Planificador de viajes 2022

Agustín Vera - Franco Leon

ORGANIZACIÓN DE COMPUTADORAS

COMISION 3

ÍNDICE

1- Definiciones y especificación de requerimientos:

Definición general del proyecto de software:

La funcionalidad de *Planificador* es la de, en base a un archivo txt, donde se encuentran ciudades con sus respectivas coordenadas **x** e **y**, procesar la información del archivo y brindar mediante las funciones correspondientes una determinada visualización de las ciudades.
El usuario tiene la posibilidad de ordenar dichas ciudades de forma ascendente y descendente dependiendo la distancia en la que se encuentren, y además puede organizar su viaje para poder reducir sus horas de manejo. Todo esto mediante un menú de usuario ejecutado por consola.
El propósito general del proyecto, es poder brindarle al usuario una herramienta para poder organizar sus viajes de la forma que deseen.

Especificación de requerimientos del proyecto:

- **Requisitos generales:** Dado un archivo txt con nombres de ciudades y sus coordenadas respectivamente, el sistema es capaz de procesarlo y brindar un menú de operaciones al usuario.

- **Requisitos funcionales:** En el inciso anterior se mencionó que el proyecto brinda un menú al usuario en el cual se visualizan cuatro opciones:
    - 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.
    - Mostrar Descendente: misma funcionalidad que “Mostrar Ascendente” pero de forma descendente.
    - Reducir Horas de 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.
    - Salir: permite finalizar el programa liberando toda la memoria utilizada para su funcionamiento.

- **Información de autoría y Legacy del proyecto:** El proyecto fue desarrollado desde el comienzo hasta el final por los autores mencionados anteriormente. Se incluyeron las definiciones correspondientes impuestas en el enunciado, y además utilizamos una segunda estructura de datos provista por la cátedra.

Todo el proceso fue llevado a cabo en Windows 10 (Versión 22H2) y en Windows 11 (Versión 22H2) en el cual se asegura el correcto funcionamiento. No se asegura la total compatibilidad con otros sistemas operativos ya que hasta el momento no ha sido testeado.

Procedimientos de instalación y prueba

- **Planificación:** Iniciamos el proyecto desarrollando el TDA Cola con Prioridad para poder almacenar la información correspondiente solicitada en el enunciado. Además, añadimos un segundo TDA Lista simple para poder operar en tiempo real con memoria dinámica y así evitar la lectura en disco. Concluyendo con los TDA, seguimos con el diseño e implementación del programa principal, que utiliza los servicios provistos por ambos TDA para almacenar la información pertinente.

- **Procedimientos de desarrollo:** Se hizo uso del lenguaje de programación C en el entorno de desarrollo integrado Code::Blocks y el control de versiones de GitHub.

- **Proceso de instalación y prueba:** Al momento de ejecutar el programa se deben seguir una serie de pasos:
  1. Ejecutar la consola en el directorio donde se encuentre el programa.
  2. Ingresar la siguiente línea de comandos, donde *gcc* es el compilador utilizado, los archivos con extensión .c son los módulos de código y *Planificador.exe* es el nombre del programa. Si lo desea, puede cambiar el nombre del programa.
     gcc -o Planificador.exe planificador.c colacp.c Lista.c
  3. Una vez compilado el código anterior, se generará un ejecutable *Planificador.exe* correspondiente al programa en cuestión.

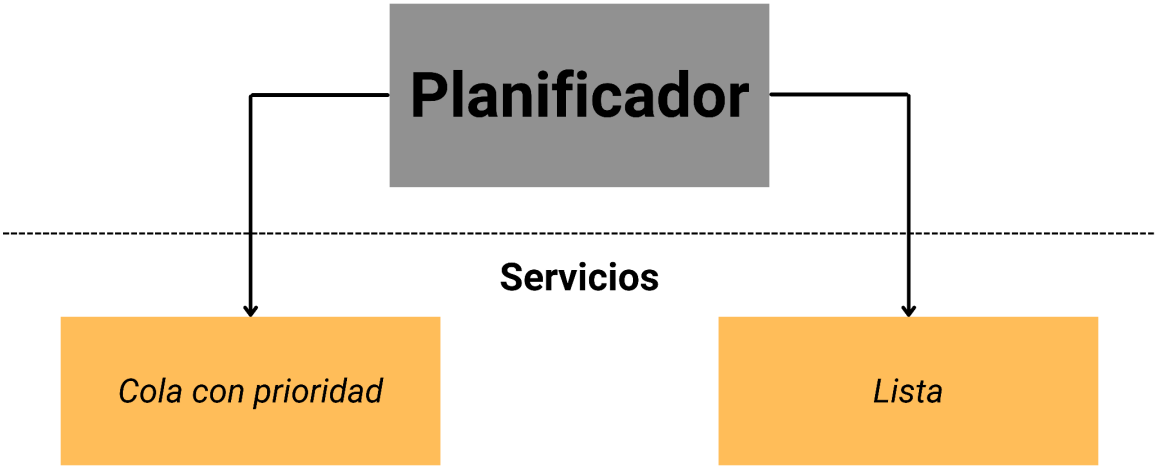
  Es fundamental para llevar a cabo la instalación contar con los servicios del compilador **GCC,** de lo contrario producirá un error.

2- Arquitectura del sistema:

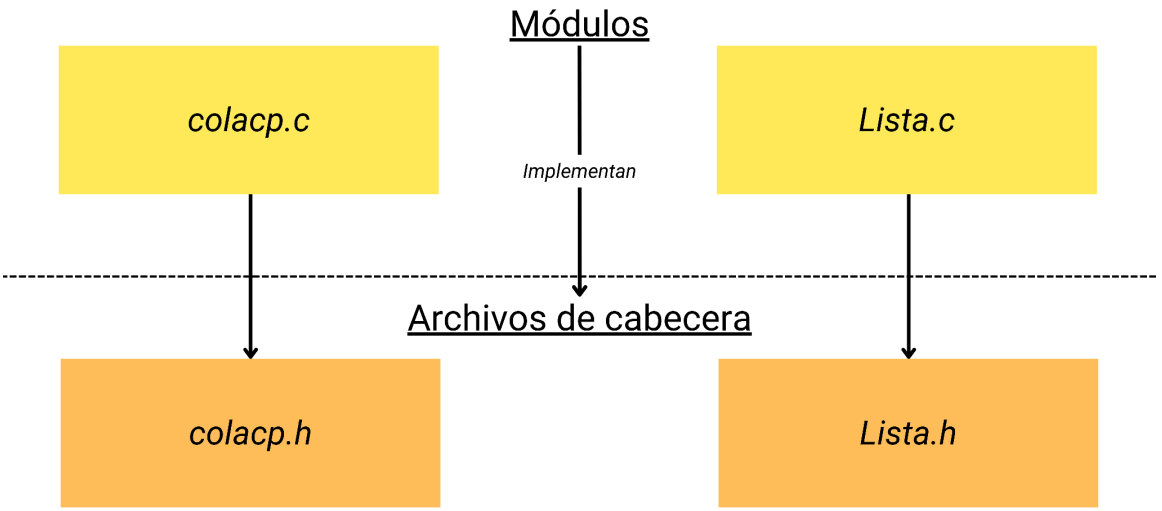
Descripción jerárquica:

La estructura del programa se divide en tres módulos, dos correspondientes a estructuras de datos y un módulo planificador donde se lleva a cabo el desarrollo de los problemas planteados en el enunciado.

Diagrama de módulos:



Planificador utiliza los servicios provistos por los módulos Cola con prioridad y Lista, los cuales se conforman de la siguiente manera:



Descripción de los módulos:

- **Cola con Prioridad:** es una estructura de datos que almacena entradas de tipo clave valor, donde la clave es el dato con el que la estructura ordena dichas entradas, y el valor es el dato a almacenar.
  La tarea principal es la de ordenar mediante una función de comparación las entradas en base a su prioridad, para luego poder removerlas en el orden especificado. Este orden se especifica en la creación de la estructura, y gracias a la función de comparación la estructura de datos cumple con la prioridad de orden total.
  Tiene la capacidad de:
    - ➔ Iniciar una cola con prioridad con una función de comparación dada.
    - ➔ Insertar una entrada en la colección.
    - ➔ Remover la entrada con mayor prioridad.
    - ➔ Conocer la cantidad de entradas insertadas.
    - ➔ Destruir toda la colección.

  Para que la estructura funcione correctamente, se debe asegurar que al momento de realizar una solicitud la cola utilizada haya sido inicializada.

  Toda la implementación se encuentra en el archivo colacp.c el cual implementa al archivo colacp.h que contiene las estructuras utilizadas y los métodos descritos.

- **Lista simple:** es una estructura de datos en la que cada elemento apunta al siguiente. De este modo, teniendo la referencia del principio de la lista podemos acceder a todos los elementos de la misma. Esta no es de nuestra autoría, sino que fue implementada por la cátedra correspondiente a la materia.
  El propósito de la utilización de esta estructura, es poder almacenar toda la información necesaria provista por el cliente en memoria dinámica. De esta forma solamente se accede al disco una única vez, optimizando la utilización de memoria y acelerando los tiempos de carga.

  Toda la implementación se encuentra en el archivo Lista.c el cual implementa al archivo Lista.h que contiene las estructuras utilizadas y los métodos descritos.

Dependencias externas:

El software utiliza las siguientes librerías:

- **stdio.h**
  - printf: muestra por pantalla una cadena con formato.
  - fscanf y scanf: introduce cadenas de datos.
  - fopen, fclose: abre y cierra un fichero.
  - feof: comprueba el indicador de final de fichero.
- **stdlib.h**
  - malloc y free: reservan y liberan memoria respectivamente.
  - exit: termina la ejecución del programa.
- **math.h:**
  - fabs: calcula y retorna el valor absoluto de un float.

Como se mencionó anteriormente, se hizo uso del lenguaje de programación C y las librerías necesarias. Las decisiones de diseño se basaron en lo solicitado por la cátedra, ya que uno de los requisitos era la utilización de C como lenguaje, en base a eso y a ciertas necesidades se incluyeron las librerías mencionadas.

3- Diseño del modelo de datos:

Datos internos:

- *entrada:* almacena una TClave y un TValor, ambos punteros genéricos.
- *nodo:* almacena una TEntrada, puntero a la estructura entrada, y a su vez almacena tres punteros a nodos correspondientes al nodo padre, hijo izquierdo e hijo derecho conformando así un árbol.
- *cola_con_prioridad:* almacena un entero con la cantidad de nodos almacenados, un TNodo raíz, y una función **f** la cual es utilizada para comparar y ordenar las entradas.
- *tCelda:* conforman las celdas de la Lista simple. Contiene un puntero a un elemento y un puntero a una tCelda, la cual corresponde a la celda siguiente.

Datos entrada/salida:

- *TClave:* punte generico.
- *TValor:* punte generico.
- *TEntrada:* puntero a la estructura **entrada**.
- *TNodo:* puntero a la estructura **nodo**.
- *TColaCP:* puntero a la estructura **cola_con_prioridad**.
- *tElem:* elemento correspondiente a una celda de la Lista.
- *tPosicion:* puntero que apunta a una celda de la Lista.
- *tLsita:* puntero a la estructura **tCelda**.

4- Descripción de procesos y servicios ofrecidos por el sistema:

Servicios ofrecidos:

En la sección número 1 correspondiente a la **definición y especificación de requerimientos**, se hace una breve introducción a lo que *Planificador* ofrece.
En esta sección se verá en detalle no solo el funcionamiento si no que también la implementación llevada a cabo.

Servicios ofrecidos:

1. *MOSTRAR ASCENDENTE:* se utilizó un *Cola con Prioridad* mediante una estructura **MINHEAP,** la cual ordena las entradas de forma ascendente, es decir, la entrada con mayor prioridad es aquella que su *TClave* sea la menor de todas.
2. *MOSTRAR DESCENDENTE:* se utilizó un *Cola con Prioridad* mediante una estructura **MAXHEAP,** la cual ordena las entradas de forma descendente, es decir, la entrada con mayor prioridad es aquella que su *TClave* sea la mayor de todas.
3. *REDUCIR HORAS DE MANEJO:* mediante una *Cola con Prioridad* ordenada de forma ascendente, se utiliza una cola del mismo tipo secundaria, en la cual se ingresan todos los datos necesarios para lograr el objetivo.
   Esto se logra mediante el siguiente algoritmo:

   Mostrar la primera entrada **y almacenar sus coordenadas X e Y.**
   X e Y ahora son nuestras coordenadas de referencia.
   i = 2 y control = cantidad de entradas más 1.
   totalDist = la clave de la primera entrada.

   Mientras que *i<control:*
   Mientras que la cola principal no este vacia:
   remuevo e inserto en la cola secundaria las entradas de la cola principal, utilizando X e Y para calcular la prioridad.

   Mostrar la primera entrada **de la cola secundaria y almacenar sus coordenadas X e Y.**
   X e Y ahora son nuestras coordenadas de referencia.
   totalDist se le suma la clave de la entrada mencionada anteriormente.

Mientras que la cola secundaria no este vacia:

remuevo e inserto en la cola principal las entradas de la cola secundaria.

Aumento en 1 a i.

Fin del bucle.

Cabe aclarar, que al momento de mostrar la entrada, se debe remover de la cola en la que esté almacenada.

4. *SALIR:* como se menciona en los requisitos funcionales, libera toda la memoria utilizada haciendo un uso controlado de las sentencias *malloc* y *free* respectivamente. Cada TDA tiene la capacidad de destruirse, se hace uso de esto para liberar la memoria requerida.

Descripción de Procesos:

A continuación se mencionan los procesos y tareas que el programa es capaz de realizar, así como también una breve descripción sobre qué hace cada uno.
Los procesos se especifican con el siguiente formato:

***tipo* nombre_funcion(argumentos):** descripción.

- Planificador:
  - ***void* ingresar_datos_lista(FILE * archivo, tLista * list):** el resultado es void porque simplemente leemos el archivo e ingresamos estos datos a una lista (las ciudades y sus coordenadas).
  - ***void* cargar_datos_colaCP(tLista list, TColaCP datos):** cada valor en las celdas de la lista es ingresado a la cola con prioridad de manera tal que cada entrada clave-valor está compuesta por el cálculo de la distancia entre las coordenadas como clave y la ciudad (que es el nombre de la ciudad junto sus coordenadas) como valor.
  - ***float* calcular_distancia(float x, float y, float cord_x, float cord_y):** es el cálculo de la distancia entre coordenadas, anteriormente mencionado $|x - cord\_x| + |y - cord\_y|$.
  - ***int* ascendente(TEntrada x, TEntrada y):** compara las *claves* de dos entradas (x , y), si x < y retorna -1, si son iguales retorna 0 y si x > y retorna 1.
  - ***int* descendente(TEntrada x, TEntrada y):** compara las *claves* de dos entradas (x , y), si x > y retorna -1, si son iguales retorna 0 y si x < y retorna 1.

- ***void* mostrar_Ciudades(TColaCP datos):** imprime el nombre de la ciudad en el *valor* de las entradas que tiene la cola con prioridad y al mismo tiempo elimina las dichas entradas.
- ***void* fEliminar(TEntrada ent):** Elimina las entradas de una cola con prioridad en su totalidad(memoria).
- ***void* camino_mas_corto(TColaCP datos):** muestra por pantalla las ciudades ordenadas de forma que se puedan recorrer todas transitando el camino más corto posible. Finalmente muestra la distancia total a recorrer.
- ***TEntrada* crear_entrada(TClave c, TValor v, float cord_x, float cord_y):** Crea y retorna una *TEntrada,* asignado los valores *c* y *v* respectivamente, calculando la prioridad en base a las coordenadas recibidas y reservando memoria suficiente para la persistencia de los datos.

➢ TDA Cola:

- ***TColaCP* crear_cola_cp(int (*f)(TEntrada, TEntrada)):** Crea y retorna una cola con prioridad vacía. El orden en que las entradas son retiradas de la cola estará dado por la función de prioridad ***f.***
- ***int* cp_insertar(TColaCP cola, TEntrada entr):** Agrega la entrada en la cola, reservando memoria suficiente. Retorna verdadero si procede con éxito, falso en caso contrario.
- ***TEntrada* cp_eliminar(TColaCP cola):** Elimina y retorna la entrada con mayor prioridad (esto es, menor clave) de la cola. Acomoda la estructura heap de forma consistente. Al finalizar libera toda la memoria utilizada para almacenar el nodo eliminado. Si la cola está vacía, retorna *ELE_NULO.*
- ***int* cp_cantidad(TColaCP cola):** Retorna la cantidad de entradas de la cola.
- ***void* cp_destruir(TColaCP cola, void (*fEliminar)(TEntrada)):** Elimina todas las entradas y libera toda la memoria utilizada por la cola cola. Para la eliminación de las entradas, utiliza la funci´on *fEliminar*.
- ***TNodo* getPadre(TColaCP cola, int pos):** Retorna el padre del nodo correspondiente a la posición *pos* de la cola.
- ***void* crearTNodo(TNodo nodo, TNodo padre, TEntrada entr):** Crea un TNodo nodo y le asigna su correspondiente entrada y su nodo padre reservando memoria suficiente.
- ***void* burbujeoInsertar(TColaCP cola, TNodo hijo, TNodo padre):** Recorre hacia arriba recursivamente la estructura *HEAP* perteneciente a la cola, intercambiando en caso de ser necesario el nodo hijo con el padre.

- ***void* burbujeoEliminar(TColaCP cola, TNodo raiz):** Recorre hacia abajo recursivamente la estructura *HEAP* perteneciente a la cola, comparando la raíz de con los hijos correspondientes.
- ***TNodo* nodoMayorPrioridad(TColaCP cola, TNodo n1, TNodo n2):** Retorna el nodo con mayor prioridad pertenecientes a la cola.
- ***void* removerTNodo(TColaCP cola, TNodo aux):** Remueve el hijo correspondiente al nodo aux. Si aux tiene *hijo derecho* lo elimina, si no tiene *hijo derecho* pero sí *hijo izquierdo* elimina este último. En ambos casos libera la memoria reservada anteriormente. Este método siempre se ejecuta en el caso de que aux tenga *hijos,* al finalizar decrementa la cantidad de elementos de la cola en 1.

➢ TDA Lista:

- ***void* crearListaVacia(tLista *L):** Crea y guarda espacio en memoria para una lista, con sus componentes internos en nulo (Elemento y Siguiente).
- ***void* insertarElemento(tElem * E, tPosicion P, tLista * L):** Inserta el Elemento E en la posición P de la Lista L.
- ***void* eliminarElemento(tPosicion P, tLista L):** Elimina el dato guardado en la posicion P.
- ***int* esListaVacia(tLista L):** Retorna 1 si no está vacía, 0 si está vacía.
- ***tPosicion* primera(tLista L):** Retorna la primera posición de lista.
- ***tPosicion* fin(tLista L):** Retorna la última posición de la lista.
- ***tPosicion* ultima(tLista L):** Retorna la última posición de la lista.
- ***tPosicion* anterior(tPosicion P, tLista L):** Busca y recupera la posición anterior a la posición *P* y la retorna.
- ***tPosicion* siguiente(tPosicion P, tLista L):** Retorna la posición siguiente a la posición *P* pasada por parámetro, perteneciente a la lista *L.*
- ***tElem** * **recuperar(tPosicion P, tLista L) :** Recupera y retorna un puntero al elemento perteneciente a la lista *L* en la posición *P*.
- ***void* vaciarLista(tLista L):** Elimina los elementos previamente almacenados, liberando la memoria reservada con anterioridad.

6- Aspectos relevantes:

Invocación del programa:

Para invocar *Planificador.exe* se hace uso de la consola con la siguiente invocación:
/.Planificador.exe “archivo.txt”

Planificador.exe es el nombre del ejecutable generado anteriormente en el proceso de instalación y *archivo.txt* es el nombre del archivo donde se encuentran las ciudades con sus respectivas coordenadas. El archivo puede estar o no escrito con comillas.
Una vez ejecutada el comando, verá en pantalla el programa en ejecución.

Inconvenientes:

Al momento de generar la documentación correspondiente al código fuente, no fue posible utilizar la herramienta sugerida (*Doxygen*). Esta misma nos trajo problemas de permisos sobre la lectura y escritura de los archivos, los cuales no pudimos solucionar.
De todas formas, en "**Descripción de Procesos**" se encuentran documentados todos los métodos utilizados.

Conclusiones:

El proceso llevado a cabo al momento de realizar este proyecto fue muy satisfactorio. Gracias a los profesores y ayudantes correspondientes a la cátedra logramos un buen desempeño incorporando nuevas herramientas y utilizando las que ya conocíamos. El tiempo pactado para la entrega fue bastante considerable, el cual aprovechamos a nuestro favor consultando y planificando todo de forma precisa.
Se destaca la predisposición de los ayudantes antes mencionados, ya que nos dieron una visión clara sobre el lenguaje y las herramientas utilizadas, algo hasta el momento desconocido para nosotros.