Jaehyeong Lee

7/20/18

Student ID: 004950987

**CS32 Project 2 Report**

* A description of the design of your implementation and why you chose it. (A couple of sentences will probably suffice, perhaps with a picture of a typical List and an empty List. Is your list circular? Does it have a dummy node? What's in your nodes?)

For CoachingStaff program, I implemented a non-cyclic double-linked list because it made traversing back through the nodes much easier and efficient. Each node contains string values that represent first name, last name, and unique number for each coach. The linked list also has head and tail pointer that consistently update their positions as first and last node of the list.

* A brief description of notable obstacles you overcame.

Notable obstacles that I’ve faced were during the implementation of mergeStaffs function. There were many scenarios to consider for this function since I had to take into account whether the object passed to the function was empty, whether two objects being compared had the same full name, and if that full name has different unique value (which in case has to be deleted), and etcetera. However, by adopting ‘flag’ variable I successfully checked for every scenario and executed corresponding action.

* Pseudocode for non-trivial algorithms (e.g., CoachingStaff::fireCoach and mergeStaffs)

For mergeStaffs, I implemeneted following algorithms:

bool mergeStaffs(const CoachingStaff & csOne,

const CoachingStaff & csTwo,

CoachingStaff & csMerged)

{

if csOne and csTwo are both empty, set csMerged empty

if csOne is empty but csTwo is not, set csMerged to csTwo

if csTwo is empty but csOne is not, set csMerged to csOne

else

{

Set csMerged to csOne

For every item in csOne, if first item in csTwo has same full name and unique number, do nothing and continue

For every item in csOne, if first item in csTwo has same full name and different number delete first item from csMerged

If none of above applies, add the item to csMerged

Now do the same until the last item in csTwo

}

}

For fireCoach function, I implemented the following algorithm:

bool CoachingStaff::fireCoach(const std::string& firstName, const std::string&

lastName)

{

if list is empty, exit

if list has only one element and its values match that of input, delete the node and set head and tail pointer to null

while node is not null, continue iterating through every node to check if any node has same full name as the input

if list has more than one element and first node is being deleted, delete the first node and reset the head pointer to the next node

if list has more than one element and last node is being deleted, delete the last node and reset the tail pointer to the previous node

if none of the node matches the input names, return false

}

* A list of test cases that would thoroughly test the functions. Be sure to indicate the purpose of the tests.

void noCoaches\_TEST()

{

CoachingStaff obj1, obj2;

obj2.hireCoach("Steven", "Gaby", "56");

assert(obj1.noCoaches() == 1);

assert(obj2.noCoaches() == 0);

std::cout << "Passed all tests" << std::endl;

}

This case tests if noCoaches function was implemented correctly; it returns true if the list is empty and false if it is non-empty

void numberOfCoaches\_TEST()

{

CoachingStaff obj1, obj2;

obj2.hireCoach("Steven", "Gaby", "56");

obj2.hireCoach("", "Schoff", "46");

obj2.hireCoach("Steven", "", "60");

obj2.hireCoach("David", "Schoff", "46");

assert(obj2.numberOfCoaches() == 4);

assert(obj1.numberOfCoaches() == 0);

std::cout << "Passed all tests" << std::endl;

}

This case tests if numberOfCoaches function was implemented correctly; it returns the number of nodes in the list.

void hireCoach\_TEST()

{

CoachingStaff obj1, obj2;

obj2.hireCoach("Steven", "Gaby", "56");

assert(obj2.hireCoach("Steven", "Gaby", "56") == 0);

assert(obj2.hireCoach("Steven", "Gaby", "60") == 0);

assert(obj1.hireCoach("", "Davidson", "25") == 1);

std::cout << "Passed all tests" << std::endl;

}

This case tests if hireCoach function was implemented correctly; it adds node with the corresponding name and value, if it already doesn’t exist within the list

void renameCoach\_TEST()

{

IType value1, value2;

std::string first, last;

CoachingStaff obj1, obj2;

obj2.hireCoach("Steven", "Gaby", "56");

assert(obj2.renameCoach("Steven", "Gaby", value1) == 1 && obj2.whichCoach(0, first, last, value2) && value1 == value2);

assert(obj1.renameCoach("Steven", "Gaby", value1) == 0);

std::cout << "Passed all tests" << std::endl;

}

This case tests if renameCoach function was implemented correctly; it reassigns the value to the node that matches the input name

void hireOrRename\_TEST()

{

IType value1, value2;

std::string first, last;

CoachingStaff obj1, obj2;

obj2.hireCoach("Steven", "Gaby", "56");

assert(obj2.renameCoach("Steven", "Gaby", value1) == 1 && obj2.whichCoach(0, first, last, value2) && value1 == value2);

assert(obj1.renameCoach("Steven", "", value1) == 0);

std::cout << "Passed all tests" << std::endl;

}

This case tests if hireOrRename function was implemented correctly; it reassigns the value to the node if it has the same name as the input, and if not, adds it.

void fireCoach\_TEST()

{

CoachingStaff obj1, obj2;

obj2.hireCoach("Steven", "Gaby", "56");

assert(obj2.fireCoach("Steven", "Gaby") == 1);

assert(obj2.fireCoach("David", "Hogg") == 0);

assert(obj1.fireCoach("", "") == 0);

std::cout << "Passed all tests" << std::endl;

}

This case tests if fireCoach function was implemented correctly; it removes the node that matches the input.

void coachOnStaff\_TEST()

{

CoachingStaff obj1, obj2;

obj2.hireCoach("Steven", "Gaby", "56");

assert(obj2.coachOnStaff("Steven", "Gaby") == 1);

assert(obj2.coachOnStaff("David", "") == 0);

assert(obj1.coachOnStaff("Jennifer", "Nguyen") == 0);

std::cout << "Passed all tests" << std::endl;

}

This case tests if coachOnStaff function was implemented correctly; it returns true if the input name exists within the list and false if otherwise.

void findCoach\_TEST()

{

IType value;

IType value2 = "5";

CoachingStaff obj1, obj2;

obj2.hireCoach("Steven", "Gaby", "56");

assert(obj2.findCoach("Steven", "Gaby", value) == 1 && value == "56");

assert(obj2.findCoach("", "Hawking", value) == 0 && value == "56");

assert(obj2.findCoach("David", "Hogg", value2) == 0 && value2 == "5");

std::cout << "Passed all tests" << std::endl;

}

This case tests if findCoach function was implemented correctly; it sets the value to the value of the node that matches the name of input. If such node doesn’t exist, returns false.

void whichCoach\_TEST()

{

std::string first, last;

IType value;

CoachingStaff obj1;

obj1.hireCoach("Gabriel", "Mandez", "26");

obj1.hireCoach("Jay", "", "27");

obj1.hireCoach("Chris", "Choi", "26");

obj1.hireCoach("", " ", "30");

assert(obj1.whichCoach(-1, first, last, value) == 0);

assert(obj1.whichCoach(0, first, last, value) && first == "" && last == " " && value == "30");

assert(obj1.whichCoach(1, first, last, value) && first == "Chris" && last == "Choi" && value == "26");

assert(obj1.whichCoach(2, first, last, value) && first == "Jay" && last == "" && value == "27");

assert(obj1.whichCoach(3, first, last, value) && first == "Gabriel" && last == "Mandez" && value == "26");

assert(obj1.whichCoach(4, first, last, value) == 0);

std::cout << "Passed all tests" << std::endl;

}

This case tests if whichCoach function was implemented correctly; it sets the first name, last name, and value to the node of corresponding index. If index is out of the boundary, it returns false.

void changeStaff\_TEST()

{

std::string first, last;

IType value;

CoachingStaff obj1, obj2;

obj1.hireCoach("Gabriel", "Mandez", "26");

obj1.hireCoach("Jay", "", "27");

obj1.hireCoach("Chris", "Choi", "26");

obj1.hireCoach("", " ", "30");

obj2.hireCoach("Steven", "Gaby", "56");

obj2.hireCoach("Steven", "Gaby", "60");

obj2.hireCoach("", "Davidson", "25");

obj1.changeStaff(obj2);

assert(obj1.whichCoach(0, first, last, value) && first == "" && last == "Davidson" && value == "25");

assert(obj1.whichCoach(1, first, last, value) && first == "Steven" && last == "Gaby" && value == "56");

assert(obj2.whichCoach(0, first, last, value) && first == "" && last == " " && value == "30");

assert(obj2.whichCoach(1, first, last, value) && first == "Chris" && last == "Choi" && value == "26");

assert(obj2.whichCoach(2, first, last, value) && first == "Jay" && last == "" && value == "27");

assert(obj2.whichCoach(3, first, last, value) && first == "Gabriel" && last == "Mandez" && value == "26");

std::cout << "Passed all tests" << std::endl;

}

This case tests if changeStaff function was implemented correctly; it swaps the contents of two CoachingStaff objects.

void mergeStaffs\_TEST()

{

std::string first, last;

IType value;

CoachingStaff obj1, obj2, merged;

obj1.hireCoach("Gabriel", "Mandez", "26");

obj1.hireCoach("Jay", "", "27");

obj1.hireCoach("Chris", "Choi", "26");

obj1.hireCoach("", " ", "30");

obj2.hireCoach("Gabriel", "Mandez", "30");

obj2.hireCoach("Jay", "", "27");

obj2.hireCoach("", "Davidson", "25");

mergeStaffs(obj1, obj2, merged);

assert(merged.numberOfCoaches() == 4);

assert(merged.whichCoach(0, first, last, value) && first == "" && last == " " && value == "30");

assert(merged.whichCoach(1, first, last, value) && first == "Chris" && last == "Choi" && value == "26");

assert(merged.whichCoach(2, first, last, value) && first == "" && last == "Davidson" && value == "25");

assert(merged.whichCoach(3, first, last, value) && first == "Jay" && last == "" && value == "27");

std::cout << "Passed all tests" << std::endl;

}

This case tests if mergeStaff function was implemented correctly; it merges two contents of CoachingStaff objects and insert it to the merged object.

void searchStaffs\_TEST()

{

std::string first, last;

IType value;

CoachingStaff obj1, obj2, result;

obj1.hireCoach("Gabriel", "Mandez", "26");

obj1.hireCoach("Jennifer", "Nguyen", "27");

obj1.hireCoach("Chris", "Mandez", "26");

obj1.hireCoach("Peter", "David", "30");

obj2.hireCoach("Gibb", "David", "30");

obj2.hireCoach("Jennifer", "Kim", "40");

obj2.hireCoach("Gibb", "Nguyen", "26");

searchStaff("\*", "Mandez", obj1, result);

assert(result.whichCoach(0, first, last, value) && first == "Chris" && last == "Mandez");

assert(result.whichCoach(1, first, last, value) && first == "Gabriel" && last == "Mandez");

searchStaff("Gibb", "\*", obj2, result);

assert(result.whichCoach(0, first, last, value) && first == "Gibb" && last == "David");

assert(result.whichCoach(1, first, last, value) && first == "Gibb" && last == "Nguyen");

result.hireCoach("Jordan", "Peter", "62");

searchStaff("\*", "\*", obj2, result);

assert(result.whichCoach(0, first, last, value) && first == "Gibb" && last == "David");

assert(result.whichCoach(1, first, last, value) && first == "Jennifer" && last == "Kim");

assert(result.whichCoach(2, first, last, value) && first == "Gibb" && last == "Nguyen");

std::cout << "Passed all tests" << std::endl;

}

This case tests if searchStaff function was implemented correctly; it searches items corresponding to the input and copies them to the result object.