**Runtimes:**

**Vector:** The runtime cost for the vector does not exceed O(n)

**Hash:** The runtime cost for hash tables does not exceed O(n)

**Tree:** The runtime cost for binary trees does not exceed O(n)

**Pros and Cons of Data Structures:**

Vectors have built in functional abilities. You can easily append data to the list. However, inserting and removing items from the middle of the list can be more difficult. The whole list needs to be shifted one way or the other when performing these functions.

Hash tables use keys to access data stored in a vector. Hash tables are best for large amounts of data since you can access the data through a key. Collisions are where the hash table struggles. If too many items have the same key you can run into trouble.

Binary trees are useful because each node is followed by a sorted subtree. Searching through a binary tree is efficient. There is no sorting needed because of the basic structure. One disadvantage of trees is that with more data the longer data access can take. Imbalanced trees also increase complexity.

void displayCourses(Courses course)

{

cout << course.courseId << ": " << course.courseName << endl;

cout << "Prerequisites: " << endl;

}

/\*\*

\* Partition the vector of courses into two parts, low and high

\*

\* @param courses Address of the vector<courses> instance to be partitioned

\* @param begin Beginning index to partition

\* @param end Ending index to partition

\*/

int partition(vector<Courses>& courses, int begin, int end) {

//set low and high equal to begin and end

int low = begin;

int high = end;

// pick the middle element as pivot point

int middle = begin + (end - begin) / 2;

bool done = false;

// while not done

while (!done)

{

// keep incrementing low index while courses[low] < courses[pivot]

while (courses.at(low).courseId.compare(courses.at(middle).courseId) < 0)

{

}

// keep decrementing high index while courses[pivot] < courses[high]

while (courses.at(middle).courseId.compare(courses.at(high).courseId) < 0)

{

high--;

}

/\* If there are zero or one elements remaining,

all bids are partitioned. Done with loop. Return high \*/

if (low >= high)

{

done = true;

}

// else swap the low and high courses (built in vector method)

else

{

swap(courses.at(low), courses.at(high));

// move low and high closer ++low, --high

low++;

high--;

}

}

return high;

}

/\*\*

\* Perform a quick sort on course number

\* @param courses address of the vector<Courses> instance to be sorted

\* @param begin the beginning index to sort on

\* @param end the ending index to sort on

\*/

void quickSort(vector<Courses>& courses, int begin, int end) {

//set mid equal to 0

int mid = 0;

/\* Base case: If there are 1 or zero courses to sort,

partition is already sorted otherwise if begin is greater

than or equal to end then return\*/

if (begin >= end)

{

return;

}

/\* Partition courses into low and high such that

midpoint is location of last element in low \*/

mid = partition(courses, begin, end);

// recursively sort low partition (begin to mid)

quickSort(courses, begin, mid);

// recursively sort high partition (mid+1 to end)

quickSort(courses, mid + 1, end);

}

case 2:

// sort the vector for ordered printing

quickSort(courses, 0, courses.size() - 1);

// Loop and display the courses read

for (int i = 0; i < courses.size(); ++i) {

displayCourses(courses[i]);

}

cout << endl;

break;