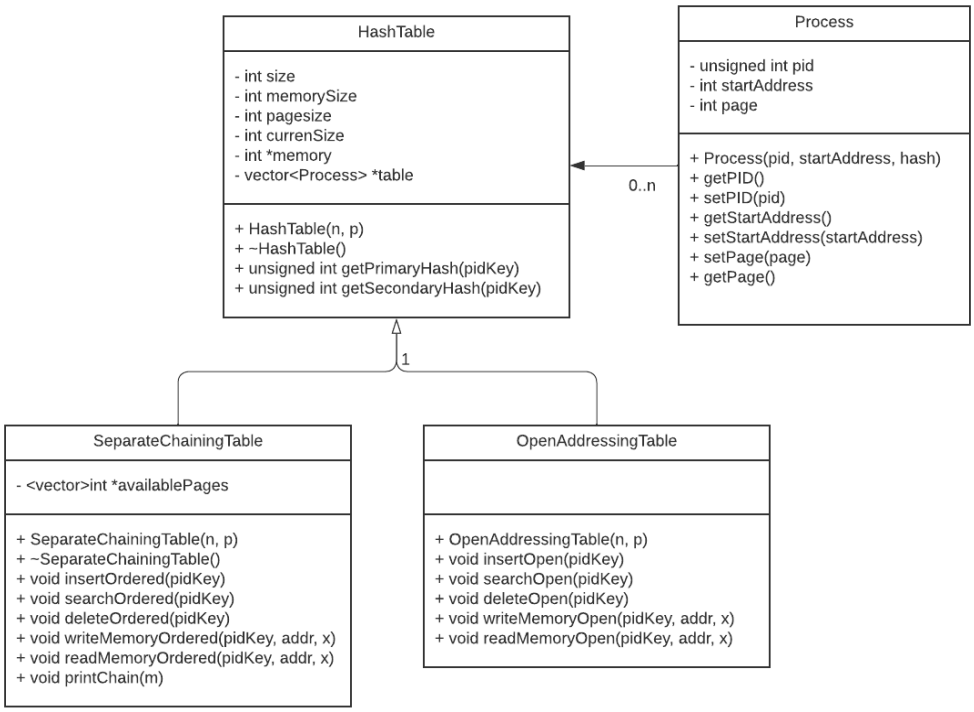
ECE 250 Project 2 Design Document

Brandon Vo

February 10, 2023

Process Class

The process class is used as an object to be stored in the vector of the HashTable class. Each process has an unsigned integer key (pid), a startAddress integer to handle the start of physical pointers to memory, and a page integer for keeping track of which process uses which page. The page integer is only used for separate chaining to handle and assign the virtual addresses of the processes. There are also getters and setter void functions used to assign and retrieve these variables. The setter function parameters take its corresponding variable type to assign to the private member variables.



HashTable Base Class

The base class for the hash table has variables for the table size (n/p), memorySize (n), pageSize (p), and currentSize. It also uses a vector of Process objects to store the keys and addresses in a variable called table. The memory is held in an integer array of size n.

This class has two useable functions used for our inherited hash tables to get hash values and handle collisions.
getPrimaryHash(unsigned int pidKey): A void function which returns pidKey % size.
getSecondaryHash(unsigned int pidKey): A void function which returns (pidKey/size) % size, and adds 1 if even.

SeparateChainingTable Inherited Class

The SeparateChainingTable class inherits the HashTable class to use commonly shared variables and functions between both hash table implementations. In my separate chaining hash table implementation, I created one private member variable for a vector of integers. This vector is used as a first-in-first-out queue to handle the available pages in the memory table. In the constructor, the available page vector is populated with n/p elements from 0 to (n/p) - 1. The destructor deletes the remaining elements in the page vector when the program ends.

insertOrdered

In the insertOrdered function, it takes in an unsigned integer for the pidKey. Initially, it determines if the table is full by checking if there are any available pages left. If there are none left, the table is full. To handle collisions, we use separate chanining by inserting the keys in each chain in an ordered and descending manner. This is done by using a while loop to stop when the iterator's PID is less than the PID key we are trying to insert. To handle the start address of the key, we first subtract the hash value from the page size if it is greater than or equal to the page size. Then, we get the front-most element from availablePages multiplied by the page size and add it to the hash value. There is an edge case where the page size is 1, meaning we do not need to use the pages to determine the virtual address. Also, if we find that the start address is an odd number, we need to subtract it by one to fit the offset rule when writing to memory. After inserting the process, we must remove the top page from the available pages vector. Assuming uniform hashing, the average time complexity of this function is O(1). This is because the hash function should minimize collisions and require only one iteration on average to insert.

searchOrdered

The search function for separate chaining uses the primary hash function to determine the bucket to search in the hash table. With the bucket, we check the elements and find a matching PID key. If found, we will print "found [key] in [position]". Otherwise, we print "not found". With uniform hashing, the average time complexity of this function is O(1), since keys will be distributed evenly throughout the table.

deleteOrdered

The delete function uses the same process as searching by using the primary hash function to search for the element in the bucket with the same PID key. Once found, we will erase the iterator in the bucket and re-add the page used by reading getPage() and pushing it back to the available pages vector. The average time complexity of this function is also O(1), as keys will be distributed evenly and we use our effective getPrimaryHash() function to find the element linearly.

writeMemoryOrdered
The function to write memory in separate chaining takes in a PID key, an address offset, and a value x to write to the memory. This finds the bucket using the getPrimaryHash() function and iterates through the bucket until we find a PID key. The memory address is defined by the iterator's start address and the offset together. If we find that the memory address is within the memory size and the address offset is smaller than the page size, we can successfully assign the memory at the physical address to value x. For the average time complexity, it is O(1) when assuming uniform hashing, as hashes are evenly distributed in the table and can usually be found directly with getPrimaryHash().

readMemoryOrdered
The read memory function takes in a pidKey and the offset value to the physical address. By using the getPrimaryHash() function, we find the bucket we need to search. We use the same method as the write memory function to determine the physical address and its validity. If it is valid, the address and value of the memory at the physical address will be printed out. The average time complexity of this function is O(1) with the assumption that there is uniform hashing since keys are spread out and the keys can be found in one iteration using the hash function.

printChain
The print chain function prints out all of the elements of the chain in a bucket by iterating through each element and checking if the PID key is not marked as a tombstone (key = 0). The time complexity of this function is O(n), where n is the size of the bucket. This is because it needs to search through the chain and print every element in the bucket.

OpenAddressingTable Inherited Class
The OpenAddressingTable class also inherits the hash table class to use commonly shared variables (size, memory, table vector) and functions for getting both primary and secondary hash values.

insertOpen
To insert PID keys into the hash table, we use double hashing to avoid collisions. This searches while the hash table at the hash index is not empty and we iterate less than the size of the hash table. If there are collisions, we check if we are trying to insert a duplicate PID key by using getPID(). We also check if we found one tombstone marker from a previously deleted key by checking if the collided PID key is 0. If so, we save the index of the tombstone and continue in case there exists a duplicate PID key after the found tombstone marker. If there are no collisions left, we insert the key at the corresponding hash value, or at the saved tombstone hash marker if we encountered one. On average, the runtime of this function is O(1) by implementing our good hash function that minimizes collisions. In the average case, the while loop does not need to run more than once or twice since we uniformly scatter the keys across the table.

searchOpen
The search function uses an iterator from the calculated hash value and iterates through the bucket of the chain until it finds that the iterator has the same PID key as being searched. If it does not find the PID key, it prints out "not found". On average, this takes O(1) time by our assumption because the hash function is uniformly distributed. We should be able to find the PID key in constant time.

deleteOpen
The delete function iterates the table at the PID key's hash value and uses double hashing for collisions. If it finds a matching PID key to the input, we use setPID() and set the key to 0 to indicate a tombstone marker. This is because we cannot delete that single PID without coincidentally deleting the entire chain. We use the PID key 0 as a marker for us to reassign at a later time if we insert another value to the hash table. The average time complexity of this function on average is O(1), since we are assuming uniform hashing and we should be able to find the key in a single iteration.

writeMemoryOpen
This function writes a value x to the memory array by determining the physical address of the memory. It does this by using double hashing if collision occurs until either the table at the calculated hash value isn't empty, the iterator PID key isnt our input PID, or until we reach the end of the table size. We then determine the memory address by adding the found PID key's start address and adding the offset addr. After checking if the physical address is within the space of the virtual address and memory array, we write the value x to the address. The average runtime of this function is O(1) assuming uniform hashing, as each slot in the hash table is equally likely to be the final destination of a key. The average number of probes is constant.

readMemoryOpen
The function to read memory in open addressing also uses double hashing to iterate until we match with a PID key of our input. We then get the memory address by adding the current start address and the address offset and checking if it is within the bounds of the memory size, and address is within the page size. If this is valid, it prints out the address and the value at the memory address location. On average, the time complexity is O(1) with our assumption, since we would only need to iterate once to find the key by using our hash function.