Aniket Gupta
2022073

DSA HW-6

Q1. Let the maximum size of the required queue be 'n'

→ Initially, allocate a queue of size 2^0

→ enqueue 2^0 elements in the list

→ when the queue is full, allocate a new queue of size 2^1

→ dequeue 2^0 elements from the old queue and enqueue them in the new queue

→ free the memory of the old queue

→ enqueue $2^1 - 2^0$ elements in the new queue

→ repeat this operation 'R' times

$n = 2^{R-1}$

$\Rightarrow R = \log_2 n + 1$

Allocation Overhead

Let the cost of allocating a new queue and freeing the old queue be 1

Since there are R allocations,
allocation overhead = $R = \log n + 1$

Enqueue overhead

Let the cost of enqueuing an element a queue be 1

$\therefore$ enqueue overhead $= 2^0 + 2^1 + 2^2 + \ldots + 2^{k-1}$

$= 2^k - 1$

Dequeue Overhead

Let the cost of dequeuing an element from a queue be 1

$\therefore$ dequeue overhead $= 2^0 + 2^1 + \ldots + 2^{k-2}$

$= 2^{k-1} - 1$

$\therefore$ total overhead = allocation overhead + enqueue overhead + dequeue overhead

$= (\log n + 1) + (2^k - 1) + (2^{k-1} - 1)$

$= (\log n + 1) + (2n - 1) + (n - 1)$

$= \log n + 3n - 1$

$= \boxed{O(n)}$

(5)

pseudocode written at the end →

Rush
PAGE NO.:
DATE: / /

Q2- Yes, we can use dynamic stack to store local variables.

Simply push elements one by one in the stack and wherever our stack gets full, increase capacity of stack

```
struct stack {
    int size;
    int capacity;
    int* arr;
};
```

```
void insert(int** arr, int* size, int* capacity,
            int element) {
    if (*size == *capacity)
        *capacity *= 2;
        int* new_arr = realloc(*arr, *capacity * sizeof(int));
        if (new_arr == NULL) {
            printf("memory allocation failed");
            exit(1);
        }
        *arr = new_arr;
    }
    (*arr)[*size] = element;
    (*size)++;
```

```
void push(struct stack* s, int val) {
    insert(&(s->arr), &(s->size), &(s->capacity),
           val);
}
```

Q3.

```
void swap(struct node* a, struct node* b) {
    int temp = a->val;
    a->val = b->val;
    b->val = temp;
}

void bubble_sort(struct node* head) {
    if (!head || !head->next)
        return;

    int swapped;
    struct node *ptr1 = head, *lptr = NULL;

    do {
        swapped = 0;
        ptr1 = head;

        while (ptr1->next != lptr) {
            if (ptr1->val > ptr1->next->val) {
                swap(ptr1, ptr1->next);
                swapped = 1;
            }
            ptr1 = ptr1->next;
        }
        lptr = ptr1;
    } while (swapped);
```

P.T.O

```
int count_nodes(struct node* head) {
    int count = 0;
    struct node* curr = head;
    while (curr != NULL) {
        count++;
        curr = curr->next;
    }
    return count;
}

int find_median(struct node* head) {
    int n = count_nodes(head);
    if (n == 0) {
        printf("Error: Empty list");
        return 0;
    }

    bubble_sort(head);

    if (n % 2 != 0) {
        struct node* middle = head;

        for (int i = 0; i < n/2; i++)
            middle = middle->next;

        return middle->val;
    }
    else {
        struct node* middle = head;
        for (int i = 0; i < n/2 - 1; i++)
            middle = middle->next;
        return (int) ((middle->val + middle->next->val) / 2);
    }
}
```

time complexity discussed at the e

→

PAGE NO.: Rush
DATE : / /

```
void insert_node(struct node **head, int new_data)
{
    struct node* new_node = malloc(sizeof(*new_node));
    new_node->val = new_data;
    new_node->next = (*head);
    (*head) = new_node;
}
```

Q5.

```
pushb(S, v)
// S is a stack of integers
// v is an integer value
// This procedure inserts v at the bottom
    of the stack S

    if stack_empty(S)
        push(S, v)
    else
        x = pop(S)
        pushb(S, v)
        push(S, x)
```

20

Q4. (a)

```
struct MaxStackNode {
    Ty val;
    Ty max;
    struct MaxStackNode* next;
};
```

10

X

As we insert elements in the linked list, we keep checking if they are greater than max, and if they are, we update max to store their values. This gets us the maximum value in O(1) operations

Rush

PAGE NO.:

DATE : / /

```
push(head, val)
// head : a pointer to a pointer to the head
          node of the stack
// val : the value which is to be pushed into
         the stack, it is of type Ty
// return : none

    new_node = allocate memory for MaxStackNode
    new_node -> val = val

    if (list_empty(*head) OR
        cmp_val(val, (*head) -> max) > 0)
        new_node -> max = val
    else
        new_node -> max = (*head) -> max

    new_node -> next = *head
    *head = new_node
```

```
pop(head)
// head : a pointer to a pointer to the head
// return : NULL if stack is empty, else the popped node

    if (list_empty(*head))
        return NULL

    tmp = *head
    *head = (*head) -> next
    tmp -> next = NULL
    return tmp
```

Rush
PAGE NO.:
DATE: / /

```
find_max (head)
// head : a pointer to the head
// return : maximum value of type Ty in our stack
   however, NULL if list is empty

   if (list_empty (*head))
       return NULL

   return head -> max -> val;
```

5

continued

Q3. time complexity:

→ in the count_nodes() function, we traverse the linked list once, taking O(n) time

→ in the bubble_sort() function, we have two nested loops, the outer loop runs n times and inner loop runs n-1, n-2, ..., 2, 1 times respectively. In the worst case, when linked list is in reverse order, the time complexity would be $O(n^2)$

→ in the find_median() function, we call count_nodes() once, traverse half the list to fetch the median value, and call bubble_sort() once

∴ time complexity = $O(n + n/2 + n^2) = \boxed{O(n^2)}$

pseudocode for Q1. :

```
insert(arr, curr_size, capacity, element)
// arr: pointer to a pointer to the array storing
   the queue elements
// curr_size: no. of elements present in the array
// capacity: capacity of the array
// element: element to be inserted in the array
// return: None
    if (curr_size == capacity) {
        capacity *= 2
    new_arr = reallocate memory for arr
    indent (inside above if)
    if (new_arr == NULL)
        error
    arr = new_arr

    arr[curr_size] = element
    curr_size += 1

delete(arr, curr_size, capacity)
// same as one used in insert function
return: None
    if (curr_size = 0)
        error
    curr_size -= 1
    if (curr_size <= capacity/4)
        capacity /= 2
        new_arr = reallocate memory for arr
        if (new_arr == NULL)
            error
        arr = new_arr
```

```
struct queue {
    int head;
    int tail;
    int curr_size;
    int capacity;
    int *arr;
};
```

```
enqueue(Q, val)
// Q is the pointer to our queue
// val is the value to be inserted in the queue
// return: none
    insert(&(Q->arr), &(Q->curr_size), &(Q->capacity), val)
    Q -> tail = Q -> curr_size
```

```
dequeue(Q)
// Q is the pointer to our queue
// return: element which is removed from queue
    if (is_empty(Q))
        error
    val = Q -> arr[Q -> head]
    delete(&(Q->arr), &(Q->curr_size), &(Q->capacity))
    Q -> head = 0
    Q -> tail = Q -> curr_size - 1
    return val
```