# A1/CIS*4150

Unit Tests and Control Flow Graphs

courselink.uoguelph.ca

A. Hamilton-Wright

## Overview

This assignment will exercise your knowledge of control flow graphs and unit tests. Note that this is an individual assignment. All of the submitted work should be your own, and based on the starting code and course examples. Any work you have included from elsewhere (*e.g.*, the examples) must be noted and cited.

## Skills

control flow (6/6)

graph structures (3/6)

testing tools (3/6)

unit testing (6/6)

(*)[The skill scale is from 0 (Fundamental Awareness) to 6 (Main Focus).]

### Image description

Checklist. Photo source pixabay.com, Pixabay license.

## CIS 4150/F18: Assignment One

For this pen and paper assignment, there are a mixture of problems from the the book, as well as specific posed problems, below.

The due date and time for this assignment (and all others) are listed in the syllabus.

## Questions from the Book

Please provide answers to each of the following questions from the textbook. When writing answers, you should take care to use complete sentences, check spelling, and proofread your results. Poor spelling, grammar and overall style will affect your mark.

Be sure to give clear, concise and complete answers.

### Chapter 1

Please answer question 1.7.

### Chapter 2

Please answer questions 2.1, 2.3, and 2.5.

### Chapter 3

Please answer questions 3.1

## Control Flow Graph problems

Consider the `LLNode` code at the end of this assignment. This code is also available on the course website if you would rather explore it in an editor.

For this task, we will draw three different program control flow graphs.

1) Draw the program control flow graph for the function `llNewNode`. When creating this graph:
   - use a separate node for each *basic block* (your *basic block* should combine together sequential program statements as described in the notes)[1]
   - assume that only the lines of code pertaining to executable instructions are to be represented
   - use the line numbers to identify each node[2]
   - use arrows to show the direction of traversal between nodes
   - circle any nodes that represent a point where control flow leaves the graph[3]
2) Draw a second program control flow graph for `inAppend`. Be sure to clearly label your graphs so it is clear which graph relates to which function.
3) Draw a third program control flow graph for `inInsertByIndexOrder`

When creating your graphs, you can do so in any drawing software or do so by hand and scan them in.

Be sure to do so neatly, so that it is clear to which nodes your labels pertain, and it is additionally clear which edges connect which nodes.

If you are drawing your graphs by hand, and you are scanning by using your phone or other camera, be sure to consider: * your light source: make sure that the page is brightly lit, and not in shadow * flatness of the paper: if your paper is curled or crumpled, it will not scan well. Weight down the corners or otherwise ensure that the page is flat.

[1] This statement was originally completely incorrect, and contradicted both the notes and the book. Please ensure that each *basic block* is represented by each node. (Sept 24)

[2] The code has been updated to now include line numbers – apologies for this confusion. (Sept 24)

[3] A question in class asked whether this meant (A) leaving the graph to go to another page (because the graph was too big) or (B) leaving the graph using a "return" or exit. As the assignment is ambiguous, either is fine. If you have already handed it in and it is clear that you did either of these, the marks will be awarded. If you think it is NOT clear, just add a note to your assignment (a separate file in the dropbox with a name like "NOTE" or "README" is fine). (Oct 6)

# A1/CIS*4150

Unit Tests and Control Flow Graphs

courselink.uoguelph.ca

A. Hamilton-Wright

## Overview

This assignment will exercise your knowledge of control flow graphs and unit tests. Note that this is an individual assignment. All of the submitted work should be your own, and based on the starting code and course examples. Any work you have included from elsewhere (*e.g.*, the examples) must be noted and cited.

## Skills

| Skill | Level |
|---|---|
| control flow (6/6) | ██████ |
| graph structures (3/6) | ███░░░ |
| testing tools (3/6) | ███░░░ |
| unit testing (6/6) | ██████ |

(*)[The skill scale is from 0 (Fundamental Awareness) to 6 (Main Focus).]

### Image description

Checklist. Photo source pixabay.com, Pixabay license.

## Programming Task

For both the Python and Java implementations of the GPA converter found in the course directory, implement a test suite that evaluates the percentage grade to grade letter conversion within the library. Your suites should evaluate the highest, lowest and midrange value for each letter grade the system supports (A through F).

*Remember that we are NOT fixing the GPA converter code. We are simply testing the code*. I realize that it is hard to resist the urge to fix any problems found – but this is entirely opposite of what we are trying to do in this course. The code intentionally has problems for you to find – but as in a real development environment, the focus during testing is to find flaws. Once found, effort can be put into fixing the flaws, but we are not doing that phase of development in this course.

Your code will be graded on the machine `linux.socs.uoguelph.ca`. Be sure that your solution works there.

- Python implementation:
  - this should be in a subdirectory called `python`
  - we should be able to run your solution by running a script called `runtests.py`, e.g.; "`python3 runtests.py`"[4]
  - this solution you can create using only `test cases`
- Java implementation:
  - this should be in a subdirectory called `java`
  - we should be able to build your Java solution by running the script `buildJavaTests.sh`
  - we should then be able to run your Java solution by typing `runJavaTests.sh`
  - for this solution, create a *fixture* that sets up the `GPAconverterTool` once, and then uses it in each of your test cases.

In a file called `README.md` in *each* directory (one for Python, one for Java) list:

- all the test cases you have created
- whether each case passes or fails

A file has been provided for you that outlines what is in the directories – add to or replace this file for your own use.

## Handing In Your Assignment

Combine all of your written answers into a single PDF or DOCX file entitled either "`A1-written.pdf`" or "`A1-written.doc`".

Collect your document as well as all of your code into a single archive using `zip` or `tar`. Please *do not* use RAR or other archive formats, as we may not have the tools to unpack your work.

Unpack and test your archive to be sure that it contains everything you intended, and that the code builds and runs on `linux.socs` as per the instructions above.

Upload your `.zip` or `.tar` file to the Assignment 2 dropbox on Courselink.

[4]This originally simply said "`python`" which on "`linux.socs`" will run a old version 2 python interpreter. (Sept 22)

```c
/*
 *      Code adapted from:
 *              The practice of programming/Brian W. Kernighan and Rob Pike,
 *                      Addison-Wesley professional computing series, 1999.
 *                      ISBN 0-201-61586-X
 *
 */

#ifndef __NAMEVAL_LIST_HEADER__
#define __NAMEVAL_LIST_HEADER__

/*
 * define our types
 */
typedef struct MyData {
        char *key;
        int     value;
        int     index;
} MyData;

/*
 * define our types
 */
typedef struct LLNode LLNode;

struct LLNode {
        MyData data;
        struct LLNode *next;
};



/* llNewNode: create and initialize data */
LLNode *llNewNode(MyData *value);

/* llPrepend: add newp to front of list */
LLNode *llPrepend(LLNode *listp, LLNode *newp);

/* llAppend: add newp to end of listp */
LLNode *llAppend(LLNode *listp, LLNode *newp);

/* llInsertSort: place the value in the list according to index ordering */
LLNode *llInsertByIndexOrder(LLNode *listp, LLNode *newp);

/* llLookupKey: sequential search for name in listp */
LLNode *llLookupKey(LLNode *listp, char *key);

/* llApplyFn: execute fn for each element of listp */
void llApplyFn(LLNode *listp, void (*fn)(LLNode*, void*), void *arg);

/* llFree : free all elements of listp */
void llFree(LLNode *listp);

#endif /*       __NAMEVAL_LIST_HEADER__ */
```

```c
/*
 *      Code adapted from:
 *              The practice of programming/Brian W. Kernighan and Rob Pike,
 *                      Addison-Wesley professional computing series, 1999.
 *                      ISBN 0-201-61586-X
 */

#include <stdlib.h>
#include <string.h>

#include "LLNode.h"


/*
 * llNewNode: create and initialize data
 */
LLNode *
llNewNode(MyData *data)
{
        LLNode *newp;

        newp = (LLNode *) malloc(sizeof(LLNode));

        /* assign data within new node */
        newp->data = *data;

        /* make sure we point at nothing */
        newp->next = NULL;

        return newp;
}


/*
 * llAppend: add newp to end of listp
 *
 * as above, we always return the value that
 * should be the new head of the list
 */
LLNode *
llAppend(LLNode *listp, LLNode *newp)
{
        LLNode *p;

        if (listp == NULL)
                return newp;

        for (p = listp;
                        p->next;
                                p = p->next)
                ;

        p->next = newp;
        return listp;
}


```

```c

/*
 * llPrepend: add newp to front of list
 *
 * by returning ourselves, we give back the
 * value that should be the new head of the list
 */
LLNode *
llPrepend(LLNode *listp, LLNode *newp)
{
        newp->next = listp;
        return newp;
}


/* llInsertSort: place the value in the list according to index ordering */
LLNode *
llInsertByIndexOrder(LLNode *listp, LLNode *newp)
{
        LLNode *p, *prevp = NULL;

        if (listp == NULL)
                return newp;

        /** find either the end, or the point where we should go */
        p = listp;
        while (p->data.index < newp->data.index
                        && p->next != NULL) {
                prevp = p;
                p = p->next;
        }

        /*
         * if we have found a larger index, then we go before it
         */
        if ( p->data.index >= newp->data.index) {
                newp->next = p;

                /*
                 * if there is only one node, there won't be a prevp,
                 * and we are the new head
                 */
                if (prevp == NULL) return newp;

                prevp->next = newp;
                return listp;
        }

        /* otherwise, we go at the end */
        p->next = newp;
        return listp;
}


```

```c

/* llLookupKey: sequential search for key in listp */
LLNode *
llLookupKey(LLNode *listp, char *key)
{
	for ( ; listp != NULL; listp = listp->next)
		if (strcmp(key, listp->data.key) == 0)
			return listp;

	return NULL; /* no match */
}


/* llApplyFn: execute fn for each element of listp */
void
llApplyFn(LLNode *listp, void (*fn)(LLNode*, void*), void *arg)
{
	for ( ; listp != NULL; listp = listp->next)
		(*fn)(listp, arg); /* call the function */
}


/* llFree : free all elements of listp */
void
llFree(LLNode *listp)
{
	LLNode *next;

	for ( ; listp != NULL; listp = next) {

		/** hang on to the next pointer */
		next = listp->next;

		/** free the list node itself */
		free(listp);
	}
}

```

```c
#include <stdio.h>
#include <strings.h>
#include "LLNode.h"



/* forward declarations */
static void printMyData(LLNode *p, void *arg);


/*
 * test data, declared statically so that we can
 * access it in the function below, but not outside
 * this file
 */
static struct {
        int ordering;
        char *tag;
        int value;
} sData[] = {
                {       1, "Not Quite Pi",          3       },
                {       0, "The Answer",            42      },
                {       4, "A Small Postitive Integer",          1       },
                {       7, "Positive Ten",          10      },
                {       6, "Negative Ten",          -10     },
                {       2, "One Hundred",           100     },
                {       3, "One Million",           1000000 },
                {       -1, NULL }
        };


/*
 * doQueries: lookup various values in the list, some we expect
 * to find, and some we do not
 */
void
doQueries(LLNode *linkedList)
{
        char *key;
        LLNode *node;


        key = "The Answer";
        node = llLookupKey(linkedList, key);
        if (node != NULL)
        {
                printf("Found '%s', value=%d\n", key, node->data.value);
        } else {
                printf("Did not find '%s'\n", key);
        }



        key = "The Question";
        node = llLookupKey(linkedList, key);
        if (node != NULL)
        {
                printf("Found '%s', value=%d\n", key, node->data.value);
        } else {
                printf("Did not find '%s'\n", key);
        }


        printf("\n");
}


```

```c

/*
 * load the data in the sData structure into a list,
 * appending each element
 */
LLNode *
loadExampleData()
{
        LLNode *listHead = NULL;
        int i;

        for (i = 0; sData[i].tag != NULL; i++) {
                MyData data;

                data.key = sData[i].tag;
                data.index = sData[i].ordering;
                data.value = sData[i].value;

                /** we could also "prepend" instead */
                listHead = llAppend(listHead, llNewNode(&data));
        }

        return listHead;
}

/*
 * load the data in the sData structure into a list, in order
 */
LLNode *
loadOrderedExampleData()
{
        LLNode *listHead = NULL;
        int i;

        for (i = 0; sData[i].tag != NULL; i++) {
                MyData data;

                data.key = sData[i].tag;
                data.index = sData[i].ordering;
                data.value = sData[i].value;

                listHead = llInsertByIndexOrder(listHead, llNewNode(&data));
        }

        return listHead;
}


```

```

/**
 * Program mainline -- this program takes no command-line arguments
 */
int main()
{
        LLNode *aList;

        /** load the data into the list */
        aList = loadExampleData();

        /* do a couple of lookups */
        doQueries(aList);

        printf("\n\nDumping unordered list\n");
        llApplyFn(aList, printMyData, "unordered");

        /** delete our data */
        llFree(aList);


        /* now repeat, but with an ordered list */
        /** load the data into the list */
        aList = loadOrderedExampleData();
        doQueries(aList);
        printf("\n\nDumping ORDERED list\n");
        llApplyFn(aList, printMyData, "in-order");
        llFree(aList);

        return 0;
}

/*
 * used in llApplyFn to print the list values.  The
 * argument here is assumed to point at a (char *) tag
 */
static void printMyData(LLNode *p, void *arg)
{
        printf(" . %2d : '%16s' %d (%s)\n",
                        p->data.index, p->data.key, p->data.value,
                        (char *) arg);
}

```