Rope

Ejercicio N° 1

Objetivos	• Buenas prácticas en programación de Tipos de Datos Abstractos (TDAs) • Modularización de sistemas • Correcto uso de recursos (memoria dinámica y archivos) • Encapsulación y manejo de Sockets
Instancias de Entrega	**Entrega 1:** clase 4 (05/09/2017). **Entrega 2:** clase 6 (19/09/2017).
Temas de Repaso	• Uso de structs y typedef • Uso de macros y archivos de cabecera • Funciones para el manejo de Strings en C • Funciones para el manejo de Sockets
Criterios de Evaluación	• Criterios de ejercicios anteriores • Cumplimiento de la totalidad del enunciado del ejercicio • Ausencia de variables globales • Ausencia de funciones globales salvo los puntos de entrada al sistema (*main*) • Correcta encapsulación en TDAs y separación en archivos • Uso de interfaces para acceder a datos contenidos en TDAs • Empleo de memoria dinámica de forma ordenada y moderada • Acceso a información de archivos de forma ordenada y moderada

Índice

Introducción

Como parte de un proyecto para la creación de un editor de textos cooperativo, similar a Google Docs, se nos pidió realizar un prototipo simplificado del servidor quien mantendrá los documentos y que dará soporte a las operaciones de inserción y borrado de texto.
La idea detrás del prototipo es poder ver el desempeño de la estructura de datos *rope*.

Descripción

Se implementarán un cliente y un servidor que deben comunicarse mediante sockets TCP.

A diferencia de Google Docs que soporta múltiples documentos siendo editados por múltiples usuarios, por tratarse de un prototipo, el servidor sólo necesita procesar las acciones de un único cliente con un único documento.
Una vez que el cliente se desconecte del servidor, este debe finalizar.

Se nos requirió implementar una estructura de datos particular en el servidor para el manejo del documento para verificar su rendimiento y escalabilidad. La estructura en cuestión es un árbol *rope*.

Para poder realizar las pruebas se requiere que el cliente lea de un archivo, o entrada estándar, una serie de comandos representando las acciones que podría realizar un usuario real.

El cliente debe enviar estos comandos al servidor vía sockets TCP[4] y realizar allí las operaciones de inserción y borrado.

Para simplificar el desarrollo se deberá implementar un único programa que en función de un parámetro se comportará como un cliente o como un servidor.

Estructura de datos *rope* del servidor

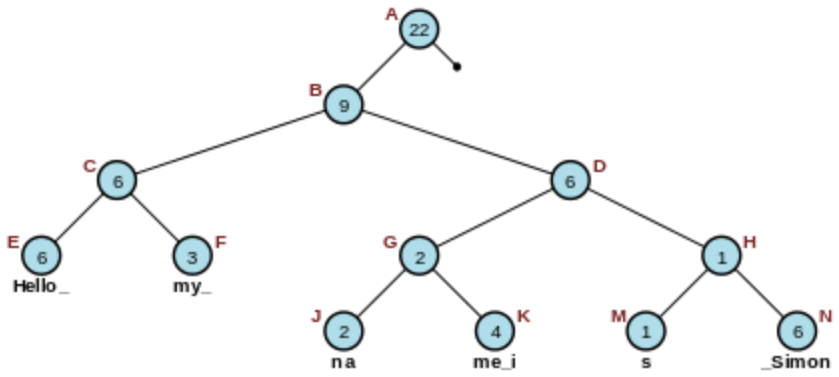
Realizar operaciones de inserción y borrado sobre un largo *string* requiere de una estructura de datos más flexible que un simple *array* contiguo en memoria.

Un *array* requiere potencialmente copiar todo el contenido para hacer lugar a una inserción o para compactar el *array* tras un borrado.

Existen distintas alternativas como una *linked list, un gap buffer*[1] o una *rope*[2].
En el presente trabajo se implementará la estructura de datos *rope*.

La estructura *rope* consiste en un árbol binario donde los nodos hoja contienen fragmentos del *string*. En cambio, los nodos no-hoja contienen la longitud total de los fragmentos que están a su izquierda.

El siguiente diagrama muestra un ejemplo de una estructura *rope*[2]:



Estructura Rope almacenando el texto “Hello_my_name_is_Simon” Ref: Wikipedia

En este ejemplo el texto `Hello_my_name_is_Simon` está contenido en los nodos hoja, fragmentado en múltiples partes debido a las inserciones y borrados que el texto original recibió.
Nótese como los nodos no-hoja contienen la longitud total de los fragmentos a la izquierda:
El nodo C tiene un valor de 6 por que el fragmento a su izquierda, `Hello_`, tiene 6 caracteres.
El nodo B tiene un valor de 9 por que en total los fragmentos `Hello_` y `my_` suman 6+3=9 caracteres.

> La mayoría de la operaciones dependen de la cantidad de niveles del arbol asi que es más óptimo trabajar con arboles balanceados.
> También, dado que los nodo hojas tienen fragmentos de texto, si estos son inmutables se los puede reutilizar y compartir múltiples veces.
> **Ninguna** de estas optimizaciones es requerida para el presente trabajo. Los tiempos y recursos son limitados, es prudente enfocarse solamente en aquello que es indispensable.

Toda estructura rope soporta al menos 2 operaciones elementales o primitivas:

- *Split*: dada una posición `p` separa el árbol en 2 árboles. El árbol de la izquierda contiene todos los fragmentos que se encuentran a la izquierda de la posición `p`. El árbol de la derecha contiene los fragmentos restantes.
- *Concat* (o *join*): dado 2 árboles, se construye un único árbol con la concatenación de los 2.

Sobre estas primitivas se pueden construir las operaciones de inserción (*insert*) y borrado (*delete*) de texto.

- *Insert*: dada una posición `p` y un texto `t`, se inserta este texto en el árbol. Esta operación puede ser realizada con un *split* en la posición `p` en dos árboles `L` y `R`, la creación de un árbol temporal `T` que solo contenga a `t` y la concatenación de `L` con `T` y este con `R`.
- *Delete*: dada las posiciones `p` y `q` donde se quiere borrar el texto entre `p` y `q`, la operación *delete* se puede implementar con 2 *splits* en `p` y `q` y una concatenación.

Sugerencia:
Implemente primero las primitivas *split* y *concat* o *join*. Verifique que funcionan correctamente haciendo algunas pruebas. Sólo entonces construya las operaciones de inserción y borrado.
Construya **siempre sobre bloques** y funcionalidades testeadas primero, jamás implemente todo en un solo intento, es más difícil de debuggear.

Cliente

El cliente debe leer de la entrada estándar o de un archivo los comandos para manipular el texto. Estos comandos deben ser enviados al servidor y será él quien los realice sobre la estructura *rope*.

En general, los argumentos estarán separados por al menos un espacio y los comandos se separan por saltos de línea **\n**. Las líneas vacías deben ser ignoradas.

Sugerencia:
Investigue **en profundidad** la familia de funciones **scanf** [3].

El formato de los comandos son:

- **`insert <pos> <text>`**
  Inserta el texto `<text>` en la posición `<pos>`.
  El texto `<text>` finaliza hasta el salto de línea y no contendrá ningún espacio.
  La posición `<pos>` es un número entero que puede ser tanto positivo como negativo.
  La mejor manera de explicar el significado de este parámetro es a través de ejemplos. Dado un *rope* con el texto `foo`,
    - Si se hace un `insert  0 x`, el resultado sería `xfoo`.
    - Si se hace un `insert  1 x`, el resultado sería `fxoo`.
    - Si se hace un `insert  2 x`, el resultado sería `foxo`.
    - Si se hace un `insert -1 x`, el resultado sería `foox`.
    - Si se hace un `insert -2 x`, el resultado sería `foxo`.

  Se garantiza que siempre las posiciones serán válidas y que están dentro de los límites del texto (no habrán inserciones fuera de rango).

- **`delete <to> <from>`**
  Borra los caracteres entre las posiciones `<to>` y `<from>`.
  Al igual que en las inserciones las posiciones pueden ser negativas.
  Por ejemplo, dado un *rope* con el texto `foobar`,
    - Si se hace un `delete  0  2`, el resultado sería `obar`.
    - Si se hace un `delete  1  3`, el resultado sería `fbar`.
    - Si se hace un `delete -2 -1`, el resultado sería `foob`.
    - Si se hace un `delete  0 -1`, el resultado sería vacío.

  Nótese como el par `<to>` y `<from>` forman un rango. Se garantiza que siempre será un rango válido.

- **`space <pos>`**
  Inserta un espacio en la posición `<pos>`. La posición tiene las mismas consideraciones que las vistas en el comando `insert`.

- **`newline <pos>`**
  Inserta un salto de línea en la posición `<pos>`. La posición tiene las mismas consideraciones que las vistas en el comando `insert`.

- **`print`**
  Este es el único comando que no modifica a la estructura *rope*. El cliente le pide al servidor el texto completo para luego imprimirlo por salida estándar.

> **Sugerencia:**
> Implemente una estructura *rope* que soporte posiciones positivas y sólo luego de haber verificado su correcto funcionamiento agregue soporte para posiciones negativas.

Se garantiza que siempre la primer instrucción del archivo será de la forma:

```
insert 0 <text>
```

y servirá para inicializar la estructura *rope*.

Protocolo de comunicación

Cada comando que el cliente lee de su entrada estándar o archivo se lo debe enviar al servidor para que lo ejecute.
Ninguno de los comandos requiere una respuesta por parte del servidor salvo el comando `print`.

Todos los comandos son enviados en binario, en big endian order, con signo, alineados a 1 byte, sin *padding* y prefijados con 4 bytes (**int**) que indican el tipo de comando (*opcode*).

`insert`

opcode:	4 bytes (debe valer **1**)
pos:	4 bytes
longitud:	2 bytes (es la longitud del texto a insertar)
texto:	**n** bytes (donde **n** es la longitud)

`delete`

opcode:	4 bytes (debe valer **2**)
to:	4 bytes
from:	4 bytes

`space`

opcode:	4 bytes (debe valer **3**)
pos:	4 bytes

`newline`

opcode: 4 bytes (debe valer **4**)
pos: 4 bytes

`print`
opcode: 4 bytes (debe valer **5**)

La respuesta del servidor ante un comando de tipo `print` debe tener el siguiente formato:

response
longitud: 4 bytes (es la longitud del texto completo a enviar)
texto: **n** bytes (donde **n** es la longitud)

> **Sugerencia:**
> Implemente la funcionalidad de a bloques, de forma **iterativa**.
> Primero un único ejecutable con la implementacion de rope previamente testeada.
> Sobre este, implemente el parser de los comandos del cliente. Solo luego de verificar que funciona correctamente separe el proyecto en los dos programas requeridos y comuníquelos vía sockets.

Formato de Línea de Comandos

La línea de comandos para el servidor es:

```
./tp server <port>
```

Donde el primer parámetro `server` le instruye al programa que debe comportarse como el servidor; donde `<port>` es el número de puerto donde esperará la conexión entrante.

La línea de comandos para el cliente es:

```
./tp client <host> <port> [<input>]
```

Donde el primer parámetro `client` le instruye al programa que debe comportarse como el cliente; donde `<host>` y `<port>` es la dirección y número de puerto del servidor.
El parámetro `<input>` es opcional, y contiene los comandos. Si el parámetro no está presente, el cliente debe leer los comandos desde la entrada estándar.

Códigos de Retorno

Tanto el cliente como el servidor deben retornar -1 como código de retorno si la cantidad de parámetros es incorrecta; 0 en otro caso.

Entrada y Salida Estándar

El servidor no interactúa ni con la entrada ni con la salida estándar.

El cliente debe leer los comandos desde su entrada estándar en caso de no indicarse un archivo por línea de

comando.
Además, el cliente debe imprimir por salida estándar los resultados de los comandos `print`.

Ejemplos de Ejecución

Sea el archivo input.txt:

```
insert  0  olamundo
insert  0  H
insert -1  !
insert  4  -
print
```

Sea un servidor corriendo en paralelo

```
./tp server 9898
```

Y sea un cliente ejecutado como

```
./tp client 127.0.0.1 9898 input.txt
```

El cliente debe enviar los siguientes mensajes al servidor vía sockets. Se muestra a continuación una captura de los mensajes usando `nc` y mostrando los bytes transferidos hacia el servidor usando `hexdump`. Como ayuda, se resalta en negrita los bytes que representan los *opcodes*.

```
00000000  00 00 00 01 00 00 00 00  00 08 6f 6c 61 6d 75 6e  |..........olamun|
00000010  64 6f 00 00 00 01 00 00  00 00 00 01 48 00 00 00  |do..........H...|
00000020  01 ff ff ff ff 00 01 21  00 00 00 01 00 00 00 04  |.......!........|
00000030  00 01 2d 00 00 00 05                              |..-....|
00000037
```

El cliente no recibe respuesta alguna por parte del servidor salvo por el comando `print`.
En este caso el servidor envía por sockets:

```
00000000  00 00 00 0b 48 6f 6c 61  2d 6d 75 6e 64 6f 21     |....Hola-mundo!|
0000000f
```

Y el cliente imprime por salida estándar

```
Hola-mundo!
```

Restricciones

La siguiente es una lista de restricciones técnicas exigidas por el cliente:

1. El sistema debe desarrollarse en ISO C (C99)
2. Está prohibido el uso de variables globales.
3. El archivo de entrada del cliente sólo puede leerse una vez y no puede cargarse a memoria. Debe procesarse a medida que se lee.

Referencias

[1] Gap buffer: https://en.wikipedia.org/wiki/Gap_buffer
[2] Rope: https://en.wikipedia.org/wiki/Rope_(data_structure)
[3] C library: http://www.cplusplus.com/reference/clibrary/
[4] Socket: https://linux.die.net/man/7/socket