List

- Plantilla que permite disponer de listas doblemente enlazadas.
- Métodos más importantes:
  - begin() -> iterador al primer elemento
  - front() -> devueve el contenido del primer elemento
  - end() -> iterador tras el último elemento
  - back() ->devuelve el contenido del último elemento
  - push_back(const value_type &v)
  - push_front(const value type &v)
  - pop_back()
  - pop_front()
  - erase (itertor position): tras borrarlo, ese iterador ya no se puede usar. La función erase devuelve un iterador al siguiente elemento, así que podemos reasignar
  - Insert(iterator position, const value_type &v**): inserción en O(1) → Esta es la ventaja frente al vector, claro que de alguna forma tenemos que tener acceso a los iteradores (muchas veces están en un árbol, o alguna otra estructura a la que vamos a poder acceder rápido)**

List

```
1  // erasing from list
2  #include <iostream>
3  #include <list>
4
5  int main ()
6  {
7    std::list<int> mylist;
8    std::list<int>::iterator it1,it2;
9
10   // set some values:
11   for (int i=1; i<10; ++i) mylist.push_back(i*10);
12
13                              // 10 20 30 40 50 60 70 80 90
14   it1 = it2 = mylist.begin(); // ^^
15   advance (it2,6);           // ^                 ^
16   ++it1;                     //    ^              ^
17
18   it1 = mylist.erase (it1);  // 10 30 40 50 60 70 80 90
19                              //    ^           ^
20
21   it2 = mylist.erase (it2);  // 10 30 40 50 60 80 90
22                              //    ^           ^
23
24   ++it1;                     //       ^        ^
25   --it2;                     //       ^     ^
26
27   mylist.erase (it1,it2);    // 10 30 60 80 90
28                              //        ^
29
30   std::cout << "mylist contains:";
31   for (it1=mylist.begin(); it1!=mylist.end(); ++it1)
32     std::cout << ' ' << *it1;
33   std::cout << '\n';
34
35   return 0;
36 }
```

- Ojo, advance es O(N)
- Erase devuelve iterador que apunta al siguiente elemento del que borramos
- Si no reasignamos, el iterador anterior es inválido

Deque

- Plantilla que actúa como un contenedor con las siguientes características:
  - Puede ampliarse o reducirse por ambos lados
  - Se puede acceder a cualquier elemento en tiempo constante usando el operador []

- Los métodos son similares a los de list, aunque en este caso además tiene el operador [], ya que se basa en iteradores de acceso aleatorio.

- No podemos insertar en medio en O(1)

Otros tipos

- Las tres plantillas anteriores son los contenedores básicos, y algunos otros tipos de datos los usan.

- De hecho, en tipos de datos como pilas y colas, podemos elegir al construirlo con qué tipo de contenedor queremos que funcione internamente, aunque casi no hay diferencia en rendimiento.

- Los tipos que hemos visto, los podríamos usar como pilas y colas. Sin embargo, para mejorar la legibilidad del código, hay tipos específicos.

stack

- Es la plantilla que implementa el tipo “pila”.
- Métodos más importantes:
  - push: insertar elemento
  - pop: eliminar elemento
  - size: número de elementos en la pila
  - empty: devuelve true si el número de elementos es 0
  - top: devuelve el primer elemento de la pila

```
1  // stack::push/pop
2  #include <iostream>       // std::cout
3  #include <stack>          // std::stack
4
5  int main ()
6  {
7    std::stack<int> mystack;
8
9    for (int i=0; i<5; ++i) mystack.push(i);
10
11   std::cout << "Popping out elements...";
12   while (!mystack.empty())
13   {
14      std::cout << ' ' << mystack.top();
15      mystack.pop();
16   }
17   std::cout << '\n';
18
19   return 0;
20 }
```

queue

- Es la plantilla que implementa el tipo “cola”.
- Métodos más importantes:
  - push: insertar elemento
  - pop: quitar el siguiente elemento
  - front: acceder al siguiente elemento

Ejercicio

a) Crear una pila, que además de tener el push, pop y top, tenga el método minElement, que devuelva el mínimo elemento presente en la pila.

b) Crear una cola, que además de tener el push, pop y top, tenga el método minElement, que devuelva el mínimo elemento presente en la cola.

Queremos que la complejidad de hacer N operaciones sea O(N)

Ejercicio

a) Crear una pila, que además de tener el push, pop y top, tenga el método minElement, que devuelva el máximo elemento presente en la pila.

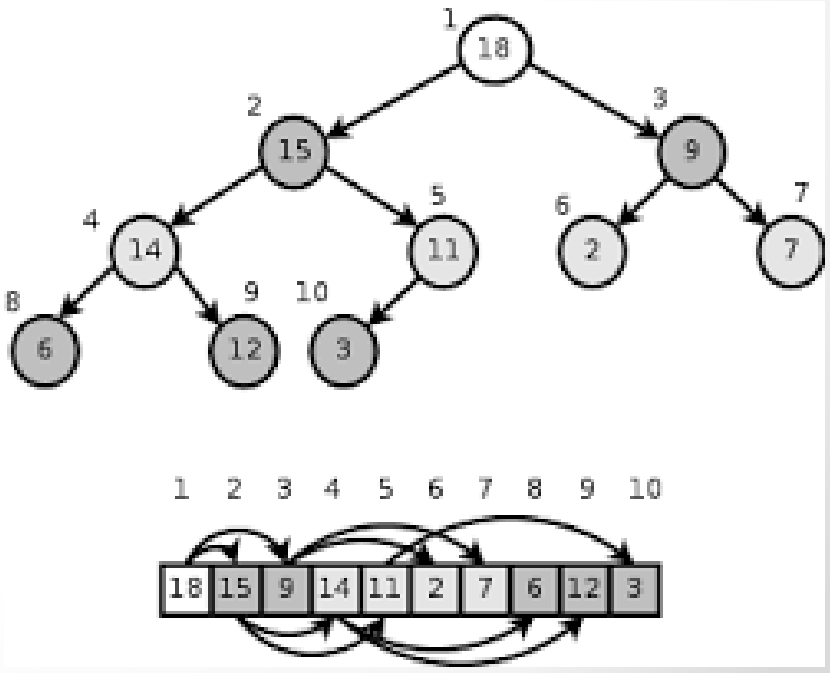
b) Crear una cola, que además de tener el push, pop y top, tenga el método minElement, que devuelva el máximo elemento presente en la cola.
   - Pista: ¿Cómo podemos crear una cola a partir de una o varias pilas?
   - Ver libro.

Colas de prioridad

- Una cola de prioridad es una estructura de datos en las que, en base a un orden dado por algún comparador, siempre "saca" primero al elemento mayor, es decir, al de mayor prioridad. Obviamente, donde dice mayor, podría decir menor.
- ¿Cómo lo implementamos?

Heap

- Un heap es una estructura de datos lineal (un arreglo) que puede ser visualizado como un árbol binario casi completo (va rellenando siempre por niveles de izquierda a derecha).
- En los heaps se cumple que cada elemento es mayor (en algunos libros se explica con menor) que sus hijos.
- El primer elemento es la raíz
- Considerando que el vector empieza en 1:
  - Left(i): 2 * i
  - Right(i): 2 * i + 1
  - Parent(i): i/2



Heap / Colas de prioridad

- ¿Cómo usamos un Heap para implementar una cola de prioridad?

Heap / Colas de prioridad

- ¿Cómo usamos un Heap para implementar una cola de prioridad?
- En una cola de prioridad necesitamos tres operaciones:
  - Acceder al elemento más grande
  - Retirar el elemento más grande
  - Insertar nuevo elemento

Heap/Colas de prioridad

- Para retirar el elemento más grande se hace lo siguiente:
  - El último elemento del arreglo sustituye al primero.
  - Con esto, tenemos que los nuevos datos pueden haber perdido la propiedad del heap. Se cumple, el subárbol izquierdo tiene la propiedad de heap, el subárbol derecho también tiene la propiedad de Heap. Desarrollamos un método Heapify, que siempre recibe una posición para la que se sabe que se cumple dicha situación y lo que debe hacer este método es recuperar la propiedad de Heap.
  - Entonces, se llama a Heapify(Arreglo, 1)
  - ¿Cómo debería funcionar el método Heapify?

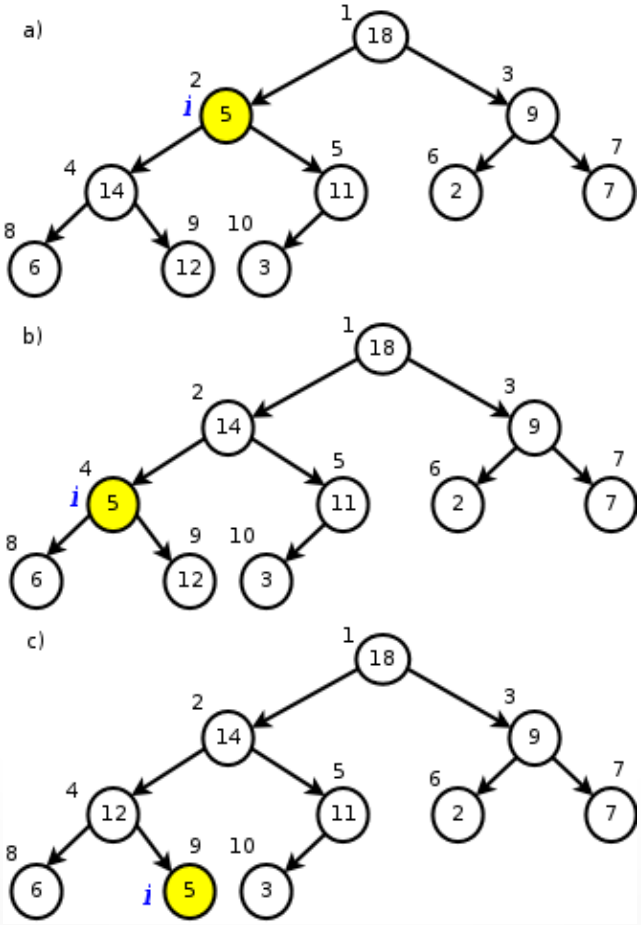
Heap/Colas de prioridad

- Para retirar el elemento más grande se hace lo siguiente:
  - El último elemento del arreglo sustituye al primero.
  - Se llama a Heapify(Arreglo, 1)

```
MAX-HEAPIFY(A, i)
 1  l ← LEFT(i)
 2  r ← RIGHT(i)
 3  if l ≤ heap-size[A] and A[l] > A[i]
 4     then largest ← l
 5     else largest ← i
 6  if r ≤ heap-size[A] and A[r] > A[largest]
 7     then largest ← r
 8  if largest ≠ i
 9     then exchange A[i] ↔ A[largest]
10          MAX-HEAPIFY(A, largest)
```

Heap/Colas de prioridad

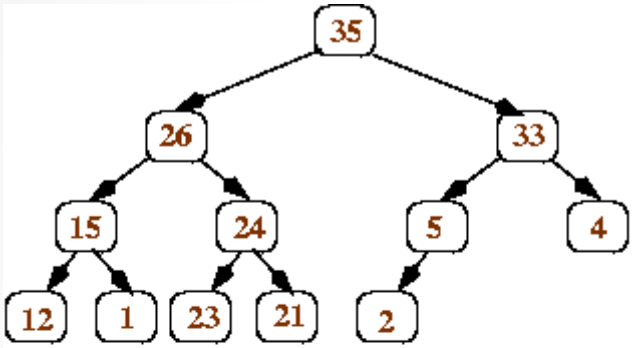
- Ejemplo MAX-Heapify(A, 2):



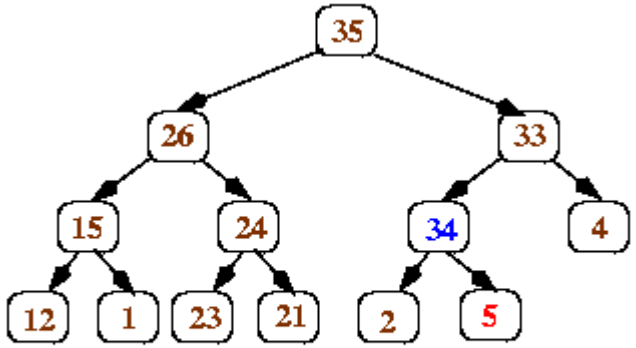
Heap/Colas de prioridad

- Para insertar un nuevo elemento:
  - Lo pone como último elemento del heap
  - Si es mayor que su padre, los intercambia, y ahora hace la misma operación con el padre
  - Prosigue hasta que el elemento no sea mayor que el siguiente padre, o hasta llegar al nodo raíz
  - Ejemplo de inserción, siguiente diapositiva.

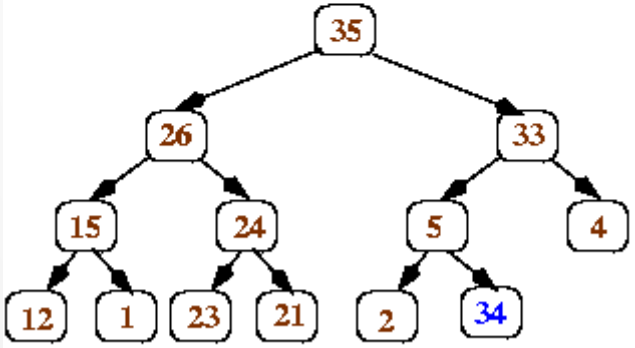
Head/Colas de prioridad



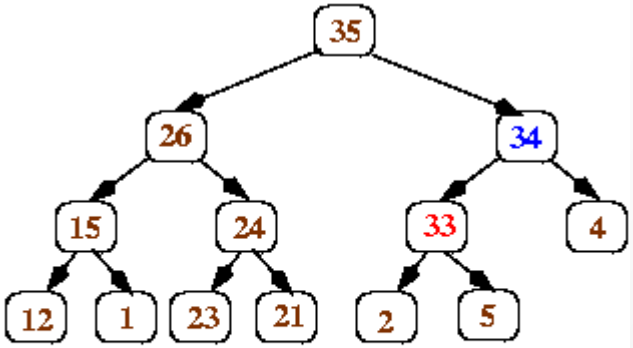
original heap



step 2 (swap with parent)



step 1 (add 34 as rightmost leaf)



step 2 again (swap with parent)

All done!

Complejidad

- En una cola de prioridad implementada con un Heap, ¿cuál es la complejidad de una operación de inserción?
- ¿Cuál es la complejidad de acceder al elemento más grande?
- ¿Cuál es la complejidad de eliminar el elemento más grande?

- ¿Cuál es la complejidad de crear un heap a partir de un arreglo de datos no ordenados?

Creación de un heap

- Como entrada tenemos un arreglo de datos en cualquier orden.
- El procedimiento max-heapify visto anteriormente asume que ambos hijos tienen la propiedad de ser heap.
- Para nuestro caso, todos los nodos hoja son heaps, con lo que para convertir el arreglo completo en un heap, hay que llamar a max-heapify para el resto, empezando desde los que tienen índices mayores.
- ¿Complejidad?

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

Creación de un heap

- En un principio, podría parecer que es complejidad O(n log n). Sin embargo, no es así.
- Un heap con n elementos tiene altura h = log2 (n). ¿Cómo se define la altura de un árbol? Si el árbol es completo:

$$n = 2^{h+1} - 1,$$

- A distancia j de los nodos hoja, hay (n +1) / (2^(j+1))

$$T(n) = \sum_{j=0}^{h} j2^{h-j} = \sum_{j=0}^{h} j\frac{2^h}{2^j}. \qquad T(n) = 2^h \sum_{j=0}^{h} \frac{j}{2^j}.$$

$$\sum_{j=0}^{\infty} x^j = \frac{1}{1-x}. \qquad \sum_{j=0}^{\infty} jx^j = \frac{x}{(1-x)^2}, \qquad \sum_{j=0}^{\infty} \frac{j}{2^j} = \frac{1/2}{(1-(1/2))^2} = \frac{1/2}{1/4} = 2.$$

- Complejidad O(n)

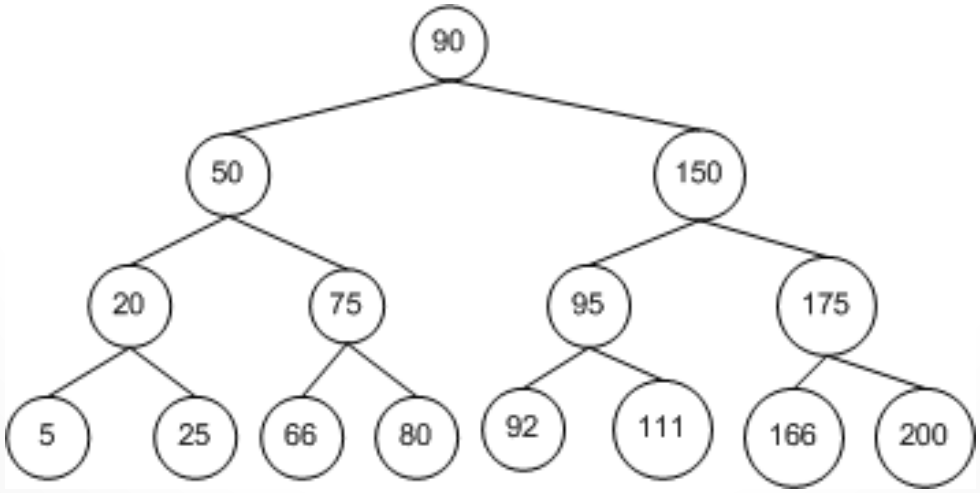
Cola Prioridad C++

```cpp
template<typename T> void print_queue(T& q) {
    while(!q.empty()) {
        std::cout << q.top() << " ";
        q.pop();
    }
    std::cout << '\n';
}

int main() {
    std::priority_queue<int> q;

    for(int n : {1,8,5,6,3,4,0,9,7,2})
        q.push(n);

    print_queue(q);

    std::priority_queue<int, std::vector<int>, std::greater<int> > q2;

    for(int n : {1,8,5,6,3,4,0,9,7,2})
        q2.push(n);

    print_queue(q2);

    // Using lambda to compare elements.
    auto cmp = [](int left, int right) { return (left ^ 1) < (right ^ 1);};
    std::priority_queue<int, std::vector<int>, decltype(cmp)> q3(cmp);

    for(int n : {1,8,5,6,3,4,0,9,7,2})
        q3.push(n);

    print_queue(q3);

}
```

```
9 8 7 6 5 4 3 2 1 0
0 1 2 3 4 5 6 7 8 9
8 9 6 7 4 5 2 3 0 1
```

Binary Search Tree

- Un árbol de búsqueda binario, es un árbol binario que cumple la propiedad siguiente:
  - Para cualquier nodo, todos los elementos de su sub-árbol izquierdo son menores o iguales que el propio nodo.
  - Para cualquier nodo, todos los elementos de su sub-árbol derecho son mayores o igual que el propio nodo.



Binary Search Tree

- Si queremos recorrer un BST en orden:

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2     then INORDER-TREE-WALK(left[x])
3          print key[x]
4          INORDER-TREE-WALK(right[x])
```

- Complejidad de insertar un elemento: O(h)
- Complejidad de buscar un elemento: O(h)
- Complejidad de borrar elemento (incluyendo la búsqueda): O(h). ¿Cómo lo hacemos el borrado?

Binary Search Tree

- Si queremos recorrer un BST en orden:

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2     then INORDER-TREE-WALK(left[x])
3          print key[x]
4          INORDER-TREE-WALK(right[x])
```

- Complejidad de insertar un elemento: O(h)
- Complejidad de buscar un elemento: O(h)
- Complejidad de borrar elemento (incluyendo la búsqueda): O(h). ¿Cómo lo hacemos el borrado? Si no tiene hijos, o un hijo, es trivial. Si tiene 2 hijos, podemos colocar en esa posición al menor valor del subárbol derecho, y borrar recursivamente ese nodo (el cual tendrá sólo un hijo o ningún hijo)

Binary Search Tree

- Otras operaciones:

```
TREE-INSERT(T, z)
 1  y = NIL
 2  x = T.root
 3  while x ≠ NIL
 4        y = x
 5        if z.key < x.key
 6              x = x.left
 7        else x = x.right
 8  z.p = y
 9  if y == NIL
10        T.root = z          // tree T was empty
11  elseif z.key < y.key
12        y.left = z
13  else y.right = z
```

- Con el fin de que las operaciones anteriores no sean costosas, hay que mantener el árbol balanceado. Se hacen modificaciones para asegurar el correcto balanceo AVL trees, red-black trees, splay trees, **treap**.

set

- La plantilla set de la STL es un contenedor de datos que permite inserción, borrado y búsqueda en O(log n), es decir, igual que un árbol balanceado.
- De hecho, aunque el estándar no indica que estructura de datos interna hay que usar, la mayoría de las implementaciones usan RB-Trees.
- Los datos quedan ordenados, con lo que podemos hacer un recorrido de los datos en orden en O(n)
- Debemos definir un comparador del objeto que va a contener el conjunto, o sobreescribir operator<
- Métodos importantes:
  - begin()
  - end()
  - insert()
  - count()
  - erase() -> por valor o con iterador
  - clear()
  - size()

set

```
 1 #include <set>
 2 #include <string>
 3 #include <iostream>
 4
 5 using namespace std;
 6
 7 class Person {
 8   public:
 9     Person(const string p, const int &a, const string &n):passport(p),age(a),name(n){}
10     bool operator<(const Person &a) const { return passport < a.passport; }
11     string getName() const { return name; }
12   private:
13     string passport;
14     int age;
15     string name;
16     //...
17 };
18
19 int main(){
20   set <Person> database;
21   Person p1(string("2"), 25, string("Carlos"));
22   Person p2(string("1"), 25, string("Pedro"));
23   Person p3(string("1"), 25, string("Silvia"));
24   database.insert(p1);
25   database.insert(p2);
26   database.insert(p3);
27   cout << database.size() << endl;
28   for (set<Person>::iterator it = database.begin(); it != database.end(); it++){
29     cout << it->getName() << endl;
30   }
31   cout << (database.count(p3)) << endl;
32 }
```

map

- La plantilla map permite crear contenedores asociativos formados por un tipo clave y un tipo al que se mapea dicho valor.
- La complejidad algorítmica asociada a las diferentes operaciones es igual a la del set.
- De hecho normalmente, al igual que los set, se implementan con RB-Trees.
- Los métodos son similares a los del set, pero en este caso para buscar el elemento se utiliza la clave. Además, está definido operator[], que recibe la clave.

map

```
 1 #include <map>
 2 #include <string>
 3 #include <iostream>
 4
 5 using namespace std;
 6
 7 int main(){
 8   map <string, int> foodQuality;
 9   foodQuality["enchiladas"] = 7;
10   foodQuality["salmon"] = 7;
11   foodQuality["albondigas"] = 8;
12
13   cout << foodQuality["albondigas"] << endl;
14   cout << foodQuality["cerdo"] << endl;
15
16   for (map<string, int>::iterator it = foodQuality.begin(); it != foodQuality.end(); it++){
17     cout << it->first << "->" << it->second << endl;
18   }
19 }
```

map

```cpp
#include <map>
#include <string>
#include <iostream>

using namespace std;

int main(){
  map <string, int> foodQuality;
  foodQuality["enchiladas"] = 7;
  foodQuality["salmon"] = 7;
  foodQuality["albondigas"] = 8;

  cout << foodQuality["albondigas"] << endl;
  cout << foodQuality["cerdo"] << endl;

  for (map<string, int>::iterator it = foodQuality.begin(); it != foodQuality.end(); it++){
    cout << it->first << "->" << it->second << endl;
  }
}
```

- **Fallo típico:** Al acceder con el operador [], si el elemento indexado no existe se crea y se le da el valor por defecto. Para comprobar si existe o no, debemos usar el método count.

multimap

- En multimap, a diferencia de en map se soportan valores repetidos.
- En un map, si se hace una asignación con una clave que ya existía, el valor antiguo desaparece.
- En un multimap, no se puede usar el operador [], sino que hay que usar insert para insertar, find para obtener un valor asociado a una clave e, equal_range para obtener todos los asociados a una clave.

multimap

```cpp
// multimap::equal_range
#include <iostream>
#include <map>

int main ()
{
  std::multimap<char,int> mymm;

  mymm.insert(std::pair<char,int>('a',10));
  mymm.insert(std::pair<char,int>('b',20));
  mymm.insert(std::pair<char,int>('b',30));
  mymm.insert(std::pair<char,int>('b',40));
  mymm.insert(std::pair<char,int>('c',50));
  mymm.insert(std::pair<char,int>('c',60));
  mymm.insert(std::pair<char,int>('d',60));

  std::cout << "mymm contains:\n";
  for (char ch='a'; ch<='d'; ch++)
  {
    std::pair <std::multimap<char,int>::iterator, std::multimap<char,int>::iterator> ret;
    ret = mymm.equal_range(ch);
    std::cout << ch << " =>";
    for (std::multimap<char,int>::iterator it=ret.first; it!=ret.second; ++it)
      std::cout << ' ' << it->second;
    std::cout << '\n';
  }

  return 0;
}
```

multiset

- Es una extensión de set que soporta valores repetidos.
- Su uso es similar, pero ahora count devuelve el número de veces que un objeto está en el multiset.
- Al hacer un erase de un valor, se borran todas las apariciones de ese valor.
- Si sólo queremos borrar una aparición, hay que realizar un find para obtener un iterador, y hacer un erase de ese iterador.

multiset

```cpp
// erasing from multiset
#include <iostream>
#include <set>

int main ()
{
  std::multiset<int> mymultiset;
  std::multiset<int>::iterator it;

  // insert some values:
  mymultiset.insert (40);                          // 40
  for (int i=1; i<7; i++) mymultiset.insert(i*10);   // 10 20 30 40 40 50 60

  it=mymultiset.begin();
  it++;                                            //    ^

  mymultiset.erase (it);                           // 10 30 40 40 50 60

  mymultiset.erase (40);                           // 10 30 50 60

  it=mymultiset.find (50);
  mymultiset.erase ( it, mymultiset.end() );       // 10 30

  std::cout << "mymultiset contains:";
  for (it=mymultiset.begin(); it!=mymultiset.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

unordered_set (hash)

- Para las 4 plantillas anteriores (set, map, multiset, multimap), la versión c++11 incluye versiones unordered.
- Su uso es similar, pero en lugar de usar RB Trees, utiliza tablas Hash.
- Con los tipos base, su uso es similar, y simplemente hay que poner unordered_set < int >
- Esto usa una función hash por defecto, que puede ser buena para nuestro caso o no.
- Otra opción (obligatoria para tipos propios) es que el programador defina la función hash. Esto se puede hacer de varias formas.

unordered_set (hash)

```cpp
#include <unordered_set>
#include <set>
#include <iostream>

using namespace std;

class Point {
  public:
    Point(int x, int y):x(x),y(y){};
    inline int getX() const { return x;}
    inline int getY() const { return y;}
    bool operator==(const Point &p) const { return ((x == p.x) && (y == p.y)); }//Required for unordered_set
    bool operator<(const Point &p) const { return ((x != p.x)?(x < p.x):(y < p.y)); }//Required for set
  private:
    int x, y;
};

namespace std{
  template<> struct hash<Point>{
    size_t operator()(Point const &p) const noexcept {
      return 100000 * p.getX() + p.getY();//0.021s
      //return 1; //1 min. 27 s
    }
  };
}

int main(){
  unordered_set<int> myIntHash;
  myIntHash.insert(5);
  myIntHash.insert(8);
  unordered_set<Point> myPointHash;
  //set<Point> myPointHash;//0.054s
  for (int i = 0; i < 100000; i++){
    Point p(0, i);
    myPointHash.insert(p);
  }
  for (int i = 0; i < 100000; i++){
    Point p(0, i);
    bool found = (myPointHash.count(p));
  }
}
```

unordered_set (hash)

- El número de buckets usado en unordered_set se va fijando de forma dinámica, según el número de elementos que se han insertado.
- Por defecto se usa una carga máxima igual a uno, lo que significa que el número de buckets es al menos igual al número de elementos.
- Cuando el número de elementos supera al número de buckets, se incrementa (pero no de 1 en 1, sino al menos duplicando habitualmente), pues es un proceso costoso en el que hay que reasignar a todos los elementos.
- Con el método reserve, podemos fijar el número de buckets en un determinado momento. Igualmente, podemos cambiar la carga que queremos.