ENSF 614 – Fall 2024

**Advanced Systems Analysis and Software Design**

**University of Calgary**

**Lab Assignment 1**

Student Name: Yael Gonzalez

Instructor: M. Moussavi, PhD, Peng

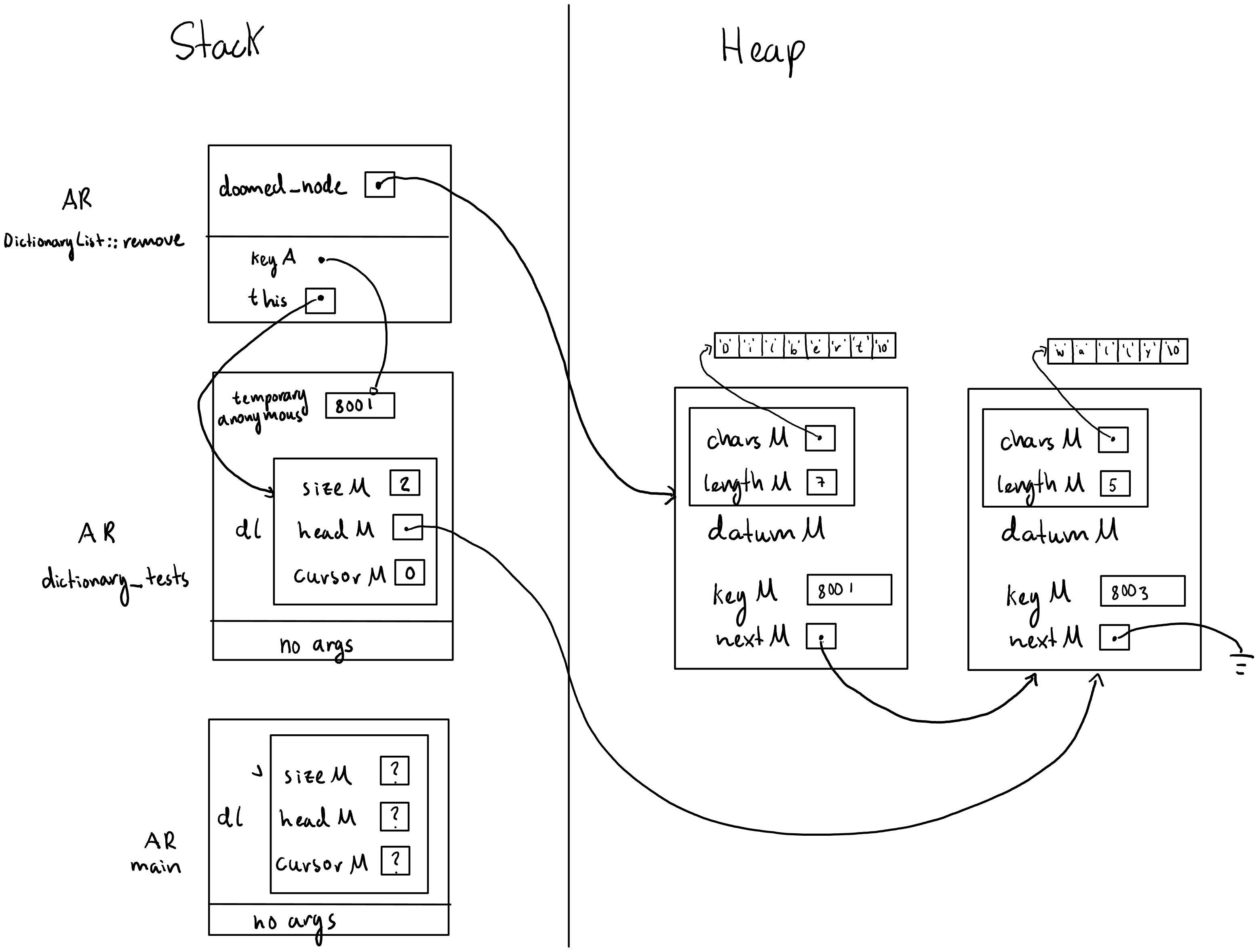
Submission Date: September 18, 2024

# Exercise A

|  |  |
| --- | --- |
| **Program output and its order** | **Your explanation (why and where is the cause for this output)** |
| **constructor with int argument is called.** | It is called at line 16 in exAmain. The statement, Mystring c = 3 is interpreted by the compiler as a call to the constructor Mystring::Mystring(int n) |
| **default constructor is called.**  **default constructor is called.** | It is called at line 22 in exAmain. The statement, Mystring x[2] is interpreted by the compiler as two calls to the default constructor Mystring::Mystring() |
| **constructor with char\* argument is called.** | It is called at line 26 in exAmain. The statement, Mystring \*z = new Mystring(“4”) is interpreted by the compiler as a call to the constructor Mystring(const char \*s) |
| **copy constructor is called.**  **copy constructor is called.** | It is called at line 28 in exAmain. The statement, x[0].append(\*z).append(x[1]) is interpreted by the compiler as two calls to the copy constructor Mystring(const Mystring &source), the first copying the Mystring object z is pointing to, and the second copying the Mystring object the second element of the pointer array x is pointing to. |
| **destructor is called.**  **destructor is called.** | It is called at line 28 in exAmain. In the statement x[0].append(\*z).append(x[1]) for the method Mystring &append(const Mystring other) the Mystring object is passed by value, which creates an extra copy (other) on each call. The compiler is called twice to delete those extra copies. |
| **copy constructor is called.** | It is called at line 30 in exAmain. The statement Mystring mars = x[0] is interpreted as a call to the copy constructor Mystring(const Mystring &source), copying in mars over the Mystring object the first element of x points to. |
| **assignment operator called.** | It is called at line 32 in exAmain. The statement x[1] = x[0] is interpreted by the compiler as a call to the assignment operator Mystring &operator=(const Mystring &rhs). |
| **constructor with char\* argument is called.**  **constructor with char\* argument is called.** | It is called at lines 34 and 36 in exAmain. The statements Mystring jupiter(“White”) and arr[0] = new Mystring(“Yellow”) are interpreted by the compiler as calls to the constructor Mystring(const char \*s). |
| **destructor is called.**  **destructor is called.**  **destructor is called.**  **destructor is called.**  **destructor is called.** | It is called 4 times when // BLOCK ENDS HERE is reached. The compiler interprets that the block of code is done so the destructor  ~Mystring() is automatically called to delete the dynamically allocated charsM of the Mystring objects that are now considered out of scope, i.e. jupiter, mars, x[1], and x[0] (in that order). The last call happens at line 41, the destructor is called for delete ar[0]. |
| **constructor with char\* argument is called.** | It is called at line 43. The compiler interprets Mystring d = "Green" as a call to the Mystring(const char \*s) constructor |
| **Program terminated successfully.** | It is called at line 45. When we reach cout << "\nProgram terminated successfully." << endl |
| **destructor is called.**  **destructor is called** | It is called at line 47, at the end of our program. The destructor is called twice to remove the dynamically allocated charsM of the remaining Mystring objects, i.e., d and c (in that order). |

# Exercise B

## Part 1



## Part 2

/\*\*

 \*  File Name: dictionaryList.cpp

 \*  Assignment: ENSF 614 Fall 2024 - Lab 1 Exercise B

 \*  Created by: Mahmood Moussavi

 \*  Completed by: Yael Gonzalez

 \*  Submission Date: September 18, 2024

 \*/

#include <assert.h>

#include <iostream>

#include <stdlib.h>

#include "dictionaryList.h"

#include "mystring\_B.h"

using namespace std;

*Node*::Node(const *Key* &*keyA*, const *Datum* &*datumA*, *Node* \**nextA*)

    : keyM(*keyA*), datumM(*datumA*), nextM(*nextA*)

{

}

*DictionaryList*::DictionaryList()

    : sizeM(0), headM(0), cursorM(0)

{

}

*DictionaryList*::DictionaryList(const *DictionaryList* &*source*)

{

  copy(*source*);

}

*DictionaryList* &*DictionaryList*::operator=(const *DictionaryList* &*rhs*)

{

  if (*this* != &*rhs*)

  {

    destroy();

    copy(*rhs*);

  }

  return \**this*;

}

*DictionaryList*::~DictionaryList()

{

  destroy();

}

int *DictionaryList*::size() const

{

  return sizeM;

}

int *DictionaryList*::cursor\_ok() const

{

  return cursorM != 0;

}

const *Key* &*DictionaryList*::cursor\_key() const

{

  assert(cursor\_ok());

  return cursorM->keyM;

}

const *Datum* &*DictionaryList*::cursor\_datum() const

{

  assert(cursor\_ok());

  return cursorM->datumM;

}

void *DictionaryList*::insert(const int &*keyA*, const *Mystring* &*datumA*)

{

  // Add new node at head?

  if (headM == 0 || *keyA* < headM->keyM)

  {

    headM = **new** *Node*(*keyA*, *datumA*, headM);

    sizeM++;

  }

  // Overwrite datum at head?

  else if (*keyA* == headM->keyM)

    headM->datumM = *datumA*;

  // Have to search ...

  else

  {

    // POINT ONE

    // if key is found in list, just overwrite data;

    for (*Node* \*p = headM; p != 0; p = p->nextM)

    {

      if (*keyA* == p->keyM)

      {

        p->datumM = *datumA*;

        return;

      }

    }

    // OK, find place to insert new node ...

*Node* \*p = headM->nextM;

*Node* \*prev = headM;

    while (p != 0 && *keyA* > p->keyM)

    {

      prev = p;

      p = p->nextM;

    }

    prev->nextM = **new** *Node*(*keyA*, *datumA*, p);

    sizeM++;

  }

  cursorM = NULL;

}

void *DictionaryList*::remove(const int &*keyA*)

{

  if (headM == 0 || *keyA* < headM->keyM)

    return;

*Node* \*doomed\_node = 0;

  if (*keyA* == headM->keyM)

  {

    doomed\_node = headM;

    headM = headM->nextM;

    // POINT TWO

  }

  else

  {

*Node* \*before = headM;

*Node* \*maybe\_doomed = headM->nextM;

    while (maybe\_doomed != 0 && *keyA* > maybe\_doomed->keyM)

    {

      before = maybe\_doomed;

      maybe\_doomed = maybe\_doomed->nextM;

    }

    if (maybe\_doomed != 0 && maybe\_doomed->keyM == *keyA*)

    {

      doomed\_node = maybe\_doomed;

      before->nextM = maybe\_doomed->nextM;

    }

  }

  if (doomed\_node == cursorM)

    cursorM = 0;

**delete** doomed\_node; // Does nothing if doomed\_node == 0.

  sizeM--;

}

void *DictionaryList*::go\_to\_first()

{

  cursorM = headM;

}

void *DictionaryList*::step\_fwd()

{

  assert(cursor\_ok());

  cursorM = cursorM->nextM;

}

void *DictionaryList*::make\_empty()

{

  destroy();

  sizeM = 0;

  cursorM = 0;

}

// The following function are supposed to be completed by the stuents, as part

// of the exercise B part II. the given fucntion are in fact place-holders for

// find, destroy and copy, in order to allow successful linking when you're

// testing insert and remove. Replace them with the definitions that work.

void *DictionaryList*::find(const *Key* &*keyA*)

{

  for (*Node* \*curr = headM; !curr; curr = curr->nextM)

  {

    if (curr->keyM == *keyA*)

    {

      cursorM = curr;

      return;

    }

  }

  cursorM = 0; // if not found put cursor in off-list state

}

void *DictionaryList*::destroy()

{

  while (headM)

  {

*Node* \*temp = headM;

    headM = temp->nextM;

**delete** temp;

  }

  headM = 0;

  cursorM = 0;

  sizeM = 0;

}

void *DictionaryList*::copy(const *DictionaryList* &*source*)

{

  // If empty, initialize the \*this with default values

  if (!*source*.headM)

  {

    headM = 0;

    sizeM = 0;

    cursorM = 0;

    return;

  }

  // Create a new head for the \*this using source's head data

  headM = **new** *Node*(*source*.headM->keyM, *source*.headM->datumM, 0);

  // Initialize pointers to traverse source and \*this lists

*Node* \*srcNode = *source*.headM->nextM;

*Node* \*thisNode = headM;

  // Copy the rest of the nodes from the source list to \*this

  while (srcNode)

  {

    thisNode->nextM = **new** *Node*(srcNode->keyM, srcNode->datumM, 0);

    thisNode = thisNode->nextM;

    srcNode = srcNode->nextM;

  }

  // Copy the size of the source list to \*this

  sizeM = *source*.sizeM;

  // Make cursor of this \*this point to the twin of whatever the source's cursor points to

  if (*source*.cursorM)

  {

*Node* \*srcCursor = *source*.headM;

*Node* \*destCursor = headM;

    while (srcCursor != *source*.cursorM)

    {

      srcCursor = srcCursor->nextM;

      destCursor = destCursor->nextM;

    }

    cursorM = destCursor;

  }

  else

  {

    cursorM = 0;

  }

}

# Exercise C

#include <string>

#include <vector>

using namespace std;

enum class *EmployeeState*

{

    Active,

    Suspended,

    Retired,

    Fired

};

class *Company*

{

private:

*string* name;

*Address* address;

*Date* dateEstablished;

    vector<*Employee*> employees;

    vector<*Customer*> customers;

public:

    // getters, setters and more...

};

class *Employee*

{

private:

*Name* name;

*Address* address;

*Date* birthday;

*EmployeeState* state;

public:

    // getters, setters and more...

};

class *Customer*

{

private:

*Name* name;

*Address* address;

*string* phone;

public:

    // getters, setters and more...

};

class *Name*

{

private:

*string* firstName;

*string* lastName;

public:

    // getters, setters and more...

};

class *Address*

{

private:

*string* street;

*string* city;

*string* postalCode;

*string* country;

public:

    // getters, setters and more...

};

class *Date*

{

private:

    int day;

    int month;

    int year;

public:

    // getters, setters and more...

};

# Exercise D

## point.h

## human.h

## human.cpp

## main.cpp