In order to implement the hash table, in an abstract manner, I created a table class which stores all the functionality Of a hash table, as well as all the necessary data. To allow the user to change the method of hashing (open addressing Or chaining), while still having he same functionality (abstraction), I created two children classes for each method of hashing (inheretance). From the users perspective, the hash table behaves the same way, regardless of hashing method (except for the print function). In the main file there exists a loop which prompts the user for inputs. This loop is not apart of The hash table, as there are other situations where you could use one without such infinite loop.

Class HashTable:

Contains all the shared data members between the two different methods of hashing such as the page, table and current size, and the memory to store the pages. Additionally, it contains the protected hashing function for its children to use. The constructor is meant to be called upon the creation of a child hashing type class. This class is purely virtual and is only meant to be used with a child class which contains most of the functionality and data of a hash table. Regardless Of the hashing method used, the hash table should behave in the same manner (polymorphism).

Class Chaining: public HashTable

Contains all functions to manipulate a hash table (read, write, del...) as well as the actual hash table itself. The implementation of this hash table is in the form of an array of linked lists, as well as an array of booleans In order to keep track of available pages. When a collision occurs, it simply inserts a new node along the linked list At its hash location. Upon the creation of this class, the hash table array, bool array are created, and the Constructor of the HashTable parent is called. In order to prevent memory leaks, upon the destruction of the class each linked list in the table must be deallocated. All the methods of this class are public as they are to be directly called on the table.

OpenAddressing Child Class:

Contains all the functions to manipulate a hash table, as well as the actual hash table itself. The implementation of this Hash table is in the form of an array of pointers to process objects which contain the data of the table. There is no need To keep track of available pages as the page and hash data locations have a one to one relationship. Upon the creation of An object, the hash table constructor is called. When a collision occurs, it continuously runs a double hash until it Finds an empty slot. In order to prevent memory leaks, upon destruction the hash table and its data are deleted.

Class ChainProcess: and Class OpenProcess:

These classes serve the same purpose as being the holders of the hash tables data. They contain the PID and the relative Address in memory where its page is stored. The only difference, is that for the chaining hash table, there must be a Pointer to the next node to allow for the creation of a linked list. All the data members are private in order to esure Proper use, and all of its functions are simple get and set methods. The constructors simply set the data of the class. For the chaining process, there are two different constructors to allow for overloading of what the next node should be (Nullptr or a next node). In the case in chaining, when an element must be inserted in the beginning or middle of the linked list, the constructor with an overload for the next node is called.

Runtimes:

Create:

For both methods of hashing, the creation is O(n) as you must dynamically allocate arrays of length n based on the users desired need data and number of pages.

Insert:

For both table, assuming the PIDs are uniformly distributed, the average runtime will be O(1) as the hash function runs in constant time. Upon hashing the PID, you should get the location to insert. If a collision occurs: for chaining you will have to traverse the linked list until you reach the in order location of your PID, for double hashing, you will need to continually hash the PID until you reach an empty spot. This could lead to O(n) for n list traversals or n hashes, but this shouldn't happen often if the PIDs are uniformly distributed. For double hashing, empty slots are denoted by either nullptr, if an element has never been stored, or PID = 0 if an element has previously been stored. In order to ensure unique PIDs, for double hashing we must continue to hash until we either reach a nullptr or search every element, and then insert in whatever the last open slot was. Additionally, the insert itself only takes O(1). Therefore O(1) average runtime.

Search:

Same as insert. Hash function runs in O(1), but could be O(n) if there were collisions in the insertion process, but this shouldn't happed in often. The search response is in O(1). Therefore O(1) average runtime.

Write:

Same as insert. Hash function runs in O(1), but could be O(n) if there were collisions in the insertion process, but this shouldn't happed in often. The write process is constant as we store the relative memory location in the process object. Write process is in constant time. Therefore O(1) average runtime.

Read:

Same as Write, except instead of writing to the memory location, we print its value, which takes O(1). Therefore O(1) average runtime.

Delete:

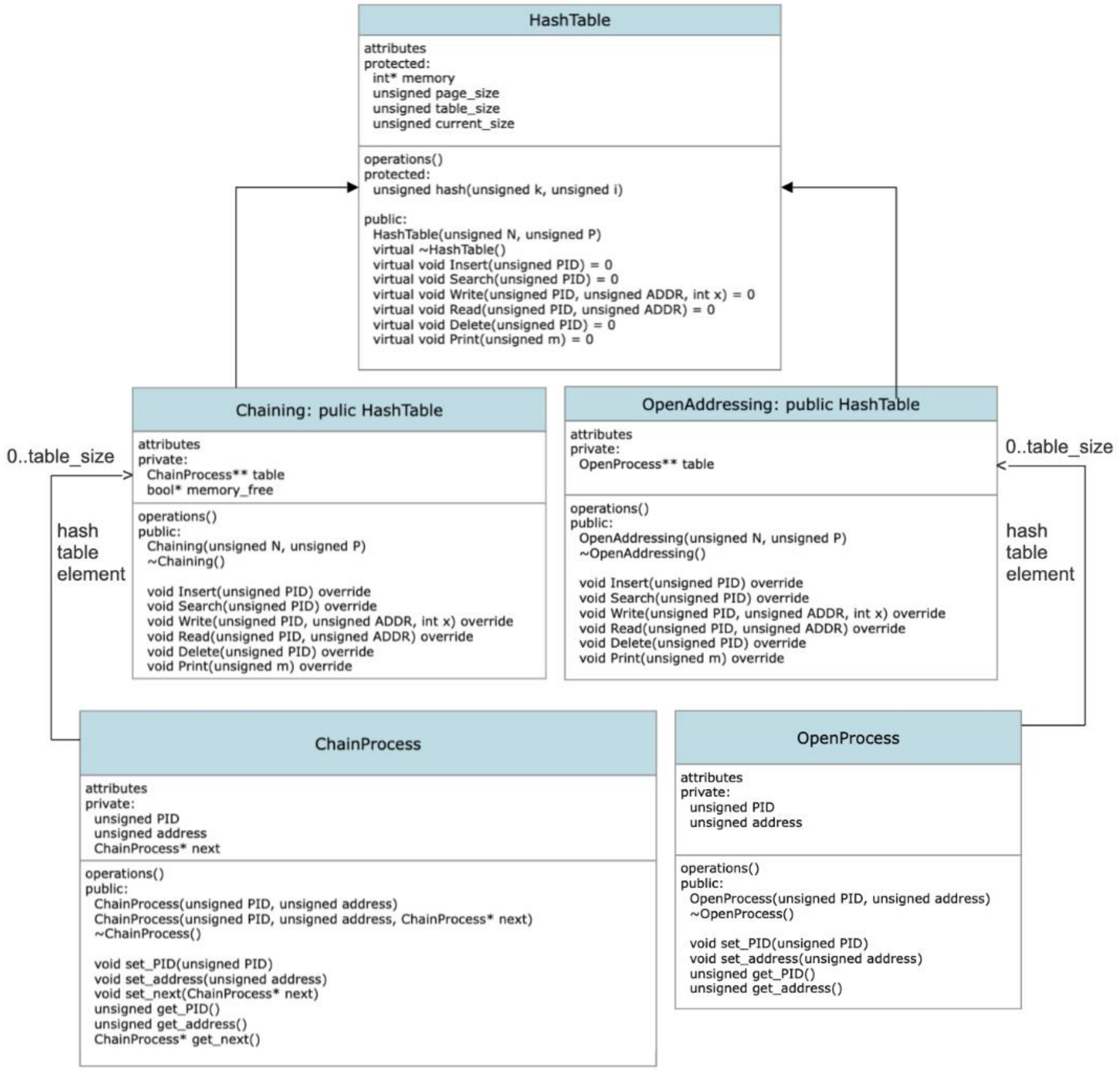
Same as Insert, except instead of inserting an element, we traverse the hash table until we find our PID, and then delete it (or set PID = 0 for open addressing). The deletion is in constant time. Therefore average runtime of O(n).

Print:

This function only runs in chaining. We first must hash in O(1), then traverse the linked list, printing all of its values. Printing takes O(1), therefore if the hash table is uniformly distributed, we should only have to do this a small constant number of times, giving O(1) time complexity. In the rare case that all the PIDs get hashed to the same slot in the table, we will need to traverse n elements giving O(n). As this case is rare, we get an average runtime of O(1).

End:

Upon ending the program the class is destroyed and the destructors are called. During this process all the dynamically allocated elements are deleted, as there are n elements and deletion is constant, we get O(n) time.



```
HashTable
attributes
protected:
  int* memory
  unsigned page_size
  unsigned table_size
  unsigned current_size

operations()
protected:
  unsigned hash(unsigned k, unsigned i)

public:
  HashTable(unsigned N, unsigned P)
  virtual ~HashTable()
  virtual void Insert(unsigned PID) = 0
  virtual void Search(unsigned PID) = 0
  virtual void Write(unsigned PID, unsigned ADDR, int x) = 0
  virtual void Read(unsigned PID, unsigned ADDR) = 0
  virtual void Delete(unsigned PID) = 0
  virtual void Print(unsigned m) = 0
```

```
Chaining: pulic HashTable
attributes
private:
  ChainProcess** table
  bool* memory_free

operations()
public:
  Chaining(unsigned N, unsigned P)
  ~Chaining()

  void Insert(unsigned PID) override
  void Search(unsigned PID) override
  void Write(unsigned PID, unsigned ADDR, int x) override
  void Read(unsigned PID, unsigned ADDR) override
  void Delete(unsigned PID) override
  void Print(unsigned m) override
```

```
OpenAddressing: public HashTable
attributes
private:
  OpenProcess** table

operations()
public:
  OpenAddressing(unsigned N, unsigned P)
  ~OpenAddressing()

  void Insert(unsigned PID) override
  void Search(unsigned PID) override
  void Write(unsigned PID, unsigned ADDR, int x) override
  void Read(unsigned PID, unsigned ADDR) override
  void Delete(unsigned PID) override
  void Print(unsigned m) override
```

```
ChainProcess
attributes
private:
  unsigned PID
  unsigned address
  ChainProcess* next

operations()
public:
  ChainProcess(unsigned PID, unsigned address)
  ChainProcess(unsigned PID, unsigned address, ChainProcess* next)
  ~ChainProcess()

  void set_PID(unsigned PID)
  void set_address(unsigned address)
  void set_next(ChainProcess* next)
  unsigned get_PID()
  unsigned get_address()
  ChainProcess* get_next()
```

```
OpenProcess
attributes
private:
  unsigned PID
  unsigned address

operations()
public:
  OpenProcess(unsigned PID, unsigned address)
  ~OpenProcess()

  void set_PID(unsigned PID)
  void set_address(unsigned address)
  unsigned get_PID()
  unsigned get_address()
```