COL106: Data Structures, Sem I 2020 – 2021

Assignments 1, 2 and 3

Dynamic Memory Allocation using Linked Lists and Trees

Introduction

This problem is based on the implementation of Doubly Linked Lists (for Assignment 1) and Binary Search Trees (for Assignment 2) and Balanced Binary Search Trees (AVL Trees for Assignment 3) in order to create a system to perform **Memory Allocation**. Now the first question that would intrigue us is *What is Memory Allocation?* Well in simple terms, it is the reservation of portions of the Computer memory for execution of processes. Thus, this system will be running on our machines and it will give out memories to the programs as requested by them (Ever heard of *malloc*?)

The above statement should get more clear once we delve into the details of this system. So mainly there are two types of Memory allocation:

- **Static Memory Allocation**: The system knows the amount of memory required in advance. From this, it can be inferred that memory allocation that would take place while defining an Array would be static as we specify it's size earlier.
- **Dynamic Memory Allocation**: The system does not exactly know the amount of memory required. And so in this case, it would get requested for memory *Dynamically.* Linked Lists creation is an example of Dynamic Memory Allocation.

We will focus on Dynamic Allocation of Memory for our Assignments. Memory in our computers is divided into different addresses. And these addresses get grouped into different blocks or segments (more on that later). Say a program demands a space of 4KB (Kilobytes), then a memory block of 4KB size is given to this program. What if some other program demands a 4KB memory and the same block gets given to it? What about the previous program which was already using that block? Think of a situation where there is an airplane with it's lavatories not having a functional vacant/occupied slider sign... disasters will happen right? The exact same will happen here too.

So this System for Memory Allocation will be having an information about the memory spaces that would be currently occupied by other programs and the memory spaces that are currently available, so that whenever a program asks for memory only the memory block that is marked **free** (or currently available) will be given to that program. Now whenever a free memory block is given to a program then it is marked as **occupied** (or **allocated**) so other programs can not access it.

Now it even happens that the program that was using this memory block realizes that it has no more work with this memory block and so it decides to release this block, enabling other programs to be able to use this block. Thus, a program can even mark a memory block as free, which then becomes available for the other programs as well. Remember, we only have a limited number of resources and if programs keep asking for memory and never free that then eventually we will run out of space and the system will crash! (Again, try linking this with the Lavatory scenario to understand these situations better)

If you were able to understand the above mentioned points, then the first question that would have struck you might be *"Well, How does the system decide which block to give if there are many free blocks available?"*. So for that there is a bunch of algorithms and strategies available in the literature, but we shall look into the following two for now:

- **First Fit Algorithm**: Here the Allocation system will go through all the Free Memory blocks, stored using any Data structure (here we focus on DLL, BST and AVL Trees), and first block that is big enough to fulfill the needs of the program will be returned. The process of finding the first such block will finish quickly in general (so it is good in time), but there will be an issue regarding the loss of memory (so it is bad in space utilisation). Say the program wants 4KB of memory and the first free block (larger than 4KB) found by the system is of the size 16KB. Then here, there is a loss of 12KB which won't be useful until the program releases this block and marks it free again!
- **Best Fit Algorithm**: Rather than finding the first adequate memory block, the system will go through all the free blocks and will return the smallest block which will be big enough to

fulfill the needs of the program. Here as it will need to go through the entire data structure of free blocks so it will be slower (so it is bad in time). But there will be a minimum loss of memory here, as it is finding the best block possible to be assigned (so it is good in space utilisation). Say in the previous scenario of the program wanting 4KB of space, there are overall four free blocks with sizes of 2KB, 12KB, 5KB, 6KB (in order). So, when used this algorithm, the system will go through all these free blocks, detect the smallest block which is larger than 4KB (in this case, it is the 5KB block), and return it to the program that requested memory.

Thus from the above two algorithms, you can see the trade-off between having the fastest algorithm for allocation and having the most efficient space utilisation algorithm for allocation. Note that along with these two criterion, another criterion which is extremely important for allocation of memory is that the free blocks that are **contiguous** should be merged into one larger free block (we will come back to this in a moment).

We know that the memory in the computers can be approximated to be an array of addresses, starting from zero. Assuming that 1KB=1024B and the size of each memory address is 1B (Byte), then we can say that the memory block with it's starting address as 0 and size as 1KB will be spanning all the memory addresses from $0 \rightarrow 1023$ (Because 1KB of memory block needs to have 1024 memory addresses). Memory block with starting address as 1024 and size 1KB will be spanning all the memory addresses from $1024 \rightarrow 2047$. For simplicity, we will say that the memory address starting at 0 with a size of 1KB will be spanning $0 \rightarrow 1KB$. Similarly, it will span $1KB \rightarrow 2KB$ for the other case and so on.

For optimal utilization of memory, it is extremely important to have all the free blocks that are contiguous to be merge as one single larger free block. For example, suppose the total memory size was $10KB$ and your program allocates 1KB blocks in a loop that runs ten times. Suppose the allocate memory blocks were $0 \rightarrow 1KB$ and $1KB \rightarrow 2KB \ldots, 9KB \rightarrow 10KB$. After this part of computation is complete, the program frees all theses ten blocks of memory. Now your free blocks list will contain ten blocks $0 \rightarrow 1KB$ and $1KB \rightarrow 2KB \ldots, 9K \rightarrow 10KB$. Now suppose your program requests for a memory allocation of $2KB$. Even though the entire memory from 0 to $10KB$ is available, you will be unable to find a free block of size $2KB$. This is so because your memory has been fragmented into smaller sized free blocks and these blocks cannot be reallocated even though the required contiguous free memory is available. In this case, fragmented memory is more likely to remain unused and thus it will again lead to wastage to space. This problem is called **Fragmentation**. Thus initially, the free memory will be long and contiguous. But over the time of use, these long contiguous regions will be fragmented into smaller and smaller contiguous areas. A **defragmentor** checks for free blocks that are next to each other and combines them into larger free blocks. Running a defragmentor periodically reduces the fragmentation of memory and avoids space wastage.

This information about memory allocation should be sufficient for the further assignments. You can look up the web for exploring these topics further more, if you wish to! We are now ready to jump into the programming assignments

Problem Situation

We will assume the memory of the system to be an array of size M (bytes), which lies between 10 million to 100 million. Each element of this array would be addressed using its index. We take the size of one memory address as 1 Byte. The Memory Allocation System will be maintaining **two data structures**, one for the **free blocks** and one for the **allocated blocks**. So initially, the data structure for allocated blocks will be empty and the data structure for the free blocks will be having just one element which is the entire memory.

Here, the system will be allocating memory using a variant of First Fit and Best Fit algorithm. These variants will be called **First Split Fit** and **Best Split Fit** algorithm. Here during the allocation, these algorithms will split the found free block into two segments; one block of size k (that is requested by the program) and the other block of the remaining size. For example, if the First Fit algorithm had returned a memory block of the range $5KB \rightarrow 10KB$ for a request of 2KB, then the semantics of the First Split Fit algorithm will be to divide that 5KB block into two segments: $5KB \rightarrow 7KB$ and $7KB \rightarrow 10KB$. Now, the first segment will be returned to the program that requested for memory (and marked as occupied) and other one shall still remain free. Thus you can see that we are creating holes in our free memory (thus fragmenting it) during this **Split** step. Thus the First Split Fit algorithm is just having this Split step on top of the First Fit algorithm before returning the block to the requesting program (and the same for Best Split Fit algorithm).

Note that the above semantics of Splitting are to be followed for both the algorithms.

Now, if we go back to the definition of $0 \to 1KB$, we can see that it means all the addresses in the range $[0, 1024)$. Similarly, $1KB \to 2KB$ will be memory addresses $[1024, 1024 + 1024)$. Thus we can see that this range can be specified by two integers: **Starting address** and **Size of the block**. This will the way of defining a memory block in the subsequent questions.

The main functionality of the Memory Allocator will be *To allocate free memory* and *To free allocated memory.* In order to solve the issue of Fragmentation, the system will also *Defragment the free memory.* It basically searches for consecutive free blocks and merges them into one bigger block. To illustrate this entire process better, let us go through the following example (Allocation is via First Fit Split):

`Initially, Free Memory Blocks:[0 → 100MB] and Allocated Memory Blocks:[]`

(1) `After` *`Allocate(5KB)`*`, FMB:[5KB → 100MB] and AMB:[0KB → 5KB]`
(2) `After` *`Allocate(10KB)`*`, FMB:[15KB → 100MB] and AMB:[0KB → 5KB, 5KB → 15KB]`
(3) `After` *`Allocate(15KB)`*`,FMB:[30KB→100MB] and AMB:[0KB→5KB,5KB→15KB,15KB→30KB]`
(4) `After` *`Free(5K)`*`, FMB:[30KB→100MB, 5KB→15KB] and AMB:[0KB→5KB, 15KB→30KB]`
(5) `After` *`Free(15K)`*`, FMB:[30KB→100MB, 5KB→15KB, 15KB→30KB] and AMB:[0KB→5KB]`
(6) `After` *`Defragment()`*`, FMB:[30KB→100MB, 5KB→30KB] and AMB:[0KB→5KB]`
(7) `After` *`Allocate(20KB)`*`,FMB:[50KB→100MB,5KB→30KB] and AMB:[0KB→5KB,30KB→50KB]`

In the above illustration, the data structure used for the Free Memory Blocks (FMB) and the Allocated Memory Blocks (AMB) is a list. So initially, the FMB list will be having one block corresponding to the memory from $0 \to 100MB$, while the AMB list will be empty. Then in **(1)** when one program requests for 5KB of memory, the First Fit algorithm will return the only memory block in FMB and the Split part for the First Split Fit will divide it into two blocks: $0 \to 5KB$ (to be given to the program) and $5KB \to 100MB$ (which will be a block in the FMB list). The 5kB size block will be added to the AMB list. The same thing will follow for **(2)** and **(3)**.

Now, in **(4)**, the program which held the memory block which had it's starting address at 5K (i.e 5×1024) tells the Allocation system that it no longer requires this block and thus frees it. Note that here, we don't use 5KB (it is `Free(5K)`) because using 5KB would technically mean that it is the size of the block. (Remember that the size of one memory address is taken as 1B here). So the Allocation system will search for a Memory block which has it's starting address at 5K, finds the block of $5KB \to 15KB$, and removes it from the AMB list and adds it at the end of the FMB list. Think what will happen if the program had requested `Free(3K)`? The same argument will follow for **(5)**. Remember that the notation $5KB \to 15KB$ is just used for easier illustration.

In **(6)**, when the `Defragment()` is called, the it will go through the FMB list, detect the free blocks of $5KB \to 15KB$ and $15KB \to 30KB$ which are adjacent and then merges them into a singe larger free block of $5KB \to 30KB$. Finally in **(7)**, the System will return the first block from FMB list which is larger than 20KB size (Since it is First Split Fit). That block will be $50KB \to 100MB$ here and then it will Split this block as described earlier and the same arguement will follow as discussed in **(1)**. Think of what changes will happen if we used Best Split Fit algorithm rather? Try thinking of how would you implement this FMB and AMB list along with these three functions? What if we used Binary Trees rather than lists? Will a BST be better?

We are going to use an abstract data type called `Dictionary` to store the free memory blocks and allocated memory blocks. The definition of this data structure is given in the file `Dictionary.java`. The `Dictionary` data structure has the following six operations defined: `Insert, Delete, Find, getFirst, getNext, sanity`. The exact semantics of all these operations are given in the file `Dictionary.java`.

Assignment-1 (100 marks)

In this assignment, we make use of Doubly Linked Lists (remember the Flipped Class Homework?) in order to implement the `Dictionary` data structure. To make your work easier, we have written a few stub java files, with the abstract classes `Dictionary, List, A1List`. The precise semantics of the six dictionary functions `Insert, Delete, Find, getFirst, getNext, sanity` is given in the file `List.java`. Read the specifications given as comments in these files carefully and implement the six dictionary functions according to the given specifications using a doubly linked list data structure.

A1.1 (40 marks): Implement the following six functions `Insert, Delete, Find, getFirst, getNext, sanity` of the class `Dictionary` in the file `A1List.java` using doubly linked lists. Thoroughly check and test your implementation. Your program should work correctly on all the inputs, not only on the test cases provided. You are expected to handle all the corner cases, as your program will be evaluated over a large set of test cases (including many more new test cases not given you earlier). In order to help you write program that work correctly on all the inputs, think of different invariants that your data structures should satisfy and code them in the function `sanity`. For debugging purposes, you might want to call this function before and after every operation.

The `sanity` function checks for the sanity of the list. You are supposed to think of a well formed meaning for this *Sanity of the list* and check. For example:

- If this list gets circular then it must fail the `sanity` test (or return False).
- If the `prev` of the head node or `next` of the tail node is not null, then the sanity test should fail.
- For any `node`, which is not head or tail if `node.next.prev != node`, then sanity test should fail.
- Can you think of other invariants for your doubly linked list, which when violated, should lead to the failure of `sanity`?

You are not supposed to use any extra space for the `sanity` function. Just use O(1) extra space, otherwise the submission will be penalised. **You are not supposed to make any changes in the stub class provided (`Dictionary.java, List.java`).**

Having completed the Doubly Linked List definition, we will now use this data structure in order to program our Dynamic Memory Allocation System. For this, we provide a class `DynamicMem` and its associated java stub class file `DynamicMem.java`, which consists of the abstract class definitions and the specifications of three functions `Allocate, Free, Defragment` that you are supposed to implement. The Public Memory Array `Memory`, Free Memory Blocks (FMB) `freeBlk` and Allocated Memory Blocks (AMB) `allocBlk` are members of this class.

A1.2 (40 marks): Using your `A1List` implementation, implement the functions `Allocate, Free` in the file `A1DynamicMem.java`. Thoroughly test your implementation. **You are not supposed to make any changes in the file `DynamicMem.java`.** The specifications of these functions have been provided in the file `DynamicMem.java` as comments. Your program should work correctly on all the inputs, not only on the test cases provided. You are expected to handle all the corner cases, as your program will be evaluated over a large set of test cases (including many more new test cases not given you earlier).

A1.3 (20 marks): A1.3.1 Assume that n operations have been performed on your DLL. What is the worst case time complexity of each of the following operations (`Insert, Delete, Find, getFirst, getNext, sanity`) that you implemented in `A1List`,
A1.3.2 Assume that n operations (`Allocate/Free`) operations have been performed on `A1DynamicMem`, what is the worst case time complexity of `Allocate, Free` operations. Submit your A1.3 in Gradescope.

Assignment-2 (100 marks)

Now we will try to make your dictionary implementation more efficient so that your dynamic memory allocator can run faster. You are provided with the abstract class definitons of the class `Tree` and the specifications of the six dictionary functions `Insert, Delete, Find, getFirst, getNext, sanity` in the file `Tree.java`. You are supposed to implement the six functions of the dictionary using a binary search tree in the file `BSTree.java`.

A2.1 (50 marks): Implement the functions `Insert, Delete, Find, getFirst, getNext, sanity` of the abstract class `Dictionary --> Tree` using binary search trees in `BSTree.java`. Thoroughly test your implementations. Your program should work correctly on all the inputs, not only on the test cases provided. You are expected to handle all the corner cases, as your program will be evaluated over a large set of test cases (including many more new test cases not given you earlier). In order to help you write program that work correctly on all the inputs, think of different invariants that your data structures should satisfy and code them in the function `sanity`. For debugging purposes, you might want to call this function before and after every operation.

Your implementation should work seamlessly with `A1DynamicMem` while using dict_type = 2 (`BSTree`). Test and check that your memory allocator of A1 still works with `BSTree` implementation instead of `A1List` implementation of dictionaries.

A2.2 (50 marks): Implement the `Defragment` function in `A2DynamicMem` using your `BSTree` implementation. This function should merge all the contiguous free memory blocks into one single free memory block thereby defragmenting the memory.

Assignment-3 (100 marks)

For this assignment we will try to further optimize your memory allocation system's performance using the concept of AVL trees learned in the lectures. As taught in the flipped classes, AVL trees are Balanced Binary Search Trees that have the heights of a logarithmic order.

The class `AVLTree` is derived from the class `BSTree` and adds a height variable. The file `AVLTree.java` contains the definitions of this class. For this assignment, you need to re-implement some of the functions of dictionary to make them time efficient. You will be evaluated on the time complexity of your algorithm as well as its correctness.

A3.1 (80 marks): To make your memory allocator efficient, implement the required functions of AVLTree.java and make your binary search trees balanced. You don't need to re-implement all the functions. Only implement the functions that are needed to speed up the implementation (`Insert, Delete, Find, getFirst, getNext, sanity`).

Thoroughly test your Dynamic Memory allocator submitted in A2 (`A2DynamicMem`) using AVL trees (`dict_type = 3 (AVLTrees)` for `Allocate, Free` as well as `Defragment` functions. For this assignment, you only need to submit `AVLTree.java`. No other changes are allowed.

Thoroughly test your implementations. Your program should work correctly on all the inputs, not only on the test cases provided. You are expected to handle all the corner cases, as your program will be evaluated over a large set of test cases (including many more new test cases not given you earlier). In order to help you write program that work correctly on all the inputs, think of different invariants that your data structures should satisfy and code them in the function `sanity`. For debugging purposes, you might want to call this function before and after every operation.

A3.2 (10 marks): Assume that n-operations have been performed on your dynamic memory allocator (using BST). What is the worst case time complexity of each of the operations (`Allocate, Free, Defragment`) of your memory allocator? Justify your answer with suitable reasoning. Submit in Gradescope.

A3.3 (10 marks): Answer the above questions if your memory allocator was using the AVL Trees in place of BST. Justify your answer with suitable reasoning. Submit in Gradescope.

Submission of the Assignments

Deadline

- Assignment1 - 18 Nov 2020 9:00PM
- Assignment2 - 24 Nov 2020 9:00PM
- Assignment3 - 30 Nov 2020 9:00PM

Instructions

- You have been provided with code stubs for all the Assignments. These stubs contain the *Class Definitions* and *Method declarations* and *Specifications* of what each method is supposed to do.
- You have to implement your programs in separate classes as per the instructions provided in each questions. Make sure you **do not modify** the base classes provided to you.
- You are supposed to take care of all the exceptions and corner cases involved.
- All the Assignments are supposed to be done **individually**.
- Submission instructions will be given separately later.
- There is no late policy for the submissions. If you are late then you won't get any marks.
- Any copying or use of unfair means will result in **strict penalties**.
- These assignments will take time. You will need to learn many new concepts of Java and Object Oriented programming not covered in the class. It is strongly recommended that you start coding as soon as possible and not wait for the weekends before the deadline.