Estructuras de datos.

Material de referencia.

Índice

1. Policy based data structures.

La STL de GNU C++ implementa algunas estructuras de datos adicionales.

1.1. Contenedores basados en árboles.

Probablemente la más interesante de todas, es el árbol. Para poder utilizarlo debemos añadir antes las siguientes librerías:

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
```

Los contenedores basados en árboles tienen la siguiente declaración:

```
tree<Key, Mapped, Cmp_Fn = std::less<Key>, Tag = rb_tree_tag, node_update =
    null_node_update, Allocator = std::allocator<char> >
```

donde

- `Key` es el tipo de las llaves.
- `Mapped` es el tipo de los datos mapeados. Esto se asemeja bastante a un `map<key, T>`. Si en su lugar lo llenamos con `null_type`, obtenemos un contenedor similar a un `set<T>`.
- `Cmp_Fn` es una función de comparación de llaves.
- `Tag` especifica la estructura de datos a utilizar. Debe ser alguno de `rb_tree_tag` (red-black tree), `splay_tree_tag` (splay tree) o `ov_tree_tag` (ordered-vector tree).
- `node_update` especifica como actualizar los invariantes de cada nodo. Por defecto este campo está lleno con `null_node_update`, es decir, no hay información adicional para los vértices.

Los contenedores basados en árboles soportan las mismas funciones que `set` y `map`. Además, soportan dos funciones adicionales:

```
A.split(T key, tree B);
A.join(tree B);
```

La función `split` mueve todos los nodos con llaves mayores que `key` del árbol `A` al árbol `B`. La función `join`, por el contrario, mueve todos los nodos del árbol `B` al árbol `A`, siempre y cuando los rangos no se traslapen. Ambas funciones tienen complejidad poli-logarítmica en el caso de árboles rojo-negro.

Además de los iteradores convencionales de `set` y `map`, los contenedores basados en árboles implementan dos tipos de iteradores adicionales, `const_node_iterator` y `node_iterator`, con los cuales podemos recorrer el árbol.