Sofía Albert Tassier
181767

Estructuras de Datos Avanzadas

Árboles de búsqueda: Trie

Instrucciones

1. Implemente las operaciones de buscar, borrar e insertar en un Trie.
2. Implemente la operación de ordenamiento lexicográfico en un Trie. Es decir que inserte n llaves y luego recorra el Trie de forma que se obtengan las llaves ordenadas (alfabéticamente).
3. Compare el desemepeño de este algoritmo de ordenamiento contra el merge sort. Utilice conjuntos desde 1000 hasta mínimo 50,000 palabras. Hay muchas fuentes de palabras en internet. Por ejemplo https://gist.github.com/h3xx/1976236
4. Elabore un documento en donde detalle el experimento y los resultados. Comente si puede usarse este algoritmo en todos los casos en los que se pueden emplear el resto de los algoritmos de ordenamiento vistos en clase. Entregue vía Github.

Introducción

Merge Sort es un algoritmo de ordenamiento recursivo, cuyo mejor, peor y tiempo promedio es n log n. Por lo tanto, es un algoritmo eficaz y veloz. Resulta, entonces, relevante compararlo con un algoritmo de ordenamiento que utilice Tries para poder analizar qué tan eficaz es esta estructura de datos.
Los Tries son árboles de búsqueda que almacenan solo un símbolo de conexión de la llave. Esta indica cómo moverte en el árbol para llegar al dato deseado.

Para poder comparar el desempeño del ordenamiento con Trie contra el Merge Sort, fue necesario implementar las siguientes operaciones en trie: buscar,borrar e insertar, así como una operación de ordenamiento lexicográfico. El código de esta operación es:

```
/*operación de ordenamiento lexicográfico: inserta n llaves y luego recorre el trie
para obtener las llaves de ordenadas alfabéticamente
    * @return arreglo de llaves ordenadas     */
  public ArrayList<String> orden() {
      ArrayList<String> ordenadas=new ArrayList();
      ordenadas.clear();
      orden(raiz, "", ordenadas);
      return ordenadas;
  }

   private void orden(NodoTrie actual,String palabra, ArrayList<String> ordenadas) {
      if(actual.isFinPalabra())
         ordenadas.add(palabra);

      for(int i=0;i<26;i++) {
         if(actual.hijos[i]!=null) {
            orden(actual.hijos[i], palabra+simbolos[i], ordenadas);
         }
      }
  }
```

Experimento

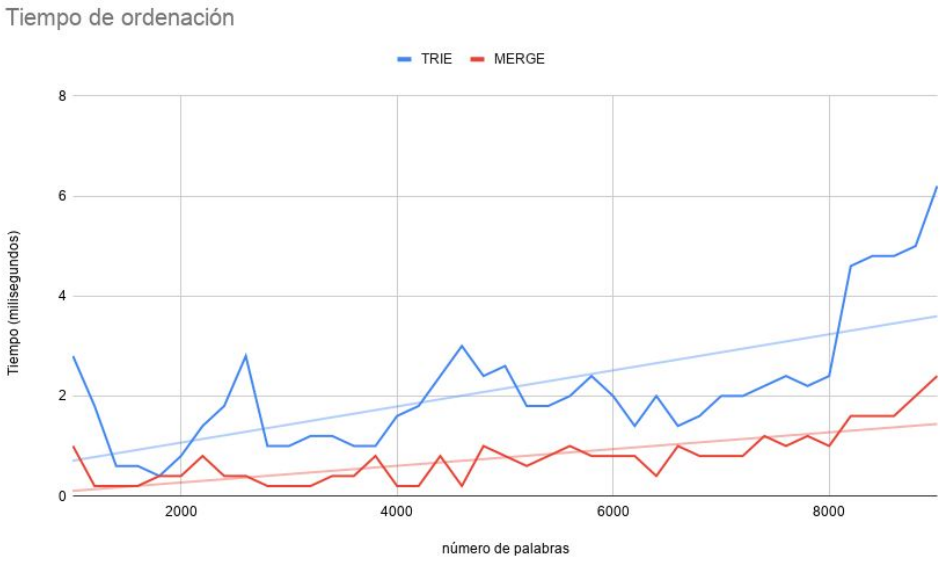
Se utilizó una lista de 9,000 palabras para llevar a cabo el experimento. Se guardaron estas palabras en un archivo de texto, que se leía y se guardaba a un arreglo tipo String. Después, se insertaron las palabras en un Trie, y se ejecutaba el método de ordenamiento de este, así como el merge sort. Para poder medir los tiempos, se usó la función System.currentTimeMillis(), midiendo el cambio de milisegundos al ordenar con cada estructura.
Para llevar a cabo este experimento con diferentes números de palabras, se usó una estructura cíclica, que repetía lo anterior pero con 200 palabras de

diferencia. De esta manera, se midieron los tiempos para ordenar desde 1,000 hasta 9,000 palabras. Para el Trie, no se midió el tiempo que toma insertar, solo ordenar.
A pesar de que fuera el mismo documento de texto, había variaciones entre diferentes ejecuciones del programa, por lo que el mismo experimento se llevó a cabo varias veces y se calculó un promedio del tiempo.

Resultados

A continuación, se presentan tanto la tabla como las gráficas de los resultados

num palabras	TRIE	MERGE
1000	2,8	1
1200	1,8	0,2
1400	0,6	0,2
1600	0,6	0,2
1800	0,4	0,4
2000	0,8	0,4
2200	1,4	0,8
2400	1,8	0,4
2600	2,8	0,4
2800	1	0,2
3000	1	0,2
3200	1,2	0,2
3400	1,2	0,4
3600	1	0,4
3800	1	0,8
4000	1,6	0,2
4200	1,8	0,2
4400	2,4	0,8
4600	3	0,2
4800	2,4	1
5000	2,6	0,8
5200	1,8	0,6
5400	1,8	0,8
5600	2	1
5800	2,4	0,8
6000	2	0,8
6200	1,4	0,8
6400	2	0,4
6600	1,4	1
6800	1,6	0,8
7000	2	0,8
7200	2	0,8
7400	2,2	1,2
7600	2,4	1
7800	2,2	1,2
8000	2,4	1
8200	4,6	1,6
8400	4,8	1,6
8600	4,8	1,6
8800	5	2
9000	6,2	2,4



Conclusiones

ESte algoritmo se puede usar en todos los casos en los que se pueden emplear el resto de los algoritmos de ordenamiento vistos en clase. No obstante, es necesario manejar las excepciones. Por ejemplo, en el caso de las palabras, es necesario definir qué pasa cuando se desea agregar una palabra con un símbolo que no existe. En este programa, simplemente no agregaba la palabra, pero también es posible lanzar una excepción o agregar al conjunto de símbolos el caracter que no está (como letras con acentos o caracteres especiales). El ordenamiento es lexicográfico, por lo que está basado en el orden alfabético; lo que hace es recorrer el árbol en el orden del conjunto de símbolos. Por lo tanto, mientras el arreglo esté ordenado (cosa que siempre ocurre, pues los constructores del Trie y del NodoTrie utilizan *Array.sort* para los símbolos), será posible utilizar este método.

Podemos ver, de manera visual en la gráfica, que el ordenamiento de Trie es menos veloz (y por lo tanto menos eficiente) que merge sort. Además, el crecimiento de merge sort se parece mucho más a la aproximación lineal que el de Trie, por lo que podemos concluir que el mejor tiempo, el peor y el promedio fluctúan mucho en ordenamiento con Trie, a diferencia de con Merge Sort.