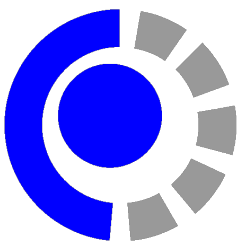
Diseño de algoritmos

Trabajo Práctico 1 - Estructuras de Datos Avanzadas y Análisis Amortizado

Manuel Latorre FAI-1931
manuel.latorre@est.fi.uncoma.edu.ar

Segundo cuatrimestre 2022



Facultad de Informática
UNIVERSIDAD NACIONAL DEL COMAHUE

Índice

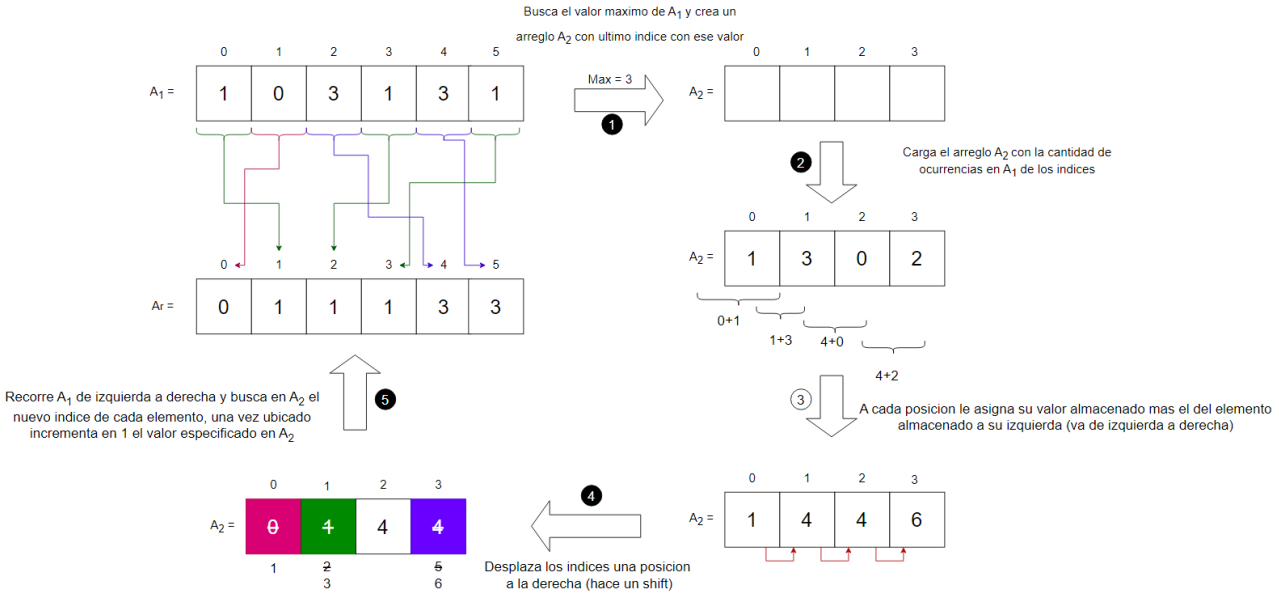


Figura 1: Traza de algoritmo de ordenamiento por conteo **(imagen en alta resolución en [1])**

1. Punto 1

a) *Cómo se comportaría el algoritmo de Ordenamiento por Conteo si en el arreglo original se permiten elementos repetidos?*

En la figura 1 se plantea una traza de ejemplo de como funciona el algoritmo de ordenamiento por conteo con elementos repetidos (en su descripción ademas se encuentra una referencia al repositorio donde se puede ver la imagen mas grande en caso de ser necesario)

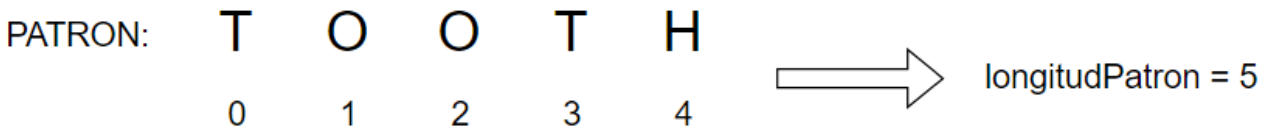
b) *Modificar el algoritmo de Ordenamiento por Conteo para poder ordenar un arreglo de caracteres, de tal manera, que sean indistintas las letras mayúsculas y minúsculas, es decir que siempre se cumpla que:*
$a < b$,
$a < B$,
$A < b$,
$A < B$,
En el arreglo resultante debe figurar cada letra en el mismo modo que en el arreglo original.

Para la implementación los únicos cambios respecto a un algoritmo planteado para números enteros serán la utilización el método `Character.compare()` para comparar caracteres y el método `Character.toLowerCase()` para convertir caracteres a minúsculas y asi realizar todas las comparaciones logrando la in-distinción entre letras mayúsculas y minúsculas pedida

Código 1 Metodo ordenarPorConteoChar

```
1  public static char [] ordenarPorConteoChar(char [] C){
2    int n = C.length;
3    int [] count = new int [n];
4    char [] sorted = new char[n];
5    for (int i = 0; i < n; i++) {
6      count[i] = 0;
7    }
```

```
    for (int i = 0; i < n-1; i++) {
      for (int j = i+1; j < n; j++) {
        if(Character.compare(Character.toLowerCase(C[i]),
           Character.toLowerCase(C[j]))<0){
          count[j]++;
        }else{
          count[i]++;
        }
      }
    }
    for (int i = 0; i < n; i++) {
      sorted[count[i]]=C[i];
    }
    return sorted;
  }
```



Valor = longitudPatron - indexLetra -1

T = 5-0-1 = 4
O = 5-1-1 = 3
O = 5-2-1 = 2
T = 5-3-1 = 1
H = 5 (Ultima letra es la longitud del patron)
* = 5 (Cualquier otra letra es la longitud del patron)

Letra	T	O	H	*
Valor	~~4~~ 1	~~3~~ 2	5	5

Figura 2: Ejemplo de creación de tabla de errores

2. Punto 2

Para buscar, mediante el algoritmo de Horspool, un patrón de longitud m en un texto de longitud n, con $n \leq m$, dar un ejemplo para:

- *Mejor caso.*
- *Peor caso.*

El algoritmo de Horspool es una version simplificada del algoritmo Boyer-Moore utilizando una única tabla. Este pre-procesa el patron que se desea encontrar generando una tabla de desplazamiento que determina cuanto desplazar el patron cuando ocurre un fallo 2.

Siempre realizaran los desplazamientos basados en el carácter del texto c alineado con el ultimo carácter comparado (fallo) en el patron de acuerdo a la entrada a la tabla de desplazamiento para c 3.

Análisis de eficiencia:

- El peor caso sera un patron en el cual coinciden todos los caracteres excepto el primero por ejemplo:
  - 1^n texto de entrada (longitud n)
  - $0111\ldots1$ patron (longitud m)

  En este caso se chequearan constantemente los m caracteres del patron por lo que se tendrá en el peor caso, es decir cuando se hagan desplazamientos de a una posición al encontrar el fallo, un tiempo de ejecución $O(nm)$.
- El mejor caso sera cuando el texto de entrada no contenga caracteres del patron por ejemplo:
  - 1^n texto de entrada (longitud n)

Letra	T	O	H	*
Valor	1	2	5	5

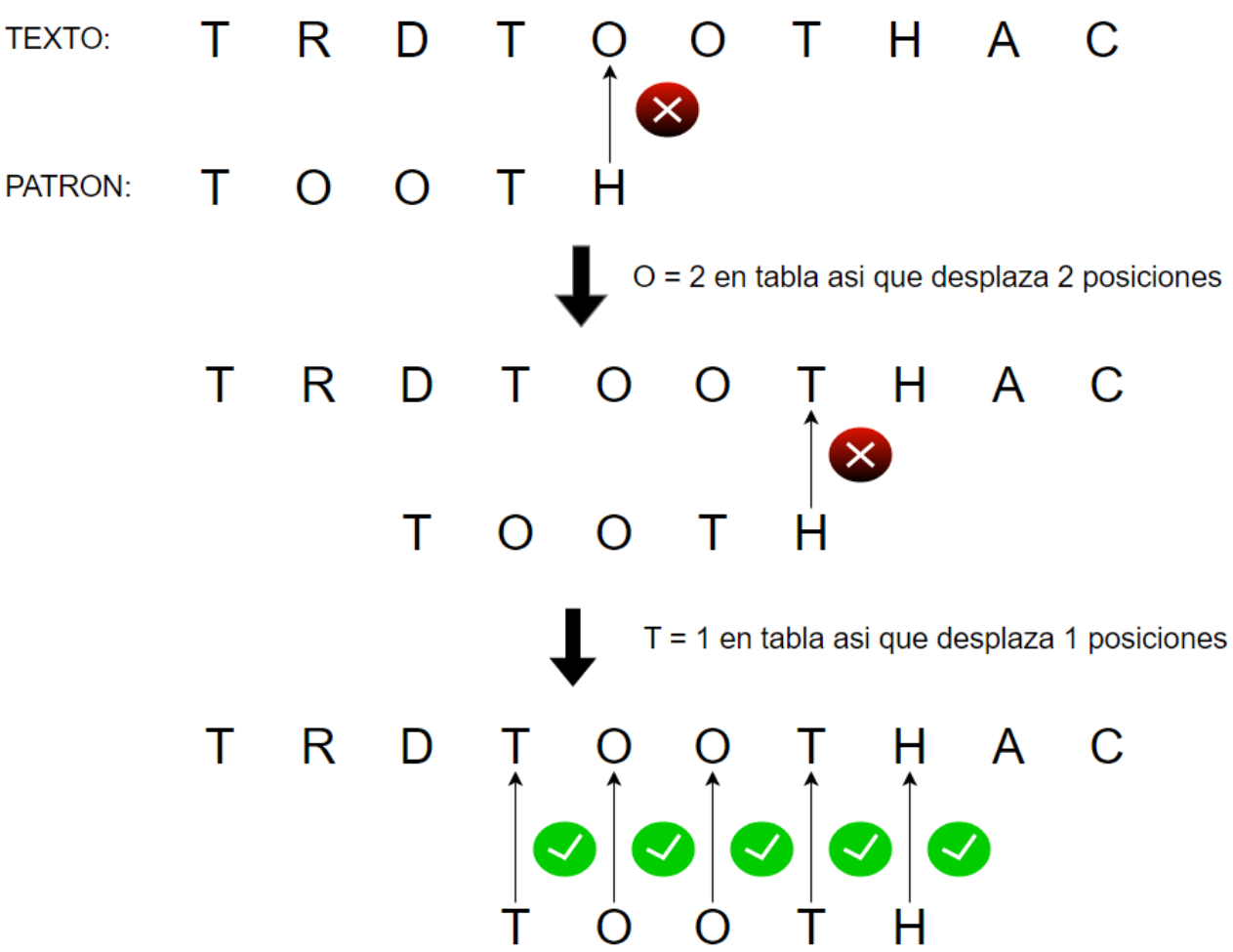


Figura 3: Traza de búsqueda de patron con algoritmo de Horspool

- 0^m patron (longitud m)

esto hará que se produzcan saltos del tamaño del patron, es decir de tamaño m, produciendo un tiempo de ejecución $O(m/n)$.

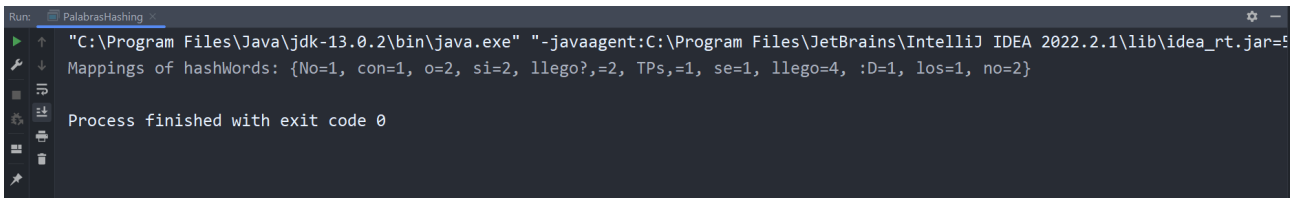


Figura 4: Salida por consola para el texto 2

3. Punto 1

Construir un algoritmo que permita, utilizando Hashing, resolver el siguiente problema: Dado el texto de un archivo, escrito en lenguaje coloquial, generar un listado de palabras diferentes y la cantidad de veces que aparece cada una en el texto.

El método implementado lee el texto que se encuentra en un archivo `.txt`, convierte los caracteres iniciales de cada palabra a minúsculas (asumiendo que no hay mayúsculas en la mitad de las palabras), toma palabra a palabra del texto y lo guarda en un arreglo para luego utilizar las palabras como key de la tabla hash, almacenando la cantidad de veces que dicha palabra aparece en el texto, cada vez que se intenta agregar una palabra, el valor que la tabla hash tiene almacenado para la key "palabra" incrementara en uno, en la figura 4 se observa la salida resultante

Código 2 Texto de entrada

```
No se si llego con los TPs
llego o no llego?
llego o no llego?
si llego :D
```

Código 3 Metodo readAndHash()[2]

```
public static void readAndHash(Hashtable<String,Integer> hashWords)
    throws IOException {
  String filePath = "src\\T1P2\\texto.txt";
  File file = new File(filePath);
  BufferedReader br = new BufferedReader(new FileReader(file));
  String st;
  String [] words =null;
  int lenghtArrayOfWords;

  while((st = br.readLine())!=null){//Continua mientras tenga lineas
      por leer, va linea por linea
    words = st.split(" "); //Genera array de palabras separadas por
       espacios
    Arrays.stream(words).map(word -> word.toLowerCase()); //Convierte a
       minusculas las mayusculas
    lenghtArrayOfWords = words.length;
    for (int i = 0; i < lenghtArrayOfWords; i++) {
      if(hashWords.containsKey(words[i])){
        hashWords.put(words[i], hashWords.get(words[i])+1);
      }else{
        hashWords.put(words[i],1);
      }
```

```
        }
      }
    }
```

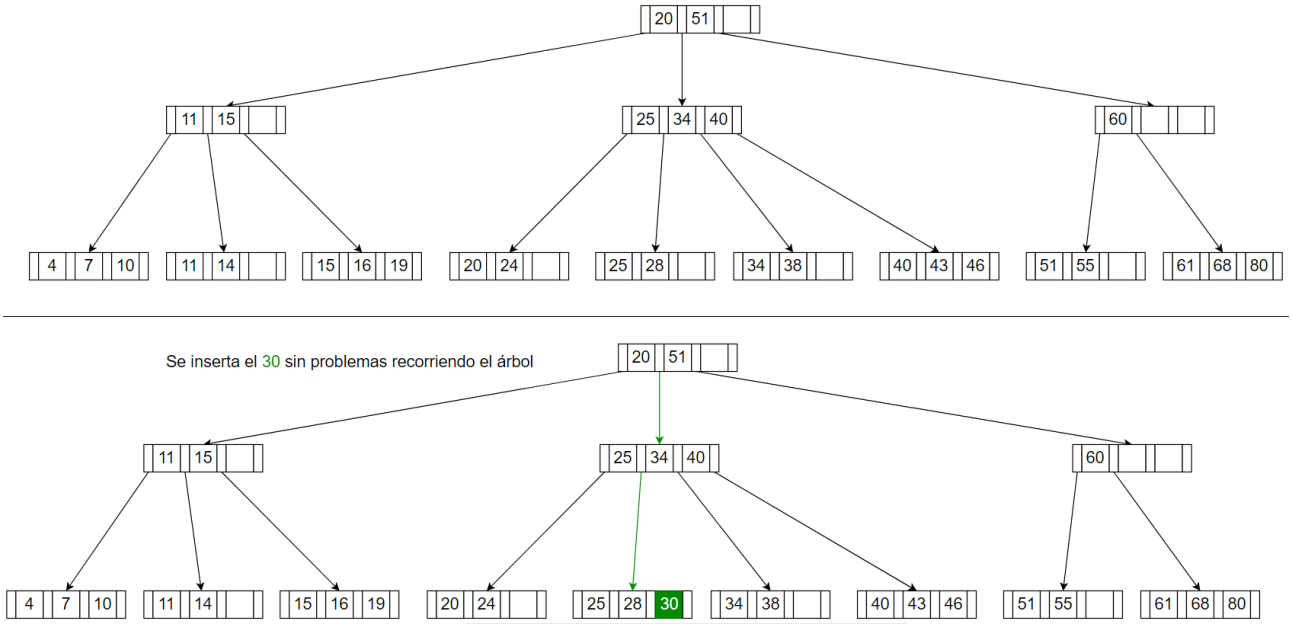


Figura 5: Inserción de los elementos 30 y 32 de manera consecutiva en árbol B **(imagen en alta resolución en [3])**

4. Punto 4

A partir del primer árbol B de la figura 5, dibujar el árbol resultante, luego de insertar consecutivamente los elementos 30 y 32.

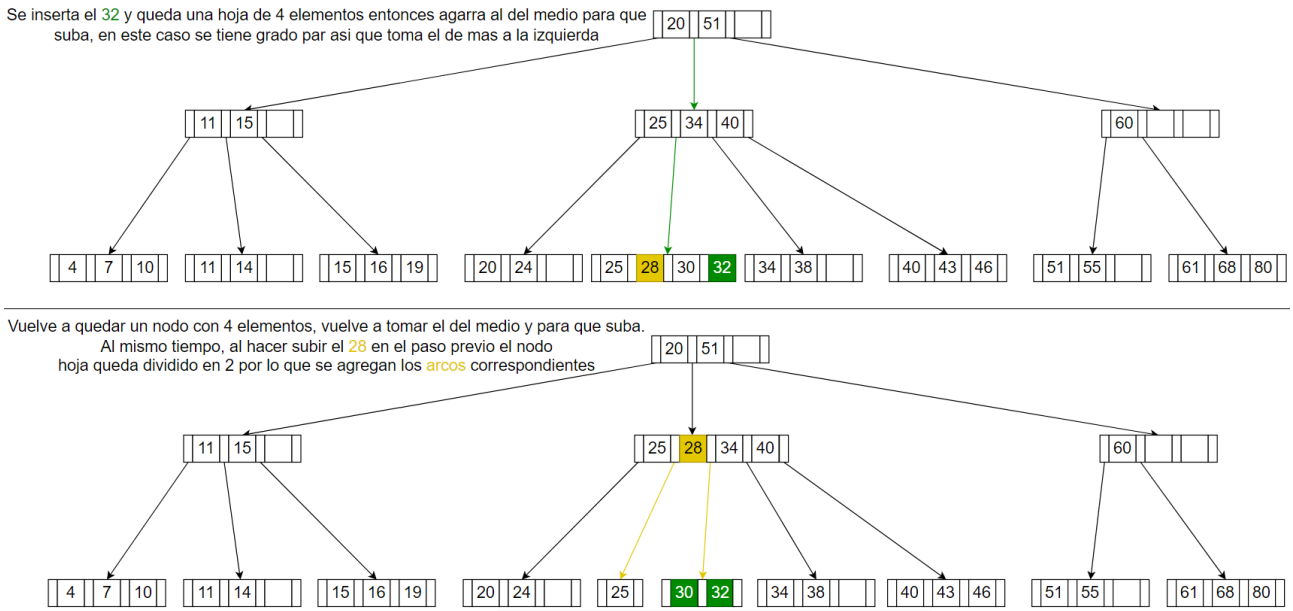


Figura 6: Inserción de los elementos 30 y 32 de manera consecutiva en árbol B **(imagen en alta resolución en [3])**

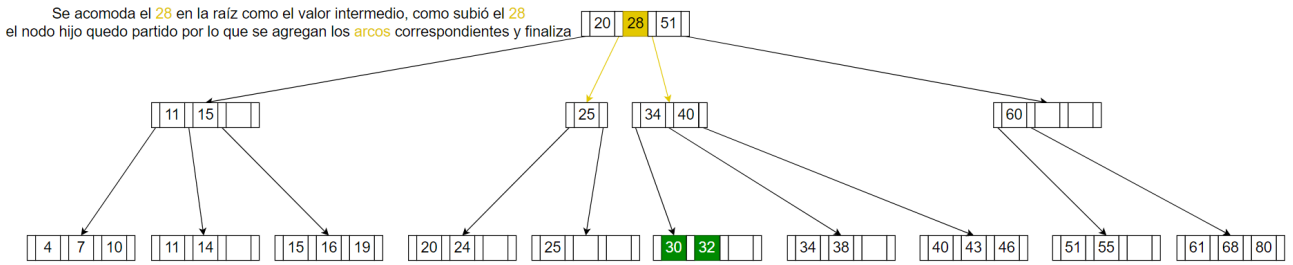


Figura 7: Inserción de los elementos 30 y 32 de manera consecutiva en árbol B **(imagen en alta resolución en [3])**

Referencias

[1] *Traza de ejecucion de ordenamiento por conteo*. https://github.com/ManuelLatorre98/Disenio-de-algoritmos/tree/main/Informes/TPO1P2/Images/Punto1. Accessed: 2022-10-16.

[2] *Codigo readAndHash*. https://github.com/ManuelLatorre98/Disenio-de-algoritmos/blob/main/src/T1P2/PalabrasHashing.java. Accessed: 2022-10-16.

[3] *Imagenes arbol B*. https://github.com/ManuelLatorre98/Disenio-de-algoritmos/tree/main/Informes/TPO1P2/Images/Punto4. Accessed: 2022-10-16.