# CS 300 Pseudocode Document

Jordan Krueger

## Example Function Signatures

Below is an example of a function signature that you can use as a guide to help address the program requirements using each data structure for the milestones. The pseudocode for finding and printing course information is also given below and depicted in bold to help you get started. The provided pseudocode is for a vector data structure, so you may use this pseudocode in your first milestone as is. The hash table and tree structures are also shown below. But these structures are left for you to do in future milestones.

//Vector - Milestone 1

void searchCourse(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**

}

//Hash Table - Milestone 2

void searchCourse(HashTable<Course> courses, String courseNumber) {

}

//Binary Search Tree – Milestone 3

void searchCourse(Tree<Course> courses, String courseNumber) {

## Example Runtime Analysis

When you are ready to analyze the runtime for the Project One data structures for which you created the pseudocode, use the example chart below to support your work. This particular 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. The example only covers the search function for the vector structure. You do not have to complete your runtime analysis until Project One. However, working on your analysis now may help you understand the changes as you complete the milestones. Don’t forget to include your charts in Project One. You will submit Project One in Module Six.

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 1 | n | n |
| **if the course is the same as courseNumber** | 1 | n | n |
| **for each prerequisite of the course** | 1 | 1 | 1 |
| **for each prerequisite of the course** | 1 | n | n |
| **print the prerequisite course information** | 1 | n | n |
| **Total Cost** | | | 4n + 1 |
| **Runtime** | | | O(n) |

Pseudo Code Linked list/Vector

Appened(bid):

Create new node

if list is empty:

head and tail = new node

else:

tail.next = new node

size++

Prepend(bid):

Create new node

If list is empty:

Head and tail = new node

else:

new node.next = head

head = new node

size++

PrintList():

start at head

while current is not null:

print current.bid

move to next node

Search(bidId):

start at head

while current is not null:

if current.bidId matches:

return current.bid

move to next node

return empty bid

remove(bidId):

if list is empty:

return

if head matches bidId:

move head to next node

delete old head

size—

return

start at head

while current.next is not null:

if current.next matches bidId:

temp = current.next

delete temp

size—

return

move to next node

END

Pseudo Code for Hash Table

START

DEFINE STRUCT Bid:

bidId, title, fund, amount

DEFINE STRUCT Node:

bid (bid)

Key (unsigned int)

Next (pointer to Node)

DEFINE CLASS HashTable:

tableSize <- default

nodes <- vector of Node (size = tableSize)

FUNCTION hash(key):

RETURN key MODULO tableSize

FUNCTION Insert(bud):

Key <- hash(atoi)bid.bidId))

IF nodes[key] is empty:

Assign new node to nodes[key]

ELSE:

Traverse linked list at nodes[key] to end

Append new node

FUNCTION search(bidId)

key<- hash(atoi(bidId))

current <\_ nodes[key]

WHILE current is not null:

IF current.bid.bidId equals bidID:

RETURN current bidId

Current <- current.next

RETURN empty bid

FUNCTION Remove(bidId):

Key<- hash(atoi(bidId))

Current <- nodes[key], prev <- null

WHILE current not null:

IF current.bidId equals bidId:

If prev is null:

IF current.next exists:

Overwrite current with next node

delete next

ELSE:

Reset current node

ELSE:

Prev.next <- current.next

Delete current

RETURN

Prev <- current

Current <- current.next

FUNCTION PrintAll():

FOR each node in table:

IF node has valid key:

Print node.bid

WHILE node.next exists:

Node<- node.next

Print node.bid

END

Pseudo Code for Binary Tree

Class: Node

-bid: Bid

-left: Node pointer

-right: Node pointer

Class: BinarySearchTree

-root: Node pointer

Constructor:

-Initialize root to nullptr

Destructor:

-Call postOrderDelete(root) to recursively delete all nodes

Insert(bid);

-If root is null

- create new Node with bid and assign to root

-Else

-call addNode(root, bid)

addNode(node, bid):

-If bid ID < node’s bid ID

-If left is null, insert left

-Else recurse left

-Else

-If right is null, insert right

-Else recurse right

Search(bidId):

-Start at root

-While current is not null

-If current.bidId matches, return current bid

-If bidId < current, go left

-Else go right

-Return default bid if not found

Remove(bidId):

-set root = removeNode(root, bidId)

removeNode(node, bidId):

-If node is null, return

-If bidId < node, recurse left

If bidId > node, recurse right

-If match found:

-If no children, delete node

-If one child, return non-null child

-If two children:

-Find smallest in right subtree

-Copy that bid to current node

-Recursively remove that successor

Traversal (inOrder, preorder, postOrder):

-inOrder: left -> Node -> Right

-preOrder: Node -> Left -> Right

-postOrder: Left -> Right -> Node

END

Linked list Advantages/Disadvantages

Advantages:

* Easy to build and understand.
* Takes up almost no memory.
* Good to store and print everything in order.

Disadvantages

* Courses are not in the order you want so you would have to sort them yourself.
* Not good for large numbers of courses.
* You would have to check for a specific course which means longer down time.

Hash Table Advantages/Disadvantages

Advantages

* Fast to find a course by the ID.
* Fast to add or remove courses.
* Does not slow down even with a lot of courses.

Disadvantages

* Can use up a lot of memory.
* Does not keep courses in alphabetical order.
* Slightly harder to build than a linked list.

Binary Tree Advantages/Disadvantages

Advantages

* Better for organizing and searching a list.
* Within the traversal in-order you can quickly print all courses, sorted, easily.
* You can search, add, and remove everything pretty fast as long as the tree stays balanced.

Disadvantages

* A bit harder to build than a list or hash table.
* If the tree becomes unbalanced it can become slower.
* Takes a bit more planning to make sure things go smoothly.

LinkedlList Analysis table:

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # times executed | Total cost |
| Create Vector | 1 | 1 | 1 |
| For each line in file | 1 | N | N |
| Create vector course item | 1 | N | N |
| While prereq exists | 1 | N | n |
| Append prereq | 1 | N | n |
| Push course item to vector | 1 | N | N |
| Total cost | 1 | N | N |
| Runtime |  |  | 7n+1 |
|  |  |  |  |
|  |  |  |  |

Hash Table Analysis

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executed | Total cost |
| Create Hash Table | 1 | 1 | 1 |
| For each new line in file | 1 | N | N |
| Create course item | 1 | N | N |
| While prereq exists | 1 | N | N |
| Append prereq | 1 | N | N |
| Insert course item | 1 | N | N |
| Create key for course | 1 | N | N |
| If no entry found | 1 | N | n |
| Assign node to key | 1 | N | n |
| Else: handle chaining or replace node | 4 | N | 4n |
| Else: traverse to end and append new node | 2 | N | 2n |
| Total cost |  |  | 15n+1 |
|  |  |  |  |

Binary Tree Analysis

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line cost | # times executed | Total cost |
| Add node method base | 1 | 1 | 1 |
| If course ID < root | 1 | N | N |
| If no left, set left | 1 | N | N |
| If course ID > root | 1 | N | N |
| If no right, set right | 1 | N | N |
| For each line in file | 1 | N | N |
| Create vector course item | 1 | N | N |
| While prereq exists | 1 | n | N |
| Append prereq | 1 | N | N |
| Insert course into tree | 1 | N | n |
| Total cost |  |  | 12n+2 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Recommendation for data structure:

After looking through all three data structures I think the binary tree search is best for this program. The binary Tree keeps everything sorted automatically so I do not have anything extra to sort (i.e manually). It is fast when it comes to searching for courses. I would say it is more organized than a linked list and gives me the benefits that the hash table does not provide. A linked list is or would be easy to use in this case but would take a while to search for courses. A hash table is fast at finding things but not very good when it comes to organization. The binary tree is the middle ground here and the best fit for the program.