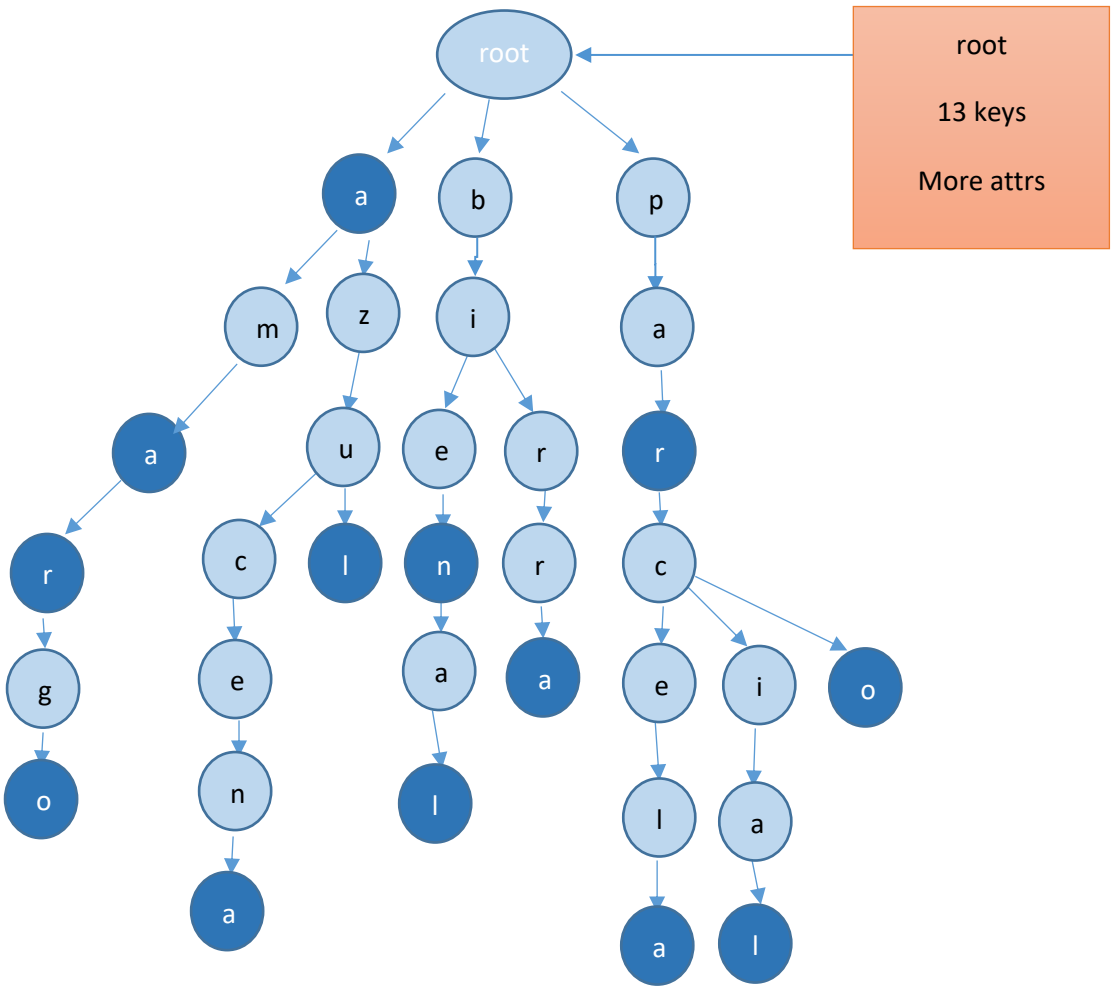
1. **Trie:** Un **`Trie`** es una estructura de datos eficiente para la recuperación de información (information re**`trie`**val). Si almacenáramos claves (por ejemplo palabras) en un árbol de búsqueda binario (diccionario binario), bien balanceado, necesitaríamos un tiempo proporcional a M*log N, donde M es la longitud máxima de las claves y N el número de claves en el árbol. Utilizando un **`trie`**, se puede buscar la clave en un tiempo proporcional a M (Orden M). Cada nodo de un **`trie`** puede contener múltiples ramificaciones. Cada ramificación representa una nueva parte de la clave (un nuevo carácter en el caso de que las claves sean palabras) y en cada nodo se indica a través de un atributo si el nodo termina de definir una clave o es solo un prefix para claves que terminan en nodos inferiores (un nodo que termina de definir una clave puede también ser prefix para otras).



En la figura se notan en color oscuro los nodos que terminan de definir una clave.

Si se definen las siguientes estructuras de datos para manejar un **`Trie`** de strings.

```
typedef enum { FALSE, TRUE } BOOL;

#define ALPHABET_SIZE 26

typedef struct trie_node {
    BOOL defineKey;           // true/false si define o no una clave
    struct trie_node *children[ALPHABET_SIZE];
} Trie_node_t;

typedef struct {
    Trie_node_t *root;        // contenedor real de datos
    unsigned int numKeys;     // número de claves en el trie
} Trie_t;
```

Notar que con esta representación no se guardan los caracteres (que en la figura anterior parecerían estar incluidos), estos están implícitos en la topología de la estructura: un nodo dado define el prefix de todos sus hijos, si **`children[i]`** es no vacío, **`children[i]`** representa al carácter **`i`** siguiendo al prefix.

Se solicita implementar las siguientes funciones:

```
// crea un trie vacio y lo retorna
Trie_t *trieCreate();

// inserta una nueva clave si no está presente o
// lo marca con defineKey si ya existía como prefix de otras claves
void trieInsertKey(Trie_t *pTrie, const char *key);


// retorna verdadero/falso indicando si la clave existe.
BOOL trieExistKey(Trie_t *pTrie, const char *key);

// Chequea la consistencia entre la cantidad de nodos que dice el
// trie y la cantidad de nodos que definen key.
// Retorna TRUE/FALSE indicando si hay o nó consistencia.
BOOL trieCheck(Trie_t *pTrie);

// Retorna la cantidad de claves que tienen un prefix determinado.
int trieNumWordsWithPrefix(Trie_t *pTrie, const char *prefix);

// destruye un trie liberando todos los recursos alocados
void trieDestroy(Trie_t *pTrie);
```

2. **Tabla de Hash:** Se propone indexar un gran número de personas (por ejemplo 10.000 individuos) para realizar búsquedas eficientes por número de documento (DNI). Las personas están dentro de una tabla maestra (vector de personas). Para ello se propone utilizar una tabla de Hash, utilizando como función de hash una que directamente retorne el número de documento, modulado en la dimensión de la tabla. Para el manejo de las colisiones, se decide utilizar listas simplemente ligadas en donde se colocarían todas las entradas que colisionan en su valor de hash. Una entrada no utilizada en la tabla de Hash

debería contener una lista vacía. Con estas premisas, las estructuras de datos a utilizar serían las siguientes:

```
#define NOMBRE_SZ 32
#define DOMICILIO_SZ 64

typedef struct {
    char nombre[NOMBRE_SZ];
    unsigned dni;
    char domicilio[DOMICILIO_SZ];
    // ...
} Persona_t;

typedef struct HashEntry_str {
    unsigned dni; // clave indexada
    int nroRegistro; // nro de registro en la tabla maestra
    struct HashEntry_str *next;
} *HashEntry_t;

typedef struct {
    unsigned numEntries;
    HashEntry_t *entries;
} HashTable_t;
```

Se solicita implementar las siguientes funciones (y toda aquella otra función que estime necesaria para cumplir el objetivo):

a. Función que crea y retorna una tabla de hash vacía. El prototipo de la función es:

```
HashTable_t *HashTableCreate(unsigned numEntries);
```

b. Función que indexa un conjunto de personas (tabla maestra). Prototipo:

```
void HashTableFill(HashTable_t *ht, Persona_t *personas,
unsigned nPers);
```

c. Función que busca una persona con un determinado nro de documento. La función debe retornar el número de registro en la tabla maestra, o -1 para indicar la ausencia del mismo. Prototipo:

```
int HashTableSearch(HashTable_t *ht, unsigned dni);
```

d. Función que retorna un indicador de eficiencia de búsqueda, definido como 1 dividido por la longitud media de las listas de colisiones no vacías. Prototipo:

```
double HashTableAverageColisionLength(HashTable_t *ht);
```

e. Función que destruye una table de hash liberando todos sus recursos asociados. Prototipo:

```
void HashTableDestroy(HashTable_t *ht);
```