Stack and Queue Implementation Report

1. Array Stack — Balanced Brackets Linter

Objective: Implement a Balanced Brackets Linter that validates if brackets (), {}, [] are properly matched and nested.

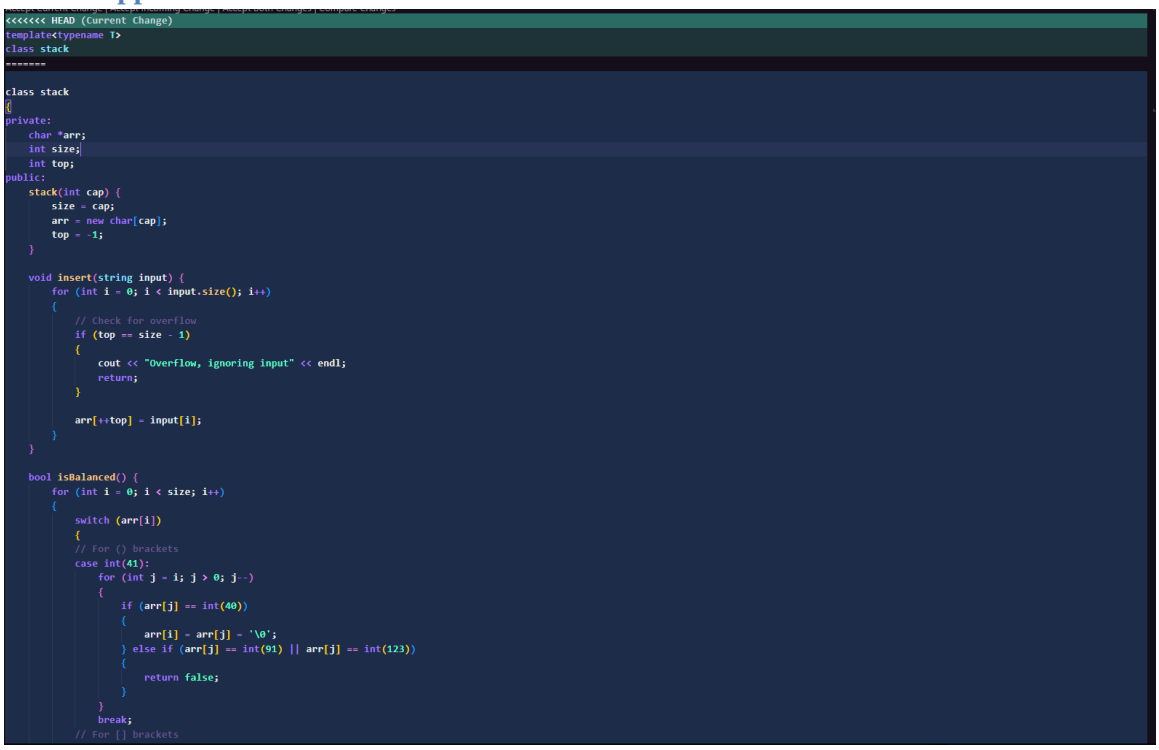
Design Explanation:

The program reads a string of characters and pushes each bracket into an array-based stack. It checks for valid pairs and order using ASCII comparisons. Non-bracket characters are ignored. The solution ensures that every opening bracket has a corresponding closing bracket of the correct type.

Edge Cases:

- Empty input → Valid
- Unmatched closing bracket → Invalid at position
- Unmatched opening bracket → Invalid
- Nested structures like [({})] → Valid; [(]) → Invalid

Code Snippet:



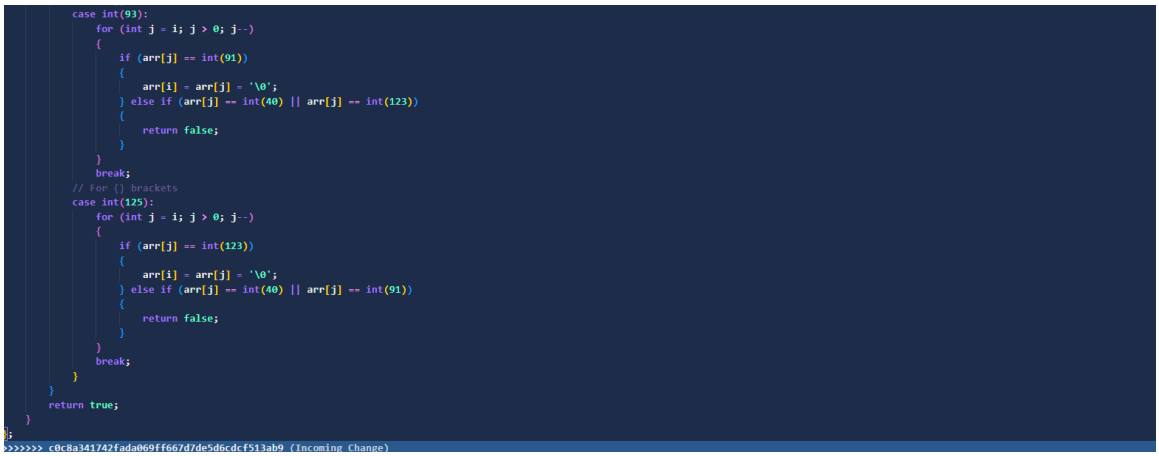
```
<<<<<< HEAD (Current Change)
template<typename T>
class stack
=======

class stack
{
private:
    char *arr;
    int size;
    int top;
public:
    stack(int cap) {
        size = cap;
        arr = new char[cap];
        top = -1;
    }

    void insert(string input) {
        for (int i = 0; i < input.size(); i++)
        {
            // Check for overflow
            if (top == size - 1)
            {
                cout << "Overflow, ignoring input" << endl;
                return;
            }

            arr[++top] = input[i];
        }
    }

    bool isBalanced() {
        for (int i = 0; i < size; i++)
        {
            switch (arr[i])
            {
            // For () brackets
            case int(41):
                for (int j = i; j > 0; j--)
                {
                    if (arr[j] == int(40))
                    {
                        arr[i] = arr[j] = '\0';
                    } else if (arr[j] == int(91) || arr[j] == int(123))
                    {
                        return false;
                    }
                }
                break;
            // For [] brackets
```



```
        case int(93):
            for (int j = i; j > 0; j--)
            {
                if (arr[j] == int(91))
                {
                    arr[i] = arr[j] = '\0';
                } else if (arr[j] == int(40) || arr[j] == int(123))
                {
                    return false;
                }
            }
            break;
        // For {} brackets
        case int(125):
            for (int j = i; j > 0; j--)
            {
                if (arr[j] == int(123))
                {
                    arr[i] = arr[j] = '\0';
                } else if (arr[j] == int(40) || arr[j] == int(91))
                {
                    return false;
                }
            }
            break;
        }
    }
    return true;
  }
};
>>>>>>> c0c8a341742fada069ff667d7de5d6cdcf513ab9 (Incoming Change)
```

2. Helpdesk Ticket Queue — Linked List Implementation

Objective: Implement a queue system for handling helpdesk tickets using a singly linked list to preserve FIFO order.

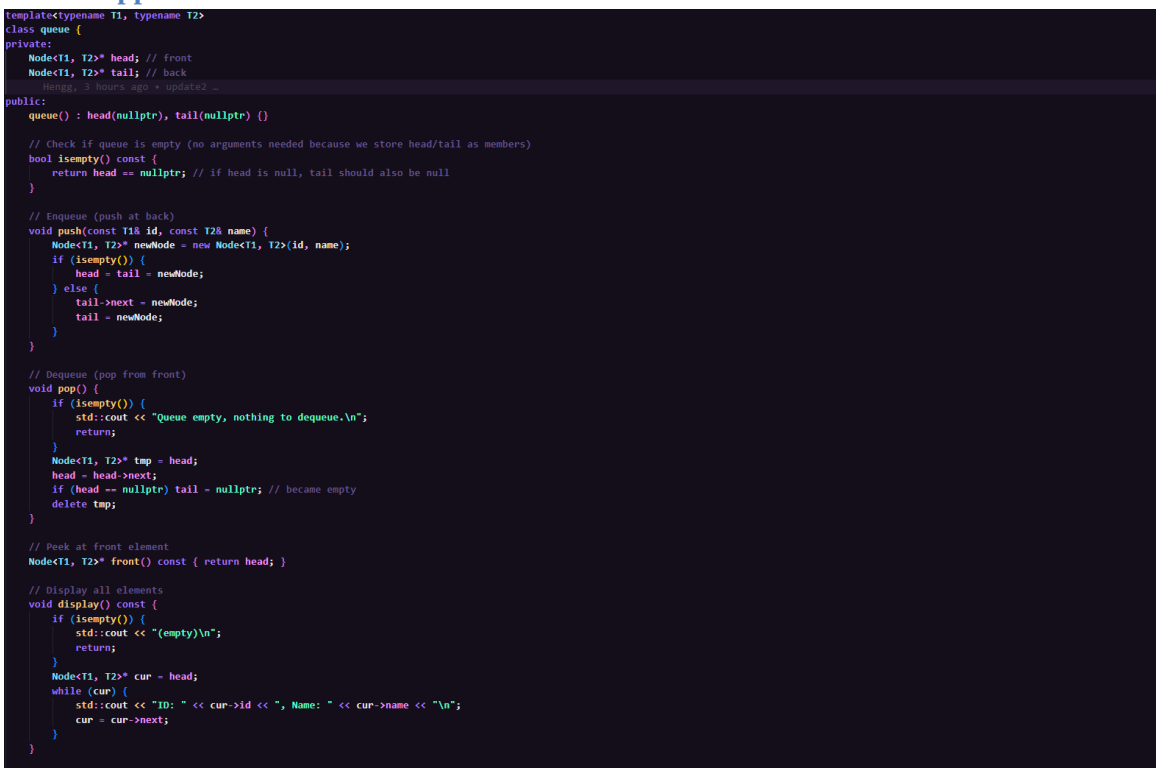
Design Explanation:

The queue stores pairs of ticket IDs and names. Each new ticket is added at the rear (tail) and processed from the front (head). The linked list allows dynamic memory usage without size limits. This structure ensures fairness and order of service.

Edge Cases:

- Dequeue on empty queue → Print message.
- Peek on empty queue → Handle safely.
- After last element is removed → Reset head and tail to null.

Code Snippet:



```
template<typename T1, typename T2>
class queue {
private:
    Node<T1, T2>* head; // front
    Node<T1, T2>* tail; // back
    Hengg, 3 hours ago • update2 …
public:
    queue() : head(nullptr), tail(nullptr) {}

    // Check if queue is empty (no arguments needed because we store head/tail as members)
    bool isempty() const {
        return head == nullptr; // if head is null, tail should also be null
    }

    // Enqueue (push at back)
    void push(const T1& id, const T2& name) {
        Node<T1, T2>* newNode = new Node<T1, T2>(id, name);
        if (isempty()) {
            head = tail = newNode;
        } else {
            tail->next = newNode;
            tail = newNode;
        }
    }

    // Dequeue (pop from front)
    void pop() {
        if (isempty()) {
            std::cout << "Queue empty, nothing to dequeue.\n";
            return;
        }
        Node<T1, T2>* tmp = head;
        head = head->next;
        if (head == nullptr) tail = nullptr; // became empty
        delete tmp;
    }

    // Peek at front element
    Node<T1, T2>* front() const { return head; }

    // Display all elements
    void display() const {
        if (isempty()) {
            std::cout << "(empty)\n";
            return;
        }
        Node<T1, T2>* cur = head;
        while (cur) {
            std::cout << "ID: " << cur->id << ", Name: " << cur->name << "\n";
            cur = cur->next;
        }
    }
```

3. Comparison between Stack and Queue

Feature	Stack	Queue
Principle	LIFO (Last In, First Out)	FIFO (First In, First Out)
Main Use	Syntax checking, function calls	Task scheduling, ticketing
Core Ops	push(), pop()	enqueue(), dequeue()
Memory Model	Array-based or Linked list	Linked list or Array

Conclusion

Both implementations showcase mastery of core data structures. The Stack ensures proper syntax checking, while the Queue ensures fairness in processing. Each model can be extended to handle real-world programming tasks efficiently.