Estructuras de datos.

Material de referencia.

Índice

1. Policy based data structures.

La STL de GNU C++ implementa algunas estructuras de datos adicionales. 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`. Si en cambio lo llenamos con `null_type`, obtenemos un contenedor similar a un `set`.
- `Cmp_Fn` es una función de comparación de llaves. Debe declararse en forma de `struct` con el operador `()` sobrecargado.
- `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. El valor por defecto, `null_node_update`, indica que los nodos no guardan información adicional.

Split y join

Los contenedores basados en árboles soportan las mismas funciones que `set` y `map`, junto con 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. En el caso de árboles rojo-negro, ambas funciones tienen complejidad poli-logarítmica.

Iteradores de nodo

Además de los iteradores convencionales de `set` y `map`, los contenedores basados en árboles implementan un tipo de iterador adicional, `node_iterator`, el cual nos permite recorrer el árbol. Así por ejemplo, las funciones

```
Tree::node_iterator root = A.node_begin();
Tree::node_iterator nil = A.node_end();
```

regresan un iterador de nodo correspondiente a la raíz y nodos nulos del árbol. Cada iterador de nodo incluye dos funciones miembro `get_l_child()` y `get_r_child()` que regresan los iteradores de nodos correspondientes a los hijos izquierdo y derecho.

Podemos hacer la conversión entre iteradores convencionales e iteradores de nodo de la siguiente manera:

```
1 it = *nd_it;
2 nd_it = it.m_p_nd;
```

La primera línea regresa el `iterator` correspondiente a un `node_iterator` mientras que la segunda realiza justamente lo contrario.

Actualización de nodos

Por otro lado, recordemos que `node_update` especifica la información adicional que guardará cada nodo así como la forma en que se actualiza. Este debe ser declarado en forma de `struct`, el cual debe definir en su interior el tipo del dato adicional como `metadata_type`, y sobrecargar el operador `()` especificando cómo se actualizará cada nodo.

El operador `()` será llamado internamente cada vez que sea necesario, recibiendo como parámetros el nodo a actualizar y el nodo nulo. Las llamadas siempre se realizarán desde las hojas hasta la raíz. De esta manera, al actualizar la información de un nodo, se presupone que la información de sus hijos ya está actualizada.

Finalmente, cada iterador de nodo tiene una función miembro `get_metadata()` que regresa una referencia al dato adicional de ese nodo. Sin embargo, al ser una variable constante, debemos hacerle antes un `const_cast<metadata_type &>` para modificarlo.

Por ejemplo, si queremos que cada nodo guarde el tamaño del sub-árbol correspondiente, podemos definir la etiqueta `size_node_update` de la siguiente manera:

```
1  template<typename node_const_iterator, typename node_iterator, typename Cmp_Fn
       , typename Allocator>
2  struct size_node_update {
3      typedef int metadata_type;
4
5      void operator() (node_iterator nd_it, node_const_iterator nil) {
6          int lsize = 0, rsize = 0;
7          if (nd_it.get_l_child() != nil)
8              lsize = nd_it.get_l_child().get_metadata();
9          if (nd_it.get_r_child() != nil)
10             rsize = nd_it.get_r_child().get_metadata();
11         const_cast<int &>(nd_it.get_metadata()) = lsize + rsize + 1;
12     }
13 };
```

Árbol de Estadísticos de Orden

La STL incluye una etiqueta `tree_order_statistics_node_update`, que le indica a cada nodo que guarde el tamaño del sub-árbol correspondiente. Esta etiqueta incorpora dos funciones nuevas:

```
1 A.find_by_order(unsigned int k);
2 A.order_of_key(T key);
```

La función `find_by_order` regresa un iterador convencional que corresponde al k-ésimo elemento de `A` (indexados en 0). La función `order_of_key`, por su parte, regresa un entero no negativo que representa el número de elementos menores que `key`.

```
1 #include <iostream>
2 #include <functional>
3 #include <ext/pb_ds/assoc_container.hpp>
4 #include <ext/pb_ds/tree_policy.hpp>
5 using namespace __gnu_pbds;
6 using namespace std;
7
8 typedef tree<int, null_type, less<int>, rb_tree_tag,
      tree_order_statistics_node_update> ordered_set;
```

```
9  typedef ordered_set::node_iterator pnode;
10
11 //Imprime el arbol rotado 90 grados hacia la izquierda.
12 void print_tree(pnode it, pnode nil, int indent = 0) {
13     if (it != nil) {
14         print_tree(it.get_l_child(), nil, indent + 2);
15         for (int i = 0; i < indent; ++i)
16             cout << ' ';
17         cout << **it << '\n';
18         print_tree(it.get_r_child(), nil, indent + 2);
19     }
20 }
21
22 int main() {
23         //Datos de ejemplo.
24     int n = 10;
25     int arr[] = {20, 15, 50, 30, 25, 36, 10, 35, 40, 21};
26
27     //Crea un arbol con los datos de arr y lo imprime.
28     ordered_set v, w;
29     for (int i = 0; i < n; ++i)
30         v.insert(arr[i]);
31     print_tree(v.node_begin(), v.node_end()); cout << '\n';
32
33     //Separa el arbol en dos y los imprime.
34     v.split(30, w);
35     print_tree(v.node_begin(), v.node_end()); cout << '\n';
36     print_tree(w.node_begin(), w.node_end()); cout << '\n';
37
38     //Vuelve a unir ambos arboles y lo imprime.
39     v.join(w);
40     print_tree(v.node_begin(), v.node_end()); cout << '\n';
41
42     //Imprime el indice de 35.
43     cout << v.order_of_key(35) << '\n';
44     //Imprime el 7-esima valor.
45     cout << *v.find_by_order(7) << '\n';
46
47     return 0;
48 }
```