Resumen Examen

# Introducción

## Algoritmo

Un **algoritmo** corresponde a un conjunto prescrito de instrucciones o reglas bien definidas, ordenadas y finitas que permiten llevar a cabo un propósito.

Los algoritmos se dicen correctos si:

1. Termina en una cantidad finito de pasos
2. Cumple su propósito mientras mantiene las propiedades

## Complejidad

La **complejidad** de un algoritmo se define como el número de pasos a seguir para que se complete la ejecución del algoritmo, y se usa como medida de eficiencia.

*\*Los algoritmos que se usan hoy en día tienden a no superar el orden n3, es decir, ya no son útiles. Por otro lado, aquellos que son O(2n) son intratables, ya que demoran mucho en finalizar.*

## Memoria

El manejo de memoria suele ser muy importante a la hora de crear algoritmos eficientes, por lo que es común que se quiera aprovechar un espacio de memoria que ocupa un elemento que no se requerirá en el futuro, por ejemplo, al querer ordenar un arreglo es posible realizarlo dentro del mismo espacio asignado (con una variable auxiliar) en vez de asignar un espacio nuevo, esto se conoce como el manejo de memoria ***in – place***.

## Estabilidad

Un algoritmo de ordenación se considera **estable** cuando dos o más elementos con la misma llave aparecen en el *output* con el mismo orden que venían del *input*.

Ejemplo: ordenemos palabras de 5 letras acorde al alfabeto, en donde el algoritmo parte viendo la primera letra de cada palabra:

*Peach Apple*

*Straw -----\ Peach*

*Apple -----/ Straw*

*Spork Spork*

En el caso anterior, sería **inestable** si *Spork* apareciera antes que *Straw*.

# Estructuras de datos

## Arreglos

Colección de elementos que están identificados por una llave en particular (*index or key*). En un computador suelen estar de forma lineal, en donde la dirección del primer elemento es la dirección predominante a partir de la cual se pueden ubicar el resto de los elementos mediante llaves relativas en O(1) (esto es mediante indexación).

Hay de múltiples dimensiones, por ejemplo, una matriz (2D) comprende un arreglo de arreglos.

Listas Ligadas

Colección de elementos cuyo orden no es representado en memoria, sino que un elemento apunta al siguiente, por lo que se le suele referir como una secuencia de nodos. Sin embargo, para ubicar un nodo en particular toma tiempo O(n) ya que hay que encontrar al nodo que le apunta, y lo mismo para éste, por lo que se suele partir desde el nodo “*head*” como referencia. A pesar de lo anterior, son bastante eficientes para representar **colas** **(*queues*)** o **pilas** **(*stacks*)** cuyas operaciones de agregar y eliminar son O(1).

## Diccionarios

Permiten asociar, actualizar y obtener un valor a partir de una llave, en donde se busca que use solo lo necesario de memoria.

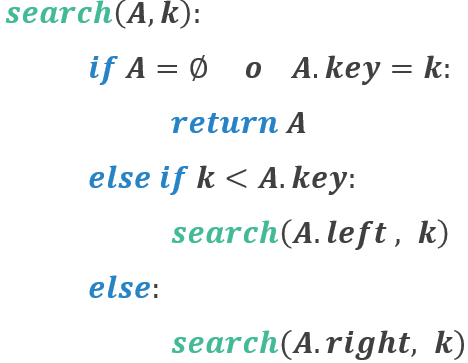
## Árboles Binarios

Estructuras de datos con un nodo raíz que tiene a lo más dos hijos, y estos, a su vez, pueden dar origen a un nuevo árbol binario de forma recursiva. El nodo raíz se dice que está a **profundidad** 1, mientras que los hijos de un mismo padre son hermanos cuyo nivel de profundidad es uno mayor que el del padre. También existe el concepto de **altura** correspondiente a la profundidad máxima del árbol.

Árbol de búsqueda binario

Estos pueden ser implementados para crear diccionarios en donde los datos están divididos en los menores y mayores según la clave del nodo actual. A partir de esta idea nace el **árbol binario de búsqueda (ABB)** que permite realizar las siguientes funciones:

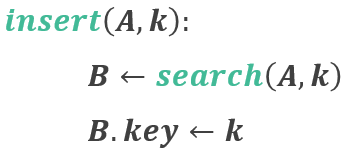
1. ***Search***: Encuentra un elemento en el árbol y retorna al nodo que lo contiene.



A: nodo actual

k: valor a encontrar

*\*Complejidad depende de la altura, en un árbol balanceado es O(log n)*

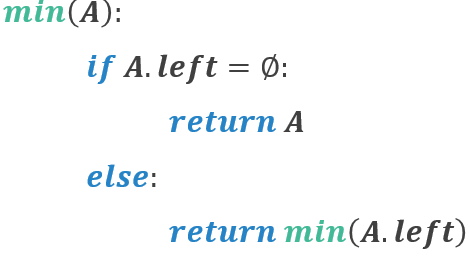
1. ***Insert***: Busca insertar un nodo en la posición que corresponda. En este caso se reemplaza el valor de la clave (se puede manejar como uno estime conveniente).

A: nodo actual

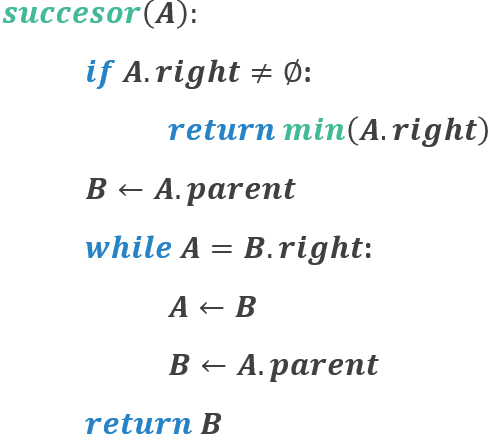
k: valor a encontrar

*\*Complejidad depende de la altura, en un árbol balanceado es O(log n)*

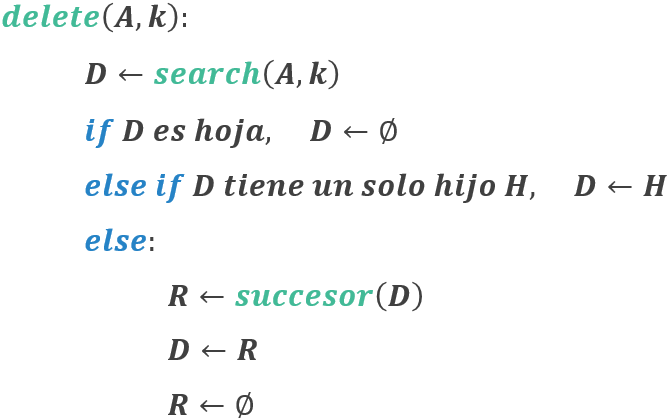
1. ***Delete***: Busca eliminar un nodo del árbol binario. Sin embargo, para lograrlo debe buscarlo realizar las asignaciones necesarias con tal de que se mantenga el orden establecido, por lo que nace el concepto de encontrar al mínimo y al sucesor:

* **Min:**

*\*El máximo se define de manera homóloga.*

* ***Succesor***:

*\*El antecesor se define de manera homóloga*



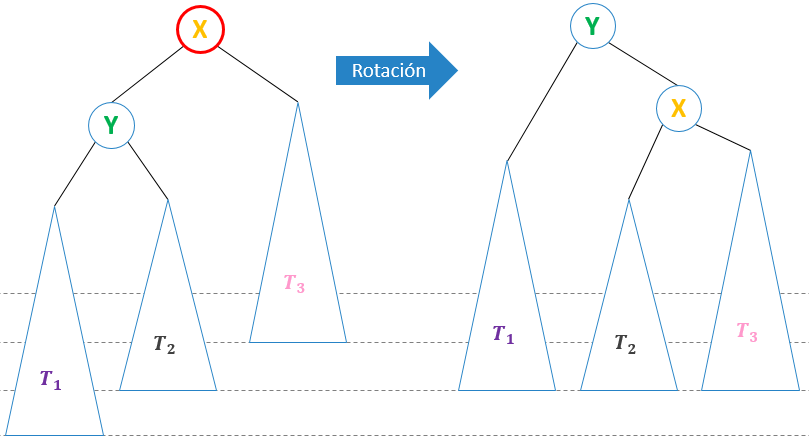
*\*La última parte es legal sólo porque sabemos que el nodo tiene dos hijos, por lo que el sucesor es una hoja*

Árbol AVL

Un ABB está **AVL – balanceado**, y por ende pasa a ser un **árbol AVL** si cumple lo siguiente:

1. Las alturas de sus hijos no difieren en más que 1 entre ellas
2. Cada hijo a su vez está AVL – balanceado

Sin embargo, es posible desbalancear un árbol AVL al aplicar las funciones de un árbol ABB, haciendo imposible asegurar una altura O(log n), por lo que se debe aplicar **rotación** para mantener las propiedades:



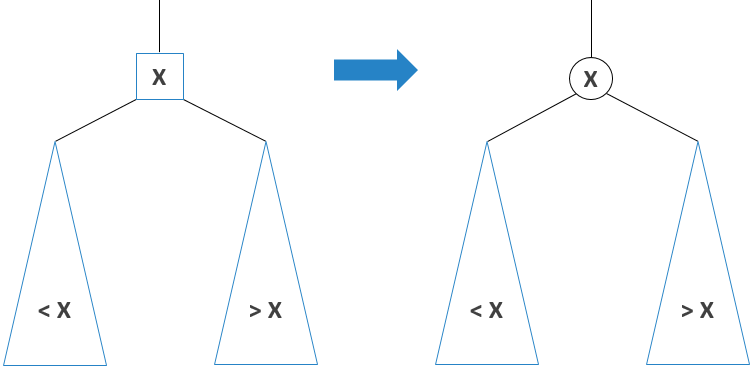
X e Y son los nodos que van camino a la inserción a partir del nodo desbalanceado más bajo (X, ya que sus hijos generan una altura con diferencia de 2). La rotación de la figura muestra una rotación en base a X-Y.

*\*En el peor caso solamente se debe realizar una rotación, ya que en caso contrario el árbol original no sería AVL*

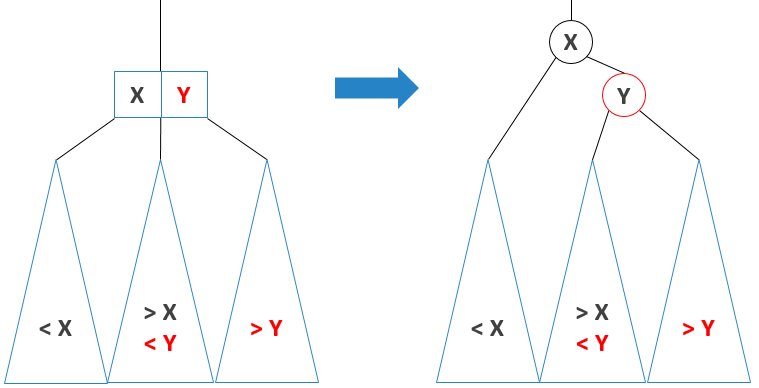
Árbol 2-3

Son árboles de búsqueda con dos tipos de nodos cuyo fin es que las hojas estén a la misma profundidad:

1. **Nodo 2**: tiene una clave y 2 hijos, por lo que es como un nodo en un ABB:



1. **Nodo 3**: tiene dos claves distintas y ordenadas con 3 hijos, por lo que es como dos nodos en un ABB:

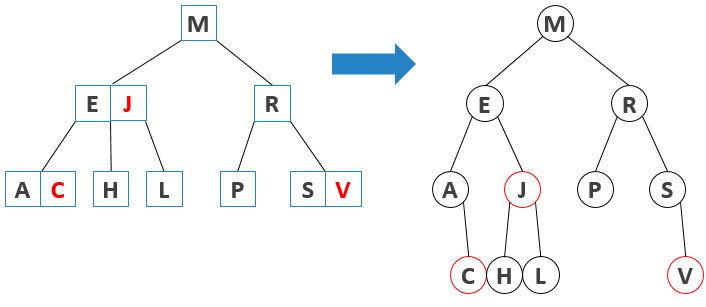


Al momento de realizar inserciones es posible que la altura cambie, pero se pierde la propiedad de que las hojas estén a la misma profundidad, por lo que la inserción sigue ciertas reglas:

1. La inserción siempre se hace en una hoja
2. Si un nodo se llena, sube el dato del medio (la clave mediana) al nodo padre
3. El árbol sólo aumenta de altura cuando se llena la raíz

Rojo – Negro

Son árboles balanceados con mucho *overhead* que pueden ser representados como ABB:

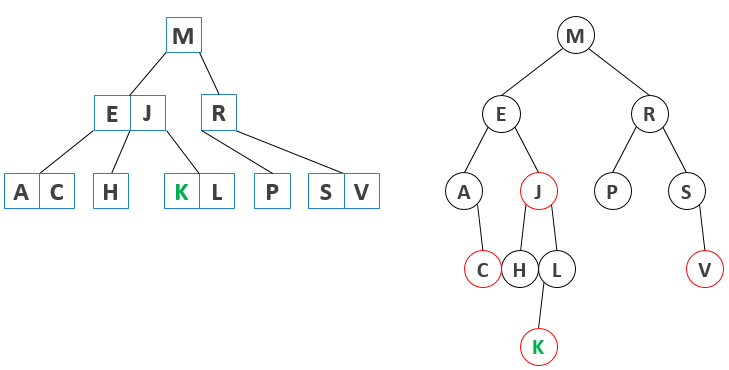


En donde los **nodos rojos** son los datos adicionales en los nodos – 3, mientras que los **negros** simbolizan los nodos del árbol 2 – 3. Este árbol resultante es conocido como **árbol rojo – negro**.

Estos árboles cumplen 4 propiedades:

1. Cada nodo es rojo o **negro**
2. La raíz es **negra**
3. Si un nodo es rojo, sus hijos deben ser **negros**
4. La cantidad de nodos **negros** camino a cada hoja debe ser la misma

Sin embargo, no todo árbol rojo – negro tiene un árbol 2 – 3, pero sí un 2 – 4:



Además, hay que tener cuidado de mantener las propiedades mediante rotaciones y actualización de colores al momento de realizar inserciones:

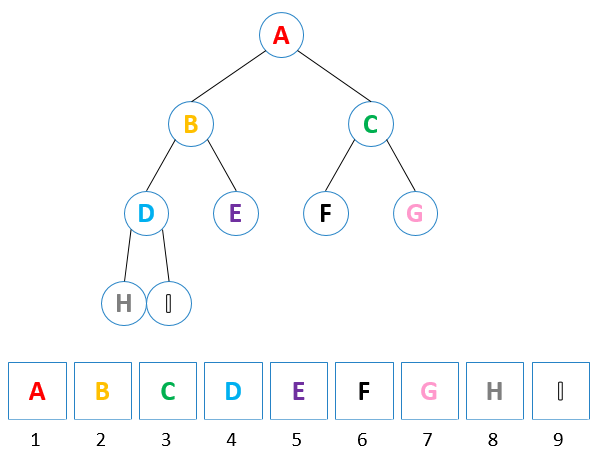
* Los nodos siempre se insertan rojos
* Si su padre es rojo, hay dos casos según el color del tío:

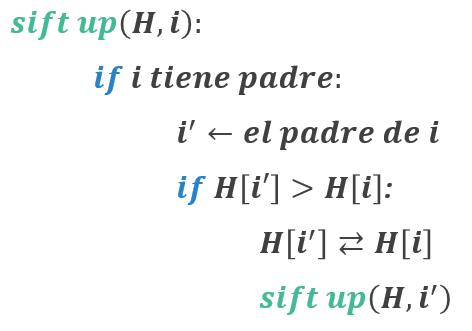
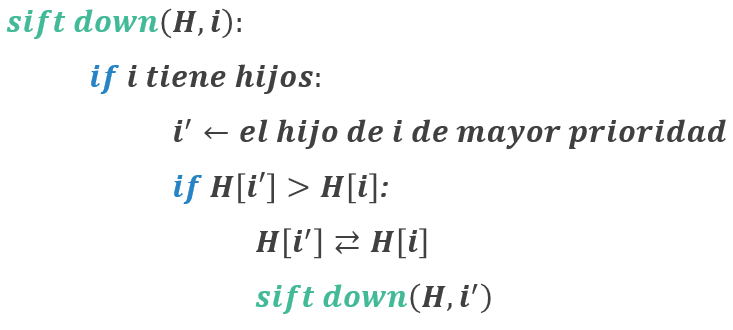
1. **Negro**: Implica el aumento de grado en el nodo del 2 – 4 que se soluciona con rotaciones y cambios de color
2. Rojo: Implica que el nodo 2 – 4 rebalsa y se soluciona cambiando colores, pudiendo generar el mismo caso de arriba

Heaps Binarios

Son estructuras recursivas (lo que simplifica la implementación de algoritmos de búsqueda) que se comportan como una **cola de prioridades**, pero en forma de árbol binario, en donde los niveles de profundidad ascienden/descienden en prioridad (***max/min heap***). Al igual que estas, buscan insertar elementos, extraerlos del nodo raíz (aquel con mayor/menor prioridad), e idealmente, cambiar prioridades de los datos insertados.

Para lo anterior resulta útil implementar un *heap* como un arreglo, en donde *2i* es el hijo izquierdo de *i*, y *2i + 1* es el derecho, asimismo, *floor(i / 2)* es el padre de *i* como se muestra en la siguiente imagen:



Así, es posible subir y bajar elementos hasta donde corresponda mediante:

## Tablas de Hash

Introducción

Son implementaciones de diccionarios que no tienen noción de orden y sus operaciones son O(1) en promedio, ya que son estructuras de datos que asocian llaves o claves a valores. Su objetivo es soportar eficientemente la búsqueda de elementos, y funciona como sigue: Transforma la clave a un **valor de hash** mediante una **función de hash** para poblar una **tabla de hash** utilizado los siguientes métodos:

1. **División**: Implica usar el módulo *m* del valor de hash de un objeto para obtener su llave en la tabla de la siguiente forma,

*h’(x) = h(x) mod m,* m es el tamaño de la tabla

Sin embargo, si *m* es potencia de 2 o 10 se puede perder información de *x*, ya que no todo valor de *h(x)* es usado para calcular *h’(x)*.

1. **Multiplicación**: Implica usar la función suelo, una constante *A* y el *mod 1* con el valor de hash para obtener la llave del objeto “hasheado” de la siguiente forma,

*h’(x) = floor(m \* (A \* h(x) mod 1)),* m es el tamaño de la tabla

notar que *A \* h(x) mod 1* es la parte fraccional de *A \* h(x)*. Esta operación es más costosa que el método de la división, pero no depende del tamaño de la tabla de hash. Suele recomendarse *A = 1 / phi*.

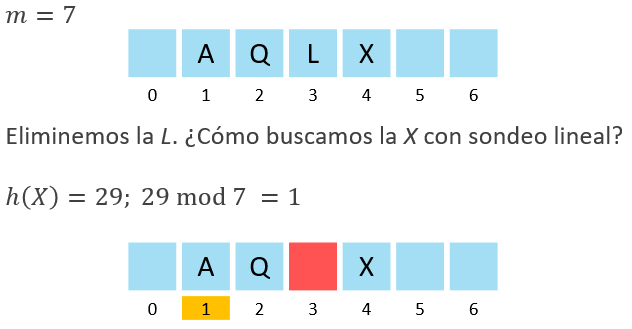
Colisiones

Al momento de asignar una llave a un objeto, es posible que dicho índice haya sido asignado con anterioridad, por lo que hay que manejar esta situación acorde a lo más conveniente. Las maneras típicas y vistas en clases son las siguientes:

1. **Direccionamiento abierto**: Busca otra celda disponible acorde a alguna regla en particular, por ejemplo:

* *Sondeo Lineal*: busca en *h*, *h + 1*, *h + 2*, y así hasta encontrar una disponible
* *Sondeo cuadrático*: busca en *h*, *h + 1c1 + 12c2*, *h + 2c1 + 22c2*, y así hasta encontrar una disponible
* *Doble hash*: busca en *h1(k)*, *h1(k) + h2(k)*, *h1(k) + 2h2(k)*, y así hasta encontrar una disponible

Sin embargo, este direccionamiento tiene varios problemas. El primer caso es el de clustering, en donde los datos comienzan a acumularse en el mismo sector de la tabla. El segundo es que no necesariamente pasa por todas las celdas, y en el peor de los casos, puede que no termine. El último problema es al momento de eliminar datos, ya que no sabrá cómo buscar los datos insertados después del eliminado -> Si hay que eliminar, no usar direccionamiento abierto:



Esto se puede arreglar mediante el uso de ***flags*** que indican si hubo eliminación o no, pero no es recomendable ya que podría aumentar el tiempo de búsqueda notoriamente.

1. **Encadenamiento (o listas ligadas)**: Si cada celda es una lista que guarda *n* datos y la tabla es de tamaño *m*, se tendrá que la inserción es de O(1), mientras que la búsqueda de O(1+n/m), siendo *n/m* el **factor de carga** al que se le puede fijar un valor máximo para aumentar el tamaño de la tabla en caso de que sea necesario (***rehashing***), debido a que mientras más datos se tengan, más probable es tener colisiones.

## Grafos

Introducción

Un **grafo *G*** es un conjunto de **nodos o vértices *V*** y un conjunto de **aristas *E*** que unen pares de nodos, en donde cada arista puede tener un **costo o capacidad** asociada. Existen tres problemas típicos en grafos con costos:

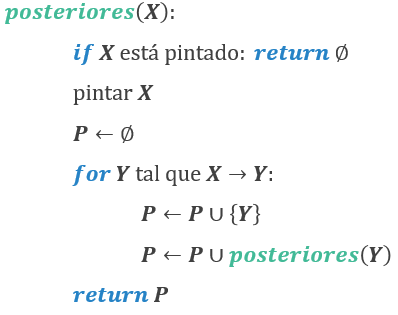
1. Encontrar el árbol de cobertura de costo mínimo en grafos no direccionales
2. Encontrar la ruta más corta desde un vértice a todos los otros en grafos direccionales
3. Entrar las rutas más cortas entre todos los pares de vértices en grafos direccionales

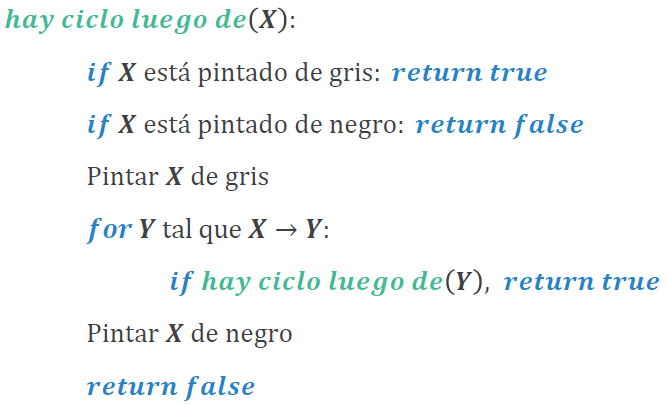
En los problemas de rutas más cortas las rutas deben ser direccionales, puede que existan vértices inalcanzables desde el vértice de partida, los costos pueden ser negativos, y las rutas más cortas pueden no ser únicas. Además, una ruta más corta cumple dos propiedades:

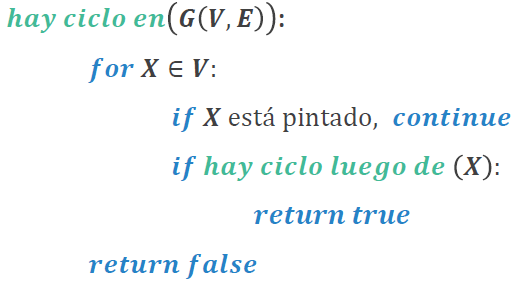
1. **Subestructura óptima**: Todas las subrutas en una ruta más corta *p* entre dos vértices, son también rutas más cortas
2. **Desigualdad triangular**: Si *c(u, z)* es el costo de la ruta más corta de *u* a *z*, y esta puede descomponerse en una ruta de *(u, v)* y *(v, z)*, entonces *c(u, w) = c(u, v) + w(v, z)*.

Ciclos

Por otra parte, hay veces en que los grafos direccionales pueden tener ciclos, por lo que podrían presentarse problemas sin solución producto a la reiteración de tareas cuando solo debiesen ocurrir una vez, o bien, se forman contradicciones por lo que resulta imposible ordenar las tareas. No obstante, la función “posteriores” permite saber cuáles son todos los nodos alcanzables a partir de uno, por lo que es posible identificar si una tarea es posterior a sí misma (definición de ciclo).

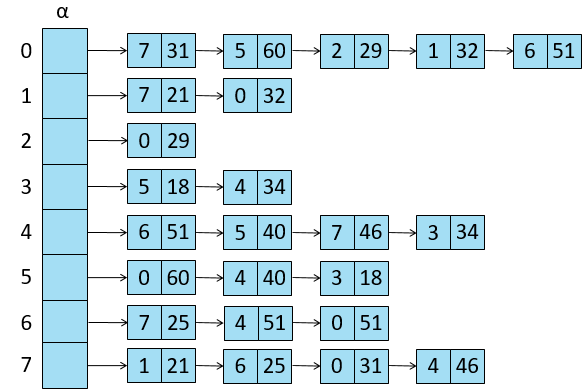
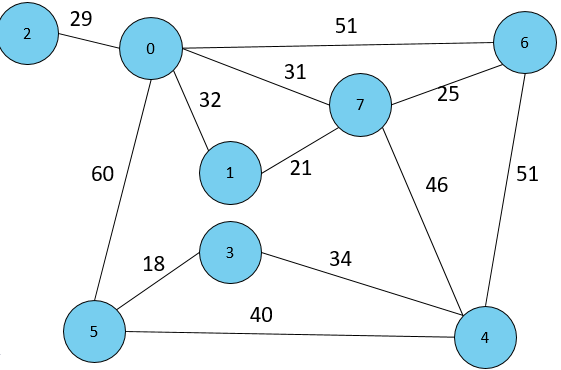
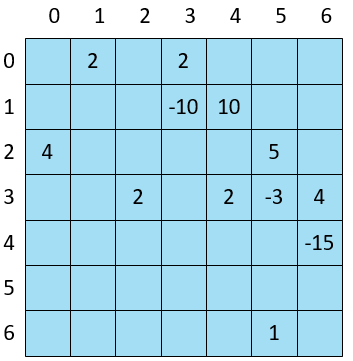
Si un nodo recién descubierto (*Y*) está pintado, entonces hay 2 posibilidades:

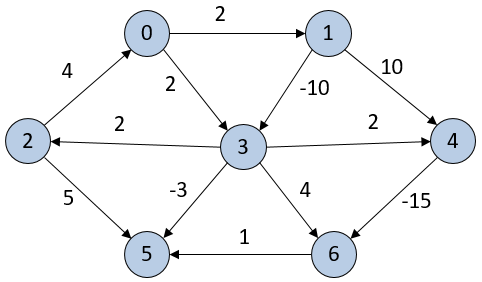
1. Si lo descubrió un nodo posterior a *Y*, hay ciclo.
2. Si lo descubrió un nodo anterior a *Y*, no hay ciclo (aún).



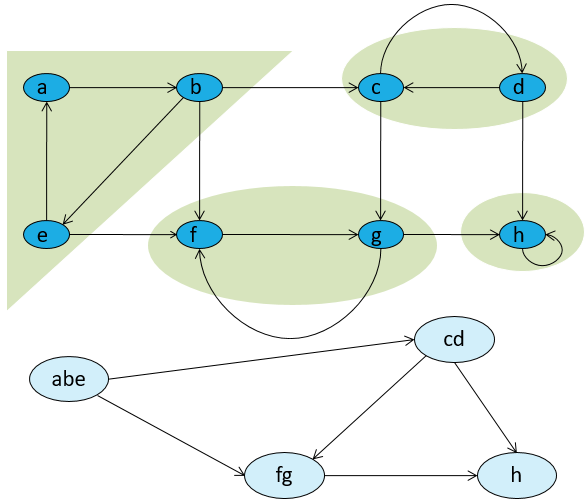
Representación en memoria

Hay dos formas principales de representar un grafo:

1. **Listas de adyacencias**: Cada nodo tiene una lista de los nodos a los que tiene una arista. El siguiente ejemplo comprende un grafo no dirigido:
2. **Matriz de adyacencias**: La coordenada (*x, y*) de la matriz indica si la arista (*x, y*) está en el grafo. El siguiente ejemplo es para un grafo dirigido:



SCCs

Las **componentes fuertemente conectadas (SCC’s)** de un grafo direccional son conjuntos de vértices tales que para todo par de vértices *u* y *v* son mutuamente alcanzables, es decir, existe un camino de *u* a *v*, y viceversa. De lo anterior, se desprende que *G* y *GT* tienen las mismas SCC’s, y que *GSCC* cumple que es un DAG:

Algoritmo para encontrar SCCs de un grafo *G (****Kosaraju****)*:

1. Realizar DFS de *G* para calcular tiempos de finalización de c/vértice
2. Determinar *GT*
3. Realizar DFS de G*T*, pero en el ciclo principal consideramos los vértices en orden decreciente en cuanto al tiempo de finalización
4. Los vértices de c/árbol en el bosque DFS recién formado serán SCCs diferentes

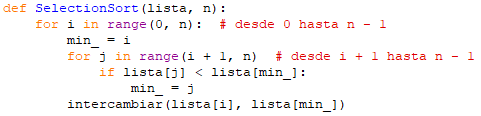
La complejidad de este algoritmo es O(|V| + |E|) al usar una lista de adyacencias (por DFS).

*\*Notar que una propiedad de los SCCs y tiempos de finalización fs, si hay una arista (u, v) entre dos componentes de una misma SCC tal que f(u) > f(v), entonces en el grafo GT ocurrirá que f(u) < f(v)*

# Algoritmos de ordenamiento

## Selection Sort

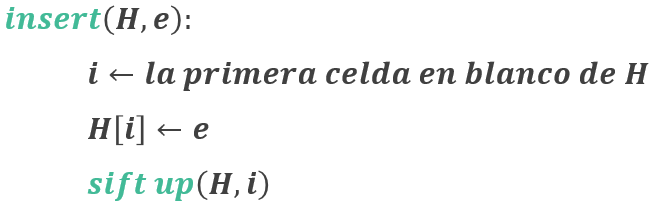
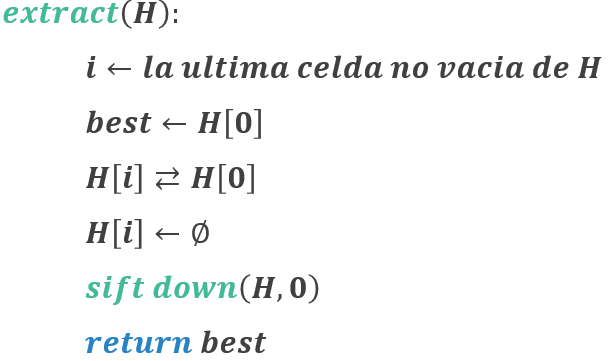
Es un algoritmo inestable que se enfoca en buscar el mínimo elemento entre una posición *i* y el final de un arreglo para intercambiarlo con el elemento de la posición *i*:



Su complejidad es de O(n2) para todos los casos ya que la cantidad de comparaciones se reduce en 1 para cada iteración, por lo que para arreglos muy grandes suele ser bastante ineficiente.

Sin embargo, permite implementar manejo de memoria *in – place* O(n), por lo que puede ser útil cuando la memoria auxiliar es limitada.

## Heap Sort

Es un algoritmo inestable similar a *Selection Sort* (puede ser pensado como su versión mejorada) ya que divide el *input* en una región ordenada y otra no, así va agarrando el elemento de mayor/menor prioridad de la zona no ordenada para insertarlo en la ordenada. Este algoritmo es mejor ya que implementa un *heap* en vez de un arreglo que toma tiempo lineal, y es el siguiente:

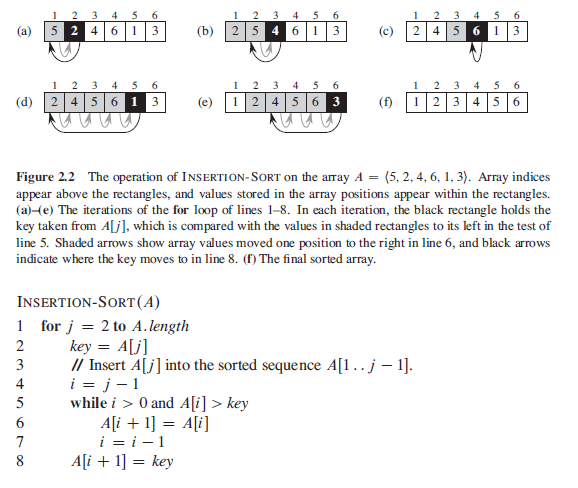
El primer paso es armar un *heap* a partir de los datos y formar un arreglo (recordar la forma de encontrar a los hijos y padre en base a *i*), esta acción se conoce como ***heapify***. Luego, se va creando un arreglo ordenado a partir de la reiterada extracción del nodo raíz del *heap* para insertarlo en el arreglo ordenado. Después de cada extracción el *heap* se actualiza para mantener su propiedad.

Armar un *heap* toma O(n), realizar *siftdown()* es O(log n) y es llamado *n* veces. Luego, la complejidad de este algoritmo es O(n + nlog n) = O(nlog n).

En cuanto a la memoria, este algoritmo puede ser llevado a cabo en formato *in – place* ya que el arreglo original puede ser dividido en dos (la parte ordenada y la no).

## Insertion Sort

Es un algoritmo estable que toma el elemento *i* del arreglo y busca su ubicación dentro de lo que lleva ordenado:

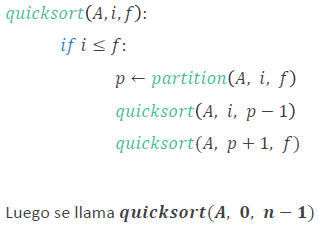


Su complejidad en el peor de los casos es O(n2) cuando está pseudo ordenado al revés, mientras que su mejor caso es O(n) cuando el arreglo viene pseudo ordenado, por lo que puede ser útil en etapas finales de ordenación.

En cuanto a la memoria, no se requiere de memoria adicional por lo que se considera *in – place* O(n).

## Quick Sort

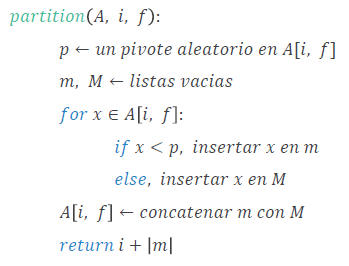
Es un algoritmo de ordenamiento inestable muy popular ya que realiza la menor cantidad de operaciones/comparaciones, causando que sea de los más eficientes en cuanto a tiempo:



Debido al funcionamiento de *partition*, se tiene que *Quick Sort* en su peor caso es O(n2), pero en su caso promedio y mejor caso tiene complejidad O(nlog n) (O(log n) por uso de “búsqueda binaria” y n veces eso por cada pivote implementado por *partition*).

Respecto a su manejo de memoria, al estar realizando intercambios en el mismo arreglo se considera que opera de manera *in – place*.

Partition

Quick Sort implementa un algoritmo llamado ***partition*** que se encarga de particionar un conjunto en dos que no se intersecten con el fin de incrementar la velocidad de procesamiento de conjuntos muy grandes, ya sea en una (línea) o dos (rectángulo) coordenadas (seguramente para más también).

A: arreglo

p: pivote

i: posición inicial

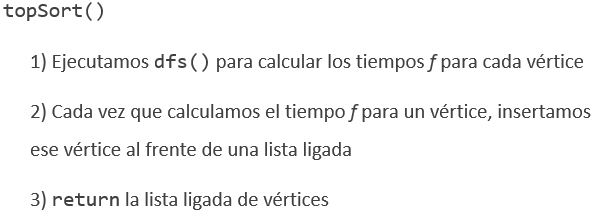
f: posición final

x: elemento entre *i* y *f*

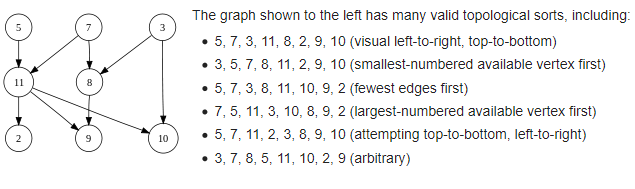
Lo que hace este algoritmo es fijar un pivote *p* aleatorio del arreglo inicial *A* para iterar desde *i* hasta *f* e insertar todo elemento menor a *p* en un conjunto de menores *m*, y todo elemento mayor en otro conjunto *M* para redefinir el arreglo inicial como *m* + *M*, y retornar la posición del pivote en el arreglo. Por lo tanto, se tiene que es un algoritmo cuya complejidad es O(n).

## Top Sort

Un grafo *G* puede ordenarse topológicamente si es **direccional acíclico (DAG)**, es decir si *G* es un DAG, entonces puede ordenarse linealmente acorde a todos los vértices de tal forma en que si *G* contiene una arista (*u, v*), entonces *u* aparece antes que *v* en la ordenación. El algoritmo es el siguiente:



La complejidad de éste algoritmo es O(|V|+|E|) por DFS, y se recomienda implementar el **algoritmo de** **Khan** que busca partir por los nodos que no tienen aristas de entrada.



En donde la “arbitraria” sería el caso típico desde mi perspectiva.

## Counting Sort

No es posible ordenar más rápido que O(n log(n)) si la única información usada por el algoritmo es el resultado de comparar, repetidamente, dos datos para determinar cuál es mayor. Al contrario, este es un algoritmo de ordenación estable que no compara los datos que está ordenando, ya que supone que cada uno de los *n* datos es un entero en el rango 0 a *k*, con *k* entero, por lo que si *k* es *O(n)*, entonces *counting sort* corre en tiempo *O(n)*.

Lo que se hace es determinar, para cada dato *x*, el número de datos menores que *x* para ubicar a *x* directamente en su posición final en el arreglo de salida, pero hay que manejar el caso en que varios datos tengan el mismo valor.

Pasos:

1. Guardar en la posición *i* la cantidad de veces que *i* se repite
2. Guardar en la posición *i* la cantidad de elementos menores o iguales al valor guardado en la posición de *i*
3. Insertar el valor *i* en la posición acorde al paso 2 y disminuir dicha posición en 1 para que los repetidos se posiciones correctamente

Es un algoritmo bastante útil cuando se quiere ordenar enteros cuyo rango es pequeño.

## Radix Sort

Es un algoritmo de ordenación tanto de enteros como de strings basándose en las posiciones/dígitos de los elementos a ordenar para realizar comparaciones, por lo que requiere implementar algún algoritmo estable de ordenación para que funcione correctamente. Puede ordenar en base al elemento más o menos significativo. El algoritmo es el siguiente:



Si el arreglo *a* contiene *n* elementos de largo *d*, en donde cada sub – elemento puede tomar hasta *k* valores posibles, entonces *radixSort* puede tomar hasta *k* valores posibles, por lo que toma tiempo *O(d(n+k))* en ordenar los *n* elementos, ergo, si *d* es constante y *k=O(n)* entonces *radixSort* es *O(n)*. Por lo que es muy útil para ordenar elementos de un mismo largo.

LSD (Least Significant Digit)



MSD (Most Significant Digit) String Sort: String de largos diferentes

Se usa *countingSort* para ordenar los strings según el primer carácter, luego, recursivamente, se ordenan los subarreglos correspondientes a cada carácter (excluyendo el primero). Así como *quicksort*, *MSD StringSort* particiona el arreglo en subarreglos que pueden ser ordenados independientemente, pero lo particiona en un **subarreglo para cada posible valor del primer carácter**, en lugar de las dos particiones de *quicksort*.



Cuidados:

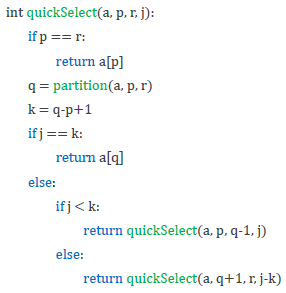
1. Fin del string: “she” es menor que “shells”

2. Alfabeto: binario (2), minúsculas (26), minúsculas + mayúsculas + dígitos (64), ASCII (128), Unicode (65536)

3. Subarreglos pequeños: tamaño < 11, cambiar a un *insertionSort* que sepa que los *p* primeros caracte res de los strings que está ordenando son iguales

# Algoritmos de Búsqueda

## Quick Select

Es otro algoritmo que implementa *partition* que es usado para encontrar el j – ésimo elemento más pequeño de una lista no ordenada:

a: arreglo

p: posición inicial

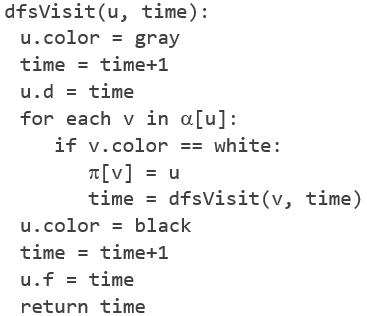
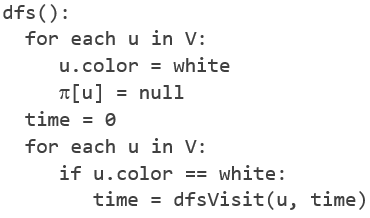
r: posición final

j: posición buscada

En cuanto a complejidad tiene peor caso O(n2), pero en promedio y mejor caso toma O(n). Un ejemplo típico sería encontrar la mediana.

## Depth First Search (DFS)

Algoritmo de búsqueda para grafos o árboles que inicia su búsqueda a partir de un nodo arbitrario (raíz generalmente) cuya regla es explorar aristas a partir del vértice descubierto más recientemente que aún no tiene aristas exploradas que salen de él de manera recursiva, y así hasta encontrar el elemento buscado o que no existan más nodos por revisar. Sirve para evaluación de proyectos.

Este algoritmo es completo siempre y cuando exista la detección de ciclos (pintar), ya que encuentra la solución óptima en caso de que exista.

*Alpha[u]*: nodos alcanzables por *u*

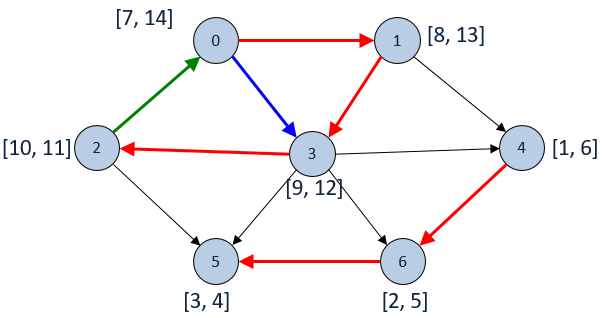
*Pi[u]*: nodo “padre” (primero en expandirlo) de *u*

*u.d*: tiempo en que *u* es descubierto por primera vez

*u.f*: tiempo en que la lista de adyacencias de *u* ha sido examinada completamente

Procedimiento:

1. Todos los vértices son inicialmente blancos
2. Un vértice se pinta gris cuando es descubierto
3. Un vértice se pinta negro cuando su lista de adyacencias ha sido examinada completamente

Notar que los intervalos a ([*u.d, u.f*]) y b ([*v.d, v.f*]) son disjuntos y ni *u* ni *v* es descendiente del otro en el DFS, por lo que, si a está contenido dentro de b, entonces *u* es descendiente de *v*, y viceversa.

*\*DFS a partir del vértice 4 y luego del 0*

Se definen 4 tipos de aristas:

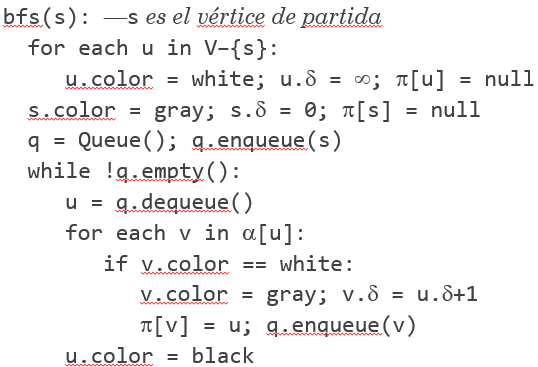
1. **De árbol**: Si *v* fue descubierta por primera vez al explorar (*u, v*)
2. **Hacia atrás**: Si *u* conecta a un ancestro de *v*
3. **Hacia adelante**: Si *u* conecta a un descendiente de *v* (solo ocurre en direccionales)
4. **Cruzadas**: Toda otra arista (solo ocurre en direccionales)

De lo anterior se deduce que un grafo direccional *G* es acíclico si y solo si DFS no puede producir aristas hacia atrás.

La complejidad de este algoritmo al implementar listas de adyacencias en el peor caso es O(|V|+|E|) (sería O(|V|2+|E|) usando matrices de adyacencias), mientras que su uso en memoria sería de O(|V|). Notar que O(|V|) puede variar entre O(1) y O(|V|2).

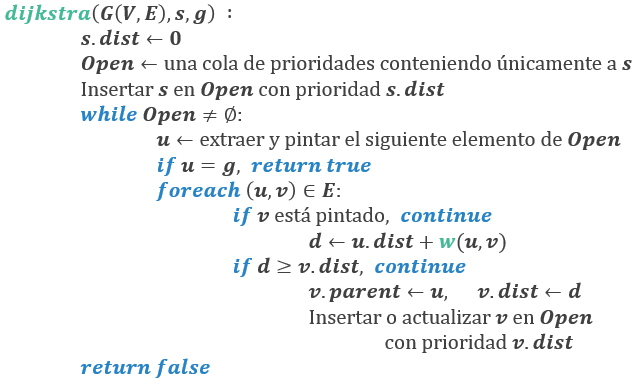
## Breadth First Search (BFS)

Algoritmo de búsqueda para grafos o árboles que inicia su búsqueda a partir de un nodo arbitrario (raíz generalmente) cuya regla es revisar el nivel completo de profundidad actual antes de ir al siguiente mediante una cola (FIFO), y así hasta encontrar el elemento buscado o que no existan más nodos por revisar. Sirve para problemas de estados y transiciones como el de la ruta más corta.

Este algoritmo es completo ya que encuentra la solución óptima en caso de que exista, y es el siguiente:

Notar que este algoritmo es similar a DFS solo que en vez de usar un *stack* se usa una *queue*, en otras palabras, si usáramos un *stack* se podría decir que BFS es una forma de implementar DFS iterativamente. Es por lo anterior que la complejidad es la misma: en el peor caso es O(|V| + |E|), mientras que su uso en memoria sería de O(|V|).

Dijkstra

Es un algoritmo para encontrar los caminos más cortos entre nodos en un grafo direccional con costos, pero limitado ya que es codicioso porque no sirve con costos negativos, por lo que puede no producir soluciones óptimas, pero en caso de que no existan costos negativos, si las produce. El algoritmo tiene complejidad O((V+E) log V) y es el siguiente: