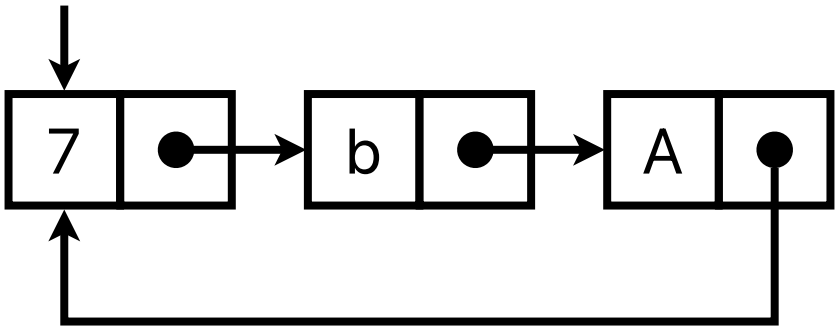
Università degli studi di Firenze

Corso di Architetture degli Elaboratori

Progetto Assembly RISC-V: Gestione liste circolari

2022 / 2023



Nome: Riccardo Giannuzzi

Data di consegna: 10/09/2023

Versione Ripes: 2.2.5

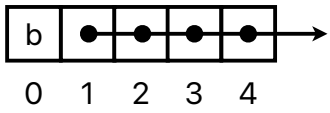
Sommario

Analisi del Problema

Riassunto testo del problema

Implementare un codice RISC-V che gestisce le operazioni fondamentali di una lista concatenata circolare. Ogni nodo della lista conterrà un carattere ASCII, con valore tra 32 e 125 compresi, e il puntatore al nodo successivo.

Quindi, in memoria, il primo byte conterrà l'informazione del nodo e i quattro successivi il puntatore al successivo.



Il puntatore alla testa della stringa (*pHead)* indicherà il primo nodo della lista e potrà essere modificato durante l'esecuzione del programma. Non sono invece previsti puntatori aggiuntivi.

Per la gestione delle operazioni da effettuare sulla lista, dovrà essere implementato un main, che dovrà elaborare la stringa di input del programma, che conterrà una serie di comandi separati da ~ (ASCII 126).

Tra i comandi contenuti nella stringa dovranno essere eseguiti soltanto quelli ben formati, ignorando tutti gli altri. La stringa di input non dovrà contenere più di 30 comandi.

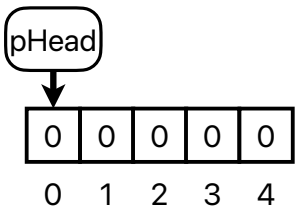
L'ordinamento dei caratteri è Maiuscole > Minuscole > Numeri > tutto il resto.

Limite al numero di comandi in input

Dato che la stringa di input non può contenere più di 30 comandi, il main controllerà se questo limite viene superato, e nel caso, si interromperà senza eseguire tutti i restanti comandi contenuti nella stringa, anche se essi fossero validi. Vengono contati come comandi anche quelli mal formati. Altrimenti si potrebbero dare in input infiniti comandi mal formati. Ad ogni modo, se tale limite si rivelasse un problema, il numero massimo di comandi può essere incrementato cambiando il valore con cui viene inizializzata la variabile *maxInputs*.

Lista vuota

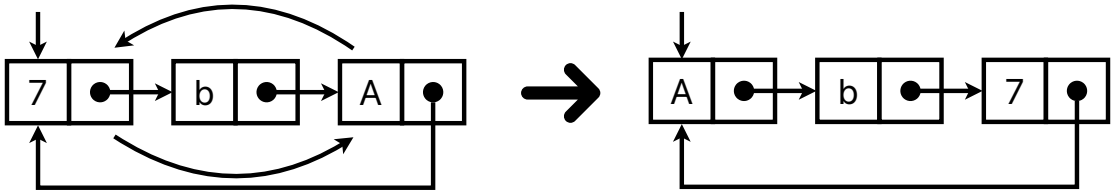
Per implementare le operazioni sulla lista, è necessario stabilire una convenzione che indichi se la lista è vuota oppure contiene almeno un nodo. Questo si può implementare in svariati modi. Nel caso di questo programma, la lista si ritiene vuota, quindi da inizializzare, nel caso in cui, il contenuto e il puntatore al successivo di pHead (il primo nodo della lista o testa della lista) siano entrambi nulli.



Scambiare due nodi

Sia la funzione REV, che la funzione SORT, necessitano di scambiare le posizioni dei nodi. Tale scambio può essere fisico, se si scambiano le posizioni dei nodi in memoria. Oppure logico, se si scambiano soltanto i contenuti dei nodi ma si lasciano invariate le posizioni in memoria.

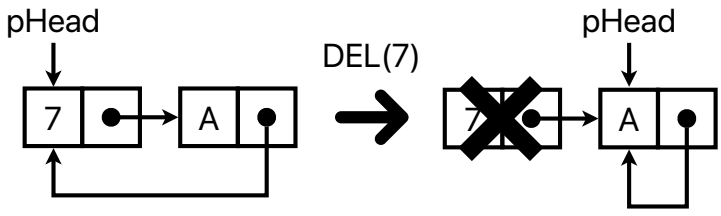
Dato che questo non genera complicazioni nell'implementazione degli altri metodi, la scelta ricade nello scambio logico, che è più semplice da implementare e richiede un minore scambio di dati, dato che il contenuto occupa un solo byte. Questo significa che la sequenza dei nodi in seguito alla chiamata di queste due funzioni resterà invariata, ma non vale lo stesso per il loro contenuto.



Eliminazione della testa

La funzione DEL elimina tutti i nodi che contengono il valore passato come parametro. Questo significa che anche la testa, se contiene quel valore, può essere eliminata. Quindi, dato che una lista deve avere una testa, è necessario stabilire il comportamento nel caso in cui essa sia stata eliminata.

Per mantenere lo stesso ordine relativo degli elementi, dopo che è stata eliminata la testa, la soluzione migliore è la più immediata. Cioè rendere la nuova testa della lista il successivo di quella eliminata.



Implementazione Main

Per interpretare il contenuto della stringa, il main inizia a scorrere la lista partendo dal primo carattere, stabilendo se i caratteri compresi tra due tilde formano un comando valido. Se il comando appena letto è ben formato, esegue la chiamata alla funzione corrispondente passando, se necessario, il parametro. La lettura dei comandi termina solo se finiscono i caratteri contenuti nella stringa (quando viene letto dalla memoria il carattere con codifica 0) oppure, se si supera il limite massimo di comandi definito nella variabile *maxInputs* (inizializzata a 30).

Controllo formattazione comandi

I comandi validi all'interno di un input possono essere di tre tipi:

1. Il primo comando " ... ADD(x) ... ~"
2. Comando intermedio "~ ... ADD(x) ... ~"
3. L'ultimo comando "~ ... ADD(x) ... "

Inoltre prima, e dopo l'operazione, devono esserci soltanto spazi vuoti.

Input tipo: " ADD(x) ~ADD(y)~ADD(z) ~ PRINT " (4 comandi validi)
Questa classificazione vale anche se il primo e l'ultimo comando sono nulli: "~ ADD(x) ~ADD(y)~" (questo input contiene 4 comandi ma solo 2 validi)
Inoltre, dato che la ADD e la DEL acquisiscono un parametro, è necessario controllare che il carattere sia accettabile (compreso tra 32 e 125) e che sia un solo carattere.

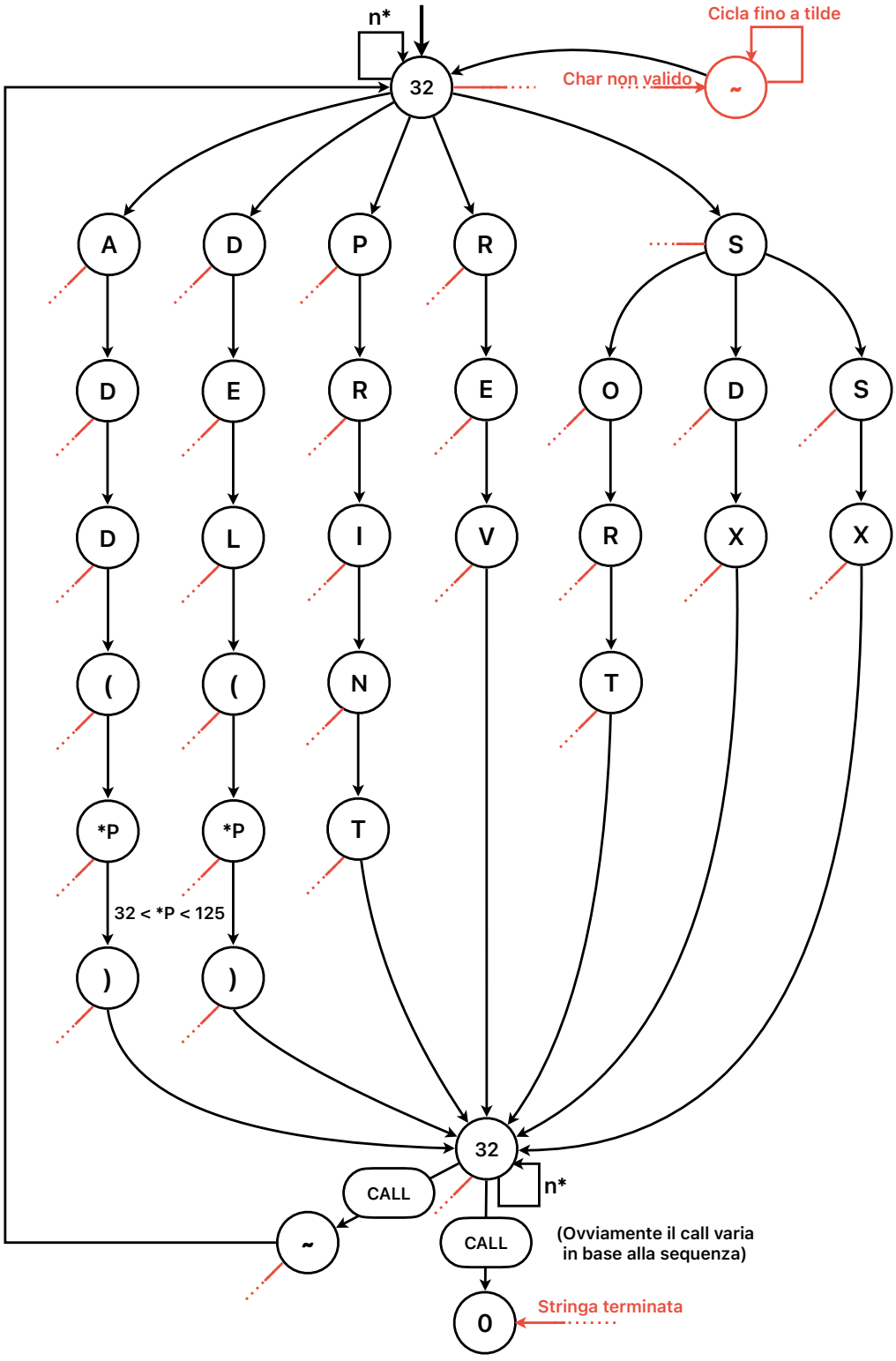
Tenendo in considerazione tutte queste condizioni, è possibile determinare se un comando è ben formato tramite uno pseudo-automa a stati, che cerca una specifica sequenza. Tale sequenza è una serie di n spazi vuoti seguita dai caratteri di una operazione, con se necessario un parametro, un'altra sequenza di n spazi vuoti e infine, per "chiudere" il comando, una tilde o la fine della stringa. Quindi si usano le tilde all'interno dell'input soltanto per chiudere un comando e confermare che nella sua interezza sia valido. Questo permette di analizzare tutti e tre i tipi di comando elencati in precedenza. Per esempio l'input riportato prima sarebbe analizzato in questo modo:

" ADD(x) ~**|**ADD(y)~**|**ADD(z) ~**|** PRINT **|**"
l'esecuzione avviene in corrispondenza del simbolo "**|**". Nei primi tre casi subito dopo aver chiuso il comando con una tilde e nell'ultimo caso invece, il comando è stato chiuso dalla fine della stringa. Così, anche se la stringa finisce, se fino a quel momento il comando letto era valido, verrà eseguito comunque.

Nel caso invece il comando si riveli non valido ad un certo carattere, si procede in avanti nella stringa, ignorando tutti i caratteri fino a che non si supera una tilde e si ricomincia a cercare la sequenza.

Schema controllo formattazione

Il seguente schema rappresenta graficamente i passaggi per controllare la formattazione dei comandi all'interno della stringa di input.



Implementazione Funzioni

Tutte le funzioni devono controllare prima di tutto se la lista è vuota e in caso, ad eccezione della ADD, non fare nulla e passare al prossimo comando, dato che non avrebbero comunque effetto su una lista vuota. Per eseguire tale controllo è sufficiente leggere il contenuto dei 5 byte puntati dal pHead e verificare se sono tutti 0.

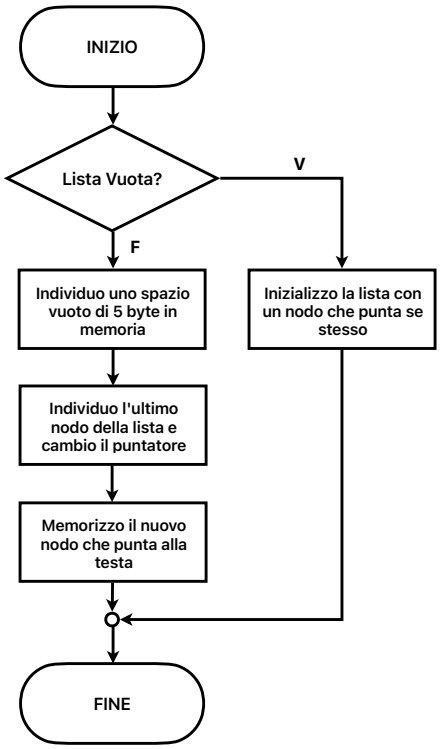
Tutte le funzioni che presentano chiamate innestate salvano subito nello Stack l'indirizzo di ritorno. Inoltre, quando è necessario salvare il valore dei registri temporanei nello Stack, si rispettano tutte le convenzioni relative ad esso, compreso l'ordine di inserimento e l'ordine di estrazione.

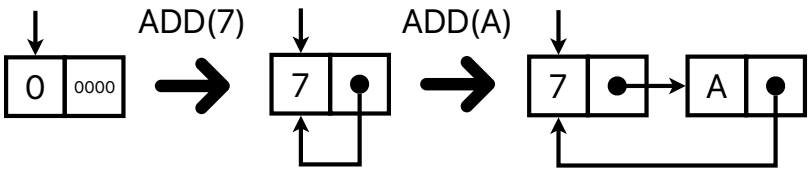
Per quanto riguarda le convenzioni sui registri delle funzioni, l'unico registro necessario per passare e restituire valori è a0, ad eccezione delle funzioni QuickSort e Compare a cui viene passato il secondo valore in a1.

ADD

Nel caso in cui la lista è vuota, si inizializza il primo nodo nello spazio di memoria puntato dal pHead, facendolo puntare a se stesso e mettendo come contenuto il carattere passato come parametro.

Invece, se la lista contiene almeno un elemento si effettua un inserimento in coda. Quindi si salva nello Stack il valore passato come parametro e si individua l'ultimo nodo. Per fare ciò si chiama la funzione cercaPrecedente, passando come parametro la pHead. Si salva nello Stack anche l'indirizzo dell'ultimo elemento e poi si chiama la funzione che cerca uno spazio di memoria libero, passando come parametro l'indirizzo dell'ultimo elemento. Tale funzione procederà in avanti in memoria con salti di 5 byte, fino a che non ne troverà 5 liberi di fila e restituirà l'indirizzo del primo byte libero. A questo punto si recuperano tutti i valori dallo Stack e si modifica il puntatore dell'ultimo nodo in modo che punti al nodo inserito. Infine nello spazio libero, si salva il parametro nel primo byte e l'indirizzo della testa nei successivi 4 byte.





DEL

Dato che questa operazione può eliminare la testa, è necessario memorizzare in un registro una testa temporanea, che cambierà durante l'esecuzione. Questo è importante perché la testa è la condizione di fermata del ciclo e se viene eliminata dovrà essere spostata in avanti di uno. Quindi, alla fine della DEL, sarà restituito al main l'indirizzo della nuova testa della lista, dopo l'eliminazione, che dovrà essere sovrascritto nel pHead.

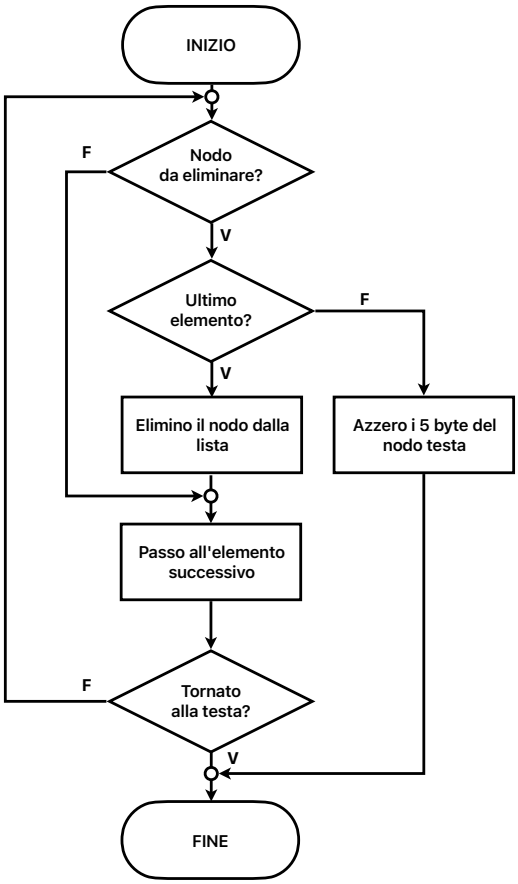
Si inizia a scorrere la lista partendo dalla testa e si controlla se il contenuto combacia con il parametro passato e nel caso si elimina il nodo dalla lista.

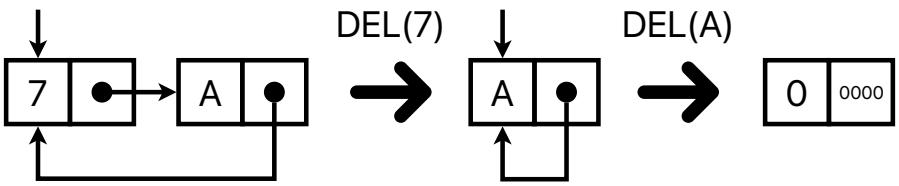
Prima di eliminarlo però è necessario controllare se è l'ultimo nodo presente nella lista, in quel caso si azzerano i 5 byte del nodo testa e la DEL termina.

Se invece sono presenti almeno due nodi si procede eliminando il nodo dalla lista. Prima di tutto si salvano nello Stack l'indice e la testa, poi si cerca il precedente dell'indice chiamando cercaPrecedente. Si recuperano i dati dallo Stack e si procede sovrascrivendo il puntatore del precedente con il puntatore del successivo del nodo da eliminare.

Questo fino a che non si sono controllati tutti i nodi della lista, cioè quando si raggiunge di nuovo il nodo testa.

Dato che non è stato implementato alcun sistema di Garbage Collection, i nodi eliminati occuperanno comunque lo spazio di memoria in cui erano stati memorizzati dalla ADD in precedenza.

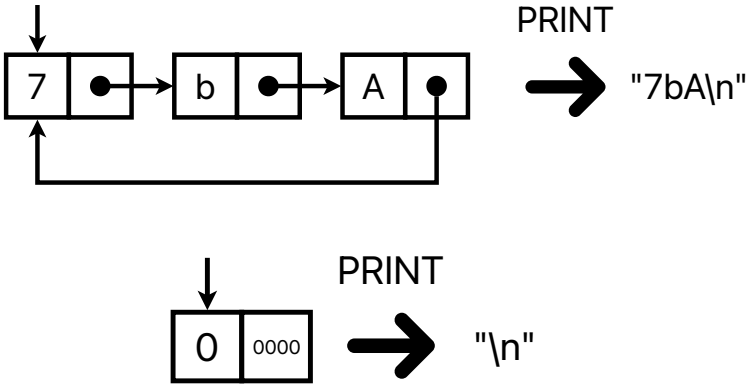


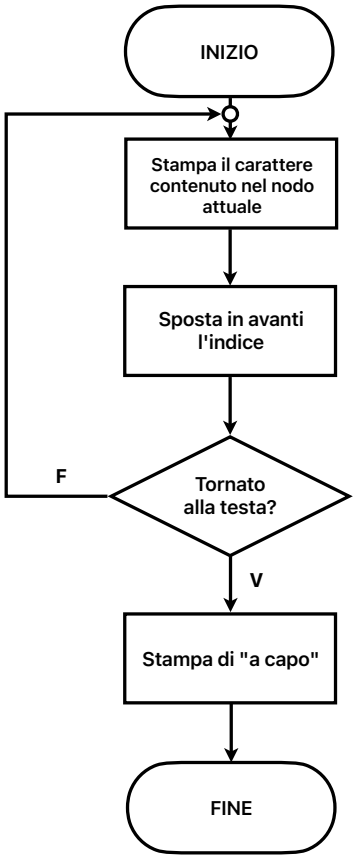


PRINT

Partendo dalla testa, stampa a schermo con delle ecall a printChar i caratteri nell'ordine in cui appaiono nella lista. Quando torna alla testa si interrompe il ciclo e viene stampato un "a capo".

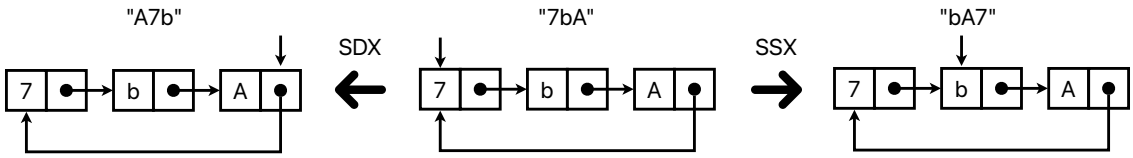
Se invece la lista è vuota non stampa alcun carattere ma stampa un "a capo".





SDX / SSX

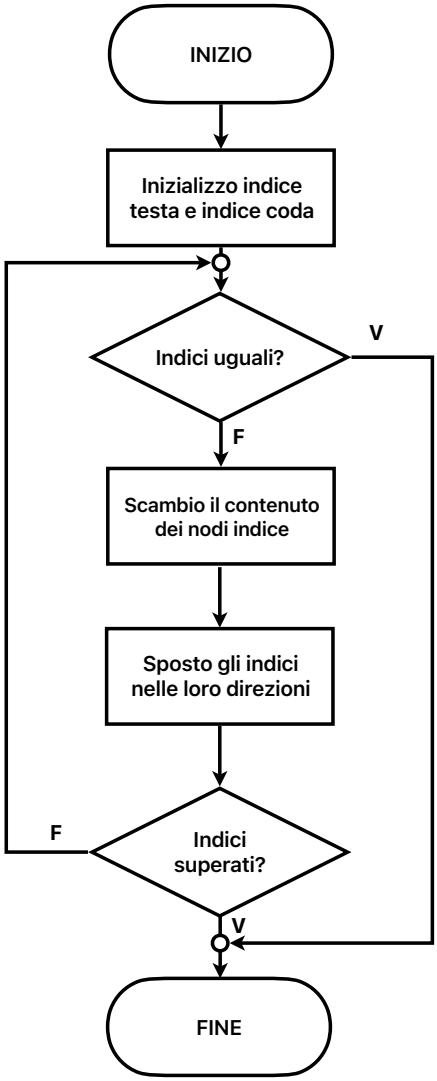
Per la proprietà delle liste circolari, è sufficiente spostare il pHead in avanti di un nodo per effettuare un SSX, oppure indietro di un nodo per effettuare un SDX.



Nel caso in cui ci fosse un solo nodo il puntatore alla testa rimane invariato in entrambi i casi perché è sia il suo successivo e il suo precedente.

REV

Prima di tutto si inizializzano due indici, il primo con l'indirizzo della testa e il secondo con l'indirizzo dell'ultimo elemento. Quindi si cerca l'ultimo elemento chiamando la funzione cercaPrecedente e passando come parametro l'indirizzo della testa della lista.

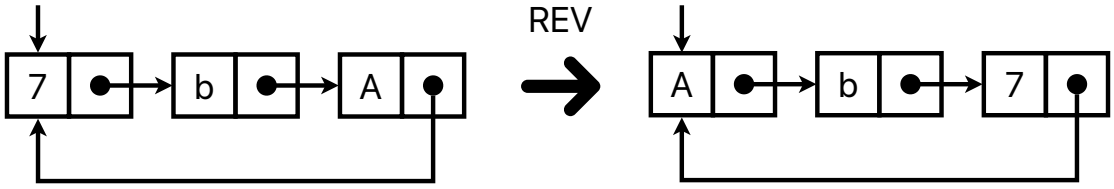


A questo punto, fino a che gli indici non sono uguali o si superano a vicenda, si continua a invertire il contenuto del nodo puntato dal primo indice, con il contenuto del nodo puntato dal secondo indice. Dopodiché si spostano gli indici nelle rispettive direzioni.

Per spostare il secondo indice indietro è necessario chiamare la funzione cercaPrecedente. Quindi prima della chiamata dovrà essere salvato nello Stack il valore del primo indice, per poi recuperarlo dopo la funzione.

Il controllo per verificare se gli indici sono uguali conviene effettuarlo effettuato prima della prima iterazione del ciclo in modo da controllare anche se la lista ha un solo elemento.

Il motivo per cui, non è sufficiente controllare solo se gli indici si sovrappongono, è il caso in cui il numero di elementi è pari, in cui gli indici non possono sovrapporre ma solo superarsi.



SORT

Compare

Dato che è stato definito un ordinamento che predilige alcune categorie di caratteri, non si possono comparare direttamente i valori ASCII dei caratteri. Quindi per compararli è necessario prima di tutto verificare quali sono le categorie dei due caratteri e aggiustare il loro "peso" in base ad esse. Questo si può fare semplicemente incrementando il valore ASCII con una add.

Quindi la compare essenzialmente si tratta di una "set less then" che aggiusta il peso dei valori nei registri a0 e a1 in modo che sia rispettato l'ordinamento definito dal testo del problema.
Restituisce 1 in a0 se a0 < a1 altrimenti restituisce 0.

QuickSort

Si tratta di una implementazione ricorsiva del QuickSort standard, che non adotta miglioramenti per i tempi di esecuzione o di tecniche ibride per ordinare le porzioni di lista più corte.

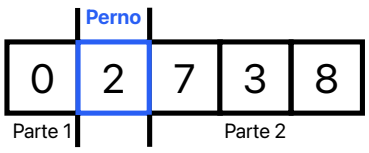
Dopo essere stato chiamato, passandogli come parametri gli indirizzi del primo e dell'ultimo elemento da ordinare, procede effettuando un partizionamento su quella parte della lista.

Una volta finita, la procedura di partizionamento, restituisce la posizione finale del perno, che sarà preceduto da elementi più piccoli e seguito da elementi più grandi.

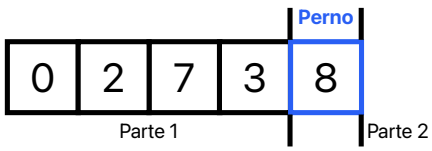
Al QuickSort non interessano i contenuti dei nodi, nemmeno del perno, perché gli scambi del contenuto degli elementi avvengono durante la procedura di partizionamento. Quello che gli interessa a questo punto è che, la parte della lista su cui era stato chiamato il QuickSort è stata divisa in due parti dal perno, che non è detto siano già ordinate

Tuttavia, ci sono due casi in cui si può dire che una parte, dopo un partizionamento, è sicuramente già ordinata, cioè se:

1. contiene un solo elemento



2. non contiene alcun elemento



Quindi si effettuano, le successive chiamate innestate al QuickSort, sulle due parti generate dal partizionamento, solo se contengono almeno due elementi. In questo modo quando tutte le sotto-parti della lista saranno ordinate, e di conseguenza lo sarà anche l'intera lista. Quando tutte le chiamate si fermano, inizia il processo di ritorno al chiamante.

Gestione ricorsività e Stack

Dato che il QuickSort esegue fino a 2 ulteriori chiamate a se stesso, oltre che alle altre funzioni, ogni volta che viene chiamato, è cruciale, per il suo funzionamento, ritornare al chiamante originale e che quindi vengano salvati tutti gli indirizzi di ritorno necessari. Per salvare tali indirizzi nell'ordine corretto si usa la memoria Stack. Questo permette in seguito, con una una serie di "pop" dallo Stack, di recuperare gli indirizzi nell'ordine corretto e di tornare, una volta che la lista è stata ordinata, al primo chiamante.

Partizionamento

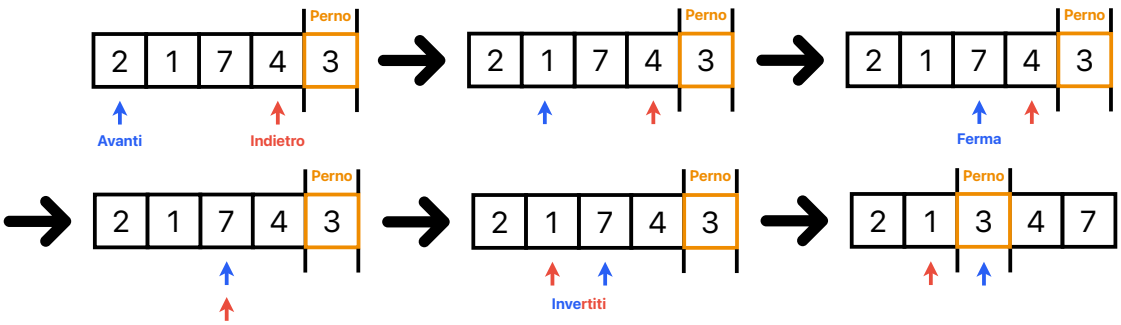
Riceve, come per il QuickSort nei registri a0 e a1 rispettivamente, l'indirizzo del primo elemento e l'indirizzo dell'ultimo elemento della porzione di lista su cui andrà ad agire. Come perno viene sempre scelto il nodo puntato dall'indirizzo a1, cioè l'ultimo elemento.

Poi viene inizializzato un primo indice con l'indirizzo del primo elemento e un secondo indice con l'indirizzo del penultimo elemento.

A questo punto si sposta in avanti il primo indice fino a che non incontra un valore maggiore o uguale del perno. Similmente dopo che si è fermato il primo indice, si sposta indietro il secondo fino a che non incontra un valore minore strettamente del perno.

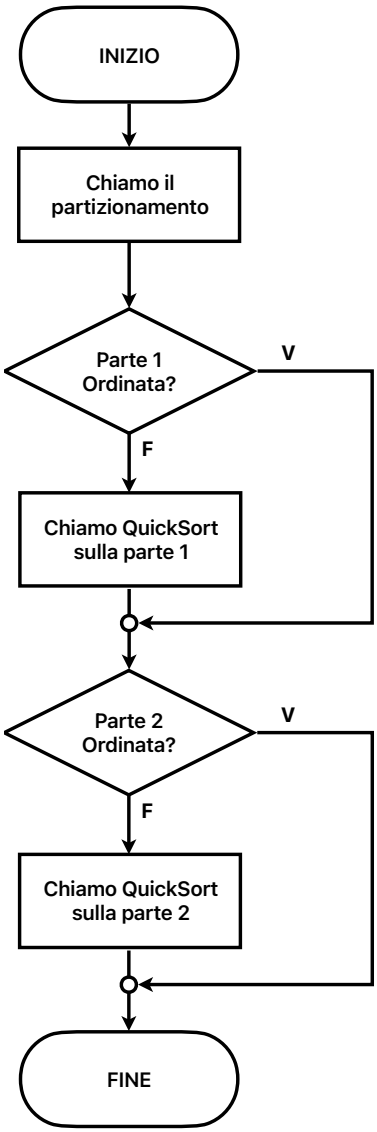
Se entrambi gli indici si fermano, significa che ci sono due nodi fuori posto e quindi vengono scambiati i loro contenuti, poi si torna a spostare gli indici nelle loro direzioni.

Nel mentre che vengono spostati gli indici si controlla se un indice supera l'altro. In quel caso la partizione è finita e la posizione finale del perno sarà quella in cui si trovava il primo indice nel momento in cui si sono superati. Quindi si scambiano i contenuti del nodo puntato dal primo indice e dell'ultimo nodo (il nodo preso come perno) e si restituisce l'indirizzo del primo indice al chiamante.

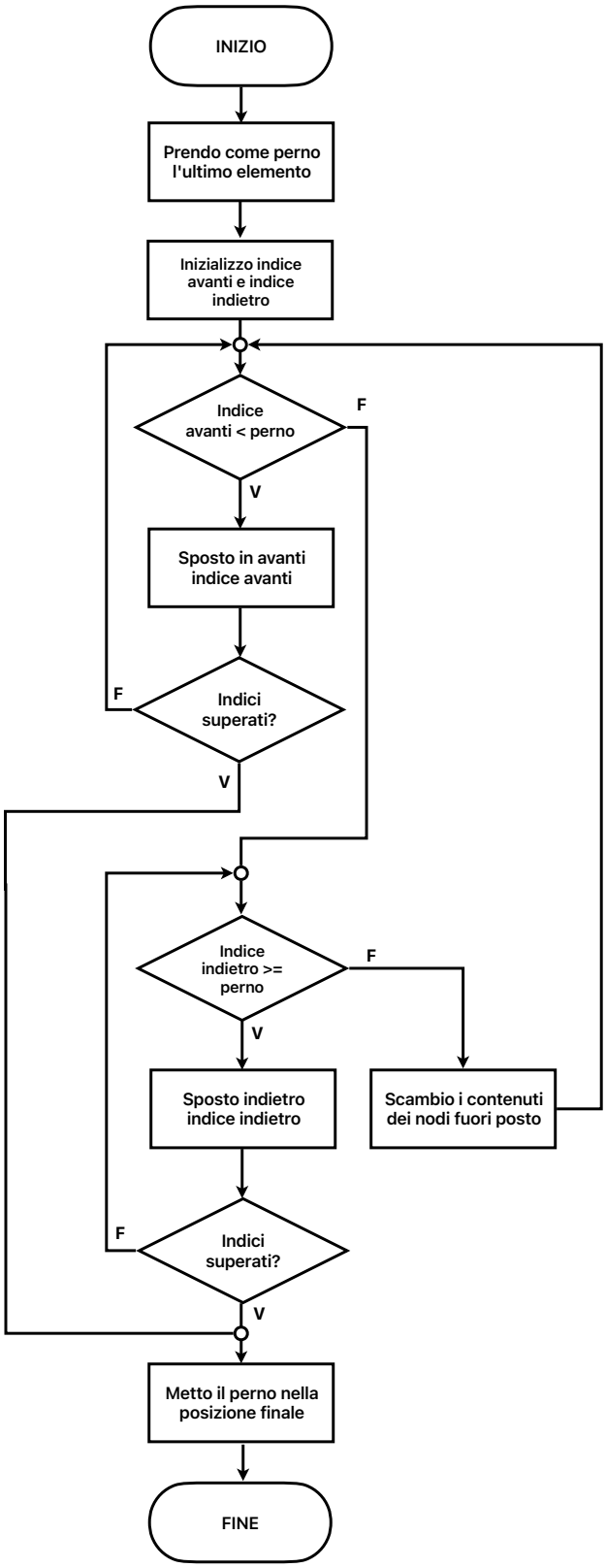


Diagrammi di flusso

QuickSort



Partizionamento



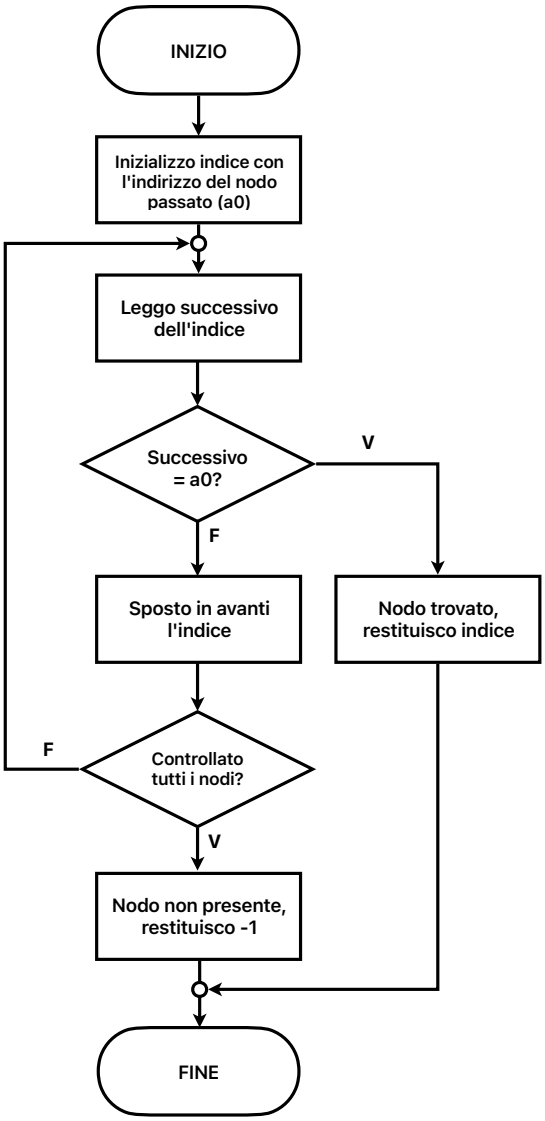
Funzioni Aggiuntive

Alcune operazioni ricorrenti, come la ricerca del precedente o la comparazione di due elementi, sono state implementate come funzioni aggiuntive oltre a quelle richieste esplicitamente dal testo. Questo per favorire la modularità del programma.

CercaPrecedente

Funzione chiamata all'interno di quasi tutti le funzioni della lista, cerca il precedente del nodo passato in a0 e restituisce il suo indirizzo sempre in a0.

Per cercare il precedente si scorre la lista partendo dal primo nodo, controllando se il *pAhead* di quel nodo contiene l'indirizzo del nodo di cui stiamo cercando il precedente. Se non corrispondono si passa al prossimo nodo e si ripete il controllo fino a che non si incontra il nodo che stiamo cercando.



Per evitare che, se venisse passato un indirizzo di un nodo che non appartiene alla lista, il ciclo continui a cercare all'infinito un nodo che non esiste è necessario controllare se si sono già controllati tutti i nodi appartenenti alla lista, cioè siamo tornati al nodo passato come parametro.

Nel caso in cui non si trovi il precedente il metodo restituisce in a0 -1 indicando che l'elemento di cui si voleva cercare il precedente non appartiene alla lista.

Dato che il testo impone di non utilizzare puntatori aggiuntivi oltre a pHead (che punta alla testa della lista), per cercare l'ultimo elemento è necessario scorrere tutta lista. Per fare ciò si può utilizzare questo metodo passando in a0 l'indirizzo della testa della lista.

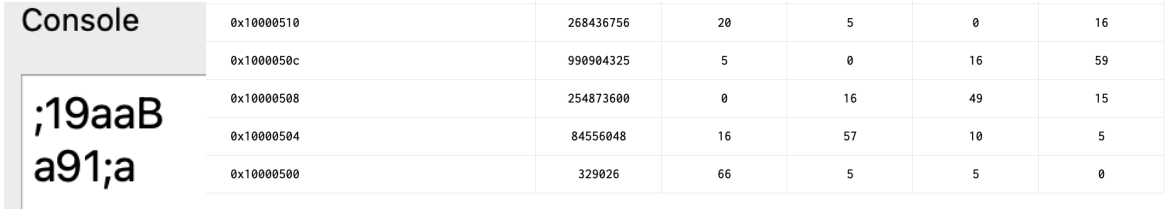
Test Data Set

Esempi del testo del problema

listInput = *"ADD(1) ~ ADD(a) ~ ADD(a) ~ ADD(B) ~ ADD(;) ~ ADD(9) ~SSX~SORT~PRINT~DEL(b)~DEL(B)~PRI~SDX~REV~PRINT"*

Output finale corretto = "a91;a"

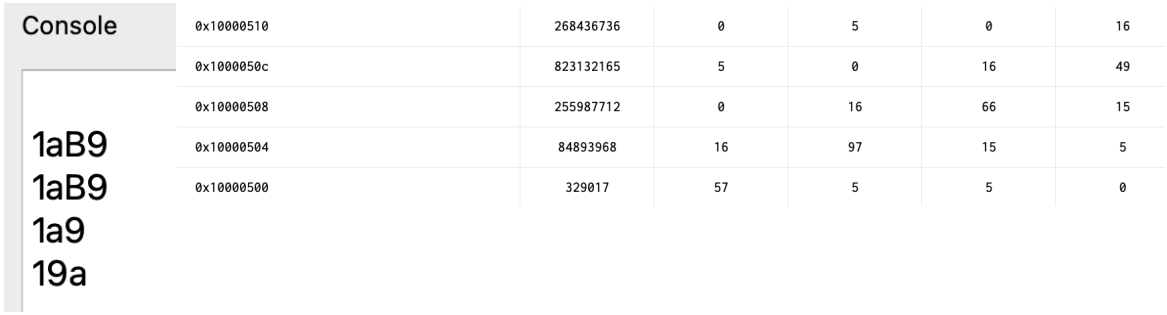
Risultato Esecuzione:



listInput = *"ADD(1) ~ SSX ~ ADD(a) ~ add(B) ~ ADD(B) ~ ADD ~ ADD(9) ~PRINT~SORT(a)~PRINT~DEL(bb)~DEL(B) ~PRINT~REV~SDX~PRINT"*

Output finale corretto = "19a"

Risultato Esecuzione:

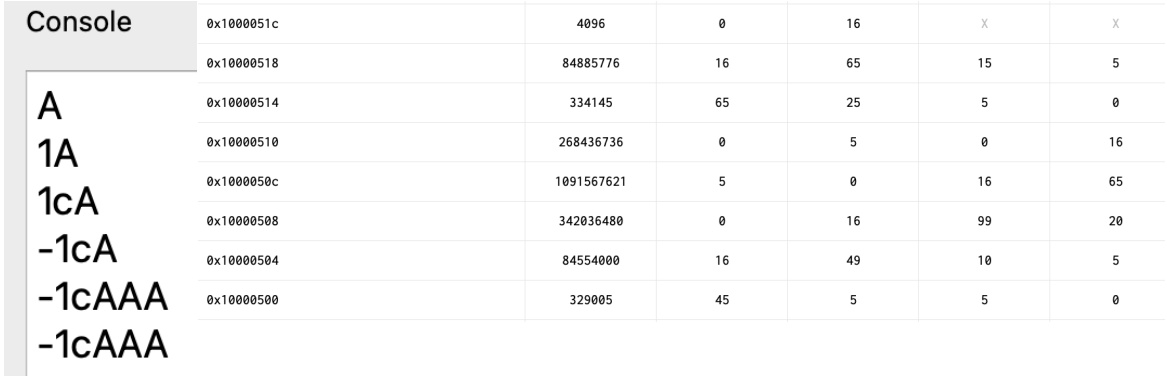


Sort

listInput = *"ADD(A) ~ SORT~PRINT~ ADD(1) ~ SORT~PRINT~ ADD(c) ~ SORT~PRINT~ADD(-) ~ SORT~PRINT~ SDX ~ ADD(A) ~ ADD(A) ~ SORT ~ PRINT~ REV ~ SSX ~ SORT ~ PRINT"*

Output finale corretto = "-1cAAA"

Risultato Esecuzione:



Casi particolari

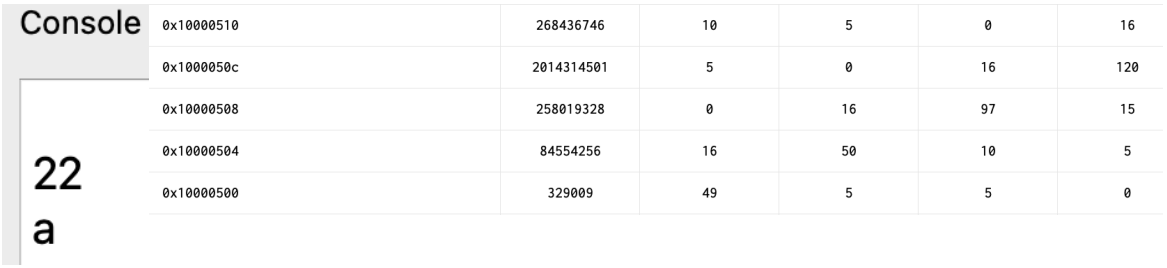
Questo input comprende:

- Rimozione di tutti i nodi dalla lista
- Superamento limite del numero di 30 comandi
- Parametri ADD e DEL sbagliati
- DEL che elimina la testa
- Nodi rimossi che restano in memoria

listInput = *"ADD(1) ~ ADD(AA) ~ DEL~DEL(11)~ ADD(2) ~ ADD(2) ~ DEL(1)~PRINT ~ DEL(2)~ ADD(a)~PRINT~ ADD(x)~~~~~~~~~~~~~~~~~~~~PRINT~"*

Output finale corretto = "a"

Risultato Esecuzione:



Chiamata delle funzioni su lista vuota

Chiamata delle funzione della lista su una lista vuota

listInput = *"ADD(1) ~ DEL(1)~PRINT~DEL(1)~SORT~SDX~SSX~REV~PRINT~ ADD(0)~ PRINT "*

Output finale corretto = "0"

Risultato Esecuzione: