December 3, 2020 Tutorial – Tips for Assignment 5 Question 1

(**SPEAKING NOTES**. Contains slides / screen shots from assignment pdf by Dr. Pavol Federl).

Last time in tutorial we talked about data structures for **dynamic partition** memory allocation implementation/*simulation*. (Lecture slide 16-memory, slide 34)

Purpose	Data Structure
Structure 1. List of all allocated and free memory partitions	Linked list (or doubly linked list). In C++: `std::list`
Structure 2. Keeping track of free/empty partitions	Binary balanced trees. In C++: `std::set`
Structure 3. Keeping track of tagged partitions	Hash table. In C++: `std::unordered_map`

Notes on the above data structures:

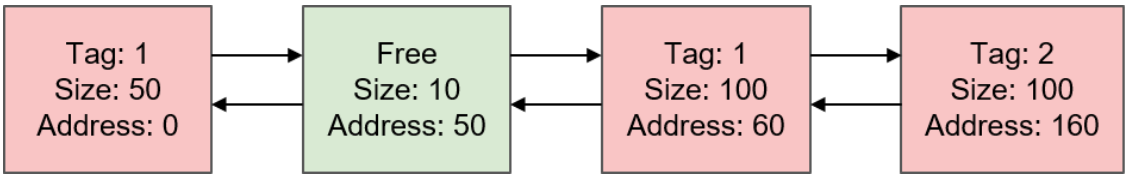
Structure 1. How to represent blocks/partitions of memory? Create a struct!

struct Partition, containing three integers

- Tag
- Size
- Address

```
struct Partition {
  long tag ;
  long size ;
  int64_t addr ;
};
```

Where "Address" is the the starting "address" (in the simulation) of the block of memory that the partition represents.



These partitions will be the "nodes" in your doubly linked list (i.e. your std::list contains Partition structs).

```
// all partitions (so that I can get std::prev() and std::next() nodes
std::list<Partition> partitions;
```

Side note: when using list, you may be using the iterator frequently. And so, it may be useful to use a create an alias for the iterator as shorthand, you can do this using typedef:

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

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

Please write this down as I use "**PartitionRef**" throughout this document…

Structure 2. Goal is to have quick access to free/empty partitions. This binary balanced tree, or std::set, should contain the iterator to Partitions that are free/empty. In other words, the std::set should consist of PartitionRef. In best fit, you will want to allocate request to the *smallest* free/empty partition that is big enough for the request: you are told that if there are multiple possible free/empty partitions that fit that criteria, you should allocate the request to the one with the smallest address. And so, you should have std::set sorted *by size* and *then by address*. How can you do this? Using a custom compare function.

Repl demo: https://repl.it/@michelleeeeeeee/List-and-Set-Examples-A5

```
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;
  }
};
```

and

```
// sorted partitions by size/address
std::set<PartitionRef,scmp> free_blocks;
```

Structure 3. Goal is to have quick access to tagged chunks. You can do this with std::unordered_map, by mapping the tag to a std::vector of PartitionRef (iterators to a partition).

```
// quick access to all tagged partitions
std::unordered_map<long, std::vector<PartitionRef>> tagged_blocks;
```

Assignment 5: memsim.cpp

Instructor already has a struct called Simulator going: you can use it as a starting point.

Recommendation: making **Structure 1, 2, 3** data elements in the Simulator(!)

```
struct Simulator {
  // all partitions (so that I can get std::prev() and std::next() nodes
  std::list<Partition> partitions;
  // sorted partitions by size/address
  std::set<PartitionRef,scmp> free_blocks;
  // quick access to all tagged partitions
  std::unordered_map<long, std::vector<PartitionRef>> tagged_blocks;

 ...
}
```

The Simulator constructor:

- Optional: You can start by setting up 1 free/empty partition of size 0 and adding it to your **Structure 1** and **Structure 2**

The Simulator allocate function:

Parameters: tag and size of the request

Reminder: you are tasked with implementing Best-fit algorithm!!

> From lecture slides: "best fit - find the smallest hole that is big enough, leftover (tiny) space becomes new hole" (show students 16-memory, slide 36)

Step 1. Get an iterator to the first (i.e. smallest address) smallest free/empty partition that is big enough for the request size. (hint: create a dummy linked list containing a dummy partition with size equal to request; then use the std::set member function lower_bound. Note to self: show students repl)

- **Repl demo:** https://repl.it/@michelleeeeeeee/List-and-Set-Examples-A5
  - **Using lower_bound**

Step 2.

If *you can find* a smallest free/empty partition that is big enough, then awesome: just allocate that partition to the request (and update any structures 1, 2, or 3).

If *not*, you need to request a page (or possibly multiple pages) before allocating.

- Make sure that if the last block was a free/empty partition, it's combined with the incoming free/empty partition (from the page request)

In either case, when you allocate, be careful. If the free/empty partition was larger than the requested size, you will need to split the free/empty partition so that only the needed size is given to the request/tagged.

The Simulator deallocate function:

Parameters: tag (i.e. any partition with this tag should be freed/deallocated)

- Use Structure 3 to get a vector of all blocks with the tag
- Because that vector is a bunch of iterators/PartitionRefs, you can then access the appropriate partitions and mark them as free.
    - After marking a partition as free/empty, check the neighbours and "merge" them if they are also free/empty partitions (Note to self: tell hint to students)
    - Note that PartitionRefs is an iterator at some partition in Structure 1 (the std::list). So you can make use of std::next() and std::prev() to get neighbouring nodes. (Note to self: show students repl)
        - **Repl demo:** https://repl.it/@michelleeeeeeee/List-and-Set-Examples-A5

The Simulator getStats function:

- Report number of pages requested and largest free partition. See assignment specs.

Tell students to read appendix:

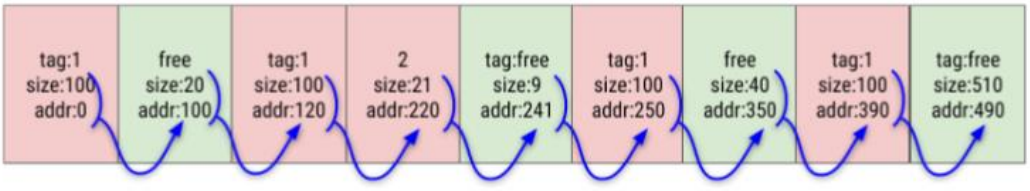
Partition address

The `partition.addr` represents the starting address of the block of memory that the partition represents. The first partition should have `addr=0`.

If a partition is in a linked list, eg. in a `std::list<Partition>`, and if `cptr` is an iterator to this partition then you can calculate its address as:

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

In other words, the address of the partition should be previous partition's address plus the previous partition's size. Another way to think about a partition address is that it is the sum of sizes of all partitions preceding it.



The reason you may want to keep track of partition addresses is because of the requirement that the best fit should choose the first partition in the list in case of ties. That means you need to sort your balanced binary tree by size, but in case of ties you sort it by address (secondary key).