Christina De Cesaris

Project 2 Report:

My program was built as a standard doubly linked list without a tail node. I did not feel the tail node was necessary for the current implementations and was that its inclusion would be more hindering than useful in this case. An empty list will always be a single head pointer set to nullptr and a m\_size int variable initialized to zero. As items are added to the list, the signPlayer function adds the Player struct alphabetically according to each name. Each node contained a previous and next pointer.

Some of the obstacles I encountered were the result of my test cases (fortunately!). For one, I learned that in my renouncePlayer function, I had a bug which would only reveal itself when every player on a given list had been renounced. In this case, the head pointer originally would not be set to null and thus be pointing somewhere random which became very problematic as the destructor then was unable to properly deallocate the memory. Luckily the fix consisted of finding where renounce player checks if the traversing pointer ->next == nullptr.

I also had a hang up when checking if the same address had been passed into the joined rosters function. Initially, if such an event occurred, when the bbJoined roster was emptied, it would also empty either bbOne or bbTwo if either of those shared the address of bbJoined. After many a debug, I found I would have to make a copy of bbJoined and then call the function again with that copy in the place of bbJoined. Of course, then I had to return to prevent the function from continuing with the original values.

**Pseudocode:**

renouncePlayer:

//check if player is on the roster first

// create a transversal pointer to the head node

//while that node is not null go down the list

//if we have found the name we want to remove

//if m is at the head and m-> next is null, then we must delete m AND set head to null to prevent a later on memory leak!!!!

//otherwise, if m is at head and there is a node after head, so we have to first set head to the next node, and then delete the isolated node, each time we delete, we must decrement size

//if we find our node we want to delete and aren’t at the head, and there is a node after wards, we then set its next’s prev equal it its prev, and its prev’s next equal to its next, now we can elimitate it (and ofc decrement the size)

//Once we remove any node from a list we should return true, I (ab)use return true to prevent the loop from continuing. Once we have removed the node we need to remove we don’t need to do anything else and don’t want to accidentally move along the list and perhaps find ourselves at a nullptr (and then try to access it D: )

joinRosters:

The following is the general outline and thought dump I created before writing this function:

//at the end of the program, if(dupes){return false;}

//make sure bbJoined and bbOne or bbTwo doesn’t share its address, if so then we need to make a temp of bbJoined and send that through the program, we cannot empty bbJoined because we will also empty one of the const lists which causes logical errors

//then if bbJoined isn’t empty we need to make it empty

//go down roster one, get each player's name and val one at a time

//store those values in a temp and see if those players with those values

//exist at all on the second list if NOT then player add vals from one to joined and add vals from two to joined

//if we have a same name different value case, then we have to not add that name at all to joined, maybe have a continue?

//maybe if that is the case, then we store that name in another temp, and we check if it is on the list when we are adding

//if we find that name in this case, we don't added it at all, we do this for both lists so it isn't added at all

//the last case is if we have the same name and same val twice, we only want to add the name once in this case

//another way would be to add everything, the sign player won't allow double adds.. so that is taken care of actually

//in this case, our issue becomes same name and different value, hmm in this case actually

//we can look into the list and if we have two players same name and different values, remove both of them.

//after creating bbJoined, we might have a value which is not supposed to exist. The signPlayer alg will take care of duplicates but if we have a duplicate that had a different value, it WILL add one of them.

//To check for this, I created a Bool value called dupes, then after bbJoined is created, I go through both lists and compare if they have a same name, different value case, if one is found, dupes is set to true, and that name is removed from bbJoined.

signPlayer:

//check if empty list aka see if head points to null and allocate node appropriately to the head.

//if the name already exists, return false, make no changes to the list

//otherwise, when a node is added it most be added alphabetically by last name and then by first name if last names are equal

//in general, strings are compared via <, >, == operators and are added above a node in the list given it is alphabetically “greater” in name than that node.

//special cases are if we are adding before the head node, adding at the last node, or adding after the head node.

checkRoster: //the comments I used to build my code

//returns bb result as a copy of elements which match the search elements

//look up wild card implementations if needed

//ensure bbResult is EMPTY because it cannt contain other eelements

//how about if \* is first name then add all with matching last name

//if \* in last name add all with matching first name

//if no star, match based on first and last add elements who match only

//if both stars are passed, then copy input list to empty

**Test cases:**

void testone(int n)

{

BballRoster empty;

BballRoster l1;

l1.signPlayer("angela", "li", 2); l1.signPlayer("veronica", "czaja", 0); l1.signPlayer("chloe", "janiser", 1);

BballRoster l2;

l2.signPlayer("joe", "distefano", 1); l2.signPlayer("giovanna", "de cesaris", 0);

SomeType x;

string ftemp, ltemp;

switch (n)

{

default: {

cout << "bad argument" << endl;

} break; case 1: {

assert(empty.howManyPlayers() == 0);

} break; case 2: {

assert(l1.howManyPlayers() == 3);

} break; case 3: {

assert(!(empty.choosePlayer(0, ftemp, ltemp, x)));

} break; case 4: {

assert(l1.choosePlayer(0, ftemp, ltemp, x) && x == 0);

} break; case 5: { //CHOSE Player

assert(l1.choosePlayer(2, ftemp, ltemp, x) && x == 2);

} break; case 6: {//Mess with signOrResign and Resign functions

BballRoster A;

assert(A.signOrResign("A", "AA", 0)); assert(A.signOrResign("A", "AB", 1));

assert(A.signOrResign("A", "AC", 2));

assert(A.resignPlayer("A", "AA", 1));

assert(A.choosePlayer(0, ftemp, ltemp, x) && x == 1);

assert(A.choosePlayer(2, ftemp, ltemp, x) && x == 2);

assert(!A.resignPlayer("B", "c", 3));

assert(A.signOrResign("B", "c", 3));

assert(A.choosePlayer(3, ftemp, ltemp, x) && x == 3);

} break; case 7: { //COPY CONSTRUCTER

BballRoster l3(l1);

assert(l3.howManyPlayers() == 3);

assert(l1.howManyPlayers() == 3);

assert(l1.choosePlayer(1, ftemp, ltemp, x) && x == 1);

assert(l3.choosePlayer(1, ftemp, ltemp, x) && x == 1);

//assert(13.resignPlayer())

} break; case 8: {//ASSIGNMENT OPERATOR TIME

BballRoster l3;

l3.signPlayer("ah", "ha", 12); //start with non empty

l3 = l1;

assert(l3.howManyPlayers() == 3);

assert(l1.howManyPlayers() == 3);

//assert(l3.assert(!(empty.choosePlayer(0, ftemp, ltemp, x)));

//assert(1, ftemp, ltemp, x) && x == 20);

} break; case 9: { //just for empy function

BballRoster l3(empty);

assert(l3.howManyPlayers() == 0);

}break; case 10: { //joinRoster function, test also for same address complicaitons

BballRoster joiner;

assert(joinRosters(l1, l2, joiner));

assert(joiner.howManyPlayers() == 5);

assert(l1.howManyPlayers() == 3);

assert(l2.howManyPlayers() == 2);

assert(joiner.choosePlayer(0, ftemp, ltemp, x) && x == 0);

assert(joiner.choosePlayer(1, ftemp, ltemp, x) && x == 0);

assert(joiner.choosePlayer(2, ftemp, ltemp, x) && x == 1);

assert(joiner.choosePlayer(3, ftemp, ltemp, x) && x == 1);

assert(joiner.choosePlayer(4, ftemp, ltemp, x) && x == 2);

assert(joinRosters(l1, joiner, joiner));

assert(joiner.choosePlayer(0, ftemp, ltemp, x) && x == 0);

assert(joiner.choosePlayer(1, ftemp, ltemp, x) && x == 0);

assert(joiner.choosePlayer(2, ftemp, ltemp, x) && x == 1);

assert(joiner.choosePlayer(3, ftemp, ltemp, x) && x == 1);

assert(joiner.choosePlayer(4, ftemp, ltemp, x) && x == 2);

BballRoster empter;

assert(joinRosters(empter, empter, empter));

assert(empter.howManyPlayers() == 0);

assert(joinRosters(joiner, joiner, joiner));

assert(joiner.choosePlayer(0, ftemp, ltemp, x) && x == 0);

assert(joiner.choosePlayer(1, ftemp, ltemp, x) && x == 0);

assert(joiner.choosePlayer(2, ftemp, ltemp, x) && x == 1);

assert(joiner.choosePlayer(3, ftemp, ltemp, x) && x == 1);

assert(joiner.choosePlayer(4, ftemp, ltemp, x) && x == 2);

}break;

case 11: {

BballRoster l4;

l4.signPlayer("angela", "li", 3); l4.signPlayer("veronica", "czaja", 9); l4.signPlayer("chloe", "janiser", 2);

BballRoster memer;

assert(!joinRosters(l4, l1, memer));

assert(memer.howManyPlayers() == 0);

} break;

case 18: { //SWAP FUNCTION

l1.swapRoster(l2);

assert(l1.howManyPlayers() == 2);

assert(l2.howManyPlayers() == 3);

assert(l1.lookupPlayer("joe", "distefano", x) && x == 1);

assert(l1.lookupPlayer("giovanna", "de cesaris", x) && x == 0);

assert(l2.lookupPlayer("angela", "li", x) && x == 2);

assert(l2.lookupPlayer("veronica", "czaja", x) && x == 0);

} break;

case 19: { //CHECKROSTER

BballRoster b1;

BballRoster b;

b1.signPlayer("angela", "li", 3); b1.signPlayer("veronica", "czaja", 1); b1.signPlayer("chloe", "janiser", 2); b1.signPlayer("izabella", "czaja", 0);

b1.dump();

checkRoster("\*","czaja",b1, b);

assert(b.howManyPlayers() == 2);

assert(b1.howManyPlayers() == 4);

string x, y;

SomeType z;

assert(b.choosePlayer(0, x, y, z) && z == 0);

assert(b.choosePlayer(1, x, y, z) && z == 1);

assert(!(b.choosePlayer(3, x, y, z)) && !(z == 2));

BballRoster a;

checkRoster("\*", "\*", b1, a);

assert(a.howManyPlayers() == 4);

assert(a.choosePlayer(0, x, y, z) && x == "izabella");

assert(a.choosePlayer(3, x, y, z) && x == "angela");

}break;

case 20: { //this code worked but ONLY when I moved my node to the public interface

//I did this because of how the testing code was written in HW1, I did move my node back

//to private

int oldn = 0;

{

BballRoster l;

recordaddrs = true;

oldn = addrs.size();

l.signPlayer("joe", "distefano", 82);

assert(l.howManyPlayers() == 1);

assert(addrs.size() == oldn + 1);

}

assert(addrs.size() == 0);

recordaddrs = false;

}

}

}