UNIVERSIDAD PRIVADA BOLIVIANA DIRECCIÓN DE PREGRADO

FACULTAD DE INGENIERÍAS Y ARQUITECTURA

COMPLEJIDAD ALGORÍTMICA

Tarea 3 – Algorítmica II

JUAN CLAUDIO CARRASCO TAPIA

LA PAZ – BOLIVIA

2022

Complejidad algorítmica Trie:

- Método Insert():

```
void insertWord(string word) {
    node *currentNode =  trie;
    for (int i = 0; i< word.length(); i++) {
        int character = word[i] - 'a';
        if(currentNode->children[character] == NULL ) {
            currentNode->children[character] = new node();
        }
        currentNode = currentNode->children[character];
        currentNode->currentCharacter = word[i];
    }
    currentNode->isWord = true;
}
```

Complejidad O(n) → (for i to n)

- Método Search():

```
bool searchWord(string word) {

    node *currentNode =  trie;

    for (int i = 0; i< word.length(); i++) {

        int character = word[i] - 'a';
        if(currentNode->children[character] == NULL ) {
            return false;
        }
        currentNode = currentNode->children[character];
    }
    return currentNode->isWord;
}
```

Complejidad O(n) → (for i to n)

- Método Delete():

```
void deleteWord(string word) {
    if(searchWord(word)) {
        node *currentNode = trie;
        int tempDepth = 0;
        int eliminacionDeNodo = 0;
        for (int i = 0; i< word.length(); i++) {
            int character = word[i] - 'a';
            if(currentNode->children[character]->isWord && word.length() != i+1) {
                eliminacionDeNodo+= tempDepth;
            }
            currentNode = currentNode->children[character];
            tempDepth++;
        }
        currentNode->isWord = false;
        currentNode = trie;
        for (int i = 0; i< eliminacionDeNodo; i++) {
            int character = word[i] - 'a';
            currentNode = currentNode->children[character];
        }
        currentNode = NULL;
        delete currentNode;
        cout << "Se elimino:" << word << " del array" << endl;
```

Complejidad O(n)+O(m) → 2 For Loops

Complejidad Algorítmica BIT:

- Método Update():

```
void update(int posicion, int valor ) {
    for(;posicion <= tamanhoVector ;posicion += posicion&-posicion) {
        BIT[posicion] *= valor;
    }
}
```

Complejidad O(log(n)) → peor caso para un árbol binario indexado

- Método Query():

```
int query(int posicion){ // F(3)
    int result = 0 ;
    for(;posicion > 0 ;posicion -= posicion&-posicion) {
        result += BIT[posicion];
    }
    return result;
}
```

Complejidad O(log(n)) → peor caso para un árbol binario indexado

Complejidad Algorítmica Union Find:

- Método init():

```
void init() {
    for(int i=0;  i<= n; i++) {
        parent[i] = i;
        rango[i] = 0;
        cont[i] = 1;
    }
}
```

Comlejidad O(n)

- Método Find:

```
int find(int x) {
    if(x == parent[x]) {
        return x;
    }
    else {
        parent[x] = find(parent[x]);

        return parent[x];
    }
}
```

Complejidad O(n)→ Máximo n padres hasta encontrar a la raíz.

- Método UnionRango:

```
void unionRango(int x,int y) {
    int xRaiz = find(x);
    int yRaiz = find(y);
    if(rango[xRaiz] > rango[yRaiz]) {
        parent[yRaiz] = xRaiz;
        cont[yRaiz] += cont[xRaiz];

    } else {
        parent[xRaiz] = yRaiz;
        cont[xRaiz] += cont[yRaiz];
        if(rango[xRaiz] == rango[yRaiz]) {
            rango[yRaiz]++;
        }
    }
}
```

Complejidad constante → No hay bucles y la búsqueda de padres es simple por el uso de rango.

- Método Unir:

```
void unir(int vertice1, int vertice2) {
    padres[vertice2] = vertice1;
}
```

Complejidad constante