Trabajo Práctico II - Estructuras de datos y algortimos 1

Valentina Prato - Ignacio Kasevich

Junio 2020

1. Estructura del proyecto

Para organizar mejor el proyecto y el trabajo en equipo se buscó tener una buena estructura de proyecto donde se separen de forma clara y limpia los distintos elementos, recursos y utilidades necesarias para este trabajo práctico. El proyecto consta de estos directorios:

```
|
| -- libs
| -- source
| -- tests
|       | -- test_cases
|
```

- **libs**: librerias/headers utlizados, implementaciones de estructuras de datos, librerias de funciones de uso común del proyecto.
- **source**: código fuente de los dos programas principales del proyecto. El interprete y otro programa 'test_shell' para debuggear el funcionamiento de este.
- **tests**: aquí se encuentran dos programas, un test muy simple realizado en c, 'test_dummy' y un programa realizado en python que testea la estructura de árbol en varios puntos utilizando el programa en c 'test_shell'. Además, aquí se en test_cases guardan los resultados de las test de python.

2. Estructura de datos

2.1. Árbol binario balanceado de intervalos

Para la implementación de esta estructura de datos se partió desde la estructura de árbol binario balanceado de números enteros previamente realizada. Para lograr una implementación sencilla y eficiente utilizando la estructura previamente diseñado se buscó encapsular el funcionamiento de los intervalos. Es decir, si antes se comparaban las claves (datos int de la estructura previa) con una función 'menor' para ordenar los datos en el árbol, entonces ahora se debía implementar una función 'intervalo_min', que indique si un intervalo es menor que otro. Para lograr encapsular este comportamiento sólo se necesitaron dos funciones, 'intervalo_min' recién explicada e 'intervalo_interseca' que indica si dos intervalos tienen intersección. Se implementaron otras funciones de intervalo, pero no son requeridas para la estructura de árbol. Los mayores problemas en cuanto a la implementación de esta estructura surgieron con la mejora de guardar el máximo de cada árbol. Ya que este máximo había que actualizarlo cada vez que la estructura del árbol cambiara (por insertar o eliminar un elemento). Aislando este comportamiento de actualizar el máximo con la función 'itree_actualizar_max' y llamando esta función cada vez que la estructura del árbol cambiara (rotaciones izquierda y derecha, insertar y eliminar un nodo) se logró implementar correctamente esta mejora. Dos cosas fueron muy útiles a la hora de implementar y corregir esta estructura: primero, identificar información o procedimientos utilizados múltiples veces para centralizarlos en funciones que realicen estas acciones y así minimizar la posibilidad de error en la estructura y la facilidad de corregir un error a la hora de encontrarlo. En segundo lugar, una función para imprimir un árbol en '2D' es decir, verlo con formato de árbol en la consola, por la utilidad de esta se decidió dejarla en el interprete a través de la función 'print'.

2.2. Cola

Se utilizó una estructura de cola (de puntero void*) solamente para implementar el recorrido bfs (Breadth-first search) en el árbol AVL de intervalos.

3. Tests

Para garantizar el correcto funcionamiento de las funciones y estructuras utilizadas se implementaron múltiples tests, algunos realizados 'a mano' y

otros implementados con programas de ayuda en python y en c. La parte mas importante del testeo fue la realizada en python, allí se escribieron múltiples tests que atacaban distintos aspectos de la estructura. Para correr los tests se hizo un programa intermediario (test_shell) que tomaba como argumento un archivo de instrucciones para el interprete realizado en c. Cada línea de este archivo sería entonces un comando a ejecutar por el interprete. Este programa es exactamente igual (utiliza la misma librería shell.h) que el programa 'interprete' (el programa principal de este proyecto), pero a diferencia que en vez de tomar los comandos por la entrada estándar, toma los comandos del archivo dado como argumento. Partiendo entonces del programa test_shell realizado en c, en python tendríamos una herramienta para correr los tests. Se diseñaron así numerosos tests de inserción, eliminación, intersección, intervalos con números positivos, negativos, chicos, grandes, aleatorios y tamaños muy grandes (100.000 nodos) de árboles. Además, uno de los tests mas útiles, es el test de valgrind, se corre un test pseudo aleatorio, donde hay muchas eliminaciones, inserciones, intersecciones e impresiones y este se hace a través del programa de valgrind, checkando asi que no haya leaks de memoria. A cada uno de estos tests se les tomo el tiempo de ejecución para tener un estimativo de eficacia de la estructura y como los distintos cambios afectaban a la estructura. Como punto negativo en este apartado, es que no se realizaron las comprobaciones de estos tests de forma programada, si no que se comprobaban 'a mano'. Esto resultó asi principalmente por una cuestión de tiempo. Aun con este problema, esta implementación de tests logró encontrar varios errores y problemas en la estructura, por lo cuál si bien no fue la mejor implementación, cumplió bien su función.

3.1. Resultados de tiempos

Los 10 test escritos en python se corrieron 10 veces en 2 máquinas distintas. Antes de ver las tablas de los resultados, veremos un resumen de lo que hace cada tests y de las computadoras donde se corrieron.

3.2. Tests:

- **dummy**: el test mas simple donde se insertan 5 intervalos y se imprime el resultado en '2D'.
- **insert**: se insertan 10000 intervalos enteros de la forma [x, x+1] y se imprime el resultado en '2D' y recorriendo el árbol por BFS y DFS.
- **delete**: se insertan 10000 intervalos enteros de la forma [x, x+1], luego

se eliminan estos mismos intervalos y se imprime el resultado en '2D' y recorriendo el árbol por BFS y DFS.

- **overlap:** se insertan 100 intervalos enteros de la forma [x, x+1], luego se pregunta la intersección por estos mismos intervalos y se imprime el resultado recorriendo el árbol por BFS.
- **negative:** se insertan 100 intervalos enteros de la forma [-1*x, -1*x + 1] (x ¿0), y se imprime el resultado en '2D' y recorriendo el árbol por BFS y DFS.
- **real number:** se insertan 10000 intervalos de la forma [x, x+0.1] (x entero) y se imprime el resultado en '2D' y recorriendo el árbol por BFS y DFS.
- **rand number**: se insertan 10000 intervalos aleatorios (reales) no necesariamente válidos y se imprime el resultado en '2D' y recorriendo el árbol por BFS y DFS.
- **rand v. number** se insertan 10000 intervalos válidos aleatorios (reales) y se imprime el resultado en '2D' y recorriendo el árbol por BFS y DFS.
- **valgrind**: testeo general con 10000 intervalos aleatorios reales, inserta, elimina e interseca intervalos de forma aleatoria y se imprime el resultado en '2D' y recorriendo el árbol por BFS y DFS. Todo este proceso se corre a través de valgrind el cual imprime por consola el resultado (sólo se incluyo uno de los resultados en el apartado 'Detalle de Valgrind' pero todos los tests corrieron sin leaks).
- **size**: se insertan 100000 intervalos enteros de la forma [x, x+1] y se imprime el resultado recorriendo el árbol por BFS.

3.3. Computadoras:

- **PC 1**: i7 4790k @4.00GHz - 16GB de ram @ 1333MHz - SSD (Linux)
- **PC 2**: i3 - 8 GB de Ram (Linux)

A continuación se muestran algunos resultados de medición de tiempo en los diferentes de tests. La última línea es el promedio de los valores.

Cuadro 1: PC 1 (en segundos)

dummy	insert	delete	overlap	negative	real number	rand number	rand v. number	valgrind	size
0.0010	0.6677	1.0465	0.0012	0.0010	0.6573	0.2150	1.0403	7.5898	76.5525
0.0012	0.6520	1.0121	0.0014	0.0015	0.6627	0.2206	0.9904	7.3972	76.6895
0.0012	0.6516	1.0154	0.0014	0.0015	0.6614	0.2192	0.9952	7.3026	76.8492
0.0011	0.6533	1.0276	0.0012	0.0013	0.6650	0.2324	1.0194	7.1889	76.4803
0.0011	0.6554	1.0143	0.0013	0.0013	0.6530	0.2216	0.9776	7.5558	75.9139
0.0011	0.6464	1.0217	0.0014	0.0014	0.6596	0.2199	0.9986	7.2587	76.1080
0.0012	0.6585	1.0190	0.0015	0.0014	0.6620	0.2221	1.0106	7.3668	77.1468
0.0013	0.6500	1.0109	0.0013	0.0014	0.6588	0.2259	0.9890	7.4477	75.8933
0.0011	0.6543	1.0142	0.0013	0.0013	0.6532	0.2303	0.9954	7.3635	76.6960
0.0012	0.6525	1.0240	0.0014	0.0013	0.6628	0.2215	0.9836	7.2285	76.6826
0.0011	**0.6542**	**1.0206**	**0.0013**	**0.0013**	**0.6596**	**0.2228**	**1.0000**	**7.3699**	**76.5012**

*la última línea es el promedio de los valores.

Cuadro 2: PC 2 (en segundos)

dummy	insert	delete	overlap	negative	real number	rand number	rand v. number	valgrind	size
0.0013	1.1442	1.8102	0.0015	0.0016	1.1476	0.4177	1.8244	13.1444	158.9486
0.0013	1.1469	1.8202	0.0015	0.0015	1.1406	0.4220	1.8543	13.0886	160.1213
0.0012	1.1331	1.8132	0.0015	0.0015	1.1453	0.4090	1.8361	12.5926	160.9481
0.0012	1.1408	1.8138	0.0015	0.0016	1.1422	0.4213	1.8444	12.6327	157.8776
0.0012	1.1420	1.8044	0.0015	0.0016	1.1480	0.4168	1.8410	12.8447	159.0193
0.0012	1.1246	1.8033	0.0016	0.0016	1.1462	0.4191	1.8214	13.3749	159.6415
0.0012	1.1331	1.8013	0.0015	0.0016	1.1507	0.3996	1.8389	13.1698	159.8144
0.0012	1.1482	1.8160	0.0015	0.0016	1.1465	0.4144	1.8157	12.9019	159.6480
0.0012	1.1383	1.8085	0.0015	0.0016	1.1402	0.4193	1.8146	12.9590	159.3566
0.0012	1.1334	1.8101	0.0015	0.0015	1.1556	0.4053	1.8035	12.7743	162.6823
0.0012	**1.1385**	**1.8101**	**0.0015**	**0.0016**	**1.1463**	**0.4144**	**1.8294**	**12.9483**	**159.8058**

*la última línea es el promedio de los valores.

4. Makefile

El archivo Makefile generado posee varias comandos, a continuación se detallan los mismos:

- **make / make all**: compila el programa principal 'interprete' y limpia los archivos objeto .o.
- **make test**: compila el programa test_shell y llama a los tests de python y limpia los archivos objeto .o.
- **make test_shell**: compila el programa test_shell y limpia los archivos objeto .o.
- **make test_dummy**: compila el programa test_dummy y limpia los archivos objeto .o.
- **make unit_test_python**: ejecuta los tests de python.

5. Detalles de Valgrind

5.1. Intreprete

Comando ejecutado: valgrind -v –leak-check=full –show-reachable=yes ./interprete

```
==76295==
==76295== HEAP SUMMARY:
==76295== in use at exit: 0 bytes in 0 blocks
==76295== total heap usage: 5 allocs, 5 frees, 2,144 bytes allocated
==76295==
==76295== All heap blocks were freed – no leaks are possible
==76295==
==76295== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Comando ejecutado: make test (extracto de uno de los tests)

```
==79180==
==79180== HEAP SUMMARY:
==79180== in use at exit: 0 bytes in 0 blocks
==79180== total heap usage: 9,555 allocs, 9,555 frees, 281,688 bytes allocated
==79180==
==79180== All heap blocks were freed – no leaks are possible
==79180==
==79180== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
End test: test_valgrind
*** Execution time: 7.530091285705566 seg.***
```

El resultado recién mostrado se consiguió a través de uno de los tests realizados en python. Si se llama al comando **make test** en uno de los tests se correra valgrind con el programa test_shell, el cual utiliza las mismas funciones que el programa interprete salvo que lee la entrada de un archivo en vez de la entrada estándar.

6. Referencias

Árbol AVL inserción: https://www.geeksforgeeks.org/avl-tree-set-1-insertion/

Árbol AVL eliminación: https://www.geeksforgeeks.org/avl-tree-set-2-deletion/

Árbol AVL intervalos: https://www.geeksforgeeks.org/interval-tree/