ANALISIS Y DISEÑO DE ALGORITMOS

INFORME TRABAJO PRATICO ESPECIAL

PRIMERA PARTE CON CORRECCIONES





INTEGRANTES: BOCCANFUSO LUCAS EZEQUIEL, BERTINO ARIEL EUGENIO

N° DE GRUPO: 36

AYUDANTE ASIGNADO: MENCHON MARTIN

E-MAIL: lucas.mdq.935@gmail.com, eugenioingenio@hotmail.com

**BORRAR COMENTARIOS QUE ESTEN DE MAS TANTO EN EL CODIGO COMO EN LOS INFORMES**

INTRODUCCION

En esta primera parte del trabajo practico especial de análisis y diseño de algoritmos nos enfocaremos en especificar (nereus) e implementar(c++, clases) los Tipos de Datos Abstractos (TDA); lista, conjunto y árbol binario.

Para llevar esto acabo y antes de meternos de lleno en los TDA y clases, haremos algunas consideraciones generales para todas las clases.

Discutimos la posibilidad de hacer una implementación con arreglos dinámicos, pero consideramos las eventuales desventajas en comparación con listas enlazadas. En cuanto a la complejidad temporal marcamos que requieren de un tiempo lineal para insertar o eliminar en una ubicación arbitraria, ya que todos los elementos siguientes deben ser movidos, mientras que las listas enlazadas se pueden hacer esto en tiempo constante. Centrándose en el manejo de memoria puede ser costoso o imposible encontrar espacio contiguo para ampliar un arreglo dinámico grande, mientras que las listas enlazadas no requieren que la estructura de datos completa se almacene de forma contigua. Como última desventaja a nuestras consideraciones vamos a remarcar que un arreglo dinámico en cada oportunidad en la que quisiéramos agregar pediría que se le duplique la memoria reservada para sí mismo, y solo se liberaría la memoria no utilizada cuando sea posible dividir en 2.

Para concluir nuestra introducción vamos a decir que decidimos implementar una estructura de nodos vinculados con punteros, ya que estos últimos nos permitirán administrar la memoria de forma más eficiente, dándonos la posibilidad de liberar memoria que ya no usemos y solo reservando una parte de ella cuando sea necesaria, además de poder acceder a distintas regiones no contiguas del heap. Por sobre un arreglo dinámico que desperdicia memoria.

CLASE LISTA

Especificación sintáctica (NEREUS) del TDA lista paramétrizado. Es un contenedor genérico.

CLASS Lista [Elemento]

IMPORTS Boolean, Natural

BASIC CONSTRUCTORS CrearL, AgregarL

EFFECTIVE

TYPE

Lista

OPERATIOS

crearLista: -> Lista;

agregar: Lista \* Elemento -> Lista;

longitudLista: Lista -> Natural;

perteneceLista: Lista \* Elemento ->Boolean;

esVacia: Lista ->Boolean;

eliminarLista: Lista(l) \* Elemento(e) -> Lista

pre: pertenceL(l,e);

RecuperarElemento: Lista -> Elemento;

AXIOMS …

.

.

.

END\_CLASS;

La estructura que utilizamos para implementar el TDA Lista la llamamos nodo y consta de un campo llamado dato de tipo Elemento y un Puntero a otro nodo del mismo tipo llamado sig.

Para detallar la complejidad de las funciones de la clase Lista, listamos los encabezados de las mismas seguidas de una breve descripción de lo que realiza y su complejidad temporal

Aclaración: Las funciones privadas a la clase lista no serán analizadas por separado sino que será contemplado su comportamiento en la/s función/es publica/s que la utilice

En la siguiente sección pasamos a analizar la complejidad junto a una breve explicación de lo que hace cada método

Funciones Implementadas

lista(): constructor de la clase, inicializo los punteros en NULL y el contador de elementos en 0.

Complejidad: O(1), solo se hacen operaciones de orden constante.

~lista(): destructor de la clase, siempre que haya un elemento en primero llamo a la función privada eliminarNodo con primero, elimino lo que almacena y devuelvo el puntero siguiente como primero.

Complejidad: O(n) con n siendo cantidad de elementos, hay que recorrer toda la lista para eliminar los elementos 1 por 1.

cantElemL(): devuelve el valor almacenado en la variable local contador, la cual aumenta su valor en las ocasiones donde se crea un nodo y disminuye cuando se elimina uno.

Complejidad: O(1) solo se hace una asignación, la cual es de orden constante

perteneceL(elemento e): recorro la lista empezando en primero y terminando cuando el elemento almacenado en un nodo sea igual al elemento que se está buscando (devolviendo false), o en el peor de los casos se finaliza cuando se termine de recorrer la lista (devolviendo true).

Complejidad: O(n) con n siendo cantidad de elementos, en el peor de los casos hay que recorrer para comparar con todos los elementos de la lista.

esVaciaL(): pregunta por el puntero primero, si apunta a algo la lista tiene cuando menos un elemento, por lo que se devuelve true. Si primero no apunta a nada se devuelve false.

Complejidad: O(1), solo se hacen operaciones de orden constante

agregarPrincipio(elemento e): crea un nuevo nodo y lo vincula a la lista como el nuevo primero

Complejidad: O(1), solo se hacen operaciones de orden constante

agregarFinal(const elemento & e): crea un nuevo nodo vinculándolo a la lista mediante el puntero siguiente al último, luego asigna este como el nuevo último, contempla el caso de que uno quiera agregar a al final y justo se lista vacía quedando así los punteros ultimo y primero apuntando al nodo recién agregado.

Complejidad: O(1) , solo se hacen operaciones de orden constante.

agregarpos(const elemento & e, int p): primero comprueba que la posición en la que se quiere agregar no sea negativa ni mayor a la cantidad de elemento de la lista. Luego avanza la lista hasta la posición deseada crea un nodo y realiza las vinculaciones necesarias con el nuevo nodo agregado.

Complejidad: Siendo n la cantidad de elemento y si al agregar se da el caso de que la posición es el último elemento se deberá recorrer toda la lista siendo así la complejidad O(n)

eliminarL(elemento e): si el elemento a eliminar pertenece a la lista, lo busca en la misma hasta encontrarlo, una vez hecho esto lo desvincula de la lista hace la revinculación de la lista resultante y realiza el delete del nodo con el valor a eliminar. Al igual que el agregarpos si elemento a eliminar es el último se deberá recorrer toda la lista entonces siendo n la cantidad de nodos la complejidad es O(n).

elemento & iteracionL(): Define un aux de tipo nodo lo iguala a un cursor que apunta a un elemento de la lista, avanza el cursor y devuelve el dato del elemento contenido en aux. Son todas operaciones de orden constante siendo así su complejidad de O(n).

vaciarL(): REVISAR EL CODIGO HABER SI ES REALMENTE NECESARIO. SON LAS 1:00 DE LA MAÑANA……….

CLASE CONJUNTO

Especificación sintáctica (NEREUS) del TDA conjunto parametrizado, contenedor genérico

CLASS Conjunto [Elemento]

IMPORTS Boolean, Natural

BASIC CONSTRUCTORS CrearC, AgregarC

EFFECTIVE

TYPE

Conjunto

OPERATIONS

crearC: ->Conjunto;

cantElemC: Conjunto -> Natural;

perteneceC: Conjunto \* Elemento ->Boolean;

agegarC: Conjunto \* Elemento -> Conjunto

eliminarC: Conjunto(c) \* Elemento(e) -> Conjunto

pre: pertenceC(c, e);

unionC: Conjunto \* Conjunto -> Conjunto;

interseccionC: Conjunto \* Conjunto -> Conjunto;

diferenciaC: Conjunto \* Conjunto -> Conjunto;

iteracionC: Conjunto -> Elemento;

AXIOMS …

.

.

.

END\_CLASS

A la hora de implementar el TDA conjunto, decidimos usar variables de tipo lista, con todas las funciones que lista posee, para hacer más llevadero codificar las funciones de conjunto, agregando las restricciones propias que un conjunto posee.

Métodos que posee la clase conjunto

conjunto(): para inicializar un conjunto se usa el constructor de lista, siendo su complejidad O(1) por realizar operaciones de orden constante.

~conjunto(): Para destruir un conjunto se utiliza equivalentemente al constructor de conjunto, el destructor de lista, que a lo sumo recorre todos los nodos (en este caso elementos del conjunto) para eliminarlo por completo. Siendo n la cantidad de elementos y quedando así la complejidad temporal O(n).

cantElemC(): usa el método cantElemL() de la lista al tener variables locales a ese método solamente realiza asignaciones de orden constante por lo tanto su complejidad es de O(1).

perteneceC(elemento e): Al utilizar el método provisto por la clase lista (perteneceL(elemento e)), busca el elemento hasta en su defecto terminar de recorrer el conjunto por lo tanto siendo n la cantidad de elemento la complejidad queda en O(n).

agregarC(const elemento & e): pregunta con el método perteceL(elemento e) si el elemento a agregar ya está en el conjunto, si está en el conjunto no hace nada pues en un conjunto no hay elementos repetidos, de lo contrario utiliza el método agregarFinal(elemento e) de lista siendo sus operaciones de orden constante su complejidad es O(1).

eliminarC(elemento e): invoca al método eliminarL(elemento e) de lista, que en su defecto recorrer toda la lista en busca del elemento a aliminar por lo tanto su complejidad es O(n) siendo n la cantidad de elementos.

unionC(conjunto<elemento> & c, conjunto<elemento> & c\_resultante): Lo que hace este método es teniendo un conjunto la doy otro conjunto que es el que le quiero unir y un tercer conjunto que es donde se va a guardar el resultado en sí de la operación unión. Entonces lo que hace este método es recorrer el conjunto implícito agregando todos sus elementos a el conjunto resultante luego recorre el segundo conjunto preguntando si el elemento actual pertenece al conjunto resultante si pertenece no lo agrega de lo contrario lo agrega. Teóricamente se llama n a la cantidad de elementos del primer conjunto y m a la cantidad del segundo conjunto entonces se diría que la complejidad es O(m.n) pero al ser muy grande la cantidad de elementos de ambos conjuntos decimos que m tiende a n siendo n muy grande pudiendo denotar así la complejidad como O(n2).

interseccionC(conjunto<elemento> & c, conjunto<elemento> & c\_resultante): Este método funciona de la siguiente manera, si ambos conjuntos tienen elementos pregunta si el elemento del segundo conjunto pertenece al conjunto inicial si esto acurre ese elemento es enviado al conjunto resultante de lo contrario continua hasta que alguno de los conjuntos se queda sin elementos. Este método pregunta si cada elemento del segundo conjunto pertenecen al conjunto inicial por lo tanto busca m veces (siendo m la cantidad de elemento de segundo conjunto) 1 elemento en un conjunto inicial de n elemento. Teóricamente la complejidad ronda el O(n.m) pero al m y el n tender a valor muy grandes son casi igual o se acercar bastante por lo tanto se expresa la complejidad como O(n2).

diferenciaC (conjunto<elemento> & c, conjunto<elemento> & c\_resultante): REVISAR EL CODIGO PORQUE NO HACE LO PEDIDO, NO LO HICE PORQUE ESTOY HACIENDO EL HOLYSHITFUCKING INFORME PERDON Y GRACIAS. SON LAS 3:40 DE LA MAÑANA.

elemento & iteracionC(): Para iterar sobre la el conjunto se utiliza el iterador de lista todas las operaciones que involucra son de orden constante por lo tanto su complejidad es de O(1).

CLASE ARBOL BINARIO

Especificación sintáctica (NEREUS) del TDA árbol binario parametrizado, contenedor genérico.

CLASS ArbolB [Elemento]

IMPORTS Boolean, Natural, Lista [Elemento]

BASIC CONSTRUCTORS CrearA, AgregarA

EFFECTIVE

TYPE

ArbolB

OPERATIOS

crearA: ->ArbolB;

agregarA: ArbolB \* ArbolB \* Elemento ->ArbolB;

perteneceA: ArbolB \* Elemento ->Boolean;

cantElemA: ArbolB -> Natural;

esVacioA: ArbolB ->Boolean;

profundidadA: ArbolB -> Natural;

fronteraA: ArbolB -> Lista;

mostrarInorder: ArbolB -> Lista;

AXIOMS …

.

.

.

END\_CLASS;

Para la implementación del TDA árbol binario el pensamiento convergió en utilizar una estructura nombrada nodoAB que tiene los siguientes campos: un campo con el dato y dos campos que se llaman izq y der que son punteros a estructuras de tipo nodoAB. También a la hora de implementar el método frontera decidimos usar un variable de tipo lista y allí ir guardando la frontera del árbol.

Los métodos para árbol se detallan a continuación

arbolB(): realiza una asignación de un cursor que es un puntero a una estructura tipo nodaAB que tiene campo elemento y dos punteros más izq y der que apuntan a otros dos nodos del mismo tipo(nodoAB), es una asignación por lo tanto su complejidad queda en O(1);

~arbolB(): REVISARRRRRRRRRRRRRR Este método(el destructor) recorre todos los nodos del árbol borrándolos uno por uno. Para el apartado de complejidad hay revisar varios casos, para empezar el caso de que sea un árbol completo balanceado donde la complejidad rondaría el O(logn), también hay que ver el caso de que se a un árbol degenerado donde el mismo tiene aspecto de lista quedando así complejidad de orden O(n) en ambos casos siendo n la cantidad de nodos o elemento del árbol REVISARRRR.

agregarAbalanceado(const elemento & e); REVISAR NO ESTA IMPLEMENTADO QUE COMPLEJIDAD LE PONGO ADEMAS NO ERA NECESARIO AGREGAR ASI…… (;….. POR FAVOR BORRAR COMENTARIOS QUE ESTEN DEMAS

perteneceA(elemento e) const: REVISAR … Este procedimiento lo que debería hacer es buscar en el árbol mientras que este no sea vacío o el valor no sea el buscado, a lo sumo recorrerá todo el árbol por lo tanto su complejidad es O(n) ya sea un árbol degenerado(formato de lista) u ordenado, si fuera un árbol completo balanceado la complejidad seria O(log n), siendo siempre n la cantidad de elementos.

cantElemA() const: Cuenta nodo a nodo la cantidad de elementos del árbol por lo tanto sea degenerado o completo balanceado el árbol siempre la complejidad caerá en O(n).

cantElemSubArbol (arbolB<elemento> subArbol) const; REVISAR ANDA A SABER QUE ERA O QUE HACIA ESTO NUNCA SE IMPLEMENTO NUNCA FUE REALMENTE NECESARIO REVISAR.

esVacioA() const: simplemente pregunta si en árbol es vacío o no por lo tanto son operaciones de orden constante su complejidad es de O(1).

profundidadA() const; REVISAR usualmente debería recorrer todo el árbol comparando profundidades de ramas izquierdas con derechas dejando su complejidad en O(n) pero al no estar implementado no se puede saber ni a priori ni a nunca REVISAR.

lista<elemento> crearListaFrontera(): REVISAR para crear la lista con los elementos de la frontera es obligatorio recorrer todos los nodos de un árbol degenerado, completo, balanceado etc, preguntando si cada nodo es hoja de ser asi agrega ese elemento a una listaFrontera, esta complejidad será siempre de O(n) REVISAR

lista<elemento> crearListaArbol(): REVISAR este método almacena todos los elementos del árbol en una lista de manera tal que si uno quiere mostrar los elementos del árbol accede a esa lista de manera más sencilla con el método iterarL de lista, al pasar por todos los nodos se genera así pues una complejidad de O(n) con n siendo la cantidad de nodos REVISAR

DIFERENTES REPRESENTACIONES

En el tda lista pudimos haber elegido usar arreglos dinámicos aunque no se nos ocurrió como podría mejorar tonto el manejo como por sobretodo la complejidad temporal de los métodos/ algoritmos.

En lo que respecta a árbol binario de búsqueda es que si hubiéramos elegido representar al mismo con un arreglo siendo n una posición arbitraria de ese arreglo y └n/2┘, 2n y 2n+1 nodos padres y nodos hijos respectivamente quizá hubiéramos reducido alguna que otra complejidad temporal pero como ya lo habíamos pensado con árboles y un también tenemos bastante vista recorridos de árboles quisimos representarlo así justamente con árboles.

CONCLUSION

Esta primera parte fue un desafío, tuvimos muchas dudas durante y después de realizar el trabajo. También tuvimos demasiados altibajos con código puntual muchísimas dudas pero en si estuvo bueno, no nos referimos a fácil simplemente bueno. Le dedicamos mucho tiempo quizá más de lo que habíamos pensado en un principio pero tenemos que decir sirvió y mucho. Si nos basamos en el aprendizaje esta experiencia definitivamente cumplió su cometido.