CS-300 Module 6 Pseudocode

Project One

Alex Mehr

06/14/2024

Pseudocode:

Vector Pseudocode

Add to the end of the linked list (Append):

* Create new node and add an input
* If the list is empty, new node is assigned to head and tail node.
* Else, tail will point to the new node and updates the tail with the new node.

Insert in the beginning of the linked list (Prepend):

* Create a new node and add input to it.
* If the list is empty, new node is assigned to head and tail.
* Else, tail will point to the new node and updates the tail with the new node.

Create a Print list Function:

* Create a node starting at the head.
* WHILE, the current node is not null.
* PRINT the current bid id, title, amount , fund.
* Set the current variable equal to next node.

Create a remove function:

* IF the selected node is the head node, remove the head node.
* WHILE, the element exists in the list and it is not equal to the key

Check if the element is the key

If it is the key, assign a temp value

If it is not the key, check for the next value

Delete the temp node

Search the linked list:

* Assign a new node to the head
* WHILE, the element exists in the list

IF the element is equals the search key

Return the element.

// Define Node structure to hold Bid data and a pointer to the next node

struct Node {

Bid bidData; // Assuming Bid is a structure or class containing bidId, title, amount, fund, etc.

Node\* next; // Pointer to the next node in case of collisions

};

// Define HashTable class

class HashTable {

private:

static const int tableSize = 100; // Fixed size of the hash table

Node\* buckets[tableSize]; // Array of pointers to Node (buckets)

public:

// Constructor to initialize hash table

HashTable() {

for each bucket in buckets {

bucket = nullptr; // Initialize each bucket to nullptr

}

}

// Method to compute hash value based on bidId

int hash(string bidId) {

// Simple hash function example: sum of ASCII values of characters in bidId

int hashValue = 0;

for each character c in bidId {

hashValue += ASCIIValue(c);

}

return hashValue % tableSize; // Modulo operation to fit within tableSize

}

// Method to insert a bid into the hash table

**Insert(Bid bid)** {

int hashValue = hash(bid.bidId); // Calculate hash value using bidId

// Create a new node with the bid data

Node\* newNode = new Node;

newNode->bidData = bid;

newNode->next = nullptr;

// Check if the bucket (hashValue) is empty

IF (buckets[hashValue] is equal to nullptr) {

buckets[hashValue] is equal to newNode; // Insert at the beginning of the bucket

} else {

// Handle collision by chaining

Node\* current is equal to buckets[hashValue];

while (current->next is not equal to nullptr) {

current = current->next;

}

current->next = newNode; // Append newNode at the end of the chain

}

}

// Method to print all bids stored in the hash table

**PrintAll()** {

for each bucket in buckets {

Node\* current = bucket;

WHILE current is not equal nullptr {

// Output bid details (assuming bidData has bidId, title, amount, fund, etc.)

PRINT (bidData) bidId, title, amount, fund

current = next;

}

}

}

// Method to search for a bid by bidId and return its Node pointer

Node\* Search(string bidId) {

hashValue equals hash(bidId);

Calculate hash value using bidId

// Traverse the chain at the calculated hashValue

Node\* current = buckets[hashValue];

WHILE current is not equal to nullptr

And current bidId is not equal to bidId {

current = current->next;

}

// Return the node if found, otherwise return nullptr

return current;

}

// Method to remove a bid from the hash table by bidId

Remove (bidId) {

hashValue equals hash(bidId);

Calculate hash value using bidId

Node\* current equals to buckets[hashValue];

Node\* prev is equal to nullptr;

Search for the bidId in the chain at hashValue

WHILE (current is not equal to nullptr and current bidId is not equal bidId) {

prev = current;

current = current->next;

}

IF current is equal to nullptr{}

Bid not found

return;

// Remove the node

IF prev is equal to nullptr){

IF current node is the first node in the chain

buckets[hashValue] = current->next;

}

ELSE{

IF current node is in the middle or end of the chain

prev->next = current->next;

}

delete current; // Free memory

}

};

**Load** text parsing libraries and headers

**Define** a struct to hold course data

**struct Course {}**

*courseID*

*courseName*

*preCount*

*prelist*

Course() (constructor) {courseID = courseName = ””; preCount = 0; preList = “”}

**Class BinaryTree{}**

-struct *Node*

*Course*

*right* pointer

*left* pointer

-*root*

*+printCourse()*

+*BinaryTree()*

**Main()**

**Create** new BinaryTree named *courseTree* of the struct-type **Course**

**Get** CSV file path from user

**If** no data passed use default location

**Call** **txtParser**() passing CSV file path

**Call** **validateList**() passing *courseTree*

**Get** user value to search for and **Store** in *userSearch*

**Call** **printCourse()** passing *userSearch*

**End**

**txtParser (String)**

**Open** file found at the path in *String* by invoking parser libraries

**Loop** row by row until end of file (eof)

**If** first and second string are present

**Add** the first String to struct at *courseID*

**Add** the second String to Struct at *courseName*

**Loop** until file handler has no value in a column (indicates no more prerequisite)

**Increment** a variable named *preCount* for each prerequisite found

**Concatenate** a localString named *preNames* for each prerequisite

**Add** *preCount* to struct at *preCount*

**Add** *preNames* to struct at *preList*

**Return** *tempList*

**End**

**searchList(String)**

**Create** *tempCourse* of type **Node**

**Set** *tempCourse* to the bucket at the hash location of *String*

**Loop** through list For Each Course

**If** *String* is the same as *courseID*

**Set** *tempCourse* to Course

**Return** *tempCourse*

**End**

**printCourse(String)**

**Create** *tempCourse* of type **bucket**

**Set** *tempCourse* equal to **root**

**Loop** until *tempCourse* is Null

**If** the Node at tempCourse contains a *bidId* equal than to *String*

**Output** *courseID* in Course struct found within *tempCourse* to console

**Output** *courseName* in Course struct found within *tempCourse* to console

**Loop** 0 to *preCount*

**For each** *Course* in *preList*

**Call** p**rintCourse**() passing *preList*

**If** the Node at *tempCourse* contains a *courseID* less than to *String*

**Set** *tempCourse* equal to the left Node

**If** the Node at *tempCourse* contains a *courseID* greater than to *String*

**Set** *tempCourse* equal to the right Node

**End**

**validateList**()

**Create** *tempCourse* of type **Node**

**Create** variable *valid* and **Set** to True

**For Each** **Course**

**If** *valid* is False break

**While** *tempCourse* next is not Null

**Loop** 0 to *preCount*

**Set** *tempCourse* equal to **searchList(***preList* token**)**

**If** *tempCourse* courseID is empty Set *valid* to False

**Return** valid

**End**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Vector | HashTable | Binary Tree |
| Load Data | O(1) | O(1) or O(N)  Depending if collision exists | O(log N) |
| Search | O(N) | O(1) – O(N)  Depending if collision exists | O(log N) – O(N) |
| Print/Sort | O(N logN) | O(N)  With assumption of in order table | O(N) |

Each of the 3 algorithms could be advantages at some stage of the data structure. The

advantage of the vector is the easy implementation and quick file reading and object access and

add and the disadvantage of the vector is memory consumption due to being an object.

The advantage of hash table is accessing the items directly and the ability of adding and

removing items in current time. Hash tables could be advantages to speed up the process if it is

implemented properly. Just like vectors, hash tables also consume large amount of memory and

also hash tables do not retrieve items in order .

Binary search has its own advantages and unlike the hash tables, the binary search retrieve data

in order and it accesses objects much faster than the other data structures.

My recommendation for a data structure is binary search tree as it provides accessing the course

list in alphabetical order and the tree traversal sorts out the courses. Binary search tree uses

O(log N) to search nodes in the tree and in the case of course list it is sufficient enough to run the

program properly.