Gradi Tshielekeja Mbuyi

Midterm Exam 1

COP3502 Computer Science 1 – Summer 2022

Reflective Questions (30 points)

Limit your answers to one paragraph or less. You may do research, but your writing must be your own; your answers may be checked for plagiarism.

1. Explain the differences between a statically allocated array, a dynamically allocated array, and a linked list.

   A statically allocated array has a sized that's predefined, whereas a dynamically allocated array allows the user to allocate n number of memory as needed when running the program. A linked list on the other hand use pointers to reference other related nodes.

2. Linked lists have terrible performance for random access or searching of internal entries. Why?

   To find an item/node within a linked list, a programmer has to use a loop to search for the item (its pointers). Unlike an array, there's no easy way to access random nodes. Each node only has a pointer to the next or occasionally its previous node. Accessing a random items in a linked list requires traversing the whole list.

3. (a) Explain the advantages of adding a tail pointer to a linked list.
   (b) Explain the advantages of doubly-linked over singly-linked lists.

   Adding a tail pointer to a linked list allows the programmer to easily remove and insert at the end of the list. Instead of using a for-loop which would take *O(n)* to traverse the list, using the tail pointer to remove and insert would only take *O(1).* One of the benefits of a doubly-linked list is its ability to keeps track of the previous and next node. This allows the programmer to easily remove items within the list, not having to worry loop through the list to find the previous node and fix the remaining list.

Introductory Performance Analysis (20 points)

4. An unknown searching algorithm took a second to find an item in a list of 250 entries, two seconds to find an item in a list of 2,500 entries, and three seconds to find an item in a list of 25,000 entries. Estimate its runtime in big-$\mathcal{O}$ terms. How did you arrive at your answer?

   log x = 1 -> x = 10 -> 250/y = 10 -> y = 25 -> therefore **$\mathcal{O}(\log n/25)$ is its estimate runtime**

5. A binary searching algorithm has logarithmic $\mathcal{O}(\log n)$ performance. It takes a second to find an item in a list of 10,000 entries. How long do you expect it to take to find an item in a list of 50,000,000 entries? How did you arrive at your answer?

   (log 50,000,000 = 7.69897000) / (log 10,000 = 4) = **approximately 1.9247425 seconds**

6. A simple sorting algorithm has quadratic $\mathcal{O}(n^2)$ performance. It takes three minutes to sort a list of 100,000 entries. How long do you expect it to take to sort a list of 200 entries? How did you arrive at your answer?

   3 mins = 180 sec -> 180 sec = 180,000 ms -> 100,000 entries = 180,000x ms-> (100,000)^2 / 180,000 = 55,555.556 -> $\mathcal{O}(200^2)$ = 40,000 -> 55,555.556x = 40,000 -> x = 0.71999… -> **approximately 0.7199 milliseconds.**

7. A naïve searching algorithm took a second to find an item in a list of 250 entries and two seconds to find an item in a list of 500 entries. Estimate its runtime in terms of entries, in milliseconds. How did you arrive at your answer?

   1 sec = 1000 ms -> 250 entries in 1000 ms -> 4 milliseconds * n -> 4n, where n represent the number of entries. In other words, **4 milliseconds per entries.**

Dynamic Memory Management (25 points)

Consider the following structures. **The dparrays in these structures are *NOT* arrays of references – they are allocated dparrays with every member also individually allocated.**

```
typedef struct {
      char *name;                             // allocated
      int commonality;
      int weight;
} monster;

typedef struct {
      char *name;                            // allocated
      char *description;                     // allocated
      double area;
      int nmonsters;
      monster **monsters;                    // fully allocated, NOT a reference array
} region;

typedef struct {
      char *name;                            // allocated
      double diameter;
      int nregions;
      region **regions;                      // fully allocated, NOT a reference array
} planet;
```

- Write the following functions:

```
/* Frees monster m and its contents. */

void dispose_monster(monster *m); // Worth 5 points

void dispose_monster(monster *m) {
     free(m->name);
     free(m);
}

/* Frees region r and its contents, including all monsters in its monsters dparray.
   This may (and should) call dispose_monster(). */

void dispose_region(region *r); // Worth 10 points

void dispose_region(region *r) {
     int i;

     free(r->name);
     free(r->description);

     for(i = 0; i < r->nmonsters; i++) {
          if(r->monsters[i] != NULL) {
               dispose_monster(r->monsters[i]);
          }
     }

     free(r->monsters);
     free(r);
}

/* Frees planet p and its contents, including all regions in its regions dparray.
   This may (and should) call dispose_region(). */

void dispose_planet(planet *p); // Worth 10 points

void dispose_planet(planet *p) {
     int i;

     free(p->name);

     for(i = 0; i < p->nregions; i++) {
          if(p->regions[i] != NULL) {
               dispose_region(p->regions[i]);
          }
     }

     free(p->regions);
     free(p);
}

/* Adds a new monster to region r. You may assume the existence of the function:

   monster *new_monster(char *name, int commonality, int weight);

   Hint: The realloc() function can make your life much easier. */

void add_monster_to_region(region *r, char *mname, int mcommonality, int mweight);

                                                                  // Worth 10 points
```

```
void add_monster_to_region(region *r, char *mname, int mcommonality, int mweight) {
     monster *m;

     m = new_monster(mname, mcommonality, mweight);
     r->monsters = realloc(r->monsters, r->nmonsters + 1 * sizeof(monster *));
     r->monsters[r->nmonsters] = m;
     r->nmonsters += 1;
}

/* Deletes a region from planet p. Fails silently, but does not cause an error, if the named
   region is not on planet p. This may (and should) call dispose_region(). */


void delete_region_from_planet(planet *p, char *rname);  // Worth 15 points

#include <string.h>

void delete_region_from_planet(planet *p, char *rname) {
     int i;

     for(i = 0; i < p->nregions; i++) {
          if(strcmp(p->regions[i]->name, rname) == 0) {
               dispose_region(p->regions[i]);
               return;
          }
     }
}
```

Linked Lists (25 points)

Coding: 15 points

With the following structures defined:

```
struct monster_struct {
      char *name;
      int commonality;
      int weight;
      struct monster_struct *next;
};

typedef struct monster_struct monster;

typedef struct {
      monster *head;
} monster_list;
```

- Write functions to return the second-most-common monster and the third-lightest monster in the list.
- Don't worry about headers.
- Don't worry about which way to resolve ties.
- The list will always have at least three monsters in it, so you don't have to worry about edge

cases.

- Your function prototypes should be:

```
monster *second_most_common(monster_list *m1);

monster *second_most_commom(monster_list *m1) {
     monster *most_common;
     monster *second_most;

     monster *temp;

     most_common = m1->head;
     temp = m1->head->next;

     while(temp != NULL) {
          if(most_common->commonality < temp->commonality) {
               most_common = temp;
          }
          temp = temp->next;
     }

     second_most = m1->head;
     temp = m1->head->next;

     while(temp != NULL) {
          if(most_common->commonality == second_most->commonality) {
               temp = temp->next;
          }else if(second_most->commonality < temp->commonality) {
               second_most = temp;
          }else{
               temp = temp->next;
          }
     }

     return second_most;
}

monster *third_lightest(monster_list *m1);

monster *third_lightest(monster_list *m1) {
     monster *temp;

     monster *lightest;
     monster *second;
     monster *third;

     lightest = m1->head;
     temp = m1->head->next;

     while(temp != NULL) {
          if(lightest->weight > temp->weight) {
               lightest = temp;
          }
          temp = temp->next;
     }

     temp = m1->head->next;
     second = m1->head;
```

```
    while(temp != NULL) {
        if(lightest->weight == second->weight) {
            temp = temp->next;
        }else if(second->weight > temp->weight) {
            second = temp;
            temp = temp->next;
        }else{
            temp = temp->next;
        }
    }

    temp = m1->head->next;
    third = m1->head;

    while(temp != NULL) {
        if(third->weight == second->weight || third->weight == lightest->weight) {
            temp = temp->next;
        }else if(third->weight > temp->weight) {
            third = temp;
            temp = temp->next;
        }else{
            temp = temp->next;
        }
    }

    return third;
}
```

Analysis: 10 points

This is a sub-question of the linked list problem; you don't need to analyze your dynamic memory management functions.

1. In big-$\mathcal{O}$ terms what is the performance of each function? Why?

   The performance of each function, in big-$\mathcal{O}$ terms would be $\mathcal{O}(n)$. Both functions requires us to traverse our linked list, and using a while loop allows us to check each item to find both the second most common and third lightest monster.

2. Would either adding a tail pointer to the list, or making it a doubly-linked list, increase performance? Why or why not?

   I'm not sure but I would say that it could. A doubly-linked list usually decreases the amount of work the computer does.