Worst-fit dynamic partition simulator

For this assignment, you will write a worst-fit dynamic partition memory allocation simulator that approximates some of the functionality of `malloc()` and `free()` in the standard C library. The input to your simulator will be a page size (a positive integer) and list of allocation and deallocation requests. Your simulator will simulate processing all requests and compute some statistics.

Throughout the simulation your program will maintain an ordered list of free and occupied partitions. Every partition will contain its size and the starting address, and the starting address of the first partition should be 0. Occupied partitions will have a numeric tag attached to them, representing their owner. Your simulator will manipulate this list of partitions as it processes the requests. Allocation requests will be processed by finding the most appropriately sized partition and then allocating a memory from it. Deallocation requests will free up any relevant occupied partitions, and merge adjacent free partitions if necessary.

The only file you should modify and submit for grading is `memsim.cpp`, in which you need to implement the function:

```
MemSimResult mem_sim(int64_t page_size, const std::vector<Request> & requests);
```

The parameter `page_size` will denote the page size and `requests` will contain a list of requests to process. The requests are described using the `Request` struct:

```
struct Request { int tag; int size; };
```

When `tag>=0`, then this is an allocation request, and the `size` field will denote the size of the request. If `tag<0` then this is a deallocation request, in which case the `size` field is not used. You will report the results of the simulation by returning an instance of `MemSimResult` structure.

Allocation requests

Each allocation request will have two parameters – a tag and a size. Your program will use **worst-fit algorithm** to find a free partition, by scanning the list of partitions from the start until the end. If more than one partition qualifies, it will pick the first partition it finds (i.e. the one with the smallest address).

If the partition is bigger than the requested size, the partition will be split in two – an occupied partition and a free partition. The tag specified with the allocation request will be stored in the occupied partition.

Your simulation will start with an empty list of partitions. When the simulator fails to find a suitably large free partition, it will simulate asking the OS for more memory. The amount of memory that can be requested from OS must be a multiple of `page_size`. The newly obtained memory will be appended at the end of your list of partitions, and if appropriate, merged with the last free partition. Your program must figure out what is the minimum number of pages that it needs to request in order to satisfy the current request.

Deallocation requests

A deallocation request will have a single parameter – a tag. In the input list of requests, this will be denoted by a negative number, which you convert to a tag by using its absolute value. To process a deallocation request, your simulator will find all occupied partitions with the given tag and mark them free. Any adjacent free partitions must be merged. If there are no partitions with the given tag, your simulator will ignore such deallocation request.

Pseudocode for processing allocation requests:

```
- search through the list of partitions from start to end, and
  find the largest partition that fits requested size
    - in case of ties, pick the first partition found
- if no suitable partition found:
    - get minimum number of pages from OS, while considering
      the case when last partition is free
    - add the new memory at the end of partition list, merge
      if appropriate
    - the last partition will be the best partition
- split the best partition in two if necessary
    - mark the first partition occupied, and store the tag in it
    - mark the second partition free
```

Pseudocode for processing deallocation requests:

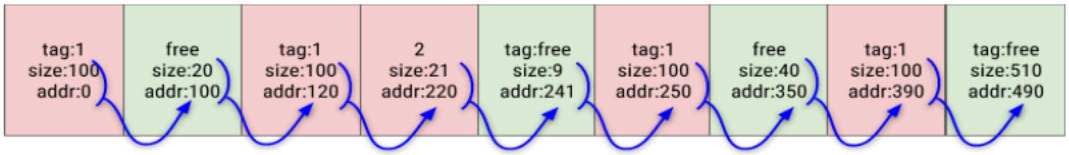
```
- for every partition
    - if partition is occupied and has a matching tag:
        - mark the partition free
        - merge it with adjacent free partitions
```

Partition addresses

For each partition you need to store the starting address (`addr`) of the block of memory that the partition represents. The first partition should have `addr=0`. The `addr` of any other partition is equal to the previous partition's `addr` plus the previous partition's `size`. If you store partitions in a linked list, such as `std::list<Partition>`, and if `cptr` is an iterator to some partition, then you can calculate its `addr` as:

```
cptr-> addr = std::prev(cptr)-> addr + std::prev(cptr)-> size;
```

Another way to think about a partition's address is that it is the sum of sizes of all partitions preceding it.



The driver program (main.cpp)

The included driver will accept a single command line argument representing the page size in bytes.

The driver will read allocation requests from standard input, until EOF. Lines containing only white spaces will be skipped. Each non-empty line will represent one request, either allocation or deallocation.

Any line with two integers will represent an allocation request. The first integer will represent the tag of the request, and the second one will represent the size of the allocation request in bytes. For example, the line `"3 100"` represents an allocation request for 100 bytes with tag 3.

A line with a single negative integer will denote a deallocation request. The absolute value of the integer will represent the tag to be deallocated. For example, the line `"-3"` will represent a deallocation request for all partitions marked with tag 3.

Reporting Results

At the end of the simulation your simulator must return an instance of `MemSimResult` structure:

- Set `n_pages_requested` to the total number of pages requested during the simulation. Notice that this could be `0`, if there are no allocation requests in the input.
- Set `max_free_partition_size` to the size of the largest free partition at the end of the simulation. If there are no free partitions, set this to `0`.
- Set `max_free_partition_address` to the address of the largest free partition at the end of the simulation. Set this to `0` if there are no free partitions. In case of ties, set this to the smallest address.

Limits

- the number of requests will be in range [0 .. 1,000,000]
- `page_size` will be in range [1 .. 1,000,000]
- each request's `tag` will be in range [-10,000,000 .. 10,000,000]
- each request's `size` will be in range [1 .. 10,000,000]

Marking

- Your code will be marked for correctness and efficiency.
- Your mark will be based on the number of test cases your solution will pass.
- A simple $O(n^3)$ solution, e.g. one that uses a vector to represent partitions, will likely only earn about 55% of marks.
- A simple $O(n^2)$ solution, e.g. one that only uses a linked list to represent partitions, will likely earn about 75% of marks.
- To earn full marks, you will need to be able to process any input with 1 million requests under 10s. I suggest you implement an $O(n \log n)$ algorithm, using advanced data structures as described in the hints section below.

Make sure you design your own test inputs to verify your code runs correctly and efficiently.

Sample input and output

```
$ cat test1.txt
5 100
-5
-6
1 100
2 20
1 100
2 30
1 100
2 40
1 100
-2
2 21
-1
3 220
3 759
3 1
3 5900
```

```
$ ./memsim 1000 < test1.txt
pages requested:                8
largest free partition size:    829
largest free partition address: 7171

$ ./memsim 1 < test1.txt
pages requested:                7030
largest free partition size:    129
largest free partition address: 221

$ ./memsim 33 < test1.txt
pages requested:                214
largest free partition size:    129
largest free partition address: 221
```

Few sample test files are provided in the GitLab repository. Make sure to design your own test files as well. Read the appendix to learn how you can obtain correct results for small inputs.

Appendix - Hints

If you use only basic data structures, such as linked lists or dynamic arrays, you will likely end up with an $O(n^2)$ or even $O(n^3)$ algorithm, which will make your program too slow for large number of requests. To get full marks, you will need to use smarter data structures. I suggest:

- `std::list` - a linked list to maintain all partitions (to make splitting and merging of blocks constant time operation)
- `std::unordered_map` - hash table to store all partitions belonging to the same tag
  - the data you store here are just pointers (iterators) to the linked list nodes
  - this will allow you to process deallocation requests very fast
- `std::set` - a balanced binary tree to keep track of free blocks, sorted by size
  - you need to store linked list iterators in this tree (aka pointers to the linked list nodes)
  - you should sort the tree by partition size (primary key), and partition address (secondary key)
  - this will allow you to process allocation requests very fast

If you use the above data structures correctly, you should be able to process every request in $O(log\ n)$ time. Here are some relevant parts of code that use the above data structures:

```
struct Partition {
  int tag;
  int64_t size, addr;
};

typedef std::list<Partition>::iterator PartitionRef;

struct scmp {
  bool operator()(const PartitionRef & c1, const PartitionRef & c2) const {
    if (c1->size == c2->size)
      return c1->addr < c2->addr;
    else
      return c1->size > c2->size;
  }
};

struct Simulator {
  // all partitions, in a linked list
  std::list<Partition> all_blocks;
  // quick access to all tagged partitions
  std::unordered_map<long, std::vector<PartitionRef>> tagged_blocks;
  // sorted partitions by size/address
  std::set<PartitionRef,scmp> free_blocks;
 ...
}
```

The `free_blocks` will keep the linked list pointers sorted so that `free_blocks.begin()` will always return the largest partition, and in case of ties, it will return the partition with the smallest address.

Before you modify a partition, make sure you `free_blocks.erase()` the partition. If you need to keep it in `free_blocks`, you can `free_blocks.insert()` it back after the modification. If you do not do this, you will likely corrupt the tree, and your program will output wrong results and/or crash. For similar reasons, make sure you never remove a partition from the linked list **before** removing it from `free_blocks` or `tagged_blocks`. When you remove an entry from linked list, any iterators to it would become invalid and lead to undefined behavior (likely a crash).

How to start

Step 1 – Start by implementing a basic solution, only using linked lists.

Step 2 – Add support for `tagged_blocks`, which will speed up deallocation requests. You will be able to re-use most of the code from step 1.

Step 3 – Add support for `free_blocks`. This will make allocation requests much faster. You should be able to re-use most of the code from Steps 1 and 2.

Debugging

I suggest you perform a consistency check of your data structures after processing each request. I found the following useful when I was debugging my own solution:

- make sure the sum of all partition sizes in your linked list is the same as number of page requests * page_size
- make sure your partition addresses & sizes are consistent
- make sure the number of all partitions in `tagged_blocks` + the number of partitions in `free_blocks` is the same as the size of the linked list
- make sure that every free partition is in `free_blocks`
- make sure that every partition in `free_blocks` is actually free
- check the return values of calls to `free_block.erase()` and `free_block.insert()` to make sure they work as you intended, e.g.:

```
auto res = free_blocks.erase(p);
assert(res == 1);
```

```
auto res = free_blocks.insert(p);
assert(res.second);
```

Important: these consistency checks will make your code run slower. You should disable these checks before you submit your code for grading.