NOMNRE DEL ALUMNO: EDDI DAVID AGUILAR MARTINEZ

NOMBRE DEL PROFESOR: Emmanuel Eduardo Sánchez Pérez

MARERIA: ALGORITMOS Y ESTRUCTURAS DE DATOS

TIPO DE TRABAJO: ensayo

LICENCIATURA: INGENIERIA EN SISTEMAS COMPUTACIONALES

CUATRIMESTRE: 5

26/03/2023

-**NOMENCLATURA BÁSICA DE ÁRBOLES**

Un árbol es una colección de elementos llamados “nodos”, uno de los cuales es la “raíz”. Existe una relación de parentesco por la cual cada nodo tiene un y sólo un “padre”, salvo la raíz que no lo tiene. El nodo es el concepto análogo al de “posición” en la lista, es decir un objeto abstracto que representa una posición en el mismo, no directamente relacionado con el “elemento” o “etiqueta” del nodo.

Descendientes y antecesores. Si existe un camino que va del nodo a al b entonces decimos que a esa antecesora de b y b es descendiente de a. Por ejemplo m1.txt es descendiente de anuser y juegos es antecesor de g2. Estrictamente hablando, un nodo es antecesor y descendiente de sí mismo ya que existe camino de longitud 0. Para diferenciar este caso trivial, decimos que a esa descendiente (antecesor) propio de b si a es descendiente (antecesor) de b, pero a 6= b. En el ejemplo de la figura 3.3 a es antecesor propio de c, f y d, Hojas.

Profundidad de un nodo. Nivel. La “profundidad” de un nodo es la longitud de único camino que va desde el nodo a la raíz. La profundidad del nodo g en el ejemplo es 2. Un “nivel” en el árbol es el conjunto de todos los nodos que están a una misma profundidad. El nivel deprofundidad 2 en el ejemplo consta de los nodos e, f y g

NODOS HERMANOS Se dice que los nodos que tienen un mismo padre son “hermanos” entre sí. Notar que no basta con que dos nodos estén en el mismo nivel para que sean hermanos. Los nodos f y g en el árbol de la figura 3.3 están en el mismo nivel, pero no son hermanos entre sí.

ORDEN DE LOS NODOS En este capítulo, estudiamos árboles para los cuales el orden entre los hermanos es relevante. Es decir, los árboles de la figura 3.4 son diferentes ya que, si bien a tiene los mismos hijos, están en diferente orden. Volviendo a la figura 3.3 decimos que el nodo c está a la derecha de b, o también que c es es el hermano derecho de b. También decimos que b es el “hijo más a la izquierda” de a. El orden entre los hermanos se propaga a los hijos, de manera que h está a la derecha de e ya que ambos son descendientes de c y b, respectivamente. A estos árboles se les llama “árboles ordenados orientados” (AOO).

PARTICIONAMIENTO DEL CONJUNTO DE NODOS, Ahora bien, dados dos nodos cualquiera m y n consideremos sus caminos a la raíz. Si m es descendiente de n entonces el camino de n está incluido en el de m o viceversa. Por ejemplo, el camino de c, que es a, c, está incluido en el de h, a, c, g, h, ya que c es antecesor de h. Si entre m y n no hay relación de descendiente o antecesor, entonces los caminos se deben bifurcar necesariamente en un cierto nivel. El orden entre m y n es el orden entre los antecesores a ese nivel. Esto demuestra que, dados dos nodos cualquiera m y n sólo una de las siguientes afirmaciones puede ser cierta

LISTADO DE LOS NODOS DE UN ÁRBOL

ORDEN PREVIO

Una forma más visual de obtener el listado en orden previo es como se muestra en la figura 3.8. Recorremos el borde del árbol en el sentido contrario a las agujas del reloj, partiendo de un punto imaginario a la izquierda del nodo raíz y terminando en otro a la derecha del mismo, como muestra la línea de puntos. Dado un nodo como el b el camino pasa cerca de él en varios puntos (3 en el caso de b, marcados con pequeños números en el camino). El orden previo consiste en listar los nodos una sola vez, la primera vez que el camino pasa cerca del árbol. Así en el caso del nodo b, este se lista al pasar por 1. Queda como ejercicio para el lector verificar el orden resultante coincide con el dado en (3.4).

ORDEN POSTERIOR

Recorriendo el borde del árbol igual que antes (esto es en sentido contrario a las agujas del reloj), listando el nodo la última vez que el recorrido pasa por al lado del mismo. Por ejemplo, el nodo b sería listado al pasar por el punto 3. • Recorriendo el borde en el sentido opuesto (es decir en el mismo sentido que las agujas del reloj), y listando los nodos la primera vez que el camino pasa cera de ellos. Una vez que la lista es obtenida, invertimos la lista. En el caso de la figura el recorrido en sentido contrario daría (a, d, c, g, h, b, f, e). Al invertirlo queda como en (3.6). Existe otro orden que se llama “simétrico”, pero este sólo tiene sentido en el caso de árboles binarios, así que no será explicado aquí.

OPERACIONES CON ÁRBOLES

ALGORITMOS PARA LISTAR NODOS Implementar un algoritmo para recorrer los nodos de un árbol es relativamente simple debido a su naturaleza intrínsecamente recursiva, expresada en (3.3). Un posible algoritmo puede observarse en el código

3.Si bien el algoritmo es genérico hemos usado ya algunos conceptos familiares de las STL, por ejemplo, las posiciones se representan con una clase iterator. Recordar que para árboles se puede llegar al “fin del contenedor”, es decir los nodos Λ, en más de un punto del contenedor. El código genera una lista de elementos L con los elementos de T en orden previo.

INSERCIÓN EN ÁRBOLES Para construir árboles necesitaremos rutinas de inserción supresiónde nodos. Como en las listas, las operaciones de inserción toman un elemento y una posición e insertan el elemento en esa posición en el árbol.

Cuando insertamos un nodo en una posición Λ entonces simplemente el elemento pasa a generar un nuevo nodo en donde estaba el nodo ficticio Λ. Por ejemplo, el resultado de insertar el elemento z en el nodo Λ3 de la figura 3.6 se puede observar en la figura 3.16 (izquierda). (Observación: En un abuso de notación estamos usando las mismas letras para denotar el contenido del nodo que el nodo en sí.) • Cuando insertamos un nodo en una posición dereferenciable, entonces simplemente el elemento pasa a generar un nuevo nodo hoja en el lugar en esa posición, tal como operaría la operación de inserción del TAD lista en la lista de hijos.

ALGORITMO PARA COPIAR ÁRBOLES

Con estas operaciones podemos escribir el seudocódigo para una función que copia un árbol (ver código 3.4). Esta función copia el subárbol del nodo nt en el árbol T en la posición nq en el árbol Q y devuelve la posición de la raíz del subárbol insertado en Q (actualiza el nodo nq ya que después de la inserción es inválido). La función es recursiva, como lo son la mayoría de las operaciones no triviales sobre árboles.

Con menores modificaciones la función puede copiar un árbol en forma espejada, es decir, de manera que todos los nodos hermanos queden en orden inverso entre sí. Para eso basta con no avanzar el iterator cq donde se copian los subárboles, es decir eliminar la línea 11 . Recordar que si en una lista se van insertando valores en una posición sin avanzarla, entonces los elementos quedan ordenados en forma inversa a como fueron ingresados. El algoritmo de copia espejo mirror_copy() puede observarse en el código 3.5. Con algunas modificaciones el algoritmo puede ser usado para obtener la copia de un árbol reordenando las hojas en un orden arbitrario, por ejemplo, dejándolas ordenadas entre sí.

IMPLEMENTACIÓN DE LA INTERFAZ BÁSICA POR PUNTEROS

Así como la implementación de listas por punteros mantiene los datos en celdas enlazadas por un campo next, es natural considerar una implementación de árboles en la cual los datos son almacenados en celdas que contienen, además del dato elem, un puntero right a la celda que corresponde al hermano derecho y otro left_child al hijo más izquierdo (ver figura 3.19). En la figura 3.20 vemos un árbol simple y su representación mediante celdas enlazadas.

EL TIPO ITERATOR

Por analogía con las listas podríamos definir el tipo iterator_t como un typedef a cell *. Esto bastaría para representar posiciones dereferenciables y posiciones no dereferenciables que provienen de haber aplicado right() al último hermano, como la posición Λ3 en la figura 3.21. Por supuesto habría que mantener el criterio de usar "posiciones adelantadas" con respecto al dato. Sin embargo no queda en claro como representar posiciones no dereferenciables como la Λ1 que provienen de aplicar lchild() a una hoja.

Una solución posible consiste en hacer que el tipo iterator_t contenga, además de un puntero a la celda que contiene el dato, punteros a celdas que de otra forma serían inaccesibles. Entonces el iterator consiste en tres punteros a celdas (ver figura 3.22) a saber, • Un puntero ptr a la celda que contiene el dato. (Este puntero es nulo en el caso de posiciones no dereferenciables). • Un puntero prev al hermano izquierdo.

Estos tres punteros tienen la suficiente información como para ubicar a todas las posiciones (dereferenciables o no) del árbol. Notar que todas las posiciones tienen un puntero father no nulo, mientras que el puntero prev puede o no ser nulo. Para tener un código más uniforme se introduce una celda de encabezamiento, al igual que con las listas. La raíz del árbol, si existe, es una celda hija de la celda de encabezamiento. Si el árbol está vacío, entonces el iterator correspondiente a la raíz (y que se obtiene llamando a begin()) corresponde a ptr=NULL, prev=NULL, father=celda de encabezamiento.

LAS CLASES CELL E ITERATOR_T

Tenemos primero las declaraciones "hacia adelante" de tree e iterator_t. Esto nos habilita a declarar friend a las clases tree e iterator_t. • La clase cell sólo declara los campos para contener al puntero al hijo más izquierdo y al hermano derecho. El constructor inicializa los punteros a NULL. • La clase iterator_t declara friend a tree, pero no es necesario hacerlo con cell. iterator_t declara los campos punteros a celdas y por comodidad declaramos un constructor privado iterator_t(cell *p,cell *pv, cell *f) que simplemente asigna a los punteros internos los valores de los argumentos.