Grafos Estructuras de datos

Lista de ARCOS: El grafo conoce una lista de vértices y una lista de arcos. Los vértices conocen sus rótulos respectivos y su posición en la lista de vértices. Los arcos conocen su peso, los vértices que unen y su posición en la lista de arcos.

Lista de Adyacencias: El grafo conoce una lista de vértices y una lista de arcos y cada vértice conoce su rótulo y los arcos que inciden en él. Los arcos conocen los vértices que unen y su peso. La mejora que propone la lista de adyacencias es introducir una lista que para cada vértice indica los arcos que emergen cuando el grafo es dirigido y los que emergen/inciden cuando el grafo es no dirigido.

Matriz de adyacencias: En la representación de matriz de adyacencias además de la matriz de arcos tendremos una lista de vértices y una lista de arcos.

Cada vértice conoce su posición en la lista de vértices, el rótulo del vértice y su índice en la matriz de arcos.

Como usamos Java, si tenemos n vértices, los índices de vértices van de 0 a n-1. Cada arco conoce su posición en la lista de arcos, los vértices que conecta y el peso del arco.

Operación	Tiempo
vertices()	O(n)
edges()	O(m)
endVertices(e), opposite(v,e)	O(1)
incidentEdges(v), areAdjacent(v,w)	O(m)
replace(v,x), replace(e,x)	O(1)
insertVertex(x), insertEdge(v,w,x), removeEdge(e)	O(1)
removeVertex(v)	O(m)

Operación	Tiempo
vertices()	O(n)
edges()	O(m)
endVertices(e), opposite(v,e)	O(1)
incidentEdges(v), areAdjacent(v,w)	O(deg(v))
replace(v,x), replave(e,x)	O(1)
insertVertex(x), insertEdge(v,w,x), removeEdge(e)	O(1)
removeVertex(v)	O(deg(v))

Operación	Tiempo
vertices()	O(n)
edges()	O(m)
endVertices(e), opposite(v,e), areAdjacent(v,w)	O(1)
incidentEdges(v)	O(n+deg(v))
replace(v,x), replace(e,x), insertEdge(v,w,x), removeEdge(e)	O(1)
insertVertex, removeVertex	$O(n^2)$

Lista de Arcos:

- Uso Común: Es útil cuando las operaciones más comunes son iterar sobre todos los arcos del grafo o buscar un arco específico.
- Ventajas: Eficiente para recorrer todos los arcos. Es compacta si el grafo es disperso y hay pocos arcos en comparación con el número total de vértices.
- Desventajas: Ineficiente para buscar rápidamente la presencia de un arco específico o para obtener los vecinos de un vértice.

Lista de Adyacencias:

- Uso Común: Óptima cuando las operaciones más frecuentes son encontrar los vecinos de un vértice o determinar si hay un arco específico entre dos vértices.
- Ventajas: Eficiente para acceder a los vecinos de un vértice. Ocupa menos espacio que una matriz de adyacencias para grafos dispersos.
- Desventajas: Menos eficiente para iterar sobre todos los arcos del grafo.

Matriz de Adyacencias:

- Uso Común: Adecuada cuando se necesitan verificar rápidamente la presencia de un arco específico o cuando el grafo es denso y la mayoría de los vértices están conectados.
- Ventajas: Eficiente para determinar si hay un arco específico entre dos vértices. Funciona bien para grafos densos.
- Desventajas: Puede ocupar más espacio en memoria que otras estructuras para grafos dispersos. Menos eficiente para recorrer todos los arcos del grafo.

Grafo Estático:

Definición: En un grafo estático, la cantidad de vértices y arcos no cambia después de su creación.

Ventajas:

- Eficiencia: Al ser estático, ciertas operaciones y estructuras pueden ser más eficientes, ya que se conocen las dimensiones fijas del grafo.
- Simplicidad: La estructura se mantiene constante, lo que puede simplificar el diseño y la implementación.

Desventajas:

- Limitaciones en la Actualización: No se pueden agregar ni eliminar vértices o arcos después de la creación del grafo.

Grafo Dinámico:

Definición: En un grafo dinámico, es posible agregar o eliminar vértices y arcos después de su creación inicial.

Ventajas:

- Flexibilidad: Permite la modificación de la estructura del grafo durante la ejecución del programa, adaptándose a cambios en el modelo o requisitos.
- Aplicaciones Dinámicas: Útil en situaciones donde la estructura del grafo debe ajustarse dinámicamente a eventos o cambios en el sistema.

Desventajas:

- Complejidad: La gestión dinámica puede aumentar la complejidad del código y la implementación.
- Posible Pérdida de Eficiencia: Algunas operaciones pueden ser menos eficientes en comparación con un grafo estático debido a la gestión dinámica.

Un grafo estático podría representar la estructura de conexiones entre ciudades en un mapa, donde las ciudades y las conexiones no cambian con frecuencia.

Un grafo dinámico podría representar las relaciones de amistad en una red social, donde es posible agregar o eliminar conexiones entre usuarios en cualquier momento.

Cola Con Prioridad

Una cola con prioridad almacena una colección de elementos que soporta:

- Inserción de elementos arbitraria
- Eliminación de elementos en orden de prioridad (el elemento con 1era prioridad puede ser eliminado en cualquier momento).

Un (mín)heap es un árbol binario que almacena una colección de entradas en sus nodos y satisface dos propiedades adicionales:

- Propiedad de orden del heap (árbol parcialmente ordenado): En un heap T, para cada nodo v distinto de la raíz, la clave almacenada en v es mayor o igual que la clave almacenada en el padre de v.
- Propiedad de árbol binario completo: Un heap T con altura h es un árbol binario completo si los nodos de los niveles 0,1,2,…,h-1 tienen el máximo número de nodos posibles y en el nivel h-1 todos los nodos internos están a la izquierda de las hojas y si hay un nodo con un hijo, éste debe ser un hijo izquierdo (y el nodo debiera ser el nodo interno de más a la derecha)

	Lista no ordenada	Lista ordenada	Heap
size();	O(1)	O(1)	O(1)
isEmpty();	O(1)	O(1)	O(1)
min()	O(n)	O(1)	O(1)
insert()	O(1)	O(n)	O(log n)
removeMin()	O(n)	O(1)	O(log n)

Aplicación: Heap Sort

- Objetivo: Ordenar un arreglo A de N enteros en forma ascendente
- Estrategia: Insertar los n elementos del arreglo en un heap inicialmente vacío y luego eliminarlos de a uno y almacenarlos en el arreglo.
- Complejidad: Theapsort(n) = c1 + c2nlog2(n) + c3nlog2(n) = **O(nlog2(n))**

Arboles de Busqueda

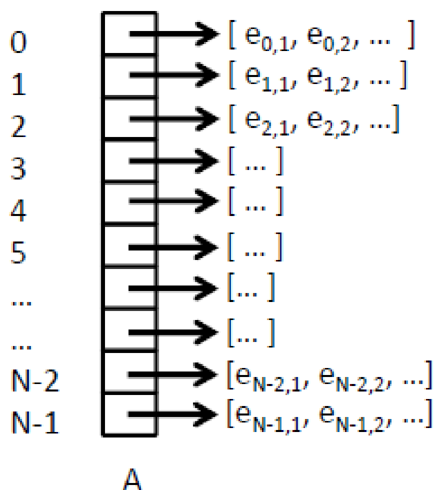
- El ABB permite implementar conjuntos y mapeos con un tiempo de operaciones buscar, insertar y eliminar con orden logarítmico en la cantidad de elementos en promedio.
- En el peor caso las operaciones tienen orden lineal en la cantidad de elementos (cuando las inserciones se realizaron en forma ascendente o descendente en cuyo caso el árbol degenera en una lista).
- Hay estructuras alternativas que garantizan tiempo de acceso de orden logarítmico en la cantidad de elementos y se los conoce como árboles de búsqueda balanceados: Árbol AVL, Árbol 2-3 y Árbol B.

Tablas Hash

Arreglo de buckets: Un arreglo de cubetas para implementar una tabla

de hash es un arreglo A de N componentes, donde cada celda de A es

una colección de entradas (pares clave-valor) ei,j



Cada colección A[i] con i=0,...,N-1 se llama *bucket* o *cubeta*.

Nota simpática: *bucket* en inglés quiere decir *balde*. Cuando los profesores éramos jóvenes, los únicos libros de computación editados en castellano eran de editoriales españolas, de ahí el término *cubeta*.

"Buena" función de hash: Una función de hash h es "buena" si h distribuye las claves uniformemente en el intervalo [0,N-1]. Intuitivamente, todas las cubetas tienen aproximadamente el mismo tamaño.

Arbol General / Binario

Los recorridos de un árbol, como preorden, inorden y postorden, son formas de visitar todos los nodos de un árbol.

Preorden:

En el recorrido preorden de un árbol T, la raíz r de T se visita primero y luego se visitan recursivamente cada uno de los subárboles de r.

Postorden:

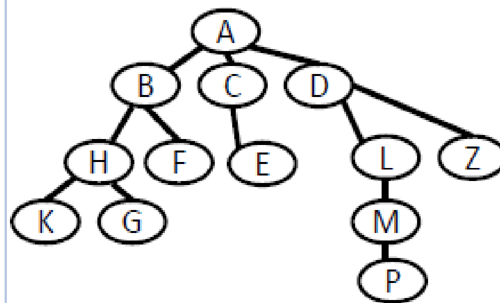
En el recorrido postorden de un árbol T, la raíz r de T se visita luego de visitar recursivamente cada uno de subárboles de r.

```
Algoritmo PreordenShell( T )
        preorden( T, T.root() )

Algoritmo preorden( T, v )
    Visitar(T, v)
    Para cada hijo w de v en T hacer
        preorden( T, w )
```

```
Algoritmo PostordenShell( T )
        postorden( T, T.root() )

Algoritmo postorden( T, v )
    Para cada hijo w de v en T hacer
        postorden( T, w )
    Visitar(T, v)
```

Inorden: En el recorrido inorder (o simétrico) de un árbol T con raíz r, primero se recorre recursivamente el primer hijo de la raíz r, luego se visita la raíz y luego se visita recursivamente al resto de los hijos de r.

```
El algoritmo se llama con inorden( T, T.root() )
•   Algoritmo inorden( T, v )
      Si v es hoja en T entonces
          Visitar( T, v )
      Sino
          w ← primer hijo de v en T
          inorden( T, w )
         Visitar(T, v)
         mientras w tiene hermano en T hacer
             w ← hermano de w en T
             inorden( T, w )
```



Salida esperada: K H G B F A E C P M L D Z

Comparador Implementado

Al implementar la interfaz Comparable<T> en tu clase, estás indicando que las instancias de tu clase son naturalmente comparables entre sí. Esto significa que defines una manera predeterminada de comparar dos instancias de esa clase.

La interfaz Comparable<T> requiere que implementes el método compareTo(T o) que devuelve un entero negativo, cero o positivo según si el objeto actual es menor que, igual a o mayor que el objeto proporcionado.

Comparador Clase Separada que Implemente Comparator<T>:

Si necesitas realizar comparaciones de maneras diferentes o si no puedes modificar la clase original, puedes crear una clase separada que implemente Comparator<T>.

Esto es útil cuando necesitas múltiples criterios de comparación o cuando no puedes modificar la implementación de la clase.

Tiempo De ejecucion:

Selectionsort : O(n^2)   InsertionSort : O(n^2)

BubbleSort : O(n ^2)