Universidad de San Carlos de Guatemala
Escuela de Ingeniería en Ciencias y Sistemas, Facultad de Ingeniería
Introducción a la programación y computación 2, 1er. Semestre 2021.

APLICACIÓN DE MEMORIA DINÁMICA Y MANIPULACIÓN DE UNA COLECCIÓN DE DATOS CON ARQUITECTURA EN MATRIZ ORTOGONAL

201709020 – Jaime Efraín Chiroy Chavez

Resumen

El proyecto consiste en el análisis de archivos de extensión xml, el cual posee los registros de imágenes y sus respectivos datos (tamaño de fila, columna y "mapa de bits"), los datos leídos se almacenarán en memoria con el uso de un TDA de tipo lista simple enlazada para lo que se nombrará como "biblioteca" en el cual será capaz de contener una N cantidad de imágenes obtenidas del archivo de entrada, el mapa de cada imagen se almacenará en la biblioteca como una sub lista ortogonal. En consideración al problema analizado, se determinó tres tipos de operaciones que se realizarán sobre las imágenes los cuales son: operaciones básicas, de edición y lógicos, los resultados de las operaciones se visualizarán en una interfaz gráfica, en ella, el usuario podrá realizar estas operaciones y visualizar los resultados además de generar los respectivos reportes de estado a través de un archivo html.

Palabras clave

TDA – Tipo de Dato Abstracto, Lista Ortogonal, Nodo, Apuntador, Operación.

Abstract

The project consists of the analysis of files with an xml extension, which has the image and their respective data (row size, column and "bitmap"), the read data will be stored in memory with the use of a TDA of type simple linked list for what will be named as "library" in which it will be able to store N images obtained from the input file, the map of each image will be stored in the library as an orthogonal sub list. In consideration of the analyzed problem, three types of operations performed on the image were determined: basic operations, editing and logical operations, the results of the operations will be displayed in a graphic interface, where the user can perform these operations and view the results in addition to generating the respective status reports through an html file.

Keywords

ADT – Abstract Data Type, Orthogonal list, Node, Pointer, Operation.

Introducción

Con la implementación de TDA's para el almacenamiento de imágenes de tamaño desconocido, el uso de memoria dinámica resulta ser muy conveniente, el uso de listas ortogonales pueden llegar a ser confusas como parte de la solución al planteamiento de operaciones sobre imágenes; para mantener las rotaciones, edición para cada "pixel" de la imagen y operaciones con dos matrices de igual tamaño (unión, intersección, etc.), la solución proveerá al usuario una interfaz gráfica que permitirá realizar operaciones con imágenes e ingresar rangos válidos según el tamaño de la misma, obteniendo así una aplicación altamente confiable; una vez realizada la operación las imágenes se visualizarán en los paneles de la interfaz gráfica, para las operaciones con dos imágenes el resultado se mostrará en una ventana emergente, se podrá generar un log del historial de ejecución en formato html.

Desarrollo del tema

Iniciando con la solución se plantea la estructura de datos para la lista ortogonal.

Los requerimientos del software son:
- -Implementación de POO y TDA
- -Lectura de archivos de entrada (Formato XML)
- -Creación de archivos de salida (Formato HTML)
- -Operaciones con TDA (Listas ortogonales)
- -Generar un grafo de una imagen seleccionada

MODULO – ClaseLista.py
Especificaciones de TDA

La solución para la estructura de datos se define con una lista simple, para sus nodos contendrán la información de una imagen, a continuación, se detallan las estructuras utilizadas.

Comenzando con la definición del TDA para la "imagen" se considera:

NodoMultidireccional
| Carácter
| Superior: NodoMultidireccional
| Inferior: NodoMultidireccional
| Anterior: NodoMultidireccional
| Siguiente: NodoMultidireccional
Fin NodoMultidireccional

La clase "**NodoMultidireccional**" contendrá todos los atributos que conforman a la "imagen", el carácter que define al pixel de la imagen, así como los punteros a un "siguiente" objeto de la misma clase, estos atributos tendrán los métodos *getters* y *setters*.

Imagen
| inicio: NodoMultidireccional
Fin Imagen

La clase "**Imagen**" se definirá con un solo atributo de tipo NodoMultidireccional, sus posibles operaciones son:

Inicializar: Colocar a null el apuntador inicio de la lista.

getInicio: Retorna el nodo inicio.

estaVacía: Retornará True si *inicio* es null o de lo contrario retornará False.

agregarNodo: Recibe como parámetro, carácter y un booleano nombrado como nuevaFila.

Si está vacía: Se crea un nodo (multidireccional) que se asigna al inicio.

No vacía: Se crea un nuevo nodo (multidireccional), con una variable temporal se creará un conector vertical capturando el apuntador inicio, seguido con un while se recorrerá mientras el nodo (inferior) no apunte a null, al encontrar el último nodo (vertical) de la lista finaliza el ciclo, se crea un

nodo temporal (multidireccional) con el carácter de entrada, seguido con un if se validará si éste corresponde a una nueva fila, sí este es True, para el conector vertical se asignará el nodo temporal en su puntero inferior y del nodo temporal se asignara el conector vertical en su puntero superior, en cambio, sí, el carácter no pertenece a una nueva fila se creará un conector horizontal que obtendrá el puntero superior del conector vertical, luego en un ciclo while se recorrerá hacia el puntero de la derecha (siguiente) del conector vertical hasta que sea igual a null, a demás si el conector horizontal no apunta a null recorrerá de igual manera su puntero hacia la derecha, una vez fuera del ciclo, el conector vertical enlazará al siguiente con el nodo temporal y el nodo temporal al anterior con el conector vertical, si el conector horizontal es distinto de null, el conector horizontal avanzara hacia un nodo siguiente y en este se enlazará al nodo temporal en su puntero inferior y el nodo temporal apuntara hacia el superior con el conector horizontal.

Se consideró también la definición del TDA para la "biblioteca" se considera:

NodoImagen
| Nombre
| Filas
| Columnas
| Img : Imagen
| Siguiente: Nodo
Fin NodoImagen

La clase "**NodoImagen**" contendrá todos los atributos que conforman a las imágenes, y su respectivo puntero al nodo siguiente, estos atributos únicamente tendrán los métodos *getters* y *setters*.

ListaImagenes
| inicio: NodoImagen
Fin ListaImagenes

La clase "**Lista Imagenes**" se definirá con un solo atributo de tipo NodoImagen, sus posibles operaciones son:

Inicializar: Colocar a null el apuntador inicio de la lista.

estaVacía: Retornará True si *inicio* es null o de lo contrario retornará False.

getInicio: Retorna el nodo inicio.

agregarImagen: Recibe como parámetros, nombre, filas, columnas, imagen (lista ortogonal).

Si está vacía: Se crea un nodo imagen que se asigna al inicio.

No vacía: Se crea una variable temporal que referencia al nodo inicio, con un while se recorren los punteros hasta el final de la lista, una vez finalizado el ciclo, se crea el nodo imagen temporal y se enlaza el nodo auxiliar con el nodo temporal como su siguiente nodo.

existeNombre: Recibe el parámetro nombre, con una variable temporal se cicla la lista hasta obtener un resultado de búsqueda si la lista tiene coincidencias retorna True o en su defecto retorna False.

MODULO – LectorXML.py

Librerias utilizadas:
- minidom de xml.dom: Permitirá la lectura de archivos xml
- time: Permitirá crear pausas para visualizar el proceso de lectura.
- *manejardor*: Módulo general de manejo, donde se aloja la lista global (biblioteca).
- *imagen* de *ClaseLista*: Permitirá crear un TAD temporal.

Función – **leerxml(ruta, nArchivo):**

Recibe como parámetro la ruta y nombre de archivo, en el cual con uso de *minidom.parse(ruta)* el archivo

es cargado al programa, y mediante *getElementsByTagName*, se separan los datos según las etiquetas indicadas según los requerimientos de estructura del archivo xml (ver anexo).

En este método se valida el nombre de la nueva matriz, con la función auxiliar *validaNombre(nombre)* que obtiene la biblioteca, ésta valida si la lista está vacía y si contiene nodos evalúa si el nombre ya pertenece a algún registro dentro de la lista, si hay coincidencias retorna True de lo contrario retorna False.

Obtiene los valores de sus tags tales como las dimensiones de la imagen y evalúa si los valores son numéricos, de lo contrario el sistema lanzará un mensaje de error y omitirá la lectura de "esa" matriz en la lista de matrices.

También se obtiene el mapa de imagen del cual se remueven los espacios en blanco, luego ésta se recorre ya que contiene los valores o caracteres de la imagen, se obtienen y se evalúan si corresponden a los valores aceptados y si la cantidad de caracteres obtenidos hasta el delimitador "\n", no exceden a tamaño especificado tanto a su tamaño de filas como de columnas, de lo contrario se indicarán los errores (*taBien = False*) y se omitirá el almacenamiento de dicha matriz.

Al finalizar el análisis de una imagen, si el booleano *taBien* es True, significa que la lectura ha sido completada sin errores; con los datos ya validados, se crea una lista ortogonal (imagen) temporal, para luego ciclar nuevamente la imagen y enviar cada carácter a la función ***agregarNodo(carácter, esNueva)*** para iniciar con la inserción de datos para la imagen, una vez completado se envían los datos para agregarlos a la lista global en el modulo manejador.

MODULO – generarGrafo.py

Librerias utilizadas:
- *system* de *os*: Permitirá usar funciones del sistema.
- *path* de *os*: Permitirá trabajar funciones con rutas de archivo.

Función – **getsource():**

A través de la librería *path* este método retornará la ruta del script que se está ejecutando.

Función – **mimetodo(di):**

Esta función recibirá un parámetro de tipo string, que contendrá un comando a ejecutar, a través de la librería *os* con la función system.

Función – **grafo(nombreM, filas, columnas, matriz, tipo):**

Esta función permitirá la escritura de un archivo de extensión *.dot*, en donde se manejará la estructura del archivo; se recorrerá la lista ortogonal (imagen) para obtener los caracteres de imagen.

La estructura utilizada para generar el grafo, será *struct* que permitirá generar un gráfico en forma de tabla; una vez generada el archivo *.dot* se llamará a la función *generagraf(nombre)* que enviará como parámetro la instrucción a ejecutarse en el sistema, generando así el grafo de la imagen.

MODULO – manejador.py

Librerias utilizadas:
- *path* de *os*: Permitirá trabajar funciones con rutas de archivo.
- *ListaImagenes, NodoImagen, imagen* de *ClaseLista*: Permitirá trabajar funciones con rutas de archivo.
- *grafo* de *generarGrafo*: Permitirá trabajar funciones con rutas de archivo.

- *leerxml* de *LectorXML*: Permitirá realizar la lectura del archivo de carga.

Variable global (única): **biblioteca**.

Función – **getLista():**
Retornará la lista global.

Función – **agregaImagen(nombre, filas, columnas, imagen):**
Utiliza la función de la lista global para almacenar nuevos registros de matrices.

Función – **cargarArchivo():**
Al llamar a esta función se validará si la biblioteca posee registros para ser limpiada y así almacenar los nuevos registros, se evaluará la ruta del archivo (si es de extensión xml) para luego llamar a la función *leerxml*.

Función – **generaImagenO():**
El usuario deberá seleccionar una imagen desde la lista de interfaz para generar su respectivo grafo, una vez el usuario haya seleccionado una imagen, se llamará a la función *grafo*, el cual generará un grafo en la carpeta de los scripts en ejecución y lo abrirá automáticamente.

Las funciones y estructuras anteriores cumplen con el requerimiento mínimo de funcionamiento, en este apartado se detallarán la soluciones y complementos realizados en los módulos de *ClaseLista* y *manejador*.

Como se indicó al principio, las operaciones se dividieron en tres tipos, de *rotación*, *edición*, *lógicos*.

Operaciones De Rotación (Horizontal y Vertical)

Estas operaciones definen un cambio en el orden de la estructura de la imagen, con el debido análisis se determinó que para una rotación horizontal una lectura de la matriz de filas de abajo hacia arriba permite describir correctamente el orden de la rotación, de igual manera para la rotación vertical, se define como el recorrido de la matriz por columnas de derecha a izquierda.

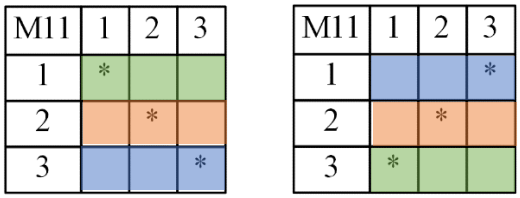


Fig 1. Rotación Horizontal (Original | Modificada)

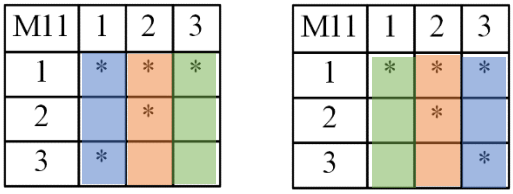
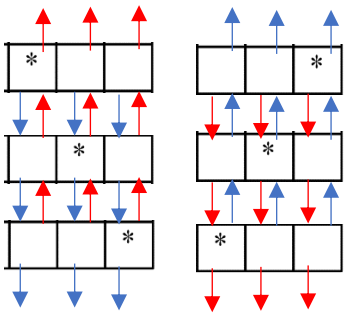


Fig 2. Rotación Vertical (Original | Modificada)

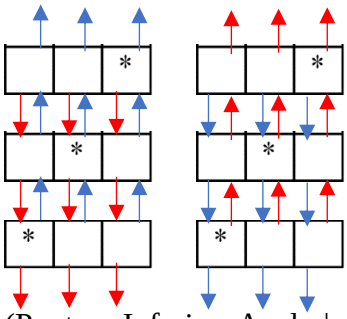
Para obtener una funcionalidad "optima" se determinó que el solo recorrer no provee la simplicidad de programación, y con ello se determina los conceptos "reapuntar", "redefinir punteros", "modificar punteros", dando como resultado la siguiente lógica.

-Que la función estática para recorrer la lista ortogonal va de izquierda a derecha de arriba hacia abajo
-Para una rotación horizontal el nodo inicio (del recorrido) debe ser el último nodo inferior de la lista hacia el nodo superior de izquierda a derecha.



(Puntero Inferior: Azul | Puntero Superior: Rojo)

-Si los apuntadores verticales (superior e inferior) de cada nodo se redefinen es decir superior ahora toma el puntero del inferior y el inferior recibe el puntero superior se obtiene un recorrido izq-der, arriba-abajo:



(Puntero Inferior: Azul | Puntero Superior: Rojo)

Dada esta lógica de "redefinición de punteros" se procede a agregar funcionalidades en el módulo *ClaseLista* en la clase *NodoMultidireccional* y la clase *imagen*.

NodoMultidireccional - ClaseLista
Función Agregada – **reflejaHorizontal():**
Esta función intercambia las direcciones de los punteros inferior y superior.

Función Agregada – **reflejaVertical ():**
Esta función intercambia las direcciones de los punteros anterior y siguiente.

Función Agregada – **rotaIzq ():**
Esta función intercambia las direcciones de los cuatro punteros una vez en sentido de las agujas del reloj, es decir que superior apuntará a siguiente, siguiente a inferior, inferior a anterior y anterior a superior, o dicho de otra forma las direcciones actuales son asignadas a los punteros que están en contra de las agujas del reloj.

imagen- ClaseLista
Función Agregada – **rotaImagen(tipo):**
Esta función evalúa tres tipos de rotación "h" – horizontal, "v" – vertical, "t" – transpuesta, según el tipo de rotación la lista ortogonal se recorrerá y para cada nodo en la imagen se ejecutará según la instrucción recibida, es decir que para una rotación "h" cada nodo de la lista utilizará *reflejaHorizontal()*, obteniendo así la "rotación" de la imagen según el tipo, también en esta función se redefine el puntero inicio, para dar lugar así a un nodo inicio según el tipo de rotación.
Cabe mencionar que para lograr la transposición la imagen primero debe ser tener una rotación "v" y luego la rotación "t":

ListaImagenes - ClaseLista
Función Agregada – **setImgNodeByName(nombre, posicion):**
Esta función realiza la búsqueda de la imagen a operar según la posición del cambio pedido, en esta función se considera que si es una transposición primero se ejecute una "v" y luego una "t".

Nota: Se obviarán ciertos análisis (como la transposición) para no extender demasiado el documento.

También se consideró que para una transposición de una matriz *no cuadrada* los valores de columna y fila deben ser modificados y para ello:

NodoImagen - ClaseLista
Función Agregada – **redimensionaImg():**
Esta función intercambia el valor de filas y columnas (dimensiones de imagen)

También se crea una función en el manejador éste para graficar la nueva imagen modificada.

Manejador
Función Agregada – **generaImagenMod(selección, tipo):**
En la interfaz el usuario selecciona el tipo de operación y la matriz sobre la cual se realizará la operación, obteniendo así el nodo (*getNodeByName(selección)*), operando y enviándolo a generar su grafo.

Operaciones De Edición (Limpiar área, línea h., línea v., rectángulo, triangulo)

Estas operaciones realizan cambios sobre una imagen modificando valores de los nodos según los rangos dados por el usuario.

Se procede a agregar funcionalidades en el módulo *ClaseLista* en la clase *imagen*, esto con el fin de modificarlo directamente dentro del objeto y tener un manejo directo mediante una llamada a sus funciones.

imagen

Función Agregada – **limpiaArea(f_ini, f_fin, c_ini, c_fin):**
Esta función tendrá dos contadores de fila y columna, los cuales serán indicadores al llegar a un "tope" (valores de parámetro), se crea un nodo auxiliar y con dos while se ubica en su respectiva posición el primero para ubicarse en la fila de inicio, y el segundo para ubicarse en la columna de inicio, luego habiendo ubicado en el lugar en un nuevo ciclo *while true* se verifica si el contador de fila y columna son iguales a los parámetros finales (fila y columna) si es así significa que únicamente hay un nodo a "limpiar" y luego con un break rompe el ciclo, de lo contrario con un while interno se ciclará para recorrer los nodos hacia la "derecha" (nodo siguiente) hasta que el contador de columnas llegue al **c_fin**, luego si el contador de filas es menor que el **f_fin**, se realiza el algoritmo de regresar tanto el contador como los nodos hasta que el contador de columnas sea igual al **c_ini**, para luego obtener el nodo inferior (fila siguiente) y aumentar el contador de filas, esto hasta llegar hasta **f_fin**, para luego romper el ciclo while true**.**

Función Agregada – **lineaHoriz(fil, c_ini, c_fin):**
Cuenta con los contadores de fila y columna, y el algoritmo de ubicación para el nodo auxiliar según los parámetros **fil** y **c_ini**, en este algoritmo únicamente se recorre hacia la derecha (nodo siguiente) mientras el contador de columnas sea menor al tope (**c_fin**), modificando los nodos que el contador permita, y por ultimo se modifica el ultimo nodo ya que si el contador de columnas es igual al **c_fin**, este nunca entrará al while, pero si debe ser moficiador ya que ese nodo pertenece al rango seleccionado.

Función Agregada – **lineaVerti (f_ini, f_fin, col):**
Es similar a la función de *lineaHoriz()* pero cambia a que el "recorrido" lo hace por filas, pues como se entiende, es un recorrido vertical.

Función Agregada – **rectángulo (f_ini, altr, c_ini, anch):**
Esta función posee contadores (fila y columna), y realiza el algoritmo para ubicar el nodo auxiliar en la fila y columna inicial, luego el algoritmo realizará cuatro ciclos, el primero que recorrerá hacia la derecha (nodo siguiente) hasta el ancho limite, luego un recorrido hacia abajo hasta el limite de altura, luego con un recorrido hacia atrás hasta el limite de la fila inicio, y luego hacia arriba hasta el limite de la columna inicio, con ello logrando el algoritmo de crear un rectángulo según los rangos.

Función Agregada – **rectángulo (fil, col, tam):**
Esta función posee contadores (fila y columna), y realiza el algoritmo para ubicar el nodo auxiliar en la fila y columna inicial, luego el algoritmo realizará un recorrido vertical hacia abajo (nodo inferior) según el tamaño limite vertical, y luego hacia la derecha (nodo siguiente) hasta el tamaño limite horizontal, una vez realizado este recorrido en "L", si el tamaño del triangulo es mayor a 2 se necesitara "graficar" la línea inclinada del triángulo, y con esto se necesita un nuevo algoritmo llamado recorrido en escalera, el cual consiste en un recorrido primero a la izquierda y luego hacia arriba, para modificar el carácter y aumentar un contador de repeticiones que esta función debe realizar hasta un limite obtenido por la relación **tam-2**, ya que el numero de veces que se debe editar una "escalera" para **tam,** excede dos unidades a esta.

Conclusiones

Las listas ortogonales son una forma muy útil, para manejar datos en una estructura en matriz, un aspecto siempre deficiente es la complejidad en el manejo, ya que sin lugar a dudas es necesario utilizar bloques de ciclados que mal controlados que pueden incurrir en ciclados infinitos, una vez bien implementados son estructuras muy robustas y con la lógica correcta se obtiene un programa optimizado.

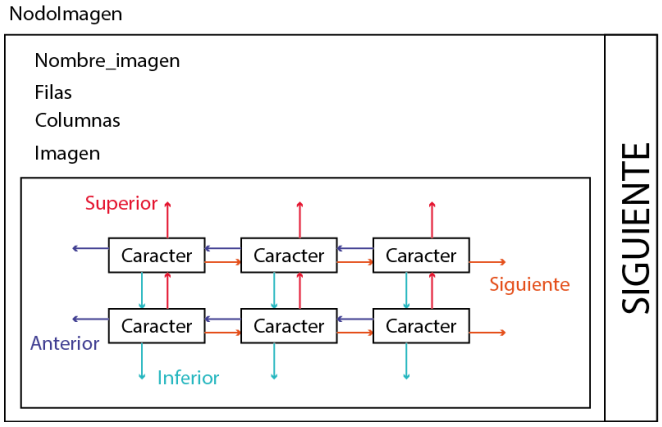
Referencias bibliográficas

Documentation. (s. f.). graphviz.org. Recuperado 27 de marzo de 2021, de https://graphviz.org/documentation/

Robinson, S. (s. f.). Reading and Writing XML Files in Python. Stack Abuse. Recuperado 27 de marzo de 2021, de https://stackabuse.com/reading-and-writing-xml-files-in-python/

Anexos

Estructura del TDA biblioteca:



Nota: Los nodos que no tienen enlace apuntan hacia Null.