UNIVERSIDAD DE CONCEPCIÓN
Departamento de Ingeniería Informática y
Ciencias de la Computación

Estructuras de Datos (2022-1)

Proyecto 1
Implementación Trie

Profesor: Alexander Irribarra
Ayudantes: Leonardo Aravena
Diego Gatica
Vicente Lermanda
Alumno: Jaime Ansorena Carrasco

10 de Junio de 2022

1. Introducción

El almacenamiento y la búsqueda eficiente de palabras es una tarea importante en ciencias de la computación. Para esto, esta definido el Trie, una estructura de datos tipo árbol, popular para la búsqueda de palabras por su complejidad lineal. Es parte básica e importante de varias aplicaciones como la recuperación de información, el procesamiento del lenguaje natural de bases de datos, compiladores y redes informáticas.

Cada palabra comienza en la raíz del árbol y termina en un nodo terminal, que además posee una variable para indicar el fin de cada palabra. De esta manera, los hijos de un nodo representan las distintas posibilidades de caracteres que prosiguen al carácter representado por el nodo padre.

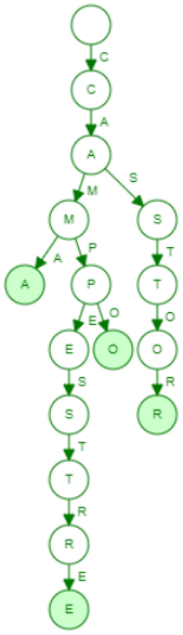

Figura 1: Ejemplo de un trie

Para el siguiente informe se realizaran dos implementaciones de esta estructura de datos. Una basada en arreglos de punteros a los distintos nodos y la segunda basada en map o diccionarios, que almacenan los punteros a los distintos nodos.

Se realizaran tanto análisis teóricos como experimentales de las soluciones propuestas.

2. Soluciones Propuestas

1. Trie Array

La primera implementación corresponde a un TrieArray. Esta basada en un struct node y contiene un arreglo de punteros a node, una variable bool, para indicar el fin de cada palabra, y un entero que almacena la frecuencia con que se ingresa cada palabra.

```
struct node{
  node *children[26]; // arreglo punteros
  bool finPalabra; // fin de cada palabra
  int contador = 0; // frecuencia
};
```

Cada arreglo de punteros es inicializado en NULL y cada posición del arreglo codifica la letra a ingresar, almacenando el puntero al siguiente nodo.

```
TrieArray::TrieArray(){ // constructor de la raiz del trie
  struct node *aux = new node;
  for (int i = 0; i < 26; ++i) aux->children[i] = NULL;
  aux->finPalabra = false;
  root = aux;
}
```

Para ejemplificar esta estructura:

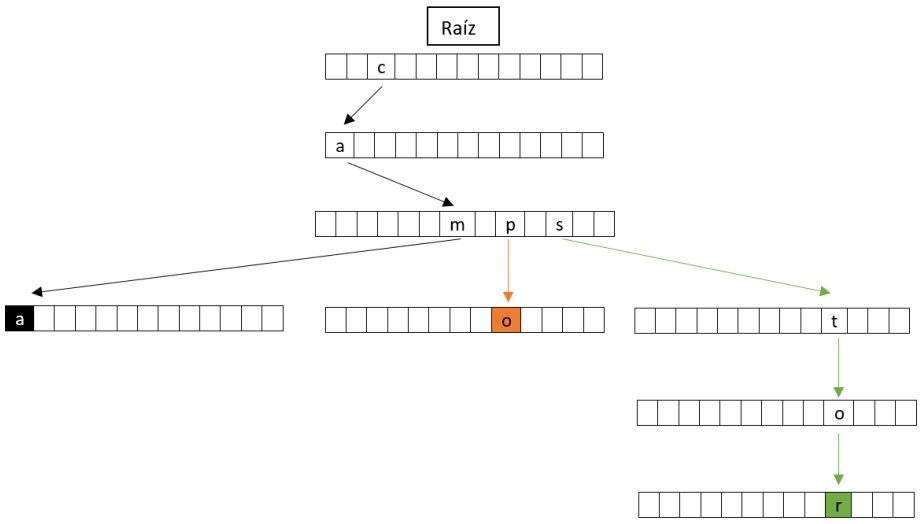

Figura 2: Trie Array

2. Trie Map

La segunda implementación corresponde a un Trie Map. En este caso el arreglo de punteros es cambiado por un diccionario o map, donde la llave es el carácter de cada palabra y su valor es un puntero al siguiente nodo.

```
struct nodeMap{
   std::map<char,nodeMap*> m;
   bool finPalabra;
   int contador = 0;
};
```

En este caso no es necesario tener punteros a NULL ya que podemos buscar en el map si existe o no el carácter correspondiente.

```
TrieMap::TrieMap(){ // constructor de la raiz del trie
   struct nodeMap *aux = new nodeMap;
   aux->finPalabra = false;
   root = aux;
}
```

Para el mismo caso anterior:

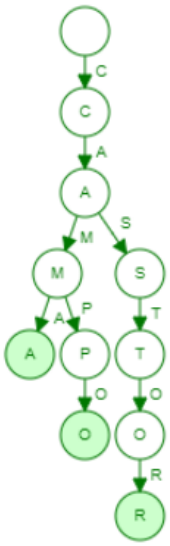

Figura 3: Trie Map

Los detalles de cada implementación se verán en la sección siguiente.

3. Detalles de la implementación

1. Trie Array

   a) TrieArray::insert

   El algoritmo para insertar las cadenas al Trie es relativamente simple. Para cada posición de la string s, buscamos su posición en el arreglo (en código ASCII), restándole 97, y creamos un nuevo nuevo que es apuntado por la posición del arreglo. Si el puntero ya existe, solo avanzamos al siguiente nodo.

```
void TrieArray::insert(const string &s){ // insertar las cadenas al trie
    node *aux = root; // raiz
    for (int i = 0; i < s.length(); ++i){ // iteracion sobre el string
        int pos = s[i]-97;
        if(aux->children[pos] == NULL){ // si no existe el puntero
            node *nuevo = new node;
            aux->children[pos] = nuevo;
        }
        aux = aux->children[pos]; // avanzar al siguiente nodo
    }
    aux->finPalabra = true; // marcar el fin del string
    aux->contador++; // aumentar la frecuencia
}
```

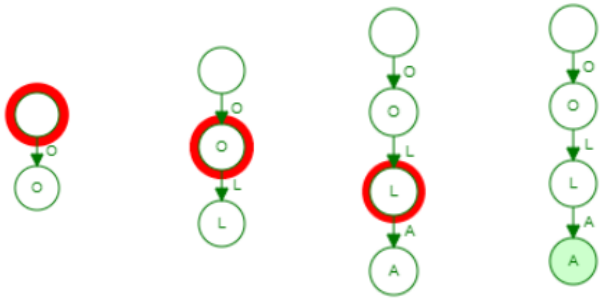

Figura 4: Ejemplo de inserción en el Trie

b) TrieArray::search

Para la función search, el algoritmo es similar al de insert. Iteramos por la string s y verificamos que la posición correspondiente en el arreglo no sea NULL. En caso de serlo retornamos false. Al final verificamos que la variable finPalabra del último nodo sea true.

```
bool TrieArray::search(const string &s){ // buscar la cadena dada en el trie
   node *aux = root;
   for (int i = 0; i < s.length(); ++i){
      int pos = s[i]-97;
      if(aux->children[pos] != NULL) aux = aux->children[pos];
      else return false;
   }
   if(aux->finPalabra) return true;
   else return false;
}
```

c) TrieArray::remove

Para la función remove iteramos por la string s, retornando false en caso que no este contenida. A medida que avanzamos marcamos los nodos que en su arreglo contengan mas de un puntero. Esto se realiza mediante la implementación de la función isEmpty(). Posteriormente verificamos si el ultimo nodo posee algún puntero. En caso que así sea solo cambiamos a false la variable finPalabra y reiniciamos su contador. En el último caso que no posea punteros, iteramos desde la posición lastNode hasta el final haciendo delete en cada nodo.

```
bool TrieArray::remove(const string &s){ // borra la cadena del trie
   node *aux = root;
   node *lastNode = root; // ultimo nodo
   int posLastNode = 0; // posicion del ultimo nodo
   // itera hasta encontrar el ultimo nodo
   for (int i = 0; i < s.length(); ++i){
      int pos = s[i]-97;
      if(aux->children[pos] != NULL) {
          // marcamos el ultimo nodo con mas de 1 puntero
         if(!isEmpty(aux,pos)) {
            lastNode = aux;
            posLastNode = i;
         }
         aux = aux->children[pos];
      }
      else return false;
   }
   if(!aux->finPalabra) return false;

   if(!isEmpty(aux,-1)) { /* si el ultimo nodo tiene punteros */
      aux->finPalabra = false;
      aux->contador = 0;
```

```
        return true;}
    // borra a partir del ultimo nodo con punteros
    for (int i = posLastNode; i < s.length(); ++i){
        node *erase = lastNode;
        int pos = s[i]-97;
        lastNode = lastNode->children[pos];
        if(i == posLastNode) erase->children[pos] = NULL;
        else delete[] erase;
    }
    return true;
}
```

```
bool isEmpty(node* aux, int pos) {
    for (int i = 0; i < 26; ++i){
        if (aux->children[i] != NULL && i != pos) return false;
    }
    return true;
}
```

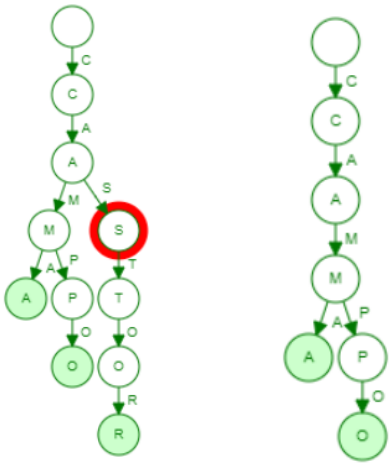

Figura 5: Ejemplo de eliminación en el Trie

d) TrieArray::getAll

La funcion getAll recorre recursivamente el árbol en preOrder sumando a una string vacía cada carácter. Cuando encuentra la variable finPalabra agrega la string al vector v.

```
// devuelve un vector con todas las cadenas del trie
vector<string> TrieArray::getAll(){
   vector<string> v;
   string s;
   node *aux = root;
   getStrings(v,aux,s);
   return v;
}
```

```
// funcion recursiva que agrega al vector v las strings contenidas en el trie
void getStrings(vector<string> &v, node* aux, string s){
   if(aux->finPalabra) v.push_back(s);
   for (int i = 0; i < 26; ++i){
      if(aux->children[i] != NULL){
         char c = i+97;
         getStrings(v,aux->children[i],s+c);
      }
   }
}
```

e) TrieArray::getKTopMatches

El algoritmo de getKtopMatches es similar al de getAll, con la diferencia que iteramos la string dada hasta el ultimo nodo y recorremos el árbol a partir de ese nodo, agregando a un vector<string, int>la palabra y su frecuencia. Luego llamamos a la funcion sort, implementada en la STL, ordenando de mayor a menor, con el uso de un comparador sortInt y copiamos a un nuevo vector según el número requerido.

```
// devuelve un vector con las n cadenas de mayor frecuencia
vector<string> TrieArray::getKTopMatches(const string &s, int n){
   vector<pair<string,int>> vec;
   node *aux = root;
   for (int i = 0; i < s.length(); ++i){
      int pos = s[i]-97;
      if(aux->children[pos] != NULL) aux = aux->children[pos];
   }
   getKTopStrings(vec,aux,s);
   sort(vec.begin(),vec.end(),sortInt);
   vector<string> v;
   for (int i = 0; i < n; ++i) if(i < vec.size()) v.push_back(vec[i].first);
   return v;
}
```

2. TrieMap

   Los algoritmos de TrieMap siguen la misma lógica que los de TrieArray, con la diferencia que se utilizan métodos ya implementados en la estructura map de la STL, como son map.count(), para saber si existe una key en el map y map.size() para conocer el tamaño.

   Para la función recursiva getKTopStrings(), que recorre recursivamente el árbol, se utiliza un iterador en vez de recorrer posiciones.

4. Análisis Teórico

1. Trie Array

   a) TrieArray::insert

   Dado que las operaciones en el algoritmo de insert son $\boldsymbol{O(1)}$ la complejidad esta determinada por el largo de la cadena a insertar (m). Así su complejidad algorítmica es $\boldsymbol{O(m)}$.

   b) TrieArray::search

   La complejidad algorítmica de search esta dado por el largo de la cadena a buscar, por lo que su complejidad es $\boldsymbol{O(m)}$.

   c) TrieArray::remove

   Al igual que search la complejidad algorítmica esta dada por el largo de la cadena a remover, por lo que su complejidad algorítmica es $\boldsymbol{O(m)}$

   d) TrieArray::getAll

   Para el algoritmo getAll debemos considerar la cantidad de cadenas en el Trie (n) y el largo de cada cadena (m) por lo que su complejidad es $\boldsymbol{O(n\,m)}$

   e) TrieArray::getKTopMatches

   Además de considerar los mismos parámetros que el algoritmo de getAll, debemos sumar la complejidad del algoritmo sort de la STL, que es $\boldsymbol{O(n\,log\,(n))}$, por lo tanto la complejidad del algoritmo es $\boldsymbol{O(n\,log\,(n)\,m)}$

2. Trie Map

   a) TrieMap::insert

   Debido a que buscar en el map requiere tiempo logarítmico $\boldsymbol{O(log\,\rho)}$, la complejidad de esta operación esta dada por el largo de cada string (m) y el tamaño del alfabeto (ρ). Asi su complejidad es $\boldsymbol{O(m\,log\,\rho)}$.

   b) TrieMap::search

   Al igual que insert la complejidad es $\boldsymbol{O(m\,log\,\rho)}$.

c) TrieMap::remove

Al igual que insert la complejidad es $\boldsymbol{O(m\,log\,\rho)}$.

d) TrieMap::getAll

Debemos iterar a lo largo del map, por lo que la complejidad esta dada por la cantidad de cadenas en el trie (n) y el largo de cada cadena (m), por lo que su complejidad es $\boldsymbol{O(n\,m)}$. A diferencia del Trie Array, no debemos iterar por las 26 posiciones, si no por la que están presentes en el map.

e) TrieMap::getKTopMatches

Al igual que el Trie Array, la complejidad es $\boldsymbol{O(n\,log\,(n)\,m)}$.

5. Análisis Experimental

Para el calculo experimental se considero el tiempo total que toman las distintas operaciones, para tener una visión general al ingresar una gran cantidad de cadenas de texto.

Como cadenas de texto se obtuvo el archivo words_alpha.txt que contiene aproximadamente 400000 palabras del diccionario ingles, para evitar el uso de caracteres no ASCII.

https://raw.githubusercontent.com/dwyl/english-words/master/words_alpha.txt

El entorno de sofware corresponde al S.O. Ubuntu 22.04 y las pruebas se realizaron bajo un procesador Intel i5-7400 con 8 GB de memoria RAM.

El archivo de texto se lee mediante la función freopen desde words_alpha.txt y se escriben los resultados en el archivo output.txt.

Para el algoritmo insert medimos el tiempo total que toma la inserción de n palabras al trie, para ambas implementaciones.

Para el algoritmo getAll medimos el tiempo total que toma el recorrido por el árbol, para ambas implementaciones y guardamos el vector v con todas las strings obtenidas.

Para el algoritmo getAllKTopMatches medimos el tiempo total de realizar 26 búsquedas en el trie, una para cada carácter, obteniendo vectores con las 10 cadenas mas comunes para cada carácter.

Para el algoritmo search medimos el tiempo total de buscar todas las cadenas en el trie. Para esto utilizamos el vector v obtenido en la función getAll.

Finalmente para el algoritmo remove medimos el tiempo total de remover todas las cadenas del trie utilizando el vector v, obtenido de la funcion getAll.

Los datos obtenidos se resumen en las siguientes tablas:

n	insert (s)	getAll (s)	search (s)	getKTopMatches (s)	remove (s)
50000	0.0287880000	0.0189870000	0.0072660000	1.1190370000	0.0288310000
100000	0.0563520000	0.0373540000	0.0142230000	2.3180530000	0.0579980000
150000	0.0847360000	0.0577750000	0.0215030000	3.7160750000	0.0885370000
200000	0.1107320000	0.0733910000	0.0289460000	4.6959190000	0.1193620000
250000	0.1409190000	0.0929430000	0.0368110000	5.9207950000	0.1527690000
300000	0.1773310000	0.1232110000	0.0441570000	7.4434190000	0.1819250000

Cuadro 1: Análisis experimental Trie Array

n	insert (s)	getAll (s)	search (s)	getKTopMatches (s)	remove (s)
50000	0.1438430000	0.0160680000	0.0987600000	1.0561960000	0.1113000000
100000	0.2964890000	0.0318760000	0.2019940000	2.2022550000	0.2304150000
150000	0.4430390000	0.0507470000	0.3046890000	3.5027610000	0.3456860000
200000	0.5947160000	0.0643560000	0.4107610000	4.4157740000	0.4612180000
250000	0.7665520000	0.0794090000	0.5274210000	5.5945200000	0.5888110000
300000	0.9181350000	0.1023020000	0.6331650000	7.0496180000	0.7057880000

Cuadro 2: Análisis experimental Trie Map

En base a las tablas de análisis experimental se realizan los siguientes gráficos comparando ambas implementaciones:

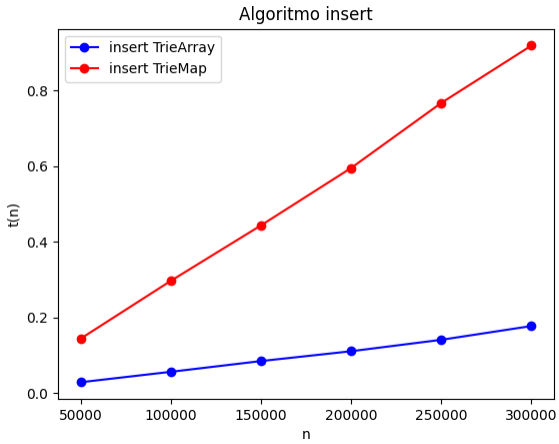

Figura 6: Algoritmo insert

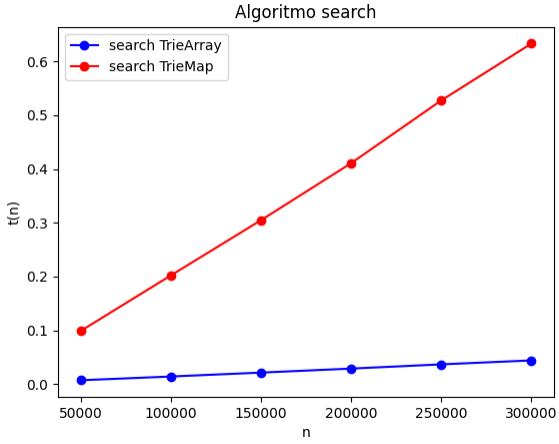

Figura 7: Algoritmo search

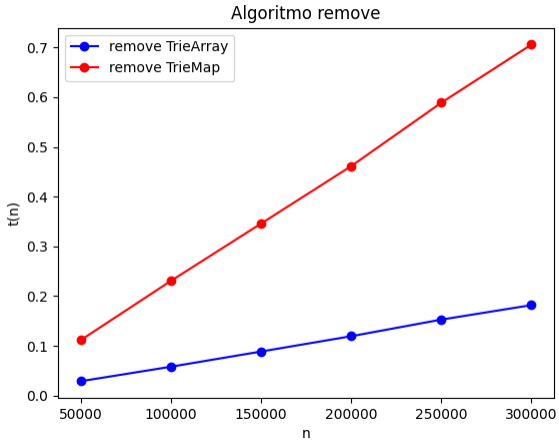

Figura 8: Algoritmo remove

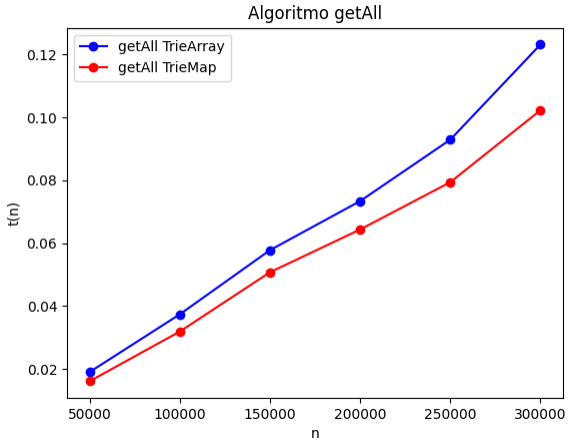

Figura 9: Algoritmo getAll

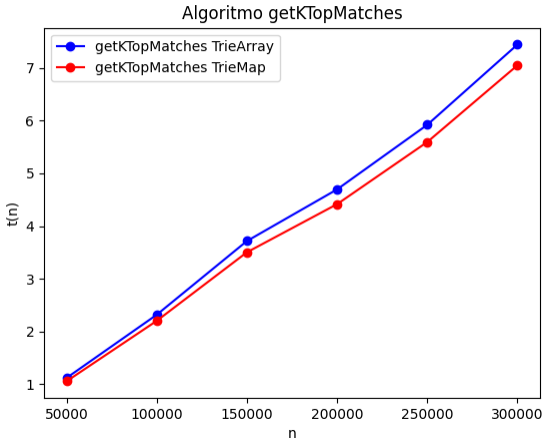

Figura 10: Algoritmo getKTopMatches

Como se observa en los gráficos, insert, remove y search el Trie Array demora $\boldsymbol{n \times O(m)}$ en realizar las operaciones, a diferencia de Trie Map que demora $\boldsymbol{n \times O(m \log \rho)}$, esto debido a que las funciones de count e insert del map toman tiempo logarítmico.

Los algoritmos de getAll y getKTopMatches obtienen curvas similares, diferenciándose en que Trie Array debe recorrer todo el array de punteros en cada nodo, mientras que Trie Map solo itera por los caracteres que están contenidos en el map.

Por esta misma razón en términos de complejidad espacial, el Trie Map es mas eficiente, ya que no almacena punteros a NULL, por lo que utiliza menos memoria que el Trie Array.

6. Referencias

1. "Data Structures and Algorithms in Java", Michael Goodrich and Roberto Tamassia, John Wiley & Sons, 2006, 4th Edition, ISBN: 0-471-73884-0.

2. Visualización de Tries https://www.cs.usfca.edu/~galles/visualization/Trie.html

3. Ejemplos Tries https://www.geeksforgeeks.org/trie-insert-and-search/

7. Anexo

De manera adicional se realizaron análisis para los otros parámetros, m (largo de la cadena) y ρ (tamaño del alfabeto), considerando un numero fijo de palabras (10000). Para esto se implemento la función printRandomString que recibe como parámetros el largo de la cadena y el tamaño del alfabeto a implementar.

```
string printRandomString(int n, int size){
  char alphabet[26] = {'a','b','c','d','e','f','g','h', 'i', 'j', 'k', 'l', 'm', 'n',
                       'o', 'p', 'q', 'r', 's', 't', 'u','v', 'w', 'x', 'y', 'z' };
  string res = "";
  for (int i = 0; i < n; i++) res = res + alphabet[rand() % size];
  return res;
}
```

Se generaron diferentes archivos de texto (incluidos en la carpeta /AnalisisExperimental/input) y se obtuvieron los siguientes resultados:

m	insert (s)	getAll (s)	search (s)	getKTopMatches (s)	remove (s)
5	0.0030470000	0.0023840000	0.0004010000	0.1166210000	0.0023970000
10	0.0060390000	0.0055500000	0.0016150000	0.2010610000	0.0054380000
15	0.0085500000	0.0082540000	0.0027150000	0.2679660000	0.0081890000
20	0.0112370000	0.0121340000	0.0039140000	0.3744650000	0.0106960000

Cuadro 3: Largo cadenas Trie Array

m	insert (s)	getAll (s)	search (s)	getKTopMatches (s)	remove (s)
5	0.0123860000	0.0017890000	0.0057360000	0.0972860000	0.0080830000
10	0.0235130000	0.0036980000	0.0100770000	0.1489970000	0.0171210000
15	0.0344770000	0.0060890000	0.0157820000	0.2028410000	0.0239910000
20	0.0460570000	0.0092090000	0.0175230000	0.2830250000	0.0307260000

Cuadro 4: Largo cadenas Trie Map

ρ	insert (s)	getAll (s)	search (s)	getKTopMatches (s)	remove (s)
5	0.0043820000	0.0037280000	0.0007800000	0.0269560000	0.0035050000
10	0.0047420000	0.0045460000	0.0012650000	0.0638910000	0.0045250000
15	0.0053370000	0.0053190000	0.0015850000	0.1036230000	0.0049710000
20	0.0053410000	0.0055650000	0.0017600000	0.1401020000	0.0051770000

Cuadro 5: Tamaño alfabeto Trie Array

ρ	insert (s)	getAll (s)	search (s)	getKTopMatches (s)	remove (s)
5	0.0193810000	0.0025760000	0.0093480000	0.0227070000	0.0133470000
10	0.0216950000	0.0032700000	0.0096880000	0.0517740000	0.0153050000
15	0.0229480000	0.0035530000	0.0097750000	0.0803610000	0.0155890000
20	0.0228880000	0.0035660000	0.0097450000	0.1085700000	0.0154680000

Cuadro 6: Tamaño alfabeto Trie Map

Los resultados obtenidos concuerdan con el análisis teórico de los distintos algoritmos.