Midterm Exam 2

COP3502 Computer Science 1 – Summer 2022

Reflective Questions (30 points)

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

1. Explain the difference between a stack and a queue. Give at least one application for each.

   A stack is a last-in-first-out data structures, whereas a queue is a first-in-first-out data structures. We can apply a queue to a linked list, implementing an adder function that will insert items at the tail of the list. The user can also implement a remove function, which will delete items at the head of the list. We can also apply a stack to a linked-list, where the user will be to add and remove items at the head of our list.

2. In terms of head/tail pointers, *and* next/previous pointers, what is the **least complex linked list** capable of implementing a queue with all basic operations requiring $\mathcal{O}(1)$ time? In general terms, how is each operation implemented?

   The least complex linked list would be a singly-linked list that keeps track of its head and tail pointers. Since we'll only be adding at the tail and removing at the head, having a previous pointer isn't necessary. Our **enqueue()** function will use our tail pointer to add items at the end of our linked-list, setting our old tail's next pointer to our new item, our item's next pointer to NULL, and our tail to our new item. Our **dequeue()** function will use our head pointer to remove items at front of our linked list, setting our head to our old head's next, returning our old head. Our **queue_peek()** function uses our head pointer to see if the item passed (in the function) is at our head. It returns 1 to indicate that it is, otherwise, it returns 0. Our **is_queue_empty()** function checks if we have an empty list, returning 0 if both our head and tail pointers = NULL, and returning non-zeroes to indicate that our that our list isn't empty.

3. Compare and contrast selection and insertion sort in terms of the number of comparisons and the number of data copies required. When would insertion sort be likely to outperform selection sort? When would selection sort be likely to outperform insertion sort?

   When attempting to sort a list, the selection sort algorithm requires more comparison between the items, whereas insertion sort requires more copies and far less comparisons. Both algorithm have an average time complexity of *O(n^2)*, with insertion sort outperforming selection sort in its best-case *O(n).* Since selection sort does not adapt to the number of data, it performance the same in its best, average, and worst case. There aren't any cases in which selection sort would outperform insertion sort.

Performance Analysis (20 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. A queue is significantly more efficient if its underlying linked list has both head and tail pointers. Why?

   The tail pointer allows us to add the end of our linked-list without having to go through the list and each of its items. Our enqueue() function therefore has a time-complexity of *O(1).* Removing items in the queue also will also take *O(1)* using our head pointer.

2. What big-$\mathcal{O}$ time do stack push, pop and peek each require when properly implemented with a linked list? Briefly justify your answer.

   When properly implemented, stack push, pop, and peek each require *O(1).* Since a stack is a last-in-first-out data structure, all of our function will utilize our head pointer to remove, add, and peek at our item. We do not need to traverse our list.

3. It's possible to ensure that an insertion sort implementation runs in linear time on a list that's already sorted. How?

   We can divide our list into elements that are already sorted and ones that aren't. Doing so will allow us to work with a smaller data set, only sorting our unsorted chunk, ensuring that our algorithm runs in linear time.

4. Merge sort runs in loglinear time, i.e, $\mathcal{O}(n \log n)$ comparisons. Merge-insertion sort and other hybrid merge sort approaches use quadratic $\mathcal{O}(n^2)$ sorts to sort sublists, but we can still refer to the overall hybrid algorithm as loglinear. Why? **Hint: Consider the definition of big-$\mathcal{O}$.**

   O(n^2 log n) = O(n log n)

Linked Lists (25 points)

Small monsters exhibit a fascinating social ordering behavior: within a given group of small monsters, when it comes time to enter a room (to be fed, to get ready to sleep, to participate in monstercise, etc.), the largest, strongest, most monstery monster in the group insists on entering the room last, and insists that the smallest, weakest, least monstery monster in the group be allowed to enter the room first.

With the following structures defined:

```
struct monster_struct {
   char *name;
   double weight;
   double monsteriness;
   struct monster_struct *next;
   struct monster_struct *prev;
};

typedef struct monster_struct monster;

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

Write a function last_first_monsters() using the following prototype:

```
void last_first_monsters(monster_list *m1);
```

- Takes the **least monstery** monster in the list and places it at the **head** of the list.
- Takes the **most monstery** monster in the list and places it at the **tail** of the list.

Don't worry about sorting the rest of the monsters.

```
// assuming that monster_list is a doubly linked list, instead of a circular list

void last_first_monsters(monster_list *m1){
   monster *m = m1->head;

   monster *least = m;
   monster *most = m;

   double monsteriness = m->monsteriness;

   if(m->next != NULL){
      m = m->next;

      while(m != NULL){
         if(monsteriness < m->monsteriness){
            most = m;
         } else if(monsteriness > m->monsteriness){
            least = m;
```

```
        }

        monsteriness = m->monsteriness;
        m = m->next;
    }

    most->prev->next = most->next;
    most->next->prev = most->prev;
    most->prev = m1->tail;
    most->next = NULL;

    least->prev->next = least->next;
    least->next->prev = least->prev;
    least->next = m1->head;
    least->prev = NULL;
  }
}
```

Queues (25 points)

With the following structures allocated:

```
struct queue_node_struct {
   int payload;
   struct queue_node_struct *next;
   struct queue_node_struct *prev;
};

typedef struct queue_node_struct queue_node;

typedef struct {
   queue_node *head;
   queue_node *tail;
} queue;
```

Implement a queue by writing the functions for these prototypes:

- Make sure to properly allocate and dispose of queue_nodes.
- Don't worry about headers.
- **Note the type of values you are accepting and returning in these prototypes.** Adapt to them!

```
void enqueue(queue *q, int i); // accepts payload
int dequeue(queue *q); // returns payload
int queue_peek(queue *q); // returns payload
int is_empty(queue *q); // return 0 or 1
```

```
queue_node *create_node(int payload){
   queue_node *n = malloc(sizeof(queue_node));
   n->next = NULL;
   n->prev = NULL;

   return n;
```

```c
}

void dispose_node(queue_node *n){
   free(n);
}

queue *create_queue(){
   queue *q = malloc(sizeof(queue));
   q->head = NULL;
   q->tail = NULL;

   return q;
}

void dispose_queue(queue *q){
   free(q);
}

int is_empty(queue *q){
   if(q->head == NULL){
      return 0;
   } else {
      return 1;
   }
}

int queue_peek(queue *q){
   if(is_empty(q) == 1){
      return q->head->payload;
   } else {
      return 0;
   }
}

void enqueue(queue *q, int i){
   queue_node *n;
   n = create_node(i);

   if(is_empty(q) == 0){
      q->head = n;
      q->tail = n;

      n->prev = NULL;
      n->next = NULL;
   } else {
      n->prev = q->tail;
      n->tail = NULL;

      q->tail->next = n;
      q->tail = n;
   }
}

int dequeue(queue *q){
   queue_node *n;
```

```c
    int payload;

    n = q->head;
    payload = n->payload;

    q->head = q->head->next;
    q->head->prev = NULL;
    n->prev = NULL;
    n->next = NULL;

    dispose_node(n);

    return payload;
}
```