ADTS and Algorithm Group Assignment
Bsc. Computer Science, Yr. 3.1

Group Members

1. **Peaches Njenga** - SCT211-0004/2022
2. **Joseph Ngure** - SCT211-0008/2022
3. **Macharia Maurice** -SCT211-0010/2022
4. **Hilda Mwangi** - SCT211-0026/2022
5. **King'ori Florence Wangechi** - SCT211-0063/2022
6. **Theuri Bonface Karue** SCT211-0573/2022

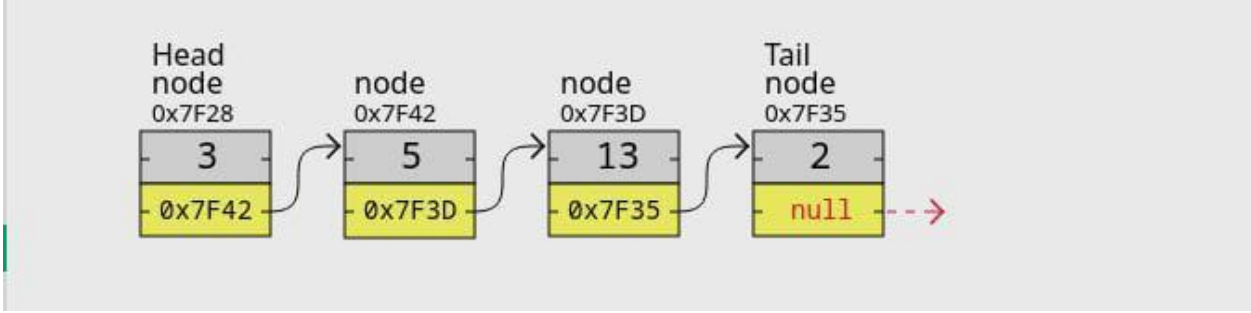
1. Describe the structure of a node in a singly linked list. What fields are typically included in this structure?

A node in a singly linked list contains two fields:

1. **Data field:**
- Stores the actual value/content of the node
- Can be of any data type eg int, string, object, etc depending on specifiifc application of linked list.
2. **Next pointer:**
- Stores the memory address or reference to the next node in the list
- Points to null/None for the last node in the list

Representation of a node in python

```
class Node:
      def __init__(self, data):
      self.data = data  # Data field
      self.next = None  # Next pointer
```



2.Implement a function to reverse a singly linked list iteratively and explain the logic behind it.

To reverse a linked list iteratively, the idea is to change the `next` pointers of each node so that they point to the previous node instead of the next node. By the end of the process, the head of the list will point to the last node, effectively reversing the list.

Pseudocode

1. Initialize three pointers:
    - `prev` as `null` (to track the previous node)
    - `current` as the `head` of the list (to track the current node being processed)
    - `next` as `null` (to temporarily store the next node)
2. Iterate through the list while `current` is not `null`:
    - Store the next node (`current.next`) in `next`
    - Reverse the `current` node's `next` pointer to point to `prev`
    - Move `prev` to `current`
    - Move `current` to `next`
3. After the loop, set the `head` of the list to `prev`, as `prev` will be the new head of the reversed list.

Java Implementation

```
class ListNode {
    int value;
    ListNode next;

    ListNode(int value) {
        this.value = value;
        this.next = null;
    }
}

public class LinkedList {
    ListNode head;

    public void reverse() {
        ListNode prev = null;
        ListNode current = head;
        ListNode next = null;
```

```
        while (current != null) {
            next = current.next;   // Store next node
            current.next = prev; // Reverse the current node's pointer
            prev = current;         // Move prev to current
            current = next;         // Move current to next
        }

        head = prev;  // Update head to the new front of the list
    }
}
```

Explanation

- **Initialize pointers**: `prev` is set to `null` to indicate the end of the reversed list, and `current` is set to `head`, which is the starting point of the list.
- **Iterate and reverse**: We use a `while` loop to traverse through each node. In each iteration:
    - We temporarily store `current.next` in `next` so that we don't lose the reference to the remaining list.
    - We reverse the `current` node's `next` pointer to point back to `prev`.
    - Then, we advance both `prev` and `current` to the next nodes in the sequence.
- **Update head**: After the loop, `prev` will point to the last node processed, which is now the first node of the reversed list. We update `head` to this new starting point.

3. Differentiate between singly, doubly, and circularly linked lists. Provide one application where each type would be most suitable.

1. **Differences**

   Singly linked list is a linear data structure in which elements are not stored in contiguous memory locations and each node has some data and a pointer to the next node. Traversing the linked list is only forward. They require less memory as compared to doubly linked lists because they only store one pointer per node.

   Doubly linked list is a more complex data structure than a singly linked list that has a pointer to the previous node as well as the next node in the sequence. This allows both forward and backward traversal of the linked list. Require more memory because of the two pointers.

   Circular linked list is a data structure where the last node has a pointer to the first node forming a loop. One can traverse the list in both directions without ever reaching an end. They have no null pointer.

2. **Applications**
   - **Singly Linked List**

     Adjacency Lists in graphs:

     In graph structures, especially for sparse graphs (these are graphs whose number of edges is smaller than the number of possible edges), singly linked lists are used to represent adjacency lists. For each vertex, a singly linked list stores its adjacent vertices. Each node in the list points to the neighboring vertex.

     This helps reduce memory use and allows quick traversal of each of the vertex's neighbors making it suitable for applications such as route mapping.

   - **Doubly Linked List**

     Game Engines for Game Objects

     In game development, doubly linked lists are useful in managing game objects and their interactions within a game engine. Each game object can be a node containing the object's data with pointers to the previous or next game object. They are useful in forward and backward movement that allows flexible management of game objects. The engine can move forward to process objects or

backward to reverse interactions. It also allows easy addition or removal of game objects.

- **Circular Linked List**

  Event Management Systems

  Circular linked lists are used in event management systems specifically those that handle repetitive events such as daily, weekly, monthly or even yearly reminders.

  The fact that circular linked lists facilitate continuous traversal is useful in these management systems where users can view upcoming events without needing to reset the list once at the end. New events can be added to the list easily and those that are no longer needed can be deleted.

4. Design a queue using two stacks

```
# Design a queue using two stacks

# Implementing a stack first using python list
class Stack:
   def __init__(self):
       self.items = []

   def push(self, item):
       self.items.append(item)

   def pop(self):
       if self.is_empty():
           return "Stack is empty"
       return self.items.pop()

   def is_empty(self):
       return len(self.items) == 0

   def peek(self):
       if self.is_empty():
           return "Stack is empty"
       return self.items[-1]

# Using the stack to implement the queue
class QueueUsingStacks:
   def __init__(self):
       # created instances of a stack
       self.enqueue_stack = Stack()
       self.dequeue_stack = Stack()

   # creating the enqueue queue method
   def enqueue(self, value):
       # using enqueue_stack
       self.enqueue_stack.push(value)

   # creating the enqueue queue method
   def dequeue(self):
       # If both stacks are empty, the queue  is empty
```

```
        if self.enqueue_stack.is_empty() and self.dequeue_stack.is_empty():
            return ": Queue is empty: Operation can not be done"

        # Transfer elements of enqueue stack to dequeue stack, while
reversing the order, then pop out the first element, effectively dequeuing
        if self.dequeue_stack.is_empty():
            while not self.enqueue_stack.is_empty():
                self.dequeue_stack.push(self.enqueue_stack.pop())

        return self.dequeue_stack.pop()

    # checking if the queue is empty
    def is_empty(self):
        return self.enqueue_stack.is_empty() and
self.dequeue_stack.is_empty()

    def peek(self):
        # check is empty
        if self.enqueue_stack.is_empty() and self.dequeue_stack.is_empty():
            return "Queue is empty"

        # Move elements back to the dequeue stack if needed
        if self.dequeue_stack.is_empty():
            while not self.enqueue_stack.is_empty():
                self.dequeue_stack.push(self.enqueue_stack.pop())

        return self.dequeue_stack.peek()


my_queue = QueueUsingStacks()

my_queue.enqueue("Orange cat")
my_queue.enqueue("Black cat")
my_queue.enqueue("Tuxedo cat")
my_queue.enqueue("Swiss Meringue cat")

print(my_queue.dequeue()) #Outputs "Orange cat"
print(my_queue.peek()) #Outputs "Black cat"

print(my_queue.peek()) #Outputs "Black cat"
```

```
print(my_queue.is_empty()) #Outputs False
```

5. How would you implement a priority queue? Describe its operations and potential applications.

A **priority queue** is an ADT where each element has an associated priority, and elements are dequeued in the order of priority rather than the order they were added. The most common and efficient way to implement a priority queue is with a **binary heap** (typically a min-heap or max-heap), which ensures efficient insertion and removal operations. A **heap** is a specialized tree-based data structure that maintains a specific order among its elements, allowing for efficient access to the “extreme” element (either the minimum or maximum, depending on the type of heap, min-heap or max-heap). Heaps are complete binary trees, meaning they’re always fully populated at every level except possibly the last level, which is filled from left to right.

Implementation of a Priority Queue Using a Min-Heap

A **min-heap** is a complete binary tree that ensures the **minimum element is always at the root**. This is ideal for a priority queue where lower-priority values have higher priority in processing.

Array Representation of a Min-Heap

A min-heap can be represented as an array with the following properties:

- For a node at index i:
    - The **left child** is at index 2i + 1.
    - The **right child** is at index 2i + 2.
    - The **parent** is at index (i - 1) / 2.

Priority Queue Operations with a Min-Heap

Using a min-heap for a priority queue provides efficient operations, outlined below:

1. **Insert (Enqueue)**:
    - Adds a new element with a specific priority.
    - The element is added to the end of the array (the last position in the heap), then "bubbled up" to restore the min-heap property if necessary.
2. **Remove (Dequeue)**:
    - Removes and returns the element with the highest priority (the minimum element in a min-heap).

    - To do this, replace the root (highest priority) with the last element, then "bubble down" to restore the min-heap property.
3. **Peek**:
    - Returns the element with the highest priority without removing it.
4. **Change Priority**:
    - Adjusts the priority of a specific element and reorders the heap.
    - This requires finding the element, updating its priority, and then either "bubbling up" or "bubbling down" as needed to maintain the min-heap property.
5. **isEmpty**:
    - Checks if the priority queue is empty.

How the Min-Heap Works

1. **Adding Priority**: When you enqueue an element, it is added with a specified priority. The element is then “bubbled up” to maintain the heap order if it has a higher priority (lower value) than its parent.
    - Example: If we add items with priorities 3, 1, and 2, the item with priority 1 will end up at the root due to its higher priority.
2. **Maintaining the Min-Heap Property**:
    - **Bubble up**: After insertion, if the new element has a higher priority (lower value) than its parent, it swaps places until the heap order is restored.
    - **Bubble down**: After removing the root, the last element is moved to the root position and “bubbled down,” swapping with the smaller of its children until it reaches its correct position.

Example: Building a Min-Heap with Priorities

Suppose we add tasks with these (priority, value) pairs:

1. (3, "Task C")
2. (1, "Task A")
3. (2, "Task B")

When added to the min-heap:

1. Add (3, "Task C") as the first element.
    - The heap is now [(3, "Task C")].
2. Add (1, "Task A").
    - The heap becomes [(3, "Task C"), (1, "Task A")].

    - We compare the newly added element with its parent. Since 1 is less than 3, they swap.
    - The heap is now [(1, "Task A"), (3, "Task C")].
3. Add (2, "Task B").
    - The heap becomes [(1, "Task A"), (3, "Task C"), (2, "Task B")].
    - No swap is needed because 2 is greater than the root (1).

After these insertions, the min-heap is:

[(1, "Task A"), (3, "Task C"), (2, "Task B")]

When we dequeue (remove the root):

- Remove (1, "Task A").
- Replace the root with the last element (2, "Task B"):
    - The heap is now [(2, "Task B"), (3, "Task C")].
- No swaps are needed, as the heap property is already maintained.

Applications of a Priority Queue Using Min-Heaps

Min-heap priority queues are useful for applications requiring efficient priority-based processing, such as:

1. **CPU Scheduling**:
    - Operating systems use priority queues to manage task scheduling, ensuring that tasks with higher priority (lower value) are executed before others.
2. **Event Simulation**:
    - In event-driven simulations, events are processed based on their scheduled time, which is efficiently managed by a priority queue.
3. **Data Compression (Huffman Coding)**:
    - Huffman coding uses priority queues to merge nodes with the lowest frequency, constructing an optimal prefix code for compression.
4. **Real-Time Systems**:
    - In systems with real-time constraints, tasks are handled based on their urgency or deadlines using a priority queue.

6. Write a function to implement a stack using an array and ensure your implementation handles overflow conditions

```c
#include <stdio.h>
#include <stdlib.h>

#define MAX_STACK_SIZE 1000000  // Maximum allowed stack size
#define SHRINK_THRESHOLD 0.25   // Shrink when stack is 25% full
#define GROWTH_FACTOR 2         // Double size when full
#define SHRINK_FACTOR 0.5      // Halve size when under threshold

typedef struct {
    int *data;
    int top;
    int maxSize;
    int initialSize;  // Store initial size for shrinking reference
} Stack;

// Error handling enum
typedef enum {
    STACK_SUCCESS,
    STACK_OVERFLOW,
    STACK_UNDERFLOW,
    STACK_MEMORY_ERROR
} StackError;

/**
* Create a new stack with error handling
*/
Stack* createStack(int initialSize, StackError* error) {
    Stack* stack = (Stack*)malloc(sizeof(Stack));
    if (!stack) {
        if (error) *error = STACK_MEMORY_ERROR;
        return NULL;
    }

    stack->data = (int*)malloc(initialSize * sizeof(int));
    if (!stack->data) {
        free(stack);
        if (error) *error = STACK_MEMORY_ERROR;
```

```
        return NULL;
    }

    stack->top = -1;
    stack->maxSize = initialSize;
    stack->initialSize = initialSize;
    if (error) *error = STACK_SUCCESS;
    return stack;
}

int isEmpty(Stack* stack) {
    return stack->top == -1;
}

/**
* Check if the stack should be resized.
* @param stack The stack to check.
* @param growth The growth factor (1 for grow, -1 for shrink).
* @return 1 if the stack should be resized, 0 otherwise.
*/
int shouldResize(Stack* stack, int growth) {
    if (growth > 0) {
        // Check for overflow when growing
        return (stack->top == stack->maxSize - 1) &&
               (stack->maxSize * GROWTH_FACTOR <= MAX_STACK_SIZE);
    } else {
        // Check for shrinking condition
        float fillPercentage = (float)(stack->top + 1) / stack->maxSize;
        return (fillPercentage < SHRINK_THRESHOLD) &&
                (stack->maxSize > stack->initialSize);
    }
}


StackError resize(Stack* stack, int newSize) {
    if (newSize > MAX_STACK_SIZE || newSize < stack->initialSize) {
        return STACK_MEMORY_ERROR;
    }

    int* newData = (int*)realloc(stack->data, newSize * sizeof(int));
```

```
    if (!newData) {
        return STACK_MEMORY_ERROR;
    }

    stack->data = newData;
    stack->maxSize = newSize;
    printf("Stack resized to %d elements\n", newSize);
    return STACK_SUCCESS;
}


StackError push(Stack* stack, int value) {
    if (!stack) return STACK_MEMORY_ERROR;

    if (stack->top == stack->maxSize - 1) {
        if (stack->maxSize >= MAX_STACK_SIZE) {
            return STACK_OVERFLOW;
        }

        int newSize = stack->maxSize * GROWTH_FACTOR;
        if (newSize > MAX_STACK_SIZE) {
            newSize = MAX_STACK_SIZE;
        }

        StackError resizeError = resize(stack, newSize);
        if (resizeError != STACK_SUCCESS) {
            return resizeError;
        }
    }

    stack->data[++stack->top] = value;
    return STACK_SUCCESS;
}


StackError pop(Stack* stack, int* value) {
    if (!stack || !value) return STACK_MEMORY_ERROR;
    if (isEmpty(stack)) return STACK_UNDERFLOW;

    *value = stack->data[stack->top--];
```

```
    // Check if we should shrink the stack
    if (shouldResize(stack, -1)) {
        int newSize = (int)(stack->maxSize * SHRINK_FACTOR);
        if (newSize < stack->initialSize) {
            newSize = stack->initialSize;
        }
        resize(stack, newSize);
    }

    return STACK_SUCCESS;
}

/**
* Free the stack and its data.
*/
void freeStack(Stack** stack) {
    if (stack && *stack) {
        free((*stack)->data);
        free(*stack);
        *stack = NULL;  // Prevent use after free
    }
}

/**
* Utility function to print stack status
*/
void printStackStatus(Stack* stack) {
    printf("\nStack Status:\n");
    printf("Current size: %d\n", stack->maxSize);
    printf("Elements: %d\n", stack->top + 1);
    printf("Fill percentage: %.2f%%\n",
           ((float)(stack->top + 1) / stack->maxSize) * 100);
}

// Error message utility
const char* getErrorMessage(StackError error) {
    switch(error) {
        case STACK_SUCCESS: return "Success";
        case STACK_OVERFLOW: return "Stack overflow";
```

```
        case STACK_UNDERFLOW: return "Stack underflow";
        case STACK_MEMORY_ERROR: return "Memory allocation error";
        default: return "Unknown error";
    }
}

int main() {
    StackError error;
    Stack* stack = createStack(5, &error);
    if (!stack) {
        printf("Failed to create stack: %s\n", getErrorMessage(error));
        return 1;
    }

    // Test pushing elements
    printf("Pushing elements...\n");
    for (int i = 1; i <= 3; i++) {
        error = push(stack, i * 10);
        if (error != STACK_SUCCESS) {
            printf("Push failed: %s\n", getErrorMessage(error));
            break;
        }
        printStackStatus(stack);
    }

    // Test popping elements
    printf("\nPopping elements...\n");
    int value;
    while ((error = pop(stack, &value)) == STACK_SUCCESS) {
        printf("Popped: %d\n", value);
        printStackStatus(stack);
    }

    if (error != STACK_UNDERFLOW) {
        printf("Unexpected error while popping: %s\n",
getErrorMessage(error));
    }

    // Clean up
    freeStack(&stack);
```

```
    if (stack == NULL) {
        printf("Stack successfully freed\n");
    }

    return 0;
}
```

7. For each of the following operations, specify which data structure (linked list, queue, or stack) is most appropriate and why:

a. Traversing in reverse order

- **Doubly linked list** is the most appropriate data structure in this case reason being:
  It has both next and previous pointers in each node that allows for bi-directional traversal. This makes it easy to traverse a list in reverse order without additional memory or complex steps

b. Ensuring first-in-first-out processing

- **Most appropriate data structure: Queue**

  Reason:

- A queue follows the First-In-First-Out (FIFO) principle
- .Elements are added to the rear of the queue and removed from the front.This guarantees that the first element added to the queue will be the first one processed.
- This property is essential for scenarios where the order of processing needs to follow the sequence in which items were added, such as task scheduling , line processing, print spooling,.

c. Tracking function calls in recursion

A **stack** is ideal for tracing function calls in recursion because:

- **LIFO Principle**: It operates on "Last In, First Out," mirroring how the latest function call is the first to complete.
- **Keeps Track of Calls**: Each time a function calls itself, it's added to the stack, and when it finishes, it's removed. This helps manage the order of active functions.

- **Traceability**: A stack clearly tracks the order of function calls, showing exactly how recursion progresses and backtracks as frames are added and removed.

8. Describe a scenario where both a queue and stack would be required to solve a problem efficiently. Outline the problem and solution approach.

"Undo and Redo" functionality in a text editor.

Problem Scenario

In a text editor, the user should be able to:

1. **Type text** (which adds new text to the editor).
2. **Undo actions** (which undoes the last text action).
3. **Redo actions** (which redoes the last undone action).

The text editor must maintain the correct sequence of actions to allow undos and redos, preserving the order in which they were performed. Here, we need a **stack** to store actions as they happen and a **queue** to manage the sequence of undone actions for redo operations.

Solution Approach

➔ **Use a Stack for Undo Operations:**
- ◆ Every time a user types text or makes a change, push that action onto an undo_stack.
- ◆ When the user performs an "Undo," pop the latest action from the undo_stack and apply it to revert the change. Push this popped action onto a redo_queue.

➔ **Use a Queue for Redo Operations:**
- ◆ When an action is undone, it moves from the undo_stack to the redo_queue.
- ◆ If the user wants to "Redo," dequeue the next action from the redo_queue and reapply it. Push this action back onto the undo_stack.

➔ **Clear the Redo Queue on New Actions:**
- ◆ If a new action is performed after undoing a previous action, the redo queue should be cleared. This prevents redoing actions that are no longer part of the current editing path.