Laboratorio 4 de Programación 3

Leandro Rabindranath León

- **Este laboratorio es evaluado y la calificación que obtengas cuenta para tu calificación definitiva.**
- **Fecha de entrega: lunes 10 de noviembre de 2014 hasta las 6pm**
- **Tu entrega consiste de un archivo llamado `dictnode.H`, Por favor, al inicio de este archivo, en comentarios, pon los apellidos, nombres y números de cédula de los miembros de la pareja que está sometiendo el archivo.**

1. Introducción

El fin de este laboratorio es desarrollar destrezas algorítmicas con el uso de árboles generales. Para cumplir este propósito en esta laboratorio instrumentaremos un tipo de árbol llamado de prefijos el cual almacena y recupera eficientemente palabras.

Para la realización de este laboratorio requieres:

1. La suite clang
2. La biblioteca $\aleph_\omega$ (Aleph-ω).

1.1. Árbol de prefijos

El fin de un árbol de prefijos es representar una especie de diccionario. Consideremos por ejemplo las siguientes palabras:

```
amigos amiga amistoso amigo amigas estudiar estudiante estudioso estudio
va van vamos vamonos ir iremos amor amante amoroso amorosos programa
programar programador programadora programadores programas programado
programando programandonos
```

Podríamos emplear un árbol binario de búsqueda, u otras estructuras que estudiaremos posteriormente, para almacenar estas palabras a efectos de su rápida recuperación. Según sea la estructura podríamos obtener tiempos de respuesta $\mathcal{O}(\lg n)$, esperado o garantizado, según el tipo de árbol binario, o $\mathcal{O}(1)$ esperado para tablas hash. Sin embargo, especialmente a la luz de la estructura de este laboratorio, cualquiera de estos enfoques exhibe los siguiente problemas:

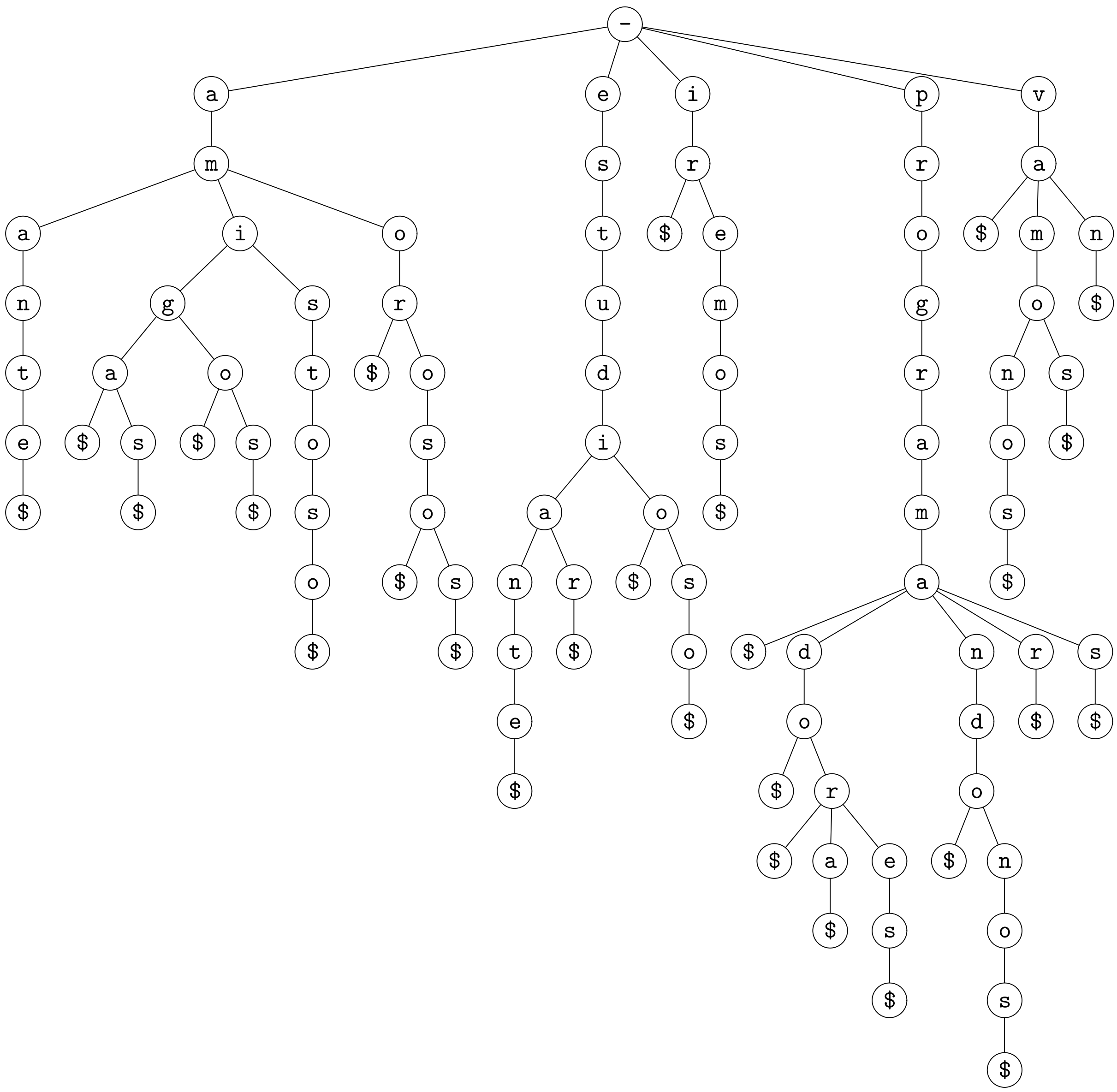

Figura 1: Árbol de prefijos de las palabras ejemplo

1. Puesto que muchas palabras comparten prefijos, se gastará tiempo comparando prefijos comunes.

   Nota que para comparar dos palabras entre sí es necesario inspeccionar enteramente las dos palabras.

2. Hay un consumo de espacio considerable dado precisamente por los prefijos comunes.

Tomando en cuenta que muchas palabras de nuestro (y otros) lenguajes, comparten prefijos, podemos emplear una estructura de árbol que represente las palabras. Por ejemplo, las palabras anteriores pueden representarse mediante el árbol de prefijos de la figura 1.

El nodo raíz es especial, pues no contiene ninguna letra, y representa la raíz de todas las palabras. Se representa mediante el símbolo `-`.

Para saber si un prefijo es una palabra, al nodo que representaría el fin de la palabra se le añade un hijo especial $. Dicho de otro modo, si un nodo `p` del árbol contiene un hijo $, entonces la palabra compuesta por el recorrido desde la raíz hasta `p` es una palabra que pertenece al diccionario. De este modo, para determinar si una palabra $w = w_1 w_2 \dots w_n$ pertenece o no al diccionario, comenzamos por buscar un hijo `p` de la raíz `-` que contenga w_1. Si hay una correspondencia, entonces continuamos buscando en los hijos de `p` un nodo que contenga w_2. Si el cualquier paso i no encontramos un hijo para w_i, entonces la palabra no existe en el diccionario. Si al final encontramos un nodo `q` para w_n y éste contiene a $ como hijo, entonces la palabra pertenece al diccionario; de lo contrario, la palabra no está en el árbol (mas si es un prefijo).

Nota que la búsqueda de una palabra es $\mathcal{O}(1)$ determinista. Puesto que la longitud de una palabra está acotada, la altura del árbol está acotada (a la palabra de mayor longitud más 2). Puesto que la cantidad de letras está acotada (por simplicidad a 26 en nuestro caso), el orden del árbol es 27 (tomando en cuenta a `'$'`). Ambas cifras (la mayor palabra y 27) son constantes; por consiguiente su producto, que representaría, el peor (y muy pesimista[1]) de los casos de búsqueda es constante. De aquí pues $\mathcal{O}(1)$. El mismo razonamiento opera para la inserción y eventualmente eliminación.

1.2. El tipo `Cnode`

En este laboratorio implementarás la clase `Cnode` la cual representa un nodo de un árbol de prefijos que contiene una letra.

Aunque en el archivo `dictnode.H` `Cnode` deriva de `Tree_Node`, en este laboratorio eres libre de implementar tu propio árbol. Por supuesto, para ello debes eliminar la derivación y algunos métodos *ad hoc* que usan a `Tree_Node`. Este tipo de decisión es común en el mundo del software y una de las metas secundarias es que la confrontes. A este tenor, **no pidas recomendación a tu Profesor**; medita, toma una decisión y asúmela. Después de transcurrido el laboratorio discutiremos en clase esta decisión.

1.2.1. Representación del árbol como cadena

`Cnode` contiene el método `to_str()`, el cual retorna una cadena que representa el árbol. Puede serte útil para depurar. Por ejemplo, para nuestro árbol ejemplo, la salida es:

```
-(a(m(a(n(t(e($)))))(i(g(a($)(s($)))(o($)(s($))))(s(t(o(s(o($)))))))
(o(r($)(o(s(o($)(s($)))))))(e(s(t(u(d(i(a(n(t(e($))))(r($)))(o($)
(s(o($)))))))))(i(r($)(e(m(o(s($)))))))(p(r(o(g(r(a(m(a($)(d(o($)(r($)
(a($))(e(s($))))))(n(d(o($)(n(o(s($)))))))(r($))(s($)))))))))(v(a($)
(m(o(n(o(s($))))(s($))))(n($))))
```

[1] Si la representación del árbol es con listas ordenadas alfabéticamente, entonces una palabra como zzzzz... zzz sería un mal caso.

`to_str()` ya está implementada para ti, pero requiere el método `children()`, el cual tú debes implementar.

1.2.2. `ntreepic`

$\aleph_\omega$ (Aleph-ω) tiene un programa llamado `ntreepic` diseñado para dibujar árboles. No es un requerimiento usarlo, pero podría serte más útil que `to_str()`.

Para usarlo, invoca el método `ntreepic()` de `Cnode`, el cual ya está implementado. Luego crea un archivo, por ejemplo, `test.Tree` con la salida de `ntreepic()`.

Ahora invoca `ntreepic`:

```
ntreepic -f test.Tree
```

El cual generará `test.eepic`. Después invoca el script

```
eepic2eps ejm.eepic
```

Este script, que te será dado con el laboratorio, crea `ejm.eps` (formato postscript). El script requiere que tengas LaTeX y algunos paquetes asociados instalados. La mayoría de los visores de pdf pueden leer este formato. También puedes convertirlo a pdf con `epstopdf` (por lo general disponible en cualquier distribución linux).

Ejecuta

```
ntreepic --help
```

Para conocer las distintas opciones de dibujado.

2. Laboratorio

La tarea de este laboratorio consiste en diseñar y codificar los diferentes métodos de `Cnode` .

Ten en cuenta que cuando hay que crear un nuevo nodo hay que reservar su memoria. Por ejemplo:

```
Cnode * ptr = new Cnode('a');
```

2.1. Constructor

Programa:

```
Cnode(const char c)
```

El cual inicializa un nodo con el símbolo `c`.

Para manejar localización se requeriría usar los tipos `wchar_t` y cadenas de tipo `wstring`. Puesto que hay algunas sutilezas con el manejo de estos tipos y además nuestra meta es aprender árboles, usaremos `ASCII` extendido, el cual maneja los acentuaciones del castellano y de otras lenguas romances.

Así pues, el constructor debe filtrar según las siguientes reglas:

1. Letras mayúsculas son convertidas a minúsculas.

2. Cualquier acento sobre una vocal es eliminado y convertido a la vocal simple. De este modo, cualquiera de los siguientes símbolos

   `á é í ó ú à è ì ò ù ä ë ï ö ü â î ô û ã õ`

   Debe convertirse a su correspondiente vocal simple.

3. La cedilla `ç` debe convertirse a una `c`.

4. La `ñ` debe convertirse a `n`.

Este conjunto de reglas permitirá además obtener un orden lexicográfico de una manera algorítmicamente simple.

Si el valor de `c` es distinto de `$` o `-` y no es un símbolo alfabético, entonces se debe disparar la excepción `invalid_argument("Invalid character")`.

Dos observaciones cruciales. La primera es que en este constructor no se requiere invocar a `new`. La segunda es que es muy importante que filtres correctamente los distintos acentos, cedilla y eñe, y que dispares excepción si el símbolo es inválido, pues es desde este mecanismo que usaremos diccionarios generales.

No olvides que las mayúsculas se convierten a minúsculas.

`clang` puede emitirte unos cuantos warnings por la codificación de las letras acentuadas, eñe y cedilla. Si te son engorrosos, apaga el warning específico.

2.2. Observador

Implementa:

```
char symbol()
```

El cual retorna el símbolo almacenado en el nodo `this`.

No es mala idea colocar una invariante que verifique que el símbolo está filtrado; o sea que no tiene acento, eñe o cedilla y está en minúscula.

2.3. Inserción de hijo

Escribe el método:

```
Cnode * insert_child(Cnode * child)
```

La cual inserta al nodo `this` el nodo `child` como un hijo. La rutina retorna un puntero al nodo hijo recién insertado.

Nota que un árbol de prefijos válido no puede contener dos hijos con el mismo símbolo. Según sea tu diseño podrás asumir que `child->symbol()` no está presente dentro de los hijos de `this` o aprovechar este punto para efectuar esta validación. Nuestra rutina no efectúa validación porque nuestro diseño se asegura de jamás repetir una letra como hijo; sin embargo, podría serte conveniente validar, en cuyo caso retorna `NULL` si no ocurre la inserción.

Nota que la única manera de evaluar esta rutina sin intervenir la implementación es a través del método siguiente. En todo caso, la evaluación no verificará una doble inserción.

2.4. Consulta de hijos

Escribe:

```
DynList<Cnode*> children()
```

El cual retorna una lista de punteros a los hijos de `this`.

Aparte de que este método puede ayudarte enormemente a instrumentar otros métodos, es sumamente importante que esté correcto, pues muchas de las validaciones conque te evaluaremos subyacen en él.

2.5. Búsqueda de hijo

Implementa:

```
Cnode * search_child(const char c)
```

El cual retorna un puntero al hijo de `this` que contiene el símbolo `c` o `NULL` si no existe tal hijo

2.6. Marcado de palabra

Nota que cualquier recorrido desde la raíz de un árbol de prefijos hasta cualquier nodo representa un prefijo. Por ejemplo, en la figura 1, la secuencia más a la izquierda desde `-` hasta `e`, representa el prefijo "amante". Puesto que este prefijo es una palabra válida, debemos marcarlo como tal añadiéndole un hijo `$`.

Bajo la consideración anterior, implementa el método:

```
void mark_end_word()
```

El cual marca al prefijo denotado por `this` como una palabra de un diccionario implementado con el árbol de prefijos.

Recuerda que marcar un prefijo como palabra consiste en añadirle al nodo un hijo con el valor `$`.

Según sea tu diseño, puedes requerir validar o no que `this` esté o no marcado. En todo caso, recuerda que jamás un nodo debe contener hijos con símbolos repetidos, el `$` inclusive. Si optas por validar, entonces es recomendable que el método retorne un valor lógico (`bool`) que indique si el marcado fue o no efectuado.

2.7. Prueba de palabra

Implementa el método:

```
bool is_end_word()
```

El cual retorna `true` si `this` es el final de una palabra válida (si está marcada); `false` de lo contrario.

Nota que este método simplemente se remite a sondear si `this` tiene un hijo `$`.

2.8. Búsqueda de prefijo

Implementa el método:

```
tuple<Cnode*, const char*> search_prefix(const char * prefix)
```

Este método recibe como parámetro un prefijo el cual desea verificar si está presente en el árbol con raíz `this`.

EL método retorna una dupla `tuple<Cnode*, const char*>`. El primer elemento es el nodo donde culmina el prefijo. El segundo elemento es el sufijo de `prefix`.

Ejemplos para el árbol de la figura 1 y `root->symbol() == '-'`):

1. `root->search_prefix("amaras")`

   Retorna como primer elemento de la dupla al nodo con el símbolo `'a'`, cual es la terminación del prefijo dentro del árbol que contiene a "`amaras`" ("ama"). El segundo elemento es la cadena "`ras`", cual es el sufijo.

2. `root->search_prefix("zapato")`

   Retorna como primer elemento a nodo con el símbolo `'-'` (la misma raíz) y el sufijo "`zapato`".

3. `root->search_prefix("amante")`

   Retorna como primer elemento al nodo con el símbolo `'e'` (terminación del prefijo y de la palabra) y el sufijo "" (la cadena vacía).

Esta rutina es "naturalmente" recursiva. Nota que si `p` es el hijo más a la izquierda de `root`, entonces

```
p->search_prefix("maras")
```

Retorna el mismo resultado que `root->search_prefix(''amaras'')`.

Para esta rutina puede serte sumamente útil recordar y entender las cadenas y la aritmética de punteros en lenguaje C. Recuerda que las cadenas terminan en `'\0'`. De este modo, si por ejemplo,

```
str = "amaras"
```

Entonces `root->search_prefix(str)` puede recursionar a `p->search_prefix(str + 1)`.

Ten sumo cuidado de descubrir y comprender las condiciones de parada de la recursión.

Recuerda que una cadena válida puede contener mayúsculas, pero que las letras guardadas en el árbol deben ser minúsculas.

Esta rutina es reusable, especialmente para la inserción de palabras. Comprende cómo podría usarse para buscar palabras. ¿Cuál el valor del primer elemento si ninguna parte del prefijo se encuentra en el árbol?

Por supuesto, una versión iterativa es posible.

2.9. Búsqueda de palabra

Programa:

```
bool contains(const string & word)
```

El cual retorna `true` si `word` está contenido como palabra; es decir, si `word` está contenido como prefijo y su última letra está marcada (árbol contiene `$`).

Puede serte útil `search_prefix()` o un método helper como:

```
Cnode * search_word(const char * word)
```

Cuya estructura es similar a `search_prefix()`, pero con condiciones de parada distintas.

2.10. Inserción de palabra

Programa:

```
bool insert_word(const string & word)
```

La cual retorna `true` si `word` está contenida en el árbol como palabra; es decir, si su última letra está marcada. De lo contrario (la palabra ya está en el árbol), retorna `false`.

El método debe generar la excepción

```
invalid_argument("word contains an invalid character")
```

si la palabra contiene un carácter inválido. Para ello podría serte útil delegar el trabajo en el constructor y capturar (con `catch(...)`) la excepción. Nota que esta excepción no es la misma que la del constructor (los mensajes son distintos).

Esta es probablemente la parte más compleja de este laboratorio. Estudia detalladamente el problema y los diferentes casos. ¿Qué hacer si `word` ya existe como prefijo? ¿Qué hacer si el árbol ya contiene un prefijo de `word`?

2.11. Destrucción

Para crear un árbol de prefijos sólo es necesario declarar su raíz. Algo así como

```
Cnode root('-');
```

o

```
Cnode * root = new Cnode('-');
```

Las reservaciones de memoria posteriores son realizadas por métodos específicos del árbol.

Con esto en cuenta, programa

```
void destroy()
```

El cual "destruye" el árbol y libera toda la memoria del árbol.

Si optas por usar `Tree_Node`, entonces puedes usar la rutina `destroy_tree()`. Si no, de todos modos pudiera serte útil revisarla.

2.12. Lista ordenada de palabras que contiene el diccionario

Instrumenta el método:

```
DynArray<string> words()
```

El cual retorna un arreglo dinámico con todas las palabras del diccionario. El arreglo debe estar ordenado lexicográficamente.

Para construir el arreglo requieres recorrer enteramente el árbol en prefijo y adjuntar al arreglo las palabras marcadas. Debes encontrar una manera de "bufferear" el camino desde la raíz hasta el nodo actual que está visitando; una pila es lo ideal. No olvides que al terminar un recorrido parcial debes desempilar. Con este enfoque, cada vez que detectes una palabra la pila contiene la secuencia que la define.

Si requieres ordenar, entonces puede serte útiles los métodos de `tpl_sort_utils.H`. Ten sabiduría con la escogencia del método de ordenamiento. Si optas por ordenar tienes varias alternativas. Una es ordenar todo el arreglo. Otra consiste en ordenar la salida de `children()` y basar tu recorrido en esta rutina.

La eficiencia de esta rutina es evaluada. En este sentido, lo más eficiente es que las inserciones de los hijos, concretamente el método `insert_child()` estén ordenadas por la letra del nodo insertado. De este modo, el recorrido prefijo es ordenado. Sin embargo, este enfoque es delicado por dos razones. La primera es que hace algo más lenta la inserción del hijo; aunque aún es $\mathcal{O}(1)$. Podría decirse que el coste de `words()` es costeado con anticipación durante la inserción. Este dilema de desempeño será dirimido en el curso siguiente de análisis y diseño de algoritmos mediante una técnica llamada "análisis amortizado". La segunda razón es que hace más difícil programar `insert_child()`.

Recuerda que la eficacia prima a la eficiencia. Es más importante tener un algoritmo operativo primero, aunque no sea el más eficiente, que uno de alto desempeño pero ineficaz (incorrecto). Planteada esta consideración, lo recomendable es escribir `insert_child()` lo más simple posible y ordenar el arreglo cuando implementes `words()`. Aunque esto hará a `words()` más lento tendrás más posibilidades de que tus rutinas estén correctas. Por otra parte, supeditado a nuestros tiempos de respuesta, puedes someter varias veces. Así que luego de asegurarte de que todo esté correcto podrías dedicarte a rediseñar `insert_child()` y `words()`.

3. Evaluación

La fecha de entrega de este laboratorio es el viernes 17 de 2014 hasta la 6pm. Puedes entregar antes, recibir corrección e intentar cuantas veces prefieras (limitado por supuesto a nuestra capacidad computacional).

Para evaluarte debes enviar el archivo `dictnode.H` a la dirección:

```
leandro.r.leon@gmail.com
```

Antes de la fecha y hora de expiración. El "subject" debe **obligatoria y exclusivamente** contener el texto **PR3-LAB-04**. Si fallas con el subject entonces probablemente tu laboratorio no será evaluado, pues el mecanismo automatizado de filtrado no podrá detectar tu trabajo.

En el mensaje y en el inicio del fuente del programa debes colocar los nombres, apellidos y números de cédula de la pareja que somete el laboratorio. **Por favor, no uses otros medios para enviar la solución ni escribas otros comentarios adicionales en el email**.

La evaluación esta dividida en dos partes. La primera parte es ejecutada automáticamente por un programa evaluador compilado con tu fuente. El programa contiene casos de prueba y se coteja con una implementación de referencia. Para cada una de tus rutinas se ejecuta una serie de pruebas (aproximadamente 10) y se comparan con las pruebas y salidas de la implementación de referencia. La más mínima desavenencia implica que el test particular tiene 0 puntos. Por cada test que falle, se te reportará la entrada que causó la falla. La evaluación de la primera parte representa aproximadamente 90 % de la calificación.

Atención: si tu programa no compila, entonces el evaluador no compila. Si una de tus rutinas se cae, entonces el evaluador se cae. Si una de tus rutinas cae en un lazo infinito o demora demasiado, entonces el evaluador cae en un lazo infinito o se demora demasiado. Exactamente sucedería lo mismo si tu programa fuese parte de un sistema de aviónica: ¡se estrella el avión!. Por esa razón, en todos estos casos será imposible darte una nota, lo que simplemente se traduce en que tienes cero. Si una de tus rutinas dispara una excepción inesperada, el evaluador hará lo posible por capturarla y reportártela; sin embargo, esto no es un compromiso; podría estrellarse el avión. En todo caso, si el evaluador no logra capturar la excepción, no se te dará nota pero se te reportará la excepción.

No envíes programas que no compilan o se caen. Hacen perder los tiempos de red, de cpu, el tuyo y el mío. Si estás al tanto de que una rutina se te cae y no la puedes corregir, entonces trata de aislar la falla y disparar una excepción cuando ésta ocurra. De este modo no tumbarás al evaluador y eventualmente éste podría darte nota para algunos casos (o todos si tienes suerte). Si no logras aislar la falla, entonces deja la rutina vacía (tal como te fue dada). De este modo, otras rutinas podrán ser evaluadas.

La segunda parte es subjetiva y manual. Consiste en revisar tu fuente con la intención de evaluar tu estilo de codificación. Es recomendable que sigas las reglas de estilo comentadas en clase, pues estas armonizan con la persona que subjetivamente te evaluará. Esta parte aporta aproximadamente el 10 % de la nota.

4. Recomendaciones

1. Lee enteramente este enunciado antes de proceder al diseño e implementación.

2. Desarrolla tus rutinas en el orden dado.

3. Esta práctica es simple, pero probablemente más compleja que la anterior del sudoku. Esto, aunado, a tu eventual inexperiencia con árboles, podría hacer a este laboratorio laborioso.

   Dicho lo anterior, **comienza cuanto antes y trata de tener una versión operativa cuando menos dos días antes de la fecha límite**. De este

modo, podrás aprovechar la evaluación y eventualmente corregir y mejorar tu nota.

4. Esta práctica no tiene casos de prueba. Es a ti el diseñarlos. Basa tus casos de prueba en asertos (`assert()` o `I()`). Asegúrate de cubrir todas las posibilidades. Diseña distintos casos de prueba por cada método de `Cnode` . Vislumbra todas las posibilidades y no olvides de verificar las excepciones cuando sean requeridas. **No sometas el laboratorio si no has diseñado tus casos de prueba y éstos han sido satisfactorios.**

5. Usa el foro para plantear tus dudas de comprensión. Pero de ninguna manera compartas código, pues es considerado **plagio**.

   Por favor, **no uses el correo electrónico ni el correo del facebook para plantear tus dudas. ¡Usa el foro!**

6. No incluyas headers en tu archivo `dictnode.H` (algo como `# include ...`), pues puedes hacer fallar la compilación. Si requieres un header especial, el cual piensas no estaría dentro del evaluador, exprésalo en el foro para así incluirlo.

7. La gran mayoría de las pruebas conque serás evaluada dependen de la correctitud de `children()`. **Asegúrate de que este método esté correcto**. Puesto que lo más probable es que tú mismo uses el constructor, asegúrate también de que esté correcto.

8. Por atención a las mayúsculas en las cadenas. Recuerda que el árbol sólo maneja minúsculas, pero algunas operaciones pueden recibir cadenas conteniendo mayúsculas. Especialmente, presta atención a las mayúsculas cuando hay acentos, eñe o cedilla.