Conceptos generales

¿Qué es un algoritmo?: Una serie finita de pasos ordenados, generales (resuelven un tipo de problema, no un problema específico) y definidos (inequívocos, sin ambigüedades).
¿Qué es un programa?: Algoritmo codificado en un determinado lenguaje de programación, que le permite ser interpretado por una computadora además de una persona y resolver un programa.
¿Qué es un proceso?: Es un programa en ejecución en este mismo momento, es decir, un programa que es ejecutado. Tiene un determinado estado del proceso: está dado por el valor que toman las variables en un momento determinado.
¿Qué es la abstracción? Observación sobre una entidad determinada con la que se trabaja, a fin de relevar los atributos importantes para la resolución del problema. Esos atributos se almacenan en la computadora y el programa puede trabajar con esa información, guardandolos en estructura de datos, que es una entidad que almacena toda la información recabada y necesaria para utilizar por el programa, en un proceso.
Archivo: una forma de darle perpetuidad a esos datos guardados en estructuras de datos, a fin de que estén disponibles en cualquier momento
Tipos de estructuras de datos:

- **Simples:** permiten guardar un único valor, del mismo tipo especificado.
- **Compuestos:** permiten guardar más de un valor, pudiendo ser un arreglo (sólo admite datos de un mismo tipo, con al menos un índice) o un struct (específico de C++, permite guardar más de un tipo de variable de datos). Otros son árbol (permite guardar datos que tengan una jerarquía entre sí) y grafo (permite guardar datos interconectados entre sí.

El orden de ejecución de sentencia es por medio de las estructuras de control:

- **De secuencia** (una debajo de otra).
- **De selección** (o condicional, con 3 alternativas: “if”, “if-else” y “case”, que permite seguir un camino de ejecución de acuerdo a una condición).
- **Iterativas** (o repetitivas, cuando se quiere indicar una o un conjunto de sentencias a ejecutar 2 o más veces, y son el “while”, que se puede o no ejecutar ninguna vez, el “do-while”, que al menos una vez se ejecutará y el “for”, que se va a ejecutar una cantidad discreta y conocida de veces). Estas se pueden “apilar” (una debajo de otras) o anidar (una dentro de otra).

El programa utiliza cosas de la memoria, guardadas en ella, y el SO se encarga de administrar todos los recursos. El programa hace un pedido al SO, el SO (con sus procesos) asigna y controla que el programa se mueva dentro de ese espacio.
Las asignaciones de memoria son:

- **Estáticas** (struct, arreglos, etc.) y permite saber la cantidad de memoria a usar previamente a ejecutarse, debiendo ser consecutiva; la asignación se realiza por consiguiente previamente a ejecutarse en la declaración y se devuelve automáticamente luego de usarse al SO; el problema es que el espacio de memoria puede contener basura y que las variables sean asignadas a ellas).
- **Dinámicas** (se puede pedir en tiempo de ejecución, también debiendo ser consecutiva por bloque, pero no en total; por consiguiente, no se puede saber cuánto se ocupará previamente a ejecutarse; se piden y devuelven de forma explícita en C++).

¿Qué es un registro?
La palabra clave **struct** indica que se trata de una definición de tipo de **estructura compuesto**, representa un valor compuesto por un número determinado de elementos, que pueden ser de distintos tipos (simples y compuestos). Para utilizar registros definiremos un nuevo tipo registro, enumerando los elementos (campos) que lo componen. Para cada campo deberemos indicar su tipo y el identificador con él nos referiremos al mismo.

La definición del tipo registro se hará utilizando la palabra reservada **struct**, seguido del **identificador** con el que haremos referencia a dicho tipo. A continuación, se enumeran, entre llaves, los campos o **miembros** que lo componen, especificando su tipo y el identificador con el que referenciar, seguido por el delimitador punto y coma (;). La llave de cierre debe ir seguida de punto y coma.

Un **valor de estructura** es una colección de valores más pequeños llamados **valores miembro** (un valor miembro por cada nombre de miembro declarado en la definición de la estructura), que puede contener valores igual que cualquier otra variable.
Se puede inicializar una estructura en el momento de declararla, colocando después de su nombre un signo de igual y una lista de los valores miembros encerrados en llaves. Por ejemplo:

```
struct Fecha {
        int dia;
        int mes;
        int anio;
};
Fecha vencimiento = {31, 12, 2004}
```

Con los valores inicializadores deben darse en el orden que corresponde al orden de las variables miembro en la definición de tipo de estructura:

- Asignación, para asignar un valor de tipo registro a una variable del mismo tipo registro; también es posible asignar un valor de tipo registro completo a una variable o campo, siempre que sea del mismo tipo.
- Paso como parámetro a subprogramas, donde podremos pasar registros como parámetros a subprogramas

```
struct Fecha {
        int dia;
        int mes;
        int anyo;
}


void leer_fecha (Fecha & f) {
        cin >> f.dia >> f.mes >> f.anyo ;
}
void escribir_fecha (const Fecha& f) {
        cout << f.dia << "/" << f.mes << "/" << f.anyo ;
}
```

¿Qué es un puntero?

- Un **puntero** es la dirección en memoria de una variable, son variables simples que pueden almacenar una dirección de memoria (en binario), y debe ser igual en tipo al destino (se indica el tipo seguido de un asterisco). Básicamente es un espacio de memoria de un determinado tipo que se reserva para guardar una dirección de memoria del mismo tipo (adonde “apunta”)

- Si una variable se implementa como tres posiciones de memoria, la dirección de la primera de esas posiciones a veces se usa como nombre para esa variable

- Al hacer uso del nombre de la variable se hace referencia al valor que tenga esa variable, ya sea para lectura o escritura (es decir, se accede a leer o escribir lo que haya en esa variable)

- Es importante remarcar que el tipo puntero, en sí mismo, es un tipo simple, aunque el tipo apuntado puede ser tanto un tipo simple, como un tipo compuesto

- Sirven para arreglos, ya que los arreglos son un tipo de punteros. Por ejemplo al crear un arreglo "a" enteró de 3 posiciones (a[]={1,2,3}), y crear un puntero también int p (int* p), se lo puede asignar al puntero, y para recorrerlo se puede hacer un ciclo for con i=0 e i<última posición del arreglo, imprimiendo por pantalla como "cout«*(p+i)"

- Los punteros del tipo void permiten apuntar a cualquier tipo de dato, pero no puede ser referenciado directamente (es decir, no permite el uso del operador * sobre ellos), debido a su longitud indeterminada; por tanto, se debe recurrir a la conversión entre tipos (**type casting**) o asignar ese puntero void a un puntero de un tipo específico

- Los punteros a funciones permiten pasar una función como parámetro a otra función, ya que no pueden pasarse por referencia

Asignación de Variables de Tipo Puntero:

- El operador & (**operador de dirección**) seguido del nombre de la variable hace referencia a la posición de memoria donde se encuentra esa variable, y es solo para la lectura

- Una asignación por ejemplo se realiza: int* puntero= &n, donde "puntero" es un puntero int, y "&n" hace referencia a una posición de memoria donde se encuentra la variable int "n". En caso de que se lea solo el nombre de puntero, se mostrará por pantalla el valor de la posición de memoria donde apunta, si se indica con el & delante de puntero se mostrará la posición de memoria que tiene el puntero

- Así mismo, a una variable de tipo puntero se le puede asignar el valor de otra variable puntero, y ambas variables de tipo puntero apuntarán a la misma variable dinámica, que será compartida por ambas. Si se libera la variable dinámica apuntada por una de ellas, la variable dinámica compartida se destruye, su memoria se desaloja y ambas variables locales de tipo puntero quedan con un valor no especificado (int *puntero 1, *puntero2; puntero1=new int; puntero2=puntero1; delete puntero1;)

- En la operación de asignación, el valor anterior que tuviese la variable de tipo puntero se pierde, por lo que habrá que tener especial cuidado de que no se pierda la variable dinámica que tuviese asignada, si tuviese alguna.

Desreferenciación de una Variable de Tipo Puntero:

- Anteponiendo el * (**operador de desreferenciación**) al nombre de la variable puntero, se des referencia y se coloca otro valor el variable donde esté apunta, modificando el valor que se halla en ella (por ejemplo, *puntero=9 le asigna el valor 9 a ese puntero).

- También si queremos declarar una variable que pueda contener apuntadores a otras variables de un tipo específico, declaramos las variables de apuntador igual que declaramos una variable ordinaria de ese tipo, pero colocamos un asterisco antes del nombre de la variable.

- Si se hace *p=1, se modifica la dirección donde apunta, y al escribir por pantalla “a”, modificará su valor y saldrá “1”.
- Sin embargo, si una variable de tipo puntero tiene el valor NULL, entonces des referenciar la variable produce un error en tiempo de ejecución que aborta la ejecución del programa. Así mismo, des referenciar un puntero con valor inespecificado produce un comportamiento anómalo en tiempo de ejecución. Es posible, así mismo, acceder a los elementos de la variable apuntada mediante el operador de desreferenciación (PPersona ptr = new Persona ; ptr->nombre = "pepe" ; ptr->teléfono = "111" ; ptr->edad = 5;).

Gestión de Memoria Dinámica:

- La asignación de memoria genera una dirección de memoria, guardada en un puntero, y será el único punto a ese bloque de memoria dinámica que nos asigne el SO.
- Cuando necesite crear una determinada variable dinámica, debe solicitar memoria dinámica con el operador **new** seguido por el tipo de la variable dinámica a crear, reservando espacio en memoria dinámica para albergar a la variable, y después creando (invocando al constructor especificado) el contenido de la variable dinámica. Produce una nueva variable sin nombre y devuelve un apuntador que apunta a esta nueva variable dinámica (porque se crean y se destruyen mientras el programa se está ejecutando).
- También se puede asignar a una variable de tipo puntero al tipo de la variable dinámica creada (int *p=new int).
- Ademas se debe liberar explícitamente dicha variable dinámica mediante el operador **delete**, primero destruyendo la variable dinámica (invocando a su destructor), y después desaloja (libera) la memoria dinámica reservada para dicha variable (si se ejecuta la operación delete sobre una variable de tipo puntero que tiene el valor NULL, entonces esta operación no hace nada).
- Se puede sumar o restar una cantidad entera a un puntero, donde la nueva dirección de memoria difiere en n (un Nº, por el tamaño del tipo apuntado por el puntero) cantidad de bytes y siempre apuntará a un puntero del mismo tipo.
- En caso de trabajar con una estructura dinámica se utiliza **new** + un tipo de dato permite generar una variable (entre corchetes el Nº de elementos), retornando un puntero que apunta al comienzo del nuevo bloque de memoria asignado durante ejecución, donde se puede acceder y operar de la misma forma que un arreglo común. En caso de trabajar en 2 dimensiones se hace:

  ```
  int **puntero=new int*[4];
  for (int i=0; i<4; i++) {
          puntero[i] = new int [3];
  }
  ```

- Cada elemento borrado debe acompañarse con **delete**, borrando ese puntero y devolviendo memoria (delete + puntero o delete puntero [] si ese puntero es un arreglo, donde previamente se borrará por medio de un ciclo for las otras dimensiones que tuviera). Por ejemplo, en caso de trabajar con 2 dimensiones:

  ```
  for (int i=0; i<4; i++) {
          delete puntero[i];
  }
  delete puntero [];
  ```

Comparación de Variables de Tipo Puntero:
Las variables del mismo tipo puntero se pueden comparar entre ellas por igualdad (==) o desigualdad (!=), para comprobar si apuntan a la misma variable dinámica. Así mismo, también se pueden comparar por igualdad o desigualdad con el puntero nulo (NULL) para saber si apunta a alguna variable dinámica, o por el contrario no apunta a nada (int *p1,*p2; if (p1 == p2) / if (p1 != NULL)).

Paso de Parámetros de Variables de Tipo Puntero:

- El tipo puntero es un tipo simple, y por lo tanto se tratará como tal. Por ejemplo:
  Void modificar (PPersona &p);
  Void buscar(PPersona p, const string& nombre);
- Al adicionar un puntero, apuntará al siguiente puntero del mismo tipo, no al siguiente byte. Por ejemplo: partiendo de que puntero apunte a un dato entero, (*puntero)++ apuntará al siguiente dato del mismo tipo, es decir, al siguiente entero
- Para que un puntero apunte a un vector por ejemplo: puntero=&v[0], donde el puntero apuntará a la primera posición del vector v; aun así no se recomiendan para recorrer arreglos los punteros

- La constante NULL es una constante especial de tipo puntero que indica que una determinada variable de tipo puntero no apunta a nada en la memoria dinámica (int *p=NULL;). Al asignarle el valor NULL no se tendrá una dirección de memoria (permitiendo por ejemplo compararlo de la forma que haya o no un valor NULL)

Casting de variables:
Permite generar un cambio de tipo de variables, devolviendo por esas variables un tipo que originalmente no podría, debido al tipo de las variables que intervienen en su formación, o bien por el tipo implícito que es. pudiendo ser **implícito** (según el tamaño del tipo de datos, por ejemplo:
float * int, o que generará un valor float) o **explícito** (int a,b,resultado; resultado=a*b float(resultado))

Recursividad

Es una técnica que permite definir una función en términos de sí misma. En otras palabras: una función es recursiva cuando se invoca a sí misma. Por lo tanto, está formado por funciones que se llaman a sí mismo

- Para su implementación los compiladores utilizan una pila (**stack**) de memoria temporal, la cual puede causar una interrupción del programa si se sobrepasa su capacidad (s**tack overflow**)

- Permite en algunos casos resolver elegantemente algoritmos complejos, pero los procedimientos son menos eficientes y performantes ya que consume demasiados recursos (sobre todo comparado con la programación iterativa)

- La **recursividad** puede ser directa o indirecta, donde una función se llama directamente a sí misma o bien la función llama a otra y esta a su vez a la primera

- Observar que en la función recursiva existe una condición (x==0) que permite abandonar el proceso recursivo cuando la expresión relacional arroje verdadero; de otro modo el proceso

- sería infinito. Se denomina **caso base**, y hay 2 alternativas: que la función se ejecute una sola vez porque es el caso base, o bien que la función se ejecute hasta el caso base, de forma lógicamente recursiva, donde en cada llamada se obtiene una función más simple
- El número máximo de llamadas anidadas se denomina **profundidad de recursión**, y debe ser finita y lo más pequeña posible, tanto para poder salir del ciclo como para no consumir en demasía los recursos del hardware

- Las funciones también tienen precondiciones, en la que los elementos pasado en los parámetros deben cumplirlas

Condiciones para que una función sea recursiva

1. Realizar llamadas a sí misma para efectuar versiones reducidas de la misma tarea.
2. Incluir uno o más casos donde la función realice su tarea sin emplear una llamada recursiva, permitiendo detener la secuencia de llamadas (condición de detención o stop)

Pila de llamada

- Las llamadas recursivas no se ejecutan indefinidamente, sino que se colocan en una pila hasta que la condición de término se halla (**caso base**). Así, se ejecutan las llamadas a la función en el camino inverso, como sacándose de la pila

- En caso de que la función tenga variables locales, se creará un conjunto diferente de variables locales, con el mismo nombre que en la declaración de la función, pero que representen distintos valores, que se almacenan en una pila. Luego, cuando el proceso se deshaga (las llamadas se sacan de la pila y siguen su ejecución) se podrá disponer de ellas

- En cada llamada de recursiva el compilador utiliza una nueva zona de la pila para almacenar las variables, con la consecuencia del enlentecimiento de la ejecución de la función y generando problemas por agotamiento de la memoria de la pila

Estructuras lineales

Pilas

- Modo de acceso a sus elementos: LIFO (Last input, first output)
- Para su manejo, se cuenta con dos operaciones básicas: enpilar (coloca elementos en la pila) y desempilar (retira el último elemento empilado)

Alta pila(dinámica):

1. Crear espacio en memoria para almacenar un nodo: **NPila *nuevo_nodo = new NPila ()**
2. Cargar el valor dentro del nodo: **nuevo_nodo->dato = n**
3. Cargar el puntero pila dentro del nodo: **nuevo_nodo->link = pila**
4. Asignar nuevo_nodo a pila: **pila = nuevo_nodo**

Baja pila(dinámica):

1. Crear una variable auxiliar: **NPila *aux = pila;**
2. Igualar n a aux->dato: **n = aux->dato;**
3. Pasar pila al nodo siguiente: **pila = aux->link;**
4. Eliminar variable: **delete aux;**

Colas:

- Estructuras FIFO (First input, First Output)
- Caracterizada por ser una secuencia de elementos en la que la operación de inserción (encolar) se realiza por un extremo, mientras que la extracción (desencolar) se realiza por el otro.

Alta cola:

1. Crear espacio en memoria para almacenar un nodo: **NCola *nuevo_nodo = new NCola()**
2. Cargar el valor dentro del nodo: **nuevo_nodo->dato = n**

3. Asigna los punteros frente y fondo hacia el nuevo nodo:

```
if(cola vacia(frente)){
        frente = nuevo_nodo;
}
else{
        fondo->link = nuevo_nodo;
}
fondo = nuevo_nodo;
}
```

Baja cola:

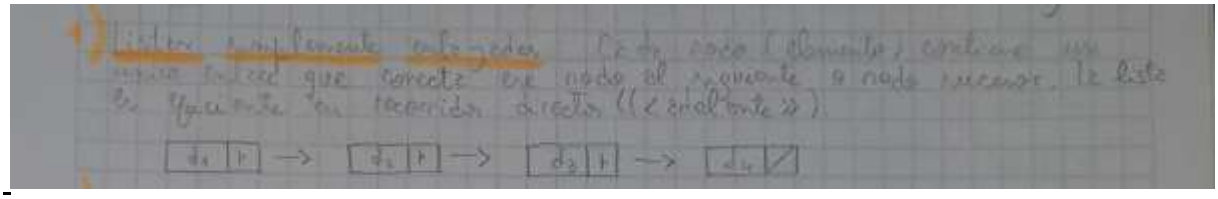
1. Obtener el valor del nodo **n = frente->dato;**
2. Crear nodo auxiliar y asignarle el frente de la cola: **NCola *aux = frente**
3. Eliminar nodo del frente de la cola:

```
if(frente == fondo){
            frente = NULL;
            fin = NULL;
}
else{
            frente = fondo->link
}
delete aux;
}
```

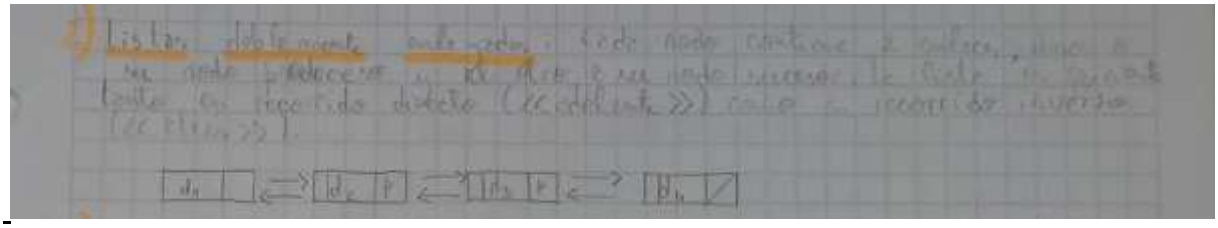
Cola circular:

- Estructura de datos lineal y estática
- Solo se puede insertar, extraer y preguntar si está vacía
- Es de acceso destructivo, impide su recorrido
- De tipo FIFO al igual que la cola (lo primero que entra es lo primero que sale)
- Tiene un límite de tamaño, al ser estática
- Se utiliza un arreglo de n posiciones, con un puntero al frente (apunta al primer elemento de la cola, es el que va a ser devuelto, válido solo si la cola no está vacía) y al fondo (apunta a la próxima posición a ser ocupada en la próxima carga)
- Llena=true (si la cola está llena) /false (si no lo está); vacía=true (si la cola está vacía) /false (si no lo está)
- Cuándo fondo alcanza a frente, la cola está llena (se detecta en la inserción de un elemento, ya que ambos quedan con el mismo índice), y cuando frente alcanza a fondo, la cola está vacía (se detecta al extraer un elemento, ya que ambos quedan con el mismo índice)
- Se maneja moviendo los punteros del frente y el fondo
- Acota el impacto en almacenamiento de lo que uno quiere guardar, al limitar la cantidad de espacios posibles donde se puede guardar información
- Se utiliza un struct con un arreglo, que contenga también sus variables

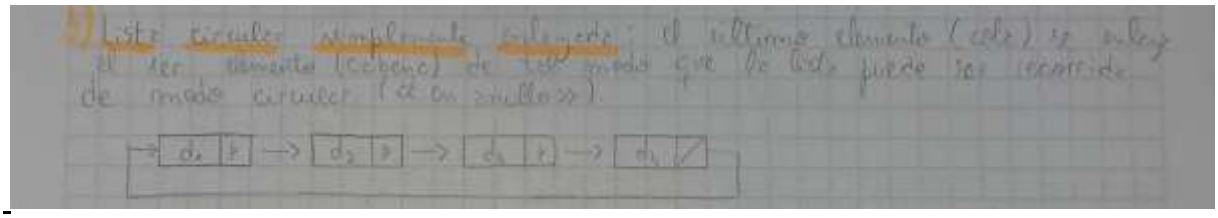
Listas: Constan de un número de nodos con 2 componentes (campos), un enlace al siguiente nodo y un valor, que puede ser de cualquier tipo. Hay diferentes tipos de listas:



- No es de acceso destructivo, por lo tanto, no hay devolución de memoria
- Se puede manipular datos de cualquier lugar
- Solamente el último elemento termina en NULL; si todos los elementos llegarán a ser NULL, la lista está vacía



- Cada nodo tiene otro puntero que va hacia el anterior, lo que permite ir y volver entre nodos, hacia delante o atrás
- El struct se compone del dato (con su tipo), más 2 punteros (del tipo del struct), uno siguiente y otro anterior



- Diferente a la cola circular
- La diferencia con la lista simplemente enlazada es que el último elemento apunta al primero, al principio, no hay un NULL
- El struct es igual a los otros
- La lógica de uso es igual
- Para salir de ella, en el recorrido la condición que evalúa el puntero actual debe ser distinto al puntero raíz

Alta lista (simplemente enlazada):

1. Crear espacio en memoria para almacenar un nodo: **NListaSE*nuevo_nodo = new NListaSE ()**
2. Cargar el valor dentro del nodo: **nuevo_nodo->dato = n**
3. Crear un nodo auxiliar y asignarle la lista: **NListaSE *aux = listase;**
4. Se establece el enlace del nuevo nodo como NULL ya que, inicialmente, será el último nodo de la lista: **nuevo_nodo->link = NULL;**
5. Se verifica si la lista esta vacía, si es así, entonces se asigna el nuevo nodo al primer nodo de la lista **if(aux == NULL) { listase = nuevo_nodo; }**
6. Si no está vacía, recorre hasta llegar al último elemento (**while(listase!=NULL)**). Una vez allí, se establece el enlace del último nodo al nuevo nodo, agregándole al final de la lista (**aux->link = nuevo_nodo**)

Árboles

- Es una estructura que sirve para representar entidades relacionadas jerárquicamente, donde cada nodo tiene uno y solo un padre (excepto la raíz) y no hay elementos aislados. Además, cada nodo no necesariamente tiene un solo sucesor, puede tener más de uno

- Se usan por ejemplo en sistema de archivos, algoritmos eficientes de búsqueda, etc.

- Puede ser la estructura de datos vacía o bien un conjunto de uno o más nodos tales que hay un nodo especial denominado raíz (n), los nodos restantes se dividen en k >= 0 conjuntos disjuntos, T1...Tk, tal que cada uno de estos es un árbol con raíces n1 … nk, entonces podemos construir un nuevo árbol que tiene a n como raíz y donde n1 … nk son hijos de n

- Terminología asociada:
  - **Rama/arco:** conexiones entre nodos
  - **Nodo hoja/terminal:** es un nodo que no tienen sucesores
  - **Nodo no hoja/interno:** es un nodo que al menos tiene un sucesor y un solo padre
  - **Nodo raíz:** es un nodo que no tiene un nodo padre, es el punto de entrada a la estructura de dato nodo padre: es el nodo predecesor al nodo actual
  - **Nodo hijo:** es el nodo sucesor al actual
  - **Nodos hermanos:** es/son el/los nodo/s que esta/n al mismo nivel, que tienen el mismo padre
  - **Nodos descendientes**: todos los nodos que desciendan del nodo actual (si existe un camino que va del nodo a al b, a es antecesor de b)
  - **Nodos ascendentes:** todos los nodos del que desciende el nodo actual (si existe un camino que va del nodo a al b, b es descendiente de a)

- Un árbol de un solo nodo es aquel que tiene solo nodo hoja y raíz al mismo tiempo

- No hay árboles sin hojas, ni raíces

- El grado del árbol es la cantidad máxima de hijos que puede tener, donde siempre cada nodo debe respetar ese grado, no puede excederse de esa cantidad de hijos: un árbol de grado G puede contener de 1 a G^n nodos, donde si es de grado G y altura A (distinta de 0) puede contener un mínimo de 1 nodo y un máximo de G^A - 1 nodos.

- Nodos adyacentes: nodos conectados por un arco

- Camino entre 2 nodos: si $n_1,n_2,\ldots,n_k$ es una secuencia de nodos tales que n_i es padre de n_{i+1} para i=1...k, entonces decimos que esta secuencia de nodos es un camino Longitud de camino: número de arcos que contiene un camino, así como el número de nodos del camino menos uno

- Altura: la altura de un nodo en un árbol es la máxima longitud de un camino que va desde el nodo a una hoja, contándose sólo los arcos (la altura de un árbol vacío es 0)

- Nivel/profundidad: es la distancia al nodo raíz desde un nodo determinado, donde la raíz tiene una distancia 0 de sí misma (se dice que la raíz está en el nivel 0), los hijos del nodo raíz están en el nivel 1, sus en el 2, etc.

- Un árbol a su vez puede dividirse en subárboles, cualquier estructura conectada por debajo de la raíz, donde cada nodo de un árbol es la raíz de un subárbol que se define por el nodo y todos o algunos de los descendientes del mismo (el primer nodo de un subárbol se conoce como el raíz del subárbol y se utiliza para nombrar el subárbol)

- Subárbol completo: sea S_a un subárbol de un árbol a, si para cada nodo n de S_a, S_a contiene también todos los descendientes de n en a. S_a se llama un subárbol completo

- Un árbol puede ser balanceado si y solo sí, sí en cada nodo sus alturas de sus subárboles difieren a lo máximo en 1 (se restan todas las alturas de sus subárboles, puede dar la diferencia 1, 0 o -1)

- Un árbol es perfectamente balanceado si y solo sí en cada nodo las alturas de sus subárboles son iguales (no hay diferencias)

- Un árbol es completo si cada nodo tiene la cantidad máxima de hijos que puede tener o ninguna

- Un árbol es equilibrado si dado un número máximo k de hijos de cada nodo y la altura del árbol h, cada nodo de nivel j < h-2 tiene exactamente k hijos

- Un árbol es perfectamente equilibrado si dado un número máximo k de hijos de cada nodo y la altura del árbol h, cada nodo de nivel j < h-1 tiene exactamente k hijos, dado un grado G, contiene Gn nodos en el nivel n con n >= 0 y n <= (altura - 1) o tiene todos los nodos que puede tener

- Un árbol perfectamente equilibrado es un árbol perfectamente balanceado con todas las hojas al mismo nivel (es perfectamente balanceado, y viceversa). Es un árbol simétrico

- Los árboles pueden presentarse como principales izquierdo y derecho: el izquierdo es cuando los punteros van desde los hijos a los padres; el derecho es cuando los punteros van desde los padres a los hijos. Por tanto, un struct de un árbol principal izquierdo implementa un solo puntero (ya que tiene un solo padre), pero un struct de un árbol principal derecho va a tener tantos punteros como grados del árbol (cada nodo puede tener tantos hijos como indique el grado)

- El árbol principal izquierdo no tiene un único punto de entrada (tantos puntos de entrada como hojas, se conforma una lista de hojas) y quedarían zonas inalcanzables del árbol. En cambio el árbol principal derecho solo tiene un punto principal derecho (solo se entra por la raíz); en el árbol principal derecho es necesario conocer la cantidad de grados, para saber la cantidad de punteros en él

- Los árboles pueden representarse como un grafo (gráficamente)
- como paréntesis anidados (en forma de lista)
- con sangría/escalonamiento/identación
- conjuntos anidados

Árbol binario

- Es aquel árbol donde ningún nodo puede tener más de 2 subárboles, ya que cada nodo puede tener 0, 1 o 2 nodos/subárboles)

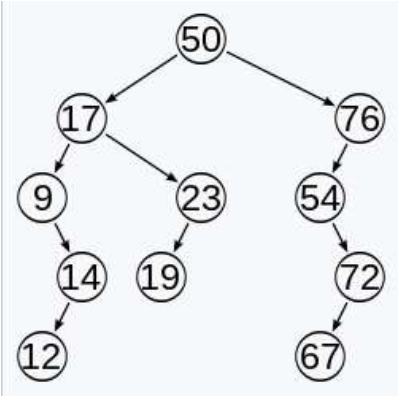
- El nodo de la izquierda es el hijo izquierdo, y el nodo de la derecha como hijo derecho
- El factor de balance es la diferencia de alturas entre los subárboles derecho e izquierdo
- Un árbol binario está perfectamente balanceado si la altura de los subárboles no difiere (balanceado si difiere en 1)
- Un árbol binario completo de profundidad n es un árbol en el que para cada nivel (del 0 al n-2) hay un conjunto lleno de nodos y todos los nodos hoja ocupan a nivel n-1 ocupan las posiciones más a la izquierda del árbol
- Un árbol binario completo que contiene 2n nodos en el nivel n es un árbol completo
- Un árbol binario degenerado es un árbol que tiene una sola hoja, ya que todos los nodos no-hoja tienen un solo hijo
- Un árbol binario de búsqueda es un árbol binario donde se cumple para todos sus subárboles que los menores van a la izquierda de la raíz y los mayores van a la derecha (los menores de acuerdo a la clave van a la izquierda y los mayores van a la derecha)

Árbol AVL

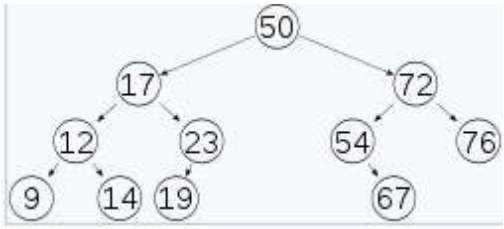
Un árbol AVL es un tipo especial de árbol binario donde están siempre equilibrados de tal modo que, para todos los nodos, la altura de la rama izquierda no difiere en más de una unidad de la altura de la rama derecha o viceversa (1, 0 o -1)

Para conseguir esta propiedad de equilibrio, la inserción y el borrado de los nodos se ha de realizar de una forma especial. Si al realizar una operación de inserción o borrado se rompe la condición de equilibrio, hay que realizar una serie de rotaciones de los nodos

Árbol no equilibrado (no AVL)



Árbol equilibrado (AVL), luego de las rotaciones en cada inserción o eliminación



Rotación simple a la derecha

De un árbol de raíz (r) y de hijos izquierdo (i) y derecho (d), lo que haremos será formar un nuevo árbol cuya raíz sea la raíz del hijo izquierdo, como hijo izquierdo colocamos el hijo izquierdo de i (nuestro i') y como hijo derecho construimos un nuevo árbol que tendrá como raíz, la raíz del árbol (r), el hijo derecho de i (d') será el hijo izquierdo y el hijo derecho será el hijo derecho del árbol (d)

Rotación simple a la izquierda

De un árbol de raíz (r) y de hijos izquierdo (i) y derecho (d), consiste en formar un nuevo árbol cuya raíz sea la raíz del hijo derecho, como hijo derecho colocamos el hijo derecho de d (nuestro d') y como hijo izquierdo construimos un nuevo árbol que tendrá como raíz la raíz del árbol (r), el hijo izquierdo de d será el hijo derecho (i') de r y el hijo izquierdo será el hijo izquierdo del árbol (i).
Precondición : Tiene que tener hijo derecho no vacío

Rotación doble a la derecha

La Rotación doble a la Derecha son dos rotaciones simples, primero rotación simple izquierda y luego rotación simple derecha

Rotación doble a la izquierda

La Rotación doble a la Izquierda son dos rotaciones simples, primero rotación simple derecha y luego rotación simple izquierda.

Inserción

La inserción en un árbol de AVL puede ser realizada insertando el valor dado en el árbol como si fuera un árbol de búsqueda binario desequilibrado y después retrocediendo hacia la raíz, rotando sobre cualquier nodo que pueda haberse desequilibrado durante la inserción.

1. Buscar hasta encontrar la posición de inserción o modificación (proceso idéntico a inserción en árbol binario de búsqueda)
2. Insertar el nuevo nodo con factor de equilibrio "equilibrado"
3. Desandar el camino de búsqueda, verificando el equilibrio de los nodos, y re-equilibrando si es necesario

Árbol de búsqueda:

Una Árbol de búsqueda también llamado BST (por acrónimo binary search tree) es un tipo de árbol binario donde se cumple para todos sus subárboles que los menores van a la izquierda de la raíz y los mayores van a la derecha. Es un tipo especial de árbol que sirve para guiar la búsqueda de un registro, dado el valor de uno de sus campos.

Struct:

```
struct nodo_arbol_binario {
        struct informacion del nodo….; // dato
        struct nodoArbol * iz;
        struct nodoArbol * de;
};
typedef struct nodo_arbol_binario NABinario;
```

Barridos:

- PREORDEN: Se caracteriza por las iniciales RID (por su orden de mostrado), Raiz , hijo Izquierdo, hijo Derecho.
- INORDEN: También llamado Simétrico, se caracteriza por las iniciales IRD (por su orden de mostrado), hijo Izquierdo, Raiz, hijo Derecho.
- POSTORDEN: Se caracteriza por las iniciales IDR, hijo Izquierdo, hijo Derecho, Raiz.
- POR NIVELES: Se caracteriza por recorrer el árbol siguiendo la información nivel por nivel desde la raiz, y dentro de cada nivel de izquierda a derecha.

Algunas características adicionales:

- Permite implementar índices, una estructura auxiliar que sirve para buscar con facilidad y rapidez
- El acceso a un archivo para buscar algo se mide en accesos, cuántas veces se entra a ese archivo
- Un índice se implementa como un árbol de datos. Caso que el dato sea menor que el nodo, se va trasladando hacia la izquierda por el árbol. Si es mayor, irá por la derecha. En ambos casos permite descartar una mitad
- Caso que el árbol este perfectamente balanceado. es igual de performante que una búsqueda binaria, desaconsejando la búsqueda secuencial. La performance se calcula como log 2 n, donde n es la cantidad de registros
- Al insertar un dato y teniendo más de un árbol, se deben actualizar los árboles
- Los nodos de un mismo dato pueden encontrarse en distintas posiciones en el árbol, pero apuntar a un mismo dato del archivo

Arboles B:

- Tiene restricciones adicionales que garantizan que el árbol siempre estará equilibrado y que el espacio desperdiciado por la eliminación, si lo hay, nunca será excesivo.
- Los algoritmos para insertar y eliminar son más complejos para poder mantener estas restricciones.
- La mayor parte de las inserciones y eliminaciones son procesos simples que se complican sólo en circunstancias especiales (por ejemplo, siempre que intentamos una inserción en un nodo que ya está lleno o una eliminación de un nodo que está a menos de la mitad de su capacidad).

Un árbol B de orden p, se puede definir formalmente de este modo:

Son árboles que cumplen con:

- Cada nodo interno tiene la forma de <p1,<k1 pr1>,p2, < k2 pr2>… < k1 pq-1>,pq>, donde q≤p (p es la cantidad máxima de hijos, el orden que puede tener un nodo). P mayúscula son punteros a nodos hijos, y Pr son punteros a registros, el dato buscado, ya que no tienen los demás datos que no se buscan, para no repetirlos. K son las claves para buscarlos (junto con el registro a esa clave para buscarlo). Q es el subíndice de k, un valor entero.
- Todo el árbol es del mismo tipo de datos, hay un solo tipo de nodo
- Dentro de cada nodo k1 es menor que k2 …, es menor que k-1. Es decir, valores ordenados de menor a mayor
- Para todos los valores del campo clave de búsqueda X del subárbol al que apunta pi tenemos:

    - $k_{i-1} < x < k_i$ para $1< i< q$ (se incluyen todos los subárboles, se dejan afuera el primer y último puntero; todas las claves se encontrarán entre el de la izq. y el de la der., entre 2 valores)
    - $x < k_i$ para i=1 (a la izquierda del primero hay valores aún menores)
    - $x > k_{i-1}$ para i=q (a la derecha del último hay valores aún mayores)

Posicionándose en cualquier puntero, se muestran todas las claves que se encontraran en un subárbol, junto a su conjunto de claves

- Cada nodo tiene a lo sumo p punteros a árbol
- Cada nodo (excepto la raíz) tiene, por lo menos, p/2 punteros a árbol, redondeado para abajo. La raíz tiene por lo menos 2 punteros, a menos que sea el único nodo
- Un nodo con q punteros a árbol, q ≤ p, tiene q-1 claves. Si yo tengo 1 sola clave, tiene que tener 2 punteros, uno a la izquierda y uno a la derecha; salvo la hoja, que no tiene punteros. La secuencia arranca a la izquierda
- Todas las hojas se encuentran al mismo nivel
- La cantidad mínima de claves es ((t/2)-1), salvo en la raíz. También redondeado para abajo.

Eliminación

El problema no es cuando el nodo está vacío, sino cuando hay menos de la cantidad mínima (subocupación), por eso actúa ahí

- En caso de eliminar una clave en la hoja, simplemente se borra, y luego se pregunta si quedo sub-ocupado para aplicar la corrección
- En caso de eliminar una clave no en una hoja, se intercambia ese valor con el mayor valor del subárbol izquierdo o el menor del subárbol derecho (siempre usando el mismo criterio, en nuestro caso, el mayor del subárbol izquierdo). El valor a eliminar se coloca en la posición del valor que se eligió del subárbol izquierdo y se elimina (siempre se encuentra en una hoja) y se coloca ese valor donde estaba, y se consulta si quedo subocupado para aplicar corrección
- En caso de que un nodo hoja quede subocupado, verifica si hay un hermano izquierdo, consulta si puede pasar una clave para colocarla sin queda subocupado; en caso negativo, verifica si hay un hermano derecho, consulta si puede pasar una clave para colocarla sin queda subocupado. En caso afirmativo, baja la clave en el padre al hijo y sube la elegida (siempre sube la clave más pequeña del hermano)
- En caso de que no tenga hermanos, o bien los hermanos no puedan ceder una clave sin quedar subocupados, el árbol fusiona el nodo bajando el menor del árbol padre y trayendo todo lo que tiene el nodo hermano
- La raíz, puede quedar subocupado, pero no vacío. Si pasara, se elimina la raíz.

Arboles B+:

- La mayoría de las implementaciones de un índice multinivel dinámico utilizan una variante de la estructura de datos en árbol B denominada árbol B +. Cada valor del campo de búsqueda aparece una vez en algún nivel del árbol, junto con un puntero de datos.
- El nodo hoja tiene clave y un puntero al disco con el registro que se corresponde con esa clave. Además, solo las hojas tienen un puntero al hermano derecho.
- Los nodos hoja de un árbol B + normalmente están enlazados para ofrecer un acceso ordenado a los registros a través del campo de búsqueda.
- Algunos valores del campo de búsqueda de los nodos hoja se repiten en los nodos internos del árbol B + con el fin de guiar la búsqueda.
- El nodo no hoja (o intermedio) es igual que el B, salvo que no tiene puntero a registro (que no se dibuja)
- Los datos pueden estar ninguna vez, una vez (solo en las hojas) o 2 veces (en las hojas y algún nodo intermedio). De este modo, los nodos hoja quedan enlazados entre sí como una lista simplemente enlazada.

1. Cada nodo interno tiene esta forma:

   $<P_1, K_1, P_2, K_2, ..., P_{q-1}, K_{q-1}, P_q>$

   donde $q = p$ y cada P_i es un **puntero de árbol**.
2. Dentro de cada nodo interno, $K_1 < K_2 < ... < K_{q-1}$.
3. Para todos los valores X del campo de búsqueda en el subárbol al cual apunta P_i, tenemos que $K_{i-1} < X \leq K_i$ para $1 < i < q$; $X \leq K_i$ para $i = 1$; y $K_{i-1} < X$ para $i = q$ (véase la Figura 14.11[a]).[9]
4. Cada nodo interno tiene, a lo sumo, p punteros de árbol.
5. Cada nodo interno, excepto el raíz, tiene al menos $\lceil (p/2) \rceil$ punteros de árbol. El nodo raíz tiene como mínimo dos punteros de árbol si es un nodo interno.
6. Un nodo interno con q punteros, $q \leq p$, tiene $q - 1$ valores del campo de búsqueda.

La estructura de los *nodos hoja* del árbol B^+ de orden p (véase la Figura 14.11[b]) es la siguiente:

1. Cada nodo hoja tiene la siguiente forma:

   $<<K_1, Pr_1>, <K_2, Pr_2>, ..., <K_{q-1}, Pr_{q-1}>, P_{sig}>$

   donde $q \leq p$, cada Pr_i es un puntero de datos, y P_{sig} apunta al siguiente *nodo hoja* del árbol B^+.
2. Dentro de cada nodo hoja, $K_1 \leq K_2 ..., K_{q-1}$, $q \leq p$.
3. Cada Pr_i es un **puntero a datos** que apunta al registro cuyo valor del campo de búsqueda es K_i o a un bloque del fichero que contiene el registro (o a un bloque de punteros de registro que apuntan a los registros cuyo valor del campo de búsqueda es K_i si el campo de búsqueda no es una clave).
4. Cada nodo hoja tiene como mínimo $\lceil (p/2) \rceil$ valores.
5. Todos los nodos hoja están en el mismo nivel.

La estructura de los nodos internos de un árbol B+ de orden p, es la siguiente:

Alta de árbol binario de búsqueda:

```
void alta_arbol(nodo_arbol_binario *&arbol,int n_dato){
        nodo_arbol_binario *aux=new (nodo_arbol_binario);
        nodo_arbol_binario *actual=arbol;
        nodo_arbol_binario *anterior=NULL;
        aux->dato=n_dato;
        if (arbol!=NULL){
                        while (actual!=NULL){
                                anterior=actual;
                                if (n_dato<actual->dato){
                                        actual=actual->izq;
                                }
                                else {
                                        actual=actual->der;
                                }
                        }
                        if (n_dato<anterior->dato){
                                anterior->izq=aux;
                        }
                        else {
                                anterior->der=aux;
                        }
        }
        else {
                        arbol=aux;
        }
        aux->izq=NULL;
        aux->der=NULL;
}
```

Grafos

- Se denota como G={P,R} (P=puntos, R=relaciones)
- Es un conjunto de nodos interconectados por medio de arcos o aristas
- Se denominan grafos irrestrictos porque no tiene restricciones; a diferencia de otras estructuras (como una lista por ejemplo, que es un grafo pero con restricciones -el primero sin antecesor, el último sin sucesor-) que son todas grafos con restricciones

Se puede hacer una definición por comprensión y extensión:

- **Comprensión:** es un conjunto P de puntos y un conjunto R de relaciones, con P igual al conjunto X de elementos, tal que X es un nodo y R igual al conjunto de pares (X,Y) tal que tanto X como Y pertenecen al conjunto P de puntos, y X se relaciona con Y

- **Extensión:** es necesario tener un grafo hecho, un grafo determinado, por ejemplo P={A,B,C,D} y R={(A,B), (C,D), (D,B)}, es decir, se expresan las relaciones

- Funciones de asignación a nodo son las características o propiedades del nodo, según el tipo del nodo

- Funciones de asignación de arco son las características o propiedades del arco, según el tipo de arco

Los nodos no pueden compartir identificador, pero los arcos si, aunque es conveniente tener un identificador global (con el problema de que se crea un dato extra, ajeno a la entidad, o el aumento de la susceptibilidad a cometer errores como cargar 2 entidades iguales con identificadores distintos)

Propiedades de los grafos

- **Paso (entre 2 nodos):** es la secuencia P (x,z) es la secuencia (y_0, y_1, y_2...y_n), con n mayor a 1, tal que se cumple que x = y_0 y z=y_n (x es el nodo de origen y z el nodo de destino), y_{i-1} es distinto a y_i (tal que 1=<y<n, es decir, los nodos son distintos al anterior) e (y_{i-1},y_i) pertenecen al conjunto r de relaciones, con y mayor a igual a 1 y menor que n. La longitud de paso |P (x,y)| es el n.º de arcos entre x y z. Importa el sentido de la flecha

- **Camino** (C(x;z), entre 2 nodos): es la secuencia (y_0, y_1, y_2...y_n), con n mayor a 1, tal que se cumple que x = y_0,, y z=y_n (x es el nodo de origen y z el nodo de destino), y_{i-1} es distinto a y_i (tal que 1=<y<n, es decir, los nodos son distintos al anterior) e (y_{i-1},y_i) pertenecen al conjunto r de relaciones o (y_i, y_{i-1}), con y mayor a igual a 1 y menor que n. La longitud de camino |P (x,y)| es el n.º de arcos entre x y z, sin importar el sentido. No importa el sentido de la flecha

- **Ciclo:** es un paso que va de x a x, con una longitud mayor o igual a 2 (|P(x,x)|>=2), importando el sentido de la flecha

- **Circuito:** es un camino que va de x a x, con una longitud mayor o igual a 2(|C(x,x)|>=2), sin importar el sentido de la flecha

- **Bucle:** es un paso que va de x a x con longitud igual a 0 (P(x,x)=0)

- **Left de x:** es el listado de nodos desde los que yo puedo llegar a x, se conectan con el {y/y E P; (y,x) E R}

- **Right de x:** es el listado de nodos que salen de x {z/z E P; (x,z) E R}

- **Conjunto ideal izquierdo:** son los y tales que y pertenece a p y existe el paso (y,x) {y/y E P; (y,x)}, es decir, todos los nodos desde los que se puede llegar a x

- **Conjunto ideal derecho:** son los y tales que y pertenece a p y existe el paso (x,y) R(x) = {z/z E P; (x,z)}, es decir, todos los nodos a los que puedo llegar desde x

- **Grado del conjunto izquierdo:** la cantidad de nodos que tiene el left | L(x)|

- **Grado del conjunto derecho:** la cantidad de nodos que tiene el right |R(x)|

- **Conjunto minimal:** es el conjunto de puntos a los que no llega ninguna flecha, el left está vacío {x / x E P, |L(x)|= 0}

- **Conjunto maximal:** es el conjunto de puntos desde los que no sale ninguna flecha, el right está vacío {z / z E P, |R(z)|= 0}
- **Mínimo:** x es mínimo si el left de x es 0 y x es único

- **Máximo:** x es máximo si el right de x es 0 y x es único

- **Grafo básico:** es libre de bucles y además para todo (x,y) pertenecientes a P, si existe un paso de x a y mayor o igual a 2, entonces (x,y) no pertenecen a R

- **Subgrafo:** es una porción de un grafo donde P' es subconjunto de P y R' = R|p

- **Grafos isomorfos:** 2 grafos G1={P1,R1} y G2={P2,R2} son isomorfos si G1 es equivalente a G2, y existe una función que mapea P1→P2 tal que para todo (x,y) existente en P1 si existe una relación entre x e y ((x,y) E R), entonces si y sólo si existe en esa función de x y de y existente en la relación, y la función de x e y guardan una relación; es decir, la misma relación se debe cumplir con los nodos de P2

Una de las formas de escribir en código el grafo es la matriz de adyacencia, que implementa una estructura de datos estática

- Se arma una matriz con tantas filas y columnas como nodos tenga el grafo, generando una matriz cuadrada, inicializada con 0, del tipo booleana de tamaño m X m (cantidad de puntos del conjunto P), donde las filas representan los orígenes y las columnas los destinos

- Es más performante que una estructura estática, pero tiene varios problemas:

  - Se usa poca memoria de la guardada en realidad
  - Se imposibilita agregar elementos nuevos (solucionando esto con la sobredimensión de la matriz)

  - En caso de tener más de una conexión entre nodos (solucionando con la suma de 1 valor más del que se tiene en la conexión, pero complicando la interpretación de la matriz)

- Si el identificador no comienza en 1 o no es un número, por ejemplo, debe despegarse del nombre que uso para cada nodo (se soluciona usando más estructuras auxiliares, como un struct que guarde el nombre de cada nodo, o el valor del identificador, complicando el struct utilizado para representar el problema)
- Que haya más de una relación entre nodos (solucionándose con darle profundidad a la matriz de relaciones, posibilitando más conexiones entre nodos, pero aumentando la complejidad de la matriz)

- Lo más conveniente sería armar una matriz de structs con todas las funciones de asignación a arco que haya, además de una matriz que contenga sólo las relaciones (generalmente numérica)

- Permiten ejecutar consultas rápidas

- Estas matrices se alojan en memoria
- Cada conexión se visualiza como con un valor en la intersección fila-columna, donde la fila representa el nodo de salida de la conexión, y la fila el nodo de llegada

- Ejemplo: **struct grafo {int m [100][100] (la matriz, se debe inicializar en 0) int t (cantidad de nodos, parte útil de la matriz) int tt (tamaño total de la matriz) }**

Potencia n-enesima de la matriz de adyacencia de un grafo

Lista de adyacencia

Se define grafos implementados como lista de adyacencia del grafo G, a la implementación usando estructuras dinámicas auto referenciadas con una estructura para los nodos y una para arcos, la implementación prevé además un puntero de acceso del tipo puntero a Nodo. Algunas características pueden ser:

- Es una estructura implementada de manera dinámica, a diferencia de la matriz de adyacencia
- M puede ser desconocido con lo cual, a diferencia de la implementación con matriz de adyacencia, es posible agregar y quitar nodos en cualquier momento.
- Optimizan el uso del almacenamiento.
- Es un poco más complejo las consultas y mantenimiento del grafo.

```
struct nodo_grafo{

int id_nodo; // carga util del nodo

struct nodo_arco* lista_arco;

struct nodo_grafo* link;

};

typedef struct nodo_grafo NGrafo;

// Nota 1: además cada nodo tiene un puntero de asignación a una estructura dinámica que contiene la información de nodos

struct nodo_arco{

int id_arco;
```

struct nodo_grafo* destino;

struct nodo_arco* link;

};

typedef struct nodo_arco NArco;

// Nota 2: además cada nodo tiene un puntero de asignación a una estructura dinámica que contiene la información de arcos