1600. Throne Inheritance

Problem Description

In the realm of a kingdom, we have a continuous line of succession from the king down to the various members of the royal family. This order of inheritance begins with the king and is followed by his children, grandchildren, and so on, in order of their ages. The problem involves managing this order of inheritance, taking into account the birth of new royal members and the death of existing ones. The challenge is to maintain a dynamic list of the succession line, which can change when a new child is born or when someone dies.

Intuition

The primary approach to solving this problem involves simulating the family tree structure with appropriate data structures and then conducting depth-first search (DFS) traversals to generate the current order of inheritance.

Let's break down the solution into the following steps:

1. **Data Structure**: We need to efficiently represent the family tree to allow for the addition of new children (via births) and to keep track of deaths. We use a graph represented by a dictionary (`self.g`) where every person is a key, and their children form a list (the values).
2. **Births**: When a child is born, we add the child to the graph by appending the child's name to the parent's list in the dictionary. This operation corresponds to adding edges to the family tree graph.
3. **Deaths**: We keep track of deaths in a set (`self.dead`). We don't remove the person from the graph because their children still need to be in the inheritance order.
4. **Order of Inheritance**: To find the current inheritance order, we use DFS starting from the king. In this traversal, we add a person to the current order only if they are not marked as dead. Through DFS, the children of each dead person are visited in the order they were added, maintaining the age-based priority.
5. **Exclusion of Dead People**: Since we want the order to contain only living members, we check the set of dead people during DFS and only include those who are not in this set.

By following this approach, we can dynamically adjust the inheritance order as births and deaths are recorded, thus maintaining an up-to-date succession line.

Solution Approach

The solution to the inheritance problem is built around several key concepts in computer science, namely data structures, depth-first search, and class design.

Let's dive into each of these points:

1. **Data Structures**: We create three main attributes within the `ThroneInheritance` class:
   - A graph `self.g`, represented as a dictionary, which captures the parent-child relationships.
   - A set `self.dead` to keep track of the deceased family members.
   - A variable `self.king` to store the name of the king, which is the root of our graph.
2. **Class Methods**: Three methods alter the state of the inheritance line:
   - `birth` receives a parent name and a child name. The child's name is appended to the parent's list of children in the graph.
   - `death` marks a family member as deceased by adding their name to the `self.dead` set.
   - `getInheritanceOrder` invokes a DFS traversal method to calculate the current inheritance order.
3. **Depth-First Search (DFS)**: We perform a DFS starting from the king every time `getInheritanceOrder` is called. Here's the breakdown of the DFS method used:
   - A helper function `dfs` is defined within `getInheritanceOrder` for recursive traversal.
   - Each time the DFS visits a person, it checks if they are alive (not in `self.dead`). If so, the person is added to the `ans` list, which represents the current inheritance order.
   - The DFS continues recursively through the children (in the order they were added to `self.g`), ensuring the age-based succession is preserved.

This implementation allows us to efficiently manage the order of inheritance dynamically as births add to the `self.g` graph and deaths are recorded in `self.dead`. The `getInheritanceOrder` function can be called at any time to provide a snapshot of the living members in the line of succession, excluding those that have passed away.

Example Walkthrough

Consider a kingdom with the following initial succession line: King (George), his children: *oldest* (John), and middle (Liam), and youngest (Claire).

In this structure, we have:

- King George at the root.
- John, Liam, and Claire as the children of George, listed in the order of their ages.

Let's set up the `ThroneInheritance` at this point:

1. `self.g` = {"George": ["John", "Liam", "Claire"]}
2. `self.dead` = set()
3. `self.king` = "George"

Now let's walk through a series of events and how the data structures would change:

Event 1: John has a child named William.

- We execute the `birth` method for John and William.
- `self.g` becomes { "George": ["John", "Liam", "Claire"], John": ["William"] }.

Event 2: Claire gives birth to a child named Emma.

- We call the `birth` method for Claire and Emma.
- `self.g` is updated to { "George": ["John", "Liam", "Claire"], John": ["William"], "Claire": ["Emma"] }.

Event 3: Liam passes away.

- `death` method is called for Liam; so Liam is added to `self.dead`.
- `self.dead` becomes { "Liam" }.
- Note that Liam is not removed from `self.g` because it will affect the children's structure (if he had any).

Current Inheritance Order: At this time, if we call `getInheritanceOrder`, the DFS would start from George, move to John (as he is the oldest child), and then to William (child of John), skip Liam because he is in the `self.dead` set, and continue with Claire, and then Emma.

- DFS order: George → John → William → Liam (skipped, he's deceased) → Claire → Emma
- The `ans` list would be ["George", "John", "William", "Claire", "Emma"].

This way, we can see how the dynamic list of succession maintains the correct order throughout the births and deaths within the royal family, representing the living members in the correct order of precedence at all times.

Python Solution

```python
from collections import defaultdict
from typing import List

class ThroneInheritance:
    def __init__(self, king_name: str):
        # Initialize a graph to represent the family tree
        self.family_tree = defaultdict(list)

        # Set to keep track of the deceased individuals
        self.deceased = set()

        # Store the king's name
        self.king = king_name

    def birth(self, parent_name: str, child_name: str) -> None:
        # Add a child under the parent in the family tree
        self.family_tree[parent_name].append(child_name)

    def death(self, name: str) -> None:
        # Mark a family member as deceased
        self.deceased.add(name)

    def get_inheritance_order(self) -> List[str]:
        # Helper function to perform depth-first search (DFS)
        def dfs(current_member):
            # Append the current member to the order if they are not deceased
            if current_member not in self.deceased:
                inheritance_order.append(current_member)
            # Continue DFS for each child of the current member
            for child in self.family_tree[current_member]:
                dfs(child)

        # List to store the current order of inheritance
        inheritance_order = []
        # Start DFS from the king
        dfs(self.king)
        # Return the inheritance order
        return inheritance_order

# Example of using the ThroneInheritance class
# throne_inheritance = ThroneInheritance("King")
# throne_inheritance.birth("King", "Andy")
# inheritance_order = throne_inheritance.get_inheritance_order()
```

Java Solution

```java
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;

class ThroneInheritance {

    // Graph to hold the family tree, where each key is a parent and the value is a list of children
    private Map<String, List<String>> familyTree = new HashMap<>();

    // Set to hold all the deceased family members
    private Set<String> deceasedMembers = new HashSet<>();

    // List to hold the order of inheritance
    private List<String> inheritanceOrder;

    // The name of the reigning King or the root of the family tree
    private String king;

    // Constructor initializes the ThroneInheritance with the name of the king
    public ThroneInheritance(String kingName) {
        king = kingName;
    }

    // Method to record the birth of a child, assigning the child to a parent in the family tree
    public void birth(String parentName, String childName) {
        familyTree.computeIfAbsent(parentName, k -> new ArrayList<>()).add(childName);
    }

    // Method to record the death of a family member by adding them to the set of deceased members
    public void death(String name) {
        deceasedMembers.add(name);
    }

    // Method to return the current order of inheritance, skipping deceased members
    public List<String> getInheritanceOrder() {
        inheritanceOrder = new ArrayList<>();
        // Start the depth-first search with the king
        dfs(king);
        return inheritanceOrder;
    }

    // Helper method to perform a Depth-First Search on the family tree starting from a given family member
    private void dfs(String familyMember) {
        // If the current member is not deceased, add them to the inheritance order
        if (!deceasedMembers.contains(familyMember)) {
            inheritanceOrder.add(familyMember);
        }
        // Recurse on all children of the current member
        for (String child : familyTree.getOrDefault(familyMember, Collections.emptyList())) {
            dfs(child);
        }
    }
}

// Usage example:
// ThroneInheritance inheritance = new ThroneInheritance("king");
// inheritance.birth("king", "andy");
// inheritance.birth("king", "bob");
// inheritance.birth("andy", "matthew");
// inheritance.death("bob");
// List<String> order = inheritance.getInheritanceOrder(); // Gets the inheritance order
```

C++ Solution

```cpp
#include <string>
#include <vector>
#include <unordered_map>
#include <unordered_set>
using namespace std;

class ThroneInheritance {
private:
    unordered_map<string, vector<string>> familyTree; // Holds the family tree as an adjacency list
    unordered_set<string> deceased; // Keeps track of deceased family members
    string monarch; // The name of the current king or queen
    vector<string> inheritanceOrder; // Used to store the computed inheritance order

public:
    // Constructor initializes the family tree with the reigning monarch.
    ThroneInheritance(string kingName) : monarch(kingName) {}

    // Function to record the birth of a child
    // parentName: The name of the parent in the family tree
    // childName: The child's name to be added under the parent
    void birth(string parentName, string childName) {
        familyTree[parentName].push_back(childName);
    }

    // Function to record the death of a family member
    // name: The name of the deceased member
    void death(string name) {
        deceased.insert(name); // Add the member to the deceased set
    }

    // Function to get the current order of inheritance which represents the
    // line of succession to the throne, filters out deceased members.
    vector<string> getInheritanceOrder() {
        inheritanceOrder.clear(); // Clear the list from the previous order
        dfs(monarch); // Perform DFS starting from the monarch to build the order
        return inheritanceOrder; // Return the computed order
    }

    // Helper function that performs a depth-first search on the family tree
    // starting from the given family member to determine the line of succession.
    void dfs(string familyMember) {
        if (!deceased.count(familyMember)) { // If the person is not deceased, add to the inheritance order
            inheritanceOrder.push_back(familyMember);
        }
        const vector<string>& children = familyTree[familyMember]; // Get the children or an empty array if there are none
        for (const string& child : children) {
            dfs(child); // Recursively search for each child
        }
    }
};

// Example of using the ThroneInheritance class:
/*
* ThroneInheritance obj = new ThroneInheritance(kingName);
* obj->birth(parentName, childName);
* obj->death(name);
* vector<string> order = obj->getInheritanceOrder();
* delete obj; // Don't forget to free allocated memory.
*/
```

Typescript Solution

```typescript
// Define an adjacency list to hold the family tree where each parent maps to a list of children
const familyTree: Record<string, string[]> = {};

// Store the set of deceased family members
const deceased: Set<string> = new Set();

// The name of the current monarch
let monarch: string;

// Array to store the computed inheritance order
let inheritanceOrder: string[] = [];

// Initializes the family tree with the reigning monarch
function ThroneInheritance(kingName: string): void {
    monarch = kingName;
}

// Records the birth of a child
function birth(parentName: string, childName: string): void {
    if (!familyTree[parentName]) {
        familyTree[parentName] = [];
    }
    familyTree[parentName].push(childName);
}

// Records the death of a family member
function death(name: string): void {
    deceased.add(name); // Add the member to the deceased set
}

// Gets the current order of inheritance, representing the line of succession to the throne
// It skips the deceased members and builds the order using DFS
function getInheritanceOrder(): string[] {
    inheritanceOrder = []; // Clear the previous order
    dfs(monarch); // Recursively compute the order starting from the monarch
    return inheritanceOrder;
}

// Performs a depth-first search on the family tree from the given family member
// to determine the line of succession
function dfs(familyMember: string): void {
    if (!deceased.has(familyMember)) { // If the person is not deceased, add to the inheritance order
        inheritanceOrder.push(familyMember);
    }
    const children = familyTree[familyMember] || []; // Get the children or an empty array if there are none
    for (const child of children) {
        dfs(child); // Recursively search for each child
    }
}

// Examples of using the above global functions and variables:
// initializeThroneInheritance('King');
// birth('King', 'Andy');
// birth('King', 'Bob');
// birth('Andy', 'Matthew');
// birth('Bob', 'Alex');
// birth('Bob', 'Asha');
// death('Bob');
// console.log(getInheritanceOrder());
```

Time and Space Complexity

`__init__` method

The `__init__` method has a time complexity of $O(1)$ as it simply initializes the data structures: a graph (`self.g` as a defaultdict of lists), a set (`self.dead`), and assigns the `self.king` variable. The space complexity of the `__init__` method is also $O(1)$ assuming the input `kingName` size does not contribute to the space complexity as it's a single value.

`birth` method

The `birth` method has a time complexity of $O(1)$ as it just appends the child to the parent's children list in the graph. The space complexity is $O(1)$ as well because it just adds one more element to the existing data structure.

`death` method

The `death` method has a time complexity of $O(1)$ as it adds a name to the `self.dead` set. The space complexity is $O(1)$ due to a single insertion into the set.

`getInheritanceOrder` method

The `getInheritanceOrder` method has a time complexity of $O(N)$ where N is the total number of people in the inheritance order (both alive and dead). This is because it performs a depth-first search (DFS) over the entire family tree, visiting each person once. The space complexity of the `getInheritanceOrder` method can be considered to be $O(N)$ for the DFS recursion stack in the worst case (assuming the family tree is heavily unbalanced and resembles a linked list), plus $O(N)$ for the `ans` list where N is the number of alive people. So, the total space complexity is $O(N)$.

Overall, the main performance concern in the code is related to how the DFS might behave on a very unbalanced family tree — in that case, the recursion could be a problem, and iterative approaches might need to be considered to keep the stack depth under control.