Estructura de Datos y Algoritmos.

Listas:

En ciencias de la computación, una lista enlazada es una de las estructuras de datos fundamentales, y puede ser usada para implementar otras estructuras de datos. Consiste en una secuencia de nodos, en los que se guardan campos de datos arbitrarios y una o dos referencias, enlaces o punteros al nodo anterior o posterior. El principal beneficio de las listas enlazadas respecto a los vectores convencionales es que el orden de los elementos enlazados puede ser diferente al orden de almacenamiento en la memoria o el disco, permitiendo que el orden de recorrido de la lista sea diferente al de almacenamiento.

Una lista enlazada es un tipo de dato auto-referenciado porque contienen un puntero o enlace (en inglés link, del mismo significado) a otro dato del mismo tipo. Las listas enlazadas permiten inserciones y eliminación de nodos en cualquier punto de la lista en tiempo constante (suponiendo que dicho punto está previamente identificado o localizado), pero no permiten un acceso aleatorio. Existen diferentes tipos de listas enlazadas: listas enlazadas simples, listas doblemente enlazadas, listas enlazadas circulares y listas enlazadas doblemente circulares.

Listas lineales:

1) Lista simplemente enlazada.
2) Lista doblemente enlazada.

Listas circulares:

3) Lista simplemente enlazada circular.
4) Lista doblemente enlazada circular.

Otras listas:

5) Lista skip.
6) Lista doblemente enlazada, XOR Linked.
7) Lista enlazada desenrollada.

Operaciones básicas:

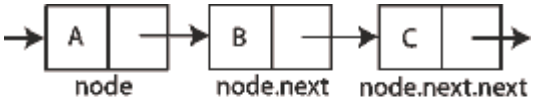
- Agregar al inicio/final.
- Insertar un elemento en una posición dada.
- Eliminar un elemento.
- Ordenar
- Etc.

Nodos centinelas

A veces las listas enlazadas tienen un nodo centinela (también llamado falso nodo o nodo ficticio) al principio o al final de la lista, el cual no es usado para guardar datos. Su propósito es simplificar o agilizar algunas operaciones, asegurando que cualquier nodo tiene otro anterior o posterior, y que toda la lista (incluso alguna que no contenga datos) siempre tenga un “primer y último” nodo.

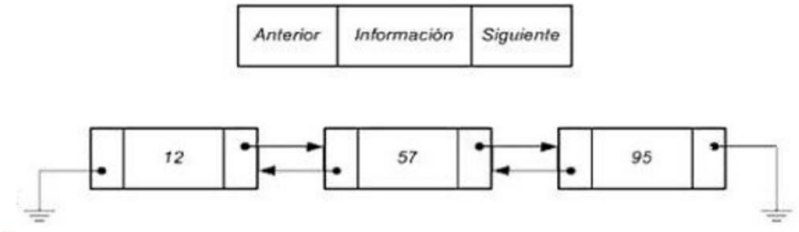
1- Listas simplemente enlazadas:

Es una lista enlazada de nodos, donde cada nodo tiene un único campo de enlace. Una variable de referencia contiene una referencia al primer nodo, cada nodo (excepto el último) enlaza con el nodo siguiente, y el enlace del último nodo contiene NULL para indicar el final de la lista. Aunque normalmente a la variable de referencia se la suele llamar top, se le podría llamar como se desee.



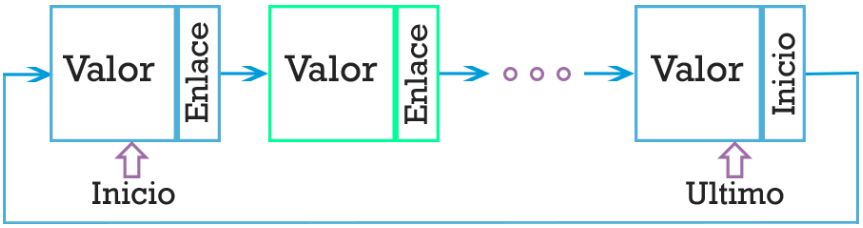
2- Lista doblemente

Un tipo de lista enlazada r da o lista enlazadas de dos vías. Cada nodo tiene apunta al valor NULL si es el primer nodo; y otro que apunta al nodo siguiente, o apunta al valor NULL si es el último nodo.



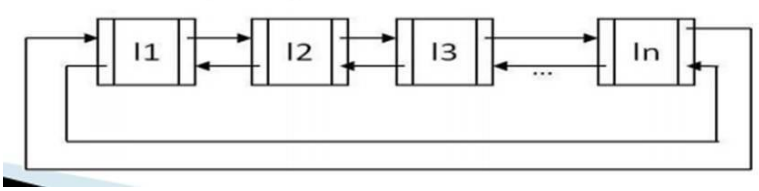
3- Lista simplemente enlazada circular:

Cada nodo tiene un enlace, similar al de las listas enlazadas simples, excepto que el siguiente nodo del último apunta al primero. Como en una lista enlazada simple, los nuevos nodos pueden ser solo eficientemente insertados después de uno que ya tengamos referenciado. Por esta razón, es usual quedarse con una referencia solamente al último elemento en una lista enlazada circular simple, esto nos permite rápidas inserciones al principio, y también permite accesos al primer nodo desde el puntero del último nodo.



4- Lista doblemente enlazada circular:

En una lista enlazada doblemente circular, cada nodo tiene dos enlaces, similares a los de la lista doblemente enlazada, excepto que el enlace anterior del primer nodo apunta al último y el enlace siguiente del último nodo, apunta al primero. Como en una lista doblemente enlazada, las inserciones y eliminaciones pueden ser hechas desde cualquier punto con acceso a algún nodo cercano. Aunque estructuralmente una lista circular doblemente enlazada no tiene ni principio ni fin, un puntero de acceso externo puede establecer el nodo apuntado que está en la cabeza o al nodo cola, y así mantener el orden tan bien como en una lista doblemente enlazada.

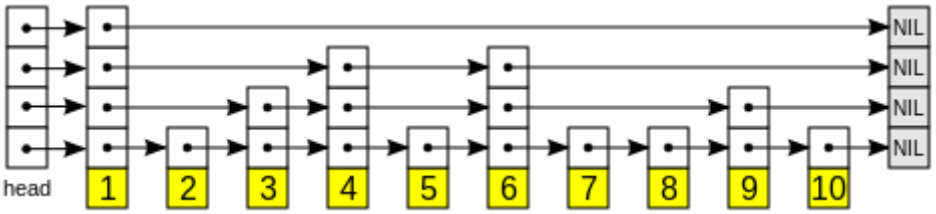


5- Skip List:

Una lista por saltos se construye por capas. La capa del fondo (la más baja) es una sencilla lista enlazada. Un elemento de la capa i aparece en la capa i+1 con una probabilidad fija p. En promedio, cada elemento aparece en 1/(1-p) listas, el elemento más alto (generalmente un elemento inicial colocado al principio de la lista por saltos) aparece en O(log(1/p) n) listas.

Para buscar un elemento se empieza con el elemento inicial de la lista de la capa más alta, hasta alcanzar el máximo elemento que es menor o igual al buscado. Luego se pasa a la capa siguiente (debajo de la actual) y se continúa la búsqueda. Se puede verificar que el número esperado de pasos en cada lista enlazada es 1/p. De manera que el costo total de búsqueda es O(log(1/p) n / p), que es lo mismo que O(log n) cuando p es una constante. Dependiendo del valor escogido para p, se puede favorecer el costo de búsqueda contra el costo de almacenamiento.

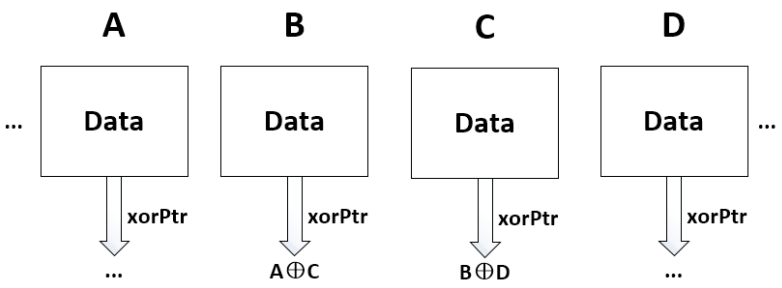
Las operaciones de inserción y borrado se implementan como las de sus correspondientes listas enlazadas, salvo que los elementos de las capas superiores deben ser insertados o borrados de más de una lista enlazada.



6- Lista doblemente enlazada, XOR Link:

Este tipo de lista, es una manera más “eficiente” de utilizar la estructura de datos de lista doblemente enlazada, ya que solo utilizaremos un puntero para hacer referencia al nodo siguiente y previo. La manera de acceder al nodo siguiente o al anterior es utilizando la lógica del operador XOR (O exclusivo) que son:

- $X \oplus X = 0$
- $X \oplus 0 = X$
- $X \oplus Y = Y \oplus X$
- $(X \oplus Y) \oplus Z = X \oplus (Y \oplus Z)$



7- Lista enlazada desenrollada:

Una lista enlazada desenrollada es una lista enlazada cuyos nodos contiene un vector de datos. Esto mejora la ejecución de la caché, siempre que las listas de elementos estén contiguas en memoria, y reducen la sobrecarga de la memoria, porque necesitas menos metadatos para guardar cada elemento de la lista. **IMPLEMENTACIONES EN ANEXO**

Pilas:

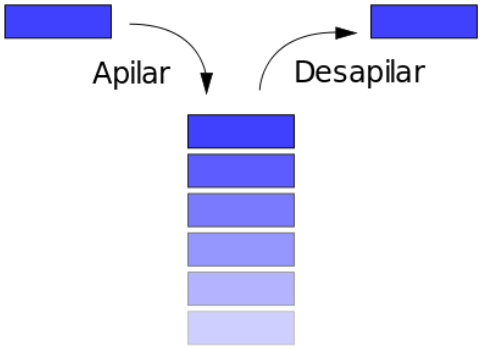
Una pila (stack en inglés) es una lista ordenada o estructura de datos que permite almacenar y recuperar datos, el modo de acceso a sus elementos es de tipo LIFO (del inglés Last In, First Out, «último en entrar, primero en salir»). Esta estructura se aplica en multitud de supuestos en el área de informática debido a su simplicidad y capacidad de dar respuesta a numerosos procesos.

Para el manejo de los datos cuenta con dos operaciones básicas: apilar (push), que coloca un objeto en la pila, y su operación inversa, retirar (o desapilar, pop), que retira el último elemento apilado.

En cada momento sólo se tiene acceso a la parte superior de la pila, es decir, al último objeto apilado (denominado TOS, Top of Stack en inglés). La operación retirar permite la obtención de este elemento, que es retirado de la pila permitiendo el acceso al anterior (apilado con anterioridad), que pasa a ser el último, el nuevo TOS.

Las pilas suelen emplearse en los siguientes contextos:

- Evaluación de expresiones en notación postfija (notación polaca inversa).
- Reconocedores sintácticos de lenguajes independientes del contexto.
- Implementación de recursividad.
- En un sistema operativo cada proceso tiene un espacio de memoria (pila) para almacenar valores y llamadas a funciones.
- Una pila acotada es una pila limitada a un tamaño máximo impuesto en su especificación.



Operaciones básicas:

- **apilar (*push)***: que coloca un objeto en la pila.
- **retirar**(o desapilar, ***pop***), que retira el último elemento apilado.

Las pilas se pueden implementar de distintas formas:

1) Con un arreglo fijo.
2) Con arreglo dinámico.
3) Con listas enlazadas.

Ver implementaciones en anexo.

Colas:

Una cola (también llamada fila) es una estructura de datos, caracterizada por ser una secuencia de elementos en la que la operación de inserción push se realiza por un extremo y la operación de extracción pop por el otro. También se le llama estructura FIFO (del inglés First In First Out), debido a que el primer elemento en entrar será también el primero en salir.

Operaciones básicas:

- **Crear**: se crea la cola vacía.
- **Encolar**: se añade un elemento a la cola. Se añade al final de esta.
- **Desencolar**: (sacar, salir, eliminar): se elimina el elemento frontal de la cola, es decir, el primer elemento que entró.
- **Frente**: se devuelve el elemento frontal de la cola, es decir, el primer elemento que entró.

Tipos de colas:

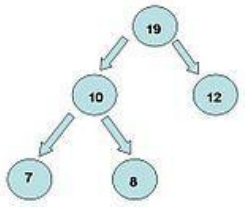
- **Colas simples**.
- **Colas circulares (anillos)**: en las que el último elemento y el primero están unidos.
- **Colas de prioridad**: los elementos se atienden en el orden indicado por una prioridad asociada a cada uno.
- **Bi-colas** (o Colas doblemente terminadas): son colas en donde los nodos se pueden añadir y quitar por ambos extremos; se les llama DEQUE (Double Ended QUEue). Para representar las bicolas lo podemos hacer con un array circular con Inicio y Fin que apunten a cada uno de los extremos. Hay variantes:

  Bi-colas de entrada restringida: Son aquellas donde la inserción sólo se hace por el final, aunque podemos eliminar al inicio ó al final.

  Bi-colas de salida restringida: Son aquellas donde sólo se elimina por el final, aunque se puede insertar al inicio y al final.

IMPLEMENTACIONES EN ANEXO

Heaps:



En computación, un montículo (o he ructura de datos del tipo árbol con información perteneciente a un con ntículos **máximos** tienen la

característica de que cada nodo padre tiene un valor mayor que el de cualquiera de sus nodos hijos, mientras que en los montículos **mínimos**, el valor del nodo padre es siempre menor al de sus nodos hijos.

Un árbol cumple la condición de montículo si satisface dicha condición y además es un árbol binario casi completo. Un árbol binario es completo cuando todos los niveles están llenos, con la excepción del último, que se llena desde la izquierda hacia la derecha.

En un montículo de prioridad, el mayor elemento (o el menor, dependiendo de la relación de orden escogida) está siempre en el nodo raíz. Por esta razón, los montículos son útiles para implementar colas de prioridad. Una ventaja que poseen los montículos es que, por ser árboles completos, se pueden implementar usando arreglos (arrays), lo cual simplifica su codificación y libera al programador del uso de punteros.

Operaciones comunes:

- Insertar
- Eliminar

Tipos de heaps:

1) Binario
2) Binomico
3) De Fibonacci
4) Suave
5) 2-3

IMPLEMENTACIONES EN ANEXO

Arboles:

En ciencias de la computación y en informática, un árbol es un tipo abstracto de datos (TAD) ampliamente usado que imita la estructura jerárquica de un árbol, con un valor en la raíz y subárboles con un nodo padre, representado como un conjunto de nodos enlazados.

Operaciones comunes:

- Enumerar todos los elementos
- Enumerar la sección de un árbol
- Buscar un elemento
- Añadir un nuevo elemento en una determinada posición del árbol
- Borrar un elemento
- Podar: Borrar una sección entera de un árbol
- Injertar: Añadir una sección entera a un árbol
- Buscar la raíz de algún nodo
- Representar cada nodo como una variable en el montículo con punteros.
- Representar el árbol con un vector

Recorridos de árboles:

El recorrido de árboles se refiere al proceso de visitar de una manera sistemática, exactamente una vez, cada nodo en una estructura de datos de árbol (examinando y/o actualizando los

datos en los nodos). Tales recorridos están clasificados por el orden en el cual son visitados los nodos. Los siguientes algoritmos son descritos para un árbol binario, pero también pueden ser generalizados a otros árboles.

Árbol binario

Pre-orden: (raíz, izquierdo, derecho).

In-orden: (izquierdo, raíz, derecho).

Post-orden: (izquierdo, derecho, raíz).

Árbol genérico

Para recorrer un árbol no vacío en orden de profundidad-primero, hay que realizar las siguientes operaciones recursivamente en cada nodo:

6) Realice la operación pre-orden
7) Para i=1 a n-1 haga:
   Visite al hijo[i], si existe
   Realice la operación in-orden
8) Visite al hijo[n], si existe
9) Realice la operación post-orden

Dependiendo del problema actual, las operaciones de pre-orden, in-orden o post-orden pueden ser vacías, o usted puede querer visitar solamente un nodo de hijo específico, así que estas operaciones pueden ser consideradas opcionales. Por ejemplo, al insertar en un árbol ternario, una operación de pre-orden es realizada comparando elementos. Una operación de post-orden puede luego ser necesitada para re-balancear el árbol.

Recorrido en anchura-primero

Los árboles también pueden ser recorridos en orden por nivel (de nivel en nivel), donde visitamos cada nodo en un nivel antes de ir a un nivel inferior. Esto también es llamado recorrido en anchura-primero o recorrido en anchura.

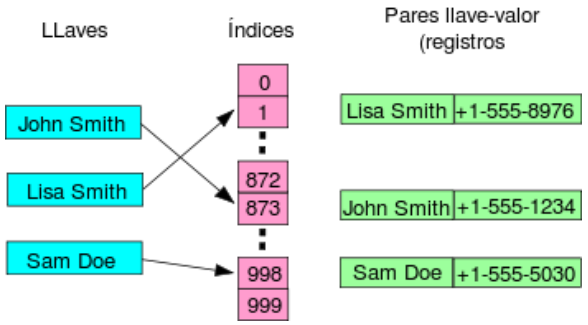
Tipos de árboles:

1) Binario.
2) De búsqueda binaria.
3) Multi-camino
4) Biselado – Splay.
5) Rojo – Negro.
6) B / B* / B+
7) 2-3
8) AVL
9) Binario de búsqueda aleatorio.
10) De segmento
11) AA

IMPLEMENTACIONES EN ANEXO

Tabla Hash:

Una tabla hash es una estructura de datos que asocia llaves o claves con valores. La operación principal que soporta de manera eficiente es la búsqueda: permite el acceso a los elementos (teléfono y dirección, por ejemplo) almacenados a partir de una clave generada (usando el nombre o número de cuenta, por ejemplo). Funciona transformando la clave con una función hash en un hash, un número que identifica la posición (casilla o cubeta) donde la tabla hash localiza el valor deseado.



Las operaciones básicas implementadas en las tablas hash son:

Inserción

- La forma de implementar en función esta operación es pidiendo la llave y el valor, para con estos poder hacer la inserción del dato.

- Para almacenar un elemento en la tabla hash se ha de convertir su clave a un número. Esto se consigue aplicando la función resumen (hash) a la clave del elemento.

- El resultado de la función resumen ha de mapearse al espacio de direcciones del vector que se emplea como soporte, lo cual se consigue con la función módulo. Tras este paso se obtiene un índice válido para la tabla.
- El elemento se almacena en la posición de la tabla obtenido en el paso anterior.

- Si en la posición de la tabla ya había otro elemento, se ha producido una colisión. Este problema se puede solucionar asociando una lista a cada posición de la tabla, aplicando otra función o buscando el siguiente elemento libre. Estas posibilidades han de considerarse a la hora de recuperar los datos.

Búsqueda

- La forma de implementar en función esta operación es pidiendo la llave y con esta devolver el valor.
- Para recuperar los datos, es necesario únicamente conocer la clave del elemento, a la cual se le aplica la función resumen.
- El valor obtenido se mapea al espacio de direcciones de la tabla.
- Si el elemento existente en la posición indicada en el paso anterior tiene la misma clave que la empleada en la búsqueda, entonces es el deseado. Si la clave es distinta, se ha de buscar el elemento según la técnica empleada para resolver el problema de las colisiones al almacenar el elemento.

Eliminar o Borrar

La mayoría de las implementaciones también incluyen la función de borrar a la cual se le envía una llave que deberá eliminar. También se pueden ofrecer funciones como iteración en la tabla,

crecimiento y vaciado. Algunas tablas hash permiten almacenar múltiples valores bajo la misma clave.

Resolución de colisiones

Si dos llaves generan un hash apuntando al mismo índice, los registros correspondientes no pueden ser almacenados en la misma posición. En estos casos, cuando una casilla ya está ocupada, debemos encontrar otra ubicación donde almacenar el nuevo registro, y hacerlo de tal manera que podamos encontrarlo cuando se requiera.

Direccionamiento Cerrado, Encadenamiento separado o Hashing abierto

En la técnica más simple de encadenamiento, cada casilla en el array referencia una lista de los registros insertados que colisionan en la misma casilla. La inserción consiste en encontrar la casilla correcta y agregar al final de la lista correspondiente. El borrado consiste en buscar y quitar de la lista.

La técnica de encadenamiento tiene ventajas sobre direccionamiento abierto. Primero el borrado es simple y segundo el crecimiento de la tabla puede ser pospuesto durante mucho más tiempo dado que el rendimiento disminuye mucho más lentamente incluso cuando todas las casillas ya están ocupadas. De hecho, muchas tablas hash encadenadas pueden no requerir crecimiento nunca, dado que la degradación de rendimiento es lineal en la medida que se va llenando la tabla. Por ejemplo, una tabla hash encadenada con dos veces el número de elementos recomendados, será dos veces más lenta en promedio que la misma tabla a su capacidad recomendada.

Las tablas hash encadenadas heredan las desventajas de las listas ligadas. Cuando se almacenan cantidades de información pequeñas, el gasto extra de las listas ligadas puede ser significativo. También los viajes a través de las listas tienen un rendimiento de caché muy pobre.

Otras estructuras de datos pueden ser utilizadas para el encadenamiento en lugar de las listas ligadas. Al usar **árboles auto-balanceables**, por ejemplo, el tiempo teórico del peor de los casos disminuye de O(n) a O(log n). Sin embargo, dado que se supone que cada lista debe ser pequeña, esta estrategia es normalmente ineficiente a menos que la tabla hash sea diseñada para correr a máxima capacidad o existan índices de colisión particularmente grandes. También se pueden utilizar **vectores dinámicos** para disminuir el espacio extra requerido y mejorar el rendimiento del caché cuando los registros son pequeños.

Direccionamiento abierto o Hashing cerrado

Las tablas hash de direccionamiento abierto pueden almacenar los registros directamente en el array. Las colisiones se resuelven mediante un sondeo del array, en el que se buscan diferentes localidades del array (secuencia de sondeo) hasta que el registro es encontrado o se llega a una casilla vacía, indicando que no existe esa llave en la tabla.

Las secuencias de sondeo más utilizadas son:

1) **Sondeo lineal**: en el que el intervalo entre cada intento es constante (frecuentemente 1). El sondeo lineal ofrece el mejor rendimiento del caché, pero es más sensible al aglomeramiento.

2) **Sondeo cuadrático**: en el que el intervalo entre los intentos aumenta linealmente (por lo que los índices son descritos por una función cuadrática. El sondeo cuadrático se sitúa entre el sondeo lineal y el doble hasheo.
3) **Doble hasheo**: en el que el intervalo entre intentos es constante para cada registro pero es calculado por otra función hash. El doble hasheo tiene pobre rendimiento en el caché pero elimina el problema de aglomeramiento. Este puede requerir más cálculos que las otras formas de sondeo.

Una influencia crítica en el rendimiento de una tabla hash de direccionamiento abierto es el porcentaje de casillas usadas en el array. Conforme el array se acerca al 100% de su capacidad, el número de saltos requeridos por el sondeo puede aumentar considerablemente. Una vez que se llena la tabla, los algoritmos de sondeo pueden incluso caer en un círculo sin fin. Incluso utilizando buenas funciones hash, el límite aceptable de capacidad es normalmente 80%. Con funciones hash pobremente diseñadas el rendimiento puede degradarse incluso con poca información, al provocar aglomeramiento significativo.