CS 246 MIDTERM EXAM 2

Spring 2021

1. You will receive an email from me with your exam on Monday at about 7am EDT. You may start your exam at any time up to 1:10 pm EDT. I must receive the exam by 2:45 pm EDT. (If you have a testing accommodation and start at 1:10 pm you have until the time specified by your accommodation + 15 minutes as below.)
2. You have 80 minutes from the time you start to complete this exam. You have an additional 15 minutes for electronic submission.
3. On the first page of your submission clearly write the time you started and the time you completed the exam. Also sign by these times to indicate that you have abided by the honor code.
4. All answers should be written on blank (or lined) sheets of paper. Also, answers may be typed or written using a tablet or computer.
5. SUBMISSION: send your answers to gtowell246@cs.brynmawr.edu. If you typed everything, or wrote on a tablet, just copy and paste or attach the document. For handwritten answers, take photos or scan. Please ensure legibility. You submission should be in PDF or some other common format.
6. Be sure your answers are clearly labeled and that every page has your name
7. This is an open everything exam. The only restriction is that you may not discuss the questions or your answers with anyone.
8. All code given in this test compiles and runs without errors. gcc may give warnings during compilation, but the code does compile and run.
9. In order to be eligible for as much partial credit as possible, show all of your work for each problem, and clearly indicate your answers. Credit cannot be given for illegible answers.
10. Code that you write should be as close to correct (runnable) as possible. Small syntax errors will not cost you points, but code that is unclear will. You may write your code using VSC; you may compile and test. I do not encourage this (I even discourage doing so as it will tend to slow you down), I just note that it is permitted.

11. If you make an assumption (or are unsure of your interpretation of a question), state that assumption or interpretation clearly. For instance you might say "I assume that the question means that I should do …". If I agree that your interpretation is valid, then your answer may be eligible for full credit even if it is not what I intended. Be warned, I may not agree with your interpretation or I may feel that you interpretation is only worth partial credit.

12. I will be available on zoom during the class period. I suggest you have an open zoom session but with the sound off. If you have a question turn the sound on and ask. I will also watch for chats, but I am notoriously poor at noticing those.

13. If you are taking the exam in an alternate time slot, I will not be available on zoom. You can send me email; I may respond quickly. Better in this case, is handle any questions using rule 11. That is, consider the question, decide on a reasonable interpretation, clearly state your interpretation, then answer the question per your interpretation.

14. You may use any function defined in stdio.h, stdlib.h, ctype.h and string.h, except if noted in particular questions.

15. Read the entire exam before answering any questions, so you have time to think about the potentially tricky problems (and/or ask questions via email).

Problem 1: 20 points

The signature and top-level comment of the altCase function is given below

```
/**
 * Make the case of the returned string alternate, starting with lower case
 * For example  given asdf return aSdF
 * Also, given ASDF return aSdF
 * Any non-alphabetic characters should be ignored in this alternation
 * for instance: give AS12D34f  return aS12d34F
 * The string returned should be a new string.
 * The original string should NOT be changed
 * @param src the string to work with
 * @return a copy of the src string, manipulated as described.
char* altCase(char * src)
```

Implement this function recursively, using only pointers. It is acceptable to use a recursive helper function if you would find that useful. It is acceptable if your implementation allocates memory that is not freed.

For a 10% penalty, you may use array indices.

For a 20% penalty, use iteration rather than recursion.

```
char* altCase(char *src){
    char *answer = malloc((strlen(src) + 1) * sizeof(char));
    //indicates whether next letter should be lower case or upper case
    //1 indicates the number should be upcased, 0 indicates it should be
    //downcased
    int lowUp = 0;
    helper(src, answer, lowUp);
    return *answer;
}
char* helper(char *src, char *answer, int lowUp){
    //checks if lowercase and if it should be upcased
    if(*src >= 97 && *src <= 122 && lowUp == 1){
        *src = *src - 32;
        //alternate between 1 and 0
        lowUp = 1 - lowUp;
    //checks if uppercase and if it should be downcased
```

```
    } else if(*src >= 65 && *src <= 90 && lowUp == 0){
        *src = *src + 32;
        //alternate between 1 and 0
        lowUp = 1 - lowUp;
}
    *answer = *src;
    *src++;
    *answer++;
    if(*src == "\0"){
return; }
    return helper(*src, *answer, lowUp);
}
```

Problem 2: 20 points

Consider the following implementation of the remove function for a Hashtable that uses linear probing. The interesting thing about this implementation is that it does not use tombstones. Describe why tombstones are not needed in this implementation. A complete answer should probably start with a definition of the use of tombstones in hashtables that use open addressing. More importantly, a complete answer should pretty thoroughly document the cb function. In this implementation, empty spots in the hashtable are indicated using a string of zero length in the keys array.

```
void cb(int htSize, int vs[htSize], char ks[htSize][20], int hv, int loc) {
    int lloc = (loc + 1) % htSize;
    int backk = loc;
    while (lloc != loc) {
        printf("cb %d\n", lloc);
        if (ks[lloc][0] == '\0')
            return;
        int llochash = hashit(ks[lloc], htSize);
        if (llochash < lloc || ((lloc<loc) && (llochash > lloc))) {
            strcpy(ks[backk], ks[lloc]);
            vs[backk] = vs[lloc];
            ks[lloc][0] = '\0';
            backk = lloc;
        }
        lloc = (lloc + 1) % htSize;
    }
}

/**
 * Remove a key from the a hashtable (if the key is in the hashtable)
 * The hashtable is implemented using two arrays, one holding values and the
other keys
 * @param htSize the size of the hashtable
 * @param values -- the values array of the hashtable (values are integers)
 * @param keys -- the keys array of the hashtable (keys are strings)
 * @param hv -- the hash value of the key to be deleted
 * @param ky -- the key to be deleted
 * **/
int remHT(int htSize, int values[htSize], char keys[htSize][20], int hv,
char* ky) {
    int try = hv;
    int wrap = 0;
    while (wrap==0 || try != hv) {
       if (keys[try][0] == '\0') { return -1; }
       if (strcmp(keys[try], ky)==0) {
            keys[try][0] = '\0';
            cb(htSize, values, keys, hv, try);
            return 1;
        }
        try++;
        if (try>=htSize) {
            try=0;
            wrap=1;
        }
    }
    return -1;
}
```

In a hashtable that uses open indexing, if a new value is hashed for insertion to an already-full index, then the value is instead assigned to the next (linearly) available index. When retrieving a value, the hash program first checks the index that results from the hash function, and then proceeds linearly until it finds the value by the key its searching for, or hits an empty index (meaning the value is not in the table). When a value is removed from the table, it's replaced by a tombstone, which is a marker so the function is aware that at one point, the index holding a tombstone once held a values instead, and other inserted values may have been displaced as a result.

This function doesn't require tombstones because after removing a value from the table, it readjusts the index of other values that had their indices changed due to the presence of the recently-removed value. Then the function doesn't need to track through tombstones where removed values used to be held.

The cb function works as follows:

- ● loc stores the index of the recently removed value.

- ● lloc starts one index later (or at the 0th index, if loc is the last index), and increments
  throughout the whole table, until it either loops around or hits an empty index.

  - ○ If lloc hits an empty index, then the function exits, because no indices beyond
    that point can possibly hold misplaced values.

  - ○ When lloc hits a full index, it hashes the value stored there, to see if it's meant to
    be stored there without open addressing, or if the value was pushed there due to
    the presence of the value previously stored at loc.

  - ○ If the value was forced there, (llochash < lloc || ((lloc<loc) && (llochash > lloc))),
    the value is copied into the spot where the hashing function

would’ve taken it
originally (captured by backk).

- ○ The value is copied over from its current location, and backk is updated to point
to the now empty index, where the next value to be updated will go.

- ○ The function enters its next iteration.

Problem 3: 20 points

Write a program that uses a recursive function to calculate the sum of the number 1 … 1,000,000. The function should NOT use loops.

Give the commands to compile and run this program such that the program is guaranteed to not die with a segmentation fault or a stack overflow.

```
#include <stdio.h> #define MAX_NUM 1000000

int recursiveSum(int index, int sum){
printf("sum: %d\n", sum);
printf("index: %d\n", index);

if (index == MAX_NUM){
  return sum;

} else {
  index++;
  return recursiveSum(index, sum + index); }

}

int main(){
printf("%d\n", recursiveSum(0, 0)); return 0;

}
```

Commands:
gcc O2 —o recursum recursiveSum.c

./recursum

Problem 4: 20 points

In celebration of it being almost tax day (normally April 15) create a “taxpayer” struct that holds the following information.

first name

last name

birth year

tax payer id (social security number)

previous years income

When appropriate, information in this structure should be stored in dynamically allocated spaces that use exactly as much space as needed to store the information.

Write constructor and destructor methods for this struct.

Write a small main function that shows the correct use of the constructor and destructor for at least 2 taxpayers.

```
typedef struct {
char *firstName;
char *lastName;
int birthYear;
char *taxID;
int income;
} taxpayer;

char* strmcopy(char* src) {
char* newstr = malloc((strlen(src)+1)*sizeof(char)); strcpy(newstr,
src);
return newstr;

}

taxpayer * makeTaxpayer(char *first, char *last, int year, char *id,
int inc) {
 taxpayer *rtn = malloc(sizeof(taxpayer));
 rtn->firstName = strmcopy(first);
 rtn->lastName = strmcopy(last);
 rtn->birthYear = year;
 rtn->taxID = strmcopy(id);
 rtn->income = inc;
 return rtn;
```

```
}

void freeTaxpayer(taxpayer *tp) {
free(tp->firstName);
free(tp->lastName);
free(tp->taxID);
free(tp);

}

int main() {
taxpayer *payer1 = makeTaxpayer("H", "M", 1990, "3030232", 100000);
taxpayer *payer2 = makeTaxpayer("A", "B", 1980, "9323232", 50000);
printf("%s %s %d %s %d\n", payer1->firstName, payer1->lastName,
payer1->birthYear, payer1->taxID, payer1-

>income);
freeTaxpayer(payer1);
freeTaxpayer(payer2);

}
```

Problem 5: 20 points

The following two .c files and one .h file give a stupidly complex program that just echos a command line argument (if it exists) and otherwise prints DEFAULT. Do not edit these "files".

file: m2_5m.c

```
#include <stdio.h>
#include "m2_5a.h"
int main(int argc, char const *argv[])
{
    printf("%s\n", inp(argc, argv));
    return 0;
}
```

file: m2_5a.c

```
#define DEFAULT_ARG "DEFAULT"
char* inp(int argc, const char *argv[]) {
    if (argc<2)
        return DEFAULT_ARG;
    return argv[1];
}
```

file: m2_5a.h

```
char *inp(int argc, const char *argv[]);
```

PART 1: Write makefile to compile this set of files into a runnable program. Include in your makefile a "clean" rule that deletes the executable and any other files that result from your compilation.

PART 2: Write a shell script that will run at least 5 tests of the executable. The shell script must use a loop to run these tests. The shell script should not take any command line arguments. The shell script should not simply have 5 lines, each of which runs one test.

PART 3: Add a rule to your makefile named "test". The test rule should recompile (and link) the executable if needed and then run your shell script.

```
OBJS = m2_5a.o m2_5m.o
SOURCE = m2_5a.c m2_5m.c
HEADER = m2_5a.h
OUT = m2
```

```
CC =gcc
FLAGS = -g -c

all: $(OBJS)
	$(CC) -g $(OBJS) -o $(OUT)

m2_5a.o: m2_5a.c
	$(CC) $(FLAGS) m2_5a.c

m2_5m.o: m2_5m.c
	$(CC) $(FLAGS) m2_5m.c

clean:
	rm  *.o m2

test: all
	chmod 777 shelly.sh
	./shelly.sh


Part 2:

#!/bin/bash

for value in {1..5} do

    ./m2 $value

done
```