Università degli Studi di Udine

DIPARTIMENTO DI SCIENZE MATEMATICHE, INFORMATICHE E FISICHE
Corso di Laurea Triennale in Informatica

Relazione del progetto
per l'insegnamento
Laboratorio di Algoritmi e Strutture Dati

Progettazione di BST a Universo finito per l'analisi di percorsi ottimi

Studente:
Filippo Callegari
callegari.filippo@spes.uniud.it
Matricola 128602

Professore dell'insegnamento:
Alberto Policriti

Anno Accademico 2016–2017 - Secondo Semestre

Indice

1 Introduzione

Il seguente progetto, che risolve il problema descritto nella sezione successiva, è stato strutturato principalmente in C[1], con l'ausilio di "Python" (ed i suoi vari pacchetti interni, come "PyDOT"[2] e "NetworkX"[3]) per il controllo e/o confronto delle soluzioni progettuali del mio progetto. Tutto il progetto è *"Valgrind*[4] *compliant"*: tutti i moduli gestiscono in maniera ordinata e protetta tutta l'*heap* del processo.

1.1 Compilazione del progetto

Per la compilazione del progetto ci si affida al semplicissimo `Make`: attraverso `Makefile` possiamo quindi generare i due eseguibili richiesti per il progetto. I comandi riconosciuti da make sono:

- `$ make all`: genera tutti gli eseguibile richiesti (e descritti di seguito);
- `$ make soluzione`: genera l'eseguibile per la risoluzione del problema e genera l'eseguibile `soluzione`;
- `$ make tempi`: genera l'eseguibile per il calcolo dei tempi e genera l'eseguibile `tempi`;
- `$ make clean`: elimina tutti i file di compilazione e gli eseguibili generati.

Non sono forniti gli ambienti per Python, i quali sono a carico di chi è interessato ad utilizzare i programmi ausiliari recuperarli attraverso `pip`[5].

1.2 Struttura del progetto

Il progetto si divide in 4 grandi categorie:

1. `./`: root directory del progetto. All'interno si trovano il file `main.c` e i file per la generazione dei vari eseguibili;
2. `./lib/`: directory per gli header. Mantiene la medesima struttura della cartella dei file sorgente, in manieria di recuperare i file sorgente velocemente;
3. `./src/`: directory per gli eseguibili. Mantiene la medesima struttura della cartella dei file header, in maniera di recuperare i file header velocemente;
4. `./test/`: directory per i vari file di test case.

Di seguito ne spieghiamo il contenuto.

[1] per il linguaggio C si utilizza lo standard "gnu11"!
[2] una interfaccia Python per GraphViz e il linguaggio DOT
[3] NetworkX è un pacchetto Python per la creazione, la manipolazione e lo studio della struttura, delle dinamiche e delle funzioni di reti complesse
[4] Valgrind è uno strumento per il debug di problemi di memoria, la ricerca dei memory leak ed il profiling del software.
[5] Pip Installs Packages

1. Librerie

All'interno di questa cartella si trovano tutti i file di libreria. Ogni libreria è presente nella cartella del `modulo` ad essa legata, rispettando la path `./lib/modulo/header.h`. All'interno del file sarà presente il commento di documentazione della funzione dichiarata nell'header.

I moduli presenti quindi sono:

1. `BST`: contiene la definizione di tutte le "well known function" (di seguito sarà sempre scritto come *WKF*) sui *BST*[6];
2. `LBST`: contiene la definizione della categoria dei *"Linear BST"* (di seguito *LBST*). Contiene tutte le *WKF* dei BST ma con la particolarità che tutte le funzioni che normalmente richiedono $O(h)$, quali *"inserimento"*,*"ricerca"*,*"eliminazione"* in un BST, richiederanno solo $O(|\Sigma|)$[7];
3. `CLBST`: contiene la definizione dei *"Counted LBST"* (di seguito *CLBST*). Come per i *LBST*, anche i *CLBST* contengono le *WKF* dei BST, con un'ulteriore particolarità: impiegano meno memoria per l'albero, specie se nei *LBST* ho più di $2 \times |\Sigma| - 1$ nodi[8];
4. `prj`: contiene gli header per i sorgenti veramente utilizzati nel progetto, ma rivisti e semplificati per aumentare le performance del problema risolto in questo progetto;
5. `RandomMaker`: contiene gli header per la libreria che genera numeri casuali basata sul meccanismo dato a disposizione dal sistema operativo stesso, attraverso il file `/dev/urandom`;
6. `timing`: contiene gli header per il *Pseudo Random Number Generator* definito a lezione e per le funzioni utilizzate per calcolare il tempo del progetto.

2. Sorgenti

Si rispetta la filosofia del modello applicato per le librerie, e quindi si avrà come percorso sempre `./src/modulo/sorgente.c`. In aggiunta sono presenti all'interno delle funzioni i relativi commenti alle relative istruzioni più interessanti *(o complicate)*. Per il contenuto dei relativi moduli si fa sempre riferimento alla sezione *"Librerie"*.

3. Test case

All'interno della cartella `./test/` si possono trovare tutti i file sorgente di test case con le varie cartele legate. Veri ed unici sorgenti degni di nota si trovano in:

- `./test/PoliReqTime/`: qui dentro si trova il sorgente utilizzato per generare il misuratore di tempi come spiegato a lezione;

[6] BST: Binary Search Tree, ***piccoli*** cenni di teoria saranno fatti in seguito
[7] tutte queste notazioni saranno sempre spiegate nella sezione successiva
[8] caso particolare, sarà spiegato sempre in seguito

- `./test/testCLBST/` : qui dentro si trovano i file di prove minime per la verifica dei tempi teorizzati come da documentazione;
- `./test/testentropy/` : qui si trovano piccoli esempi di verifica della distribuzione dei dati generati *pseudocasualmente*[9]. Unico file non generato da me è la *suite "entropy"* (la quale non fa effettivamente parte del corso, messa a disposizione da *"fourmilab.ch"*);
- `./test/testLBST/` : qui si trovano esempi e prove di utilizzo della classe di alberi *LBST*. Nel file `./test/testLBST/timingLBST.c` risiedono alcune prime sperimentazioni su come effettivamente l'albero lavori in $O(|\Sigma|)$ per le operazioni che normalmente hanno tempo $O(h)$;
- `./test/testSol` : contiene il file in Python utilizzato per la verifica della soluzione del problema con *cross-testing*.

[9] non propriamente vero: si è vero che i pc sono macchine deterministiche, ma bisogna tener conto che un pc è soggetto a interrupt di vario genere: essendo il file /dev/urandom generato dall'insieme di segnali "fisico-logici" in possesso dalla macchina, possiamo dire quasi con certezza che in realtà siano veramente casuali! Unico problema noto è che chiunque abbia accesso *fisico* alla macchina potrà avere sempre il nostro medesimo risultato...

2 Presentazione del problema

2.1 Il problema

Definizione 1 (Alfabeto) *si definisce come alfabeto del problema*

$$\Sigma = \{a, b, \ldots, z\},$$

l'insieme di caratteri con intervallo $\big[97; 122\big] \subset ASCII$*, ove l'intervallo è sottoinsieme della codifica ASCII, la quale è definita dall'intervallo* $\big[0; 127\big]$*. Inoltre d'ora in poi denoteremo con* σ *come* $\sigma \in \Sigma$*, e* σ_i *la lettera dell'alfabeto in posizione i-esima, secondo ordine lessico-grafico (d'ora in poi OLG).*

Definizione 2 (Relazioni d'ordine) *siano* $i, j \in \Sigma$*. Si definiscono le relazioni d'ordine nell'alfabeto con i seguenti simboli:*

- $i \prec j$*: si dice che i precede j in quando i è precedente secondo OLG;*
- $i \preceq j$*: si dice che i precede o è uguale a j secondo OLG;*
- $i \succ j$*: si dice che i segue j secondo OLG;*
- $i \succeq j$*: si dice che i segue o è uguale j secondo OLG;*
- $i = j$*: si dice che i e j sono il medesimo simbolo.*

Definizione 3 (Stringa) *si definisce come* ***stringa propria*** *(o più semplicemente* ***stringa****)* α *la stringa* $\alpha \in \Sigma^*$*, con* Σ^*[1] *detto universo linguistico.*
Si definisce invece come ***stringa impropria*** $\bar{\alpha}$ *la stringa* $\bar{\alpha} \notin \Sigma^*$*, e quindi qualunque stringa contenga un carattere non definito nell'alfabeto.*
Denoteremo d'ora in poi come $\alpha\big[i\big]$ *la lettera in posizione i-esima in* α*.*
Inoltre, è noto che $\alpha\big[0\big] \equiv \sigma_r$ *e che* $\alpha\big[i\big] \equiv \alpha_i$*.*
Per completezza, ribadiamo che con $|\alpha|$ *si denota la lunghezza di* α*.*

Definizione 4 (Albero della stringa) *data* α*, sia* $T\big(\alpha\big)$ *l'albero binario di ricerca (BST) ottenuto come segue:*

- *se* $\alpha = \epsilon$ *(e quindi* $|\epsilon| = |\alpha| = 0$*) allora* $T\big(\alpha\big) = \emptyset$*;*
- *se* $|\alpha| > 0$ *(e quindi* $\alpha \neq \epsilon$*):*
  - *la radice di* $T\big(\alpha\big)$ *è il nodo etichettato con* $\alpha\big[0\big]$*;*
  - *il sotto-albero di sinistra di* $T\big(\alpha\big)$ *è* $T\big(\lambda\big)$*, con* λ *la sotto-stringa di* $\alpha\big[1, \ldots, |\alpha|\big]$ *costituita dai caratteri lessicograficamente minori o uguali di* σ_r *(e quindi* $\big\{\alpha_i \mid i \in \big[1; |\alpha|\big] \wedge \alpha\big[i\big] \preceq \sigma_r\big\}$*);*
  - *il sotto-albero di destra di* $T\big(\alpha\big)$ *è* $T\big(\rho\big)$*, con* ρ *la sotto-stringa di* $\alpha\big[1, \ldots, |\alpha|\big]$ *costituita dai caratteri lessicograficamente maggiori di* σ_r *(e quindi* $\big\{\alpha_i \mid i \in \big[1; |\alpha|\big] \wedge \alpha\big[i\big] \succ \sigma_r\big\}$*);*

Per comodità definiamo la funzione $leaf(T\big(\alpha\big))$ *come la funzione che restituisce tutte le foglie dell'albero.*

[1] noto come operatore di Kleene

Definizione 5 (Cardinalità) *data una stringa α, denoteremo $\mathcal{W}_\sigma$ come la cardinalità delle "comparse" della lettera σ in α. Assumiamo d'ora in poi che*

$$m = \min_{i\in[0;|\Sigma|]} \{\mathcal{W}_{\sigma_i}\},$$
$$M = \max_{i\in[0;|\Sigma|]} \{\mathcal{W}_{\sigma_i}\}.$$

Ora che abbiamo definito m e M, σ_M sarà la lettera con cardinalità massima, e σ_m la lettera con cardinalità minima.

Definizione 6 (Cammino tra due foglie) *dato $T(\alpha)$, e date due sue foglie u,v, definiamo come $u \rightsquigarrow v$ il percorso di due foglie raggiungibili nell'albero. Definiamo invece come $\mathscr{W}_{u\rightsquigarrow v}$ il peso del cammino tra i due.*

Problema 1 (Costruzione del grafo) *data una stringa α, definito $T(\alpha)$, assunti m, M, costruire un grafo non orientato $G = \langle V, E\rangle$, tale che:*

1. $V = \{$*tutte le foglie presenti in* $T(\alpha)\}$;
2. $E = \{(u,v) \mid u, v$ *foglie in* $T(\alpha)$, $m \leq \mathscr{W}_{u\rightsquigarrow v} \leq M\}$.

2.2 Input ed output

Tutto l'input viene letto dallo standard input, e può essere una stringa arbitrariamente lunga[2], la quale deve terminare con il carattere "EOF", normalmente presente nello standard input se non presente nessun carattere da leggere. Qualunque altro carattere letto dallo standard input porterà a terminare immediatamente il programma.
Lo standard output per l'eseguibile `soluzione` presenterà quindi, dopo la lettura del carattere "EOF" la soluzione richiesta, quale $|V|$ e $|E|$, seguita dalla stampa in formato DOT del grafo; Nel formato DOT, la rappresentazione di un vertice sarà quindi σ_i `[label='`i`']`, dove i sarà la posizione della lettera σ_i.
Invece per l'eseguibile `tempi` stamperà in formato `x` $E(\mathcal{X})$ $Var(\mathcal{X})$, dove `x`$= |\alpha|$, $E(\mathcal{X})$ il valore atteso per α e $Var(\mathcal{X})$ sarà la varianza per α.

2.3 Considerazioni iniziali notevoli

In questa sezione verranno mostrate e dimostrate alcune piccole considerazioni sullo studio del problema.

Teorema 1 (Partizionamenti di Σ^*) *fissato $n \in \mathbb{N}$, sia Σ^n e sia data α t.c. $|\alpha| = n$ e $T(\alpha)$. Esiste un partizionamento di Σ^n t.c. se α,$\beta \in \Sigma^n$, $\alpha \neq \beta$, $\forall\iota \in \alpha(\exists\kappa \in \beta(\iota = \kappa \wedge \mathcal{W}_\iota = \mathcal{W}_\kappa))$, e l'ordine di inserimento delle lettere per la costruzione di $T(\beta)$ rende identica la struttura per $T(\alpha)$,allora $\exists p$ t.c. Σ^n_p, $\alpha \in \Sigma^n_p \wedge \beta \in \Sigma^n_p$ sse $T(\alpha) \equiv T(\beta)$, altrimenti $\alpha \notin \Sigma^n_p \vee \beta \notin \Sigma^n_p$.*

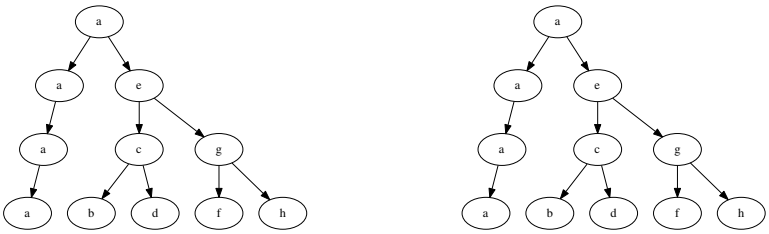
[2]secondo definizione dei limiti del sistema operativo in uso, normalmente per una shell Linux il buffer di stdin è di 64KB, eccetto casi particolari. Consultare le linee guida del proprio OS per saperne di più

Dim: siano date α e β, t.c. $|\alpha| = |\beta| = n$, e $\forall\iota \in \alpha\big(\exists\kappa \in \beta\big(\iota = \kappa \wedge \mathcal{W}_\iota = \mathcal{W}_\kappa\big)\big)$. Costruiamo per α e β rispettivamente $T\big(\alpha\big)$ e $T\big(\beta\big)$. Se $T\big(\alpha\big) \equiv T\big(\beta\big)$, allora per certo avremo che l'ordine di inserimento delle lettere nell'albero risultante è equivalente, allora α,$\beta \in \Sigma_p^n$, dove p identifica la classe di partizione di Σ_p^n. Se $T\big(\alpha\big) \not\equiv T\big(\beta\big)$, allora l'ordine di inserimento delle lettere nell'albero non è equivalente o le stringhe non contengono le medesime lettere, e questo non rispetta le ipotesi iniziali.

■

Esempio 1 (Stringhe per alberi equivalenti) *prendiamo come esempio* $\alpha =$ "*aaaaecbdgfh*" *e* $\beta =$ "*aeacbadgafh*". *Se costruiamo* $T\big(\alpha\big)$ *e* $T\big(\beta\big)$, *vedremo che i due alberi sono equivalenti.*

Albero per α	*Albero per* β



Lemma 1 *per definizione dei partizionamenti di* Σ^*, *ricordiamo che:*

- $\Sigma^* = \bigcup_{n\in\mathbb{N},p\in\mathbb{N}} \Sigma_p^n$;
- $\forall p \in \mathbb{N}\big(\ \nexists\bar{p} \in \mathbb{N}\big(\Sigma_p^* \cap \Sigma_{\bar{p}}^* \neq \emptyset\big)\big)$

Corollario 1 *come sappiamo dal corso di "Fondamenti di informatica",* Σ^* *è numerabile ed infinito, come son numerabili ed infiniti i suoi partizionamenti. Sono invece finiti e numerabili i partizionamenti e le stringhe in questi, una volta definito* Σ^n.

Teorema 2 (Totalità degli archi in G) *sia G il grafo* $G=\langle V,E\rangle$ *per* α. *La cardinalità massima di E sarà data da*

$$|E| = \binom{|V|}{2} = \frac{(|V|)\times(|V|-1)}{2}.$$

Se $|V| = |\Sigma|$, *allora*

$$|E| \leq \frac{26\times25}{2} = 325.$$

Dim: si prenda in considerazione $\alpha =$ "*zzzzzzzzzzzzzzzzzzzzzzzzzzaabbccddeeffgghhiijjkkllmmnnooppqqrrssttuuvvwwxxyyz*": si vedrà che G per α sarà un grafo connesso[3] *25-completo*[4]. Inoltre, $\nexists\bar{\sigma}(\bar{\sigma} \in \Sigma \mid \bar{\sigma} \succ \sigma_{|\Sigma|})$ ed essendo $|\Sigma| = 26$, non esiste configurazione tale per cui possano esistere più di 325 archi. Si invita a leggere il corollario seguente per ulteriori dettagli sulla totalità degli archi

[3] un grafo G è detto connesso se, per ogni coppia di vertici (u, v), u,$v \in V$ esiste un cammino che collega u a v.

[4] un grafo completo G è un grafo nel quale ogni vertice è collegato a tutti i vertici rimanenti.

■

Corollario 2 *per conseguenza del "Partizionamenti di Σ^*", sappiamo con certezza che la stringa citata nella dimostrazione non è l'unica a generare un grafo "k-completo".*

A questo punto possiamo ragionare su tutte le varie configurazioni di α tali che possa avere un grafo connesso e k-completo.

Lemma 2 *esiste un solo schema di partizionamento di Σ^* tale che tutte stringhe appartenenti ad una delle due partizioni possa generare un grafo "k-completo".*

Lemma 3 *sia $\Delta \subseteq \Sigma$ l'insieme minimo dei caratteri per $\alpha \in \Delta^*$. Esisterà sempre α t.c. G per α sia $|\Delta|$-completo.*

Definizione 7 *d'ora in poi aggiungiamo Δ come $\Delta \subseteq \Sigma$ l'insieme minimo dei caratteri per $\alpha \in \Delta^*$, e con δ_i la lettera in posizione i-esima secondo OLG in Δ.*

Teorema 3 *data α, sia $|V|$=$leaf(T(\alpha))$ e $c = |\Delta|$. Allora $|V| \leq c$.*

Dim: immediato per la definizione di BST e per l'albero della stringa.

■

3 BST, BST Lineari, BST Lineari Compressi

In questa sezione analizzeremo alcune importanti caratteristiche degli alberi adottati nel progetto. Nella fattispecie, analizzeremo inizialmente gli alberi binari di ricerca e le loro "well-known-function", dopodichè faremo dei confronti tra BST e gli altri alberi sviluppati nel corso del progetto.

3.1 BST o Alberi Binari di Ricerca

Definiamo fin da subito la struttura dati che identifica un BST e le loro funzioni fondamentali.

3.1.1 Identificazione di un BST

Definizione 8 (Nodo di un BST) *un nodo per un BST è un nodo definito con:*

- *un campo per il contenuto del nodo (o chiave del nodo);*
- *un puntatore rispettivamente per il padre, ed i suoi due figli, rispettivamente destro(λ) e sinistro(ρ).*

Definiamo quindi le sue caratteristiche.

Definizione 9 (Proprietà di un BST) *un BST presenta:*

- *un nodo radice (τ), il quale è l'unico a non aver padre;*
- *per ogni nodo, se è presente almeno un suo figlio, rispettivamente si avrà:*
  - *per ogni figlio sinistro, tutte le chiavi nel sotto-albero sinistro avranno un campo che saranno minori o uguali al campo del padre:*
  - *per ogni figlio destro, tutte le chiavi nel sotto-albero destro avranno un campo che saranno maggiori al campo del padre;*
- *tutti i nodi dell'albero che non presentano figli saranno chiamati foglie dell'albero (e saranno restituite dalla funzione $leafs(\tau)$.*

Notiamo quindi che non poniamo nessuna limitazione a come inserire i nodi nell'albero, se non condizioni di appartenenza al sotto-albero. Nasce quindi il problema che l'albero possa essere sbilanciato. Questa tematica verrà affrontata nella sezione riguardante i CLBST, in quanto hanno alcune proprietà interessanti in merito.

3.1.2 Funzioni fondamentali per un BST

Date queste ulteriori note, possiamo tranquillamente definire le sue funzioni.

Definizione 10 (Predecessore e successore) *definiamo come "predecessore" di un nodo il nodo con chiave massima nel sotto-albero di sinistra (e quindi il nodo tutto a sinistra nel sotto-albero sinistro), mentre definiamo come "successore" di un nodo il nodo con chiave minima nel sotto-albero di destra (e quindi il nodo tutto a destra nel sotto-albero destro).*

Definizione 11 *dato T albero BST, definiamo con $\eta = cards(\tau)$ il numero di nodi presenti in un albero, data dalla funzione*

$$card(nodo) = \begin{cases} 0 & \textit{nodo inesistente} \\ 1 + cards(nodo_\lambda) + cards(nodo_\rho) & \textit{altrimenti} \end{cases}$$

Definizione 12 (Altezza dell'albero) *dato T albero BST, definiamo come altezza dell'albero la funzione $h(\tau)$ così definita:*

$$h(nodo) = \begin{cases} 1 & \textit{se nodo è foglia} \\ \max(h(nodo_\lambda), h(nodo_\rho)) + 1 & \textit{altrimenti} \end{cases}$$

Teorema 4 *dato un albero BST T, la sua altezza sarà sempre compresa tra $\log_2(\eta) \leq h(\tau) \leq \eta$, dove se $h(\tau)$ è molto vicino a $\log_2(\eta)$ si dice bilanciato, mentre se $h(\tau)$ è molto vicino a η si dice sbilanciato.*

Dim: un albero è bilanciato per η molto grande se è costruito con inserzione di elementi in maniera totalmente casuale. Questo è dato dal fatto che ogni chiave è equiprobabilmente inserita in T, e quindi sia a destra che a sinistra dell'albero ci saranno $\frac{\eta}{2}$, e quindi l'altezza di $h(\tau_\lambda) \approx h(\tau_\rho)$.
Un albero è totalmente sbilanciato se $h(\tau) \approx \eta$. Questo è dovuto al fatto che l'inserimento non è per nulla casuale, e si stanno inserendo nell'albero valori in ordine crescente. Date queste informazioni, è immediato verificare che $h(\tau)$ è compreso sempre tra $\log_2(\eta) \leq h(\tau) \leq \eta$.

■

Ottenere le foglie

Viene qui presentato lo pseudocodice necessario ad ottenere le foglie di un BST. Verrà utilizzata una lista doppiamente concatenata.

```
1  leafs(root, lista) {
2      if(root == NIL) {
3              return;
4      }
5
6      if(right[root] == NIL && left[root] == NIL) {
7              push(lista, root);
8      }
9
10     leafs(right[root], lista);
11     leafs(left[root], lista);
12 }
```

Il tempo di esecuzione di `leafs` è di $\Theta(\eta)$, in quanto necessariamente devo scandagliare tutto l'albero per ottenere tutte le foglie.

Tutto questo ci porta a comprendere come tutte le funzioni ben note dei BST siano limitate in tempo come $O\big(h(\tau)\big)$. Si rimanda quindi all'analisi delle funzioni come definite durante al corso, e si ricorda che:

- `inserimento(`T`, nodo)` è in $O\big(h(\tau)\big)$;
- `ricerca(`T`, nodo)` è in $O\big(h(\tau)\big)$;
- `eliminazione(`T`, nodo)` è in $O\big(h(\tau)\big)$;
- `successore(`T`, nodo)` è in $O\big(h(\tau)\big)$;
- `predecessore(`T`, nodo)` è in $O\big(h(\tau)\big)$;
- `leafs(`T`, lista)` è in $\Theta(\eta)$.

3.2 LBST o BST Lineari

I LBST sono una specializzazione del BST. Godono infatti di aver definito all'inizio della propria vita il dominio applicativo $\mathbb{D}$. Questo, come vedremo, gli permette di lavorare in tempo costante sull'albero T, in quanto scaricheranno la complessità su $\mathbb{D}$ e non più su $h(\tau)$. Vengono chiamati quindi "lineari" in quanto il tempo di inserimento di η nodi dipende solamente da η stesso e non nel percorrere l'albero.

3.2.1 Identificazione di un LBST

In aggiunta alle normali componenti di un BST, i LBST necessitano della definizioni di alcune funzioni ausiliarie per il suo funzionamento, dipendenti dal dominio applicativo.

Definizione 13 (Equivalenza tra BST e LBST) *un albero LBST si dice equivalente a un albero BST se "hanno la medesima forma", e ogni etichetta dei rispettivi nodi ha medesima posizione e funzionalità su entrambi gli alberi. Viene quindi indicato* $T_{LBST} \equiv T_{BST}$.

Definizione 14 (Funzione di comparazione) *con funzione di comparazione si intende la funzione tale che riesca a definire una relazione d'ordine in* $\mathbb{D}$. *La funzione è definita*

$$\mu\colon \mathbb{D}^2 \to \{-1; 0; 1\}$$

e, dati $\iota, \kappa \in \mathbb{D}$, *definirà come segue* μ:

$$\mu(\iota, \kappa) = \begin{cases} -1 & \text{se } \iota \prec \kappa \\ 0 & \text{se } \iota = \kappa \\ 1 & \text{se } \iota \succ \kappa \end{cases}$$

Definizione 15 (Funzione di hash o mappa) *con funzione di hash definiamo una funzione tale che riesca a creare un monomorfismo tra gli elementi di* $\mathbb{D}$ *e la sua dimensione. Sia quindi*

$$\varphi\colon \mathbb{D} \to \mathbb{N}$$

funzione iniettiva[1] *tale che riesca a mappare in maniera continua ogni elemento di* $\mathbb{D}$ *in* $\mathbb{N}$*; inoltre* φ *rispetti l'ordine definito dalla funzione* μ*:*

$$\begin{aligned}\forall \iota \in \mathbb{D}\big(\forall \kappa \in \mathbb{D}\big((\mu(\iota,\kappa) = 1 \Rightarrow \varphi(\iota) > \varphi(\kappa)) \\ \vee(\mu(\iota,\kappa) = 0 \Rightarrow \varphi(\iota) = \varphi(\kappa) \\ \vee(\mu(\iota,\kappa) = -1 \Rightarrow \varphi(\iota) < \varphi(\kappa)))\big).\end{aligned}$$

Definizione 16 (Catena dinamica e mappa) *per ogni elemento in* $\mathbb{D}$*, avremo una cella di memora la quale conterrà:*

- *un puntatore al primo elemento della catena dinamica;*
- *un puntatore all'ultimo elemento della catena dinamica.*

Ogni elemento di una catena dinamica sarà quindi formato da:

- *un puntatore ad un nodo del LBST;*
- *un puntatore al nodo della catena dinamica precedente;*
- *un puntatore al nodo della catena dinamica successivo.*

Definizione 17 (Nodo di un LBST) *un nodo di un LBST si compone quindi di:*

- *tutte le componenti di un BST;*
- *un puntatore alla catena dinamica della mappa.*

3.2.2 Funzioni fondamentali per un LBST

Tutte le funzioni qui nominate andranno ad aggiornare tutte le funzioni precedentemente nominate. Tutte quelle non nominate rimarranno quindi invariate.

Ricerca

Definizione 18 (Ricerca) *la funzione ricerca restituirà sempre la prima istanza nell'albero della lettera cercata.*

La funzione di ricerca sarà quindi così definita:

```
1 ricerca(T, lettera) {
2     return first[cella[φ(lettera)]];
3 }
```

Come possiamo notare, il tempo di esecuzione per `ricerca` è in $\Theta(1)$.

[1] sarebbe accettata anche suriettiva se come meccanismo di hash accettassimo hash con collisioni, ma per semplificazione e dimensioni esigue di $|\Sigma|$, e quindi precondizioni progettuali, ho optato per una funzione iniettiva

Catena dinamica e mappa

La catena dinamica è una semplice lista doppia che tiene nota dell'ordine temporale di inserimento di ogni lettera nell'albero. La mappa invece un'istanza di memoria che memorizzerà i puntatori dell'inizio e della fine catena. Questo componente è fondamentale per il corretto funzionamento dell'albero lineare, in quanto permette di ridurre drasticamente tutti i tempi di accesso, poichè non dovremo più scorrere tutta la catena per poter arrivare all'ultima cella.

Corollario 3 (Tempi nella mappa con catena dinamica) *le funzioni di inserimento o di eliminazione nella catena dinamica vengono fatti in tempo* $\Theta(1)$*, in quanto semplici operazioni di inserimento o eliminazione in testa, in coda o in mezzo tramite puntatori diretti tramite lista doppia.*

Viene di seguito fornito il codice di esempio.

```
HM_insert(cellamappa, nodo) {
    #catena_dinamica tmp;
    nodo[tmp]=nodo;
    if(first[cella] == NIL){
        first[cellamappa] = tmp;
    } else {
        prec[tmp] = last[cellamappa];
        next[last[cellamappa]] = tmp
    }

    last[cellamappa] = tmp;

    return tmp;

}
```

```
HM_unlink(cellamappa, pos_catena) {
    #in mezzo
    if(first[cellamappa] != pos_catena &&
        last[cellamappa] != pos_catena) {
        next[prec[pos_catena]] = next[pos_catena];
        prec[next[pos_catena]] = prec[pos_catena];
    } else {
        #prima per la catena
        if(first[cellamappa] == pos_catena) {
            first[cellamappa] = next[pos_catena];
            if(next[pos_catena] != NIL){
                prec[next[pos_catena]] = NULL;
            }
        }
        #l'ultima per la catena
        if(last[cellamappa] == pos_catena) {
            last[cellamappa] = prec[pos_catena];
            if(prec[pos_catena] != NIL) {
                next[prec[pos_catena]] = NIL;
            }
```

```
21          }
22      }
23
24      return pos_catena;
25  }
```

Predecessori e successori

Invece per il successore ed il predecessore dovremo limitare le sue potenzialità: per far sì che la complessità sia costante, siamo costretti a far si che il nodo passato alla funzione sia sempre il primo allocato nell'albero per il campo del nodo.

Teorema 5 (Predecessori e successori nei LBST) *sia data α, il suo albero T, e un nodo con $\sigma_n = campo[nodo]$, il quale è prima istanza per σ_n. Troviamo quindi il suo predecessore β ed il suo successore γ con il metodo descritto per i BST. Sia quindi:*

- *$\hat{\beta}$ il predecessore in LBST, trovato o come la prima istanza successiva per σ_n, o come prima istanza della catena dinamica per la prima lettera κ non nulla t.c. $\kappa \prec \sigma_n$;*
- *$\hat{\gamma}$ il successore in LBST, trovato come l'ultima istanza della catena dinamica per la prima lettera κ non nulla t.c. $\kappa \succ \sigma_n$.*

Allora $\beta \equiv \hat{\beta}$ e $\gamma \equiv \hat{\gamma}$ sse nodo è la prima istanza per σ_n.

Dim: analizziamo la struttura per l'albero. Sappiamo che per $T(nodo_\lambda)$ avrò tutti i nodi con campo precedente o uguale rispetto al nodo, e che per $T(nodo_\rho)$ tutte quelle maggiori, come definito per la costruzione del BST.
Analizziamo in dettaglio per il predecessore β e $\hat{\beta}$. Per il sotto-albero $T(nodo_\lambda)$ avremo che:

- se siamo con la funzione standard, allora il nodo sarà sicuramente una lettera compresa tra il nodo stesso e il limite inferiore, dato da:
  - il carattere σ_{r+1} se $\sigma_n \succ \sigma_r$;
  - il carattere σ_1 altrimenti;

  e quindi il primo nodo massimo del sotto-albero $T(nodo_\lambda)$;
- se siamo con la definizione per i LBST, la definizione sarà la medesima.

Avremo quindi che $\beta \equiv \hat{\beta}$.
Analizziamo ora in dettaglio per il successore γ e $\hat{\gamma}$. Per il sotto-albero $T(nodo_\rho)$ avremo che:

- se sono con la funzione standard, allora il nodo sarà sicuramente una lettera compresa tra il nodo stesso e il limite superiore, dato da:
  - il carattere σ_r se $\sigma_n \prec \sigma_r$;
  - il carattere $\sigma_{|\Sigma|}$ altrimenti;

  e quindi l'ultimo nodo minimo del sotto-albero $T(nodo_\rho)$;

- se sono con la definizione per i LBST, la definizione sarà la medesima.

Avremo quindi che $\gamma \equiv \hat{\gamma}$.

■

Proponiamo di seguito la funzione per trovare il precedente in un LBST:

```
1  predecessore(T,lettera) {
2      predecessore = NIL;
3      nodo = ricerca(T, lettera);
4
5      if (nodo == NIL )
6          return predecessore;
7
8      if (last[cella[φ(lettera)]] ≠ nodo) {
9          return nodo[next[mappa[nodo]]];
10     }
11
12     min = μ(campo[nodo],campo[root[T]]) <= 0 ? 0 : φ(σ_{r+1});
13
14     for(j = φ(lettera) - 1; j >= min; j = j - 1){
15         if(first[cella[φ(σ_j)]]!=NIL) {
16             predecessore = nodo[first[cella[φ(σ_j)]]];
17             break;
18         }
19     }
20
21     return predecessore;
22 }
```

Analizziamo quindi `predecessore`:

- l'istruzione a riga 3 viene fatta in $\Theta(1)$;
- le istruzioni a righe 8 e 9 vengono fatte in $\Theta(1)$;
- l'istruzione a riga 12 viene fatta in $\Theta(1)$;
- il ciclo for a riga 14 istruzione viene fatto in

  $$O(\max([0; \varphi(\sigma_r)], [\varphi(\sigma_r) + 1; |\Sigma|]))$$

  e quindi, siccome $|\Sigma|$ è costante per tutta la vita del LBST, sarà ridotto a $O(1)$.

Quindi in totale, $3 \times \Theta(1) + O(1) = \Theta(1)$.

Inserimento

Per l'inserimento la funzionalità è identica alla precedente.

Teorema 6 (Equivalenza di un LBST e costruzione sistematica) *sia data α, sia T il suo albero e sia δ la prima lettera da inserire nell'albero. Chiamiamo invece $\hat{T}$ l'albero LBST costruito come segue:*

- *se $\hat{T}$ è nullo, allora il carattere δ sarà la radice dell'albero e $\delta = \sigma_r$, e verranno generati tutti i dati della catena dinamica;*
- *se $\hat{T}$ non è nullo:*
  - *se δ non è il primo carattere istanziato $\hat{T}$, e se l'ultima istanza per il carattere δ in $\hat{T}$ è ancora foglia, si aggiungerà δ a sinistra di tale istanza, e si aggiornerà il puntatore della catena dinamica all'ultima istanza della mappa per la lettera δ;*
  - *se δ non è istanziato in $\hat{T}$, o se δ non è il primo carattere istanziato $\hat{T}$ e l'ultima istanza per il carattere δ in $\hat{T}$ non è foglia, allora si procederà come segue:*
    - *si cerca solo ed esclusivamente la prima lettera κ (ove κ sia maggiore di σ_1 se $\delta \prec \sigma_r$, σ_r altrimenti) t.c.*
      `right[nodo[first[cella[`$\varphi(\kappa)$`]]] == NIL,`
      e quindi `right[nodo[first[cella[`$\varphi(\kappa)$`]]] =` δ*;*
    - *altrimenti si cerca solo ed esclusivamente la prima lettera ι t.c.:*
      `left[nodo[last[cella[`$\varphi(\iota)$`]]]] == NIL,`
      e quindi `left[nodo[last[cella[`$\varphi(\iota)$`]]]] =` δ*;*

    e si aggiorneranno conseguentemente i record per la catena dinamica;

allora $T \equiv \hat{T}$.

Dim: prendiamo un BST T e un LBST $\hat{T}$ t.c. $T \equiv \hat{T}$, e una lettera δ. Procediamo quindi con l'inserimento ed analizziamo i rispettivi casi in entrambi gli alberi:

1. T e $\hat{T}$ nulli. Procediamo quindi a istanziare quanto dovuto come da teorema;
2. T e $\hat{T}$ non nulli, il padre per δ sarà un nodo con etichetta equivalente: procediamo quindi ad inserire come descritto;
3. T e $\hat{T}$ non nulli, δ non presente negli alberi: cerchiamo ricorsivamente in ogni sottoalbero di T il nodo padre; per $\hat{T}$ si cerca secondo la seguente costruzione sistematica:
   - fissiamo il minimo raggiungibile;
   - cerchiamo la prima lettera a sinistra superiore al limite inferiore non nulla la cui prima istanza non ha figli a destra: sarà il nodo padre;
   - cerchiamo la prima lettera a sinistra non nulla: questa sarà necessariamente il padre, in quanto la sua ultima istanza sicuramente non avrà figli a sinistra;

   si istanzierà quanto necessario;
4. : T e $\hat{T}$ non nulli, δ presente negli alberi: identico al caso precedente.

In tutti i casi, il padre per δ è il medesimo: allora la costruzione sistematica di $\hat{T}$ rende vero che $T \equiv \hat{T}$.

■

Proponiamo di seguito la funzione per inserire in un LBST:

```
inserisci(T, lettera) {
    campo[new] = lettera;

    #caso 1
    if(T == NIL){
        mappa[new] = HM_insert(cella[φ(lettera)], new);
        root[T] = new;
        campo[root[T]] = lettera;
        return;
    }

    #altri casi
    min = μ(lettera,campo[root[T]]) <= 0 ? 0 : φ(campo[root[T]]);
    i = φ(lettera);

    #caso 2
    if(first[cella[φ(σ_i)]] != NIL &&
        nodo[last[cella[φ(σ_i)]]] == NIL) {

        parent[new] = nodo[last[cella[φ(σ_i)]]];
        left[nodo[last[cella[φ(σ_i)]]]] = new;
        mappa[new] = HM_insert(cella[φ(σ_i)], new);
        return;
    }

    #caso 3/4 cerco verso sx
    for(j = i - 1; j >= min; j = j - 1) {
        if(first[cella[φ(σ_j)]] != NIL) {
            if(right[nodo[first[cella[φ(σ_j)]]]] != NIL)
                    break;

                    right[nodo[first[cella[φ(σ_j)]]]] = new;
                    mappa[new] = HM_insert(cella[φ(σ_i)], new);
                    parent[new] = nodo[first[cella[φ(σ_j)]]];
                    return;
        }
    }

    #caso 3/4 cerco verso dx
    for(j = i + 1; ; j = j + 1) {
        if(first[cella[φ(σ_j)]] != NIL) {
            left[nodo[first[cella[φ(σ_j)]]]] = new;
            mappa[new] = HM_insert(cella[φ(σ_i)], new);
            parent[new] = nodo[first[cella[φ(σ_j)]]];
        }
    }

}
```

Analizziamo quindi `inserisci`:

- il caso 1 termina in $\Theta(1)$;
- il caso 2 termina in $\Theta(1)$;
- i casi 3 e 4 verso sinistra termina in

  $$O(\max([0;\varphi(\sigma_r)],[\varphi(\sigma_r)+1;|\Sigma|]))$$

  e quindi, siccome Σ è costante per tutta la vita del LBST, è equivalente a $O(1)$;
- i casi 3 e 4 verso destra termina in $O(|\Sigma|)$, e quindi, siccome Σ è costante per tutta la vita del LBST, è equivalente a $O(1)$;

Quindi in totale, $2 \times \Theta(1) + 2 \times O(1) = \Theta(1)$.

Corollario 4 *dopo aver visto come viene si costruisce un LBST, possiamo tranquillamente affermare che la somma delle lunghezze delle catene dinamiche nella mappa è pari a η, poichè avrò un nodo per la catena per ogni nodo appartenente all'albero.*

Eliminazione

L'eliminazione in un LBST ha alcune limitazioni derivanti dalla dipendenza delle funzioni di predecessore e successore. Ad ogni modo l'eliminazione di un nodo,viene trattata identicamente come visto a lezione.

Teorema 7 (Equivalenza nell'eliminazione tra LBST e BST) *sia dato T albero BST, sia dato $\hat{T}$ albero LBST, $T \equiv \hat{T}$. Allora, se dato un nodo etichettato con δ che è prima istanza nell'albero, se elimino δ da T e da $\hat{T}$, allora $T \equiv \hat{T}$.*

Dim: Immediata dall'equivalenza delle operazioni effettuate.

■

Proponiamo di seguito la funzione per eliminare un nodo in un LBST:

```
1  elimina(T,lettera) {
2      if(T == NIL) {
3          return;
4      }
5
6      if((todlt = ricerca(T, lettera)) == NIL) {
7          return;
8      }
9
10     pos_catena = mappa[todlt];
11     yf = campo[todlt];
12     y=NIL, x = NIL;
13
14     if(left[todlt] == NIL  right[todlt]) {
15         y = todlt;
```

```
16      } else {
17              y = predecessore(T, lettera);
18      }
19
20      x = left[y] != NIL ? left[y] : right[y];
21
22      if(x != NIL) {
23              parent[x] = parent[y];
24      }
25
26      if(parent[y] == NIL) {
27          root[T] = x;
28      } else if(y == left[parent[y]]) {
29          left[parent[y]] = x;
30      } else {
31          right[parent[y]] = x;
32      }
33
34      if(y != x) {
35              campo[todlt] = campo[y];
36              mappa[todlt] = mappa[y];
37              nodo[mappa[todlt]]] = todlt;
38      }
39
40      HM_unlink(pos, pos_catena);
41
42      #ora y, yf e pos_catena sono slinkate.
43  }
```

Analizziamo quindi `elimina`:

- l'istruzione a riga 6 rispetta i tempi di `ricerca`, e quindi è in $\Theta(1)$;
- l'istruzione a riga 17 rispetta i tempi di `precedente`, e quindi è in $O(1)$;
- l'istruzione a riga 40 rispetta i tempi di `HM_unlink`, e quindi è in $\Theta(1)$;
- tutto il resto è in tempo $\Theta(1)$.

Quindi in totale, $3 \times \Theta(1) + O(1) = \Theta(1)$.

Ottenere le foglie

Avendo a disposizione una mappa che memorizza anche il puntatore all'ultimo nodo della catena dinamica per ogni campo, possiamo quindi rivedere la funzione per ottenere le foglie precedentemente descritta.

Corollario 5 (Tempi per ottenere le foglie) *il tempo per l'esecuzione della funzione "leafs" è strettamente dipendente dalla cardinalità delle foglie stesse, e quindi, come visto per il teorema 3, dipenderà strettamente da* Σ.

Proponiamo quindi un codice esemplificativo:

```
leafs(T, lista) {
    for(i=1; i<|Σ|; i=i+1) {
        nodo = nodo[last[cella[φ(σ_i)]]];
        if(right[nodo] == NIL && left[nodo] == NIL) {
                push(lista, nodo);
        }
    }
}
```

Analizziamo quindi i tempi:

- il ciclo dura $\Theta(|\Sigma|)$, e quindi, essendo Σ costante, sarà ridotto a $\Theta(1)$;

quindi, in totale, il tempo sarà $\Theta(1)$.

3.3 CLBST o LBST Compressi

I BST Lineari Compressi o CLBST, sono degli alberi BST i quali, comprimono un BST e utilizzano le proprietà precedenti dei LBST per rimanere in tempo costante.

3.3.1 Identificazione di un CLBST

In aggiunta ai normali campi dei BST e LBST, i CLBST necessitano di alcune accortezze nella costruzione del nodo

Definizione 19 (Nodo di un CLBST) *un nodo di un CLBST ha tutti i campi di un LBST e presenta un ulteriore campo che conta quanti nodi sta rappresentando in quel momento.*

Definizione 20 (Funzione di grandezza del campo) *per necessità della generalità del campo chiave del nodo, e poichè comprimiamo l'albero, non sempre dovremo salvare tutte le chiavi che ci vengono passate: ad esempio, in caso di eliminazione, avremo la necessità quindi di saper come duplicare la chiave. Definiamo quindi la funzione* Γ *come*

$$\Gamma : \mathbb{D} \to \mathbb{N}$$

tale che restituisca la grandezza del campo chiave del nodo analizzato.

Viste queste definizioni, necessariamente dobbiamo ridefinire cos'è una foglia per un CLBST.

Definizione 21 (Foglia in un CLBST) *un nodo foglia in un CLBST è o un nodo con il campo contatore pari a 1 e i figli* λ *e* ρ *nulli, o con il contatore maggiore di 1 e il figlio* ρ *nullo.*

Teorema 8 (Equivalenza di un CLBST a un BST) *sia data* α*, e siano dati i suoi alberi* T *BST,* $\hat{T}$ *LBST e* $\bar{T}$ *CLBST. Allora* $T \equiv \hat{T} \equiv \bar{T}$*.*

Dim: diamo per assunto che $T \equiv \hat{T}$. Vediamo ora l'equivalenza tra LBST e CLBST. Sia nota per entrambi la funzione `leafs`, tenendo conto della definizione di foglia per $\bar{T}$. Iniziamo a dimostrare l'equivalenza.
Scegliamo un qualunque nodo in $\hat{T}$ e analizziamo per casi:

- se è foglia: allora esiste un altro nodo in $\bar{T}$ tale che sia foglia e abbia medesima etichetta. Sfruttiamo quindi la funzione `leafs` per entrambi gli alberi: se confrontiamo le due liste restituiteci, notiamo che l'etichetta del nodo scelto è presente in entrambe. Allora

$$\forall f \in leafs(\hat{\tau}) \Rightarrow \exists f' \in leafs(\bar{\tau})$$

  e quindi il nodo scelto è sicuramente foglia in entrambi;

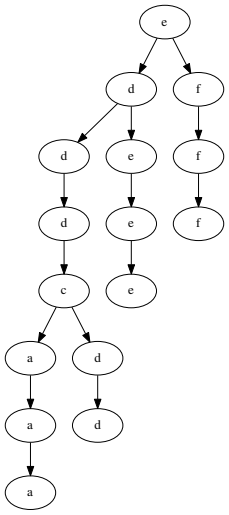
- se è nodo: allora esiste un nodo qualunque[2] in $\bar{T}$ il quale, espandendolo, abbia tutti i puntatori presenti in un BST identici a quello in $\hat{T}$, con medesima funzione. Allora il nodo scelto in $\hat{T}$ è presente anche in $\bar{T}$.

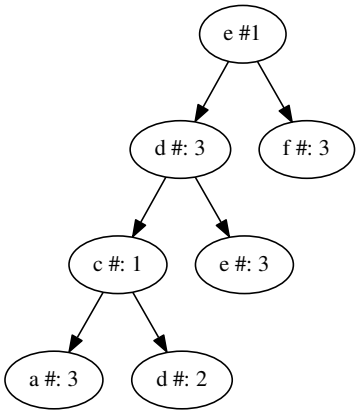
Allora $\hat{T} \equiv \bar{T}$ e $T \equiv \bar{T}$.

■

Esempio 2 *di seguito mostriamo come la medesima stringa $\alpha =$"edfddeecaefdafda" componga gli alberi BST, LBST e CLBST.*

BST per α	*CLBST per α*





Teorema 9 (Altezza di un CLBST) *sia $\bar{T}$ CLBST. Allora*

$$\log_2(\eta) \leq h(\bar{\tau}) \leq 2|\Delta| - 1$$

Dim: la dimostrazione dell'altezza massima dell'albero viene effettuata per induzione sulla cardinalità di Δ.
$|\Delta| = 1, 2$ *o caso base*: se $|\Delta| = 1$ allora è immediata. Se $|\Delta| = 2$, allora dobbiamo creare un α t.c. porti $\bar{T}$ ad avere l'altezza massima.
Analizziamo alcune composizioni di α:

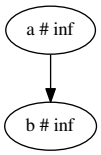
- $\alpha =$ "a^+b^+" $\vee$ "b^+a^+": l'altezza massima ottenibile in $h(\bar{\tau})$ è pari a 2;

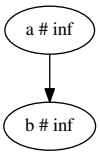
[2] che possa esser categorizzato anche come "foglia" per definizione!

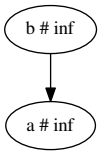
- $\alpha =$ "$a^+b^*a^*$": le lettere "a" e "b" fungono da "pozzo"[3]. Allora l'altezza massima ottenibile in $h(\bar{\tau})$ è pari a 2;
- $\alpha =$ "$b^+a^*b^*$": notiamo che il primo nodo assorbirà tutte le prime ι "b"; all'arrivo della prima "a" si crea un nuovo nodo, e fungerà da pozzo per tutte le seguenti "a". All'arrivo della prima "b" che segue le prime κ "a", dovrò creare un nodo a destra di "a", ed assorbirà tutte le "b" seguenti, diventando un nuovo pozzo per "b". Qualunque altra lettera in una qualunque combinazione seguente a questa situazione finirà nei rispettivi pozzi. Possiamo notare quindi che l'altezza massima ottenibile in $h(\bar{\tau})$ è pari a 3;
- $\alpha =$ "$(a^*b^*)^*$": per l'analisi di tutti i casi compresi in questa espressione regolare ed analizzati precedentemente, possiamo notare che l'altezza massima è sempre 3.

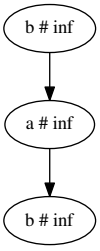
Abbiamo verificato che per $|\Delta| = 2$, abbiamo $\log_2(\eta) \leq h(\bar{\tau}) \leq 2|\Delta| - 1$, ed è data dall'alternanza di $\delta^*_{|\Delta|}\delta^*_1\delta^*_{|\Delta|}$.

a^*b^*, b^*a^*	$a^*b^*a^*$, $b^*a^*b^*$









$|\Delta| = i, i \in \mathbb{N}$ *o passo induttivo*: fissiamo α come una qualunque stringa per Δ^*. Per qualunque i, vediamo che

$$\max(h(\bar{\tau}(\alpha))) = 2|\Delta| - 1,$$
$$\alpha = \bigoplus_{\iota=1}^{|\Delta|-1} \delta^+_{|\Delta|}\Big(\bigoplus_{\kappa=1}^{\iota-1}\delta^*_\kappa\Big)^*\delta^+_\iota$$

dove $\oplus$ indica la funzione n-aria di concatenazione su un'intervallo.

■

[3] ovvero tutte le prossime lettere finiranno nel medesimo nodo

3.3.2 Funzioni fondamentali per un CLBST

Tutte le funzioni qui nominate andranno ad aggiornare tutte le funzioni precedentemente nominate nei LBST. Tutte quelle non nominate rimarranno quindi invariate.

Inserimenti nel mezzo della catena dinamica

Per soddisfare un'ulteriore necessità dei CLBST, dovremmo definire quindi una funzione per aggiungere un elemento nel mezzo della catena dinamica.

Corollario 6 (Tempi nella catena dinamica) *come per il corollario per i LBST, anche per i CLBST i tempi di inserimento ed eliminazione rimangono in tempo* $\Theta(1)$.

Viene di seguito fornito il codice di esempio.

```
HM_insert_middle(cellamappa, pos_catena, nodo) {
    nodo[tmp] = nodo;
    next[tmp] = pos_catena;
    prec[tmp] = prec[pos_catena];
    prec[pos_catena] = tmp;

    if(pos_catena == first[cellamappa]) {
        first[cellamappa] = tmp;
    }

    return tmp;

}
```

Inserimento

La funzione d'inserimento modifica minimamente la funzione nota per i LBST, par far si che sia utilizzato correttamente il contatore in ogni nodo.

```
inserisci(T, lettera) {
    campo[new] = lettera;
    counter[new] = 1;

    #caso 1
    if(T == NIL){
        mappa[new] = HM_insert(cella[φ(lettera)], new);
        root[T] = new;
        campo[root[T]] = lettera;
        return;
    }

    #altri casi
    min = μ(lettera,campo[root[T]]) <= 0 ? 0 : φ(campo[root[T]]);
    i = φ(lettera);

```

```
17      #caso 2
18      if(first[cella[φ(σ_i)]] != NIL &&
19          nodo[last[cella[φ(σ_i)]]] == NIL){
20          #new si perde!
21          padre = nodo[last[cella[φ(σ_i)]]]
22          counter[padre] = counter[padre] + 1;
23          return;
24      }
25
26      #caso 3/4 cerco verso sx
27      for(j = i - 1; j >= min; j = j - 1) {
28          if(first[cella[φ(σ_j)]] != NIL) {
29              if(right[nodo[first[cella[φ(σ_j)]]]] != NIL)
30                      break;
31
32                      right[nodo[first[cella[φ(σ_j)]]]] = new;
33                      mappa[new] = HM_insert(cella[φ(σ_i)], new);
34                      parent[new] = nodo[first[cella[φ(σ_j)]]];
35                      return;
36          }
37      }
38
39      #caso 3/4 cerco verso dx
40      for(j = i + 1; ; j = j + 1) {
41          if(first[cella[j]] != NIL) {
42              left[nodo[first[cella[φ(σ_j)]]]] = new;
43              mappa[new] = HM_insert(cella[φ(σ_i)], new);
44              parent[new] = nodo[first[cella[φ(σ_j)]]];
45          }
46      }
47
48  }
```

Vengono modificate quindi solamente le righe per il caso 1: come possiamo notare, le modifiche non fanno cambiare la complessità della funzione, la quale rimane sempre $\Theta(1)$.

Eliminazione

L'eliminazione nei CLBST è pressochè identica all'eliminazione in un LBST, ma con la differenza che, poichè comprimiamo i nodi, c'è la necessità, in certe situazioni, di duplicare il nodo.

Teorema 10 (Equivalenza nell'eliminazione tra CLBST e LBST) *sia $\hat{T}$ LBST e $\bar{T}$ e δ chiave da rimuovere. Se $\hat{T} \equiv \bar{T}$ prima dell'eliminazione, allora $\hat{T} \equiv \bar{T}$ anche dopo l'eliminazione della chiave.*

Dim: dobbiamo quindi evidenziare i punti in cui l'eliminazione in $\hat{T}$ e in $\bar{T}$ si differenziano. Va trovato quindi δ per $\hat{T}$ ed eliminato. Procediamo quindi a trovare il δ in $\bar{T}$: notiamo che la funzione `ricerca` restituisce il medesimo nodo. Se il nodo trovato non è nullo, procediamo quindi ad analizzare i casi di eliminazione:

1. nodo con $counter[nodo] > 1$: che sia foglia o nodo, il successore naturale del nodo con chiave δ è sempre equivalente al primo nodo a sinistra. Questo significa che è equivalente ad eliminare un nodo con figlio il quale ha un solo figlio[4]: allora decremento il contatore di uno e l'albero;

2. nodo con $counter[nodo] = 1$ e nodo "sacrificale"[5] $counter[sacrificale] > 1$: il nodo sacrificale comprime più nodi, e quindi la prima istanza compressa del nodo sacrificale verrà spostata come chiave del nodo indicato per δ. Questo significa che il primo successore del nodo è il sacrificale. Ora però dividiamo per casi:

   - $sacrificale_\rho = NIL \wedge sacrificale = nodo_\lambda$: allora vuol dire che si possono eliminare subito *nodo* e agganciare al suo posto il sacrificale;
   - in qualunque altro caso, si deve scomporre il sacrificale: quindi si copia la chiave del sacrificale nel nodo, si decrementa il contatore del sacrificale, e si sistema la posizione nella catena dinamica del nuovo nodo sacrificale;

3. si deve effettivamente eliminare un nodo dall'albero: questo significa che si deve spostare i campi del sacrificale nel *nodo* e successivamente eliminarlo, sistemando la catena dinamica;

allora per il teorema di equivalenza tra $\hat{T}$ e $\bar{T}$ sarà ancora equivalente.

■

Proponiamo di seguito la funzione per eliminare in un LBST:

```
1  elimina(T,lettera) {
2      if(T == NIL) {
3          return;
4      }
5
6      if((todlt = ricerca(T, lettera)) == NIL) {
7          return;
8      }
9
10     #caso 1
11     if(counter[todlt] > 1){
12             counter[todlt] = counter[todlt] - 1;
13             return;
14     }
15
16     pos_catena = mappa[todlt];
17     yf = campo[todlt];
18     y=NIL, x = NIL;
19
```

[4] qualunque chiave maggiore del campo chiave del nodo che comprime sarà sempre agganciato alla prima istanza del medesimo nodo: questo lo posso dimostrare grazie alla costruzione sistematica di un LBST.

[5] ovvero il nodo che in realtà verrà spezzato in un CLBST, mentre in un LBST sarà il nodo che avrà la sua chiave spostata, ma il nodo ospitante eliminato

```
20      #sacrificale
21      if(left[todlt] == NIL  right[todlt]) {
22          y = todlt;
23      } else {
24              y = predecessore(T, lettera);
25      }
26
27      #caso 2 counter[todlt]=1 && counter[y]>1
28      if(y != NIL && counter[y] > 1) {
29          #caso 2.1: sopprimo un nodo
30          if(y == left[todlt]){
31              right[y] = right[todlt];
32              parent[y] = parent[todlt];
33
34              if(parent[todlt] != NIL) {
35                  if(left[parent[todlt]] == todlt) {
36                      left[parent[todlt]] = y;
37                  } else {
38                      right[parent[todlt]] = y;
39                  }
40              } else {
41                  root[T] = y;
42              }
43
44              HM_unlink(pos, pos_catena);
45
46              return;
47          } else {
48          #caso 2.2: spezzo un nodo
49              size = Γ(campo[y]);
50              campo[todlt] = memdump(campo[y], size);
51              counter[y] = counter[y] - 1;
52
53              HM_unlink(pos, pos_catena);
54              catena = HM_insert_middle(cella[φ(campo[y])],
55                                        mappa[y], todlt);
56
57              mappa[todlt] = catena;
58
59              return;
60          }
61      }
62
63      #caso 3: sposto un intero nodo
64      x = left[y] != NIL ? left[y] : right[y];
65
66      if(x != NIL) {
67              parent[x] = parent[y];
68      }
69
```

```
70      if(parent[y] == NIL) {
71          root[T] = x;
72      } else if(y == left[parent[y]]) {
73          left[parent[y]] = x;
74      } else {
75          right[parent[y]] = x;
76      }
77
78      if(y != x) {
79              campo[todlt] = campo[y];
80              mappa[todlt] = mappa[y];
81              nodo[mappa[todlt]]] = todlt;
82      }
83
84      HM_unlink(pos, pos_catena);
85
86      #ora y, yf e pos_catena sono slinkate.
87  }
```

Analizziamo quindi i tempi per `elimina`:

- per la `ricerca` la complessità rimane $\Theta(1)$;
- per il caso 1 si tratta di una sola operazione, e quindi il tempo sarà $\Theta(1)$;
- per la scelta del sacrificale si guarda la complessità di `predecessore`, la quale è $O(1)$;
- per il caso 2.1 la complessità è $\Theta(1)$;
- per il caso 2.2 la complessità è $\Theta(1)$;
- per il caso 3 la complessità è $\Theta(1)$;
- per le operazioni nella catena dinamica la complessità è $\Theta(1)$;

allora la complessità di `elimina` è $6 \times \Theta(1) + O(1) = \Theta(1)$.

4 Analisi dell'albero e costruzione del grafo

In questa sezione inizieremo ad analizzare le scelte effettuate per l'analisi dell'albero della stringa e come vengono definiti i collegamenti tra i vari nodi. Le scelte effettuate tengono conto di molteplici fattori, dalla facilità implementativa, all'efficienza per l'uso reale nel progetto. Questo fa sì che, nonostante alcune scelte possano apparire incaute, nello specifico problema non aumentano in maniera considerevole la complessità totale dell'algoritmo.

4.1 Ottenere i percorsi

Analizziamo quindi come ottenere tutti i percorsi $foglia \rightsquigarrow \bar{\tau}$.

Definizione 22 (Lista del parente) *definiamo come nodo della lista del parente il nodo costituito da:*

- *un puntatore al precedente ed al successore;*
- *un puntatore all'etichetta del nodo parente;*
- *un puntatore al campo foglia;*
- *un contatore per $\mathscr{W}_{u \rightsquigarrow v}$, dove u è il nodo attuale e v è il nodo precedente nella lista;*

e che rappresenta $\bar{\tau} \rightsquigarrow v$, con $v \in leafs(\bar{\tau})$.

Definizione 23 (Lista dei parenti) *definiamo come nodo della lista dei parenti il nodo costituito da:*

- *un puntatore alla lista del parente;*
- *un puntatore al successore ed al predecessore della lista dei parenti;*

e che rappresenta l'insieme foglia $\rightsquigarrow \bar{\tau}$ per tutto l'albero $\bar{T}$.

Teorema 11 (Numero di nodi percorsi) *sia dato $\bar{T}$ albero CLBST. Allora il numero di nodi analizzati $\forall v \in leafs(\bar{\tau})(v \rightsquigarrow \bar{\tau})$ sarà sempre minore o uguale dell'altezza massima di $\bar{T}$, ovvero $2|\Delta| - 1$.*

Dim: immediata per il teorema dell'altezza massima dell'albero.

■

Presentiamo quindi il codice di esempio per ottenere tutti i parenti nell'albero

```
1 getParent_path(foglia) {
2     if(foglia == NIL){
3         return NIL;
4     }
5
6     leaf = campo[foglia];
7
8     campo[coda] = leaf;
9     len[coda] = counter[foglia];
```

```
10
11      foglia = parent[foglia];
12
13      while(foglia != NIL) {
14          campo[aux] = campo[foglia];
15          len[coda] = μ(campo[coda], campo[foglia]) < 0 ?
16                      counter[foglia] : 1;
17          next[aux] = coda;
18          coda = aux;
19          foglia = parent[foglia];
20      }
21      leaf[coda] = leaf;
22
23      return coda;
24  }
25
26  allParents_path(leafs) {
27      if(leafs == NIL){
28          return NIL;
29      }
30
31      aux = head;
32      single[aux] = getParent_path(leaf[leafs]);
33      leafs = next[leafs];
34
35      while(leafs != NIL) {
36          single[a1] = getParent_path(leaf[leafs]);
37          leafs = next[leafs];
38          next[a1] = aux;
39          aux = a1;
40      }
41
42      return head;
43  }
```

Analizziamo quindi i tempi per `getParent_path`:

- il ciclo while impiega $O(1)$, come visto per il teorema dei nodi percorsi;
- tutte le altre operazioni hanno tempo $\Theta(1)$;

allora `getParent_path` impiegherà $\Theta(1)$.
Analizziamo invece i tempi per `allParent_path`:

- il ciclo while impiega $O(1)$, poichè dipende dal numero di foglie in $\bar{T}$, che al più possono essere $|\Delta|$;
- tutte le altre operazioni hanno tempo $\Theta(1)$;

allora `allParent_path` impiegherà $\Theta(1)$.

Corollario 7 (Vertici e foglie) *all'interno della lista di tutti i percorsi* $\bar{\tau} \rightsquigarrow foglia$ *saranno presenti solo i vertici del grafo per la risoluzione del problema.*

4.2 Lowest Common Ancestor ed archi nel grafo

Ora analizziamo come verrà trattato l'insieme dei percorsi $\bar{\tau} \rightsquigarrow foglia$ per trovare l'antenato minimo comune, in maniera di poter decretare se due vertici sono collegati tra loro.

Teorema 12 (Nodi analizzati per LCA) *sia $\bar{T}$ CLBST e*

$$leafsPaths = allParents\_path(leafs(\bar{\tau})).$$

Allora per $\forall \iota, \kappa \in leafsPaths$, $LCA(\iota, kappa)$ non analizzerà mai più di $4|\Delta|-2$ nodi.

Dim: immediata per la costruzione di allParents_path.

■

Presentiamo quindi il codice di esempio per la verifica della raggiungibilità di due vertici per il problema considerato

```
1  is_path_connect(from,to,max) {
2      #LCA
3      while(from != NIL && to != NIL &&
4              μ(campo[from], campo[to]) == 0 &&
5              len[from] == len[to]) {
6          to = next[to];
7          from = next[from];
8      }
9      #distanza
10     dist = 0;
11
12     while(from != NIL  to != NIL) {
13         if(dist > max)    return 0;
14         if(from != NIL) {
15             dist = dist + len[from];
16             from = next[from];
17         }
18         if(dist > max)    return 0;
19         if(to != NIL) {
20             dist = dist + len[to];
21             to = next[to];
22         }
23     }
24     return dist;
25 }
```

Vediamo quindi il tempo per l'esecuzione di is_path_connect:

- LCA si riduce a $O(1)$ come visto per il teorema di LCA nei CLBST;
- la distanza si riduce ai nodi non comuni per LCA, e quindi rimane sempre in tempo $O(1)$;

allora il tempo per is_path_connect è di $O(1)$.

5 Grafo del problema

In questa parte analizzeremo la struttura per la descrizione del grafo non orientato e com'è costruito.

5.1 Identificazione del grafo

Il grafo del problema viene descritto come lista di adiacenze, e quindi verrà occupata la memoria minima possibile per la descrizione dello stesso.

Definizione 24 (Lista degli archi) *la lista di adiacenza per un vertice sarà definita da un nodo che presenta:*

- *un puntatore all'etichetta del vertice raggiunto;*
- *un puntatore al vertice adiacente successivo.*

Definizione 25 (Lista dei vertici) *la lista per la rappresentazione dei vertici del grafo sarà definita da un nodo che presenta:*

- *un puntatore all'etichetta del vertice locale;*
- *un puntatore alla lista degli archi per il vertice;*
- *un puntatore al vertice successivo.*

Definizione 26 (Grafo) *per descrivere un grafo quindi utilizziamo un record che contiene:*

- *un puntatore al primo vertice;*
- *un contatore dei vertici in G;*
- *un contatore degli archi in G.*

5.2 Costruzione del grafo

Vediamo quindi come si costruisce il Grafo G per un CLBST $\bar{T}$.

Corollario 8 (Tempo di costruzione del grafo) *il tempo di costruzione del grafo è strettamente legato al numero di vertici in $\bar{T}$.*

Lemma 4 (Combinazioni di vertici) *per la costruzione di G grafo non orientato, per poter verificare la raggiungibilità di ogni vertice, si dovrà effettuare il confronto tra un nodo e tutti i suoi successivi. Così facendo avremo al più*

$$\sum_{i=1}^{|\Delta|} \big(|\Delta| - i\big) = \binom{|\Delta|}{2} = \frac{|\Delta| \times (|\Delta| - 1)}{2}$$

archi, come definito per il teorema della "totalità degli archi in G".

Proponiamo quindi il codice di esempio atto alla risoluzione del problema, costruendo il grafo G.

```
makeGraph(T̄, min, max) {
    leafs(T̄,leafs);
    G = NIL;
    paths = allParents_path(leafs);

    #loc record del grafo: e counter, v  counter, g ptr a vertice
    #ga appoggio di G (ptr next, v(ertex),e(dge)s)

        #vertici
        for(i = paths; i != NIL; i= next[i]){
            v[loc] = v[loc] + 1;
            v[ga] = leaf[single[i]];
            next[ga] = G;
            G = ga;
            #archi
            for(j = next[i]; j != NIL; j = next[j]) {
                dist = is_path_connect(i, j);
                if(dist >= min && dist <= max) {
                    #ea appoggio vertici adiacenti
                    e[loc] = e[loc] + 1;
                    v[ea] = leaf[single[j]];
                    next[ea] = es[ga];
                    es[G] = ea;
                }
            }
        }

        g[loc] = G;

        return G

}
```

Analizziamo quindi i tempi per `makeGraph`:

- la funzione `leafs`, come già analizzato, impiega $\Theta(1)$;
- la funzione `allParents_path`, come già analizzato, impiega $\Theta(1)$;
- il doppio ciclo for impiega

$$O\Big(\sum_{i=1}^{|\Delta|}\big(|\Delta|-i\big)\Big) = O\Big(\frac{|\Delta|\times(|\Delta|-1)}{2}\Big) = O\big(|\Delta|^2-|\Delta|\big)$$

  e, siccome $\Delta \subseteq \Sigma$, e Σ è sempre costante, allora sostituendo Δ con Σ, possiamo affermare che, essendo Σ sempre costante, si riduce a $O(1)$;

e quindi in totale impiegheremo $\Theta(1)$.

5.3 Stampa del grafo

Rimane quindi solo la stampa del grafo in formato DOT. Per la stampa dobbiamo necessariamente scorrere tutti i vertici, e per ogni vertice dobbiamo

quindi scorrere la lista delle adiacenze.
Presentiamo di seguito del codice di esempio:

```
1  printDot(G) {
2      print("graph G {");
3      for(G; G != NIL; G = next[G]) {
4          print("%c [label=\"%d\"];", v[G], φ(v[G]));
5          for(es[G]; es[G] != NIL; es[G] = next[es[G]]){
6              print("%c -- %c;", v[G], v[es[G]]);
7          }
8      }
9      print("}");
10 }
```

Analizziamo quindi i tempi:

- il ciclo esterno durerà esattamente $\Theta(|V|)$;
- il ciclo interno durerà esattamente nella peggiore delle ipotesi $O(|V|-1)$;

avremo quindi $O(\frac{|V| \times (|V|-1)}{2})$[1]. Ora sappiamo che $|V|$ al più è pari a Σ, e quindi la stampa verrà effettuata in tempo $O(|V|^2) = \Theta(1)$.

[1] si divide per due poichè se si stampasse tutti gli n-1 nodi per ogni nodo, si stamperebbero 2 volte i medesimi nodi, quindi se ne deve considerare solo la metà, anche per come è costruito G;

6 Ottimizzazioni del problema e tempi teorici

In questa sezione prenderemo in esame le ottimizzazioni effettuate nello sviluppo della soluzione al problema; successivamente analizzeremo i veri tempi teorici applicati.

6.1 Ottimizzazioni per gli alberi ed utilizzo memoria

Nell'applicazione degli alberi BST Lineari Compressi, vengono solamente implementate le funzioni strettamente necessarie alla risoluzione del problema. In primis, è stata semplificata la catena dinamica con un array bidimensionale di puntatori di dimensione costante, pari alla cardinalità dell'alfabeto del problema. In secondo luogo, quindi, sono state semplificate tutte le funzioni, legate ad esempio all'inserimento, limitando il numero di istruzioni (ad esempio utilizzando le funzioni di calloc o memcpy). L'utilizzo della memoria per gli alberi quindi si attesta come segue:

- per ogni nodo dell'albero Compresso, vengono utilizzati 4 puntatori, un campo intero senza segno e la memoria per la chiave;
- per la struttura contenitrice per rendere lineare l'inserimento nell'albero, vengono usati 2 puntatori per la radice, 2 puntatori a funzione, un puntatore alla mappa e un campo intero per la cardinalità dell'alfabeto;
- per la mappa vengono creati quindi 2*Σ celle di puntatori;

per un totale di $\eta(4*PTR+INT+CHAR)+(2*|\Sigma|+5)PTR+INT$. Secondo codifica dei tipi ANSI-C si ha che per PTR a 8Byte, INT 4Byte e CHAR 1Byte, quindi $(37)\eta + 16(26) + 44 = (37\eta + 460)$ Byte.

6.2 Ottimizzazioni per il grafo ed utilizzo memoria

Per il grafo non si sono adottate particolari ottimizzazioni, se non per ottenere il più velocemente possibile la lista delle foglie nell'albero.

L'utilizzo della memoria per la lista delle foglie è quindi, per il caso peggiore (ovvero con tutti i percorsi radice foglia massimi), di $26 * 16 = 416$ Byte.

Per la lista del percorso radice foglia invece si ha, sempre nel caso peggiore, che ogni nodo ha $3PTR + INT$, e quindi considerando 26 nodi, un consumo di $26(2 * 26 - 1)(24 + 4) = 37128$ Byte; invece per la lista di tutti i percorsi si ha $2PTR$, e sempre nella peggiore delle ipotesi, si ha $26 * 8$ Byte.

Per il grafo, quindi si avrà che per ogni vertice avremo un puntatore all'etichetta, uno alla liste di adiacenze ed uno al vertice successivo; per ogni arco invece avremo un puntatore all'etichetta del vertice adiacente ed uno al nodo successivo. Possiamo quindi riassumere che per un grafo utilizzeremo, sempre nel caso peggiore, $26 * V + 13 * 25 * E = 26 * 24 + 13 * 25 * 16 = 5824$ Byte.

6.3 Ottimizzazione del progetto

Per quanto riguarda le ottimizzazioni di codice, oltre ad aver utilizzato funzioni che riducono il numero di operazioni, viene settato un controllo per un flag in maniera che, se non vogliamo "perdere tempo" ad eliminare la memoria

che non è più richiesta, si può disabilitare la chiamata di tale funzioni atte a deallocare la memoria dinamica utilizzata.

6.4 Tempi teorici

Passiamo quindi all'analisi reale dei tempi teorici. Di seguito proponiamo la tabella con tutte le chiamate di funzioni e il tempo richiesto.

Funzione	Tempo
CLBST_init	$\Theta(1)$
inizializzazione del vettore delle cardinalità	$\Theta(\|\Sigma\|)$
c = getchar()	$\Theta(1)$
while(getchar())[...]CLBST_insert[...]	$\Theta(\|\alpha\| * O(1)) = \Theta(\|\alpha\|)$
trovare cardinalità minima e massima	$\Theta(\|\Sigma\|)$
CLBST_getLeaf	$\Theta(\|\Sigma\|)$
allparents_path	$\Theta(\|\Sigma\| * (2\|\Sigma\| - 1)) = \Theta(\|\Sigma\|^2)$
CLBST_leafLst_destroy	$\Theta(\|\Sigma\|)$
is_path_connect	$O(2\|\Sigma\| - 1)$
creazione del grafo	$O(\|\Sigma\|) + O(\|\Sigma\|^2) = O(\|\Sigma\|^2)$
free()	$\Theta(1)$
destroy_multiPartentPath	$O(\|\Sigma\|^2)$
printDot	$O(\|V\| + \|E\|) = O(\|\Sigma\|^2)$
destroy_VEdet	$O(\|V\| + \|E\|) = O(\|\Sigma\|^2)$
CLBST_destroy	$\Theta(\eta) = O(\|\alpha\|)$
Totale	$\Theta(\|\alpha\|)$

Il problema proposto viene quindi risolto completamente in $\Theta(|\alpha|)$.

7 Analisi dei tempi

L'analisi dei tempi viene effettuata secondo gli algoritmi presentati a lezione, con alcune minime modifiche per meglio generalizzare il problema di misura dei tempi.

7.1 Procedure per la misura dei tempi

Si esegue innanzitutto la funzione per determinare la granularità dell'orologio del computer, in maniera da vedere qual'è la sua "sensibilità". Questo ci permette di passare alle fasi successive, tra cui capire qual'è il numero di ripetizioni da dover eseguire per testare il programma in una particolare situazione, e successivamente le stime legate a tale situazione.

La funzione che calcola il numero minimo di esecuzioni si basa sulla granularità del tempo, in modo da vedere l'errore minimo definito nel ciclo: chiediamo esplicitamente, infatti, che la misurazione minima della funzione sia almeno più grande della granularità fratto l'errore di misurazione minimo voluto, in quanto dobbiamo avere la certezza che la misura che effettuiamo non risulti errata[1]. Questa verrà utilizzata di seguito per stimare il tempo d' esecuzione e la sua varianza, seguendo quindi i metodi di inferenza statistica.

Le procedure sono state modificate in maniera da potersi meglio adattare alla misurazione di qualunque evento, senza specializzare un gruppo di funzioni ad un solo compito: vengono richiesti come parametri delle funzioni, in maniera da poter generalizzare la struttura. Inoltre, esludendo dal calcolo dei tempi tutto il codice non strettamente necessario alla misurazione dell'evento, sarà possibile non effettuare le misurazioni di tara per eliminare i tempi di scostamento costante, in quanto la funzione di misurazione del tempo misura solo ed esclusivamente solo lo stretto necessario.

Le funzioni richieste per il funzionamento sono quindi:

1. `prepara`: funzione che ha il compito di inizializzare (in questo caso) lo standard input;
2. `execute`: funzione che si occupa di eseguire la procedura da misurare per l'ambiente preparato da `prepara`;
3. `riavvolgi`: funzione che ha il compito di portare alla situazione precedente a `execute`, in maniera da poter rieseguire l'evento.

7.2 Presentazione dei tempi reali

Nel grafico che segue possiamo quindi notare come l'analisi teorica dei tempi rispetti l'analisi dei tempi reali misurati attraverso l'inferenza statistica. Nell'analisi dei tempi vengono create quindi delle stringhe lunghe da 0 a 30000 lettere, con incrementi di lunghezza pari a 1000 lettere. La stima delle funzioni che descrivono i tempi avviene tramite l'ausilio di `Gnuplot` e la funzione `fit`,

[1] errata per la situazione di carico reale del computer che la esegue in quel momento, in quanto essendo un pc a time-sharing la misurazione deve tener conto che il pc non è utilizzato solo per quel programma

la quale permette di interpolare i punti ottenuti con il metodo dei minimi quadrati con 30 gradi di libertà. In questo caso, abbiamo descritto la funzione f come $f(x) = a * x + b$, e l'errore di stima ottenuto è il seguente:

- per "a" abbiamo un errore del 0.7206%;
- per "b" abbiamo un errore del 24.7%[2].

Per la costruzione dell'albero abbiamo descritto la funzione g come $g(x) = a * x + b$, abbiamo valutato i tempi di esecuzione con l'ausilio di `Gnuplot` con i seguenti l'errori di stima per 30 gradi di libertà:

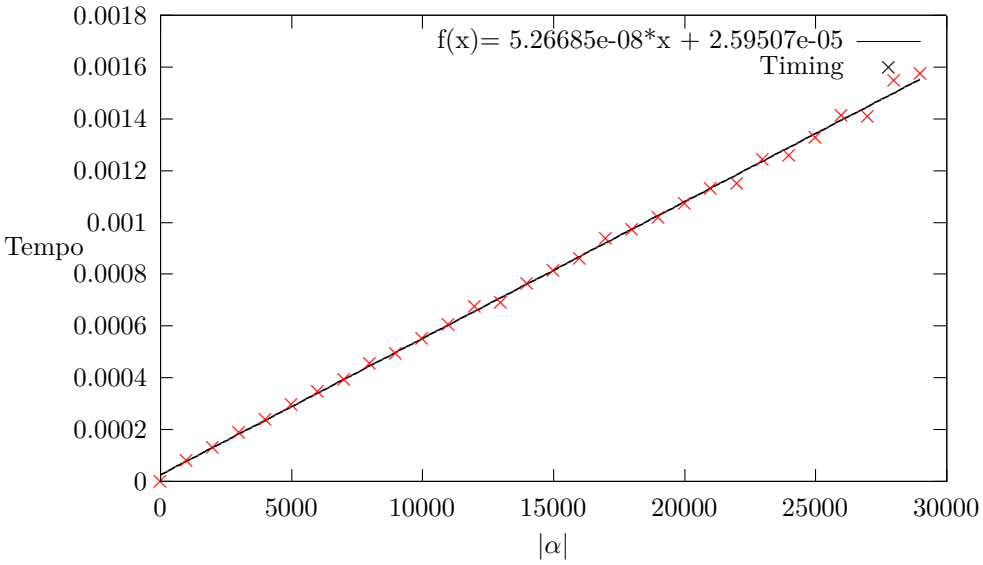
- per "a" abbiamo un errore del 0.5109%;
- per "b" abbiamo un errore del 263.9%.

Possiamo quindi concludere l'analisi dei tempi affermando che i tempi teorizzati rispettano i termpi reali.

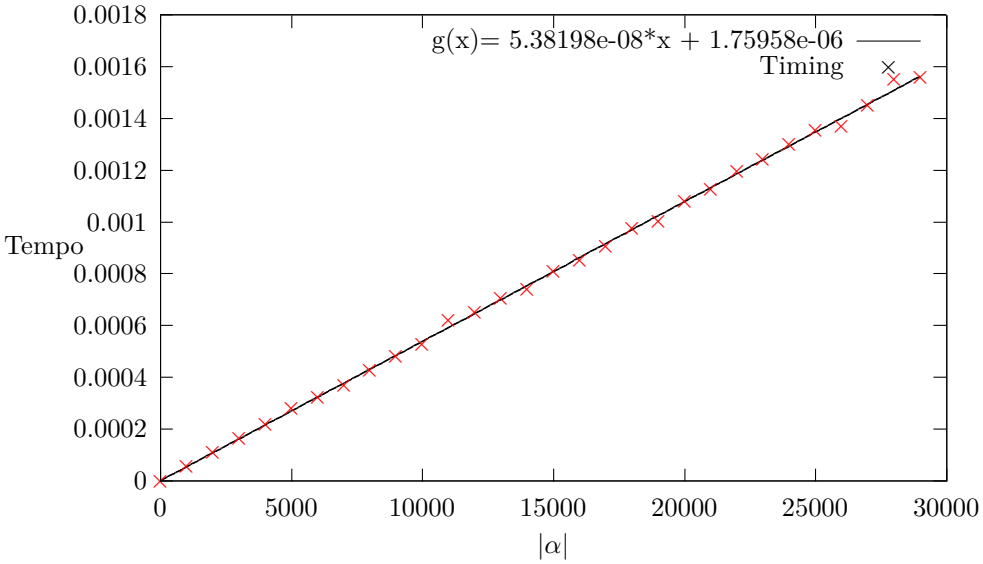
Tutte le misurazioni sono state effettuate con un computer con 8GB di RAM, e processore *Intel i5-4210U*. Il sistema operativo in uso era *Ubuntu 16.04.5 GNOME*, con kernel *Linux 4.15.0-38-generic*. I test sono stati effettuati in recovery mode, in quanto si eliminano tutti i servizi non necessari, i quali possono interferire con le misurazioni.

[2]questo perchè in $|\alpha| = 0$ si ha che termina subito

Tempi di risoluzione del problema



Tempi di costruzione del CLBST



8 Conclusioni

Sono state scoperte alcune interessanti proprietà dei BST, specie a riguardo del dominio applicativo che essi trattano. Simili tecniche di riduzione della complessità si trovano in alcuni importanti alberi come i *Van Emde Boas tree*, *y-fast tries*[1], ed i *Compressed C-Trie*, i quali comprimono gli *Hash array mapped trie*[2].

Inoltre, la compressione dell'albero fornisce ulteriori semplificazioni, tra cui poter dimostrare che l'inserimento in un qualunque BST compresso può dare alle funzioni ben note una complessità costante (rendendo quindi non più necessaria una funzione φ!).

Di fondamentale importanza è stato il ragionamento sulla composizione dell'albero e la generazione del grafo, in quanto non è facile intuire che:

1. non è possibile scegliere a priori se un grafo è totalmente connesso, parzialmente connesso o sconnesso, in quanto l'albero si basa anche sull'ordine di apparizione delle lettere, e non solo sulla loro frequenza;

2. non è possibile anticipare alcune computazioni del grafo, in quanto dovremo per qualunque foglia testare tutte le foglie successive, le quali possono mutare di continuo;

3. se si considerano alcuni punti fondamentali, quali l'altezza massima dell'albero, sappiamo per certo che sarà possibile rendere soluzioni apparentemente troppo complesse accettabili;

4. i compilatori, se impostati con l'ottimizzare il codice, possono dare risultati sorprendenti.

Sarebbe interessante poter pensare ad ulteriori utilizzi dei CLBST nella vita reale, nonostante penso ci siano pochi utilizzi concreti, in quanto strettamente dipendente dall'ampiezza del dominio applicativo: infatti sono rari i casi in cui ci sia un piccolo dominio applicativo con un enorme numero di inserimenti.

[1] i quali analizzano in maniera fondamentale la complessità e l'eterogeneità del dominio
[2] alberi con inserimenti e ricerche basate su tabelle e/o funzioni di hash

Riferimenti bibliografici

[1] NetworkX, *Software for complex networks*, https://networkx.github.io/

[2] PyDOT, libreria Python per DOT, https://pypi.org/project/pydot/

[3] Valgrind, *an instrumentation framework for building dynamic analysis tools*, http://www.valgrind.org/

[4] Trevis Rothwell, James Youngman, *The GNU C Reference Manual*, https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.pdf

[5] Alberto Policriti(2017-2018), Appunti per la stima dei tempi del progetto, http://users.dimi.uniud.it/∽alberto.policriti/home/sites/default/files/ASD/appunti_17-18.pdf

[6] pipe(7) - Linux man page(2018), *limiti di STDIN*, https://linux.die.net/man/7/pipe

[7] Wikipedia(2018), *Monomorfismo*, https://it.wikipedia.org/wiki/Omomorfismo

[8] A. Dovier, R. Giacobazzi, *Fondamenti dell'Informatica: Linguaggi Formali, Calcolabilità e Complessità*, https://users.dimi.uniud.it/∽agostino.dovier/DID/Dispensa.pdf

[9] Cormen, Thomas H., Leierson, Charles E., Rivest, Ronald L., Stein, Clifford (2010), *Introduzione agli algoritmi e alle strutture dati*, Terza Edizione, McGraw-Hill, Milano

[10] Peter van Emde Boas, P*reserving Order in a Forest in less than Logarithmic Time*, 16th Annual Symposium on Foundations of Computer Science, 75–84, 1975.

[11] Dan Willard, *Log-Logarithmic Worst-Case Range Queries are Possible in Space* $\Theta(n)$, Inf. Process. Lett., 17(2):81–84, 1983.

[12] Phil Bagwell, *Ideal Hash Trees*, https://infoscience.epfl.ch/record/64398/files/idealhashtrees.pdf