Name: SATYAJIT GHANA CSE SEC-C Reg No: 17ETCS002159

Laboratory 2

Title of the Laboratory Exercise: Stacks and queues

Introduction and Purpose of Experiment

Stacks and queues are very important data structures used in many real time applications. This experiment introduces the development of stack and queue ADT and applying them together.

1. Aim and Objectives

Aim

- To develop stack and queue ADT and to use them for string applications

Objectives

At the end of this lab, the student will be able to

- Design and develop and use stack and demonstrate its operations
- Design and develop and use queue and demonstrate its operations

2. Pseudo Codes

```
1.  Stack – Push
    If top > MAX
       Print Stack Overflow
       End
    Else
       Set top = top + 1
       Set stack[top] = value
    End

2.  Stack – Pop
    If top  < 0
       Print Stack Underflow
       End
    Else
       Set top = top - 1
       Return stack[top + 1]
    End

3.  Stack – Display
    Set i = 0
    While i < top
       Print stack[i]
       Set i = i +1;
    End
```

```
1.  Queue – Enqueue
    If tail > MAX
        Print Queue Overflow
        End
    Else
        Set tail = tail + 1
        Set queue[tail] = value
    End

2.  Queue – Dequeue
    If head  > tail or tail == -1
        Print Queue Underflow
        End
    Else
        Set head = head + 1
        Return queue[head - 1]
    End

3.  Queue – Display
    Set i = head
    While i < tail
        Print queue[i]
        Set i = i +1;
    End
```

3. Implementation in C

```c
void push(Stack* mystack, void* data) {

    if (mystack -> top >= mystack -> MAX) {
        printf("\n*Stack Overflow Detected !*\n");
        return;
    }
    mystack -> data = realloc(mystack -> data, (mystack -> top + 2) * sizeof *(mystack -> data));
    if (mystack -> data != NULL) (mystack -> data)[(++mystack -> top)] = data;
    else printf("\n*cannot allocate memory !*\n");
}

void pop(Stack* mystack) {
    if (mystack -> top -1 < 0) {
        printf("\n*Stack Underflow detected !*\n");
        return;
    }
    mystack -> top--;
    /* Resize the data array as the elements are removed */
    mystack -> data = realloc(mystack -> data, (mystack -> top + 1) * sizeof *(mystack-> data));
    /* Print the removed data */
    printf("removed");
    ds((char*)(mystack->data)[mystack -> top+1]);
}

void display(Stack* mystack) {
    for (int i = mystack -> top  ; i >= 0 ; i--) ds(*(char**)(mystack->data + i))
}
```

Figure 1 Stack

```c
void enqueue(Queue* myqueue, void* data) {

    if (myqueue -> tail >= myqueue -> MAX) {
        printf("\n*Queue Overflow Detected !*\n");
        return;
    }
    myqueue -> data = realloc(myqueue -> data, (myqueue -> tail + 2) * sizeof *(myqueue -> data));
    if (myqueue -> data != NULL) {
        if (myqueue -> head == -1) {
            myqueue -> head = 0;
        }
        (myqueue -> data)[++myqueue -> tail] = data;
    } else {
        printf("\n*cannot allocate memory !*\n");
        return;
    }
}

void dequeue(Queue* myqueue) {
    if (myqueue -> head > myqueue -> tail) {
        printf("\n*Queue Underflow detected !*\n");
        return;
    }
    myqueue -> head ++;
    printf("removed"); ds((char*)(myqueue->data)[myqueue -> head - 1]); /* Remove the data */
}

void display(Queue* myqueue) {
    if (myqueue -> head > myqueue -> tail || myqueue -> head == -1) {
        printf("\nQueue is Empty\n");
        return;
    }
    for (int i = myqueue -> head  ; i <= myqueue -> tail ; i++) {
        ds(*(char**)(myqueue->data + i)) /* Display the data */
    }
}
```

Figure 2 Queue

4. Presentation of Results

--- STACKS USING DYNAMIC ALLOCATIONS --- 1.Push 2.Pop 3.Display 4.Exit Enter your choice : 1 Enter your data : Satyajit Ghana	--- QUEUES USING DYNAMIC ALLOCATIONS --- 1.Enqueue 2.Dequeue 3.Display 4.Exit Enter your choice : 1 Enter your data : Satyajit Ghana
--- STACKS USING DYNAMIC ALLOCATIONS --- 1.Push 2.Pop 3.Display 4.Exit Enter your choice : 3 DEBUG--**(char**)(mystack->data + i) : Satyajit Ghana*	--- QUEUES USING DYNAMIC ALLOCATIONS --- 1.Enqueue 2.Dequeue 3.Display 4.Exit Enter your choice : 3 DEBUG--**(char**)(myqueue->data + i) : Satyajit Ghana*
--- STACKS USING DYNAMIC ALLOCATIONS --- 1.Push 2.Pop 3.Display 4.Exit Enter your choice : 2 removed DEBUG--*(char*)(mystack->data)[(mystack -> top)+1] : Satyajit Ghana*	--- QUEUES USING DYNAMIC ALLOCATIONS --- 1.Enqueue 2.Dequeue 3.Display 4.Exit Enter your choice : 2 removed DEBUG--*(char*)(myqueue->data)[myqueue -> head - 1] : Satyajit Ghana*

5. Conclusions

Backtracking is used in algorithms in which there are steps along some path (state) from some starting point to some goal. In all of these cases, there are choices to be made among a number of options. We need some way to remember these decision points in case we want/need to come back and try the alternative Again, stacks can be used as part of the solution. Recursion is another, typically more favored, solution, which is actually implemented by a stack.

The simplest two search techniques are known as Depth-First Search (DFS) and Breadth-First Search (BFS). These two searches are described by looking at how the search tree (representing all the possible paths from the start) will be traversed.

Breadth-First Search with a Queue, in breadth-first search we explore all the nearest possibilities by finding all possible successors and enqueue them to a queue.