Worksheet 5

Overview

The workshop for Week 6 will start with a tutorial on Hash Functions and Hash Tables.

Tutorial Questions

Question 5.1 You are given a hash table of size 13 and a hash function *hash(key) = key % 13*.
Insert the following keys into the table, one-by-one, using linear probing for collision resolution.

Keys to insert: 14, 30, 17, 55, 31, 29, 16

Now insert the same keys into an (initially empty) table of the same size using double hashing for collision resolution, with *hash2(key) = (key % 5) + 1*.

Location:	0	1	2	3	4	5	6	7	8	9	10	11	12
Linear Probing:													
Double Hashing:													

Programming Exercises

Programming 5.1 Prior to the semester we've planned to release Assignment 1 such that you can use this time to work on Assignment 1 with a tutor present. So feel free to use this time for that.

Programming 5.2 If you've managed to complete the assignment and your report and don't have any questions about those, try implementing the two hash tables as above. This exercise is an opportunity to become a bit more familiar with function pointers. Function pointers aren't likely to come up in an exam, but do arise in some very generally written libraries, such as the qsort and bsearch stdlib implementations of quicksort and binary search (respectively). The right-left rule is a good trick for getting these definitions right, so you might find it useful to look that up if the definitions seem strange to you.

```
/* hashT.h */
struct hashTable;

/* Insert the given key into the given hash table comparing the
    hash of the given key to the keys in the table already,
    placing it at the next free spot after where it should
    have otherwise been placed. If used >= size, the function
    returns, doing nothing. */
void insertLP(struct hashTable *table, int *key, void *value);

/* Calculates the interval of the given key using the equation
    (key % 5) + 1 and then calls insertDH with this value. */
void insertDH5(struct hashTable *table, int *key, void *value);

/* Insert the given key into the given hash table comparing the
    hash of the given key to the keys in the table already,
    trying to place it at the next interval hash2key distance away
    from where it would otherwise have been placed,
    continuing this process until a free spot is found and then
    the value is placed in this location. If used >= size,
    the function returns doing nothing. */
void insertDH(struct hashTable *table, int *key, void *value, int hash2key);
```

```
/* Allocates space for a hash table and assigns its hash, insert
    and print (for a single data item) functions. */
struct hashTable *create(int tableSize, int *(*hash)(void *),
    void (*insert)(struct hashTable *, int *, void *),
    void (*print)(void *));

/*
    Insert the item in the hash table.
*/
void insert(struct hashTable *table, void *value);

/* Prints the given hash table. */
void printTable(struct hashTable *table);

/* hashT.c */
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

struct hashTable {
    int size;
    int used;
    void **data;
    int *(*hash)(void *);
    void (*insert)(struct hashTable *, int *, void *);
    void (*print)(void *);
};

void insertLP(struct hashTable *table, int *key, void *value){
    /* Write this. NOTE: if you have already written insertDH,
        this is one line of code. */

}

void insertDH5(struct hashTable *table, int *key, void *value){
    /* Write code to calculate the interval from the key and
        then call insertDH with this value. */

    insertDH(table, key, value, interval);
}

void insertDH(struct hashTable *table, int *key, void *value, int hash2key){
    /* Write this. */




}

struct hashTable *create(int tableSize, int (*hash)(void *),
    void (*insert)(struct hashTable *, void *, void *), void (*print)(void *)){
    int i;
    struct hashTable *returnTable = (struct hashTable *) malloc(sizeof(struct hashTable));
    assert(returnTable);

    returnTable->size = tableSize;
    returnTable->used = 0;
    returnTable->hash = hash;
    returnTable->insert = insert;
    returnTable->print = print;
    returnTable->data = (void **) malloc(sizeof(void *)*tableSize);
    assert(returnTable->data);

    /* Set all elements to NULL, so we know which elements
        are used. */
    for(i = 0; i < tableSize; i++){
        (returnTable->data)[i] = NULL;
    }

    return returnTable;
}

void insert(struct hashTable *table, void *value){
    int key;
```

```
    key = table->hash(value);
    table->insert(table, key, value);
    free(key);
}

void printTable(struct hashTable *table){
    int i;
    printf("|");
    for(i = 0; i < table->size; i++){
        if(table->data[i]){
            table->print(table->data[i]);
        } else {
            printf(" ");
        }
        printf("|");
    }
    printf("\n");
}

/* main.c */
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
/* #include "hashT.h" */

#define TABLESIZE 13
#define INPUTSIZE 7

int *hashFunc(void *val){
    int *retVal = (int *) malloc(sizeof(int));
    assert(retVal);

    *retVal = *((int *)val) % TABLESIZE;

    printf("Hashing %d: %d\n",*((int *)val), *retVal);

    return retVal;
}

void print(void *val){
    if(val){
        printf("%d",*((int *) val));
    }
}

int main(int argc, char **argv){

    int inputs[INPUTSIZE] = {14, 30, 17, 55, 31, 29, 16};

    struct hashTable *DH5Table;
    struct hashTable *LPTable;
    int i;

    LPTable = create(TABLESIZE, &hashFunc, &insertLP, &print);
    printf("Created Linear Probing hash table of size 13\n");
    printTable(LPTable);

    for(i = 0; i < INPUTSIZE; i++){
        printf("Inserting %d into Linear Probing hash table\n",inputs[i]);
        insert(LPTable, &(inputs[i]));
        printTable(LPTable);
    }
    printf("Finished inserting items into Linear Probing hash table\n\n");

    DH5Table = create(TABLESIZE, &hashFunc, &insertDH5, &print);
    printf("Created Double Hashing table of size 13 using hash2(x) = (key %% 5) + 1\n");
    printTable(DH5Table);

    for(i = 0; i < INPUTSIZE; i++){
        printf("Inserting %d into Double Hashing hash table\n",inputs[i]);
        insert(DH5Table, &(inputs[i]));
        printTable(DH5Table);
    }
    printf("Finished inserting items into Double Hashing hash table\n");
```

```
    return 0;
}
```

Programming Challenge 5.1 Your friend in the police force has sent you an encrypted text file with evidence for an ongoing court case (the files were just ominously called "CRIME1.txt" and "CRIME2.txt"). Half of the data needs to be encrypted and the other half encrypted to find what the full message actually was. The police forensic unit also managed to find the encryption and decryption keys as well as the modulus (n).

In this programming challenge, you will implement a simple 7-bit version of RSA, a very commonly used encryption algorithm. RSA works a lot like a hash function, though it is usually performed over much larger input spaces (typically in the region of 2048-bit numbers). Typically, RSA uses what is known as the square-and-multiply algorithm to do its encryption and decryption. This algorithm looks at each bit and performs either a square or multiply by 2. After each multiplication, you can mod the result by the modulus given.

For example, for the value x^10 which has binary representation 0b1010 the square and multiply algorithm would look something like this:

1. We start with res = 1, in case we had x^0.
2. We start with the highest order bit, which in this case is a 1, so we multiply by x, giving us x^1.
3. Since there's another bit to look at, we square, which is equivalent to doing (x^(0b1))^2, giving us x^(0b10).
4. Since the next bit is 0, we don't multiply by x.
5. Since there's another bit to look at, we square, which is equivalent to doing (x^(0b10))^2, giving us x^(0b100).
6. Since the next bit is 1, we multiply by x, giving us x^(0b100 + 0b1) = x^(0b101).
7. Since there's another bit to look at, we square, giving us x^(0b1010).
8. Since the next bit is 0, we don't multiply by x.
9. Since there's not another bit to look at, we're done.

NOTE: This is the most intuitive way to explain the algorithm, but you might find doing the square before the possible multiply is more intuitive way to write the algorithm (since you don't have to check if there's another bit).

It may be easiest to use the binary right-shift operator (>>) and the binary operator & (with 1) to work out how many bits there are to look through. Note also this algorithm is only correct for positive indices.

Note also: This works because the d, e and n chosen have specific properties to do with prime numbers and properties of modular arithmetic.

```
#include <stdio.h>
#include <stdlib.h>

#define N 133
#define E 5
#define D 11

/* Returns an int representing the given
    value encrypted using RSA using the
    given parameters. */
unsigned int encrypt(unsigned int n,
    unsigned int e, unsigned int value);

/* Returns an int representing the given
    value decrypted using RSA using the
    given parameters.
*/
unsigned int decrypt(unsigned int n,
    unsigned int d, unsigned int value);

unsigned int encrypt(unsigned int n,
    unsigned int e, unsigned int value){
    /* Do (value^e) mod n */
```

```
}

unsigned int decrypt(unsigned int n,
    unsigned int d, unsigned int value){
    /* Do (value^d) mod n. */
    return encrypt(n, d, value);
}

int main(int argc, char *argv){
    /* The keys to encrypt */
    unsigned int toEncrypt[] = {72, 63, 14, 72, 72, 131, 95, 107, 92, 104, 51,
        24, 54, 14, 92, 72, 10, 95, 131, 96, 21, 104, 47, 25, 131, 95, 24};
    unsigned int toDecrypt[] = {96, 51, 13, 51, 129, 96, 73, 7, 41, 7, 80, 6,
        109, 80, 2, 8, 2, 6, 56, 7, 41, 41, 124, 8, 57, 7};

    int i;

    for(i = 0; i < sizeof(toEncrypt)/sizeof(toEncrypt[0]); i++){
        printf("%c",encrypt(N, E, toEncrypt[i]));
    }
    for(i = 0; i < sizeof(toDecrypt)/sizeof(toDecrypt[0]); i++){
        printf("%c",decrypt(N, D, toDecrypt[i]));
    }
    printf("\n");

    return 0;
}
```