Estructura de datos y Algoritmos
Práctica Diccionario

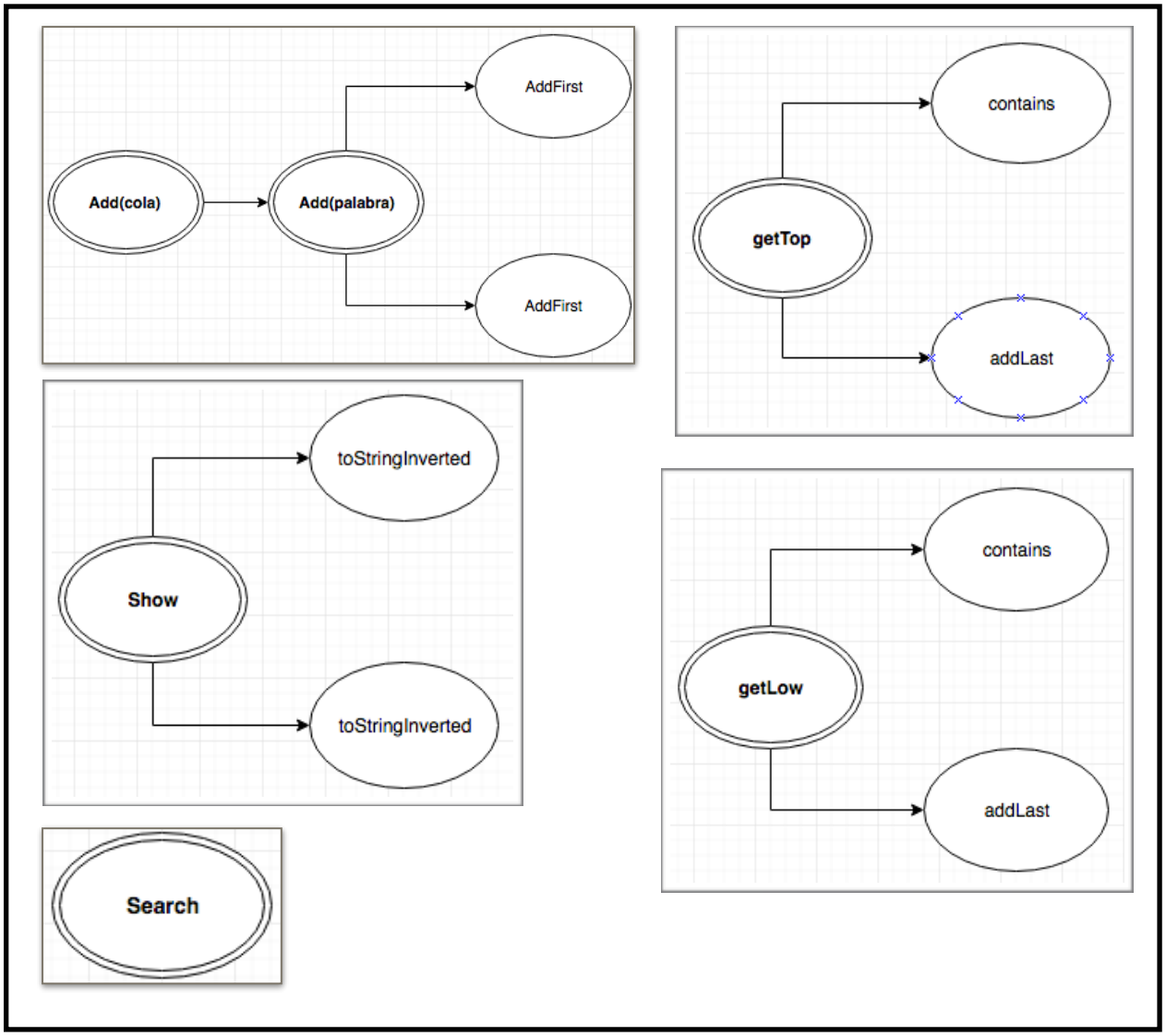
Pablo Escrivá Gallardo 100348802
Cristian Gómez López 100349207

Estructura de datos y Algoritmos

Práctica Diccionario

Fase 1

A)Atributos necesarios para implementar el diccionario



C)Complejidad de cada uno de los métodos

DList	
Método	Complejidad
Add(cola)	//O(n)
Add(palabra)	//O(n)
show	//O(n)
search	//O(n)
getTop	//O(n^3)
getLow	//O(n^3)

Decisiones de diseño

La implementación de una lista doblemente enlazada se ha basado en las siguientes ventajas frente a una simplemente enlazada:

I. El método **addLast** se mantiene constante ya que no es necesario hacer un recorrido de toda lista para insertar al final, basta con consultar el *prev* del *tailer.*
II. El método **removeLast** se mantiene constante ya que no es necesario hacer un recorrido de toda lista para borrar el nodo final.
III. El método **getTop** aprovechará el acceso al último nodo a la hora de hacer la inserción en la lista auxiliar que se crea como objeto de devolución, ya que lo hace al final.
IV. El método **getLow** aprovechará el acceso al último nodo a la hora de hacer la inserción en la lista auxiliar que se crea como objeto de devolución, ya que lo hace al final.
V. El método **show** en nuestra lista doblemente enlazada, para la opción de impresión en orden inverso, usará directamente como nodo de inicio el *tailer.prev* .De esta forma se evitará hacer un doble recorrido de la lista.
VI. El método **getLast** se evitará recorrer la lista.

Fase 2

A)Pregunta corta: clave y valor asociado

Para almacenar palabras en orden alfabético (junto con sus frecuencias)en un BST usaríamos la palabra como clave y la frecuencia como valor asociado.

C)Complejidad de métodos + comparación con métodos Fase 1

DList		DictionaryTree	
Método	Complejidad	Método	Complejidad
Add(cola)	//O(n)	Add(cola)	//O(n)
Add(palabra)	//O(n)	Add(palabra)	//O(n^2)
show	//O(n)	show	-Recursivo-
search	//O(n)	search	-Recursivo-
getTop	//O(n^3)	getTop	-Recursivo-
getLow	//O(n^3)	getLow	-Recursivo-

A primera vista podemos asegurar que la complejidad de DList y DictionaryTree ha empeorado claramente. No obstante es probable que los métodos getTop y getLow del DictionaryTree hayan mejorado una complejidad de [n^3] de la que partíamos en DList.

D)Pregunta corta: Diccionario que recibe lista de palabras ya ordenadas alfabéticamente

Al hacerse la inserción a partir de una lista de palabras previamente ordenadas, el árbol resultante será una “copia” de la lista inicial ya que todas las palabras estarían dentro de una misma rama debido a que siempre que se inserta una palabra se hace en el la misma rama del último nodo.

Suponiendo que el método de inserción en este nuevo árbol no fuese recursivo, la complejidad de inserción, sería la misma que en una lista simplemente enlazada.

Una buena forma de facilitar el trabajo de inserción al final sería equilibrar previamente el árbol. De esta manera conseguiríamos reducir la complejidad drásticamente ya que de ninguna recorreríamos de nuevo todos y cada uno de los nodos.

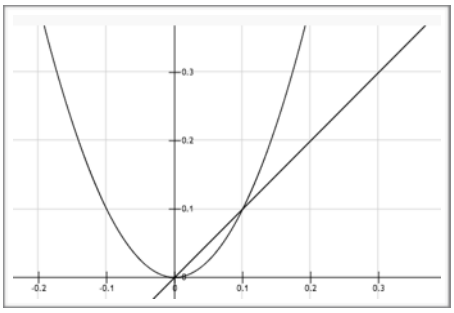
Fase 3

Decisiones de implementación

La primera decisión de diseño que tomamos fue cómo **escoger palabras aleatoriamente** a partir de una lista en la que previamente las habíamos cargado(lista sin palabras repetidas de la Fase 1).
Estuvimos valorando dos opciones:

- Recorrer la lista “n” veces a la posición indicada por un número aleatorio e ir guardando la palabra encontrada en otra lista comprobando si ésta ya había sido almacenada
- A partir de la lista inicial, ir borrando palabras hasta obtener el número deseado n.
- Combinar ambas soluciones para así obtener una posible mejora dependiendo de cual fuese el valor de n.

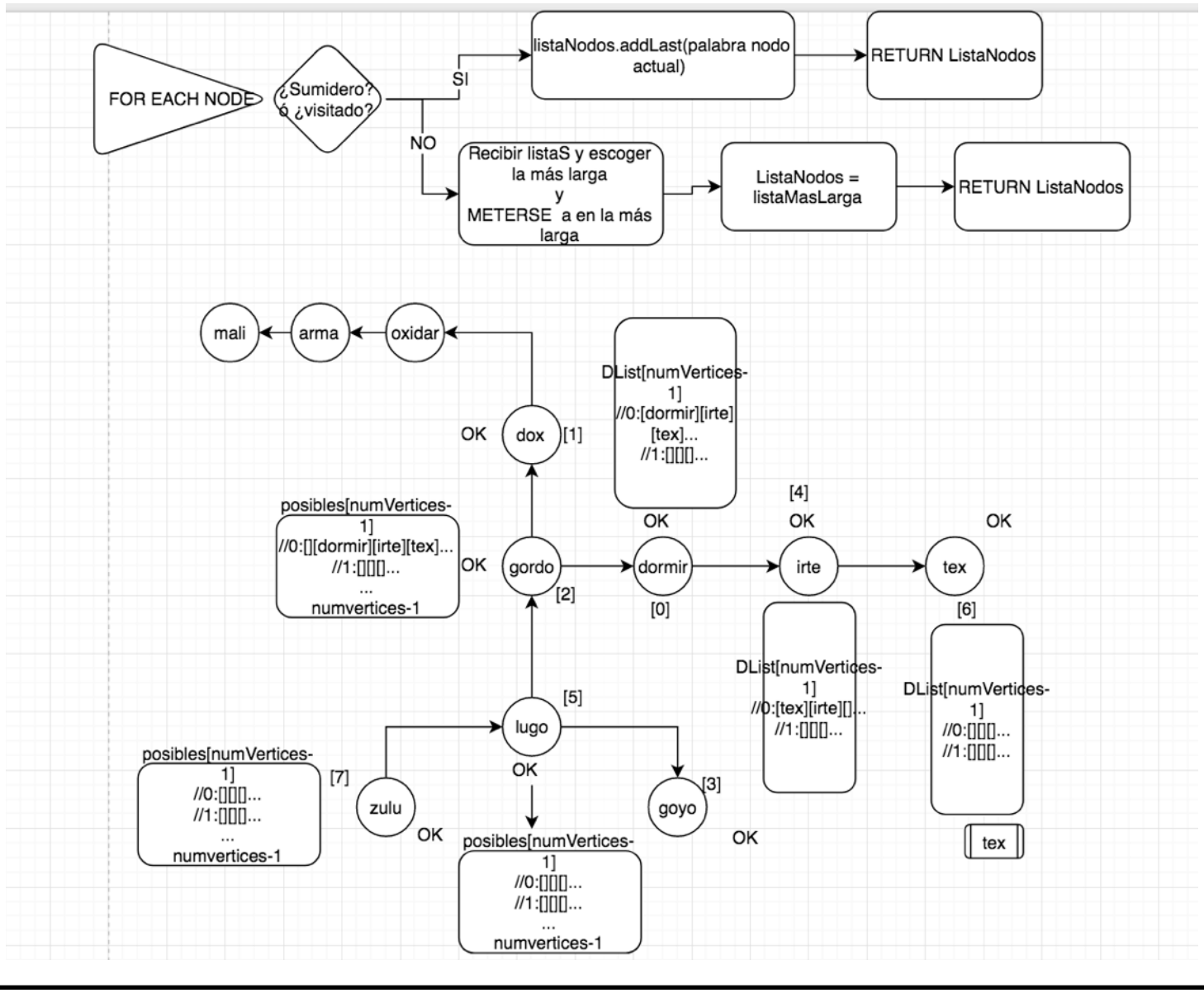
Tras calcular ambas complejidades nos encontramos frente a la siguiente situación



A partir del gráfico concluimos que es mejor borrar las palabras.(borrar: parábola lañadir: recta)

Por otro lado para resolver el problema de las **palabras sumidero** lo que hacemos es probar para cada palabra sumidero si la podemos sustituir por otra que no lo sea sin importarnos si esto altera la condición de no sumidero de las anteriores.

Después para buscar la **cadena más larga** nos basamos en un método recursivo cuyo caso base fuese encontrar una palabra sumidero, y en tal caso devolver la palabra hacia atrás en el grafo. Para los nodos que tuviesen salidas a varios nodos, compararemos el tamaño de las listas que recursivamente le hubiese llegado de cada uno de sus nodos adyacentes, eligiendo entonces la más larga. Haciendo esto llegamos a la lista más larga **a partir de una palabra**



Por último para comprobar cual es la **cadena mas larga de todo el grafo**, calculamos los caminos mas largos partiendo dese cada uno de los vértices, y entre ellos escogemos el mayor.