Mars Habitat Builder Bot Simulation

Name: RAGHUL

Roll Number: ME24I1028

Date: 15 April 2025

Problem Statement

Design and simulate a Mars Habitat Builder Bot that constructs and manages habitat modules (e.g., "Wall", "Roof", "Door", etc.) using multiple data structures such as Queue, Stack, Array, and various Linked Lists. The goal is to efficiently manage construction sequencing, storage, damage tracking, and upgrade scheduling.

Key Objectives

- Simulate construction using a Queue (Part Delivery) and Stack (Bot Task Manager).

- Manage storage using a fixed-size Array with overwriting logic.

- Track damaged and repaired habitats using Singly and Doubly Linked Lists.

- Schedule urgent upgrades using a Circular Linked List.

Design Explanation

Why these data structures?

- Queue models first-in delivery of construction parts.

- Stack ensures Last-In-First-Out building logic, simulating how outer structures (e.g., vents) are added last.

- Array gives fixed habitat slots with overwrite control, ideal for constrained Martian storage.

- Singly Linked List offers fast insertion of damaged habitats.

- Doubly Linked List allows forward/backward inspection post-repair.

- Circular Linked List supports repeatable traversal for upgrade loops.

Code Logic Overview

a) Part Delivery and Bot Task Manager

```
Queue partQueue;
enqueue(partQueue, "Wall");
enqueue(partQueue, "Roof");
enqueue(partQueue, "Door");
enqueue(partQueue, "Window");
enqueue(partQueue, "Airlock");
enqueue(partQueue, "Vent");

Stack botStack;
while (!isEmpty(partQueue)) {
   part = dequeue(partQueue);
   push(botStack, part);
}

while (!isEmpty(botStack)) {
   part = pop(botStack);
   print("Bot constructs: ", part);
}
```

Why LIFO? The bot stacks parts to ensure air-sealing elements like "Vent" are installed last, following structural integrity rules—like baking a cake before icing it.

b) Assembly Storage Unit

```
string colony[5];
int start = 0;
int count = 0;

string newHabs[7] = {"Hab1", "Hab2", "Hab3", "Hab4", "Hab5", "Hab6", "Hab7"};
for (int i = 0; i < 7; i++) {
   if (count < 5) {
      colony[count++] = newHabs[i];
   } else {
      colony[start] = newHabs[i];
      start = (start + 1) % 5;
   }
}
```

Why overwrite oldest? As new settlers arrive, older habitats are vacated or refurbished. The circular overwrite ensures continual usage of limited space.

c) Damaged Habitat Tracker

```
insertSingly(damagedList, "Hab3");
insertSingly(damagedList, "Hab6");

deleteSingly(damagedList, "Hab3");
insertDoubly(repairedList, "Hab3");

traverseForward(repairedList);
traverseBackward(repairedList);
```

Damage and Fix Example: "Hab3" had its door cracked during a storm. Bots patched it using polymer sealant and confirmed it passed air-tightness tests before inspection.

d) Priority Upgrades

```
insertCircular(priorityList, "Hab1");
insertCircular(priorityList, "Hab4");

for (int i = 0; i < 2; i++) {
   traverseOnce(priorityList);
}
```

Upgrade Example: "Hab1" is upgraded with next-gen solar panels, while "Hab4" gets a greenhouse dome to support sustainable oxygen and food cycles.

Key Variables and Functions

- enqueue(), dequeue() – Queue operations for parts.

- push(), pop() – Stack operations for construction order.

- insertSingly(), deleteSingly() – Singly linked list for damage tracking.

- insertDoubly(), traverseForward(), traverseBackward() – Repair inspection.

- insertCircular(), traverseOnce() – Upgrade scheduler.