Total size of the object including the header.
flags	1	Used to mark objects.
nextFree	2	Used to link together unused memory blocks in the free list.
pointer1	3	The first field that points to another object.
pointer2	4	The second field that points to another object.
data	5	This is not used for anything other than taking up space.

The object starts with three header fields: `size`, `flags` and `nextFree`. They are used by the memory manager and are not visible to the program. The rest (`pointer1`, `pointer2`, and `data`) represent the actual Hava object.

A variable that points to the object contains the index of the `size` field.

Memory blocks on the free list have the same header fields as objects.

You will find a Java class `Heap` that implements this model in appendix A.

Problem 1: Mark and sweep GC (counts 20+10%)

The mark and sweep garbage collector is implemented as a subclass of Heap. Appendix B contains the skeleton code for the class. The `mark()` and `sweep()` methods must be finished.

a) Write the `mark()` method according to the description above!

b) Write the `sweep()` method!

Compacting garbage collection

One problem with mark-and-sweep garbage collection is that the free blocks of memory tends to become smaller, because large blocks will be split up when memory is allocated to small objects, and it becomes increasingly difficult to find a block of memory for large objects. (see Figure 1: the object does not fit at the first possible position but at the second position leaves a small gap of free memory.) Compacting garbage collectors address this by moving the usable objects to the start of the heap. This will create a big unused memory block at the end. Now we can replace the free list with a pointer to the start of the free area. When new objects are allocated at the start of the free area, we just have to add the object size to the free area pointer.

A compacting garbage collector consists of four phases: *Mark*, *Calculate addresses*, *Update pointers*, and *Move objects*. The mark phase is identical to that described for the mark and sweep algorithm. The second and third phases update all pointers in the program, so the program will still work after moving the objects.

Calculate addresses and update pointers

When the garbage collector moves an object, it must also set all the variables that point to it to point to its new location. This is done by looping through all blocks of memory. All blocks marked as usable will be moved. We are going to pack the objects densely at the start of the heap, starting at address 0. Each object that is moved will require a block of memory equal to its size, so the new address an object will be the sum of sizes of the objects that were processed before it. The new address must be stored in the object. It will be needed twice: First to update the variables that point to it, and second when we actually move the object.

To update the pointers, we traverse the heap once again. If a usable object contains pointers to other objects, they must be set to the new location of the other object.

Move phase
Finally we can move the objects to their new location. We traverse the heap once more, and move each object to its new location. At the end of the move phase, there will be a large block of free memory from the end of the last object to the end of the heap.

The whole process is illustrated in Figure 2.

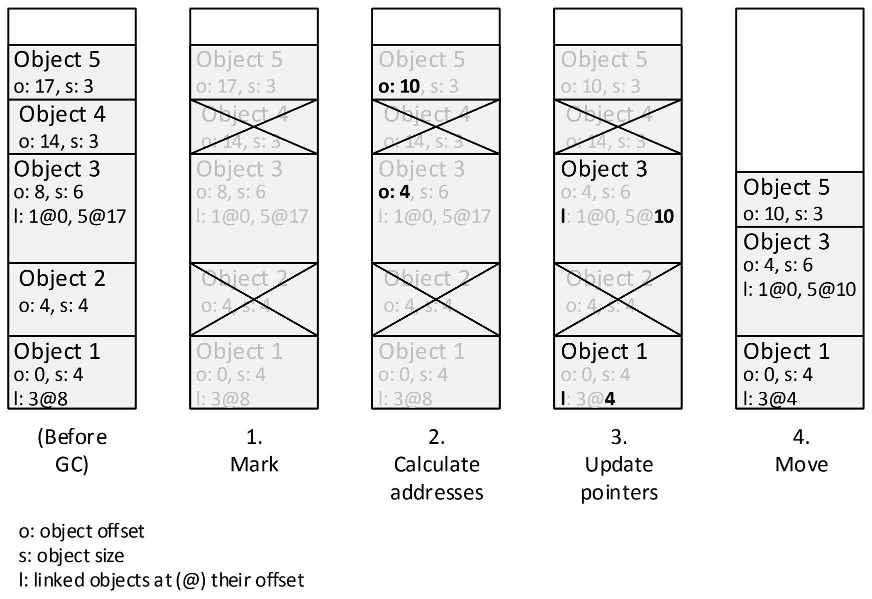

Figure 2: Mark and Compact Example

Problem 2: A compacting garbage collector (counts 10+10+20%)

Class `CompactingGC`, in appendix C, is a partial implementation of a compacting garbage collector.

a) Write the `calculateAdrresses()` method! You can use the `next` field in the object headers to store the new address of each object.

b) Write the `updatePointers()` method! For each usable object you must change the addresses in `ptr1` and `ptr2` to the new address of the objects they point to.

c) Write the `moveObjects()` method! Traverse the heap a final time, and move each usable object as far towards the start of the heap as possible.

Merknad [TM2]: Should we really use toe short version here? This is oncosistent with the usage for mark and sweep.

Problem 3: Discussion (counts 30%)

For each of the methods above, describe which data structures you have used, and why. You may also comment on the structures that were given in the problem set. You should also consider alternatives. Could any of the other structures taught in the lectures have been used? Why (not)?

Appendix A – class Heap

```
public class Heap
{
    public static final int HEAP_SIZE = 100;
    public static final int NULL = -1;         // null pointer

    // flags
    protected static final int FREE = -2;      // on  the free list
    protected static final int REACHABLE = -3; // in use
    protected static final int GARBAGE = -4;   // potential garbage

    // offsets from start of object
    protected static final int SIZE_OFFSET = 0;
    protected static final int FLAG_OFFSET = 1;
    protected static final int NEXT_OFFSET = 2;
    protected static final int PTR1_OFFSET = 3;
    protected static final int PTR2_OFFSET = 4;
    protected static final int DATA_OFFSET = 5;
    protected static final int HEADER_SIZE = PTR1_OFFSET;

    /**
     * This array represents the memory that is available to
     * the program for object creation.
     */
    int[] memory;
    int root;

    public Heap(int size) {
        memory = new int[size];
    }

    public void setRoot(int addr) {
        root = addr;
    }

    // Convenience getters and setters for the objects
    // You can assume there are similar getters and setters
    // for flag, next, ptr1, ptr2. You do not need to write
    // them.
    public int getSize(int objAddr) {
        return memory[objAddr + SIZE_OFFSET];
    }

    public void setSize(int objAddr, int size) {
        memory[objAddr + SIZE_OFFSET] = size;
    }
}
```

Appendix B – class MarkAndSweepGC

```
public class MarkAndSweepGC extends Heap
{
    /**
     * pointer to the first element in the free list
     */
    int freeList;

    public void setRoot(int root) {
        this.root = root;
    }

    /**
     * This method allocates memory, and calls the "constructor"
     * when the program creates a new object.
     */
    public int alloc(int size, int ptr1, int ptr2, String data) {
    }

    /**
     * This method is called from alloc(), when there is no
     * (large enough) block of memory on the free list
     */
    public void gc() {
        mark(root);
        sweep();
    }

    /**
     * Finds all usable objects, starting from root, and following
     * non-NULL values in ptr1 and ptr2
     */
    private void mark(int objAddr) {
        // problem 1a
    }

    /**
     * Returns unusable objects to the freeList, so the memory
     * blocks can be reused for other objects.
     * Here and in any other problem where you have to visit every
     * memory block in the heap, you can assume that the first
     * block has address 0 (zero), and that the address of the
     * next block is addr + getSize(addr), where addr is the address
     * of the current block */
    private void sweep() {
        // problem 1b
    }

}
```

Appendix C – class CompactingGC

```
public class CompactingGC extends Heap
{
    /**
     * Pointer to the start of the free area
     */
    int free;

    public CompactingGC(int size) {
        super(size);
        setFree(0);
    }

    /**
     * Allocate memory for an object
     */
    public int alloc(int externalsize, int ptr1, int ptr2) {
        int size = externalsize + HEADER_SIZE;
        if (size > HEAP_SIZE - free) {
            gc();
            if (size > HEAP_SIZE -free)throw new OutOfMemoryError();
        }
        int addr = free;
        setSize(addr, size);
        setNext(addr, NULL);
        setFlag(addr, GARBAGE);
        // should have called constructor here
        setFree(addr + size);
        return addr;
    }

    /**
     * move the start of the free area
     */
    private void setFree(int addr) {
        free = addr;
        setSize(free, memory.length - free);
        setFlag(free, FREE);
    }

    public void gc() {
        mark(root);
        int newFree = calculateAddresses();
        updatePointers();
        moveObjects();
        setFree(newFree);
    }

    /** calculate the addresse each object will be moved to */
    private int calculateAddresses() {
    }

    /** update all addresses with the new addresses of the objects */
    private void updatePointers() {
    }

    /** move the objects to their new address */
    private void moveObjects() {

    }
}
```