Decisiones de implementación del string_map

Acerca de nodos y tamaño

Implementamos el `string_map` como un trie, un diccionario especializado para strings, guardando las claves en nodos (`Nodo`) los cuales podemos recorrer según el largo de los string guardados, dándonos una complejidad óptima para nuestra implementación. Le dimos una variable tamaño para poder ver la cantidad de claves en tiempo constante, además de un puntero al nodo `raiz`, cuya clave es un string vacío. La estructura `Nodo` tiene: Una `clave`, palabra con la que identifico al Nodo. Puntero a `significado`, el cual apunta a null en caso de ser un nodo no definido. Uno a su `padre`, que en el caso de la raíz es un puntero a `null`. Uno a un `pair<clave, significado>`. Otro a un array de 96 posiciones, que corresponden a cada uno de sus `hijos`. Nótese que son 96 caracteres posibles por clave, y los hijos tienen la clave del padre concatenada al caracter que les corresponda.

Acerca de contenedorSignificado y claveSignificado

Nos parece de especial interés hacer incapié en los tipos `contenedorSignificado` y `claveSignificado`, ambas clases contenidas en el tipo `string_map`. Es esto así porque presentan un caso de redundancia de información que puede que llame la atención a la hora de leer nuestra implementación; y creemos oportuno aclarar el porqué de esta decisión.

De entrada, el tipo `contenedorSignificado` existe por la imposibilidad de declarar, sin un constructor vacío, instancias de ciertos tipos; como ocurre con el tipo `Dato`. Se decidió, para saltear este inconveniente, que el tipo habría de ser un puntero que se mantuviera indeclarado hasta la hora que fuera requerido; que en el caso de los nodos habla del momento en el que son 'definidos', entendiéndose esto como el proceso de adquisición de un significado.

Ahora, el problema surge cuando se utilizan los operadores `*` y `->` de las clases `string_map::iterator` y `string_map::const_iterator`. Ambas funciones tienen de tipo de retorno a nuestro definido `value_type`, que en el caso del diccionario implementado se trata de un `pair<const key_type, mapped_type>`, donde `key_type` es un string y `mapped_type` el tipo de los significados de la estructura. A la hora de construir dicho par, se desea que esté guardado en un lugar estático, a fin de poder referenciarlo con un puntero; por lo que la opción de acabar dichos métodos con un `return make_pair (...)` se desvanece.

Esto nos deja con el impedimento, al menos hasta donde se nos ocurrió e investigamos, de que si el usuario cambia el significado de un nodo usando el `operator[]` del `string_map`, cambiará el significado usando el `operator=` del tipo del significado, por lo que no podemos sobrecargarlo. Esto deviene en que, de almacenar el par, no podemos actualizarlo inmediatamente conforme el significado cambie. Y esto es un problema, porque si el usuario cambiase el significado mediante este operador y luego requierera el par, la tupla provista no contendría el significado real del nodo.

No encontramos como opción tampoco que el par se conformara con una referencia al significado, pues su tipo pasaría a ser `pair<const key_type, &mapped_type>`, lo que no concordaría con el tipo

definido en la aridad de los operadores de los iteradores.

Optamos, entonces, por mantener el par en un contenedor fuera del nodo, para no tener que modificar el nodo en funciones const. Pero además le fue provista una función '`refrescarClaveSignificado`', que lo que hace es crear una nueva tupla con el significado actual del nodo al que sirve. Esta función es llamada desde todo método que requiera devolver el par antes de devolverlo, por lo que, aunque no todo el tiempo la tupla se corresponda con el significado actual del nodo, sí lo hará al momento en el que dicho par haya de ser devuelto.