Nombre: ..... Fecha: / 12 / 2008

Grupo: 1 ☐ 2 ☐ 3 ☐ 4 ☐

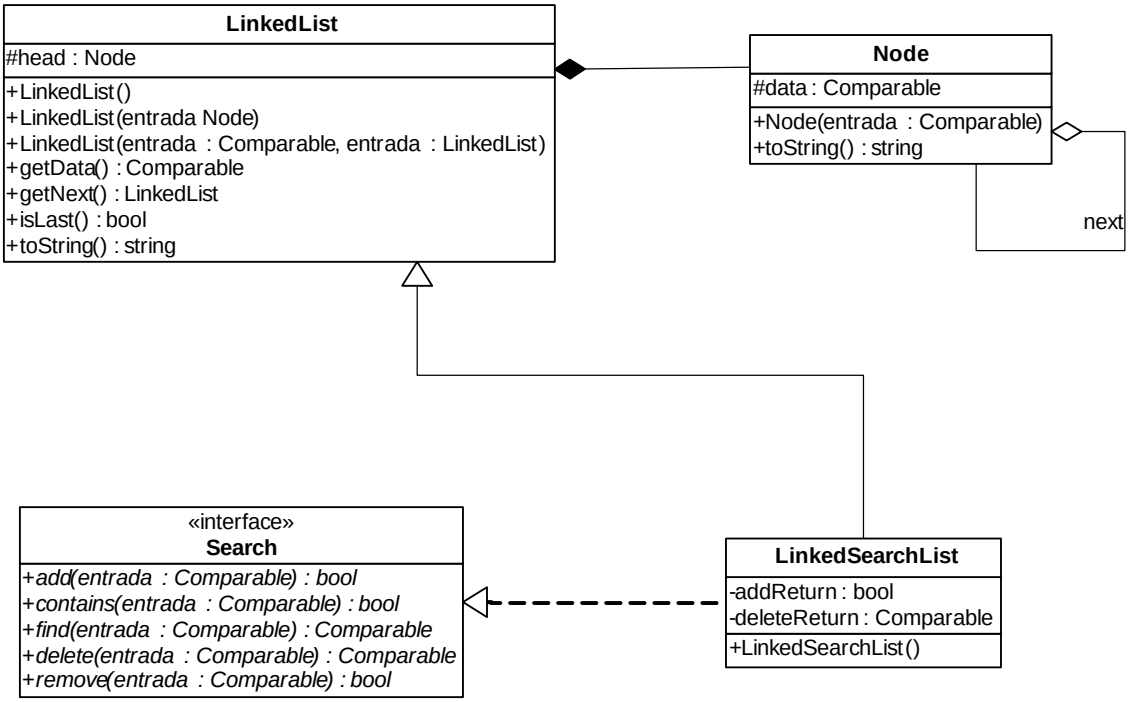
PRÁCTICA 10
USO DE CLASES ABSTRACTAS EN C++ E INTERFACES EN JAVA

En esta práctica, partiendo del código de que dispones sobre árboles binarios con búsqueda, implementaremos una lista enlazada ordenada sobre la cual también seamos capaces de hacer búsquedas de forma más optimizada.

En dicho desarrollo haremos uso de las nociones de "clase completamente abstracta" de C++ y de "interface" en Java.

Parte 1. Interfaces en Java.

Partimos del siguiente diagrama de clases en UML:



1. Implementa el mismo en Java.

Si tienes dudas sobre el comportamiento de alguno de lo métodos anteriores puedes consultar el ejemplo de "BinarySearchTree" en Java.

El método "toString(): String" en "LinkedList" debe mostrar todos los "data" que contenga la lista (es decir, tendrás que redefinir su comportamiento con respecto al de la clase "Object").

Debes tener en cuenta que los métodos correspondientes a la "interface" deben estar optimizados y deben preservar el orden lexicográfico en la lista. Es decir:

a) "add (Comparable): bool" debe introducir su argumento en la lista de tal forma que todos los objetos que haya antes sean menores que él, y todos los que haya después mayores. Devolverá "true" si el argumento no se encontraba previamente en la lista, y "false" en caso contrario. Observa que ningún objeto puede aparecer duplicado en la lista.

b) "contains(Comparable): bool" debe buscar si su argumento se encuentra en la lista hasta que encuentre el objeto igual que él o el primer objeto "mayor" que él. En ese momento cesará la búsqueda y devolverá el booleano correspondiente.

c) "find(Comparable): bool" se comportará de modo similar a "contains(Comparable): bool", es decir, no recorrerá toda la lista, sino sólo hasta que encuentre un objeto igual que él, o el primer objeto mayor que él.

d) "delete(Comparable): Comparable" y "remove(Comparable): bool" sólo deben recorrer la lista hasta que encuentren el objeto que deben remover de la misma, o el primer objeto mayor que su argumento. Devolverán el objeto borrado o el booleano correspondiente, respectivamente.

Si necesitas algún método auxiliar en la clase "LinkedSearchList", decláralo como "private" para no modificar el comportamiento de la misma. (Si quieres implementar los métodos de forma recursiva puedes reutilizar partes de los definidos en "BinarySearchTree")

Puedes encontrar información relativa a la "interface" "Comparable" en http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Comparable.html. Es suficiente con que sepas que implementa el método "compareTo(entrada): int", y que devuelve un valor menor que cero, igual a cero o mayor que cero dependiendo de que el argumento que le pasamos sea menor, igual o mayor que el objeto desde el que se llama.

2. Vamos a comprobar que la clase "String" implementa la "interface" "Comparable" de Java. Crea un fichero de nombre "principal_prac10.java" que realice las siguientes acciones:

a) Crea un objeto "listaOrd1" de tipo "LinkedSearchList" con el constructor de dicha clase.

b) Introduce en "listaOrd1", por medio del método "add(Comparable): bool", y en este orden, las cadenas de caracteres:

"Me" "he" "despertado" "casi" "a" "las" "diez" "y" "me" "he" "quedado" "en" "la" "cama" "mas" "de" "tres" "cuartos" "de" "hora" "y" "ha" "merecido" "la" "pena" "ha" "entrado" "el" "sol" "por" "la" "ventana" "y" "han" "brillado" "en" "el" "aire" "algunas" "motas" "de" "polvo" "he" "salido" "a" "la" "ventana" "y" "hacia" "una" "estupenda" "mañana"

c) Utiliza el método "toString(): String" sobre "listaOrd1". Comprueba que en "listaOrd1" no existe ningún elemento duplicado. Comprueba también que los elementos están ordenados con respecto al orden lexicográfico.

d) Comprueba que la cadena "ventana" está en "listaOrd1" por medio del método "contains(Comparable): bool".

e) Elimina de la lista, por medio del método "delete(Comparable): Comparable", la cadena "ventana".

d) Comprueba que la cadena "ventana" no está en "listaOrd1" por medio del método "contains(Comparable): bool".

3. Veamos ahora la versatilidad del código que hemos creado. Nuestra clase "LinkedSearchList" es capaz de alojar objetos de cualquier clase que implemente la "interface" "Comparable".

En concreto, la clase "Integer" de Java lo hace. Lo puedes comprobar en http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Integer.html.

a) Crea una nueva lista "listaOrd2" de la clase "LinkedSearchList". Declara un bucle con la siguiente estructura:

```
//Debes añadir en la cabecera la librería "java.util.Random"

Random randomGenerator = new Random();
for (int i = 0; i < 250; i++){
                listaOrd2.add (new Integer(randomGenerator.nextInt()));
}
```

Puedes obtener información adicional sobre la clase "Random" en http://java.sun.com/j2se/1.5.0/docs/api/java/util/Random.html.

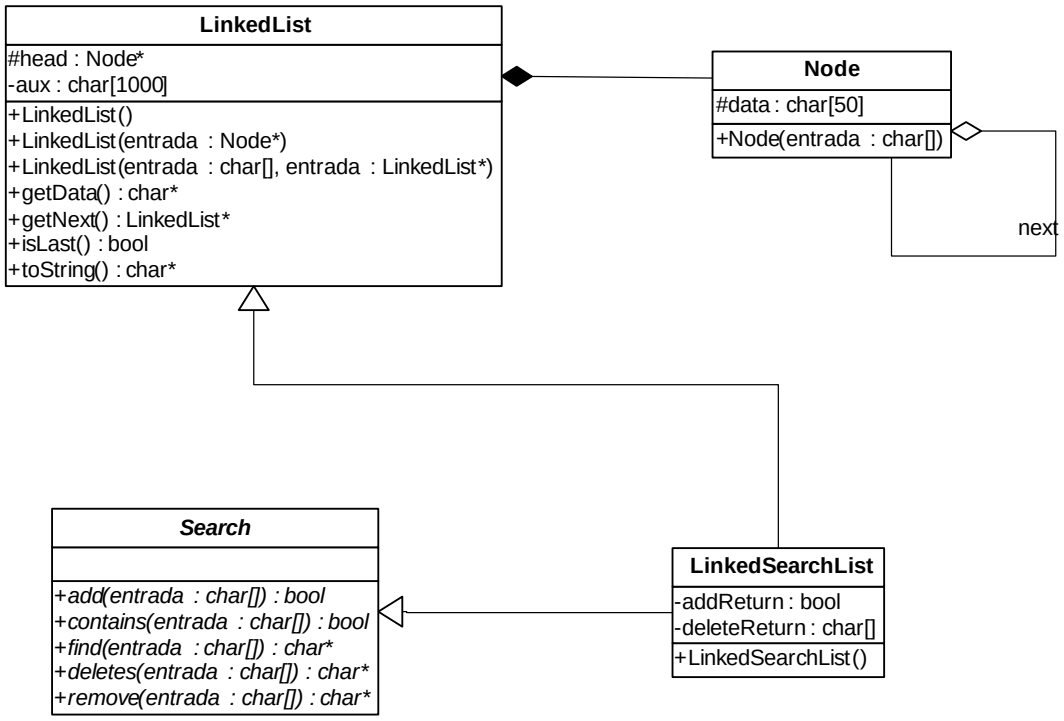
b) Muestra por pantalla "listaOrd2" por medio del método "toString(): String", y comprueba que está ordenada de menor a mayor. Comprueba el enlace

http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Integer.html#compareTo(java.lang.Integer) ya que dicho método ha sido el encargado de ordenar la lista de números (al estar declarado en la "interface" "Comparable").

c) Comprueba que el resto de métodos propios de "LinkedSearchList" funcionan de modo correcto sobre "listaOrd2" ("delete (Comparable): Comparable", "remove(Comparable): bool", ...)

Parte 2. Clases completamente abstractas en C++.

Partimos del siguiente diagrama de clases en UML:



1. Implementa el mismo en C++.

Si tienes dudas sobre el comportamiento de alguno de lo métodos anteriores puedes consultar el ejemplo de "BinarySearchTree" en C++.

El método "toString(): char *" en "LinkedList" debe mostrar todos los "data" que contenga la lista.

Debes tener en cuenta que los métodos correspondientes a la clase abstracta "*Search*" deben estar optimizados y deben preservar el orden lexicográfico en la lista. Es decir:

a) "add (char []): bool" debe introducir su argumento en la lista de tal forma que todas los cadenas que haya antes sean menores que ella, y todas los que haya después mayores. Devolverá "true" si la cadena no se encontraba previamente en la lista, y "false" en caso contrario. Observa que ningún objeto puede aparecer duplicado en la lista.

b) "contains(char []): bool" debe buscar si su argumento se encuentra en la lista hasta que encuentre la cadena igual que ella o la primera "mayor" que ella. En ese momento cesará la búsqueda y devolverá el booleano correspondiente.

c) "find(char []): bool" se comportará de modo similar a "contains(char []): bool", es decir, no recorrerá toda la lista, sino sólo hasta que encuentre una cadena igual que ella, o la primera cadena mayor que ella.

d) "delete(char []): char *" y "remove(char []): bool" sólo deben recorrer la lista hasta que encuentren la cadena que deben remover de la misma, o la primera cadena mayor que su argumento. Devolverán la cadena borrada o el booleano correspondiente, respectivamente.

Si necesitas algún método auxiliar en la clase "LinkedSearchList", decláralo como "private" para no modificar el comportamiento de la misma. (Si quieres implementar los métodos de forma recursiva puedes reutilizar partes de los definidos en "BinarySearchTree")

2. Vamos a comprobar que los objetos de la clase "LinkedSearchList" tienen el comportamiento esperado. Crea un fichero de nombre "principal_prac10.cpp" que realice las siguientes acciones:

a) Crea un objeto "listaOrd1" de tipo "LinkedSearchList" con el constructor de dicha clase.

b) Introduce en "listaOrd1", por medio del método "add(char []): bool", y en este orden, las cadenas de caracteres:

"Me" "he" "despertado" "casi" "a" "las" "diez" "y" "me" "he" "quedado" "en" "la" "cama" "mas" "de" "tres" "cuartos" "de" "hora" "y" "ha" "merecido" "la" "pena" "ha" "entrado" "el" "sol" "por" "la" "ventana" "y" "han" "brillado" "en" "el" "aire" "algunas" "motas" "de" "polvo" "he" "salido" "a" "la" "ventana" "y" "hacia" "una" "estupenda" "mañana"

c) Utiliza el método "toString(): char *". Comprueba que en "listaOrd1" no existe ningún elemento duplicado. Comprueba también que los elementos están ordenados con respecto al orden lexicográfico.

d) Comprueba que la cadena "ventana" está en "listaOrd1" por medio del método "contains(char []): bool".

e) Elimina de la lista, por medio del método "delete(char []): char *", la cadena "ventana".

d) Comprueba que la cadena "ventana" no está en "listaOrd1" por medio del método "contains(char []): bool".

ENTREGAR los archivos *Node.java*, *LinkedList.java*, *LinkedSearchList.java*, *Search.java*, *principal_prac10.java*, *Node.h*, *Node.cpp*, *LinkedList.h*, *LinkedList.cpp*, *LinkedSearchList.h*, *LinkedSearchList.cpp*, *Search.h*, *principal_prac10.cpp*. Los archivos deben contener **todos los cambios** realizados durante los ejercicios de la práctica. La entrega es a través de Belenus en el repositorio de Julio Rubio. También deben contener unas cabeceras que permitan identificarte:

```
/* Nombre: ___
     Grupo: _____
     Nombre del fichero: ____
     Descripción: _______*/
```

Guarda una copia de los ficheros para las prácticas sucesivas