Name: Shamin Rahman

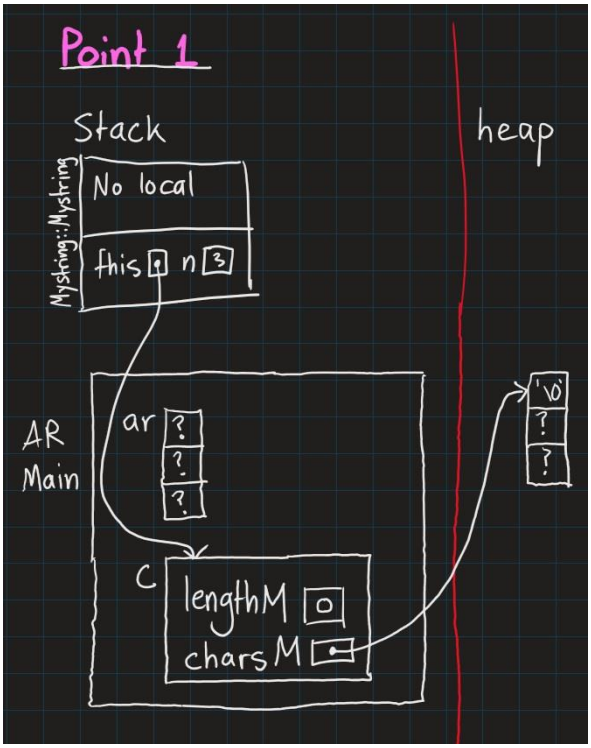
Course Name: Principles of Software Design

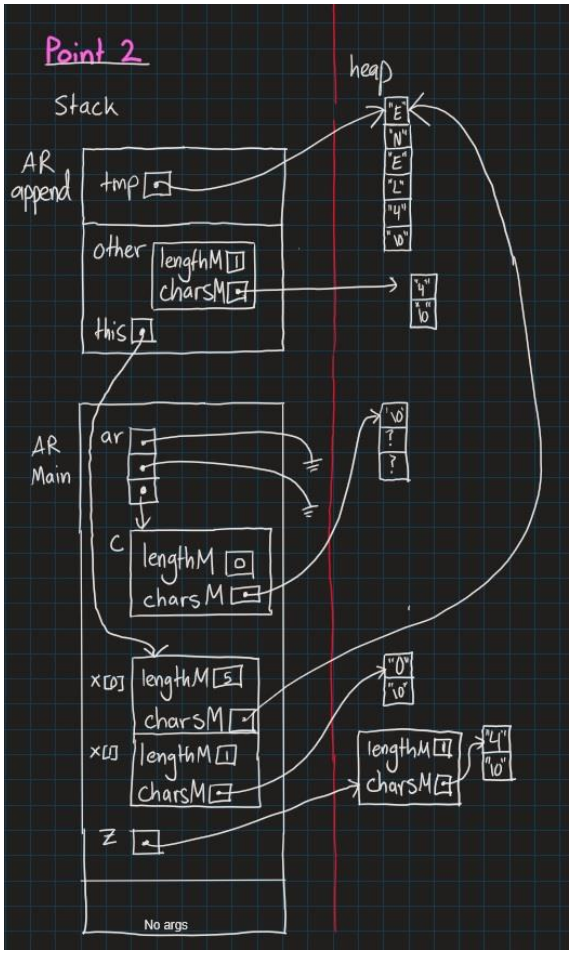
Course Code:ENSF 480

Assignment Number: For example,Lab-1

Submission Date and Time: 27/09/2019

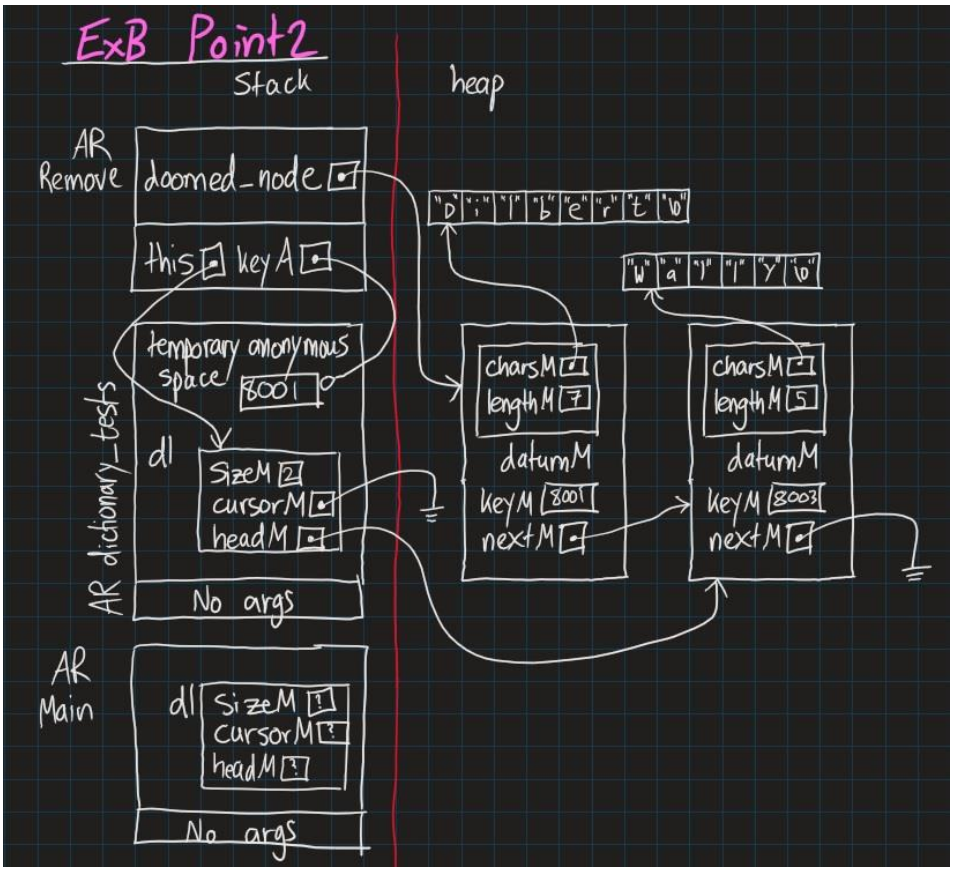
Ex A


Point 1
Stack
heap
Mystring::Mystring
No local
this
n
3
AR
Main
ar
?
?
?
C
lengthM
charsM
'\0'
?
?


Point 2
Stack
heap
AR
append
tmp
other
lengthM
charsM
this
'E'
'N'
'E'
'L'
'4'
'\0'
'4'
'\0'
AR
Main
ar
C
lengthM
charsM
'\0'
?
?
x[0]
lengthM
charsM
x[1]
lengthM
charsM
z
'0'
'\0'
lengthM
charsM
'4'
'\0'
No args

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 12 in exAmain when:Mystring c = 3. Constructor Mystring::Mystring(int n)is called
default constructor is called. **default constructor is called.**	Called at line18 of main when array x is declared and Mystring::Mystring() is called.
constructor with char* argument is called.	Called at line 23 of main when pointer z is initialized with “4”, calling Mystring::Mystring(const char *s)
copy constructor is called. **copy constructor is called.**	Called when the append functions are called, because they pass Mystring objects by value for the argument, meaning that a copy of the object needs to be made implicitly
destructor is called. **destructor is called.**	The two argument objects that were passed by value into the append functions are deleted when the append function has ended and out of scope
copy constructor is called.	Called when object mars is initialized with x[0]
assignment operator called.	Called when x[1] is assigned as x[0]
constructor with char* argument is called. **constructor with char* argument is called.**	These are called when variables Jupiter is initialized with “white” and ar[0] is assigned “yellow”, which calls Mystring::Mystring(const char *s)
destructor is called. **destructor is called.** **destructor is called.** **destructor is called.** **destructor is called.**	These are called when the following objects of Mystring are deleted when they become out of scope because the block of code ends: x[0], x[1], *z, mars, jupiter
constructor with char* argument is called.	Called when variable d is initialized with “Green”
Program terminated successfully.	Program Reaches line 41, cout
destructor is called. **destructor is called**	Return from line 42, all remaining destructors are called.

Ex B



Ex B outputs

```
Let's look up some names ...
  name for 8001 is: Allen.
  Sorry, I couldn't find 8000 in the list.
  name for 8002 is: Peter.
  name for 8004 is: PointyHair.
***----Finished tests of finding -------------------------***
```

```
Printing list--keys should be 315, 319
  315  Shocks
  319  Randomness
Printing list--keys should be 315, 319, 335
  315  Shocks
  319  Randomness
  335  ParseErrors
Printing list--keys should be 315, 335
  315  Shocks
  335  ParseErrors
Printing list--keys should be 319, 335
  319  Randomness
  335  ParseErrors
Printing list--keys should be 315, 319, 335
  315  Shocks
  319  Randomness
  335  ParseErrors
***----Finished tests of copying---------------------***
```

Ex C outputs

```
PS C:\Users\shami\Desktop\480\labs\lab1> g++ -Wall exBmain.cpp dictionaryList.cpp mystring_B.cpp
PS C:\Users\shami\Desktop\480\labs\lab1> .\a.exe

Printing list just after its creation ...
  List is EMPTY.

Printing list after inserting 3 new keys ...
  8001  Dilbert
  8002  Alice
  8003  Wally

Printing list after removing two keys and inserting PointyHair ...
  8003  Wally
  8004  PointyHair

Printing list after changing data for one of the keys ...
  8003  Sam
  8004  PointyHair

Printing list after inserting 2 more keys ...
  8001  Allen
  8002  Peter
  8003  Sam
  8004  PointyHair
***----Finished dictionary tests---------------------------***
```

```
Testig a few comparison and insertion operators.

Peter is greater than or equal Allen
Allen is less than Peter
Peter is not equal to Allen
Peter is greater than Allen
Peter is not less than Allen
Peter is not equal to Allen

Using square bracket [] to access elements of Mystring objects.
The socond element of Peter is: e
The socond element of Poter is: o

Using << to display key/datum pairs in a Dictionary list:
8001 Allen
8002 Peter
8003 Sam
8004 PointyHair


Using [] to display the datum only:
Allen
Peter
Sam
PointyHair


Using [] to display sequence of charaters in a datum:
A
l
l
e
n


***----Finished tests for overloading operators ----------***
```

Code:

```
/* File: mystring_B.h
 *
 *
 */
// ENSF 480 - Lab 1 - Exercise B and C
```

```
#include <iostream>
#include <string>
using namespace std;

#ifndef MYSTRING_H
#define MYSTRING_H

class Mystring {

public:
 //overloaded operators:
 bool operator >=(const Mystring& rhs)const;
 bool operator <=(const Mystring& rhs)const;
 bool operator <(const Mystring& rhs)const;
 bool operator >(const Mystring& rhs)const;
 bool operator !=(const Mystring& rhs)const;
 bool operator ==(const Mystring& rhs)const;

 char& operator [](int index)const;
 // char& operator [](int index);

 friend ostream& operator << (ostream &out, const Mystring &c);

 //end of overloaded operators

 Mystring();
 // PROMISES: Empty string object is created.

 Mystring(int n);
 // PROMISES: Creates an empty string with a total capacity of n.
 //         In other words, dynamically allocates n elements for
 //         charsM,sets the lengthM to zero, and fills the first
 //         element of charsM with '\0'.

 Mystring(const char *s);
 // REQUIRES: s points to first char of a built-in string.
 // REQUIRES: Mystring object is created by copying chars from s.

 ~Mystring(); // destructor

 Mystring(const Mystring& source); // copy constructor

 Mystring& operator =(const Mystring& rhs); // assignment operator
 // REQUIRES: rhs is reference to a Mystring as a source
 // PROMISES: to make this-object (object that this is pointing to, as  a copy
 //         of rhs.

 int length() const;
 // PROMISES: Return value is number of chars in charsM.

 char get_char(int pos) const;
 // REQUIRES: pos >= 0 && pos < length()
 // PROMISES:
```

```
 // Return value is char at position pos.
 // (The first char in the charsM is at position 0.)

 const char * c_str() const;
 // PROMISES:
 //   Return value points to first char in built-in string
 //   containing the chars of the string object.

 void set_char(int pos, char c);
 // REQUIRES: pos >= 0 && pos < length(), c != '\0'
 // PROMISES: Character at position pos is set equal to c.

 Mystring& append(const Mystring& other);

 // PROMISES: extends the size of charsM to allow concatenate other.charsM to
 //         to the end of charsM. For example if charsM points to "ABC", and
 //        other.charsM points to XYZ, extends charsM to "ABCXYZ".
 //

 void set_str(char* s);
 // REQUIRES: s is a valid C++ string of characters (a built-in string)
 // PROMISES:copys s into charsM, if the length of s is less than or equal lengthM.
 //        Othrewise, extends the size of the charsM to s.lengthM+1, and copies
 //        s into the charsM.

 int isGreaterThan( const Mystring& s)const;
 // REQUIRES: s refers to an object of class Mystring
 // PROMISES: retruns true if charsM is greater than s.charsM.

 int isLessThan (const Mystring& s)const;
 // REQUIRES: s refers to an object of class Mystring
 // PROMISES: retruns true if charsM is less than s.charsM.

 int isEqual (const Mystring& s)const;
 // REQUIRES: s refers to an object of class Mystring
 // PROMISES: retruns true if charsM equal s.charsM.

 int isNotEqual(const Mystring& s)const;
 // REQUIRES: s refers to an object of class Mystring
 // PROMISES: retruns true if charsM is not equal s.charsM.

private:

 int lengthM; // the string length - number of characters excluding \0
 char* charsM; // a pointer to the beginning of an array of characters, allocated dynamically.
 void memory_check(char* s);
 // PROMISES: if s points to NULL terminates the program.
};
#endif
```

```
/*  mystring_B.cpp
 *
```

```
 *
 */
// ENSF 480 - Lab 1 - Exercise B and C
#include "mystring_B.h"
#include <string.h>
#include <iostream>
using namespace std;

//overloaded operators
bool Mystring::operator>=(const Mystring& rhs)const{
   if(isGreaterThan(rhs) || isEqual(rhs)){
      return true;
   }
   return false;
}
bool Mystring::operator>(const Mystring& rhs)const{
   if(isGreaterThan(rhs)){
      return true;
   }
   return false;
}
bool Mystring::operator<=(const Mystring& rhs)const{
   if(isLessThan(rhs) || isEqual(rhs)){
      return true;
   }
   return false;
}
bool Mystring::operator<(const Mystring& rhs)const{
   if(isLessThan(rhs)){
      return true;
   }
   return false;
}
bool Mystring::operator ==(const Mystring& rhs)const{
   if(isEqual(rhs))
      return true;
   return false;
}
bool Mystring::operator !=(const Mystring& rhs)const{
   if(isNotEqual(rhs))
      return true;
   return false;
}
char& Mystring::operator [](int index)const{
   return charsM[index];
}
// char& Mystring::operator[](int index){
//  return charsM[index];
// }

ostream& operator << (ostream &out, const Mystring &c){
   out << c.charsM;
   return out;
```

```
}
//end of overloaded operators

Mystring::Mystring()
{
   charsM = new char[1];

   // make sure memory is allocated.
   memory_check(charsM);
   charsM[0] = '\0';
   lengthM = 0;
}

Mystring::Mystring(const char *s)
   : lengthM(strlen(s))
{
   charsM = new char[lengthM + 1];

 // make sure memory is allocated.
   memory_check(charsM);

   strcpy(charsM, s);
}

Mystring::Mystring(int n)
   : lengthM(0), charsM(new char[n])
{
   // make sure memory is allocated.
   memory_check(charsM);
   charsM[0] = '\0';
}

Mystring::Mystring(const Mystring& source):
   lengthM(source.lengthM), charsM(new char[source.lengthM+1])
{
   memory_check(charsM);
   strcpy (charsM, source.charsM);
}

Mystring::~Mystring()
{
   delete [] charsM;
}

int Mystring::length() const
{
   return lengthM;
}

char Mystring::get_char(int pos) const
{
   if(pos < 0 && pos >= length()){
      cerr << "\nERROR: get_char: the position is out of boundary." ;
```

```
    }

    return charsM[pos];
}

const char * Mystring::c_str() const
{
    return charsM;
}

void Mystring::set_char(int pos, char c)
{
    if(pos < 0 && pos >= length()){
        cerr << "\nset_char: the position is out of boundary."
     << " Nothing was changed.";
        return;
    }

    if (c != '\0'){
        cerr << "\nset_char: char c is empty."
     << " Nothing was changed.";
        return;
    }

    charsM[pos] = c;
}

Mystring& Mystring::operator =(const Mystring& S)
{
    if(this == &S)
        return *this;
    delete [] charsM;
    lengthM = (int)strlen(S.charsM);
    charsM = new char [lengthM+1];
    memory_check(charsM);
    strcpy(charsM,S.charsM);

    return *this;
}

Mystring& Mystring::append(const Mystring& other)
{
    char *tmp = new char [lengthM + other.lengthM + 1];
    memory_check(tmp);
    lengthM+=other.lengthM;
    strcpy(tmp, charsM);
    strcat(tmp, other.charsM);
    delete []charsM;
    charsM = tmp;

    return *this;
}
```

```
 void Mystring::set_str(char* s)
{
      delete []charsM;
      lengthM = (int)strlen(s);
      charsM=new char[lengthM+1];
      memory_check(charsM);

      strcpy(charsM, s);
}

int Mystring::isNotEqual (const Mystring& s)const
{
   return (strcmp(charsM, s.charsM)!= 0);
}

int Mystring::isEqual (const Mystring& s)const
{
   return (strcmp(charsM, s.charsM)== 0);
}

int Mystring::isGreaterThan (const Mystring& s)const
{
   return (strcmp(charsM, s.charsM)> 0);
}

int Mystring::isLessThan (const Mystring& s)const
{
   return (strcmp(charsM, s.charsM)< 0);
}

void Mystring::memory_check(char* s)
{
   if(s == 0)
      {
         cerr <<"Memory not available.";
         exit(1);
      }
}
```

```
// dictionaryList.h
// ENSF 480 - Lab 1 - Exercise B & C

#ifndef DICTIONARY_H
#define DICTIONARY_H
#include <iostream>
using namespace std;

// class DictionaryList: GENERAL CONCEPTS
//
//   key/datum pairs are ordered.  The first pair is the pair with
//   the lowest key, the second pair is the pair with the second
//   lowest key, and so on.  This implies that you must be able to
```

```
//   compare two keys with the < operator.
//
//   Each DictionaryList object has a "cursor" that is either attached
//   to a particular key/datum pair or is in an "off-list" state, not
//   attached to any key/datum pair.  If a DictionaryList is empty, the
//   cursor is automatically in the "off-list" state.

#include "mystring_B.h"

// Edit these typedefs to change the key or datum types, if necessary.
typedef int Key;
typedef Mystring Datum;

class DictionaryList;

// THE NODE TYPE
//   In this exercise the node type is a class, that has a ctor.
//   Data members of Node are private, and class DictionaryList
//   is declared as a friend. For details on the friend keyword refer to your
//   lecture notes.

class Node {
  friend class DictionaryList;
  friend ostream& operator << (ostream &out, const DictionaryList &c);
private:
  Key keyM;
  Datum datumM;
  Node *nextM;

  // This ctor should be convenient in insert and copy operations.
  Node(const Key& keyA, const Datum& datumA, Node *nextA);
};

class DictionaryList {
public:
  DictionaryList();
  DictionaryList(const DictionaryList& source);
  DictionaryList& operator =(const DictionaryList& rhs);

  friend ostream& operator << (ostream &out, const DictionaryList &c);
  Datum& operator [](int index)const;

  ~DictionaryList();

  int size() const;
  // PROMISES: Returns number of keys in the table.

  int cursor_ok() const;
  // PROMISES:
  //   Returns 1 if the cursor is attached to a key/datum pair,
  //   and 0 if the cursor is in the off-list state.

  const Key& cursor_key() const;
```

```
  // REQUIRES: cursor_ok()
  // PROMISES: Returns key of key/datum pair to which cursor is attached.

  const Datum& cursor_datum() const;
  // REQUIRES: cursor_ok()
  // PROMISES: Returns datum of key/datum pair to which cursor is attached.

  void insert(const Key& keyA, const Datum& datumA);
  // PROMISES:
  //   If keyA matches a key in the table, the datum for that
  //   key is set equal to datumA.
  //   If keyA does not match an existing key, keyA and datumM are
  //   used to create a new key/datum pair in the table.
  //   In either case, the cursor goes to the off-list state.

  void remove(const Key& keyA);
  // PROMISES:
  //   If keyA matches a key in the table, the corresponding
  //   key/datum pair is removed from the table.
  //   If keyA does not match an existing key, the table is unchanged.
  //   In either case, the cursor goes to the off-list state.

  void find(const Key& keyA);
  // PROMISES:
  //   If keyA matches a key in the table, the cursor is attached
  //   to the corresponding key/datum pair.
  //   If keyA does not match an existing key, the cursor is put in
  //   the off-list state.

  void go_to_first();
  // PROMISES: If size() > 0, cursor is moved to the first key/datum pair
  //   in the table.

  void step_fwd();
  // REQUIRES: cursor_ok()
  // PROMISES:
  //   If cursor is at the last key/datum pair in the list, cursor
  //   goes to the off-list state.
  //   Otherwise the cursor moves forward from one pair to the next.

  void make_empty();
  // PROMISES: size() == 0.

private:
  int sizeM;
  Node *headM;
  Node *cursorM;

  void destroy();
  // Deallocate all nodes, set headM to zero.

  void copy(const DictionaryList& source);
  // Establishes *this as a copy of source.  Cursor of *this will
```

```
  // point to the twin of whatever the source's cursor points to.

};

#endif
```

```
// lookuptable.cpp

// ENSF 480 - Lab 1 - Exercise B & C

// Completed by:

#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){
   if(headM == 0){
      cursorM = 0;
      return;
   }

   else if(headM->keyM == keyA){
      cursorM = headM;
      return;
   }

   else{
      Node *before = headM;
      Node *after = headM->nextM;

      while(after != 0 && keyA >= before->keyM){

         if(before->keyM == keyA){
            cursorM = before;
            return;
         }

         before = after;
         after = before->nextM;
      }

      if(before->keyM == keyA){
         cursorM = before;
         return;
      }
   }

   cursorM = 0;
   return;
}

void DictionaryList::destroy(){
```

```
   if(headM == 0){
      return;
   }

   cursorM = 0;
   sizeM = 0;

   Node* before = headM;
   Node* after = before->nextM;

   do{

      delete before;

      before = after;
      after = before->nextM;
   }while(after != 0);

   delete before;
   headM = 0;
}
/* void DictionaryList::destroy1(){

} */

void DictionaryList::copy(const DictionaryList& source){
   if (source.headM == 0){
      this->headM = 0;
      this->cursorM = 0;
      this->sizeM = 0;
      return;
   }

   this->cursorM = 0;
   this->sizeM = 0;
   Node* before = source.headM;
   Node* after = before->nextM;

   Node* copy =  new Node(before->keyM, before->datumM, 0);
   Node* head = copy;
   this->sizeM++;

   while(after != 0){

      copy->nextM = new Node(after->keyM, after->datumM, 0);

      if(source.cursorM == before){
         this->cursorM = copy;
      }

      copy = copy->nextM;
      this->sizeM++;
```

```
      before = after;
      after = before -> nextM;
   }

   copy->nextM = after;    //after should == 0
   this->headM = head;
}

ostream& operator << (ostream &out, const DictionaryList &c){
   Node* p = c.headM;
   Node* after = p->nextM;

   while(after != 0){
      out << p->keyM<<" "<<p->datumM <<endl;
      p = after;
      after = p->nextM;
   }

   out << p->keyM<<" "<<p->datumM <<endl;
   return out;
}

Datum& DictionaryList::operator [](int index)const{
   if(index > size()){
      cerr<<endl<<"index is too high"<<endl;
   }
   Node* p = headM;

   while(index > 0){
      p = p->nextM;
      index--;
   }

   return p->datumM;

}
```