**SNHU**

**CS-300 Analysis and Design**

**6-2 Project One**

**Professor: Michael Rissover**

**Student: Michael Wood**

**10/12/2023**

**Vector Pseudocode:**

* Start Program
* Execute Print User Menu
* 1. Load Bids
* 2. Display All Bida
* 3. Selection Sort All Bids
* 4. Quick Sort All Bids
* 9. Exit
* While input is not 9 or equal to, continue
* Get and store User input from menu

Option 1 (Load Bids)

* Set variable ticks equal to clock () method. Store starting clock tick.
* Initialize .csv file

1. Parse .csv file and store data within bid object
2. Define vector to store all bids
3. Return Bids

* Print size of bids
* Calculate time elapsed = Clock () – ticks
* Output time

Option 2 (Display All Bids)

* Loop Bids Vector
* Output Bids
* Insert Break

Option 3 (Selection Sort All Bids)

* Set variable ticks equal to clock () method. Store starting clock tick.
* Use the selection sorting method and supply the vector bids as an input.
* Calculate time elapsed = Clock () – ticks
* Output elapsed time
* Insert Break

Option 4 (Quick Sort All Bids)

* Set variable ticks equal to clock () method. Store starting clock tick.
* Use the quick sort method and supply the vector bids as an input, starting and ending (bid size -1)
* Calculate time elapsed = Clock () – ticks
* Output elapsed time
* Insert Break
* Else
* Output “Goodbye”
* Return 0
* End Program

**Linked Lists Pseudocode:**

* LinkedList method: Initialize housekeeping variables
* SET head equal to null
* SET tail equal to null
* virtual ~LinkedList method:
* CREATE new node starting at head
* CREATE temp node
* WHILE current node is not null
* SET temp node to current
* SET current node to next node
* DELETE temp node
* Append method: Append a new bid to the end of the list
* CREATE new node
* IF head is null
* SET head and tail equal to new node
* ELSE
* SET tail equal to new node
* INCREASE sizePrepend method: Prepend a new bid to the start of the list
* Create new node
* IF head is not null
* New node points to head as the next nodeSET current node to point beyond next node
* DELETE temp node
* DECREASE size
* RETURN
* SET current node to next node
* Search method: Search for the specified bidId
* IF head is not null && head bidId is equal to input bidId
* SET head equal to next
* DECREASE size
* RETURN head bid
* CREATE new node starting at head
* WHILE current node is not null
* IF current bidId is equal to input bidId
* RETURN current bid
* Otherwise SET current node to next node
* RETURN bid

**Hash Tables Pseudocode:**

* START PROGRAM
* WHILE choice IS NOT equal to 9
* EXECUTE
* PRINT user menu:
* Load Bids
* Display All Bids
* Find Bid
* Remove Bid
* Insert Bid
* Exit
* READ IN user input and store to variable choice for menu selection.
* CHOICE 1 - (Load Bids)
* SET variable ticks equal to clock() method. Stores starting clock tick.
* INITIALIZE csv file
* PARSE csv file and stores data into bid object
* DEFINE vector to store collection of bids
* RETURN bids
* PRINT size of bids
* CALCULATE elapsed time
* Clock() – ticks
* PRINT elapsed time
* CHOICE 2 (Display All Bids)
* INVOKE PrintAll() // Loops through hash table and prints all elements
* BREAK;
* CHOICE 3 - (Find Bid)
* SET variable ticks equal to clock() method. Stores starting clock tick.
* SET bid equal to Search() method and pass in bidkey as a parameter.
* INVOKE Search()
* Search() will do the following:
* SET key equal to hashed bidID
* SET new pointer called “node” equal to reference of nodes element at keyth position.
* IF node IS NOT equal to null pointer AND node key IS NOT equal to UINT\_MAX AND node bidID compared to passed in bidID is a match
* RETURN node bidID;
* IF node IS EQUAL to null pointer OR node key IS equal to UINT\_MAX
* RETURN bid;
* WHILE node IS NOT equal to null pointer
* IF node key IS NOT equal to UINT\_MAX AND node bidID compare bidID IS a match
* RETURN node bidID;
* SET node equal to next node;
* RETURN bid;
* CALCULATE elapsed time
* IF bidId IS NOT empty
* INVOKE displayBid() method and pass in bid;
* ELSE
* PRINT not found message;
* PRINT clock ticks
* PRINT clock ticks per second
* BREAK;
* CHOICE 4 - (Remove Bid)
* INVOKE Remove() method with passed in bidId;
* Remove() method does the following:
* SET key equal to hashed bidID
* ERASE node starting at the beginning with the hashed key
* RESIZE nodes table
* CHOICE 5 - (Insert Bid)
* CREATE getBid() method as global method
* GetBid() method obtains input from user.
* SET bid equal to getBid() method
* INVOKE Insert() method and pass in bid
* Insert method does the following:
* SET key to hashed bidID
* IF the nodes keys at the keyth position IS equal to UINT\_MAX
* CREATE new node
* SET new node equal to Node with passed in bid and key
* SET nodes at keyth position equal to new node
* ELSE
* CREATE new pointer currNode // to store current node
* SET currNode equal to reference nodes at keyth position
* CREATE new pointer newNode // to store NEW node
* SET newNode equal to a new Node with passed in bid and key
* WHILE next currNode IS NOT equal to NULL
* POINT currNode to the next currNode
* SET currNode next equal to the newNode
* RESIZE nodes table
* INVOKE displayBid() and pass in bid
* BREAK;
* ELSE
* PRINT “Goodbye”
* RETURN 0
* END PROGRAM

**Tree Pseudocode:**

* Create root variable and set it equal to nullptr;
* CREATE a left and right Node;
  + CREATE function for inOrder so that it passes the root variable
  + CREATE funct for postOrder so that it passes the root variable;
  + CREATE funct for preOrder so that it passes the root variable;
* CREATE insert bid function
  + IF root == nullptr;
    - SET root equal to a new Node for a bid
  + ELSE simply add the Node root and bid
  + CREATE Remove function so that it takes the parameters root, and bidID
  + CREATE Search function with the parameter bidId
  + SET the current node equal to the root
  + WHILE current != nullptr
    - IF current bid is found, return current bid
    - IF the bid is smaller than current node transverse to the left
  + ELSE
    - transverse to left
* CREATE addNode function with parameters Node\*node, Bid bid
  + IF node is greater
    - THEN add to the left
  + IF there is no left node
    - THEN this node becomes left
  + ELSE
    - IF there is no right node,
    - THEN this node becomes right
  + ELSE recurse to the left
* CREATE inOrder function with parameters Node\*node
  + IF node != nullptr
    - SET inOrder to not left
      * PRINT bidID, title,amt, and fund
    - SET inOrder to right
* CREATE preOrder function with parameters Node\*node
  + IF node != nullptr
    - SET preOrder to not left
      * PRINT bidID, title,amt, and fund
    - SET preOrder to right
* CREATE postOrder function with parameters Node\*node
  + IF node != nullptr
    - SET postOrder to not left
      * PRINT bidID, title,amt, and fund
    - SET postOrder to right

**Menu Pseudocode:**

* print welcome message
* first char string convert to int
* while choice does not equal exit
* print menu
* if choice = 1 //Load Data Structure
* Output " Enter File Name”
* If correct file entered
* print “loaded successfully”
* else incorrect file
* print “no file found”
* return menu
* if choice = 2 //Print Course List
* print “sample”
* print sample schedule()s //course IDs alphabetized
* else
* print “load classes first”
* if choice = 3 //Print Course Info
* //validate choice
* print “enter courseID from list for info”
* for userChoice(type) = coursed
* print course information
* else
* print “load classes first”
* if choice != 9
* print “ invalid entry”
* else
* print “ Thank you for using course planner”
* End

**Runtime Analysis**

**Vector**

* void printCourseInformation(Vector<Course> courses, String courseNumber) {
* for all courses
* if the course is the same as courseNumber
* print out the course information
* for each prerequisite of the course
* print the prerequisite course information

A white sheet with black text

Description automatically generated

**Hashtable**

* void printCourseInformation(Hashtable courses, String courseNumber) {
* for all courses
* if the course is the same as courseNumber
* print out the course information
* for each prerequisite of the Hashtable[course]
* print the prerequisite course information }

A white sheet with black text

Description automatically generated

**Binary Search Tree**

* void printCourseInformation(Tree courses, String courseNumber) {
* for all Nodes
* if the course is the same as courseNumber
* print out the node's information
* if course has left node
* print left node as prerequisite course information
* if course has right node
* print right node as prerequisite course information
* end Function
* else
* if course has left node
* go to left node
* if course has right node
* go to right node

A white sheet with black text

Description automatically generated

All three have their pro and cons. So, I am going to focus on memory usage. Between vector and trees, trees have the advantage because of their use of nodes that use small, fixed memory chunks. Whereas vectors use large continuous memory blocks which could cause a system that has limited memory or where fragmentation happens often.

Now, that leaves us with hash tables and BST. BSTs are memory efficient when balanced versus hash tables in which can suffer from collisions and may require more memory upfront to handle those collisions and maintain performance. Hash tables and BST are very useful depending upon the application. Such as when you need ordered data, efficient range queries, and the ability to handle the complexity of maintaining balance. Now, if you need fast access, insertion, and deletion without any worries for order you may want to go with the hash table structure.

I believe that you need to know what your design is, complexities and what future growth will be to have a better understanding of which structure you need to use.