CSCI 2100: Course Project (Fall 2023)

Absolute Due Date For Grade Assignment:

11:59 p.m., 20 December 2023

(Late penalty: 25% of maximum score

deducted for each 24 hours being late)

Goals of Course Project

1. To conduct the design and performance analysis of a solution for solving an application problem using the data-structure concepts learned in CSCI2100. Emphasis will be placed on both program development and analysis of solutions.

2. Programming is an essential element of your project. As there are many existing implementations of data-structure algorithms on the Internet, it would not be helpful for you to re-implement such algorithms (e.g. `https://www.sanfoundry.com/c-programming-examples-data-structures/`). Instead, we will provide you with the programs on data structures and ask you to solve an application problem using these programs.

3. In the assignment, a good design and an evaluation of your solutions are equally important and will be graded accordingly. Your submission will be graded according to the following guidelines:

   - Tests passed (35% total score): this part is related to the correctness of the outputs and the performance of your program (points will be deducted if the running time is long or the program generates incomplete/wrong outcomes);
   - Report and analysis of results (35% total score):

- (12%) The design of your program, including a clear illustration of the data structures used and their algorithms. Use one or more figures to explain your design.
- (15%) The tradeoffs made in your design.
  * The tradeoffs include the parameter combinations studied in your implementation and a detailed analysis of your design choices.
  * Examples of tradeoffs include the hash table size and the hash function parameters used.

  Discuss with justifications the best choice of the parameters to support the application. Analyze the results obtained and propose other approaches to improve them.

  A high score will be given only on a very detailed analysis.
- (5%) A summary of the amortized performance in processing the queries, including the space/time complexity of your algorithm, the total time, and the amortized times in generating your results on the test data for each type of query. Also, show a summary of the amortized performance relative to the case of a hash table of one entry. You should discuss the effects of your tradeoffs on solution times.
- Readability of the report (points will be deducted for poor readability).
- (3%) A VeriGuide report on your submission.

- Program and functions implemented with proper documentation and comments (30% total score)
  - (15%) Implementing the hash tables that support insertions, deletions, and querying.
  - (15%) Quality of program code includes proper documentation and instructions for the teaching assistants to compile and run your program. Provide a README file on how to compile and

run your program. Also, explain the outputs generated by your program.

4. Your submission (through Blackboard) should have the following parts:
   - A short (2-3 page) typed report. We do not accept handwritten submissions.
   - A zipped directory containing all your programs and outputs and a README file to explain the structure.
   - A VeriGuide report on your report.

Project Specifications: A Garbage Collection System in Memory

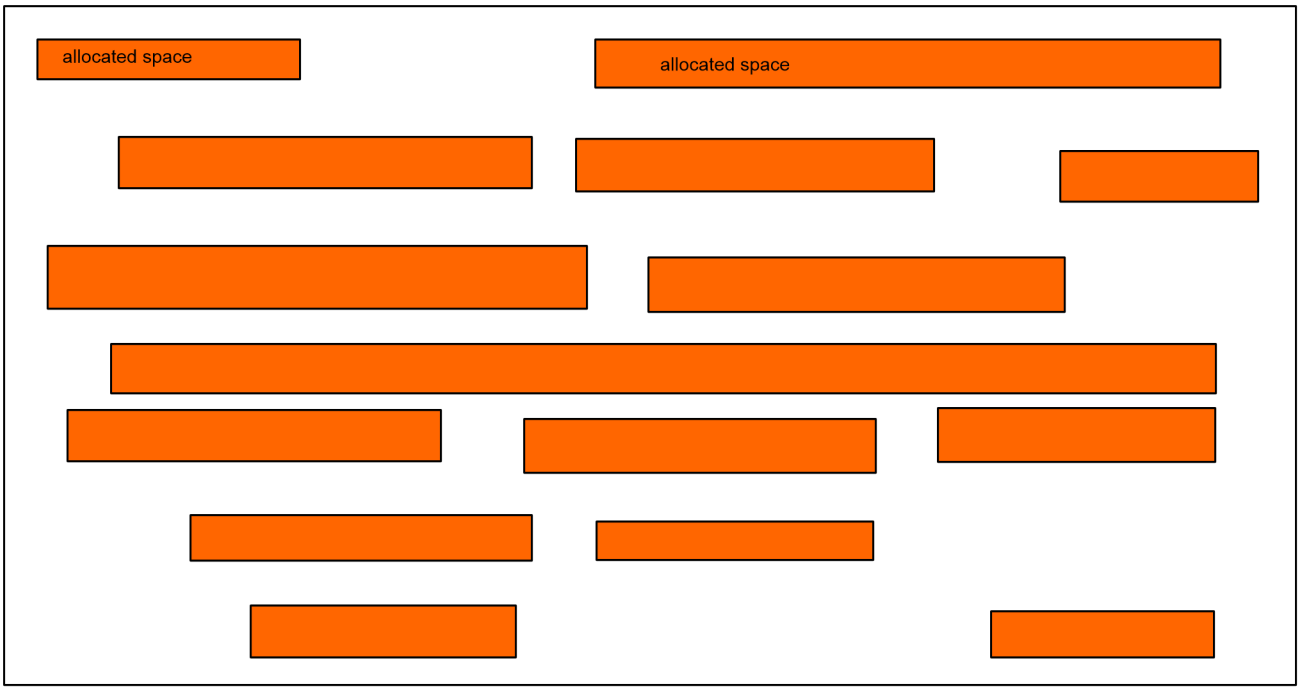
1. You are given a sequence of memory-block requests to a memory space. Each query takes the form of either a space request (of a specified number of bytes that may include an optional starting byte address) or a space release (with a defined starting byte address and its corresponding number of bytes).

memory request

memory release

2. Memory is fragmented. That is, there are allocated blocks of space and free blocks. Once a block is issued, it is impossible to move the allocated space around and coalesce the free fragments into some contiguous blocks of free space until the area is released.

fragmented memory showing blocks allocated earlier



3. There are three allocation strategies for allocating free space.

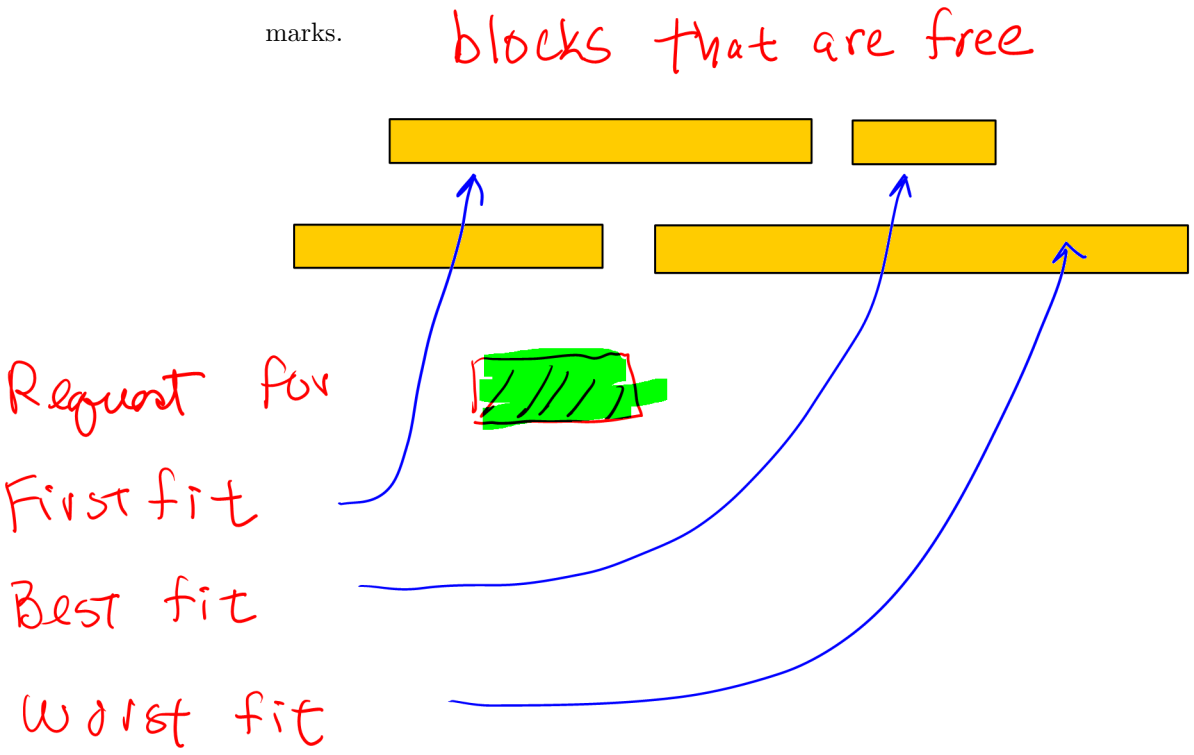
   (a) First-fit: find the first available free block of space that fits the request, with possibly some left-over unused space.

   (b) Best-fit: find the best available free block of space that leaves the least unused space. This strategy requires finding the block of free space with the minimum wasted free space after allocation. (The goal is to minimize the wasted free space. The downside is that it leads to many small blocks of free space in the memory.)

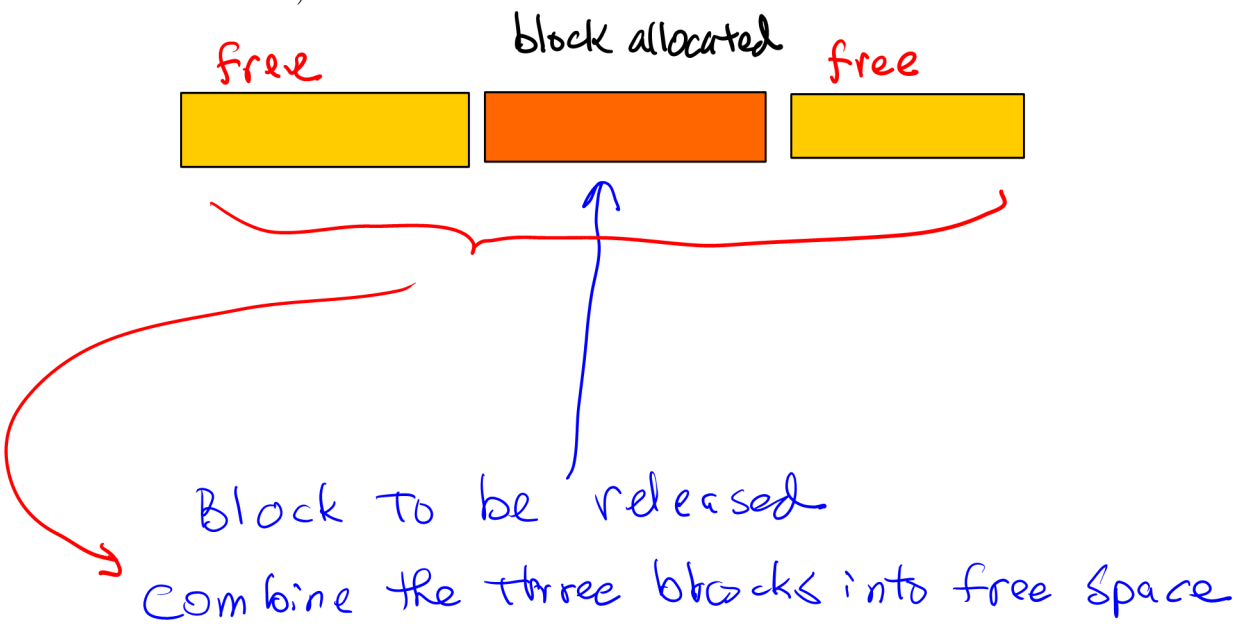
   (c) Worst-fit: find the best available free block of space that leaves the most unused space. This strategy requires finding the block of free space with the maximum unused space after allocation. (The goal is to keep the unallocated blocks of free space as large as possible.)

   In each case, after serving the space request, the left-over free space goes back into the pool of free-space blocks. Note that the same strategy is applied across all requests. After evaluating the benchmark requests for

one method, switch to a different method and re-evaluate the same benchmarks.



4. When a block of memory space is released, it returns to the pool of free blocks. Each block consists of a starting address and its corresponding size. When a memory block returns to the pool, you must merge adjacent free-space blocks into a single contiguous block of free space (if such blocks exist).



5. You will design a data structure based on hashing to support the allocation and release of space requests.

6. We will provide you with the following program files that can be downloaded from Blackboard (as a zip file)

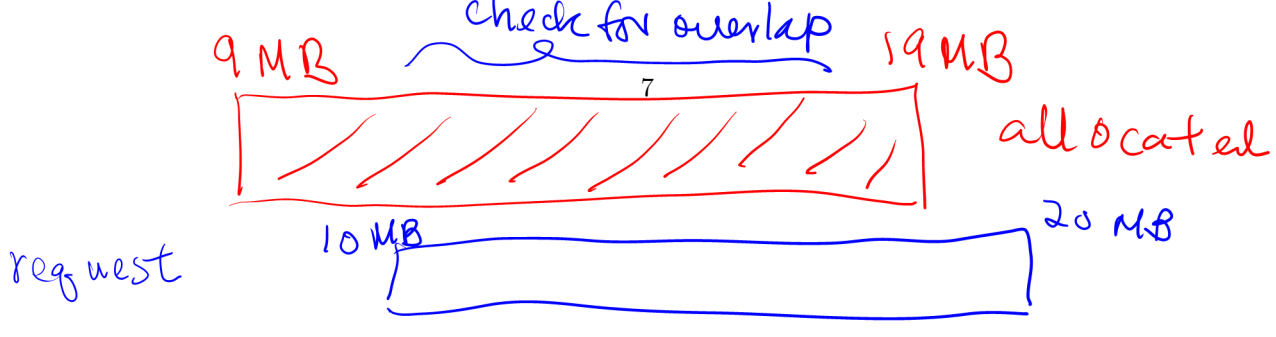
   - *HashTable.py, HashTable.cpp, HashTable.java*: a reference code for the basic implementation of the hash table,
   - *MemoryOperation.py, MemoryOperation.cpp, MemoryOperation.java*: for storing the input memory instructions (memory requests/releases),
   - *MemoryManager.py, MemoryManager.cpp, MemoryManager.java*: the class of MemoryManager that defines interfaces of high-level methods you will implement,

- *Test.py, Test.cpp, Test.java*: codes for reading the test cases from files and testing your program.

You may also use any existing source codes online provided that you cite the source URL properly.

7. The following are the parameters in the implementation.

   - Assume a physical memory space of 4GB for supporting memory-block requests/releases.
   - Each memory-block request consists of an optional starting address and a size parameter (32 bits). If the starting address is absent, the block can be allocated to any large enough free space. On the other hand, a memory-block release has a defined starting address and a size parameter (each 32 bits). One can derive the ending memory address based on the starting address and the size.
   - A memory map defines the allocation of the 4GB physical memory space. A naive solution is to define a memory map as large as the physical memory space that shows the allocation of every byte in the physical memory. However, this is impossible, as the space required for the map is too large. A more efficient solution is to define a memory map based on the allocated and the free blocks.
   - You will need to know the starting address and the size of each (allocated or free) block in the memory map. To access the memory map efficiently, you must set up some hash tables to index into the logical memory map.
   - For a memory-block request, assuming some entries already exist in the logical memory map, you will need to find a suitable block of memory space based on the defined memory-block allocation strategy. For example, in the best-fit strategy, you will find a block of free space that incurs the least amount of unused space after allocation. You may use a hash table to index the sizes of the free blocks to



access a suitable block for allocation. Since a request may also include an optional starting address, you will need another hash table to manage the starting addresses of the free blocks and a third hash table to index the blocks allocated.

- For a memory-block release, you will need to find out whether the adjacent blocks before and after the block released are free. If they are, then you must merge the blocks before inserting the single block back. You will use the hash tables designed in the last paragraph to manage the memory-block releases.

A table containing information on all the logical (free and allocated) blocks in the memory system. It can be implemented as a linked list, as each block is only accessed by a pointer in one of the hash tables. These blocks are logical because they only contain information on the corresponding phyical block, like starting address, size, ownership, protection, etc.

- You will need to check and report error conditions, such as the release of a non-existent block, the allocation of a block that is too large, or insufficient memory space.
- Your design should not be hard-coded for a given memory map. (Your program should work for a smaller or larger memory space of 4GB.)
  - Your solution should be able to adapt to the dynamic growth and shrinking of the memory map.
  - Your solution should be based on hashing only; all the queries should be run using the hash tables maintained; you should not maintain any sorted list of the keys. However, you may keep auxiliary information derived from the hash tables.
  - The hash table size may change over time as more blocks are allocated.
  - If you do not use hashing but a linear search, a binary search, or any sorted list to manage the blocks in your solution, you will get a very low score.

Suggested Solution Approach

1. To support memory-block releases and the coalescing of adjacent free blocks into large blocks, you can design two hash tables to support these functions: one indexed on the starting address of a free block and another indexed on the size of each block.
2. To support memory-block requests, you can design a third hash table indexed on the starting address of the memory block allocated. Based on the information found, you can check whether the request is valid (say, not overlapping with any existing allocated blocks). This part is essential when the starting address of the request is specified.
3. All the three hash tables point to a common database (which is actually the physical memory allocated/free blocks). That is, the hash tables are

just index tables by themselves. They are related because they all point to a shared database that stores blocks of memory allocated. Each block in the database contains information such as the physical memory address, protection status, processes owning the block, etc., otherwise not stored in the hash table. You may want to implement the shared database to avoid keeping this redundant information in each hash table.

4. For each hash table, you should create an initially empty hash table of m entries; each entry of the hash table should include fields for the forward and backward pointers of the linked list maintained by this table entry, the range of key values hashed into this location (b_1, b_2), the number of elements in the linked list pointed to by this table entry, and other fields if necessary. Note that a backward pointer is optional. It is included here for consistency with the lecture notes. An example of a hash table entry is as follows:

lower bound of key range b_1	upper bound of key range b_2	forward ptr to LL	backward ptr to LL	# entries in linked list

Each node in the linked list of the hash table entry should also include pertinent information on the memory block (such as whether it is free, its starting address, and its size. It would also have forward and backward pointers to other nodes in the linked list, if any. An example of a node in the linked list is as follows:

Backward pointer in LL	forward pointer in LL	memory block info.

ptr to memory map

5. You should populate each hash table with an initial set of memory-block allocations and releases.

6. You will then process the queries one at a time. As memory-block requests and releases with new keys may be inserted or existing records deleted, you must dynamically update the hash table as queries are processed.

7. You will report the result of each query, the times taken, and the number of logical steps (you need to decide how to measure the number of logical steps).

8. Since you need to search for keys in a specific range, you should divide the keys into ranges and use hashing on a range of keys into a unique hash

table location. Each hash table entry should contain a range of keys that can be hashed into and a pointer to a linked list of the records in this list.

9. By adequately modifying the hash tables, You must decide how to support the best-fit, the first-bit, and the worst-fit strategy.

10. You should report the following summary results over all the queries processed for each of the first-bit, best-fit, and worst-fit strategies:

    (a) amortized time of serving a memory-block request,

    (b) amortized time of serving a memory-block release,

    (c) the amortized times in (a) to (b) normalized when compared to a reference hash table of size one,

    (d) the distribution of the lengths of the linked lists in the hash tables after processing all the queries.

11. The tune-able parameters evaluated in your program are the size of a hash table (and the range of hash keys to go into each table entry) and whether you allow the hash table size to change dynamically. You may also consider tuning the parameters used in the hash function.

12. You can implement your solution using C++, python, or Java.

Zero Tolerance for Cheating. You are required to implement your solution. You may be required to attend a personal interview to explain the details of your code. All submissions must pass through VeriGuide and may be subject to additional plagiarism checks. Any confirmed plagiarism case will receive a zero mark and be reported to the university for disciplinary action.