Semillero de Programación

Ordenamiento Topológico, Componentes Fuertemente Conexas y Estructuras de Datos

Ana Echavarría Juan Francisco Cardona

Universidad EAFIT

8 de marzo de 2013

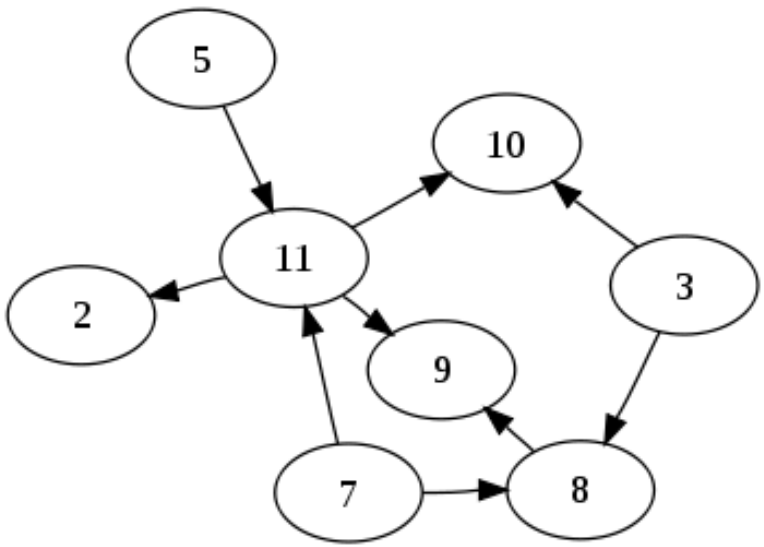
Contenido

Contenido

DAG

DAG

Un DAG (Directed Acyclic Graph) es un grafo dirigido que no tiene ciclos.



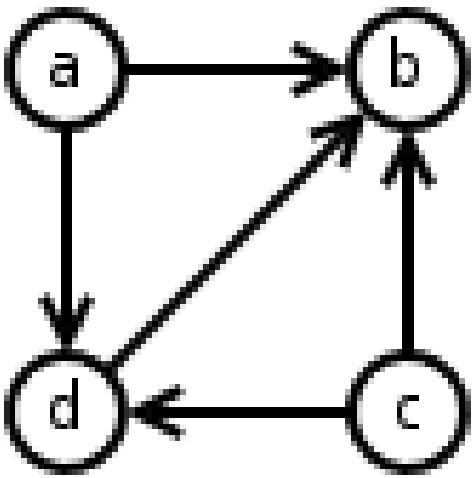
Ordenamiento Topológico

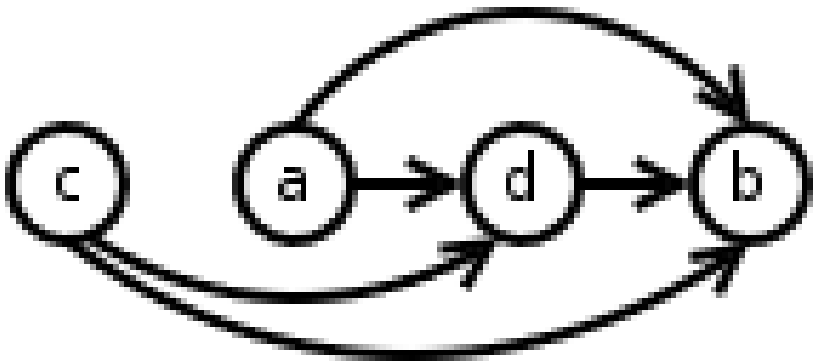
Ordenamiento Topológico

Un ordenamiento topológico o topological sort de un DAG $G = (V, E)$ es un ordenamiento lineal de sus nodos V de tal forma que si $(u, v) \in E$ entonces u aparece antes que v en el ordenamiento.

Este ordenamiento se puede ver como una forma de poner todos los nodos en una línea recta y que las aristas vayan todas de izquierda a derecha.

Ejemplo





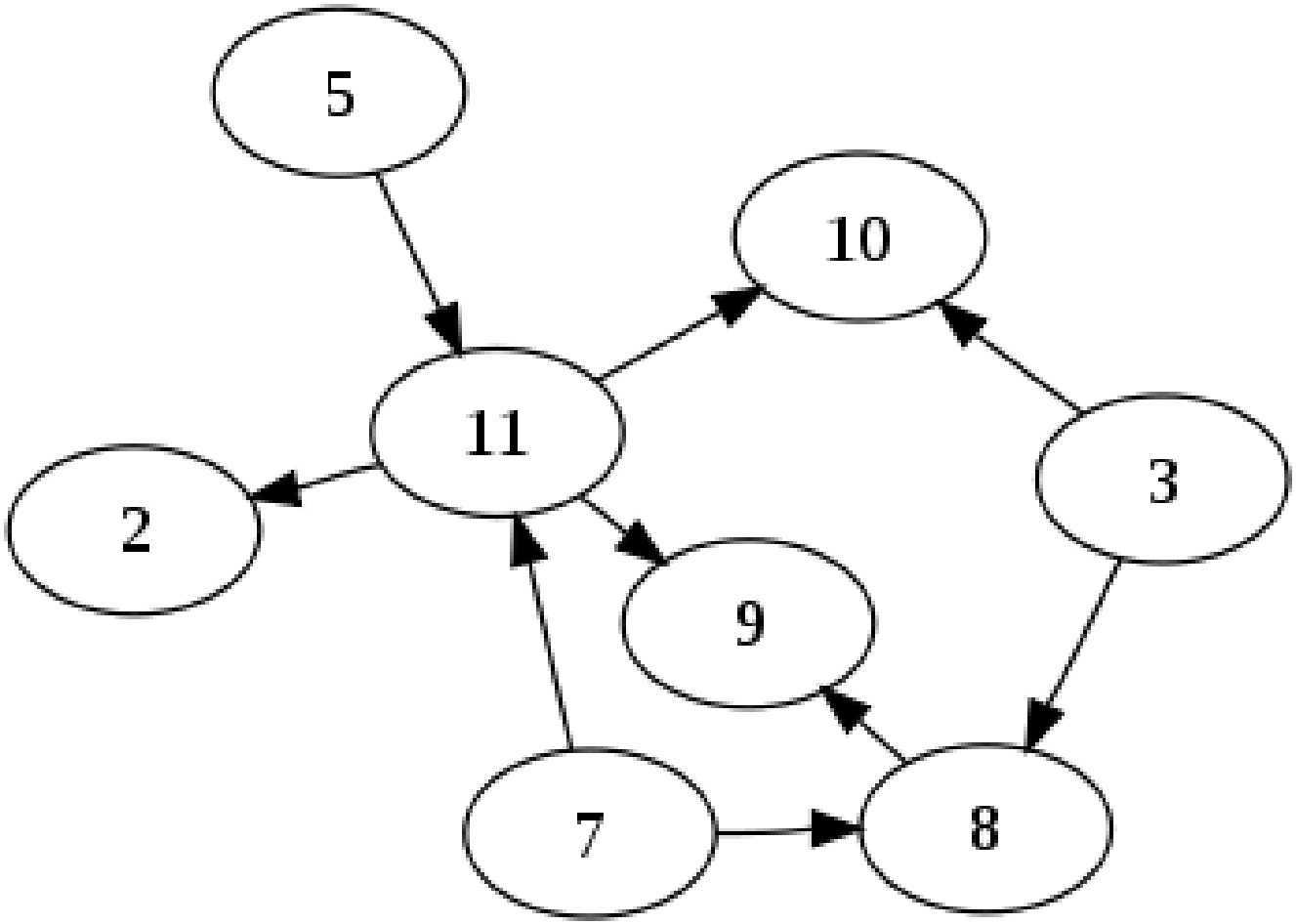
Algoritmo

1. Hacer un DFS con el grafo
2. Cuando marco un nodo como negro, lo inserto a un vector
3. Reversar el orden de los elementos del vector
4. El vector contiene un ordenamiento topológico del grafo

Algoritmo

```
1  vector <int> g[MAXN];
2  bool seen[MAXN];
3  vector <int> topo_sort;
4
5  void dfs(int u){
6     seen[u] = true;
7     for (int i = 0; i < g[u].size(); ++i){
8        int v = g[u][i];
9        if (!seen[v]) dfs(v);
10    }
11    topo_sort.push_back(u); // Agregar el nodo al vector
12 }
13 int main(){
14    // Build graph: n = verices
15    topo_sort.clear();
16    for (int i = 0; i < n; ++i) seen[i] = false;
17    for (int i = 0; i < n; ++i) if (!seen[i]) dfs(i);
18    reverse(topo_sort.begin(), topo_sort.end());
19    return 0;
20 }
```

Ejemplo



¿Por qué funciona?

- Cuando meto un nodo a la lista, es porque ya procesé todos sus vecinos.
- Si ya procesé todos sus vecinos, ellos ya están en la lista.
- Cuando meto un nodo a la lista, todos sus vecinos ya están antes que él en la lista, entonces en el ordenamiento van a quedar después de él.
- En conclusión, en el ordenamiento que generamos, los vecinos de cada nodo van a estar después de él por lo que es un ordenamiento topológico.

Complejidad

Complejidad

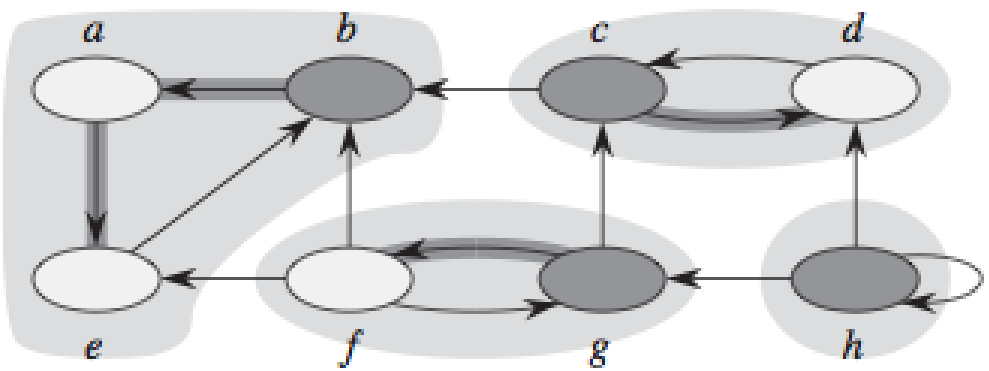
Hacer el ordenamiento topológico toma $O(V + E)$ para el dfs y $O(V)$ para reversar la lista. En total la complejidad es $O(V + E)$.

Contenido

Componentes Fuertemente Conexas

SCC

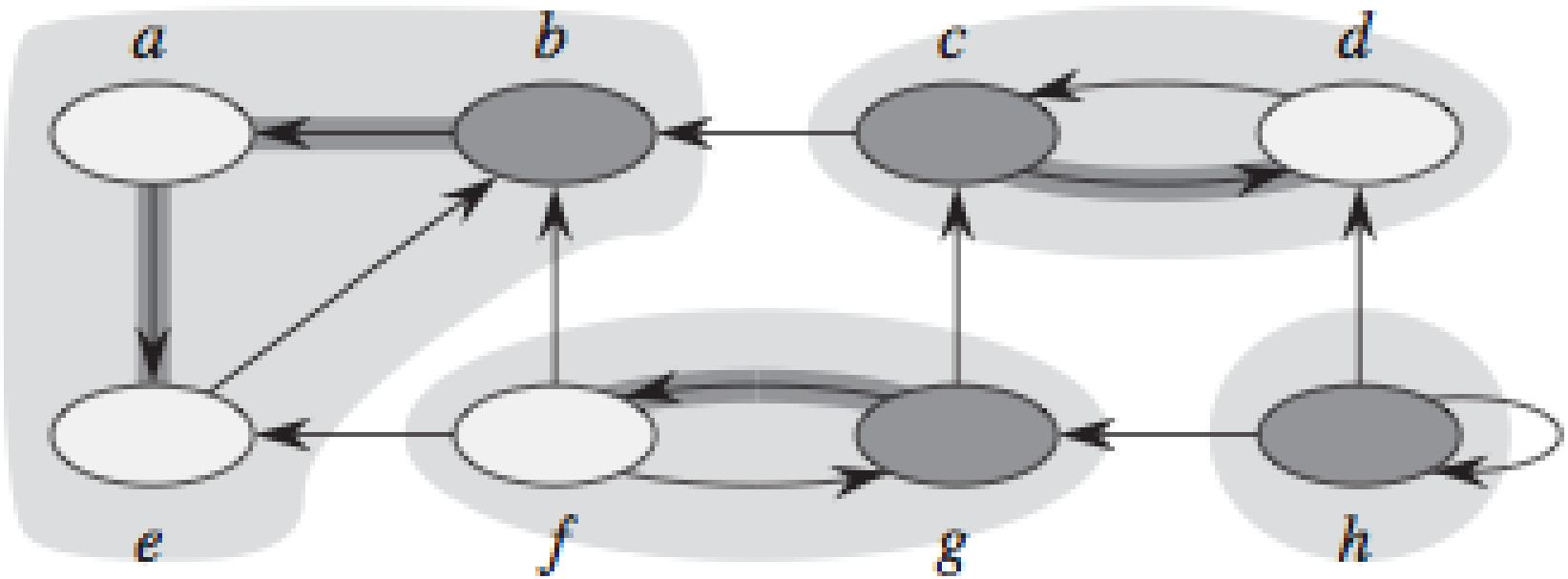
Dado un grafo dirigido $G = (V, E)$, un componente fuertemente conexa o strongly connected component (SCC) de G es un subconjunto C de nodos que cumple que para cada pareja $u, v \in C$ existe un camino de u a v y de v a u y que C es lo más grande posible.



Algoritmo

1. Crear el grafo G y el grafo G_{rev} que es el mismo que G pero con las aristas invertidas.
2. Hacer DFS en el grafo G y generar su "ordenamiento topológico" (incluir un nodo a la lista solo cuando haya visto todos los nodos alcanzables desde él.)
3. Hacer un DFS en el grafo reversado G_{rev} pero hacer las llamadas en el orden del "ordenamiento topológico".
4. Cada llamado a este último DFS halla una componente fuertemente conexa.

Ejemplo



¿Por qué funciona? I

1. Las componentes fuertemente conexas de G son las mismas que las de G_{rev}.
2. Si comprimo los nodos de una misma componente en un solo nodo, quedo con un DAG.
3. Si tengo dos componentes distintas C_1 y C_2 de manera que haya una arista de un nodo de C_1 a un nodo de C_2, entonces todos los nodos de C_1 van a quedar después que los nodos de C_2 en el "ordenamiento topológico" que se hace con el primer DFS.

¿Por qué funciona? II

4. Los nodos que quedan de primeros en el "ordenamiento topológico" son los nodos de una componente C a la cual no llega ninguna arista.
5. En el grafo G_{rev}, de la componente C no sale ninguna arista.
6. Cuando llamo el segundo DFS lo hago desde C y sólo descubro los elementos de C.
7. Cuando llamo el segundo DFS desde otro nodo este puede no tener aristas salientes o tener aristas salientes a C pero como ya descubrí todo en C sólo voy a descubrir lo que hay en la componente de ese nodo

Contenido

Problema 11504 - Dominos

Problema

Hallar el mínimo número de dominós que hay que derribar a mano para que todos los dominós se derriben.

Ideas

Ideas

- ¿Qué pasa con las cadenas de dominós que forman un ciclo? ¿Cuántos necesito máximo para derribarlas?

Ideas

Ideas

- ¿Qué pasa con las cadenas de dominós que forman un ciclo? ¿Cuántos necesito máximo para derribarlas?
- ¿Puedo entonces considerar los ciclos como un sólo dominó? ¿Qué algoritmo estoy utilizando?

Ideas

Ideas

- ¿Qué pasa con las cadenas de dominós que forman un ciclo? ¿Cuántos necesito máximo para derribarlas?
- ¿Puedo entonces considerar los ciclos como un sólo dominó? ¿Qué algoritmo estoy utilizando?
- ¿En el grafo que se forma cuando uno los elementos de una misma componentes cuántos dominós tengo que derribar?

Solución

1. Crear el grafo dirigido y su grafo invertido
2. Hallar la componente fuertemente conexa de cada nodo
3. Hallar cuantas aristas llegan a cada componente conexa
4. Contar cuantas componentes hay a las cuales no lleguen aristas

Variables globales

```
1  // El maximo numero de dominos
2  const int MAXN = 100005;
3  // El grafo
4  vector <int> g[MAXN];
5  // El grafo reversado
6  vector <int> grev[MAXN];
7  // El "ordenamiento topologico" del grafo G
8  vector <int> topo_sort;
9  // La componente fuertemente conexa a la que pertenece cada nodo
10 int scc[MAXN];
11 // El arreglo de visitado para el primer DFS
12 bool seen[MAXN];
13  // El numero de aristas entrantes a cada componente
14 int in[MAXN];
```

DFS

```
1  // DFS donde se halla el ordenamiento topologico
2  void dfs1(int u){
3     seen[u] = true;
4     for (int i = 0; i < g[u].size(); ++i){
5        int v = g[u][i];
6        if (!seen[v]) dfs1(v);
7     }
8     topo_sort.push_back(u);
9  }
10 // DFS donde se hallan las componentes
11 void dfs2(int u, int comp){
12    scc[u] = comp;
13    for (int i = 0; i < grev[u].size(); ++i){
14       int v = grev[u][i];
15       if (scc[v] == -1) dfs2(v, comp);
16    }
17 }
```

Main I

```
1  int main(){
2     int cases; cin >> cases;
3     while (cases--){
4        int n, m;
5        cin >> n >> m;
6
7        // Limpiar las variables entre caso y caso
8        for (int i = 0; i <= n; ++i){
9           g[i].clear();
10          grev[i].clear();
11          topo_sort.clear();
12          scc[i] = -1;
13          seen[i] = false;
14          in[i] = 0;
15       }
16
17
18
```

Main II

```
19          // Crear el grafo y el grafo reversado
20          for (int i = 0; i < m; ++i){
21              int u, v; cin >> u >> v;
22              u--; v--;
23              g[u].push_back(v);
24              grev[v].push_back(u);
25          }
26
27          // Llamar el primer dfs
28          for (int i = 0; i < n; ++i){
29              if (!seen[i]) dfs1(i);
30          }
31          reverse(topo_sort.begin(), topo_sort.end());
32          // Llamar el segundo dfs
33          int comp = 0;
34          for (int i = 0; i < n; ++i){
35              int u = topo_sort[i];
36              if (scc[u] == -1) dfs2(u, comp++);
37          }
```

Main III

```
38
39          // Ver cuantas aristas entrantes tiene cada componente
40          for (int u = 0; u < n; ++u){
41              for (int i = 0; i < g[u].size(); ++i){
42                  int v = g[u][i];
43                  if (scc[u] != scc[v]) in[scc[v]]++;
44              }
45          }
46
47          // Sumar las componentes que tienen 0 aristas entrantes
48          int count = 0;
49          for (int u = 0; u < comp; ++u){
50              if (in[u] == 0) count++;
51          }
52          cout << count << endl;
53      }
54       return 0;
55  }
```

Contenido

Map

Map

Un mapa es un contenedor que guarda parejas de elementos. El primer elemento de la pareja (key) sirve para identificarla y el segundo elemento (mapped value) es el valor asociado a la llave.

Declaración

```
#include <map>
map <tipo_dato_key, tipo_dato_value> nombre;
```

Ejemplos:

```
map <string, int> m;
map <char, int> char2int;
```

Acceso a elementos de un mapa

Los elementos de un mapa se llaman por su llave así:

```
map <string, int> m;
m["Hola"] = 3;
int a = m["Cangrejo"];
```

Para ingresar un elemento se puede hacer así:

```
if (m.count["Nuevo"] == 0) m["Nuevo"] = 123;
```

Con la función `count` se busca cuántas veces está el elemento con la llave `"Nuevo"`. Si el elemento no está, cuando accedemos a él con `[]` éste se crea automáticamente.
Cuando accedo con `[]` a un elemento del mapa que no existe este se crea con valores por defecto. Para strings es el string vacío `""` y para enteros es el número 0.

Complejidad del mapa

Complejidad

- Insertar / acceder un elemento al mapa es $O(log\, n \times k)$ donde n es el número de elementos en el mapa y k es el tiempo que toma comparar dos llaves del mapa.
- Comparar dos enteros es $O(1)$, comparar dos strings es $O(m)$ donde m es la longitud de los strings.
- Insertar / acceder un elemento a un mapa con llaves strings es $O(log\, n \times m)$ donde n son los elementos del mapa y m es la longitud del string.

Nota

Los elementos de un mapa se almacenan en orden de acuerdo a una función de comparación. Por defecto la función de comparación es la de menor, es decir que los elementos se almacenan de menor a mayor.

Recorrer un mapa

Para recorrer un mapa es necesario usar iteradores.

```
1  #include <iostream>
2  #include <map>
3  using namespace std;
4
5  int main(){
6     map <string, int> m;
7     m["b"] = 4;
8     m["bc"] = 1;
9     m["a"] = 3;
10    map <string, int> :: iterator it;
11    for (it = m.begin(); it != m.end(); it++){
12       cout << "( " << it->first << " " << it->second << " ) ";
13       // cout << "( " << (*it).first << " " << (*it).second << "
       ) ";
14    }
15    return 0;
16 }
17 // La funcion imprime ( a 3 ) ( b 4 ) ( bc 1 )
```

Otras funciones en el mapa

Otras funciones que se pueden hacer con el mapa son:

- Recorrerlo al revés con `rbegin` y `rend`
- Obtener el tamaño con `size`
- Borrar el contenido con `clear`
- Insertar elementos (si no están antes) con `emplace`
- Borrar elementos con `erase`
- Buscar un elemento con `find`

Para más información mirar
`http://www.cplusplus.com/reference/map/map/`

Contenido

Set

Set

Un set es un contenedor que guarda conjuntos, es decir grupos de elementos iguales donde cada elemento aparece una sola vez. Los elementos de un set se almacenan en orden de acuerdo a una función de comparación. Por defecto la función de comparación es la de menor, es decir que los elementos se almacenan de menor a mayor.

Declaración

```
#include <set>
set <tipo_dato> nombre;
```

Ejemplos:

```
set <string> s;
set <int> amigos;
```

Operaciones sobre un set

Sobre un set se pueden hacer las siguientes operaciones.

- **insert** Inserta un elemento al set. Ejemplo: `amigos.insert(9)`
- **count** Cuenta cuántas veces aparece un elemento (0 o 1 vez). Ejemplo: `amigos.count(3)`
- **find** Retorna un iterador al lugar donde está el elemento. Ejemplo: `amigos.find(3)`
- **erase** Elimina un elemento del set. Ejemplo: `amigos.erase(amigos.find(3))`

Todas las operaciones anteriores tienen una complejidad $O(log\, n \times k)$ donde n es el número de elementos del set y k es el tiempo que toma comparar dos elementos.
Para más información mirar
`http://www.cplusplus.com/reference/set/set/`

Recorrer un set

Para recorrer un set es necesario usar iteradores.

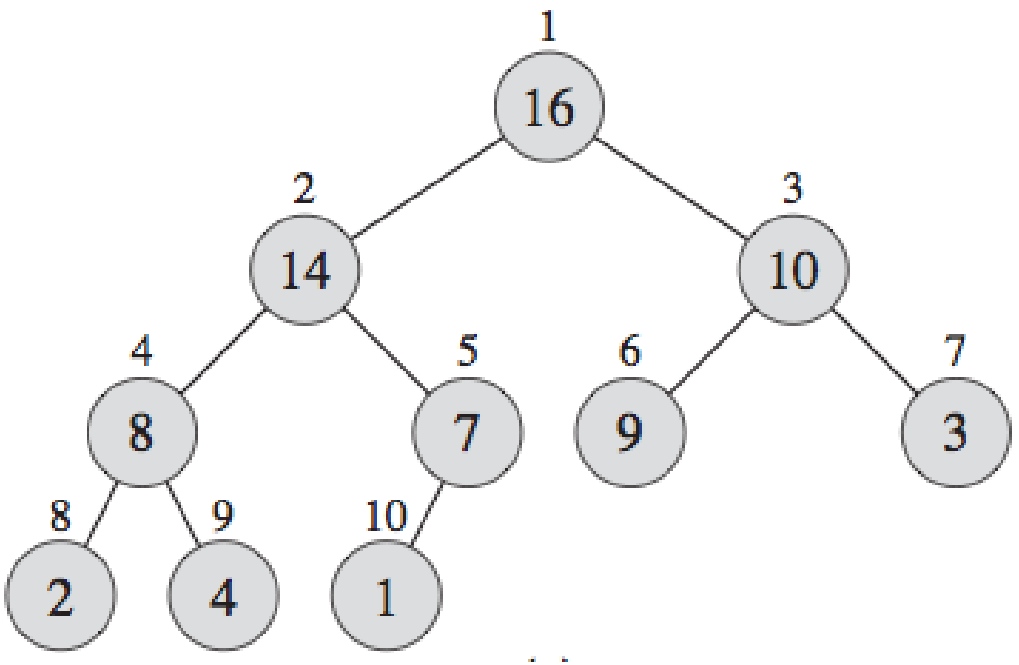
```
1  #include <iostream>
2  #include <set>
3  using namespace std;
4
5  int main(){
6      set <int> s;
7      s.insert(4);
8      s.insert(-1);
9      s.insert(3);
10     s.insert(4);
11     set <int> :: iterator it;
12     for (it = s.begin(); it != s.end(); it++){
13         cout << *it << " ";
14     }
15     return 0;
16 }
17 // La funcion imprime -1 3 4
```

Contenido

Heap

Heap

Un heap es una estructura de datos en forma de árbol binario en la que cada padre tiene un valor mayor o igual al de sus dos hijos



Representación en C++

Un heap se puede representar en C++ como una cola de prioridades así: `#include <queue>`

```
priority_queue <tipo_dato> nombre;
```

Ejemplos:

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

Para más información mirar: `http://www.cplusplus.com/reference/queue/priority_queue/`

Operaciones

La cola de prioridades (heap) soporta las siguientes operaciones

push	Inserta un elemento
pop	Extrae un elemento
top	Retorna el máximo elemento de la cola (heap)
size	Retorna el tamaño de la cola (heap)

La cola de prioridades se ordena de acuerdo a la función de ordenamiento $<$ (menor que) por lo que retorna el elemento con el cual todos los demás comparan menor que él, es decir, el mayor elemento.

Contenido

Tarea

Tarea

Resolver los problemas de
`http://contests.factorcomun.org/contests/51`

Problema A

- ¿Qué estructura de datos de las que han aprendido se puede usar para almacenar los datos?
- Mirar que los nombres pueden tener espacios (usar getline).
- Recordar que cuando leo con cin el cursor queda justo después del elemento donde leí, si luego hago getline me termina de leer esa línea.
- Para imprimir con precisión de 4 cifras decimales usar `printf("%.4lf", percent);` donde `percent` es la variable a imprimir.

Problema B

- ¿Qué estructura de datos de las que han aprendido se puede usar para verificar que cuando cierro un paréntesis / corchete sí lo acabe de abrir?
- La cadena puede tener espacios (usar getline)
- La cadena puede ser vacía
- Ensayar los siguientes casos de prueba

Entrada	Salida
6	
([])	Yes
	Yes
(	No
(]	No
)(	No
([)]	No

Problema C

- Leer la implementación y tratar de entenderla
- Hacer su propia implementación

Problema D

- Mmmm ¿Mínimo número de pasos? Me suena conocido ¿Qué algoritmo es el que hay que usar?
- ¿Cómo se construye el grafo? ¿Cuándo uno dos nodos (palabras)?
- ¿Qué estructura de datos puedo usar para que un string lo pueda representar como un entero y poder hace el algoritmo normal?
- Usar getline y stringstream para leer la entrada.