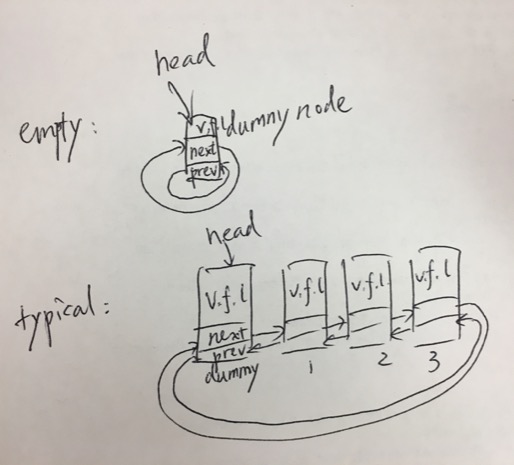
A description of the design of your implementation and why you chose it. (A couple of sentences will probably suffice, perhaps with a picture of a typical List and an empty List. Is your list circular? Does it have a dummy node? What's in your nodes?)

It is a circularized linked list that also contains a dummy node pointed by head. An empty list just has the dummy node pointed by head, and this dummy node’s next and prev pointers both point to the dummy node itself. 

A typical list is that based on empty list, more nodes were added to the circle after the dummy node.

The reason why I chose it is because I don’t need to consider special cases such as inserting nodes in the beginning or at the end. Also you don’t need to deal with special cases when there is no node in the list.

A brief description of notable obstacles you overcame.

1. Treat “” as the same as other non-empty strings.

It took me some time to figure out how to overlook the dummy node that contains the empty strings.

2. Comparing strings and find the right position to insert node

It took me some time to think thoroughly to find the position accurately.

3. Takes some time to find a trivial bug that I accidentally switched an assignment.

4. Figured out “psearch” function using the “get” function.

Pseudocode for non-trivial algorithms (e.g., PeopleList::remove and combine).

**PeopleList::add (const std::string& firstName, const std::string& lastName, const InfoType& value)**

{

Generate a pointer n that will go from the first node until the position where addition will occur, that is lastName <= n->next-m\_lastName using for loop;

if n->next== head, which means that n goes to the end of the list, insert the name after the node pointed by n.

else if lastName is less than n->next->m\_lastName, insert the name after the node pointed by n.

else if lastName equals to n->next->m\_lastName,

{

go through every node with the same last name using for loop { if firstName <= n->next->m\_firstName, break;}

if n->next->m\_lastName is not equal to lastName, which means n is pointing to the last Node of the same last name, insert the node after n.

if firstName is less than n->next->m\_firstName, insert the node after n.

else return false because we find a Node with the same first and last name.

}

return false

}

**bool PeopleList::change(const std::string &firstName, const std::string &lastName, const InfoType& value)**

{

Generate a pointer n that will go from the first node until the position where next node has the same last and first name and different value.

if n->next==head, which means n goes to the end and didn’t find the node, return false;

else n->next->m\_value=value and return true;

}

**bool PeopleList::addOrChange(std::string firstName, std::string lastName, const InfoType& value)**

{

Generate a pointer n that will go from the first node until the position where addition will occur, that is lastName <= n->next-m\_lastName using for loop;

if n->next== head, which means that n goes to the end of the list, insert the name after the node pointed by n.

else if lastName is less than n->next->m\_lastName, insert the name after the node pointed by n.

else is the case when lastName equals to n->next->m\_lastName,

{

go through every node with the same last name using for loop { if firstName <= n->next->m\_firstName, break;}

if n->next->m\_lastName is not equal to lastName, which means n is pointing to the last Node of the same last name, insert the node after n, and return true.

if firstName is less than n->next->m\_firstName, insert the node after n.

else is the case when the node with same last name and first name is found, change the value and return true.

}

}

**PeopleList::remove ()**

{

Initialize Node\* n to head and go through Nodes to search for the given first name and last name;

break the loop if n finds the Node with the given first and last names;

if n couldn’t find the Node with the given first and last names, return false;

else

Initialize a pointer pointing to the node that is to be removed.

Deal with the previous and next Nodes of the node to be removed.

delete the pointer and return true;

}

**bool PeopleList::get(int i, std::string& firstName, std::string& lastName, InfoType& value) const**

{

if i within range of size,

{

Generate a pointer starting from the head;

Initiate a counter equals to 0;

for ( n->next is not equal to head, n=n->next)

{

counter++ and if i==count-1, pass the first, last name and value of the Node pointed by n to the string and value and return true

}

else return false

}

**void PeopleList::swap(PeopleList& other)**

{

swap the head pointer between two objects

}

**PeopleList::~PeopleList()**

{

go through every node while head->next!=head,

{

Generate a pointer p that equals to head pointer. Head pointer goes to the next node, and carefully handle the pointers behind and after the node pointed by p, and delete p

}

finally there is one dummy node pointed by head. Delete head.

}

**PeopleList::PeopleList (const PeopleList& other)**

{

Generate a dummy node pointed by head. Its next and prev pointers are pointing the dummy node itself.

Generate a pointer p equals to head,

for ( a pointer n that points to every node in “other” list) insert a node with the same value of the node pointed by n into the current object, and p goes to the next position.

}

**PeopleList& PeopleList::operator = (const PeopleList&other)**

{

Use the copy constructor to generate a duplicate object of “other”.

use swap function to swap current object with the duplicate “other”.

return \*this

}

**void psearch(const std::string& fsearch, const std::string& lsearch, const PeopleList& p1, PeopleList& result)**

{

Generate a new object “new result” that will finally replace “result”.

if both fsearch and lsearch are “\*”, “new result” = p1

if fsearch is “\*”, but lsearch is not, go through every node of p1, get the information of each node via “get” function. if lsearch equals to the last name in the nodes of p1, “new result” is added this node via “add” function.

if lsearch is “\*”, but fsearch is not, similar logic as above, with the replacement of lsearch with fsearch.

if both fsearch and lsearch match, use the “loopup” function to get the value v. “new result” will be added this node via “add” function.

result = “new result”

}

**combine(m1, m2, result)**

{

Initialize a “new result” (different from “result”) that's the copy of m1;

Initialize an integer “event” to record the number of Nodes with same first and last names but different values in m1 and m2.

Implement a loop that checks every Node of m2

{

If m1 contains the full name of the Node of m2

{

check if the Nodes has the same value, if yes, continue, if not, “event”++ and remove this node from “new result”, and continue;

}

If m1 does not contain the full name of the Node of m2, add this node of m2 into “new result”;

}

use the copy assignment function to assign “result” with “new result”;

if “event” >0, return false, else return true;

}

The tests were performed on a map from strings to doubles.

#include <iostream>

#include "PeopleList.h"

#include <assert.h>

typedef double InfoType;

int main()

{

// default constructor

PeopleList m;

// For an empty list:

assert(m.size() == 0); // test size

assert(m.empty()); // test empty

assert(!m.remove("Ricky", "Ricardo")); // nothing to erase

m.add("Sijia","Zheng", 34.5);//Add Nodes with similar last names

m.add("Siyuan","Zhang", 45);//Add Nodes with similar last names

m.add("Huajun","Zhou", 33);// Add Nodes with similar last names

m.add("Yifu","Zheng", 35); // Add Nodes with similar last names

m.add("Huajun","Zhou",28);// It should not add since it is already in the list.

m.change("Huajun","Zhou",26);//It should change the value

assert (!m.add("Yifu","Zheng", 35)); //It should not add since it is already in the list, and it returns false.

m.addOrChange("Zhiyuan", "Zhang", 29); // Test addorchange

m.addOrChange("Zhiyuan", "Zhang", 35);// It changes the value.

assert(m.size()==5); //size should be 5 now.

m.remove("Zhiyuan", "Zhang"); //Test remove

assert(m.size()==4); //Test size again after remove

assert(m.contains("Yifu","Zheng")); //TEst contains.

assert(!m.contains("Zhiyuan","Zhang")); //TEst contains.

InfoType v=0;

assert(!m.lookup("Yifu","Zhang",v)); //Test lookup. Should return false

assert(m.lookup("Yifu","Zheng",v) && v==35);//Should return true

assert(!m.lookup("","",v)); //Shouldn't find empty string.

assert(!m.contains("","")); //Shouldn't find empty string.

assert(!m.change("","", 45)); //Shouldn't change anything since it does not contain empty string.

assert(m.addOrChange("","", 45)); //Add an empty string

assert(m.size()==5); //Size should be 5 now.

assert(m.addOrChange("","", 440)); //Should change the value

std::string f;

std::string l;

assert(m.get(0,f,l,v) && f=="" && l=="" && v==440); //Test "get".

assert(!m.get(5,f,l,v)); //Test "get". Returns false since out of range

assert(v==440);//v still equals to 440

PeopleList n;

n.add("Jena","Cleveland", 35);

m.swap(n);//Test swap

assert(m.size()==1 && !m.empty()); //Test swapped m size.

m.swap(n);//Swap back.

PeopleList o(n); //test copy constructor

PeopleList p=n;//test copy constructor

o=m;//Test copy assignment. o should change to m.

o=n; //Test copy assignment. o should change to n.

o=n=m; //Test copy assignment. o and n should equal to m.

combine (m,n,o); //TEst combine. Since m=n, o should return as m or n.

combine (m,p,o);//test combine. Now o increases, since m and p are different.

p.add("Sijia","Zheng", 550); // Add a node into p with same name but different value;

assert(!combine(m,p,o));// "combine" should return false since there is a conflict, also "Sijia""Zheng" will be removed.

m.add("Sijia","Zhang",55);

search("Sijia","\*", m, o);// should return two people with first name "Sijia".

search("\*","Zhang",m,o); // Shoud return two people with last name "zhang"

search ("\*","\*",m,o); // o should return m

search ("Sijia","Zheng",m,o); //should return "Sijia""Zheng"

search ("Siyuan","Zheng",m,o);//Should return nothing

for (int n = 0; n < o.size(); n++)

{

std::string f;

std::string l;

InfoType v;

o.get(n, f, l, v);

std::cout << f << " " << l << " " << v << std::endl;

}

return 0;

}

#include <iostream>

#include "PeopleList.h"

#include <assert.h>

typedef std::string InfoType;

int main()

{

PeopleList m;

// For an empty list:

assert(m.size() == 0); // test size

assert(m.empty()); // test empty

assert(!m.remove("Ricky", "Ricardo")); // nothing to erase

m.add("Sijia","Zheng","34.5"); //Add Nodes with similar last names

m.add("Siyuan","Zhang", "45");//Add Nodes with similar last names

m.add("Huajun","Zhou", "33");// Add Nodes with similar last names

m.add("Yifu","Zheng", "35"); // Add Nodes with similar last names

m.add("Huajun","Zhou","28");// It should not add since it is already in the list.

m.change("Huajun","Zhou","26");//It should change the value

assert (!m.add("Yifu","Zheng", "35")); //It should not add since it is already in the list, and it returns false.

m.addOrChange("Zhiyuan", "Zhang", "29"); // Test addorchange

m.addOrChange("Zhiyuan", "Zhang", "35");// It changes the value.

assert(m.size()==5); //size should be 5 now.

m.remove("Zhiyuan", "Zhang"); //Test remove

assert(m.size()==4); //Test size again after remove

assert(m.contains("Yifu","Zheng")); //TEst contains.

assert(!m.contains("Zhiyuan","Zhang")); //TEst contains.

InfoType v="0";

assert(!m.lookup("Yifu","Zhang",v)); //Test lookup. Should return false

assert(m.lookup("Yifu","Zheng",v) && v=="35");//Should return true

assert(!m.lookup("","",v)); //Shouldn't find empty string.

assert(!m.contains("","")); //Shouldn't find empty string.

assert(!m.change("","", "45")); //Shouldn't change anything since it does not contain empty string.

assert(m.addOrChange("","", "45")); //Add an empty string

assert(m.size()==5); //Size should be 5 now.

assert(m.addOrChange("","", "440")); //Should change the value

std::string f;

std::string l;

assert(m.get(0,f,l,v) && f=="" && l=="" && v=="440"); //Test "get".

assert(!m.get(5,f,l,v)); //Test "get". Returns false since out of range

assert(v=="440");//v still equals to 440

PeopleList n;

n.add("Jena","Cleveland", "35");

m.swap(n);//Test swap

assert(m.size()==1 && !m.empty()); //Test swapped m size.

m.swap(n);//Swap back.

PeopleList o(n); //test copy constructor

PeopleList p=n;//test copy constructor

o=m;//Test copy assignment. o should change to m.

o=n; //Test copy assignment. o should change to n.

o=n=m; //Test copy assignment. o and n should equal to m.

combine (m,n,o); //TEst combine. Since m=n, o should return as m or n.

combine (m,p,o);//test combine. Now o increases, since m and p are different.

p.add("Sijia","Zheng", "550"); // Add a node into p with same name but different value;

assert(!combine(m,p,o));// "combine" should return false since there is a conflict, also "Sijia""Zheng" will be removed.

m.add("Sijia","Zhang","55");

search("Sijia","\*", m, o);// should return two people with first name "Sijia".

search("\*","Zhang",m,o); // Shoud return two people with last name "zhang"

search ("\*","\*",m,o); // o should return m

search ("Sijia","Zheng",m,o); //should return "Sijia""Zheng"

//search ("Siyuan","Zheng",m,o);//Should return nothing

for (int n = 0; n < o.size(); n++)

{

std::string f;

std::string l;

InfoType v;

o.get(n, f, l, v);

std::cout << f << " " << l << " " << v << std::endl;

}

return 0;

}