# CS 300 Pseudocode Document

**Shared object, CSV loader, and “print one course”**  
  
define Course:

id ← string

title ← string

prereqs ← list of strings

loadCsv(filePath, dataStructure):

open the CSV file

for each line in the file:

split the line by commas into tokens

if there are fewer than 2 tokens: skip the line

make a new Course called c

c.id ← tokens[0]

c.title ← tokens[1]

for every token starting at index 2:

add that token to c.prereqs

insert c into the chosen data structure # vector | hash table | bst

close the file

printCourse(c):

if c is null:

print "Course not found"

else:

print c.id + " " + c.title

for each prereq in c.prereqs:

print " prereq: " + prereq

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
  
**Menu loop**repeat

print "1 Load 2 Print-All 3 Find Course 9 Exit"

read choice

if choice is 1:

loadCsv(csvPath, ds)

if choice is 2:

ds.printAllSorted()

if choice is 3:

read searchId

result ← ds.search(searchId)

printCourse(result)

until choice is 9  
  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
  
**Vector pseudocode (Milestone 1)**  
  
# storage: a simple list called vec

insertVector(c):

append c to vec

selectionSort(vec):

for each index i from 0 to (size − 2):

set minIndex to i

for each index j from (i + 1) to (size − 1):

if s at j is less than title at minIndex:

update minIndex to j

if minIndex is not i:

swap the elements at i and minIndex

quickSort(vec, begin, end):

if begin ≥ end: return

mid ← partition(vec, begin, end)

quickSort(vec, begin, mid)

quickSort(vec, mid + 1, end)

partition(vec, begin, end):

set low ← begin

set high ← end

set pivot ← title at the middle index

loop until low ≥ high:

while title at low < pivot: increment low

while title at high > pivot: decrement high

if low < high:

swap elements at low and high

increment low; decrement high

return high

printAllSorted\_Vector():

quickSort(vec, 0, size − 1)

for each course in vec:

printCourse(course)

searchVector(id):

for each course in vec:

if course.id equals id: return course

return null  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
  
**Hash-table pseudocode (Milestone 2)**# storage: vector of 179 buckets, each bucket is a linked list

hash(key):

return key mod 179

insertHash(c):

index ← hash(c.id as int)

add c to the head of bucket[index]

searchHash(id):

index ← hash(id as int)

node ← bucket[index]

while node is not null:

if node.course.id equals id: return node.course

move to node.next

return null

removeHash(id):

index ← hash(id as int)

prev ← null

node ← bucket[index]

while node is not null:

if node.course.id equals id:

if prev is null: bucket[index] ← node.next

else: prev.next ← node.next

delete node

return

prev ← node

node ← node.next

printAllSorted\_Hash():

tempList ← empty list

for every bucket in the table:

node ← bucket head

while node is not null:

append node.course to tempList

node ← node.next

quickSort(tempList, 0, tempList.size − 1) # reuse vector quick sort

for each course in tempList:

printCourse(course)  
  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
  
**Binary-search-tree pseudocode (Milestone 3)**# each tree node holds a Course and left / right links

insertBST(node, c):

if node is null:

return new Node(c)

if c.id < node.course.id:

node.left ← insertBST(node.left, c)

else:

node.right ← insertBST(node.right, c)

return node

searchBST(node, id):

while node is not null:

if id equals node.course.id: return node.course

if id < node.course.id: node ← node.left

else: node ← node.right

return null

removeBST(node, id):

if node is null: return null

if id < node.course.id:

node.left ← removeBST(node.left, id)

else if id > node.course.id:

node.right ← removeBST(node.right, id)

else:

# node to delete found

if node has no children: delete node; return null

if node has one child: return that child

# two children – find successor

succ ← node.right

while succ.left is not null: succ ← succ.left

node.course ← succ.course

node.right ← removeBST(node.right, succ.course.id)

return node

inOrder(node):

if node is null: return

inOrder(node.left)

printCourse(node.course)

inOrder(node.right)

printAllSorted\_BST():

inOrder(root)  
  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

| **Data structure** | **Per-line cost** | **# lines (n)** | **Worst-case time** | **Why** |
| --- | --- | --- | --- | --- |
| **Vector** (append) | 1 to read & parse + 1 to pushBack | n | **Θ(n)** | pushBack is constant-time (occasionally resizes, but that’s amortized O(1)). So the loop is simply 2 × n ≈ Θ(n). |
| **Hash table** (separate chaining) | same 2 as above + 1 to compute hash + 1 to prepend into bucket | n | **Θ(n)** | Hash and head-insert are both O(1), so it’s 4 × n total. |
| **Binary search tree** (unbalanced) | read/parse 2 + **log n** comparisons on average per insert | n | **Θ(n log n)** average;**Θ(n²)** worst case | Each insert walks the tree depth. Balanced ≈ log n, so n × log n. If the file arrives already sorted the tree degenerates to a list and depth becomes n, giving n × n. |

Cost Model and Memory Breakdown:

Every basic line = 1. Calling a helper? Count that helper’s run-time.

Memory-wise, all three keep one Course object per line. Vector adds nothing. Hash adds one next pointer on collision. BST adds two child pointers per node.

| **Data structure** | **Advantages** | **Disadvantages** |
| --- | --- | --- |
| **Vector** | • Very compact in memory (contiguous array).  • Quick-sort once ⇒ fast sequential prints.  • Simple code—no pointers to manage. | • Search is linear (O (n)).  • Inserting or deleting in the middle costs O (n).  • Quadratic selection-sort becomes unusable on big files. |
| **Hash table** | • Average-case O (1) insert, search, delete.• Fastest single-course lookup on this dataset.• No need to re-size for moderate growth. | • Output is not ordered; must gather + sort to print.  • Extra memory for buckets + links.• Worst-case lookup degrades to O (n) if many collisions. |
| **Binary search tree** | • Maintains alphabetical order by design.  • O (log n) search / insert / delete when balanced.  • In-order traversal prints the list without extra sorting. | • Each node stores two child pointers (memory overhead).  • Load is slower than vector/hash due to pointer churn.  • If data arrives already sorted and tree isn’t balanced, performance falls to O (n). |

My Recommendation:  
Use the binary search tree for ABCU’s advising system.

* It does lookups in log-time. On our dataset, that rounds to 0 ms. Hash table's O(1) doesn’t make a difference when you’re pulling just one course.
* BST stores courses in sorted order by default. That means printing all courses (option 2) just needs an in-order traversal, done in 2.47s. The hash table had to gather and sort, taking 3.32s.
* Load time was 0.054s, only a few ms slower than vector/hash. Memory overhead is minor: just two pointers per node.
* Worst-case depth (O(n))? Not an issue if you use a self-balancing tree like AVL or Red-Black.

You get instant searches and a ready-to-print sorted list. The BST handles both advising tasks cleanly with the least hassle.