#### Jonathan Liao

#### CS 305 HW 3 Report

**1. Questions (include these in your write-up):**

1a. (.5 point) Meets all specifications. HOWEVER, my list appears backwards to Vandegrift’s sample output which is due to her contradictory instructions. She said that the items with higher ID values should be put at the front of the list, not at the back: her sample output is contrary to this objective with the receipt printing items from lowest to highest instead. For example, my receipt starts with the highest ID numbers descending to the lowest ID numbers. Nonetheless, I could easily change the print\_receipt function to reverse my printing if I wanted to.

1b. (.5 point) output.txt included

2. (.5 point) testRun1: I put print statements before successfully adding the retail\_item to the linked list. This ensured that add\_item worked.

testRun2: By putting a print statement that said “Not here” this showed me the computer checking each item in the receipt for a matching identification number. This ensured that delete\_item worked.

3. (.5 point) receipt \*\* rec\_ptr is the head of the linked list. All we care about is getting a pointer to the first node, which will give us access to elements further down the list. Unlike arrays, we don’t pass on an entire array; a linked list can be navigated through the use of a head pointer.

4a. (.5 point) Picture 1: (assuming we don’t sort based on ID number yet)

cookie

NULL

candy

coffee

Shop(head)

4b. (.5 point) Picture 2: (assuming we don’t sort based on ID number yet)

NULL

5a. (.25 point) With an array, you’d have to check every element to see if it’s NULL because you have no other way to find the tail location of the array. To stick a tail element to an array of N size, it would take N runs to find the end of the array and stick the new element there.

candy

coffee

cookie

shirt

Shop(head)

eggs

HOWEVER, if you were given the number of elements in the array (i.e. given array size has ten elements 10), you could skip checking each element for NULL and stick the new element to the end (at index 10), which would only be 1 step.

5b. (.25 point) The worst case for adding a new item to a linked list in HM3 would happen if you had to also check every element for an ID# value that is less than it.number—this means the new retail\_item would go at the tail. You’d have to take N steps to check every element in a linked list that is N elements long.

5c. (.25 point) Since an array must collapse on itself, the worst case would be to remove the first element in the array and causing a ripple effect, making all of the other elements move down 1 index. That would take (N-1) steps to perform.

5d. (.25 point) Since deleting and freeing a node from a linked list in HM3 doesn’t move other nodes, it doesn’t matter how big the list is. All that is happening is that the node before the deleted node points to the node after the deleted node and the deleted node is freed. Only two steps to perform really.

5e. (.25 point) In HM1, there would still be 1990 spaces left in receipt. That means the computer wasted 1990\*sizeof(retail\_item) bytes of memory.

5f. (.25 point) A linked list can be dynamically allocated. If we are ever unsure of how much memory will be used and want to be efficient with memory usage at the same time, a linked list is a good choice.

5g. (.25 point) Arrays are easier to implement and do not take up so much memory for storing pointers. It is also easier to access the index values and transverse the elements.

6. (.25 point) I spent about 10 hours.

**Appendix A: (copy this statement if it applies to you)** I verify that the code and this write-up were authored by me. I have documented the help I have received in comments in the code files.

**Appendix B**: Copy and paste your receipt.c code file and main.c file here (use Courier New 8pt font so the characters line up correctly)

MAIN.C

/\* CS 305, SPRING 2016

\* HOMEWORK 3

\*

\* main.c

\* author: Tammy VanDeGrift

\*

\* creates receipts and retail\_items and tests them

\*

\* compile with receipt.c and retail\_item.c

\* gcc receipt.c retail\_item.c main.c

\* OR

\* gcc -o runtest receipt.c retail\_item.c main.c

\* OR

\* use makefile:

\* make runtest

\* If creating produce object file, then run with command:

\* produce

\* Otherwise, run with command:

\* a.out

\*/

#include <stdio.h>

#include <stdlib.h>

#include "receipt.h"

#include "retail\_item.h"

/\* prototypes \*/

int run\_test(void);

/\* main

\* using void paramater since we are not using command line arguments

\* for this program

\*/

int main(void) {

run\_test();

}

/\* run\_test

\* runs a test, creating receipts and adding/deleting items

\*/

int run\_test(void) {

/\* create two receipts \*/

/\* note: using pointers for receipts so the receipt data can be

modified \*/

receipt \*shop = create\_receipt();

receipt \*fred = create\_receipt();

/\* create retail items \*/

/\* note: not using pointers to retail\_items since once they are created

\* they are not later modified -- only used as data \*/

retail\_item candy = create\_retail\_item(1001, "Snickers", 75);

retail\_item coffee = create\_retail\_item(3033, "Starbucks Blend", 699);

retail\_item cookie = create\_retail\_item(2222, "Chips Ahoy", 329);

retail\_item shirt = create\_retail\_item(9212, "Large T Shirt", 1599);

retail\_item eggs = create\_retail\_item(1234, "Dozen Eggs", 369);

retail\_item oranges = create\_retail\_item(5545, "Clementines", 799);

retail\_item bread = create\_retail\_item(3222, "Orowheat bread", 499);

retail\_item grapes = create\_retail\_item(5122, "Red grapes", 549);

retail\_item yogurt = create\_retail\_item(1121, "Vanilla yogurt", 75);

retail\_item peanuts = create\_retail\_item(2121, "Planters Peanuts Salt", 802);

retail\_item shampoo = create\_retail\_item(4424, "Pantene Shampoo", -633);

retail\_item gift\_card = create\_retail\_item(8000, "Fred Meyer Gift Card", 5000);

/\* put items into receipt \*/

add\_item(&shop, candy);

add\_item(&shop, coffee);

add\_item(&shop, cookie);

/\* at point to draw picture 1 in report \*/

add\_item(&shop, shirt);

add\_item(&shop, eggs);

/\* at point to draw picture 2 in report \*/

add\_item(&shop, eggs);

add\_item(&shop, oranges);

add\_item(&shop, bread);

add\_item(&shop, grapes);

add\_item(&shop, yogurt);

add\_item(&shop, peanuts);

/\* print the receipt \*/

print\_receipt(shop);

/\* delete item \*/

printf("Trying to delete item 1234\n");

delete\_item(&shop, 1234);

/\* print the receipt \*/

print\_receipt(shop);

/\* delete item \*/

printf("Trying to delete item 10000\n");

delete\_item(&shop, 10000);

/\* add items to fred \*/

add\_item(&fred, coffee);

add\_item(&fred, peanuts);

add\_item(&fred, bread);

add\_item(&fred, shampoo);

print\_receipt(fred);

/\* delete items \*/

printf("Trying to delete item 3222\n");

delete\_item(&fred, 3222);

printf("Trying to delete item 3033\n");

delete\_item(&fred, 3033);

printf("Trying to delete item 2121\n");

delete\_item(&fred, 2121);

printf("Trying to delete item 1234\n");

delete\_item(&fred, 1234);

print\_receipt(fred);

/\* add items to fred \*/

add\_item(&fred, shampoo);

add\_item(&fred, bread);

add\_item(&fred, gift\_card);

print\_receipt(fred);

/\* free memory for receipts \*/

free\_receipt(fred);

free\_receipt(shop);

return EXIT\_SUCCESS;

} /\* end main \*/

RECIEPT.C  
//Jonathan Liao

//Receipt.c

#include <stdio.h>

#include <stdlib.h>

#include "retail\_item.h"

#include "receipt.h"

receipt \* create\_receipt() {

return NULL; //linked list is empty

}

void add\_item(receipt \*\* rec\_ptr, retail\_item it){

receipt \* rec = \*rec\_ptr;

receipt \* prev = NULL; //I don't really need this, but vandegrift insists

//case1

if(rec == NULL) {

rec = (receipt \*)malloc(sizeof(receipt) );

rec->item = it;

rec->next = NULL;

\*rec\_ptr = rec;

return;

}

//case2

if(it.number >= rec->item.number) {//item ID# of added item is greater than the item ID# of previous item

receipt \* newRec = (receipt \*)malloc(sizeof(receipt) );

newRec->item = it;

newRec->next = \*rec\_ptr;

\*rec\_ptr = newRec;

return;

}

//case3

if(it.number < rec->item.number) {

receipt \* newRec = (receipt \*)malloc(sizeof(receipt) );

newRec->item = it;

newRec->next = NULL; //for now

rec = \*rec\_ptr; //assigned rec as starting head value DO I REALLY NEED THIS??????????????????????

while(rec->next != NULL) {

if(it.number >= rec->next->item.number) {

newRec->next = rec->next;

rec->next = newRec;

return;

}

rec = rec->next;

}

//case4 (inside case 3 since if the loop traverses through the entire linked list and reaches where

//rec->next == NULL, that means we've reached the end. The difference of Case 3/4 is case 4 assigns

//newRec->NULL since it's the end

if(rec->next == NULL) {

rec->next = newRec;

newRec->next = NULL;

return;

}

}

}

int delete\_item(receipt \*\* rec\_ptr, int item\_num){

receipt \* rec = \*rec\_ptr;

receipt \* prev = NULL;

//case1, the receipt is empty to begin with

if(rec == NULL) {

printf("Nothing to delete. Receipt is empty\n");

return 0;

}

//case2, first item in receipt is thing we want to delete

if(rec->item.number == item\_num) {

\*rec\_ptr = rec->next;

free(rec);

printf("Succesfully deleted item %d\n", item\_num);

return 1;

}

//case3, item wanted to delete is somewhere in the receipt. Must traverse through

if(rec->item.number != item\_num) {

while(rec->next != NULL) {

prev = rec; //keep track of previous node before moving to next one

rec = rec->next;

if(rec->item.number == item\_num) {

prev->next =rec->next;

free(rec);

printf("Succesfully deleted item %d\n", item\_num);

return 1;

}

}

printf("could not find item %d. Failed to delete\n", item\_num);

return 0;

}

}

void print\_receipt(receipt \* rec){

int sum = 0;

printf("Receipt:\n");

while(rec != NULL) {

print(rec->item);

sum += rec->item.price\_cents;

rec = rec->next;

}

printf("Total sum of this receipt is $%.02f\n\n", ((double)sum/100.00) );

return;

//Below, I tried to use a recursive function, but was problematic for printing $sum total

/\*if(rec == NULL) {

return; Nothing to print. The receipt is empty

}

else {

print(rec->item);

print\_receipt(rec->next);

return;

}

\*/

}

void free\_receipt(receipt \* rec){

if(rec == NULL) {

return; //nothing to free

}

if(rec->next == NULL) {

free(rec); //nothing else to free after this

return;

}

else {

free\_receipt(rec->next);

free(rec);

return;

}

}