Organización de computadoras

Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur
Segundo Cuatrimestre de 2018

Proyecto N° 1
Programación en lenguaje C

Propósito

El objetivo principal del proyecto es implementar en lenguaje C un programa que le permita al usuario, visualizar información referente a posibles planes de viajes para un conjunto de ciudades que desea visitar. Con este objetivo se debe implementar:

- *TDA Lista*, para almacenar elementos de tipo genérico.
- *TDA Cola Con Prioridad*, para almacenar entradas con clave y valor de tipo genérico, ordenadas en función de su prioridad. La prioridad de las entradas será determinada por una función diseñada específicamente para ese propósito.
- Un *programa principal*, el cual debe tomar como argumento por línea de comandos el nombre de un archivo de texto y a partir de este determinar la ubicación actual y las ciudades a visitar por el usuario, permitiendo luego un conjunto de operaciones sobre estos datos.

1. TDA Lista

Implementar un *TDA Lista* en lenguaje C, cuyos elementos sean punteros genéricos. La lista debe ser doblemente enlazada, cuya implementación provea las siguientes operaciones:

1. `TLista crear_lista()` Crea y retorna una lista vacía.
2. `int l_insertar(TLista *lista, TPosicion pos, TElemento elem)` Agrega el elemento `elem` en la posición `pos`, dentro de la lista. Si `pos` es `POS_NULA`, inserta el elemento en la primera posición de la lista. Retorna verdadero si procede con éxito, falso en caso contrario.
3. `int l_eliminar(TLista *lista, TPosicion pos)` Elimina el elemento en la posición `pos`. Reacomoda la lista adecuadamente. Retorna verdadero si procede con éxito, falso en caso contrario.

   0 para falso
   1 para verdadero
4. `TPosicion l_primera(TLista lista)` Retorna la primera posición de la lista. Si la lista es vacía, retorna `POS_NULA`.
5. `TPosicion l_ultima(TLista lista)` Retorna la última posición de la lista. Si la lista es vacía, retorna `POS_NULA`.
6. `TPosicion l_anterior(TLista lista, TPosicion pos)` Retorna la posición anterior a `pos` en la lista `lista`. Si `pos` es la primera posición de la lista, retorna `POS_NULA`.

7. `TPosicion l_siguiente(TLista lista, TPosicion pos)` Retorna la posición siguiente a `pos` en la lista `lista`. Si `pos` es la última posición de la lista, retorna `POS_NULA`.

8. `TElemento l_recuperar(TLista lista, TPosicion pos)` Retorna el elemento correspondiente a la posición `pos`. Si la posición es `POS_NULA`, retorna `ELE_NULO`.

9. `int l_size(TLista lista)` Retorna la cantidad de elementos de la lista.

10. `int l_destruir(TLista *lista)` Elimina todos los elementos y libera toda la memoria utilizada por la lista `lista`. Retorna verdadero si procede con éxito, falso en caso contrario.

En los casos anteriormente indicados, sin considerar la operación `crear_lista`, si la lista parametrizada no está inicializada, se debe abortar con *exit status* `LST_NO_INI`.

Para la implementación se debe considerar que los tipos `TLista`, `TPosicion` y `TElemento`, están definidos de la siguiente manera:

```
typedef struct celda {
    TElemento elemento;
    struct celda *celda_anterior;
    struct celda *celda_siguiente;
} *TLista;


typedef struct celda *TPosicion;
typedef void *TElemento;
```

2. TDA Cola Con Prioridad

Implementar un *TDA Cola Con Prioridad* en lenguaje C, mediante una estructura *Heap*, cuyos elementos sean entradas con clave y valor como punteros genéricos. El orden en que las entradas se retiran de la cola se especifica al momento de la creación, a través de una función de prioridad. La implementación debe proveer las siguientes operaciones:

1. `TColaCP crear_cola_cp(int (*f)(TEntrada, TEntrada))` Crea y retorna una cola con prioridad vacía. El orden en que las entradas deben ser retiradas de la cola estará dado por la función de prioridad `int f(TEntrada, TEntrada)`. Se considera que la función `f` devuelve -1 si la clave de la entrada como primer argumento tiene menor prioridad que la clave de la entrada del segundo argumento, 0 si la prioridad es la misma, y 1 si la prioridad es mayor.

2. `int cp_insertar(TColaCP cola, TEntrada entr)` Agrega la entrada `entr` en la cola. Retorna verdadero si procede con éxito, falso en caso contrario.

3. `TEntrada cp_eliminar(TColaCP cola)` Elimina y retorna la entrada con mayor prioridad de la cola. Reacomoda la estructura heap de forma consistente. Si la cola es vacía, retorna `ELE_NULO`.

4. `int cp_size(TColaCP cola)` Retorna la cantidad de entradas de la cola.

5. `int cp_destruir(TColaCP cola)` Elimina todos los elementos y libera toda la memoria utilizada por la cola `cola`. Retorna verdadero si procede con éxito, falso en caso contrario.

En los casos anteriormente indicados, sin considerar la operación `crear_cola_cp`, si la cola parametrizada no está inicializada, se debe abortar con *exit status* `CCP_NO_INI`.

Para la implementación, se debe considerar que `TColaCP`, `TNodo`, `TEntrada`, `TClave` y `TValor` están definidos de la siguiente manera:

```
typedef struct cola_con_prioridad {
   unsigned int cantidad_elementos;
   TNodo raiz;
} *TColaCP;

typedef struct nodo {
   TEntrada entrada;
   struct nodo *padre;
   struct nodo *hijo_izquierdo;
   struct nodo *hijo_derecho;
} *TNodo;

typedef struct entrada {
   TClave clave;
   TValor valor;
} *TEntrada;

typedef void *TClave;
typedef void *TValor;
```

3. Programa Principal

Implementar una aplicación de consola que, recibiendo como argumento por línea de comandos el nombre de un archivo de texto con el formato indicado en el *Ejemplo 1*, reconozca y mantenga la ubicación actual del usuario y una lista de ciudades a visitar. El programa debe ofrecer un menú de operaciones, con las que el usuario luego puede:

1. **Mostrar ascendente:** permite visualizar el listado de todas las ciudades a visitar, ordenadas de forma ascendente en función de la distancia que existe entre la ubicación de estas ciudades y la ubicación actual del usuario.

2. **Mostrar descendente:** permite visualizar el listado de todas las ciudades a visitar, ordenadas de forma descendente en función de la distancia que existe entre la ubicación de estas ciudades y la ubicación actual del usuario.

3. **Reducir horas manejo:** permite visualizar un listado con el orden en el que todas las ciudades a visitar deben ser visitadas, de forma tal que el usuario ubicado en una ciudad de origen conduzca siempre a la próxima ciudad más cercana al origen, reduciendo las horas de manejo entre las ciudades visitadas. Finalmente, se debe listar la distancia total recorrida con esta planificación.

4. **Salir:** permite finalizar el programa liberando toda la memoria utilizada para su funcionamiento.

El programa implementado, denominado `planificador`, debe conformar la siguiente especificación al ser invocado desde la línea de comandos:

```
$ planificador <archivo_texto>
```

El parámetro `archivo_texto`, indica el archivo a partir del cual se conocerá la ubicación actual del usuario, y la lista de ciudades que desea visitar. En caso de que la invocación no sea la indicada, se debe mostrar un mensaje indicando el error, y finalizar la ejecución.

Consideraciones para el programa principal:

1. Una ciudad será representada a través de un nombre, y una ubicación $< X, Y >$. Considerar para esto el `TCiudad` especificado luego.

2. La distancia entre dos ciudades deberá calcularse mediante *Distancia de Manhattan*, esto es, $|X_2 - X_1| + |Y_2 - Y_1|$.

3. Para mantener el listado de ciudades a visitar, obtenidas desde el archivo de texto, se deberá utilizar el *TDA Lista*.

4. Para implementar las operaciones **Mostrar ascendente**, **Mostrar descendente**, y **Reducir horas de manejo**, se deberá utilizar adecuadamente el listado de ciudades a visitar, junto con el *TDA Cola Con Prioridad*, especificando en cada caso la función de prioridad que permita realizar lo solicitado. No se considerará válida ninguna otra solución que no haga uso de estos TDAs.

```
typedef struct ciudad {
   char *nombre;
   float pos_x;
   float pos_y;
} *TCiudad;
```

Ejemplo 1

Considere a modo de ejemplo, el funcionamiento del programa solicitado en función a la siguiente invocación: `$ planificador viajes.txt`

viajes.txt:	**Mostrar ascendente:**	**Mostrar descendente:**	**Reducir horas manejo**
1;1	1. Salliqueló.	1. Bahía Blanca.	1. Salliqueló.
Salliqueló;2;2	2. Carhué.	2. Trenque Lauquen.	2. Carhué.
Bahía Blanca;4;4	3. Trenque Lauquen.	3. Carhué.	3. Bahía Blanca.
Trenque Lauquen;4;0	4. Bahía Blanca.	4. Salliqueló.	4. Trenque Lauquen.
Carhué;0;3			Total recorrido: 14.

Del ejemplo se puede deducir que, el usuario se encuentra actualmente en la ubicación $< 1; 1 >$, y que Salliqueló es una ciudad que el usuario desea visitar al igual que las ciudades de Bahía Blanca, Trenque Lauquen y Carhué. En particular, Carhué se encuentra en la ubicación $< X, Y > = < 0, 3 >$.

Sobre las constantes a utilizar

Se considerarán los siguientes valores para las constantes definidas en las especificaciones de las operaciones del proyecto:

Constante	***Valor***	***Significado***
`FALSE`	`0`	Valor lógico falso.
`TRUE`	`1`	Valor lógico verdadero.
`LST_NO_INI`	`2`	Intento de acceso inválido sobre lista sin inicializar.
`LST_VAC`	`3`	Intento de acceso a posición inválida en lista.
`CCP_NO_INI`	`4`	Intento de acceso inválido sobre lista vacía.
`POS_NULA`	`5`	Intento de acceso inválido sobre Cola CP sin inicializar.
`ELE_NULO`	`NULL`	Posición nula.
	`NULL`	Elemento nulo.

Sobre la implementación

- Los archivos fuente principales se deben denominar **lista.c**, **colacp.c** y **planificador.c** respectivamente. En el caso de las librerías, también se deben adjuntar los respectivos archivos de encabezados **lista.h y colacp.h**, los cuales han de ser incluidos en los archivos fuente de los programas que hagan uso de las mismas.
- Es importante que durante la implementación del proyecto se haga un uso cuidadoso y eficiente de la memoria, tanto para reservar (`malloc`), como para liberar (`free`) el espacio asociado a variables y estructuras.
- Se deben respetar con exactitud los nombres de tipos y encabezados de funciones especificados en el enunciado. Los proyectos que no cumplan esta condición quedarán automáticamente desaprobados.
- La compilación debe realizarse con el *flag* `-Wall` habilitado. El código debe compilar **sin advertencias** de ningún tipo.
- La copia o plagio del proyecto es una falta grave. Quien incurra en estos actos de deshonestidad académica, desaprobará automáticamente el proyecto.

Sobre el estilo de programación

- El código implementado debe reflejar la aplicación de las técnicas de programación modular estudiadas a lo largo de la carrera.
- En el código, entre eficiencia y claridad, se debe optar por la claridad. Toda decisión en este sentido debe constar en la documentación que acompaña al programa implementado.
- El código debe estar indentado, comentado, y debe reflejar el uso adecuado de nombres significativos para la definición de variables, funciones y parámetros.

Sobre la documentación

Los proyectos que no incluyan documentación estarán automáticamente desaprobados. La misma debe:

- Estar dirigida a desarrolladores.
- Explicar detalladamente los programas realizados, incluyendo el diseño de la aplicación y el modelo de datos utilizado, así como toda decisión de diseño tomada, y toda observación que se considere pertinente.
- Incluir explicación de todas las funciones implementadas, indicando su prototipo y el uso de los parámetros de entrada y de salida (tanto dentro del código fuente como en la documentación del proyecto). Se espera que la explicación no sea una mera copia del código fuente, sino más bien una síntesis de lo implementado a través de diagramas, pseudocódigos, o cualquier representación que considere adecuada.
- En general, se deben respetar las consignas indicadas en la "Guía para la documentación de proyectos de software" entregada por la cátedra.

Sobre la entrega

Toda comisión que no cumpla con los requerimientos, estará automáticamente desaprobada. Los mismos son:

- Las comisiones estarán conformadas por 2 alumnos, y serán las que oportunamente registró y notificó la cátedra.
- La entrega del código fuente y la documentación se realizará a través de un archivo comprimido **zip** o **rar**, denominado ***PR1_COMISION_X***, que debe incluir las siguientes carpetas:
  - **Fuentes**, donde se deben incorporar los archivos fuente ".c" y ".h" (ningún otro).
  - **Documentación**, donde se debe incorporar el informe del proyecto en formato PDF (ningún otro).
- El archivo comprimido debe enviarse por e-mail, respetando el siguiente formato:
  - **Para:** *gabriela.diaz@cs.uns.edu.ar*
  - **Asunto:** *[OC][PR1][COM XX] Apellido1 - Apellido2*
  - **Cuerpo del e-mail:**
    Se adjunta Proyecto Nº 1, de la comisión XX:
    Apellido, Nombre 1 - LU 1
    Apellido, Nombre 2 - LU 2
- El e-mail debe ser enviado con anterioridad al día **Martes 09 de Octubre de 2018, 23:59 hs**. Se considerará como hora de ingreso, la registrada en el servidor de e-mail del DCIC.

Sobre la corrección

- La cátedra evaluará tanto el **diseño** e **implementación** como la **documentación** y **presentación** del proyecto, y el cumpliemiento de **todas** las condiciones de entrega.
- Tanto para compilar el proyecto, como para verificar su funcionamiento, se utilizará la máquina virtual "OCUNS" publicada en el sitio web de la cátedra.