# Introduzione

Il progetto si articola in 5 file sorgenti, di cui uno, swordx.c, è il programma principale mentre gli altri sono moduli contenenti le strutture dati necessarie.

I file stack.c e threadIdStack.c sono due simili implementazioni di uno stack, l’uno atto a contenere pathnames e l’altro per contenere puntatori a threads. I due file presentano funzioni molto simili con la sola differenza che si trova nel tipo di dato contenuto in ogni nodo. Tale duplicazione di codice si sarebbe potuta evitare utilizzando dei puntatori a void.

Il file trie.c contiene l’implementazione di un albero digitale, o trie, che è la struttura dati che si occupa di mantenere in memoria le parole lette dai file.

Il file BST.c implementa un albero binario di ricerca collegato strettamente ai nodi di un trie, da cui dipende. Questo collegamento stretto si concretizza nel fatto che il contenuto di un nodo di tale albero altro non è un puntatore a un corrispettivo nodo di un trie.

Tutti i moduli sopra presentati hanno anche un corrispettivo header file che funge da interfaccia verso il programma utilizzatore e contenente le definizioni delle strutture dati e le dichiarazioni di funzioni accessibili dall’esterno.

Il programma utilizzatore è ovviamente swordx.c, il quale invece non è dotato di header file e le cui dichiarazioni di funzioni sono localizzate nel file stesso, prima delle loro definizioni. Il programma svolge un conteggio delle parole contenute nei file passati in input, offrendo una serie di funzioni aggiuntive osservabili mediante il comando –help. L’esecuzione avviene per gran parte single-threaded, con il multi-threading limitato all’analisi contemporanea dei file di input, a ognuno dei quali viene assegnato un thread. Una migliore implementazione di ciò sarebbe potuta essere la fusione dei file di input in uno unico file (a livello logico o fisico) e quindi l’uso dei thread per analizzare diversi settori di questo file. Una tale implementazione risolve l’inconveniente di avere file di diverse dimensioni in input : mentre i file più lunghi vengono letti, i thread dei file più brevi rimangono in attesa senza nulla da fare, riducendo il grado di parallelismo.

Il progetto è corredato di un makefile per la compilazione e la risoluzione delle dipendenze.

# Librerie

Tutte le librerie utilizzate fanno parte della libreria standard di C implementata da GNU, la glibc. Nei sistemi GNU/Linux non è necessaria alcuna installazione aggiuntiva, mentre nel porting del programma su altri sistemi, come Windows, potrebbe essere necessario importare nel programma il codice sorgente di Gnulib, la libreria GNU per la portabilità del software su altri sistemi.

Le librerie ISO utilizzate sono ctypes.h, stdio.h, stdlib.h e string.h. La libreria ctypes.h in particolare contiene delle utili funzioni per stabilire di che tipo sia un dato carattere e viene usata principalmente nella lettura di caratteri da file.

Le librerie POSIX utilizzate sono dirent.h, sys/stat.h, glob.h e pthread.h. La libreria dirent.h permette la gestione delle cartelle e viene utilizzata per scandire i file e eventualmente le sottocartelle delle directory passate in input. Sys/stat.h viene utilizzata per la funzione lstat e quindi per distinguere il tipo di file, tra file regolari, cartelle e link simbolici. La libreria glob.h viene usata per il globbing delle espressioni regolari nel caso queste siano riferite a –explude o –ignore, mentre per i file di input si fa affidamento sul globbing effettuato dalla shell. Infine, pthread.h è la libreria usata per poter utilizzare i threads.

L’unica libreria né ISO né POSIX, ma presente unicamente in glibc, è getopt.h, che serve per la lettura di opzioni e i loro argomenti in input a linea di comando.

# Trie.c

Questo modulo implementa un trie (anche noto come albero digitale) per il mantenimento in memoria delle parole lette dai file. Un trie è un tipo di albero di ricerca dove l’ordinamento dei nodi è ottenuto mediante chiavi di tipo stringa; la radice è la stringa vuota e ha un figlio per ogni possibile carattere del *CHARSET* (alfabeto latino, alfabeto ASCII, in questo caso l’alfabeto italiano più le 10 cifre). Ad ogni nodo è quindi associata una data stringa corrispondente alla concatenazione di una lettera per ogni nodo attraversato nel cammino dalla radice al nodo stesso. Ogni nodo quindi contiene 3 dati:

* La stringa associata al nodo *value*
* Il numero di occorrenze di quella stringa nei file *occurrencies*
* Un array di puntatori ad altri nodi per ogni carattere del *CHARSET* *children*

Una implementazione alternativa migliore consiste nel salvare nel nodo solo il carattere associato invece che l’intera stringa, mentre un’implementazione ancora migliore consisterebbe nel non salvare alcun carattere e ottenere la stringa valutando volta per volta a quale puntatore si è acceduto. La complessità spaziale dell’implementazione corrente è dove è il numero di parole ed è il numero di caratteri della parola i-esima.

## Add()

La funzione *add()* aggiunge una data parola al trie, richiamando una seconda funzione *\_add()* ricorsiva che necessità di avere anche un parametro *level* che rappresenta, durante le chiamate ricorsive, a che livello di profondità dell’albero ci si trova. La funzione *\_add()* inizialmente scorre l’albero selezionando il nodo corrispondente al carattere in posizione *level* e prosegue se il nodo non è NULL. In caso affermativo, se *level* è pari alla lunghezza della stringa, la stringa è già presente nel trie e il valore di *occurrencies* si incrementa di 1, altrimenti si incrementa *level* e si chiama ricorsivamente **\_***add()­* col nuovo valore di *level* e col nodo corrente al posto della radice. Se invece si è incontrato un nodo NULL, allora la parola che si vuole inserire non è presente nel trie e viene chiamata un’altra funzione, *addRecord()*, per aggiungerla. Il flag serve per evitare di richiamare nuovamente addRecord quando le chiamate ricorsive ritornano.

## AddRecord()

La funzione crea uno o più nodi per salvare una nuova parola nel trie, dove il numero di nodi creati sarà pari alla differenza tra *level*e *strlen*() della parola. Si assume che il nodo ricevuto come argomento sia l’ultimo nodo non NULL nella discesa nell’albero. Viene creato un nuovo nodo che andrà a contenere:

* I primi *level* caratteri della parola in *value*
* Il valore 0 di *occurrencies*
* Un array di puntatori a NULL per ogni carattere del *CHARSET* in *children*

Il puntatore al nuovo nodo viene quindi salvato nella posizione consona nell’array di puntatori del nodo passato come argomento.

Quindi se *level* è uguale a *strlen()* della parola, si incrementa *occurrencies* di 1, altrimenti si incrementa *level* e si richiama *addRecord()* col nuovo valore di level e col nodo appena creato come nodo di partenza.

## Search()

La funzione search restituisce 1 se la stringa è non NULL e presente nell’albero e 0 altrimenti. Si basa su una seconda funzione \_*search()* che necessita del parametro *level* per le chiamate ricorsive. La funzione si comporta in modo molto simile a \_*add()*, prendendo in considerazione il nodo corrispondente al carattere in posizione *level* nella stringa e controllando che il nodo non sia NULL. In caso affermativo, se *level* è uguale a *strlen()* della parola, si ritorna 1, altrimenti si incrementa *level* e si richiama **\_***search()* col nuovo valore di *level* e col nodo corrente come nodo di partenza. Se ad un qualsiasi livello di ricorsione non si ha una corrispondenza, si ritorna 0.

## getIndex()

Questa funzione dato un carattere ottiene l’indice corrispondente nell’array di puntatori di ogni nodo, sfruttando laddove possibile le proprietà del codice ASCII. Se il carattere è una cifra, viene convertito in un indice da 0 a 9 (quindi i primi 10 puntatori di ogni nodo corrispondono alle 10 cifre), altrimenti si ottiene l’indice da 0 a 26 corrispondente alla lettera a cui si somma 10. Modificando in modo opportuno questa funzione e cambiando il valore di *CHARSET* si può facilmente modificare il set di caratteri supportati.

## writeTrie()

La funzione scrive il trie sul file indicato. Se il valore di *occurrencies* è maggiore di 0 il nodo contiene una parola valida e la funzione *writeNodeInformation()*scrive una nuova riga sul file. Quindi, si richiama ricorsivamente *writeTrie()* su ognuno dei puntatori contenuti nel nodo. Dato che i puntatori sono fatti corrispondere a caratteri ordinati alfabeticamente (con prima le cifre da 0 a 9 e poi le lettere dell’alfabeto) da *getIndex()* e che un nodo è sempre lessicograficamente inferiore rispetto ai suoi figli, questo assicura la scrittura su file in ordine alfabetico.

# BST.c

Questo modulo implementa un albero binario di ricerca associato al trie delle parole, dove il contenuto di un nodo è costituito, oltre ai puntatori al figlio destro e sinistro, da un puntatore a un nodo contenente una parola valida del corrispondente trie. A causa di ciò questo modulo dipende da trie.c e nel suo header viene incluso trie.h.

La chiave che stabilisce l’ordinamento di quest’albero è il valore di *occurrencies* dei nodi corrispondenti a parole del trie.

Le uniche funzioni di questo modulo sono l’aggiunta all’albero e la scrittura dell’albero su file, e il suo uso nel progetto è limitato all’implementazione della funzionalità --sortbyoccurrency.

## addBST()

La funzione aggiunge il nodo del trie al BST. Se il nodo corrente è NULL, viene creato un nuovo nodo nel quale viene salvato il nodo del trie, altrimenti si visita il figlio sinistro o destro dell’albero, a seconda del valore di *occurrencies***,** e si richiama ricorsivamente *addBST()* usando come nodo di partenza il figlio sinistro o destro.

## writeTree()

La funzione scrive il BST sul file indicato, mediante una visita simmetrica dell’albero e utilizzando *writeNodeInformation()* contenuta in trie.c per la scrittura sul file del campo *wordInfo*, che altro non è che un nodo del trie.

# Swordx.c

## absPath()

La funzione richiama *canonicalize*\_*file*\_*name()* per restituire la path assoluta del file, risolvendo eventualmente i link simbolici.

## \_isalphanum()

La funzione restituisce 1 se la stringa passatagli contiene almeno una cifra numerica, 0 altrimenti

## getBlacklist()

Questa funzione popola un trie con le parole dei file contenuti nello stack passato in input. In questo modo è implementata la funzionalità –ignore: i file contenenti parole da ignorare sono contenuti nello stack ed ad uno ad uno vengono estratti e le loro parole aggiunte al trie; successivamente quando si andranno a prelevare parole dai file in input al programma basterà controllare per ogni parola se essa è contenuta in questo trie ( se lo stack è vuoto, questo trie sarà NULL e la ricerca ritornerà banalmente 0 ). La scelta di utilizzare un trie invece di un array o una linkedList è per motivi di efficienza : la ricerca nel trie è nel caso medio e in quello peggiore (dato che non c’è nessuna operazione di bilanciamento), mentre l’inserimento è . In una linkedList avremmo inserimento in e una ricerca in , mentre con un array, oltre a un costo iniziale per determinarne la dimensione, si può inserire in e dopo aver speso un tempo medio di per ordinarlo mediante un quicksort si possono cercare elementi con un tempo nel caso peggiore. Tuttavia abbiamo considerato più critico il tempo di ricerca che di inserimento, poiché in applicazioni comuni abbiamo pensato che il numero di parole valide da confrontare con la blacklist debba essere strettamente maggiore del numero di parole da ignorare. Inoltre per utilizzare gli array si sarebbero dovuti allocare spazi di memoria spropositamente grandi per poterli utilizzare in (quasi) ogni contesto o in alternativa effettuare letture preliminari dei file per stabilire una dimensione consona. Quindi abbiamo deciso di utilizzare un trie, con l’ulteriore vantaggio di aver già implementato tutte le funzioni per la sua gestione.

## getWord() e \_getWord()

Queste due funzioni si occupano di leggere una parola da un file. La funzione *getWord()* richiama la seconda ed effettua dei controlli aggiuntivi, dei filtri, per saltare parole che per una ragione od un altra non vanno contate. La funzione \_*getWord()* considera una parola come una qualsiasi sequenza di caratteri contigui esclusivamente alfanumerici, ovvero due parole possono essere spezzate, oltre che dagli spazi o dalla newline, anche da un qualsiasi simbolo, segno di punteggiatura o carattere speciale.

### \_getWord()

Una limitazione che questa funzione impone al programma è che la dimensione massima di una parola è 500 caratteri, resa evidente dall’allocazione del buffer nella stack area. Questa limitazione è necessaria perché un limite alla dimensione della parola deve essere stabilito, per evitare di sottomettere in input parole spropositatamente lunghe, tuttavia un limite di 500 caratteri è anche tutt’altro che stretto e quindi non offre una reale limitazione alle capacità del programma. Alternativamente si poteva allocare uno spazio nell’heap da ampliare dinamicamente se necessario ma non abbiamo considerato plausibile uno scenario comune in cui era necessario contare parole così lunghe. La presenza di questa limitazione inoltre permette di interrompere l’analisi di un file binario, nel caso sia più grande di 500 byte, vista l’assenza di controlli sul tipo di file.

Mediante un ciclo while la funzione legge e scarta ogni carattere non alfanumerico e non EOF, quindi se EOF è stato incontrato ritorna NULL ( e ciò indica che non c’è più nulla da leggere dal file) altrimenti rimette in posizione l’ultimo carattere letto. Ora un secondo ciclo while legge caratteri finché questi sono alfanumerici e ogni carattere viene inserito nel buffer e un contatore, che rappresenta sia il numero di caratteri letti che la prossima posizione vuota nel buffer, viene incrementato. Al termine di questo ciclo si alloca uno spazio sull’heap consono alla parola e si copiano i caratteri scritti sul buffer in questo spazio, quindi se ne ritorna il puntatore.

### getWord()

Questa funzione svolge un’operazione di filtro sulle parole ritornare da *\_getWord()*, a seconda del valore dei parametri passati. Il parametro *min* indica la lunghezza minima della parola per essere considerata ed implementa la funzionalità —min, il parametro *ignoreTrie* è l’albero contenente le parole da ignorare ed è collegato alla funzionalità –ignore, invece *flags* è un bit field che a questa funzione interessa per il bit associato alla funzionalità –alpha.

Il ciclo do-while preleva una parola *ret*mediante *\_getWord()*, poi la confronta con una serie di condizioni, che se sono verificate provocano il rilascio della parola corrente e la lettura della prossima. La condizione che però deve essere verificata è prima di tutto la *ret != NULL* **:** in caso contrario significa che il file non ha più parole da leggere e quindi non ha senso ripetere il ciclo. Le altre condizioni sono :

* *Strlen(ret) < min* : Se la parola letta è più corta del parametro *min* (In questo modo è implementata la funzionalità –min)
* *Search(ret,ignoreTrie)* : Se la parola letta è tra le parole da ignorare (In questo modo è implementata la funzionalità –ignore)
* *(flags & ALPHA\_FLAG) && \_isalphanum(ret)* : Se il flag della funzionalità –alpha è settato e la parola contiene almeno una cifra (In questo modo è implementata la funzionalità –alpha)

Il ciclo dunque non si ripete e la parola letta viene ritornata solo se almeno una delle seguenti condizioni è verificata:

* La parola corrente è NULL
* Le 3 condizioni precedenti non sono verificate

## sbo() e sortTrie()

Queste funzioni si occupano di implementare la funzionalità –sortbyoccurrency, con sbo che richiama la seconda per creare un BST ordinato per numero di occorrenze delle parole contenute nel trie e che poi chiama la funzione writeTree di BST.c per scrivere il BST su file. *sortTrie()* prende in input un trie e un BST, e aggiunge ogni nodo con l’attributo occurrencies strettamente positivo al BST.

arraytoStack()

Questa funzione converte un array di path, riferiti a file o cartelle, in uno stack di soli file. Inoltre richiede uno stack di file da escludere (per la funzionalità –explude) e il bit field *flags*dal quale ci interessano i bit riferiti alle opzioni –recursive e –follow. Lo stack dei file da escludere viene risolto dalla funzione *expand()* in uno stack di soli file. Quindi si usano la struct e le macro definite da stat.h per distinguere i diversi tipi di file nell’array di input. I file regolari vengono inseriti direttamente nello stack di output, mentre le cartelle vengono analizzate nella funzione *scan()*, tenendo conto anche dello stack dei file da escludere e dei flag.

## Expand()

Questa funzione ricevuto uno stack di pathname lo converte in uno stack analogo dove però eventuali espressioni regolari sono state risolte nelle path corrispondenti. La funzione viene usata esclusivamente nel contesto di convertire i pathname delle opzioni –ignore e –explude, permettendo di utilizzare espressioni regolari come argomenti, passandole tra doppi apici. Ogni elemento dello stack di input viene passato alla funzione *glob()* e in caso di matching, se ci sono cartelle, vengono passate alla funzione *scan()* senza flag e con uno stack anonimo (di norma, sarebbe stato lo stack di –explude), se ci sono file o link vengono inseriti direttamente nello stack di output (i link vengono risolti da *absPath()*).

## Main()

La gestione degli argomenti a linea di comando avviene attraverso la libreria getopt.h, in particolare mediante la funzione *getopt\_long\_only()*, che permette di accettare opzioni sia a singolo trattino che doppio trattino, e supporta opzioni a singolo trattino multi-carattere. Le opzioni multi-carattere sono gestite attraverso un array di *struct option*. Ogni struct contiene il nome dell’opzione, se richiede o no argomenti, un puntatore che non è necessario ai nostri scopi e il valore di ritorno della funzione *getopt\_long\_only()* in caso di matching. Le opzioni singolo-carattere invece sono gestite direttamente nella chiamata a *getopt\_long\_only()*, la quale accetta *argc*, *argv*, una stringa che rappresenta le opzioni singolo-carattere (con i “*:*”che stanno a significare che l’opzione della lettera precedente richiede un argomento), l’array di *struct option* e un ultimo parametro a NULL che non ci interessa. La funzione restituisce il valore dell’opzione (il carattere nel caso di quelle singolo-carattere e il numero definito nelle struct option nell’altro caso) se viene incontrata una opzione valida in *argv* e -1 quando non ci sono più corrispondenze. L’argomento di una opzione viene salvato nella variabile *extern optarg*, mentre argomenti non relativi ad alcuna opzione sono permutati in fondo ad *argv* automaticamente. Una volta analizzate tutte le opzioni, in *argv* restano quindi da analizzare tutti i file e le cartelle considerati di input al programma; le espressioni regolari sono gestite direttamente dalla shell. Nel caso delle opzioni –min e –output il valore viene salvato in una variabile, per –ignore e –explude il valore viene invece inserito in un apposito stack. Per le opzioni –recurisve, --follow, --alpha e –sortbyoccurrency viene effettuato un OR tra delle macro e una variabile *unsigned char* da 1 byte inizialmente di valore 0. Le macro sono definite in questo modo:

#define RECURSE\_FLAG (1<<0) //00000001

#define FOLLOW\_FLAG (1<<1) //00000010

#define ALPHA\_FLAG (1<<2) //00000100

#define SBO\_FLAG (1<<3) //00001000

Quindi per sapere se un’opzione è attiva basterà fare l’AND tra il bit field e la macro adatta, l’espressione ritornerà un valore diverso da 0 se l’opzione è attiva.

I valori di –min e –output vengono salvati in un array *args*, anche se NULL, dato che successivamente possono essergli assegnati valori di default.

Una volta terminato di analizzare le opzioni, restano da recuperare gli argomenti di input. La variabile *extern optind* indica il prossimo indice da leggere in *argv* e tutti gli elementi di *argv* da quell’indice fino ad *argc* sono gli argomenti di input, che vengono salvati in un array *params*. Successivamente questo array, che può contenere sia file che cartelle, viene convertito in uno stack di soli file dalla funzione *arrayToStack()*.

A questo punto viene analizzato il parametro dell’opzione –ignore, anche nel caso sia NULL, trasformando lo stack dei suoi parametri in uno stack di soli file, e successivamente creando un trie di parole da ignorare che viene popolato dalla funzione *getBlacklist()* utilizzando quest’ultimo stack.

Quindi viene chiamata la funzione *execute()* che esegue l’analisi dei file di input considerate le varie opzioni, passate attraverso l’array *args*, il trie delle parole da ignorare *ignoreTrie* e il bit field *flags*.