LAB7

Funzione Member: Questa funzione ha lo scopo di controllare se esiste l'etichetta l (se il nodo corrente ha etichetta l).
La prima if molto semplice, controlla se l'albero è vuoto.
La seconda if molto facile, controlla se l'etichetta dell'albero esiste, restituisco true.
Ora facciamo la ricorsione:
Creo un puntatore auxT che punta al firstChild, finchè auxT è diverso da emptyTree (while) scorro tutti i nextSibling solo se la chiamata ricorsiva mi da false (!member(l, auxT).
Else return true, alla fine della funzione ho un return False in modo da entrare sempre nel ciclo while, se dopo tutte le chiamate necessarie ho ancora false vuol dire che non esiste l'etichetta e allora la funzione mi resituisce false.
Esempio: vogliamo cercare R.
Partiamo da A , A non è R.. allora vado da suo figlio che è M. Da M passo a C. Ora C non ha altri figli e allora scorro tutti i fratelli raggiungendo O. Da O ritorno ad M e da M passo a L.
Da L vado a R che è il nodo che cercavamo, la funzione restituisce TRUE.

Funzione Father: 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.

Devo usare una funzione ausiliare di father, partiamo a spiegare questa:
hasChildWithLabel: funzione ausiliaria di father:
restituisce true se il nodo puntato da t ha un figlio con l'etichetta l.
Creo un puntatore child che punta al first child, finchè il child non è vuoto entro nel ciclo e se child -> label == l allora restituisco true sennò scorro i suoi fratelli .

Ora torniamo nella funzione father.
Creo un puntatore che punta al firstChild e un Label auxL.
Entro nel ciclo while (la sua condizione è quella che il child non sia vuoto) e ho il seguente codice:
auxL = father(l, child);: Questa riga richiama in modo ricorsivo la funzione father per cercare il padre del nodo con etichetta l all'interno del sottoalbero radicato nel nodo child. L'etichetta del padre trovato viene assegnata alla variabile auxL.
if (auxL != emptyLabel) return auxL;: Dopo aver ottenuto l'etichetta del padre, viene controllato se questa è diversa da emptyLabel. Se è così, significa che è stata trovata l'etichetta del padre del nodo con etichetta l. In tal caso, la funzione restituisce l'etichetta del padre.
else child = child->nextSibling;: Se non viene trovato un padre nel sottoalbero radicato nel nodo child, la ricerca continua esaminando il prossimo fratello del nodo child. Questo viene fatto assegnando a child il puntatore al prossimo fratello attraverso l'attributo nextSibling.

In sostanza, questo blocco di codice si occupa di cercare ricorsivamente il padre del nodo con etichetta l all'interno del sottoalbero radicato nel nodo child, fino a quando non viene trovato o fino a quando non viene esplorato tutto il sottoalbero.

Ricapitolando:

La funzione father ha il compito di restituire l'etichetta del padre del nodo con etichetta l, all'interno dell'albero t, se tale nodo esiste e se ha un padre.

Ecco come funziona:

Se l'albero è vuoto, la funzione restituisce emptyLabel, indicando che non c'è alcun nodo con l'etichetta l all'interno dell'albero t.
Se il nodo con etichetta l è la radice dell'albero, la funzione restituisce l'etichetta della radice stessa, poiché la radice non ha un padre.
Se il nodo con etichetta l è un nodo interno (non la radice) e ha un padre, la funzione risale l'albero ricorsivamente per trovare il padre del nodo con etichetta l. Questo avviene esplorando i figli di ogni nodo finché non viene trovato il nodo con etichetta l. Quando viene trovato, la funzione risale l'albero fino alla radice, mantenendo traccia delle etichette dei nodi visitati, e restituisce l'etichetta del primo genitore trovato.
La funzione ausiliaria hasChildWithLabel viene utilizzata per verificare se un dato nodo ha almeno un figlio con l'etichetta cercata. Questa verifica è utile per determinare se un nodo può avere un padre o meno. Se un nodo ha almeno un figlio con l'etichetta cercata, significa che il nodo ha un padre, altrimenti no. La funzione father fa uso di questa verifica per decidere se risalire l'albero o restituire emptyLabel nel caso in cui il nodo non abbia un padre.

Funzione getNode: funzione ausiliaria utile per esempio per Degree ma anche per altre funzioni. Il suo scopo è il seguente:

Questa funzione cerca ricorsivamente un nodo con l'etichetta l nell'albero t.
Se l'albero è vuoto o il nodo corrente ha l'etichetta cercata, restituisce il nodo corrente.
Altrimenti, cerca ricorsivamente nei figli del nodo corrente.

Vediamo come si implementa:

Ho due **if** :
la prima controlla se l'albero è vuoto o l'etichetta è vuota, restituisco il nodo vuoto.
La seconda controllo se l'etichetta dell'albero è quella cercata, restituisco t.

Creo un puntatore auxT che punta al firstChild e un altro puntatore resNode.
Faccio una Chiamata ricorsiva di getNode su ciascuno dei figli di t, finché una delle chiamate non restituisce un valore diverso da emptyTree.
Se ottengo emptyTree rientro nella prima if, non ho trovato cercando in questo sottoalbero, devo proseguire la scansione dei fratelli.
Se trovo restituisco resNode.

Funzione children: children restituisce una lista di Label, contenente le etichette di tutti i figli nell'albero t del nodo etichettato con l.
Creo un puntatore Tree auxT e lo itero con la chiamata di getNode.
Creo la lista con : list::List lst = list::createEmpty();
Se auxT non è empty allora ho un puntatore child che punta a auxT -> firstChild.
Finchè child non è vuoto aggiungo (in coda) la label del figlio e incremento child (child = child -> nextSibling).
La funzione ritorna lst;

N.B *Per aggiungere in coda uso list::addBack(child -> label, lst)*

Funzione degree: degree restituisce il numero di archi uscenti dal nodo etichettato con l; restituisce -1 se non esiste nessuna etichetta l nell'albero.
La funzione degree restituisce il numero di archi uscenti dal nodo etichettato con l nell'albero t. Gli "archi uscenti" si riferiscono ai collegamenti diretti dal nodo etichettato con l ad altri nodi nell'albero.

Il codice controlla prima se il nodo con etichetta l esiste nell'albero t utilizzando la funzione member. Se il nodo non esiste, la funzione restituisce -1. In caso contrario, viene chiamata la funzione children per ottenere la lista dei figli del nodo con etichetta l, e la dimensione di questa lista viene restituita come grado del nodo.

VARIANTE: posso fare una variante usando la funzione getNode per rendere tutto ancora piu facile.
Il codice sarebbe il seguente:

```
int tree::degree(const Label l, const Tree& t)
{
  // Troviamo il nodo con etichetta l nell'albero
  Tree node = getNode(l, t);

  // Se il nodo non è presente nell'albero, restituiamo -1
  if (isEmpty(node)) {
    return -1;
  }

  // Contiamo il numero di figli del nodo
  int numChildren = 0;
  Tree child = node->firstChild;
  while (child != emptyTree) {
    numChildren++; // Incrementiamo il conteggio dei figli
    child = child->nextSibling; // Passiamo al prossimo figlio
  }
```

```
    return numChildren;
}
```

In questa implementazione, troviamo il nodo con etichetta l nell'albero utilizzando la funzione ausiliaria getNode. Se il nodo non è presente, restituiamo -1. Altrimenti, contiamo il numero di figli del nodo e restituiamo questo valore come il grado del nodo.

Funzione numNodes: numNodes restituisce il numero di nodi nell'albero t mediante una visita ricorsiva in depthfirst.
Se la lista è vuota restituisco 0.
Negli altri casi creo un puntatore auxT al first child di t, creo un contatore res inizializzato a 0.
Per ogni nodo, chiamiamo ricorsivamente la funzione numNodes passando il nodo corrente come argomento. Questo calcola il numero di nodi nel sottoalbero radicato in quel nodo e lo aggiunge al risultato res. Dopo aver esaminato tutti i nodi figli del nodo corrente, aggiorniamo il puntatore auxT al fratello successivo del nodo corrente, utilizzando il campo nextSibling.
Una volta esaminati tutti i nodi dell'albero, aggiungiamo 1 al conteggio totale res per considerare anche il nodo corrente e restituiamo il risultato.

Funzione addElem: addElem aggiunge il nodo etichettato con labelOfNodeToAdd come figlio del nodo etichettato con labelOfNodeInTree.
Ci serve una funzione ausiliare createNode, partiamo da lei.

createNode: data un'etichetta l crea il nodo con quella etichetta e ne restituisce il puntatore.
Andiamo a creare un nuovo nodo e gli assegniamo tutti i parametri necessari.

Tornando alla nostra funzione addElem partiamo col caso particolare.

Caso aggiunta alla radice, questo caso puo succedere solo se labelOfNodeInTree == emptyLabel e t è empty.
Assegniamo t ad createNode e facciamo un return Ok.

Il secondo caso da controllare è quello di verificare se esiste gia la mia label, richiamiamo la funzione member.

Ora per tutti gli altri casi:

Creo un puntatore auxT e gli assegno tramite getNode il nodo dell'albero con l'etichetta labelOfNodeInTree.
Se auxT è nullo vuol dire che non esiste il nodo e restituisco FAIL.
Se invece esiste allora creo un nodo child con l'etichetta labelOfNodeToAdd.

Assegno poi:
child->nextSibling = auxT->firstChild; il primo fratello di child sarà quello che era il primo figlio di auxT
auxT->firstChild = child; child diventa il primo figlio di auxT.

Ritorno OK.

Funzione deleteElemI: 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.

Il primo controllo da effettuare è il seguente:

SE member ritorna false allora ritorno FAIL in quanto non esiste il nodo.

Creo un puntatotre al padre e tramite getNode cerco il puntatore al padre del nodo da rimuovere.

Dobbiamo ora tenere conto di due casi:

1) Radice
2) Nodo interno

Partiamo dal primo caso.
Se il puntatore al padre (fatherTree) è vuoto vuol dire che stiamo tentando di eliminare la radice.
Possiamo eliminare la radice solo se non ha figli, quindi richiamiamo la degree e deve darmi come valore 0.
Se ottengo 0 allora posso procedere l'eliminazione. Senno return FAIL.

Ora entriamo nella else, ovvero sto tentando di rimuovere un nodo interno.
Creo un puntatore al nodo da rimuovere (nodeToRemove).
Creo un puntatore al nodo dell'ultimo fratello del nodo da rimuovere(lastCh).
Per questa ho bisogno di una funzione ausiliare che chiamo lastChild e gli passiamo fatherTree.

lastChild : dato un puntatore a un nodo restituisce il puntatore al suo ultimo figlio ovvero a quello piu a destra nella catena dei fratelli.
Controllo se l'albero è vuoto o se il primo figlio è vuoto, se siamo in uno dei due casi facciamo un return a emptyTree.

Dopo questo controllo creo due puntatori. Il pirmo è il prevChild, lo inizializzo col t -> firsthChild. Il currentChild lo pongo al nextSibling di prevChild .

Il ciclo è simile a quello che facevamo nelle liste semplici. Nelle liste semplici facevamo per esempio prev = cur e cur = cur -> next.
In questo caso facciamo:
while(!isEmpty(currentChild) {
prevChild = currentChild;
currentChild = currentChild -> nextSibling;

alla fine della funzione facciamo un return di prevChild (sarebbe l'ultimo nodo).

Torniamo alla funzione deleteElemI.
Poniamo al nextSibling di lastCh al firstChild del nodo da rimuovere, i figli del nodo da rimuovere diventano suoi fratelli.
Creiamo un altro puntatore di nome prevSibl e gli assegno la chiamata della funzione ausiliare prevSibling. (recupero il puntatore al fratello precedente rispetto al nodo da rimuovere).

PrevSibling: dato un albero t e un'etichetta l, restituisce il puntatore al fratello precedente (ovvero più a sinistra) a quello con l'etichetta data, se esiste nell'albero.
Il primo controllo da fare è quello di verificare se t è empty o se t -> firsChild è empty o se l == emptyLabel, se si verifica uno di questi casi facciamo un return emptyTree.
La seconda if è quello di verificare se la label del t -> firstChild corrisponde con la label passata alla funzione, in caso positivo restituisco emptyTree in quanto non ha fratelli.
Se non si verifica questa if allora passiamo a creare i seguenti puntatori:

Tree prevChild = t->firstChild; firstChild non e' vuoto; pongo prevChild uguale al firstChild

Tree currentChild = prevChild->nextSibling; pongo currentChild uguale al figlio successivo

Tree auxT; ci serve per dopo(nel caso di ricorsione)

Ora entriamo nel ciclo, la condizione è che currentChild sia diverso da empty e che currentChild -> label sia diverso da l.
L'interno di questo ciclo è simile a quello delle liste semplice, vado ad assegnare currentChild a prevChild e incremeneto currentChild aggiornandolo al suo nextSibling.

Controlliamo se alla fine di questo ciclo siamo arrivati alla label, currentChild deve essere diverso da empty e la sua label deve corrispondere a l, in caso positivo facciamo return prevChild.
In caso negativo proseguiamo con la ricorsione, ovvero:
aggiorno currentChild inizializzandolo a t -> firstChild.
While finche curretnChild è diverso da emptyTree.
Assegno ad auxT la chiamata ricorsiva della funzione (prevSibling) passando come parametro l e currentChild.
Se auxT è diverso da empty allora restituisco auxT sennò mi sposto al nextSibling di current.

Ritorniamo alla deleteElemI, se la chiamata a prevSibling mi da un empty allora non abbiamo nessun fratello precedente, il nodo da rimuovere è quindi il primo.
Dovviamo cambiare il puntatore al firstChild nel padre.
Quindi aggiorniamo fatherTree -> firstChild con (fatherTree -> firstChild) -> nextSibling.

Altrimenti devo "saltare" il nodo da rimuovere nella catena dei Fratelli , quindi:
prevSibl -> nextSibling = nodeToRemove -> nextSibling.

Alla fine di tutto eliminiamo il nodo e restituiamo OK.

Funzione deleteElemR: funzione ricorsiva, decisamente piu facile rispetto a quella iterativa.
Controlliamo se il nodo da rimuovere è la radice, per farlo usiamo una if e mettiamo che t deve essere diverso da empty e che t -> label sia == a l.
Come nella scorsa possiamo rimuovere soltanto se la degree mi restituisce 0.
Alla fine di tutto facciamo un return a deleteElemAux(l, t).

deleteElemAux(l, t): funzione ricorsiva ausiliare , rimuove dall'albero il nodo etichettato con la label l, se esiste.

Se t è vuoto restiutisco FAIL in quanto non abbiamo nulla da eliminare.

Facciamo una chiamata alla funzione hasChildWithLabel, se t è il padre del nodo da rimuovere allora rimuovo il figlio di t etichettato con l e restituisco OK.
Per eliminare il figlio devo chiamare una funzione ausiliare di nome deleteChild.

DeleteChild: funzione ausiliaria, viene chiamata solo se si e' verificato che il nodo etichettato con l e' uno dei figli di t.