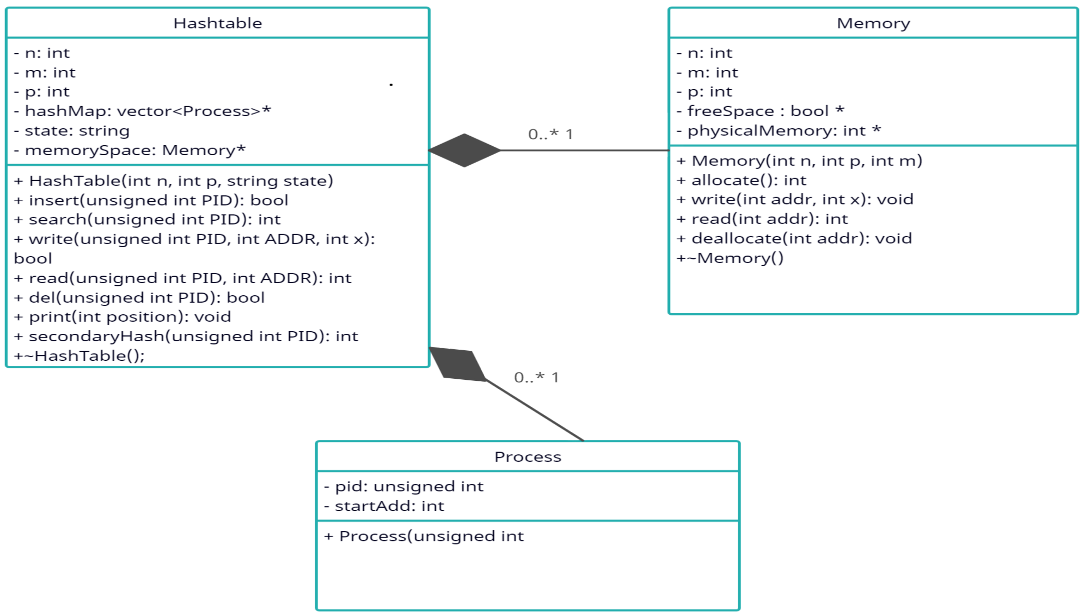
# Jefferson Fu Project 2 Design Document

## UML Diagram:


Hashtable
- n: int
- m: int
- p: int
- hashMap: vector<Process>*
- state: string
- memorySpace: Memory*
+ HashTable(int n, int p, string state)
+ insert(unsigned int PID): bool
+ search(unsigned int PID): int
+ write(unsigned int PID, int ADDR, int x): bool
+ read(unsigned int PID, int ADDR): int
+ del(unsigned int PID): bool
+ print(int position): void
+ secondaryHash(unsigned int PID): int
+~HashTable();
Memory
- n: int
- m: int
- p: int
- freeSpace : bool *
- physicalMemory: int *
+ Memory(int n, int p, int m)
+ allocate(): int
+ write(int addr, int x): void
+ read(int addr): int
+ deallocate(int addr): void
+~Memory()
Process
- pid: unsigned int
- startAdd: int
+ Process(unsigned int
0..* 1
0..* 1

## HashTable:

My HashTable class incorporates functions for inserting, searching, writing, reading, deleting, and printing and within every function, it splits into 2 different cases based on which type of hashing I am using. It's worth noting that the return types for each function corresponds to what output is needed, so the function is boolean if the required output is only "success" or "failure" and integers if the output requires a position or value to be printed. The output is then formatted in the driver file. Within the class, it contains the size of the memory space, page size, number of pages, a string "state" to denote open or ordered hashing, a pointer to a memory object, and a pointer to a 2D vector for the hashmap which is all initialized within my constructor. I used a 2D vector to cover the main keys for the hashmap and then the second dimension is for chaining when collisions occur for ordered hashing. This also allows me to use the same data structure for open hashing too as the logic in the program makes it so at every index of the first dimension, the second dimension's max length is 1 element. I also have another small function for a secondary hash that takes in a PID and runs the secondary hash to get an offset for open cases. Lastly, my destructor deletes the pointers for the memory object and the 2D Vector.

Ordered: For ordered inserting, I first make a check to see if the PID already exists within the program using my search function. The next check will be to try and allocate a page if there's space in the physical space in the memory. If those checks fail, I will return false indicating to the driver that I failed to insert the PID. Otherwise, I create a process object containing the PID and its page's starting address, iterate through the vector at the hashmap's hash value index to insert the new process at the correct location ensuring the PID's are in descending order.

For ordered searching, I first take a PID that I want to search for as a parameter, use the primary hash function to get the hash value index, and then just iterate through the secondary array at the hashmap's hash value index. If any of the processes's PID value is the one I am searching for, I return the hash value index.

For ordered write and read, they are similar functions except the only difference is that I am returning an integer for read, while write I change the value at the location in memory. Within these functions, I first search to see if the PID's exist and if the address offset that is given as a parameter will be greater than the page size. If these checks fail, I can return a failure. Otherwise, I iterate through the hashmap's hash value index to get the desired process's starting address and then call the memory class's write or read function based on the function.

For ordered deletion, I use our search function to see if a process with that PID already exists and if it does, I return false to indicate failure. I iterate the array at the hashmap's hash value index to search for the desired PID given in the parameter and then I will remove that process while noting its start address. I then call our memory object's deallocate function with the starting address as a parameter to free up the page in physical memory.

For printing, this is the only function that is declared as a void function because I handle all the outputs within this function due to its unique outputs. I first check if it's empty and return the failure message if it is. Then I iterate through the array at the hashmap's desired position that I took as a parameter and just print each of the process's PIDs since it's already sorted.

Open: For open insertion, I first make the same checks as the ordered insertion to make sure I can proceed. Then I get our offset, which is the secondary hash on the PID, to repeatedly add to our initial hash value until I find an open spot in our hashmap and then the process is added to that index.

For open searching, I make an initial check to see if the desired PID is at the primary hash value index and return the index if it is. If not, I will repeatedly add an offset until I iterate to where I started again indicating it doesn't exist, or until I get the right PID and return the index in which I found it.

For open write and read, I make our initial checks like I did in the ordered functions. Since the search gives the direct index where the PID lies in the hashmap, I can directly retrieve the process I want and take the starting address from it and use it for parameters for the following calls. From there I call our memory object's write or read to end the function.

For open deletion, it's the same as ordered but my search function will give me the exact location of the PID I want, thus, making retrieving the start address a little quicker.

**Memory:**

My memory class incorporates functions for allocation, writing, reading, and deallocation. The constructor is called when the hashmap's constructor is called and it contains the same n, p and m values as the hashMap. It also contains an integer array for the data contained in the actual physical memory and a boolean array to see which pages are free with size of m. The destructor for this class simply deletes the space for the two dynamic arrays.

For allocate, I iterate through the boolean free space boolean array to see if any of the values are false meaning they are free, if they are false, I switch them to true and return the iteration index multiplied by the page size to return the correct starting address.

For write, I take the address as an index for my physical memory array and write to that location the value to put in that location given in the parameter. The read function is the same except I change the value at that location but just return it back to the hashmap, For deallocate, I take a given starting address, divide it by p to get the correct free space array index, and set the value at that index in my free space array to false to show that the page is free to use.

**Time Complexity:**

Assuming uniform hashing of constant time, this tells us that search is constant time. This means that search does not dominate the time complexity in any of the other functions. In terms of time complexity, the rest of the functions at most will either make a couple of assignments or iterate through the vectors at a hash value index in the hashmap. First, it is known that assignments are constant time. Second, the number of these iterations will be the same number as search which we know to be constant, therefore, each function will be running at average constant time.