Capítulo 2 – Estructuras de datos

Introducción

Las estructuras de datos son conjuntos de datos unificados bajo determinadas características altamente usadas en las ciencias de la computación, las mismas son utilizadas tanto para modelizar problemas como así también para simplificar la programación utilizando su potencia y dinámica.

En este capítulo se busca que el lector obtenga una visión global de las características de cada una de ellas, como así también de cuando conviene y puede utilizarse cada una.

Se analizaran en el capítulo, las diferentes estructuras, con sus características diferenciales, su representación computacional y su utilización.

Concepto de Estructura de Datos

Una **estructura de datos** es un grafo dirigido y restricto, con las características de unicidad en sus relaciones, esto es que en orden de predecesor, cada nodo solo puede tener un nodo predecesor a él. Dicho de otra manera, hablamos de una estructura de datos cuando a cada nodo solo le llega un arco o flecha.

Al igual que en los grafos una **estructura de datos** mantiene una determinada dinámica de altas y bajas, esta dinámica es la que permite clasificar a las diferentes estructuras en función de cómo se comportan.

Cada estructura ofrece ventajas y desventajas en relación con la simplicidad y eficiencia para la realización de cada operación. De esta forma, la elección de la estructura de datos apropiada para cada problema depende de factores como la frecuencia y el orden en que se realiza cada operación sobre los datos, como también de los conceptos que deben ser aplicados según el modelo, como ser prioridad o jerarquía. En función de ellos, se deberá optar por la utilización de la estructura de datos que simplifique la resolución del problema a modelar.

Es importante destacar que las estructuras de datos son utilizadas para modelar problemas reales al igual que los grafos; la ventaja comparativa es que debido a las limitaciones de las mismas por ser grafos restrictos unívocos, se simplifica su administración.

También es fundamental la utilización de estas estructuras como un medio de simplificar la programación en función de dar soporte a algoritmos utilizando las diferentes características de cada una de ellas de forma tal de disminuir la tarea de programación evitando realizar engorrosos algoritmos que resuelvan un problema determinado.

Clasificación de las Estructuras de Datos

Las estructuras de datos se pueden dividir inicialmente en **estructuras biunívocas y unívocas. Las primeras** se caracterizan por ser unívocas en ambos sentidos de la relación manteniendo uno o ningún predecesor y uno o ningún sucesor. Dentro de ellas encontramos las pilas, las colas y las listas.

Por otro lado, encontramos a los **árboles,** que a diferencia de las anteriores son solo **unívocas** manteniendo un solo predecesor pero pudiendo tener más de un sucesor.

A continuación enunciaremos las características de cada una de estas estructuras.

Listas

La lista es una estructura de datos que tiene una dinámica abierta, esto es que dentro de una lista a la hora de realizar un alta se recorre toda la lista y se coloca el elemento a insertar en la posición que se requiera, esto dependerá de que se desee o no mantener la lista ordenada por algún valor de los nodos que la componen.

Del mismo modo al querer realizar una baja en la estructura se deben recorrer desde el inicio todos los valores hasta encontrar el elemento que se desee eliminar de la estructura.

Al igual que vimos en los grafos las estructuras de datos son abstractas o sea que no existen en si mismas sino que solo existen a través de una representación computacional de ellas. De esta forma podemos representar computacionalmente una lista en forma dinámica o estática.

La representación dinámica de una lista se ejemplifica en la Figura 1 a continuación:

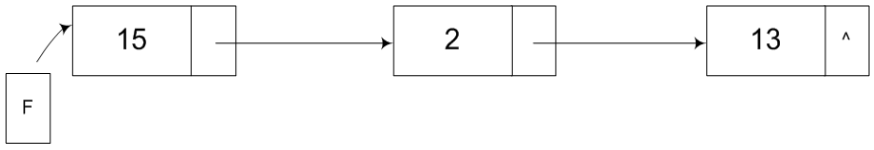

Figura 1 – Representación computacional dinámica de una Lista

Esta figura nos muestra una forma de implementar una lista sobre una representación dinámica de forma tal que solo se utiliza el espacio ocupado por los nodos que existen en un momento determinado, en este caso la lista esta compuesta por tres nodos donde cada uno de ellos cuenta con un número entero en este caso 15, luego 2 y por último 13.

Como se observa toda lista implementada en forma dinámica con nodos conectados a través de punteros debe tener un puntero inicial a toda la estructura que indica en que posición de memoria se encuentra el primer elemento de la lista que dará origen a la misma, este puntero normalmente se denomina First (derivado de primero en

inglés), el cual se representa en nuestro ejemplo como **F**, luego de ello cada nodo tiene dos componentes un componente de dato, en nuestro caso compuesto solamente por un número entero y un componente apuntador que contiene la posición de memoria del siguiente elemento de la lista, de esta forma al final de la lista el puntero que le corresponde al último nodo se encuentra en NULL indicando que es el último elemento de la lista, lo cual permite identificar que se ha llegado al final de la estructura,

Cuando el puntero que usamos para acceder a la lista (en nuestro caso F) vale NULL, diremos que la lista está vacía.

Considerando como ya se expuso que en función de la dinámica de las listas, para realizar altas o bajas debe ser recorrida la estructura para saber la posición en la cual se ingresara el elemento o para encontrar el que se quiere eliminar, las listas implementadas sobre una estructura linkeada como la que se ve en la figura mantienen los punteros en sentido hacia adelante o sea del primer elemento al último

El nodo típico utilizado para la construcción de representaciones computacionales dinámicas de una lista es:

```
struct nodo {
  int dato;
  struct nodo *siguiente;
};
```

Donde tenemos una plantilla armada con una componente de dato y una componente de puntero, encargada de encadenar la estructura apuntando al siguiente nodo de la lista.

En el ejemplo, cada elemento de la lista sólo contiene un dato de tipo entero, pero en la práctica no hay límite en cuanto a la complejidad de los datos a almacenar.

Una lista puede representarse también en forma estática, la Figura 2, nos muestra dicha representación.

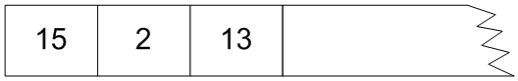

Figura 2 – Representación computacional estática de una Lista

En está representación la lista se representa sobre un vector, el cual se define de una dimensión predeterminada y se va completando en función de los elementos que se ingresan a la estructura.

A diferencia de lo que ocurre en la representación dinámica, aquí no se requieren de apuntadores al próximo elemento, dado que el siguiente elemento es el que esta a la derecha y el anterior el que esta a la izquierda de un elemento determinado.

Es así que se guardan lugares para los futuros elementos a ingresar, de forma tal que si se produce la baja o extracción de un elemento determinado deben moverse todos los elementos que estén a la derecha del mismo una posición a la izquierda de forma tal de mantener la contigüidad, del mismo modo que si se ingresa un nuevo elemento en una posición intermedia, deben moverse una posición a la derecha todos los elementos que se encuentren a la derecha de donde se va a insertar el nuevo elemento.

Tipos de listas

Una lista normalmente tiene un principio y un final, pero en función de la forma que puede tomar en su presentación encontramos tres tipos diferentes de listas estas son:

Lista lineal: es la lista tradicional, o sea, aquella que comienza con un elemento y en la cual el último puntero del nodo apunta a NULL.

Lista circular: es la lista en la cual el último nodo no lleva un apuntador en NULL, sino que apunta al primero. La Figura 3 muestra la representación de una lista circular.

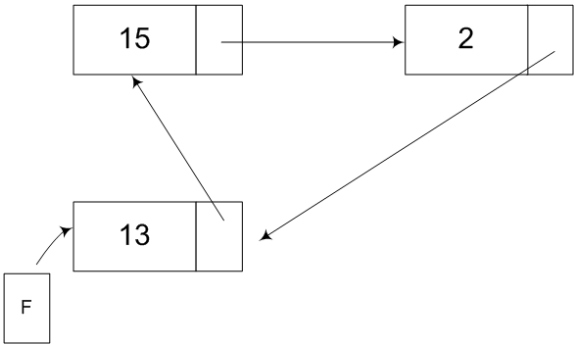

Figura 3 – Representación computacional de una Lista Circular

Aquí vemos que el último nodo de la lista en nuestro caso el que contiene el valor 13 no tiene a su apuntador en NULL, sino que apunta al primer elemento de la lista que en nuestro ejemplo es el que contiene al valor 15.

Lista doblemente enlazada: son las listas que se implementan con la posibilidad de que los nodos, aparte de tener un apuntador al nodo siguiente, tengan un

apuntador al nodo anterior. La figura 4 muestra la representación computacional de este tipo de listas.

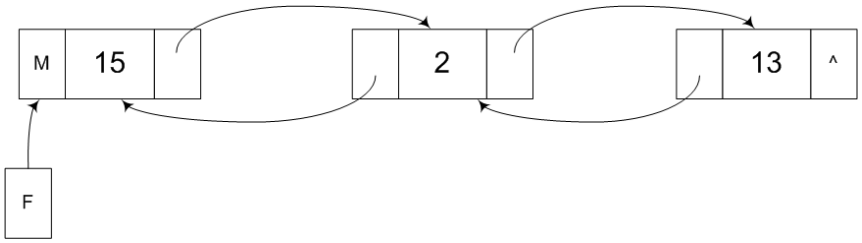

Figura 4 – Representación computacional de una Lista doblemente enlazada

Como se observa en la figura, cada uno de los nodos no solo tiene un apuntador al próximo como las listas lineales, sino también mantienen un apuntador al nodo anterior, lo cual permite la posibilidad de recorrer a la estructura en ambos sentidos del adelante para atrás y de atrás hacia delante.

Pilas

La pila es una estructura de datos que tiene como característica diferencial que su dinámica de ingreso y egreso es de tipo LIFO (del inglés *Last In First Out*, último en entrar, primero en salir) de forma tal que la forma de ingresar los datos es por un extremo de la pila y por el mismo extremo se realizan las extracciones de la misma.

En función de sus características de ingreso y egreso, podemos observar que la pila anula la prioridad, dado que al insertar y eliminar elementos por el mismo lugar el último elemento ingresado es el primero en salir, con lo cual esta estructura no administra jerarquía de llegada o ingreso.

Este tipo de estructura es de mucha utilidad cuando justamente lo que se requiere es anular la jerarquía, como ejemplo podemos mencionar la pila de procesos utilizada por los sistemas operativos, de forma tal que al finalizar el proceso en ejecución se devuelve el comando al proceso anterior a este y así sucesivamente.

En esta estructura solo se tiene acceso a una parte de la estructura comunemente denominada ***"tope"***, por ello no puede intercalarse elementos, ni borrarse en cualquier orden como en la lista, sino que por el contrario solo puede insertarse en el tope y eliminar solo el elemento que se encuentra en el tope de la pila.

Por sus características las pilas suelen utilizarse para simplificar algoritmos tales como las evaluaciones de expresiones matemáticas, convirtiendo la notación in fijo de las mismas en notación postfijo y luego resolviendo las expresiones.

Otra aplicación fundamental de las pilas es para poder implementar la recursividad, dado que esta estructura anula la prioridad y al final la ejecución de un proceso da el comando al anterior en ejecución.

Al igual que los grafos y que cualquier estructura de datos, las pilas pueden representarse computacionalmente en forma estática o dinámica, la Figura 5 representa la representación dinámica de una pila.

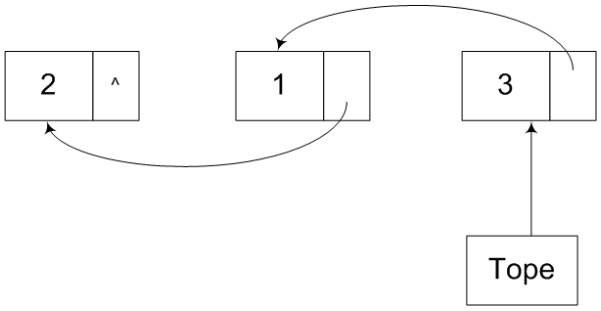

Figura 5 – Representación computacional dinámica de una Pila

Una primer característica a observar en esta representación computacional es que a diferencia de lo que ocurre en la representación computacional dinámica de las listas los apuntadores apuntan al elemento anterior, o sea de derecha a izquierda y no de izquierda a derecha, este se debe a que al realizar las bajas y altas por el mismo extremo, las pilas requieren saber cual es el elemento que quedará en el tope de la misma y ese elemento es el que ingreso inmediatamente antes que el que va a salir.

De esto podemos sacar una norma importante a tener en cuanta en las representaciones computacionales dinámicas, en ellas los apuntadores siempre deben ir en dirección inversa a la salida de los elementos, esto es si los elementos se extraen de izquierda a derecha las flechas que representan los apuntadores deben ir de derecha a izquierda, esto se debe a que cuando se requiere extraer un elemento es necesario saber cual quedará en su posición y la única forma saberlo es apuntar en el sentido inverso al orden de salida de los elementos.

En forma estática una pila al igual que una lista se representa sobre un vector, dicha representación computacional se grafica en la figura 6.

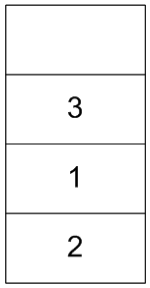

Figura 6 – Representación computacional estática de una Pila

En este tipo de representación el último elemento ocupado del vector es el que puede eliminarse y en la posición siguiente al mismo en donde se pueden insertar elementos.

Como vemos la representación estática puede asemejarse a la forma de representar una pila de platos o de cualquier otro elemento.

Colas

Una cola es una estructura de datos que se caracteriza por privilegiar el orden y la jerarquía en una estructura de datos.

Al igual que cualquier cola generada en el mundo real, como ser una cola frente a una caja de un banco o de un supermercado, donde cada elemento es una persona aguardando su turno de atención, la inserción en la cola de datos se realiza en orden, realizando el ingreso de los elementos por un extremo de la misma y la extracción de elementos por el extremo opuesto.

A este tipo de dinámica se la conoce como FIFO (del inglés *First In First Out*), debido a que el primer elemento en entrar será también el primero en salir respetando el orden de llegada.

En la siguiente figura graficamos la presentación computacional estática de una cola.

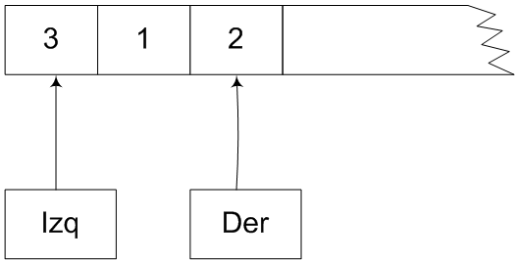

Figura 7 – Representación computacional estática de una Cola

En este tipo de representación se puede observar la presencia de dos punteros externos al vector, uno apuntando al primer elemento ingresado y otro apuntando al último elemento ingresado, de esta forma a la izquierda como primer elemento a salir encontramos al que contiene el valor 3 y que será el primer elemento a salir de la estructura, mientras que el último elemento ingresado es el que contiene el valor 2 y será el último elemento a salir, si se desea ingresar un elemento esto se realizará sobre la derecha y ahora este nuevo elemento ingresado será el último elemento a salir.

La Figura 8 muestra la representación computacional dinámica de una Cola.

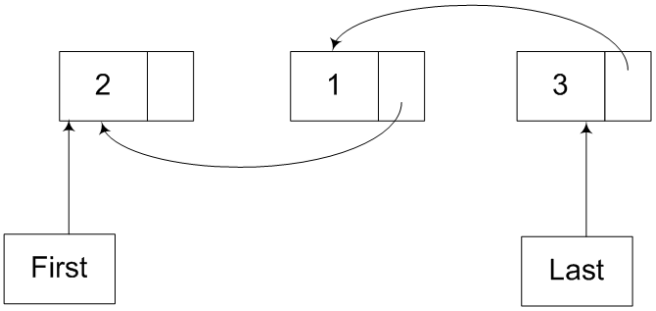

Figura 8 – Representación computacional dinámica de una Cola

En esta representación cada uno de los nodos se encuentra unido a través de un apuntador, que como ya fue explicitado anteriormente se encuentran en el sentido inverso a la salida de los elementos en este caso de derecha a izquierda, al igual que en la representación computacional estática nos encontramos con la presencia de dos punteros, los cuales apuntan al primer elemento a salir (Last) y al último elemento ingresado (First).

Un caso particular de colas son las denominadas *"colas dobles"*, las cuales tienen como característica saliente que los elementos pueden ingresar y egresar por ambos extremos, lo cual obliga a que la representación requiere de dos apuntadores uno al anterior y otro al siguiente, lo cual permite eliminar e ingresar elementos por cualquiera de los extremos.

La Figura 8 ilustra la representación computacional de este tipo de colas.

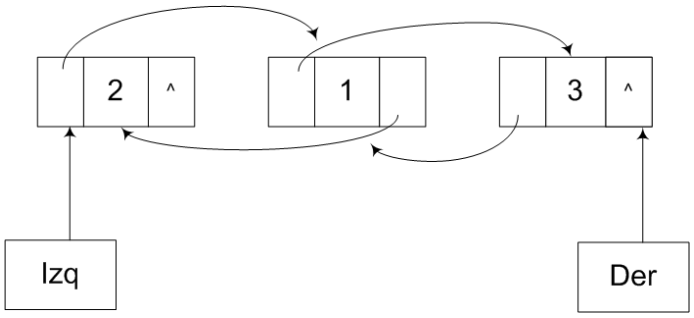

Figura 9 – Representación computacional dinámica de una Cola doble

Aquí se observa que los elementos pueden extraerse por izquierda o por derecha en función de cómo se requiera para la utilización que se le está dando a la cola.

A este tipo de colas también se las conoce como DEQUE (**D**ouble **E**nded **QUE**ue), o sea colas de doble final en su traducción al castellano.

Dentro de las aplicaciones de las colas encontramos las colas de prioridad, donde los elementos se atienden en el orden indicado por una prioridad asociada a cada uno, como puede ser una cola de impresión en un sistema operativo, donde si varios

elementos tienen la misma prioridad, se atenderán de modo convencional según la posición que ocupen.

Árboles

La última estructura de datos es el árbol, este tipo de estructura tiene la característica de no ser biunívoco, dado que solo cumple la unicidad en un sentido sabiendo que cada elemento tiene un solo predecesor pero que puede tener más de un sucesor, cosa que no ocurre con el resto de las estructuras de datos vistas hasta ahora.

Esta diferencia establece diversas características diferenciales para los árboles, que si bien pueden estudiarse para el resto de las estructuras no tiene sentido hacerlo por su carácter biunívoco.

Dentro de estas características encontramos el ***"grado"***, el grado de un árbol es la máxima cantidad de sucesores que por definición puede tener cada uno de los nodos, o sea, es el grado de expansión o crecimiento que el mismo puede tener, este grado en el resto de las estructuras analizadas es 1 (uno) y por ello no se estudia.

Al igual que como ocurre con los árboles reales, esta estructura de datos posee una raíz que da inicio y que va creciendo generando ramas que tienen origen en la raíz, el objetivo de esta estructura es justamente emular el comportamiento de los árboles reales.

El árbol es una estructura fundamental en las ciencias de la computación, se lo utiliza principalmente para representar jerarquía y ser utilizado en todo lo que conlleve establecer un orden en un conjunto de valores.

Los árboles tienen muchas aplicaciones, casi todos los sistemas operativos utilizan estructuras de árbol para almacenar sus archivos. También son usados para el diseño de compiladores, procesamiento de textos y algoritmos de búsqueda.

Al igual que el resto de las estructuras de datos, los árboles pueden representarse en forma estática o dinámica.

La figura 10 grafica la representación dinámica de un árbol binario.

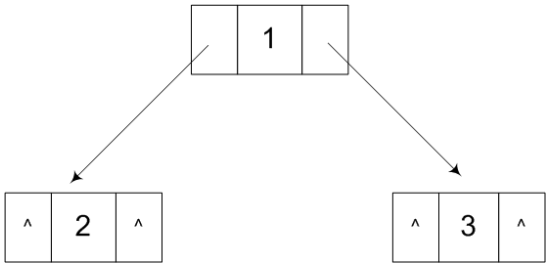

Figura 10 – Representación computacional dinámica de un árbol binario

Aquí vemos, que en función del grado del árbol en este caso dos, cada elemento componente del árbol puede tener hasta dos sucesores o hijos, de esta forma la representación dinámica utiliza una parte para representar la componente de datos de cada nodo y dos punteros, uno por cada uno de los grados que lo conforman en este caso dos, si en lugar de dos el grado fuera tres o cuatro la componente de puntero estaría compuesta por tres o cuatro punteros respectivamente.

Del mismo modo, se puede representar un árbol en forma estática, esta representación se realiza a través de un vector sin necesidad de utilizar punteros para referenciar a sus hijos.

La posición del nodo en el array corresponde a su posición en el árbol. El nodo en la posición 0 es la raíz, el nodo en la posición 1 es el hijo izquierdo, y el nodo de la posición 2 es el hijo derecho. Se sigue con este procedimiento de izquierda a derecha en cada nivel del árbol.

En cada posición del árbol, sin importar si representa un nodo existente o no, corresponde un elemento en el vector. Los elementos que representan posiciones del árbol sin nodos son rellenados con 0 o con *null*.

La siguiente figura nos muestra la representación computacional estática de un árbol binario.

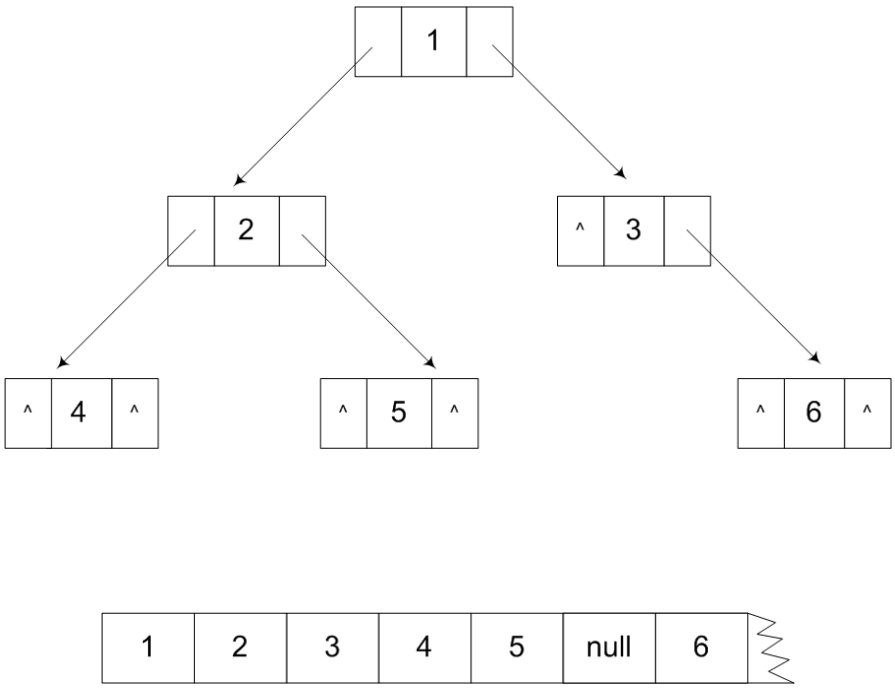

Figura 11 – Representación computacional estática de un árbol binario

Como se ve en la figura anterior, cada nodo se representa en una posición del vector desde el inicio, considerando que al ser una representación estática, se debe dejar sin utilizar el espacio destinado a aquel nodo que no existiese en un momento

determinado, como ocurre en nuestro ejemplo con el hijo izquierdo del nodo con contenido 3.

Basándose en este esquema, los hijos y el padre de un nodo pueden ser encontrados haciendo cálculos en base al índice de cada nodo en el array. Si el índice de un nodo es ***i*** entonces:

El hijo izquierdo del nodo es: ***2i + 1***

El hijo derecho del nodo es: ***2i + 2***

El padre de ***i***, si ***i*** es impar es: ***(i-1)/2*** y si ***i*** es par es: ***(i-2)/2***

En muchas situaciones, representar un árbol con un vector resulta muy eficiente. Los nodos sin llenar o nodos borrados dejan huecos en el vector, desperdiciando memoria. Sin embargo, si el borrado de nodos no está permitido, puede ser una buena alternativa, y además es útil dado que es muy costoso obtener memoria para cada nodo de forma dinámica.

Propiedades de los árboles

En los árboles se analizan determinadas propiedades o características que no tiene sentido evaluar en otras estructuras de datos debido a que son estructuras biunívocas, o sea, que tienen solo uno o ningún predecesor y uno o ningún sucesor, cosa que no ocurre en un árbol el cual puede tener más de un sucesor.

En este sentido lo primer propiedad a analizar es el ***grado***, el ***grado*** de un árbol es la máxima cantidad de hijos que puede tener cada nodo por definición, de esta forma si el grado de un árbol es dos hablamos de un árbol binario, si es tres árbol ternario, si es n árbol n-ario.

De esta forma vemos que a diferencia del resto de las estructuras de datos, el árbol tiene un crecimiento exponencial y dicho crecimiento está dado por el grado, dado que en función de este se ve la máxima cantidad de elementos que puede tener.

Otra propiedad a analizar en el árbol es el ***nivel***, el ***nivel*** es la posición en altura en la que se encuentra cada nodo considerando que se comienza en el nivel 0, que es donde reside la raíz, de esta forma en el nivel 1 se encuentran los hijos de la raíz en el nivel dos los nietos de la raíz y así sucesivamente.

De igual forma se define la ***profundidad*** de un árbol, como la cantidad de niveles que el mismo posee.

A continuación analizaremos la relación entre el grado, los niveles y la cantidad de elementos posibles en un árbol, para ello utilizaremos el árbol de la figura 12.

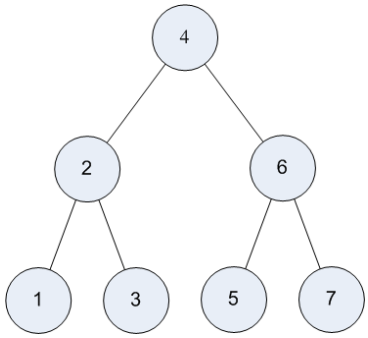

Figura 12 – Árbol binario

En función de la figura observamos que el árbol representado es de grado 2, el cual tiene tres niveles, el nivel 0 con un solo elemento que es la raíz, el nivel uno con dos elementos y el nivel dos con cuatro elementos. De esta forma vemos que en cada nivel la máxima cantidad de elementos que puede contener dicho nivel $\mathbf{grado}^{nivel}$.

Luego, si cada nivel cumple esta norma, la máxima cantidad de elementos que puede tener un árbol es $\boldsymbol{grado}^{profundidad} - 1$.

Como vemos, a diferencia del resto de las estructuras de datos que son biunívocas el árbol puede tener muchos más elementos que niveles, dado que cada elemento tiene un único predecesor, pero puede tener tantos sucesores como lo indique el grado del cual se haya definido.

Esta característica es primordial y justifica la expresión que indica “el árbol es la estructura de búsqueda por naturaleza”, esto se da, dado que las búsquedas de elementos se realizan por niveles y no por elementos, o sea, si elegimos una rama determinada del árbol para continuar con una búsqueda es porque se descartaron todos los restantes elementos que colgaban de las ramas restantes, esto es porque el árbol en función de su grado es capaz de representar más de una relación a la vez.

Por ejemplo, si quisiéramos tener una lista ordenada de números la lista representaría la relación “ser menor o igual que”, de esta forma cada elemento sería menor que el siguiente en la lista, con lo cual para ubicar un elemento habría que recorrer la lista en forma lineal hasta encontrarlo o hasta encontrar un mayor y afirmar que no existe. Si en lugar de una lista utilizamos un árbol de grado dos, son dos las relaciones que podemos representar una para los hijos izquierdos y otra para los hijos derechos, con lo cual podríamos decir que los hijos izquierdos implementan la relación “ser menor o igual que” y los hijos derechos la relación “ser mayor que”, de este modo al comparar desde la raíz si el elemento que estamos buscando es menor o igual iremos por la rama izquierda descartando todos los elementos que se encuentren colgando de la rama derecha, si por el contrario el elemento que buscamos es mayo iremos por la rama derecha descartando todos los elementos que cuelguen de la rama izquierda, así se observa, que no realizamos una búsqueda en función de la cantidad de elementos, sino que la realizamos en función de la cantidad de niveles, realizando una consulta por nivel y no todas las que corresponderían en función de la cantidad de elementos existentes en ese nivel.

A continuación analizaremos el esfuerzo de búsqueda para un conjunto de elementos representados en una lista y para el mismo conjunto representado en un árbol

binario con una relación doble, para ello consideremos el siguiente conjunto de números {3,2,8,6,5,9,4,7}, si representamos este conjunto en un lista ordenada quedaría como:

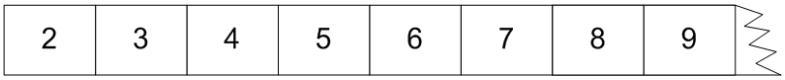

Figura 13 – Lista ordenada de menor a mayor.

Si en lugar de utilizar una lista utilizáramos un árbol binario, donde a izquierda se insertan los menores o iguales y a derecha los mayores, nos quedaría así:

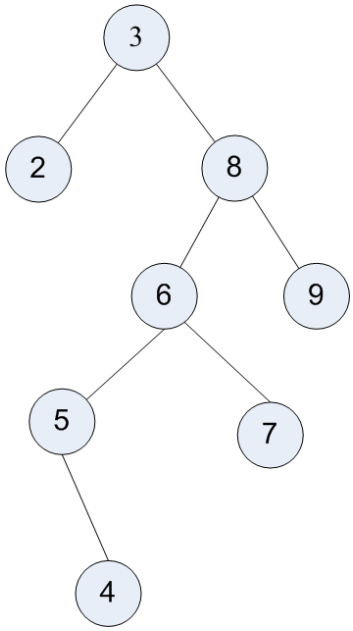

Figura 14 – Árbol binario de búsqueda

En función de la potencia de búsqueda del árbol, la cual se basa en la cantidad de niveles y no en la cantidad de elementos vemos que en el caso de la lista la máxima cantidad de comparaciones necesarias para encontrar un elemento es ocho, lo cual coincide con la cantidad total de elementos, sin embargo para el caso del árbol la máxima cantidad de comparaciones a realizar es cinco, que es la cantidad de niveles del mismo, o sea, su profundidad que casi siempre va a hacer muy inferior que la cantidad de elementos, dependiendo esto del grado de balanceo y completo que este el árbol, conceptos que trataremos a continuación.

Recorrido en árboles

A diferencia del resto de las estructuras de datos, los árboles puede ser recorridos de diferentes formas, muchos de los algoritmos que utilizan árboles tienen en común que visitan a los nodos del árbol en forma sistemática. Es decir que el algoritmo recorre la estructura de datos del árbol y realiza algún cálculo en cada uno de los nodos. El proceso de moverse entre los nodos de un árbol es llamado recorrido de árbol.

Esencialmente hay dos métodos distintos para visitar todos los nodos de un árbol: recorrido ***primero en profundidad*** y recorrido ***primero en anchura (u orden de nivel)***.

Cada uno de estos métodos tiene características diferentes, el método primero en profundidad eligen la rama izquierda y continúa por ella hasta no poder avanzar más en cuyo caso sube un nivel y pasa a la rama derecha para luego volver a intentar por la rama izquierda del nuevo nodo hasta agotar este camino, de esta forma vemos que no se pasa a una nueva rama hasta no haber agotado la anterior.

Entre los de *primero en profundidad* se encuentran: ***preorden, postorden, inorden***, la diferencia entre los mismos no es la forma de recorrer que es en profundidad, sino la forma en que se leen los nodos del árbol, el recorrido ***preorden***, lee el nodo apenas lo visita, el recorrido ***postorden*** lee el nodo cuando se va a ir de él y no volverá a esa posición, por último el recorrido ***inorden*** lee el nodo cuando lo visita por segunda vez.

El método primero en anchura en lugar de insistir hacia ala profundidad del árbol por la primer rama, recorre al árbol por niveles, o sea no avanza a visitar un nuevo nivel si haber visitado primero todos los nodos del nivel anterior.

A continuación se analizará en particular cada uno de ellos.

Preorden

El recorrido en preorden puede ser definido recursivamente de la siguiente manera:

1. **Visitar primero la raíz.**
2. **Hacer un recorrido preorden a cada uno de los subárboles de la raíz, uno por uno en un orden determinado.**

En el caso de de un árbol binario, el algoritmo resulta:

1. **Visitar primero la raíz**
2. **Recorrer el subárbol izquierdo**

3. **Recorrer el subárbol derecho**

Inorden

El recorrido inorden, también llamado simétrico, solo puede ser usado en árboles binarios. En este recorrido, se visita la raíz entre medio de las visitas entre el subárbol izquierdo y derecho.

1. **Recorrer el subárbol izquierdo**
2. **Visitar la raíz**
3. **Recorrer el subárbol derecho.**

Postorden

El último de los métodos de recorrido de árboles de *primero en profundidad* es el postorden. A diferencia del preorden donde se visita primero la raíz, en el recorrido postorden se visita la raíz a lo último.

1. **Hacer un recorrido preorden a cada uno de los subárboles de la raíz, uno por uno en un orden determinado**
2. **Visitar por ultimo la raíz**

En el caso de los árboles binarios:

1. **Recorrer el subárbol izquierdo**
2. **Recorrer el subárbol derecho**
3. **Visitar por último la raíz**

Primero en anchura

Mientras que los recorridos *primero en profundidad* son definidos de forma recursiva, a los de *primero en anchura* no se los considera recursivos, aunque podrían serlo. El motivo por el cual no serían recursivos puros es por que la recursividad utiliza como apoyo la “píla” de procesos del sistema operativo, de esta forma dicha pila anula

prioridad, dado que el último ingresado es el primero en salir, de esta forma no podría aplicarse al recorrido primero en anchura debido a que este tipo de recorrido, recorre el árbol por niveles, o sea, respetando la prioridad del nivel.

Visto así, para poder realizar una búsqueda de *primero en anchura* es necesario valerse de una estructura de datos auxiliar que permita mantener la prioridad de los elementos a evaluar, está estructura como ya lo vimos anteriormente es una cola.

De esta forma el proceso sería el siguiente:

1. **Poner en cola la raíz**
2. **Mientras que la cola no esté vacía**
   a. **Quitar primero de la cola y asignarlo a una variable auxiliar: *Nodo*.**
   b. **Imprimir el contenido de *Nodo***
   c. **Si *Nodo* tiene hijo izquierdo, poner al hijo izquierdo en la cola**
   d. **Si *Nodo* tiene hijo derecho, poner al hijo derecho en la cola**

La siguiente figura nos representa en forma visual como se realizará en un árbol binario determinado el recorrido de los nodos en las cuatro formas tratadas.

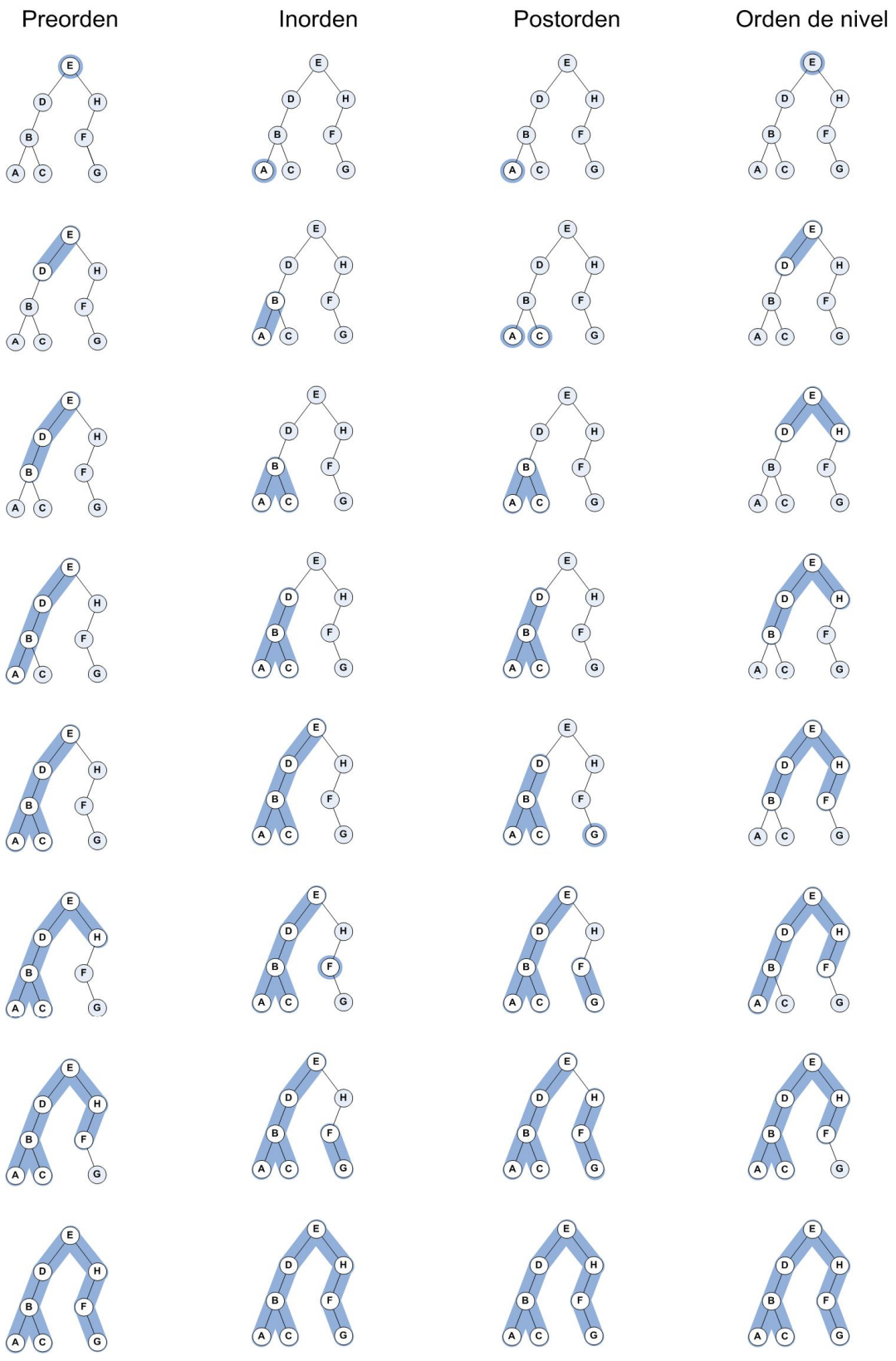

Figura 12 – Recorrido en árboles

Características específicas de los árboles

En función de las propiedades de los árboles es importante analizar determinadas características específicas de los mismos, dichas características generaran diferentes tratamientos y permitirán establecer determinadas reglas cuando estas se cumplan.

Estas características persiguen acercarse lo más posible al concepto del árbol perfecto, o sea, aquel árbol que tiene todas las hojas al mismo nivel permitiendo que se cumpla el efecto de búsqueda logarítmico del árbol, el cual solo se da si la cantidad de elementos es igual al grado del árbol elevado a la cantidad de niveles menos uno, como ya fue tratado en este capítulo.

Dentro de estas características encontramos el concepto de completo, balanceado y perfectamente balanceado las cuales trataremos a continuación.

Árbol Completo

Un árbol se encuentra completo, cuando todos los nodos del árbol cumplen el grado del mismo o son hojas.

Analizando la definición anterior encontramos que para que un árbol sea considerado completo, todos sus nodos deben tener la cantidad de hijos exacta que se corresponde con el grado del árbol (cumplir el grado del árbol) o no tener ningún hijo (ser hoja).

La siguiente figura nos muestra un árbol completo:

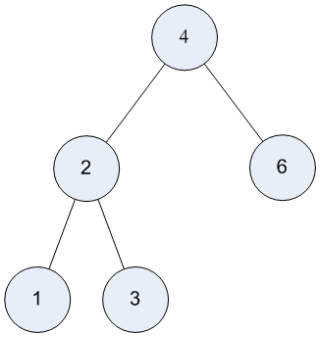

Figura 13 – Árbol Completo

Como se observa en la figura está característica no asegura que todas las hojas del árbol queden al mismo nivel, obteniendo un árbol perfecto, o sea que potencie al máximo el grado logarítmico de búsqueda del mismo.

Árbol Balanceado

Un árbol se encuentra balanceado, cuando todos sus subárboles tienen el mismo peso o una diferencia indivisible entre ellos. Esto es que cada subárbol que compone el árbol debe tener la misma cantidad de elementos o una diferencia que no pueda dividirse, o sea, que no pueda pasarse elementos de un subárbol a otro debido a que en ese caso la diferencia sería la misma pero para un subárbol diferente.

La siguiente figura nos muestra un árbol balanceado:

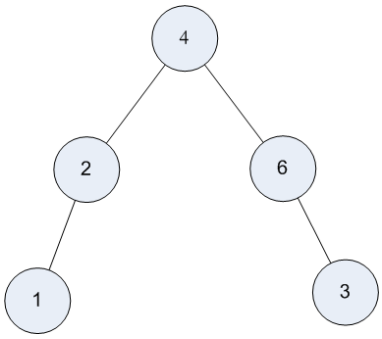

Figura 14 – Árbol Balanceado con misma cantidad de elementos entre subárboles

Como se observa en la figura el árbol está balanceado, debido a que el subárbol derecho desde la raíz y el subárbol izquierdo desde la raíz tienen la misma cantidad de elementos.

A pesar de que el ejemplo se exponga con un árbol binario, cabe señalar que el concepto de balanceado esta asociado a todos los árboles no solo a los binarios, sino también a los ternarios, cuaternarios y n-arios.

En la siguiente figura analizamos otro árbol también balanceado, pero con la particularidad que la cantidad de elementos de cada subárbol no es la misma, sino que es balanceado porque dicha diferencia es indivisible.

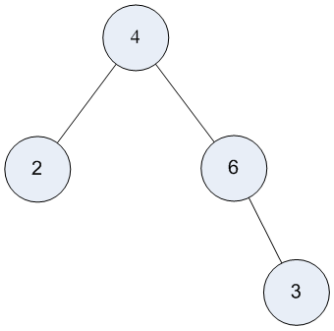

Figura 15 – Árbol Balanceado con distinta cantidad de elementos entre subárboles

Árbol Perfectamente Balanceado

Como vimos en los ejemplos anteriores, no alcanza con que un árbol se encuentre completo para que consiga tener la máxima cantidad de elementos para los niveles con los que cuenta, del mismo modo tampoco alcanza con que un árbol se encuentre completo y balanceado, dado que puede ocurrir que cumpla ambas condiciones y sin embargo no tenga la máxima cantidad de elementos posibles para los niveles con que cuenta. Esta situación se gráfica en la siguiente Figura:

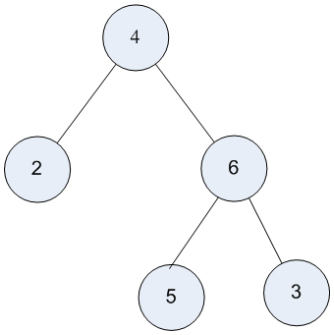

Figura 16 – Árbol Completo y Balanceado

Como se observa en la combinación de que un árbol se encuentre completo y balanceado no asegura que todas las hojas del árbol queden al mismo nivel, obteniendo un árbol perfecto, o sea que potencie al máximo el grado logarítmico de búsqueda del mismo.

De aquí surge el concepto de perfectamente balanceado, diremos que un árbol está perfectamente balanceado cuando el mismo se encuentra balanceado en todos sus niveles, o sea, que no solo se considera el balanceo parados en la raíz, sino, que después

de evaluar el balanceo desde la raíz se debe evaluar el balanceo de cada uno de sus subárboles y así sucesivamente hasta llegar a las hojas.

De este modo nos encontraremos que los árboles perfectamente balanceados son aquellos que tienen todas las hojas al mismo nivel, cumpliendo de esta forma que la cantidad de elementos que tienen es la máxima que puede haber para la cantidad de niveles con que cuenta el árbol.

La siguiente figura ilustra un árbol perfectamente balanceado:

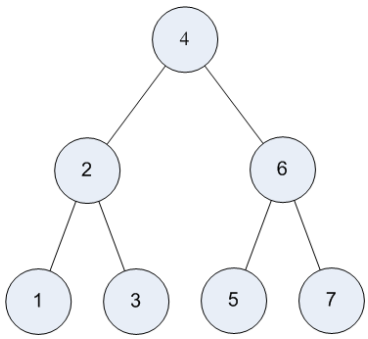

Figura 17 – Árbol Perfectamente balanceado

Árboles Típicos

En las ciencias de la computación existen determinados tipos de árboles que forman el conjunto de árboles típicos utilizados para resolver problemas específicos. A continuación trataremos en este capítulo los árboles típicos más relevantes para nuestro estudio.

Árbol Binario de Búsqueda

Como lo indica su nombre, este árbol en particular tiene la característica de poder mantener datos ordenados y permitir búsquedas de valores, es un árbol de grado 2 con la particularidad que la regla de armado consiste en que los valores que se van añadiendo al árbol si son menores o iguales que la raíz se ubican a la izquierda de la misma y si son mayores a la derecha de dicha raíz.

De esta forma se va construyendo el árbol en función a como ingresan los elementos permitiendo luego acceder a los mismos sin pasar por todas las posiciones posibles, tomando solo el camino correcto a la solución.

Supongamos que tenemos una lista de números que se ingresan en forma desordenada, como ser, {14, 9, 5, 6, 17, 34, 4, 2, 11, 19}, de acuerdo a lo definido como

construcción del árbol binario de búsqueda cada elemento que se ingresa se va insertando en la posición que corresponda, inicialmente el primer elemento que ingresa en este ejemplo es “14”, por ello este tomará la posición de raíz debido que inicialmente el árbol está vacío.

El próximo elemento a ingresar es “9”, como dicho elemento es menor o igual que la raíz lo colocaremos a la izquierda de la misma siendo este el hijo izquierdo de la raíz, el próximo elemento es “5”, al compararlo con la raíz que es “14” es menor por lo cual va a la izquierda, como el hijo izquierdo de la raíz es “9” y también es menor que nueve se inserta como hijo izquierdo del nodo que contiene el “9”, del mismo modo se van ingresando el resto de los elementos, realizando desde la raíz en adelante las comparaciones con los valores existentes en el árbol y tomando el camino por izquierda o por derecha según corresponda en función del valor que van teniendo los elementos a ingresar.

La siguiente figura ilustra como queda armado el árbol binario de búsqueda luego de ingresar todos los elementos:

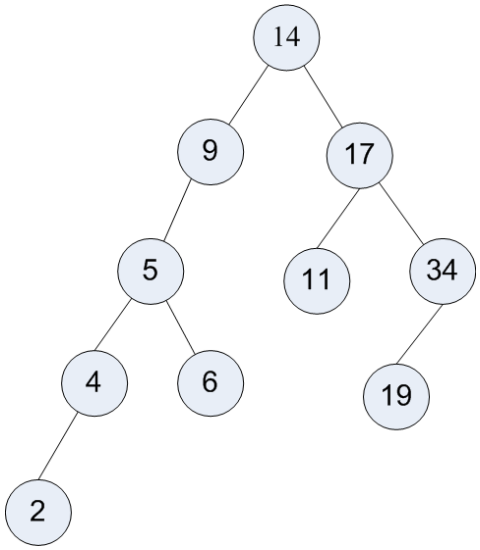

Figura 18 – Árbol Binario de búsqueda

De esta forma se puede ver que al querer acceder a cualquier elemento del árbol, la cantidad de comparaciones a realizar es mucho menor que si los elementos se hubieran cargado en una lista, esto se debe a que a lo sumo se realizan tantas comparaciones como niveles tenga el árbol, en este caso el árbol cuenta con cinco niveles, por lo cual para el peor caso que desee ubicar un valor a lo sumo habría que realizar 5 comparaciones, situación que en otra estructura de datos sería igual a la cantidad de elementos ingresaos en que en este caso es diez.

Este es el motivo por el cual se considera a los árboles como las estructuras de ordenamiento y búsqueda más óptimas.

Árbol Heap

Este árbol en particular se caracteriza por ser binario, o sea de grado 2 y completo, esto es que su armado se realiza en orden de arriba hacia abajo y de izquierda a derecha, de forma tal que no queden huecos intermedios. Esto lo convierte en un árbol donde la cantidad de niveles es la mínima con la que se puede contar para la cantidad de elementos que se están ingresando.

Además de las características mencionadas la característica saliente del mismo es que los elementos deben ingresarse de forma tal que en todos los niveles del árbol, el elemento padre debe ser mayor que sus hijos, por ello cuando se ingresa un elemento que no cumple esta condición deben rotarse los elementos para que la cumplan.

Siguiendo con el ejemplo anterior utilizado para el árbol binario de búsqueda supongamos que desea mantener un árbol heap con la lista de números {14, 9, 5, 6, 17, 34, 4, 2, 11, 19}, en función de las características del heap, el árbol debe construirse de arriba hacia debajo de izquierda a derecha por ello el elemento que ocupará el lugar de la raíz es el primero a ingresar, luego el siguiente elemento a ingresar debería ser el hijo izquierdo de la raíz para respetar el armado del árbol en orden si huecos, es allí cuando se comparan los valores para ver si es necesario rotar los elementos, en este caso no lo es, dado que el “9” es menor que el “14” y se cumple la condición de que el padre debe ser mayor que sus hijos.

Luego se deberá ingresar el siguiente elemento que deberá ir colocado como hijo derecho de la raíz, en este caso tampoco es necesario rotar los elementos por el elemento siguiente es el “5” que cumple la condición de ser menor que el “14” que es su padre, a este momento el árbol heap completado en forma parcial quedaría armado como lo muestra la siguiente figura:

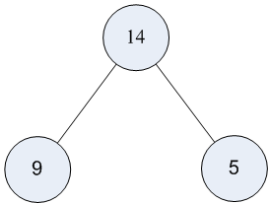

Figura 19 – Árbol Heap Parcial

Continuando con los elementos a ingresar el próximo elemento en orden posicional debe ser el hijo izquierdo del 9, en esa posición irá el 6 que como es menor que el nuevo no requiere que se roten los elementos. Al ingresar el siguiente elemento que es el “17” y que debe ir en la posición de hijo derecho del “9” ocurre que este elemento es mayor que su padre por lo cual hay que rotar los elementos y colocar el “17” en la posición del “9” y viceversa, Luego ocurre que el elemento recién ingresado que es el “17” tampoco cumple la condición de ser menor que su padre que ahora es el

“14”, por lo cual también deben rotarse estos elementos quedando el árbol como lo ilustra la siguiente figura:

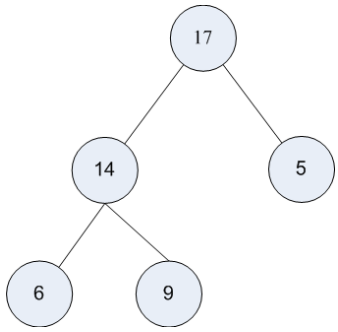

Figura 20 – Árbol Heap Parcial

Continuando con el resto de los elementos con la misma operatoria realizada hasta ahora nos quedará el árbol heap completo y conteniendo la mínima cantidad de niveles posibles para la cantidad de elementos existente como lo ilustra la siguiente figura:

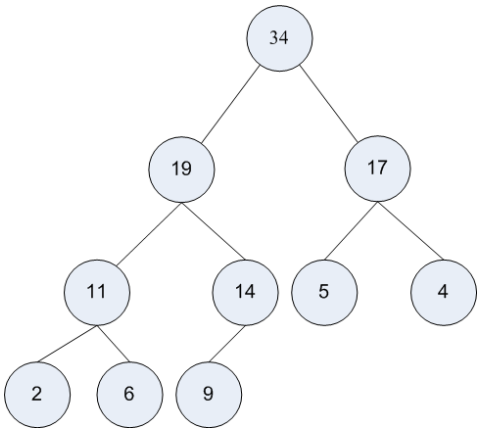

Figura 21 – Árbol Heap Final

Árboles M-arios

Los árboles m-arios, son aquellos que se caracterizan por poseer un grado m, esto es, un grado normalmente mucho mayor a 2, si bien su administración es más compleja que la de los árboles binarios también es muy superior su ventaja en la búsqueda y mantención ordenada de valores.

Su mayor utilización se aplica a la construcción de índices y este tipo de árboles en particular será tratado en capítulos posteriores cuando se expliquen los distintos mecanismos de creación de índices y búsqueda.

Utilización de las Estructuras de Datos

Las estructuras de datos son una de las bases fundamentales de las ciencias de la computación; al igual que los grafos, las estructuras de datos se utilizan para modelizar distintas situaciones que permitan encontrar la solución de un problema determinado.

En función de la dinámica de cada una de las estructuras, esto es en función de cómo se administran las altas y las bajas, las estructuras se clasifican en diferentes tipos como lo vimos hasta el momento, donde cada una de ellas tiene una administración y operatoria diferente que permite que se utilice en función de la necesidad.

De esta forma, vemos por ejemplo que una pila se caracteriza por anular la prioridad, debido a que el último elemento ingresado es el primero en salir, de forma tal que no considera prioridad de ingreso. Esto la hace muy útil para problemas donde se requiera volver hacia atrás sobre los mismos pasos que se dieron y no en el orden de prioridad. A partir de ello vemos por ejemplo que los sistemas operativos implementan una pila para administrar los procesos pendientes que se están ejecutando y que invocaron a otros procesos nuevos, de forma tal de que al finalizar un proceso, se retorna a la instancia inmediata anterior y no a la primera que se ejecutó. Esto resulta muy útil también para la resolución de algoritmos iterativos que repitan una operación constantemente ya que permite aplicar el concepto de recursividad (dado cuando una función específica se invoca a sí misma), de forma tal de realizar el mismo conjunto de instrucciones pero en instancias diferentes y con valores de variables diferentes.

Por otro lado, una cola es la situación inversa a la pila, dado que su dinámica de ingreso y egreso se basa en la prioridad, esto es que el primero que ingresó es el primero que saldrá, de esta forma estas estructuras son utilizadas por ejemplo para administrar colas de impresión, donde los procesos de impresión deben resolverse manteniendo un orden de ejecución, ejecutando primero los más antiguos y luego los nuevos en ingresar.

También en lo que respecta a algoritmos, las colas son muy utilizadas como se observó anteriormente: cuando quiere recorrerse un grafo en anchura, primero requiero visitar a todos los nodos conectados directamente con el actual y luego a los conectados indirectamente; de esta forma la pila permite mantener ese orden de evaluación marcado en este caso por la prioridad de relación establecida (directa o indirecta).

Para los casos donde la prioridad es establecida externamente y no por el orden lógico de ingreso y egreso existen las listas, donde es posible ingresar y egresar elementos en cualquiera de las posiciones, permitiendo de esta manera administrar la dinámica de la estructura a través de un parámetro externo. Por ejemplo, supongamos que se quiere implementar un proceso para administrar la impresión en un sistema operativo, pero que se debe administrar con prioridades o sea que no todos los usuarios tienen la misma prioridad de impresión, existiendo usuarios con prioridad más alta que otros. En este caso, una cola no podría ser utilizada para resolver el problema, dado que la cola manejaría la prioridad ligada al ingreso del trabajo de impresión y no tendría en cuenta la jerarquía del usuario que lo envío. Por lo tanto, en lugar de una cola se requiere implementarlo sobre una lista, de forma tal que la lista en función de la jerarquía del usuario que envía el trabajo lo inserta en la posición correspondiente, ejecutando los trabajos en orden de jerarquía y no de llegada.

Por último, nos encontramos con los árboles, como ya se mencionó, esas estructuras tienen la característica de no ser biunívocas; esto es que su crecimiento será

exponencial en función del grado, en este caso dichas estructuras permiten modelizar relaciones que no necesariamente sean del tipo uno contra uno, situación obligatoria para las pilas, colas y listas.

Supongamos por ejemplo que pretendemos mantener un conjunto de valores ordenados, si planteamos la solución creando una lista donde la relación establecida es ser menor o igual que el siguiente, nos quedará una lista ordenada de menor a mayor, lo que permitirá ubicar a un elemento determinado recorriendo la lista. Ahora bien, dado que un árbol tiene la virtud de poder implementar relaciones múltiples, se podría aplicar la misma solución sobre el árbol indicando que los menores de un valor dado sean hijos izquierdos y los mayores hijos derechos. De esta forma, estamos implementando dos relaciones diferentes sobre la misma estructura y esto nos permite ubicar un elemento determinado realizando muchas menos comparaciones que en una lista, dado que si el elemento que estamos buscando es menor iremos por la izquierda y si es mayor solo por la derecha, descartando todos los demás elementos. Es por ello, que se indica que los árboles son las estructuras más eficientes de ordenamiento y búsqueda, dado que permiten acceder a los diferentes elementos que lo componen realizando muchas menos comparaciones que en una estructura lineal como la lista.