Progetto Assembly RISC-V per il Corso di Architetture degli Elaboratori – A.A. 2020/2021 –

Gestione di Liste Concatenate

Pistolesi Boni Gabriele

7049722

gabriele.pistolesi1@stud.unifi.it

20/02/2022

Il programma implementa una lista concatenata doppia e mette a disposizione dell'utente 5 funzioni per manipolarla:

ADD: aggiunge un nodo e il rispettivo carattere, specificato tramite la codifica ASCII, in coda alla lista.

PRINT: stampa in console il contenuto di ogni nodo su una stessa riga, dal primo all'ultimo.

DEL: elimina un nodo dalla lista, cambiando gli opportuni puntatori e liberando la memoria occupata.

SORT: ordina i valori di ogni nodo secondo l'ordine specificato dal progetto.

REV: inverte la lista scambiando fra loro i puntatori di ogni nodo.

Per specificare le operazioni che si desidera eseguire è necessario fornirle, con la giusta formattazione tramite una stringa listInput che le conterrà.

Fornita la stringa di comandi il programma inizierà decodificandola e richiamando le opportune funzioni.

DECODING

Funzione che decodifica la stringa listInput contente i comandi da eseguire.

La procedura inizia con un loop che ignora eventuali spazi fino a che non trova un carattere da esaminare, una volta trovato si salta al "**CHECK_ADD**" che controlla se il carattere corrente è corrisponde alla codifica ASCII di "A", in caso positivo salta a "**CHECK_D1**" che incrementa il puntatore al carattere successivo e controlla che sia una D, se ha successo salta a "**CHECK_D2**" che effettua lo stesso controllo.

Se il controllo per la A fallisce l'istruzione non è una **ADD** e quindi si può saltare al check successivo, ovvero il "**CHECK_PRINT**", se invece falliscono i controlli per le lettere successive l'istruzione in esame è mal formattata e si salta a "**check_next_instruction**". In ultimo se tutti i controlli sulle lettere hanno successo si esamina la formattazione del valore di input, viene effettuato un salto a "**check_value_ADD**" si controlla quindi che il carattere dopo la seconda D sia una "(" salva il carattere successivo nel registro che verrà utilizzato dalla ADD, incrementando il puntatore si controlla che sia presente il carattere ")", se almeno uno di questi controlli fallisce si salta a "**check_next_instruction**", altrimenti continua con il controllo per la corretta formattazione, dove si ignorano eventuali spazi con un loop e al primo carattere rilevato si controlla che sia una tilde, se lo è viene chiamata la **ADD**, altrimenti si controlla che si sia arrivati a fine stringa, in caso positivo la formattazione risulta corretta e si chiama la **ADD** altrimenti si salta a "**check_next_instruction**".

Il "**CHECK_DEL**" segue la stessa implementazione del "**CHECK_ADD**", mentre gli altri check possono fare a meno di controllare la formattazione corretta del valore di ingresso. Finito il **DECODING** si entra un ultima volta in "**check_next_instruction**".

È necessario specificare che ogni funzione primaria (**ADD**, **PRINT**, **DEL**, **SORT**, **REV**) torna al **DECODING** una volta conclusa.

Funzioni principali:

ADD

La funzione **ADD** si occupa di aggiungere un nodo in coda alla linked list:

Appena viene eseguito un salto alla label **ADD** vieni fatta una jal a “**address_generator**” (procedura ausiliaria) che genera l’indirizzo di memoria pseudo-casuale dove verrà salvato il nodo da aggiungere. In seguito si effettua un controllo per assicurarsi che l’operazione che si sta eseguendo non sia la prima del suo tipo, se questo è il caso il PBACK (puntatore al precedente) e il PAHEAD (puntatore al successivo) vengono messi null (0xffffffff), il valore da aggiungere viene inserito in memoria e il puntatore alla coda si aggiorna copiandovi l’indirizzo pseudo-casuale generato.

Se invece è già stata fatta una operazione di **ADD** viene salvato il valore da aggiungere nell’opportuno indirizzo di memoria, il PAHEAD del nuovo e quindi ultimo nodo viene impostato null, il nuovo indirizzo viene sovrascritto al PAHEAD dell’ormai penultimo nodo ed infine il puntatore alla coda della linked list viene aggiornato con il nuovo indirizzo.

PRINT

La funzione **PRINT** scorre attraverso tutta la lista e ne stampa il contenuto su una riga:

Inizialmente si copia il puntatore alla testa della linked list in un registro temporaneo, così da poter iterare sulla struttura senza modificare il puntatore originale. Quindi si effettua un controllo sulla testa per verificare che sia stata eseguita almeno un’operazione di **ADD**, se non è così l’esecuzione di **PRINT** termina.

Se al contrario si può iterare sulla lista si esegue un ciclo denominato “**PRINT_loop**” dove, come prima cosa, si effettua un controllo sul puntatore: se è vuoto allora si esce dal ciclo si stampa il carattere “\n” per andare a nuova linea per un’eventuale futura **PRINT**; altrimenti si carica in a0 il carattere da stampare e viene chiamata la relativa System Call (con codice 11), si effettua la stampa, il puntatore viene aggiornato al nodo successivo e il ciclo riparte.

DEL

La funzione **DEL** cerca il nodo da eliminare e, se lo trova, libera la memoria occupata dallo stesso.

Anche questa funzione inizia copiando il puntatore alla lista e controllando se è stata fatta almeno una ADD, in caso negativo l’esecuzione della funzione termina.

Se è presente almeno un nodo il programma entra nel ciclo “**DEL_loop**” che scorre lungo tutta la linked list fino a che o trova il nodo da eliminare o arriva in fondo senza averlo trovato, in questo caso esce dalla funzione.

Nel primo caso, invece, trovato l’elemento ci si accerta della sua posizione:

Se il nodo si trova in testa alla lista, si salta a “**del_first_element**” il PBACK del secondo elemento viene impostato a null, mentre la memoria occupata dal primo elemento viene liberata in caso l’address_generator generi un nuovo indirizzo proprio dove è stato eliminato il nodo, infine il puntatore alla testa della lista deve essere aggiornato per puntare al nodo successivo a quello eliminato;

Se il nodo si trova in coda alla lista si salta a “**del_last_element**” ad essere messo null è il PAHEAD del penultimo elemento, mentre l’ultimo nodo viene eliminato liberando la memoria, il puntatore alla coda viene inoltre aggiornato per puntare al nuovo ultimo nodo;

Se invece il nodo da eliminare si trova in una posizione intermedia viene effettuato un salto a “**del_element**” il PAHEAD del nodo precedente deve essere messo uguale al PBACK del nodo successivo e viceversa, in questo modo il nodo da eliminare non sarà più raggiungibile e la memoria occupata viene liberata;

Se infine il nodo da eliminare è anche l’unico si salta a “**del_only_element**” e basta semplicemente liberare la memoria occupata e resettare i puntatori alla testa e alla coda della linked list.

SORT

Funzione che ordina la linked list implementando il bubble sort con flag (scambi avvenuti) secondo l’ordine specificato nel testo del progetto.

La procedura comincia copiando il puntatore alla testa della lista in un registro temporaneo e settando la flag al valore di default 0.

Dopodiché si entra nel ciclo “**SORT_loop**” che carica il primo elemento da confrontare, il puntatore al successivo e infine il secondo elemento. Successivamente si controlla che non si sia arrivati a fine coda (e quindi anche che ci sia un solo nodo) e, in caso positivo si effettua una jal a “**swap_check**” che decide se i due elementi in considerazione vanno scambiati oppure no. Se si devono scambiare si salta a “swap_element” dove vengono scambiati soltanto i contenuti nella posizione 4 dei nodi interessati, e non i nodi interi. Una volta scambiati i valori si ricomincia il **“SORT_loop”**, in questo modo ci si assicura che il valore più grande al termine del “**SORT_loop**” si trovi in coda alla lista, se è il secondo si troverà nel penultimo nodo e così via, garantendo una corretta implementazione del Bubble Sort.

Se invece siamo arrivati a fine lista si salta a “**check_swapped**” dove si controlla il valore della flag: se flag == 1 sono stati effettuati degli scambi e quindi si si ricomincia la procedura di **SORT**, se invece flag == 0 significa che non ci sono stati scambi e la lista è quindi ordinata e si può uscire dalla funzione.

REV

La funzione **REV** inverte l’ordine degli elementi scambiando i rispettivi puntatori al precedente e al successivo

Se la linked list è vuota si esce dalla funzione, altrimenti si effettua una copia della testa della lista e si entra nel ciclo “**REV_loop**”.

Nel ciclo vengono caricati i puntatori del nodo corrente in dei registri temporanei, del PAHEAD viene fatta un’ulteriore copia per poter passare poi al nodo successivo anche dopo aver scambiato i puntatori.

Per invertire il PAHEAD e il PBACK basta salvare il primo al posto del secondo e viceversa, si controlla quindi che il puntatore al nodo corrente non sia null, se lo è significa che si è arrivati a fine lista e si devo scambiare i puntatori alla testa e alla coda saltando a “**head_rear_swap**”, se invece la condizione non si verifica il puntatore al nodo corrente viene aggiornato con quello al successivo.

Per scambiare la testa e la coda basta salvare in un registro temporaneo la coda, copiare la testa al suo posto e copiare infine il contenuto del registro temporaneo nella testa della coda. A questo punto la **DEL** è conclusa e si può uscire dalla procedura.

Funzioni ausiliarie:

address_generator

Questa procedura genera un indirizzo di memoria pseudo-casuale tra un seed fornito.

La procedura comincia effettuando 4 shift (di 1, 2^2, 2^3, 2^5) a destra del seed salvandoli in 4 registri temporanei diversi, poi questi risultati vengono messi in uno xor, il seed viene shiftato a destra di 2 e il risultato dello xor viene shiftato a sinistra di 2^15, questi due valori vengono messi in or. Per considerare solo 16 del risultato dell'or si mette in and il risultato stesso e 0x0000ffff, l'output di questa operazione viene messo in or con 0x00010000 e il risultato è l'indirizzo generato, che viene sovrascritto al seed precedente in caso di una futura chiamata della procedura.

Infine si controlla che la memoria puntata dall'indirizzo sia libera, in caso positivo la funzione termina e ritorna in output l'indirizzo generato, altrimenti richiama se stessa e ne genera uno nuovo.

swap_check

La funzione ausiliaria "**swap_check**" determina se i due elementi a, b in input dal SORT devono esse scambiati confrontando le priorità dei due valori assegnati nel tramite la funzione stessa

Inizialmente si confronta il valore ASCII di a con i limiti dei diversi intervalli d'interesse per l'ordinamento:

Se a < 65 si salta a "**check_number_first**", altrimenti si controlla a > 90, in caso si salta a "**check_minuscola_first**". Se entrambi questi controlli falliscono significa che a è una lettera maiuscola e gli viene quindi assegnata una priorità massima p(a) = 4 e si salta a "**check_second**".

Se si è saltati a "**check_number_first**" si controlla che 48 ≤ a ≤ 57, se la condizione è vera si setta la priorità p(a) = 2, si salta a "**check_second**". Invece se il controllo è fallito a sarà sicuramente un carattere speciale e si salta a "**set_special_char_first**" dove p(a) = 1, si salta a "**check_second**".

Infine se si è saltati a "**check_minuscola_first**" si controlla 97 ≤ a ≤ 122, se è vera allora p(a) = 3 e si salta a "**check_second**", altrimenti come di sopra a è un carattere speciale con p(a) = 1 saltando poi a "**check_second**". Il controllo per il secondo elemento si svolge allo stesso modo, ma quando anche p(b) è settata si salta alla label "**check_priority**" che confronta le priorità di a e b: se p(a) > p(b) si salta a "**set_swapper**", dove una variabile di output c viene settata c = 1 e si esce dalla funzione, se invece il check fallisce si controlla che p(a) == p(b), se è falso non c'è bisogno di effettuare nessuno scambio e si può uscire dalla funzione, se invece il controllo ha successo si salta a "**check_elements**" dove si controlla a > b, se è vero si setta c = 1 e si torna al rigo successivo alla chiamata jal, se in a non è più grande di b non si deve fare nessuno scambio, la funzione è conclusa e si esce dalla funzione.

check_next_instruction

Questa funzione ausiliaria viene chiamata dal **DECODING** per cercare la prossima istruzione nella listInput

La procedura inizia con un ciclo "**check_spaces**" che ignora eventuali spazi, al primo carattere salta al loop "**check_tilde**" che controlla che il carattere corrente sia una tilde, in caso salta a "**next**", altrimenti incrementa il puntatore e controlla che il carattere corrisponda al fine stringa, se è vero salta a "**exit**" e il programma finisce poiché tutta la stringa è stata decodificata. Se invece il carattere non è il fine stringa il ciclo ricomincia e va avanti finché non trova una tilde o il fine stringa. Trovata una tilde viene richiamato il DECODING per analizzare la nuova istruzione.

TEST

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

In console viene stampato:

1aaB;9
;19aaB
aa91;

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

In console viene stampato:

1aB9
1aB9
1a9
9a1