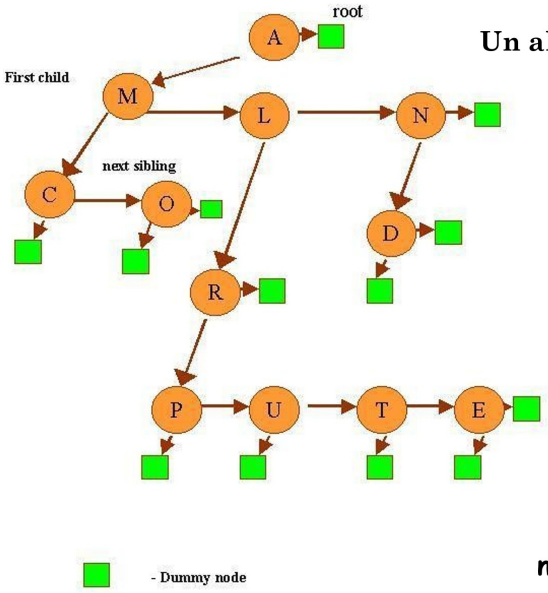
1 Obiettivo del laboratorio

In questo laboratorio viene richiesto di ***completare il TDD Albero generico*** rappresentato con struttura dati primo figlio - prossimo fratello (vedere figura sotto ripresa dalle slide di teoria):



Un albero è un puntatore **a treeNode**

```
struct tree::treeNode {
    string label;
    treeNode *firstChild;
    treeNode *nextSibling;
};
typedef treeNode* Tree;
```

numero arbitrario di figli!

2 Funzioni da implementare

Dovrete quindi progettare e implementare il codice delle funzioni offerte dall'interfaccia del TDD e definite, in modo incompleto, nel file `tree-chsib.cpp` che troverete, assieme ad altri file, all'interno del file .zip scaricabile da Aulaweb nella sezione relativa al laboratorio 7. Più in dettaglio le funzioni da implementare sono le seguenti:

```
namespace tree{

enum Error {OK, FAIL};

typedef string Label;

const Label emptyLabel = ....

struct treeNode; // definita nel file tree.cpp

typedef treeNode* Tree; // un albero e' identificato dal puntatore alla sua radice;
                        // useremo indistintamente "albero" e "puntatore a nodo"

const Tree emptyTree = NULL;

// isEmpty restituisce true se l'albero t e' vuoto, false altrimenti
bool isEmpty(const Tree&);

// addElem aggiunge un nodo etichettato con la prima label
// come figlio del nodo etichettato la seconda label
Error addElem(const Label, const Label, Tree&);

// deleteElem (versione iterativa) rimuove dall'albero il nodo etichettato con la Label l
// e collega al padre di tale nodo tutti i suoi figli
// Restituisce FAIL se si tenta di cancellare la radice e questa ha
// dei figli (non si saprebbe a che padre attaccarli) oppure se non esiste
// un nodo nell'albero etichettato con la Label; cancella e restituisce OK altrimenti
Error deleteElemR(const Label, Tree&);
```

```
// deleteElem (come sopra ma versione ricorsiva)
Error deleteElemI(const Label, Tree&);

// father restituisce l'etichetta del padre del nodo con etichetta l
// se il nodo esiste nell'albero (o sottoalbero) t e se il padre esiste.
// Restituisce la costante emptyLabel altrimenti
Label father(const Label, const Tree&);

// children restituisce una lista di Label, contenente le etichette
// di tutti i figli nell'albero t del nodo etichettato con l
list::List children(const Label, const Tree&);

// degree restituisce il numero di archi uscenti dal nodo etichettato con l;
// restituisce -1 se non esiste nessuna etichetta l nell'albero
int degree(const Label, const Tree&);

// ancestors (versione ricorsiva) restituisce una lista di Label, contenente le etichette
// di tutti gli antenati del nodo l ESCLUSA l'etichetta del nodo stesso
list::List ancestorsR(const Label, const Tree&);

// ancestors (come sopra ma versione iterativa)
list::List ancestorsI(const Label, const Tree&);

// leastCommonAncestor restituisce l'etichetta del minimo antenato comune
// ai nodi etichettati con label1 e label2
Label leastCommonAncestor(const Label, const Label, const Tree&);

// member restituisce
// true se il nodo etichettato con la label l appartiene all'albero t e false altrimenti
bool member(const Label, const Tree&);

// numNodes restituisce il numero di nodi nell'albero t
// mediante una visita ricorsiva in depthfirst
int numNodes(const Tree&);

// createEmptyTree restituisce l'albero vuoto
Tree createEmpty();
}

/* Funzioni che non caratterizzano il TDD Tree, ma che servono per input/output */
tree::Tree readFromFile(string);
void printTree(const tree::Tree&);
```

IMPORTANTE: nel file `tree-chsib.cpp` è ***riportata una descrizione più dettagliata per ogni funzione*** da implementare che non riportiamo qui per brevità.

3 Note su implementazione

Il nodo di un albero è caratterizzato dall'etichetta, il puntatore al primo figlio e il puntatore al prossimo fratello
NON manteniamo il puntatore al padre, anche se sarebbe comodo per alcune funzioni (ma del tutto inutile per altre)

```
struct tree::treeNode {
    Label label;
    treeNode *firstChild;
    treeNode *nextSibling;
};
```

- Si noti che per due delle funzioni, `deleteElem` e `ancestors`, si richiede esplicitamente un'implementazione iterativa e un'implementazione ricorsiva.
- Si noti che le funzioni ancestors (antenati) restituiscono la lista degli antenati di un nodo. Per realizzare queste funzioni avrete bisogno del TDD lista e quindi di due file `.h` e `.cpp` per implementare le liste. Nel codice da completare troverete una delle implementazioni viste durante il corso (`list-array.h` e `list-array.cpp`). Usate quella e quindi non usate i `Vector` del C++. Le liste servono solo per questo scopo e NON per implementare i fratelli nell'albero.
- la funzione `leastCommonAncestor`, chiamata su due nodi di cui uno è la radice cosa restituisce? Nulla, perché la radice non ha ancestors e non possono esserci quindi ancestors in comune.

4 Suggerimenti

- Le prime due funzioni da implementare sono la `addElem`, che si può testare dal main selezionando l'opzione "b", e la `printTree`
- addElem viene richiamata dalla funzione `readFromStream(istream& str)` ed è quindi necessaria per leggere dati da file
- Si consiglia di implementare poi le funzioni più facili (`member`, `father`, `degree`, `numNode`) e di affrontare solo all'ultimo la funzione `deleteElem`, nelle sue due varianti, e le funzioni `ancestors`, nelle sue due varianti, e `leastCommonAncestor`.

5 Formato file di Input

Il formato dei file che contengono la rappresentazione degli alberi è il seguente:

```
radice
radice nodo1 nodo2 nodo3 nodo4 ...
nodo1 nodo5 nodo6 nodo7 ...
```

ovvero la prima riga del file deve contenere l'etichetta della radice e le righe seguenti devono contenere come prima etichetta quella di un nodo – che deve già essere stato elencato prima – seguita dalle etichette dei suoi figli.
Esempio ("liberamente" tratto dalla classificazione dei mammiferi di Carlo Linneo):

```
mammalia
mammalia eutheria
eutheria primates
primates haplorrhini
haplorrhini catarrhini platyrrhini
catarrhini cercopithecoidea hominoidea
hominoidea hominidae hylobatidae
hominidae homininae ponginae
homininae gorilla homo pan
homo homoAbilis homoErectus homoSapiens
homoSapiens homoSapiensNeanderthaliensis homoSapiensSapiens
ponginae pongo
```

6 Formato di Stampa

La funzione di stampa deve stampare l'albero in maniera strutturata usando l'indentazione per rendere esplicito il livello di un nodo nell'albero. Ecco come deve essere stampato l'albero preso come esempio sopra:

```
mammalia
--eutheria
----primates
------haplorrhini
--------platyrrhini
--------catarrhini
----------hominoidea
------------hylobatidae
------------hominidae
--------------ponginae
----------------pongo
--------------homininae
----------------pan
----------------homo
------------------homosapiens
--------------------homosapienssapiens
--------------------homosapiensneanderthaliensis
------------------homoerectus
------------------homoabilis
----------------gorilla
----------cercopithecoidea
```

Se non riuscite a implementare la stampa strutturata così come indicato dovete comunque stampare le informazioni sull'albero in modo che sia chiara la struttura gerarchica, ad esempio riproponendo lo stesso formato del file di input, in cui è chiaro "chi

è il padre di chi". Una stampa che elenchi i nodi dell'albero perdendo l'informazione della struttura gerarchica è, in questo contesto, non corretta.

7 Tests manuali

Il file `main.cpp` contiene il `main` di un programma per aiutarvi a svolgere dei tests, in modo simile a quanto fatto nei precedenti laboratori.
Per potere usare questo programma con la vostra nuova implementazione, potete compilarlo così:
`g++ -std=c++11 -Wall tree-main.cpp string-utility.cpp tree-chsib.cpp list-array.cpp -o tree-main`
e poi eseguirlo con `./tree-main`.

Nella traccia trovate anche il file `homo.txt` che riporta il file di input per la classificazione di Carlo Linneo vista sopra.
Per svolgere dei test preliminari potete creare delle versioni semplificate di quel file oppure crearne altri ad-hoc a vostro piacimento.

8 Consegna

Per la consegna, creare uno `zip` con tutti i file forniti.