# CS 300 6-2 Project One

**Vector Data Structure Pesudode**

/\*\*

\* Default constructor

\*/

LinkedList::LinkedList() {

// Initialize housekeeping variables

//set head and tail equal to null

/\*\*

\* Append a new bid to the end of the list

\*/

void LinkedList::Append(Bid bid) {

// Implement append logic

//Create new node

//if there is nothing at the head...

// new node becomes the head and the tail

//else

// make current tail node point to the new node

// and tail becomes the new node

//increase size count

}

/\*\*

\* Prepend a new bid to the start of the list

\*/

void LinkedList::Prepend(Bid bid) {

// Implement prepend logic

// Create new node

// if there is already something at the head...

// new node points to current head as its next node

// head now becomes the new node

//increase size count

}

/\*\*

\* Simple output of all bids in the list

\*/

void LinkedList::PrintList() {

// Implement print logic

// start at the head

// while loop over each node looking for a match

//output current bidID, title, amount and fund

//set current equal to next

}

/\*\*

\* Remove a specified bid

\*

\* @param bidId The bid id to remove from the list

\*/

void LinkedList::Remove(string bidId) {

// Implement remove logic

// special case if matching node is the head

// make head point to the next node in the list

//decrease size count

//return

// start at the head

// while loop over each node looking for a match

// if the next node bidID is equal to the current bidID

// hold onto the next node temporarily

// make current node point beyond the next node

// now free up memory held by temp

// decrease size count

//return

// current node is equal to next node

}

/\*\*

\* Search for the specified bidId

\*

\* @param bidId The bid id to search for

\*/

Bid LinkedList::Search(string bidId) {

// Implement search logic

// special case if matching node is the head

// make head point to the next node in the list

//decrease size count

//return

// start at the head of the list

// keep searching until end reached with while loop (next != nullptr

// if the current node matches, return it

// else current node is equal to next node

//return bid

}

/\*\*

\* Returns the current size (number of elements) in the list

\*/

**Hash Tables Data Structure Pesudode**

START

CREATE hashTable class

INITIALIZE course <Node>nodes

VECTOR prerequisite\_list\_check = []

CREATE class object for COURSES(courseNumber, name, prerequisite1, prerequisite2,...,prerequisiteN)

Str courseNumber = “”

Str name = “”

Vector prerequisite\_list = empty list able to take strings

IF file is open then proceed to next step

ELSE PRINT(File Not Found)

Exit Program

OPEN file

WHILE line in file not end

GETLINE from file

CHECK line has min 2 entries

ADD prerequisite\_list\_check.append(line[0])

WHILE line in file not end

GETLINE from file

INITILIZE course object with line data

Create Node(course, key)

COURSE object

courseNumber = line[0]

name = line[1]

“Examples from document showed at most 2 prerequisites if more are needed would be easy enough to convert to a loop and start on the 3 elements and exit loop at end of the line”

If line[2] == True and in prerequisite\_list\_check

prerequisite\_list.append(line[2])

If line[3] == True and in prerequisite\_list\_check

prerequisite\_list.append(line[3])

Update node with nullptr to current

Insert new node with key and next to nullptr

PRINTALL()

FOR i=0 i<hashTable.size() i++

PRINT(Key, courseNumber, name,

FOR(j=0, j<prerequisite\_list.size(), j++)

PRINT(prerequisite)

END

**Tree Data Structure Pesudode**

START

CREATE binarySearchTree

INITIALIZE course <Node>nodes

VECTOR prerequisite\_list\_check = []

CREATE class object for COURSES(courseNumber, name, prerequisite1, prerequisite2,...,prerequisiteN)

Str courseNumber = “”

Str name = “”

Vector prerequisite\_list = empty list able to take strings

IF file is open then proceed to next step

ELSE PRINT(File Not Found)

Exit Program

OPEN file

WHILE line in file not end

GETLINE from file

CHECK line has min 2 entries

ADD prerequisite\_list\_check.append(line[0])

WHILE line in file not end

GETLINE from file

INITILIZE course object with line data

Create Node(course, key)

COURSE object

courseNumber = line[0]

name = line[1]

“Examples from document showed at most 2 prerequisites if more are needed would be easy enough to convert to a loop and start on the 3 elements and exit loop at end of the line”

If line[2] == True and in prerequisite\_list\_check

prerequisite\_list.append(line[2])

If line[3] == True and in prerequisite\_list\_check

prerequisite\_list.append(line[3])

INSERT course CREATE newNode

IF root == nullptr:

root == newNode

ELSE

insertNode(root, newNode) \* recursively called

insertNode(node, newNode)

if (newNode->courseNumber < node->courseNumber)

// New node should go to the left subtree

if (node->left == nullptr)

node->left = newNode

else

insertNode(node->left, newNode)

else

// New node should go to the right subtree

if (node->right == nullptr)

node->right = newNode

else

insertNode(node->right, newNode)

PRINTALL()

print(node)

if (node != nullptr) {

// Recursively traverse the left subtree

inOrder(node->left);

// Process the current node (print its course details)

cout << "Course Number: " << node->courseNumber << endl;

cout << "Name: " << node->name << endl;

FOR item in prerequisite\_list:

cout << "Prerequisite: " << item << endl;

// Recursively traverse the right subtree

inOrder(node->right);

}

END

## Example Runtime Analysis

When you are ready to begin analyzing the runtime for the data structures that you have created pseudocode for, use the chart below to support your work. This example is for printing course information when using the vector data structure. As a reminder, this is the same pairing that was bolded in the pseudocode from the first part of this document.

**Vector Data Structure**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **construct LinkedList** | 4 | 1 | 4 |
| **create course object** | 5 | 1 | 1 |
| **open file** | 1 | 1 | 1 |
| **loop through file check for errors** | 5 | n | n |
| **loop through file add data to new courses object** | 6 | n | n |
| **loop through courses and print the course information** | 6 | n | n |
| **Total Cost** | | | 3n + 6 |
| **Runtime** | | | O(n) |

**Hash Table Data Structure**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **construct hashtable** | 1 | 1 | n |
| **create course object** | 1 | n | n |
| **open file** | 1 | 1 | 1 |
| **loop through file check for errors** | 5 | n | n |
| **loop through file add data to new courses object** | 6 | n | n |
| **loop through courses and print the course information** | 6 | n | n |
| **Total Cost** | | | 3n + 6 |
| **Runtime** | | | O(n) |

**Binary Search Tree Data Structure**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **construct hashtable** | 1 | 1 | n |
| **create course object** | 1 | n | n |
| **open file** | 1 | 1 | 1 |
| **loop through file check for errors** | 5 | n | n |
| **loop through file add data to new courses object** | 6 | n | n |
| **loop through courses and print the course information** | 6 | n | n |
| **Total Cost** | | | 3n + 6 |
| **Runtime** | | | O(n) |

**Evaluation**

Three different data structures were used to preform the same operations. Take a file with data separated by commas and newlines chunk the data and test it for correctness. Then group it into objects and place it in a container. Finally, go through each object in the container and print it. Inserting objects into an unsorted container and then reading them out again one by one offers little advantage to any one of these data structures. Having the structures stored based on key or index would increase the efficiency of hashtables and binary search trees by a great deal. Insert in order for a hashtable is on average an O(1) operation. Insert in a binary search tree is O(log n). Insert operation for a vector is O(n) which is the worst possible case for the other two data structures.

The efficiencies of these data structures also shine through on searching for an individual object. They are all but eliminated when doing a one-by-one unsorted insert and unsorted printout. If the vector was to be sorted this would greatly increase the time it would take to properly sort the indexes and store the objects. Each Item would have to scan through the indexes of the items till it was able to find the proper location and reorder the vector. HashTables on the other hand use a function to compute a key that corresponds to the values being stored and this key is unique and laid out uniformly in an array and can be traversed easily with the known key. This allows operations to be efficient.

Binary Search Trees are a set of nodes with at most two children. The left child is less than the value of the parent, and the right is more. Inserting unordered data into this type of structure is very efficient because with every branch that is traversed approximately half of the nodes are eliminated. A BST with 32 nodes can be traversed in at most five moves. If the data is ordered, this can ruin the efficiency and make it perform in the range of O(n). In conclusion given a few factors vectors are by far the least efficient data structures. Hashtables are more efficient, but binary search trees are the most.