Práctica 1 Estructura de datos

Alumno: Federico Hernán Suárez Palavecino

Fecha: 15/04/2020

Contents

Pila estática

Como primera parte de la práctica nos encontramos con la pila estática, una estructura sencilla que consiste en una estructura con un puntero de enteros el cual nos permite tener un vector de elementos indefinido para el que podemos reservar el espacio necesario, y un puntero al final de dicho vector dinámico.

La implementación consiste en una estructura LIFO (last in first out), es decir, cuando vamos a sacar un elemento de la lista comenzamos por el final.

Esta estructura nos permite trabajar en varios campos en los cuales se trabaja con pila, como por ejemplo la famosa pila de registros de ARM. Donde los registros se guardan en una pila, o incluso al trabajar con los datos de los procesos al trabajar a nivel de threads en asignaturas como FSO.

Consideraciones respecto a la práctica:

El enunciado de la práctica dejaba bastante claro todos los aspectos de la práctica, por lo tanto, no hay a grandes rasgos nada relevante que especificar en este apartado.

Respecto a aspectos adicionales/opcionales de la práctica, se han tenido en cuenta diversas cuestiones.

Las implementaciones extras han sido:

- Poder duplicar un valor (se saca el último elemento, y se introduce dos veces)
- Poder nPlicar un valor (se saca el último elemento, y se introduce n veces)
- Permutacion de nElementos (se extraen los n últimos elementos, y se introducen en orden inverso)
- Función exponencial (se coge el último valor, se calcula factorial y se introduce el valor del factorial del valor sacado)
- Elevado a una potencia (se cogen los dos últimos elementos y se eleva el segundo al primero, si tenemos 1-2, se eleva 1 a 2 y se introduce dicho valor)
- Valor absoluto (cambia el ultimo valor por su valor absoluto, 1 = 1, -1 = 1)

Control de errores de ejecución y decisiones de implementación:

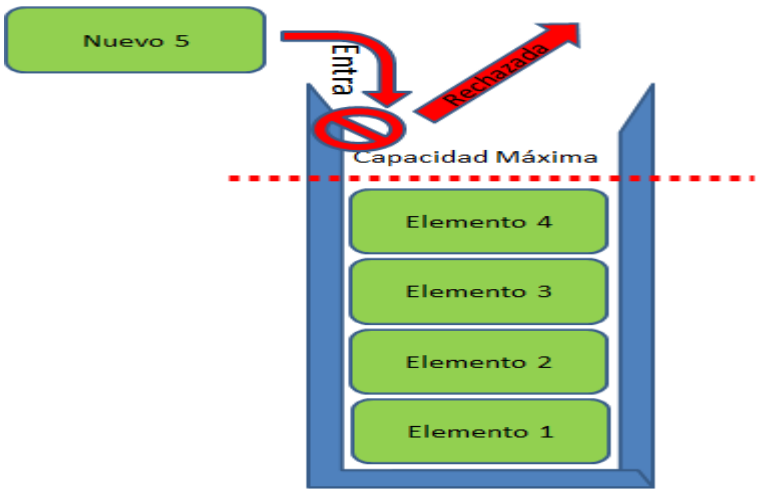
Entre algunos de los controles que he tenido que realizar a lo largo del código se encuentran:

- Control de la existencia de la pila a la hora de acceder a ella.
- Control sobre el vector dinámico de elementos, no acceder a posiciones nulas.
- Comprobar que la pila no se encuentra llena al insertar elementos.
- Comprobar que la pila no se encuentra vacía al querer sacar un elemento.
- Controlar que realmente se saca el elemento de la pila y deja de existir.

En cuanto a la implementación de la estructura de la calculadora, en un principio fue implementada para leer tan solo una línea y tratarla, pero finalmente al hablar con mi profesor de laboratorio, acabó siendo implementada para ser capaz de tratar diversas líneas y realizar todas las operaciones.

Para tratar el fichero, se abre el fichero y se guardan todos los datos del fichero en un vector de caracteres (un *string*), posteriormente dicho vector lo tratamos elemento a elemento, realizamos un *split* del contenido con la coma como separador.

Finalmente hacemos uso de un *switch* y de la función *isdigit*, en caso de que el carácter a tratar sea un digito, lo agregamos a la pila, ya que dicho carácter será realmente un número que estamos colocando en la pila, y en caso de que no fuese un digito, se trataría de una operación y aquí es donde entra en juego el *switch* y dependiendo de que operación sea, realizamos una operación u otra.



Cola estática

La cola a implementar consiste en una cola estática circular, es decir, cuando acabamos de introducir elementos hasta el MAX_ELEMS, volvemos al principio, como si se tratara de una serpiente que se come la cola.

Respecto a la cola, se trata de una estructura FIFO (First in first out), es decir, el primer elemento en entrar será el primero en salir.

Consideraciones respecto a la práctica:

La estructura al tratarse de una FIFO ha sido implementada con una estructura la cual consiste de un vector dinámico de enteros, un 'puntero' al principio de la cola y otro al final.

Cuando llegamos al final, el puntero del final se "resetea" y vuelve al principio, con lo cual vuelve a estar a 0, pero solo seguirá introduciendo valores en caso de que no esté llena, no ha sido implementado de manera que se sobrescriban valores.

Consideramos que la pila está llena en caso de que los punteros estén en la misma posición y la posición anterior contenga un valor.

Y en caso de que los punteros sean iguales pero la posición anterior sea "NULL" consideramos que la pila está vacía.

No se han agregado partes opcionales.

Control de errores de ejecución y toma de decisiones de implementación:

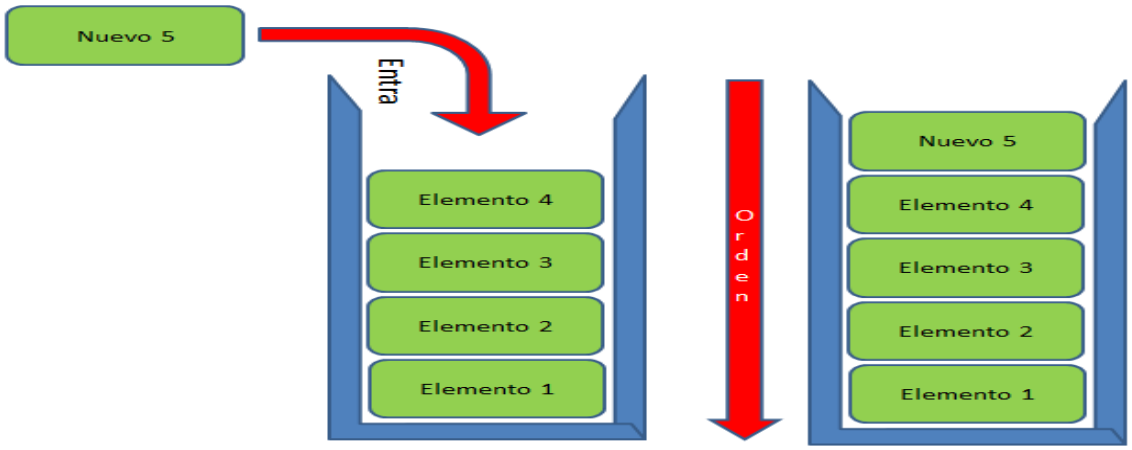
Para implementar la cola se han debido tener en cuenta varios factores entre los cuales podemos encontrar:

- Control de la existencia de dicha cola.
- Control en cuanto al acceso de las posiciones del vector.
- No superar el número máximo de elementos a insertar.
- Realizar efectivamente el funcionamiento de una cola circular
- Controlar que el elemento extraído es el primero y no el último ni alguno intermedio.

Como utilización de la cola se nos pide que realicemos el triángulo de pascal a partir del uso de la cola. Para dicha implementación en mi caso he decido crear una función, a la cual se le pasa un entero por parámetro, el cual se le pide previamente al usuario. Una vez dentro de la función declaramos un array de colas y comenzamos a crear desde el nivel 1 de la cola hasta el nivel n-1, para realizar los niveles simplemente si la posición es la 0 o la n-1 el valor corresponde a 1, en caso contrario se realiza la suma de los valores de la cola anterior de la posición donde nos encontramos mas el valor de la posición anterior.

Un ejemplo:

Si estamos en la iteración 2, el resultado que iría en esta posición sería la suma de la posición 2 de la cola anterior mas el de la posición 1.



Doubly linked list

La doubly linked list, o lista doblemente enlazada en español, consiste en la implementación de una serie de nodos enlazados entre si, los cuales contienen un puntero hacia el nodo anterior y un puntero hacia el nodo posterior.

Respecto a esta estructura haciendo recerca en internet hay cientos de maneras de implementarlas, debido a la práctica la manera de implementarla ha sido con dos nodos fantasmas, uno al principio y otro al final, los cuales simplemente nos ayudas a trabajar con el PDI (Punto de interés) el cual nos permite colocarnos en una posición a lo largo de la lista (un nodo) e insertar elementos justo delante de este (atrás).

Peculiaridades de la lista:

La lista como tal tiene una peculiaridad, y es que, en un principio, a no ser que introduzcamos un algoritmo de ordenación, se trata de una lista totalmente desordenada, con lo cual dificulta la búsqueda en esta, pero la inserción es realmente rápida. El tiempo aproximado de búsqueda es O(n) por lo contrario en el caso de inserción se trata de un O(1).

La estructura de la lista contiene un puntero al primer nodo de la lista, un puntero al último y un puntero al pdi. El pdi es el que iremos utilizando para movernos por la lista e ir insertando los elementos donde más nos convenga y los punteros primero y final nos permitirán colocarnos al principio o al final para facilitarnos el trabajo.

Finalmente, la lista en cada nodo contiene punteros al nodo siguiente y al nodo anterior, así como una variable de tipo entero donde almacenaremos nuestro dato.

Resultados de la doubly linked list:

mida	cost	desv.Est			
1000	762	329.95			
2000	1594	628.94			
3000	2367	957.6			
4000	3150	1261.89			
5000	3983	1562.95			
6000	4660	1971.23			
7000	5454	2245.36			
8000	6147	2636.18			
9000	7101	2839.73			
10000	7870	3215.14			
11000	8745	3410.58			
12000	9616	3760.69			
13000	10191	4162.55			
14000	10788	4589.82			
15000	11836	4790.96			
16000	12371	5182.7			
17000	13298	5510.58			
18000	14194	5647.51			
19000	14840	6270.57			
20000	15647	6438.48			
21000	16570	6650.77			
22000	17507	6847.9			
23000	18362	7233.01	37000	29578	11772.3
24000	18691	7821.08	38000	28663	12970.99
25000	19441	8070.99	39000	31471	11966
26000	20427	8392.92	40000	31460	13027.2
27000	20910	8872.27	41000	32276	13320.4
28000	21668	9131.37	42000	32655	13566.18
29000	22982	9155.66	43000	33769	13860.28
30000	23241	9939	44000	33809	14452.6
31000	24798	9645.03	45000	35587	14305.42
32000	24558	10603.64	46000	35361	15164.96
33000	25688	10700.23	47000	37786	14408.07
34000	26672	11043.06	48000	37520	15368.12
35000	27736	10902.6	49000	38296	15752.78
36000	28138	11657.48	50000	38983	16188.89

Imagen 1: Resultados con tamaños de lista desde 1000-50000 elementos

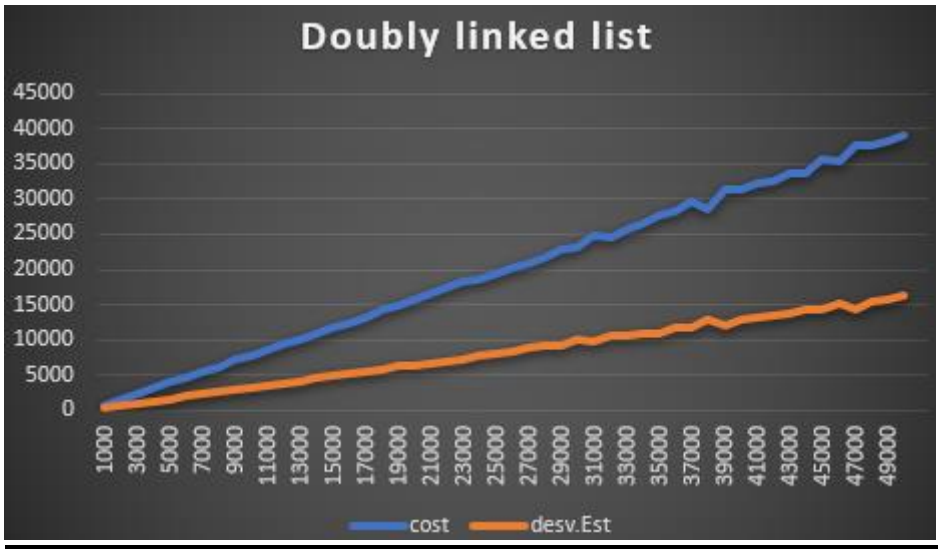


Imagen 2: resultados vistos en una gráfica

Como podemos apreciar tanto en la tabla de resultados como en la gráfica, se puede observar que a cuantos mas elementos agregamos a la lista, mas le cuesta a esta procesarlos, siendo cada vez mas y mas elevados, en un orden creciente casi que constante a la par de elementos insertados.

Por lo tanto, se trata de una lista poco eficiente para búsquedas, pero eficiente en el ámbito de inserción ya que su coste es lineal.

Por último, hay que comentar que es una lista de sencilla implementación y la cual nos puede ser útil en caso de que las búsquedas no tengan mucho uso en nuestro programa. También aclarar que búsquedas dicotómicas, no nos servirían ya que la propia estructura en si no permitiría dicho tipo de búsqueda al no saber dónde está el punto medio y en un principio tampoco la tendríamos ordenada.

Consideraciones de implementación:

Para la implementación de la DLL hace falta tener mucho control sobre los punteros y que no apunten a posiciones NULL ya que luego no podremos acceder a ellos.

Por lo tanto, para poder trabajar con esto, como para poder insertar un elemento en la última posición necesitaríamos que el pdi se encontrase en el sector NULL del final, lo que hacemos es crear un nodo fantasma, el cual realmente haría de

NULL, pero como no podemos trabajar con el valor NULL directamente ya que nos daría errores de acceso a la memoria, utilizamos este nodo.

Una vez insertado este nodo ya podemos insertar al final de la lista e ir trabajando con ella.

Las mayores complicaciones con esta práctica fue la estructura como tal (los registros) y el tema de reservar memoria, ya que buscando mucha información por internet la mayoría hacían uso de funciones declaradas que devolvían el tipo de lista que estaban tratando, o en su defecto pasaban punteros a punteros. En nuestro caso no teníamos dicha posibilidad y fue lo que más problemática causo de cara a realizar el trabajo.

La estructura consiste en una lista con un puntero al principio, otro al final y otro al di actual el cual es el que se va moviendo. También tenemos la estructura nodo que es la que vamos utilizando para crear cada 'bloque' de nuestra lista e ir entrelazándolos entre sí. Los punteros definidos previamente no son mas que punteros a dichos nodos.

Cuando creamos la lista se crea vacía, al introducir el primer elemento comprueba si existen primero y último, como no existen nos crea dos nodos fantasmas y se los asigna, y coloca el elemento en un nodo y el nodo entre medio de los elementos fantasmas. En las posteriores ejecuciones ya no entra en esta condición y por lo tanto salta directamente a insertar un nodo donde corresponda.

En caso de que se de el caso de que eliminemos todos los elementos de la lista, los dos elementos fantasmas se borran también, así nos aseguramos que en cuanto volvamos a insertar algún elemento no haya ningún problema y se ejecute como la primera vez.

Finalmente, se nos pide calcular el costo de búsqueda de la lista para listas de entre 1000 y 50000 elementos, con búsquedas de 1000 elementos aleatorios.

Para poder implementar esto simplemente hacemos uso de una función que inserta todos los elementos, y luego realiza búsquedas aleatorias, y va cogiendo los valores para calcular la media y la desviación estándar posteriormente utilizando la fórmula.

Skip list

Skip list o lista de saltos en español, consiste en una lista ordenada (al contario en primera instancia, de la DLL) la cual puede ser implementada de diversas maneras, puede hacer uso de listas simplemente enlazadas o de listas doblemente enlazadas.

Haciendo recerca, se puede ver que hay muchas implementaciones de dicho tipo de estructura, pero en el caso de la práctica se nos pide implementar la skip list siguiendo un modelo de lista doblemente enlazada, con nodos que contengan punteros a atrás y hacia delante, pero no solo esto, sino que también ha de contener punteros hacían arriba y hacia abajo, con lo cual la lista acaba teniendo diversos nodos de izquierda a derecha, y algunos de estos hacia arriba también.

Finalmente, debido a que los niveles de la lista y las repeticiones de los nodos hacia arriba en lo que sería la lista, consiste en un sistema de aleatoriedad, acabamos con un numero de log(n) aproximadamente de niveles.

Consideraciones de la skip list

Como acabo de explicar, la skip list no es más que una lista doblemente enlazada (al menos en nuestro caso) en la cual hay referencias a dichos nodos hacia arriba de estos, es decir, cuando creamos un nodo, podemos por ejemplo comenzar a tirar una moneda, si sale cara subimos un nivel, en otras palabras, creamos una copia del nodo a insertar en un nivel más arriba, el cual va a estar referenciado por el “propio” nodo pero en la posición de debajo, y por los nodos que contenga a sus costados en el mismo nivel, así hasta que salga cruz y dejemos de crear niveles.

Con todo esto, la skip list es una lista un tanto impredecible en el sentido de que el número de niveles, y por lo tanto de nodos repetidos, es algo totalmente aleatorio. Aun y así podemos decir que aproximadamente el tiempo de búsqueda de dicha lista es O(logn) ya que facilita mucho el termino de buscar un elemento debido a que con un poco de suerte no tendremos que recorrer todos los elementos, si no tan solo algunos de los niveles superiores hasta encontrar nuestro elemento, y nos iremos saltando varios nodos de elementos que son inferiores al que buscamos.

Grafica resultados de la skip list:

mida	cost	desv.Est			
1000	8	3.64			
2000	8	3.72			
3000	10	4.29			
4000	10	4.34			
5000	11	4.61			
6000	10	4.05			
7000	12	4.83			
8000	13	4.9			
9000	11	4.5			
10000	12	5.01			
11000	13	4.45			
12000	11	4.76			
13000	12	4.27			
14000	12	4.67			
15000	13	4.8			
16000	11	4.3			
17000	12	4.54			
18000	13	4.6			
19000	13	5.11			
20000	14	5.38			
21000	11	4.24			
22000	12	4.05			
23000	14	5.24	37000	13	5.02
24000	14	5.31	38000	13	4.87
25000	12	4.16	39000	13	4.31
26000	13	4.8	40000	13	4.75
27000	13	4.83	41000	14	5.03
28000	13	4.62	42000	13	5.23
29000	14	5.14	43000	14	4.86
30000	12	4.7	44000	14	4.72
31000	13	4.72	45000	15	4.99
32000	14	5.75	46000	13	4.52
33000	13	4.69	47000	12	4.56
34000	13	4.79	48000	12	4.46
35000	13	4.74	49000	14	4.8
36000	14	4.87	50000	15	5.32

Imagen 3: resultados de costo de búsqueda y desviación estándar en listas de 1000 a 50000 elementos

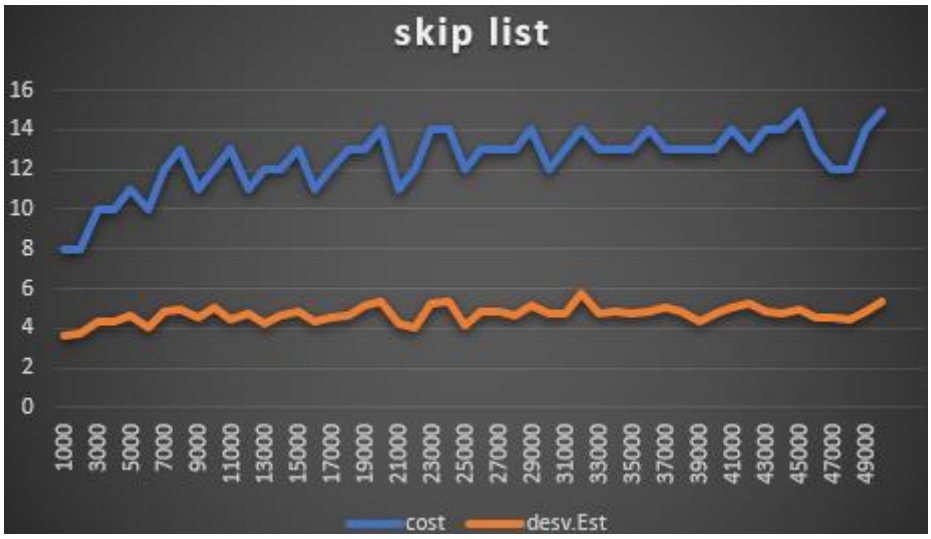


Imagen 4: Gráfica de resultados de la skip list

Como podemos observar tanto en la gráfica como en la tabla de resultados, aquí los resultados de costo de búsqueda son sumamente inferiores a los obtenidos en el caso de la DLL, con lo cual hace de esta una lista mucho mas eficaz en términos de búsqueda.

Podemos observar que el costo de búsqueda acaba siendo logN aproximadamente.

Consideraciones de implementación:

La skip list se trata simplemente como una lista doblemente enlazada con diferentes niveles, con lo cual para implementarla debemos hacer uso de punteros hacia arriba y hacia abajo.

Para poder trabajar con la skip list, hacemos uso de tres funciones auxiliares, random_level la cual nos devuelve el nivel que tendrá el nodo a insertar, y buscarKey y buscarElemIgual, la primera la utilizamos para buscar la posición donde hemos de colocar el elemento para que la lista se encuentre ordenada, siempre nos devolver un puntero al nodo anterior, y el nodo nuevo se colocara inmediatamente después.

En el caso de la segunda función de buscar, la utilizamos para buscar un elemento que sea igual, esta función es utilizada en la función Esborrar, con lo cual puede ser que dicha función nos devuelva NULL debido a que es posible que el elemento a buscar no existe.

El máximo de niveles de la skip list lo definimos como una constante a 15. Cuantos más niveles más eficientes es el tiempo de búsqueda de la lista acercándose a O(logN).

En esta lista al igual que con la DLL los mayores problemas vienen con el tema punteros y el control de acceso a posiciones de memorias nulas.

El problema con este tipo de lista es que impredecible el numero de niveles que va a tener en cada ejecución ya que depende de la función rand, con lo cual seguir la ejecución y controlar todos los errores puede ser una problemática y para seguir los errores hay que ir paso a paso. Por lo tanto, para tratar este punto he optado por ejecutarlo cientos de veces yendo paso a paso y observando como trabaja la lista con el debugger, bajando y subiendo niveles, comprobando que la lista estaba ordenada, que los niveles entre si correspondían al elemento correcto, etc.

Conclusiones finales

A lo largo de la práctica he ido viendo que cada estructura tiene su propio estilo de implementación, y dentro de cada estructura e implementación, hay diversas maneras de implementarlas, desde trabajarlas con un pdi, a simplemente insertar después de un elemento dado, o al final o al principio. Colas estáticas o dinámicas, pilas estáticas o dinámicas, skip list con dll o con listas simples, etc.

Con lo cual podemos observar que no hay ninguna estructura que sea mejor que otra, simplemente se aprecia que cada estructura está diseñada con un propósito.

Si nuestro objetivo es una inserción extremadamente rápida, porque no utilizar una DLL, o una pila, o incluso una cola en vez de una skip list donde tenemos que ir hallando el hueco donde colocar el nodo.

Si por el contrario deseamos una búsqueda rápida está claro que la skip list es la ganadora con diferencia. Para otro tema de datos, o de funcionalidades nos pueden servir más la cola o la pila como ya se ha comentado más arriba.

Por lo tanto, como conclusiones es que hay que saber un poco de cada uno y saber aplicar cada una cuando sea necesario, y en caso de no acordarse o no conocer alguna estructura, ser capaz de buscar información y acabar implementándola.

Si una de las estructuras fuese mejor que todas las demás en todo, ¿Para qué íbamos a estudiar a las otras?

WEBGRAFÍA

https://docs.microsoft.com/en-us/archive/blogs/galstertechblog/implementing-a-skiplist-in-c

https://brilliant.org/wiki/skip-lists/

http://www.mathcs.emory.edu/~cheung/Courses/323/Syllabus/Map/skip-list.html

https://www.youtube.com/watch?v=2g9OSRKJuzM

https://en.wikipedia.org/wiki/Doubly_linked_list

https://www.javatpoint.com/doubly-linked-list