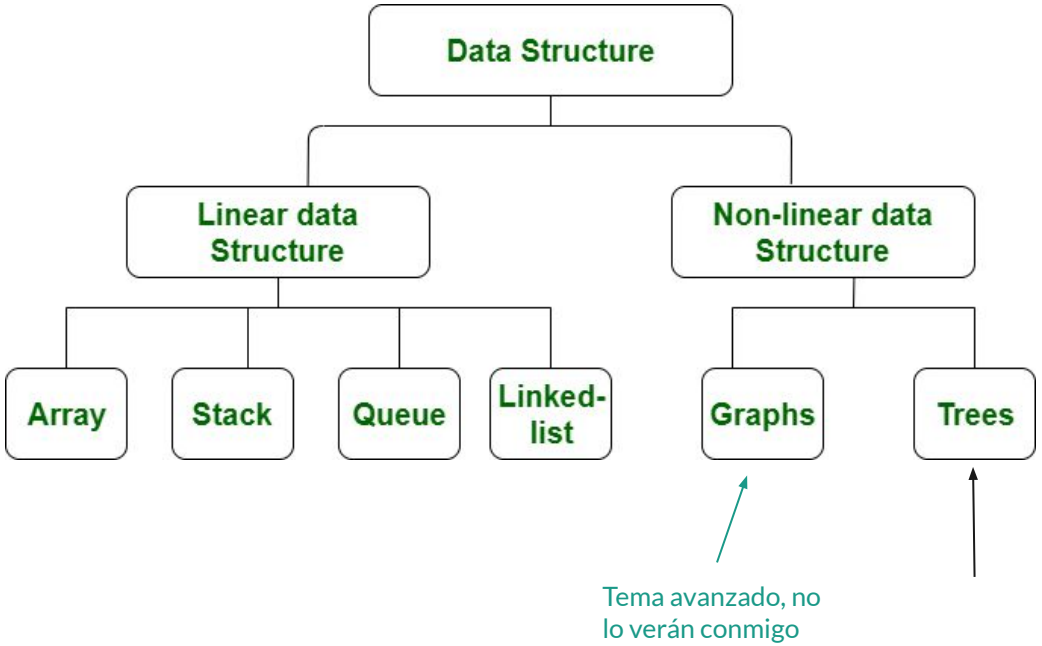
Clase 3 - C++ STL

Estructuras de Datos II

Preparación GPC-UPC
Adaptado de las diapositivas de Rodolfo Mercado.
Complementado por Oscar Burga.
Tutor: Oscar Burga

Por ver hoy

- Comparadores
- Concepto general: Heaps
- Colas de prioridad (Priority queues)
- Concepto general: Árboles binarios
- Contenedores asociativos (!)
- Set
- Map
- Multiset, Multimap
- Funciones importantes de Set y Map



Comparadores

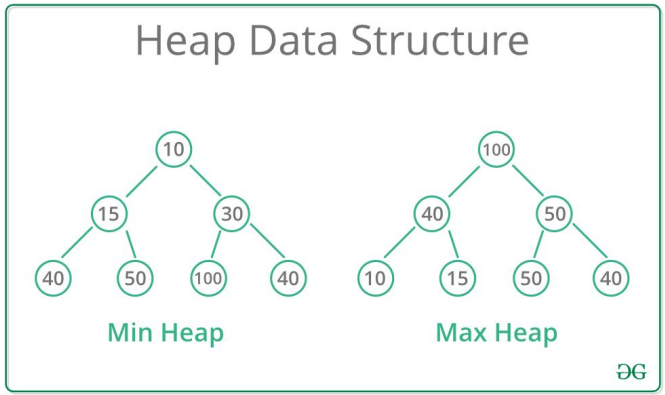
- Sirven para… comparar elementos, pero de manera especial según sea necesario.
- Útil para ordenar un conjunto de elementos por criterios específicos
- Pueden crear comparadores para sus propias estructuras
- El comparador debe ser estricto, dos elementos iguales deben retornar falso (no usar <= ó >=).

```cpp
//Ejemplo: Ordenar coordenadas por X ascendiente, Y descendiente
bool cmp(pair<int,int> &a, pair<int,int> &b){
    if (a.first == b.first) return a.second > b.second;
    return a.first < b.first;
}
int main(){
    vector<pair<int, int>> v = {{2, 9}, {1, 2}, {1, 3}};
    sort(v.begin(), v.end(), cmp);  //Ordenar usando el comparador
    // (1, 3), (1, 2), (2, 9)

    return 0;
}
```

Concepto general: Binary Heap

- Estructura de datos con forma de *árbol binario* que mantiene a sus elementos ordenados de tal forma que se cumpla la **propiedad de montículo**
- **Propiedad de montículo:** Todos los descendientes de un nodo son *menor-iguales* o *mayor-iguales a él*.
- Dos tipos: Min-heap y Max-heap (ordenar por mínimo y por máximo, respectivamente).
- Soporta las siguientes operaciones:
  - Obtener min/max en O(1)
  - Eliminar min/max en O(log N)
  - Insertar elem en O(log N)



Contenedor: Priority Queue

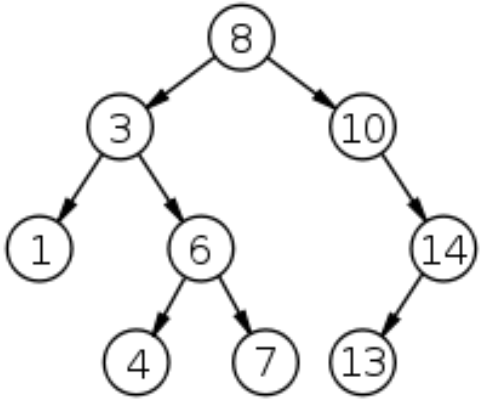
```
priority_queue<pair<int, int>> q;
```

- #include <queue>
- Comportamiento de Heap. Similar a una cola, pero ahora se mantienen ordenados los elementos.
- Max-heap por defecto (soporta comparadores personalizados).
- No es iterable :(Solo se puede acceder al primer elemento (max/min) mediante el método top().
- Ventaja: Factor constante bajo a comparación de las otras estructuras STL basadas en árboles binarios (set, map, etc.)

```
while(q.size()){ //Cola de prioridad de pares
    //Obtener primer elemento, como si fuera una cola
    pair<int, int> maximo = q.top(); // O(1)
    q.pop(); //Eliminarlo O(logN)
    //Hacer cosas
    q.push({x, y}); //Agregar otro elemento O(logN)
}
```

Concepto general: Árbol binario de búsqueda

- Estructura de datos que mantiene elementos ordenados por un criterio específico.
- Cada nodo tiene un hijo izquierdo y un hijo derecho (generalmente menor y mayor, respectivamente).
- Buscar, insertar y remover, todo en O(logN) (amortizado).
- Factor constante alto, y el rebalanceo puede ser costoso. Se debe usar cuidadosa e inteligentemente.



- ***Estructura MUY poderosa*** (si los saben usar)

Las estructuras basadas en árboles binarios del STL de C++ están implementadas con un tipo de árbol binario autobalanceado llamado Red-Black Tree

Contenedor: Set

- #include <set>
- Permite almacenar datos (llamados llaves) de valor único.
- Las llaves son guardadas en orden ascendente por defecto (soporta comparadores personalizados)
- Búsqueda, inserción y eliminación en O(logN)
- Con creatividad, se pueden usar para muchísimas cosas (luego veremos funciones muy útiles).
- Son iterables, pero limitadas. Los iteradores de contenedores basados en árboles binarios son especiales y no son tan libremente manipulables como los que hemos visto anteriormente. (Ej: No se pueden sumar/restar directamente, solo de a uno en uno mediante los operadores it++ y it--).

```
set<int> s;                 //declarar set
s.size();                   //tamaño O(1)
s.insert(x);                //insertar O(logN)
s.erase(x);                 //eliminar O(logN)
set<int>::iterator it;      //iterador
it = s.find(x);             //buscar O(logN)
for(int val: s)             //Iterar sobre el set O(N)
    printf("%d ", val);
```

Contenedor: Map

```
map<int, double> m;
```

- #include <map>
- Es la misma estructura que set, pero ahora permite asociar un valor a cada llave (como si fueran pairs). Guarda pares de la forma <llave, valor> (la llave sigue siendo única).
- Escritura como si fuera un arreglo, con la llave como índice.
- Ojo: Si accedes directamente a un elemento que no existe, **se va a crear** en la estructura automáticamente. Mucho cuidado con esto, porque estar añadiendo elementos innecesariamente puede ralentizar mucho la solución.

```
map<string, int> edad;
edad.size();              //tamaño O(1)
edad["marco"] = 30;       //Insertar O(logN)
edad["marco"];            //acceso O(logN)
//OJO: CUIDADO CON ACCEDER A ELEMENTOS QUE NO EXISTEN
edad.erase("marco");      //Eliminar O(logN)
map<string, int>::iterator it; //iterador
it = edad.find("marco"); //buscar O(logN)
```

Variaciones: Multiset y Multimap

- Iguales que set y map, respectivamente, pero soportan llaves repetidas.
- Ojo: Hay que ser muy cuidadoso al eliminar elementos en estos contenedores. El erase(key) en multiset/multimap eliminará TODOS los elementos con llaves que coincidan con el key especificado. Para este caso, es recomendable pasar un iterador al elemento específico que se desea eliminar.

Funciones principales: Set y Map

- **m.insert(x):** Inserta el elemento x en el contenedor. No hace nada si ya existe la llave en el caso de set y map.
- **m.count(x):** Contar las ocurrencias de x en el contenedor. En set y map solo puede retornar 0 ó 1. Útil para verificar existencia de elementos. O(logN + C), donde C es la cantidad de ocurrencias de x en el contenedor (en caso de ser multiset).
- **m.find(x):** Retorna iterador apuntando a X en el contenedor. Si X no existe, retorna el iterador al final m.end(). O(logN)
- **m.lower_bound(x):** Obtiene un iterador al primer elemento igual o mayor a x, o retorna m.end() si no existe. O(logN)
- **m.upper_bound(x):** Obtiene un iterador al primer elemento estrictamente mayor a x, o retorna m.end() si no existe. O(logN)
- **m.erase(x):** x puede ser una llave o un iterador. En el caso de ser llave, elimina todas las coincidencias. Si es iterador, elimina el elemento apuntado por el iterador. O(C) (amortizado), donde C es la cantidad de coincidencias.

Problemas de práctica - Importantes

- Es muy importante que se familiaricen con estas 2 estructuras, pueden hacer muchísimas cosas y se van a volver de vital importancia cuando aprendan a usarlas apropiadamente.
- Contest de la clase con dos problemas sencillos
- Ver la siguiente diapositiva para algunos ejercicios propuestos

Ejercicios propuestos - Set y Map

- Implemente una función que reciba dos llaves X, Y. Si la llave X existe en el contenedor, deberá cambiarse a Y. En caso de que Y ya exista en el contenedor, X solo debe ser eliminado. Si X no existe en el contenedor, no hacer nada. **Debe preservar el valor asociado a la llave para el caso del contenedor map.**

- Aprovechando la función lower_bound de set/map, implemente una función que reciba una llave X y que retorne un iterador al primer elemento con llave **estrictamente menor** a X en O(logN). En caso de que no exista un elemento que cumpla con dicha condición, retorne el iterador al final del contenedor.

- Dado un conjunto de N pares que representen rangos numéricos cerrados (ej: { [2, 5], [6, 8] }) implemente una solución que permita responder Q consultas en O(logN) como máximo por cada una. Las consultas son de la siguiente forma: "*Dados 2 enteros L y R, encontrar el/los rango(s) que sean intersecados por L, R o ambos.*"
    - Nota: Está garantizado que todos los rangos (pares) que se te dan son disjuntos entre sí (no hay dos rangos que cubran un mismo número).
    - Está permitido realizar procesamiento previo a las consultas en tiempo máximo O(NlogN).

Pistas - Ejercicios Propuestos

- Primer ejercicio: No podemos modificar directamente las llaves. ¿Tal vez deberíamos extraerla del contenedor primero, y luego insertar la modificada?

- Segundo ejercicio: Recordar que estos contenedores están ordenados, son iterables, y sus iteradores soportan las operaciones de incremento y decremento (++ y --).

- Tercer ejercicio: Pueden reutilizar la solución al segundo ejercicio de manera inteligente.