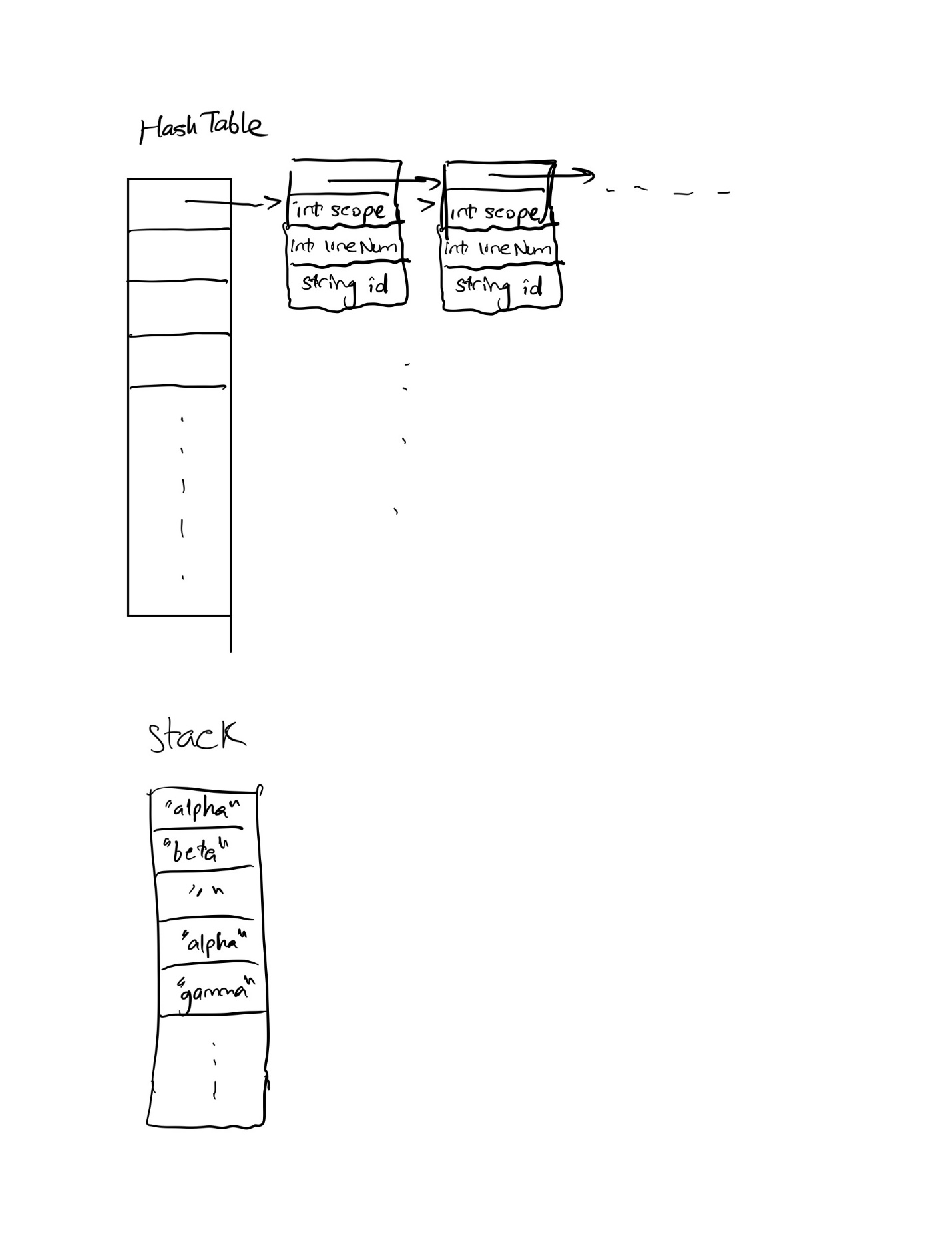
* a description of your algorithms and data structures (good diagrams may help reduce the amount you have to write), and why you made the choices you did. You can assume we know all the data structures and algorithms discussed in class and their names.



class variable

{

public:

variable();

variable(const variable& copy);

variable(string ID, int LINE, int SCOPE);

int line;

int scope;

string id;

};

class HashTable

{

public:

HashTable();

HashTable(int c);

void insertItem(const variable& var);

int findItemReturnLine(const string& id) const;

int findItemReturnScope(const string& id) const;

bool canIDeclare(const string& id, const int& scope) const;

void deleteItem(const std::string& id, const int& scope);

int hashFunction(int key) const;

int hashFunction(int key, int cap) const;

void displayHash();

private:

int capacity;

list<variable>\* table;

};

To solve the problem given in the project 4 spec I used 2 data structures:

A stack of strings that keeps track of scopes and the declared ids

A hash table which, given a string, gives a list of class variable (open hashing)

I defined the variable class in order to keep track of the lineNumber of an id declaration and the scope it was declared in. I used this in the hash table.

I used the hash table class above to find the last instance of the id that was declared and to check if an id can be declared I the current scope. I used the std::hash function to hash strings and then bounded it according to the hash table size.

I used a hash table because look up time is O(1) whereas other data structure’s like trees have a look up time of O(log(n)) or data structures like list have a look up time of O(n).

class NameTableImpl

{

public:

void enterScope();

bool exitScope();

bool declare(const string& id, int lineNum);

int find(const string& id) const;

private:

HashTable m\_find;

stack<string> m\_dec;

int m\_scope = 0;

};

I used m\_scope to keep track of the scope the program was currently in. I used the m\_dec stack of strings similar to the way m\_ids was used in the inefficient program. I push all ids that are declared into m\_dec. I also push the empty string into it whenever we enter a new scope. Whenever we exit a scope, I pop everything off the stack until I reach an empty string (the last scope entry). I then pop that empty string of the stack.

I used a stack because it was a relatively simple data structure and because removal time is O(1).

* a note about the time complexity of the NameTable functions. For example, in our inefficient implementation, enterScope is constant time, exitScope is linear in the number of identifiers going out of scope, declare is linear in the number of identifiers in the current scope (because of the check for duplicates), and find is linear in the number of identifiers currently accessible.

void NameTableImpl::enterScope() **==> O(1)**

{

m\_dec.push(""); **==> O(1)**

m\_scope++; **==> O(1)**

}

bool NameTableImpl::exitScope() **==> O(n) where in is the number of elements in the scope that we are exiting**

{

if (m\_scope != 0)

{

while (m\_dec.top() != "") **==> O(n) where in is the number of elements in the scope that we are exiting**

{

m\_find.deleteItem(m\_dec.top(), m\_scope); **==> O(1) on avg. (O(n) for worst case where n is the number of collisions for the key produced by m\_dec.top())**

m\_dec.pop(); **==> O(1)**

}

m\_dec.pop(); **==> O(1)**

m\_scope--; **==> O(1)**

return(true);

}

else

{

return(false);

}

}

bool NameTableImpl::declare(const string& id, int lineNum) **==> O(1)**

{

if (m\_scope == m\_find.findItemReturnScope(id)) **==> O(1)**

{

return(false);

}

else

{

m\_find.insertItem(variable(id, lineNum, m\_scope)); **==> O(1) on avg. (O(n) for worst case where n is the number of collisions for the key produced by id)**

m\_dec.push(id); **==> O(1)**

return(true);

}

}

int NameTableImpl::find(const string& id) const **==> O(1)**

{

return(m\_find.findItemReturnLine(id)); **==> O(1) on avg. (O(n) for worst case where n is the number of collisions for the key produced by id)**

}

* [pseudocode](http://web.cs.ucla.edu/classes/spring22/cs32/pseudocode.html) for non-trivial algorithms

class HashTable

{

public:

HashTable();

HashTable(int c);

void insertItem(const variable& var);

int findItemReturnLine(const string& id) const;

int findItemReturnScope(const string& id) const;

bool canIDeclare(const string& id, const int& scope) const;

void deleteItem(const std::string& id, const int& scope);

int hashFunction(int key) const;

int hashFunction(int key, int cap) const;

void displayHash();

private:

int capacity;

list<variable>\* table;

};

int HashTable::hashFunction(int key) const

{

Return the magnitude of key modulo capacity

}

int HashTable::hashFunction(int key, int cap) const

{

Return the magnitude of key modulo cap

}

HashTable::HashTable(int c)

{

store the closest prime number to c using getPrime(c) in capacity

assign to table a dynamically allocated list of class variable of size capacity.

}

int HashTable::findItemReturnLine(const string& id) const

{

Use the std::hash<string> function with id to get a key.

While iterating through the linked list in table[key] if it’s the current element’s id is equal to the id we are trying to find, return that variable’s line

After you exit the while loop return -1

}

int HashTable::findItemReturnScope(const string& id) const

{

Use the std::hash<string> function with id to get a key.

While iterating through the linked list in table[key] if it’s the current element’s id is equal to the id we are trying to find, return that variable’s scope

After you exit the while loop return -1

}

bool HashTable::canIDeclare(const string& id, const int& scope) const

{

Use the std::hash<string> function with id to get a key.

While iterating through the linked list in table[key]

if it’s the current element’s id is equal to the id we are trying to find

if this element’s scope is equal or less than scope

return true

else

return false

after you have exited the while loop return false

}

void HashTable::insertItem(const variable& var) // insert/update item

{

Use the std::hash<string> function with var.id to get a key.

Add var to the front of the list at table[key]

}

void HashTable::deleteItem(const std::string& id, const int& scope)

{

Use the std::hash<string> function with var.id to get a key.

While iterating through the linked list in table[key]

if it’s the current element’s id and scope is equal to the id and scope we are trying to find

erase that element from the list and return

}