Università Politecnica delle Marche

Ingegneria Informatica e dell’Automazione

AA 2017/18



Liste Dinamiche Semplici:

Confronto tra Algoritmi Iterativi e Ricorsivi

**Svolta da:**

* Alessandro Napoletano

# Indice.

**01. Introduzione. pag. 3**

01.1 Le Liste pag. 3

01.2 Tipologie di Liste pag. 4

- Concatenata

- Doppia

- Circolare

- Ordinata

01.3 Operazioni su una Lista pag. 6

**02. Liste Dinamiche Semplici: Algoritmi Iterativi pag. 7**

02.1 Creazione/Distruzione di una Lista Ordinata pag. 7

- Codice

02.2 Visita di una Lista Ordinata pag. 8

- Codice

02.3 Ricerca di un elemento in una Lista Ordinata pag. 8

- Codice

02.3 Inserimento di un elemento in una Lista Ordinata pag. 10

- Codice

02.3 Cancellazione di un elemento in una Lista Ordinata pag. 11

- Codice

**03. Liste Dinamiche Semplici: Algoritmi Ricorsivi pag. 12**

03.2 Visita di una Lista Ordinata pag. 13

- Codice

03.3 Ricerca di un elemento in una Lista Ordinata pag. 13

- Codice

03.3 Inserimento di un elemento in una Lista Ordinata pag. 14

- Codice

03.3 Cancellazione di un elemento in una Lista Ordinata pag. 15

- Codice

**04. Confronto Codice Iterativo e Codice Ricorsivo. pag. 17**

04.1 Codice Iterativo pag. 17

- Codice

04.2 Commento dell’Output pag. 21

- Use Case

04.3 Codice Ricorsivo pag. 22

- Codice

04.4 Commento dell’Output pag. 26

- Use Case

04.5 Confronto. Pag. 27

**05. Bibliografia e Fonti pag. 28**

# 01. Introduzione

In informatica, la rappresentazione di un insieme di elementi in un programma è uno degli aspetti più ricorrenti, una struttura dati è un’entità usata per organizzare un insieme di dati all’interno della memoria del computer, ed eventualmente per memorizzarli in una memoria di massa. Vi sono differenti tipologie di strutture dati, e la scelta su quali utilizzare è strettamente legata a quella degli algoritmi; per questo, spesso essi vengono considerati insieme. Infatti, la scelta della struttura dati influisce inevitabilmente sull’efficienza degli algoritmi che la manipolano.

La principale struttura dati messa a disposizione dal Linguaggio C è l’array, tuttavia queste, per alcune applicazioni, non costituiscono una struttura dati ideale in quanto presentano due problemi:

1. Il numero degli elementi dell’array deve essere noto nel momento in cui il programma viene compilato.
2. Mentre l’accesso all’array è una semplice operazione di complessità computazionale pari a θ(1), altre operazioni come l’inserimento o l’accesso ad un elemento in base al suo contenuto possono essere realizzate con una complessità significativamente minore ricorrendo ad altri tipi di strutture.

Ricorriamo così all’utilizzo di *Strutture Dati Dinamiche* come: *Liste Dinamiche*, *Tabelle Hash* ed *Alberi*.

In questa relazione analizzeremo le Liste: le liste sono uno degli argomenti più delicati da trattare, per il semplice fatto che sono uno strumento potente e versatile, ma anche molto fragile. Basti pensare che l’uso delle liste permette di impostare programmi di ordinamento in modo molto efficiente, ed offre quella dinamicità, tra l’altro tipica del C, di cui si può avere bisogno durante lo sviluppo di un programma.

# Le Liste

La Lista (in inglese: *List*) è una struttura dati astratta e dinamica costituita da una collezione di elementi omogenei, ma, a differenza dell’array, occupa in memoria una posizione qualsiasi che può cambiare dinamicamente durante l’utilizzo della lista stessa; Inoltre, la sua dimensione non è nota a priori e può variare nel tempo.

Abbiamo detto che la lista è una struttura dati astratta e dinamica, ma che significa?

Struttura Dati Astratta: Si parla di strutture dati astratte quando si vuole distinguere le strutture stesse dalle loro implementazioni.

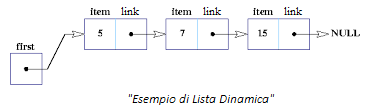
Struttura Dati Dinamica: Sono strutture basate sull’uso di dati di tipo puntatore e sull’allocazione dinamica della memoria.

Gli elementi possono essere allocati e deallocati all’occorrenza, collegati tra loro in modi diversi, e questi collegamenti possono a loro volta mutare durante l’esecuzione del programma.

La lista adotta uno schema di rappresentazione semplice anche se con evidenti limiti di efficienza computazionale. Come in ogni altra struttura dati, sono contemporaneamente rappresentate due tipologie di informazioni:

* **Informazione Utile**: L’informazione che si intende realmente rappresentare (ad esempio, il *tipo* di dati).
* **Informazione Strutturale**: L’informazione che serve a realizzare l’intera struttura, ovvero, un *campo**che punta ad un altro elemento della lista*, questo sarà NULL se non ci sono altri elementi, mentre risulta essere una struttura elemento nel caso vi sia un successore (è l’informazione aggiuntiva che serve per stabilire l’ordine con il quale gli elementi sono disposti).

Un semplice schema per rappresentare una lista consiste nell’introdurre degli elementi informativi, detti nodi, in numero pari al numero di nominativi dell’insieme, in ognuno dei quali viene rappresentata oltre all’informazione utile, anche l’informazione strutturale. Risulta quindi evidente che i collegamenti esistenti tra i vari nodi consentono di identificare un ordine lineare.



# Tipologie di Lista

Esistono diverse tipologie di liste, queste infatti si dividono in:

* Lista Concatenata
* Lista Doppia
* Lista Circolare
* Lista Ordinata

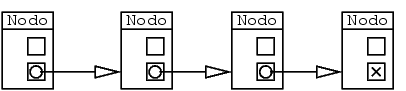
Analizziamole una ad una:

* Lista Concatenata

La Lista Concatenata (in inglese: *Linked List*) è la più semplice fra le tipologie esistenti, dallo schema notiamo che il primo elemento, la “*testa*”, è successore di nessun altro nodo della lista. Dalla testa è possibile conoscere il riferimento al nodo successore che occupa la seconda posizione, ed in maniera analoga, da questi, il terzo nodo e così via.

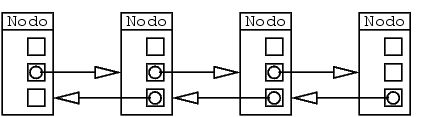
L’ultimo elemento della lista viene invece chiamato “*coda*”, ed è per definizione l’unico elemento della lista che non possiede un successore e simbolicamente si denota con “NULL”, ovvero il successore inesistente della coda.

È dunque chiaro che la lista concatenata è una struttura dati dinamica che consiste in una sequenza di nodi, ognuno dei quali contiene l’informazione utile e l’informazione strutturale.



* Lista Doppia

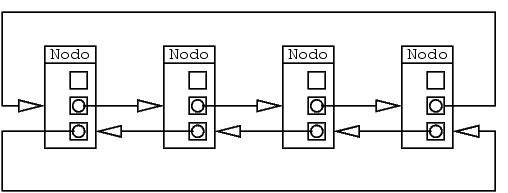
La lista doppia è una lista nella quale ogni nodo possiede, nell’informazione strutturale, due collegamenti: uno al nodo successore ed uno al predecessore. Generalmente per una lista doppia è noto sia il riferimento di testa “*first*”, che il riferimento alla coda “*last*”.



* Lista Circolare

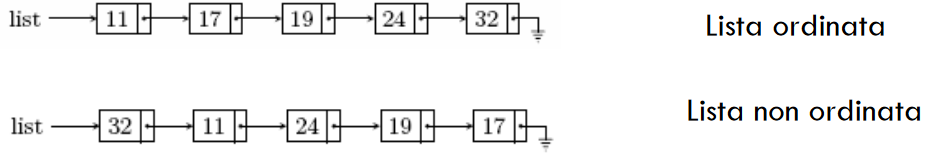
Una lista circolare è una lista in cui la coda, è collegata al primo elemento della lista, la testa. In questo caso è noto il riferimento “list” alla testa della lista.

La lista circolare può anche essere doppia e viene rappresentata nel seguente modo:



* Lista Ordinata

Una lista ordinata è caratterizzata dal fatto che l’ordine fisico e l’ordine logico coincidono, quindi il nodo con il valore minimo di informazione occuperà la prima posizione, il nodo con il valore immediatamente maggiore occuperà la seconda e così via fino alla coda.



# Operazioni su una Lista

# Le operazioni di base che si possono definire su una lista ordinata sono:

# La Creazione di una Lista: crea e restituisce una lista vuota.

# La Distruzione di una Lista: distrugge la lista, deallocando tutti gli elementi di cui si compone.

# La Visita di una Lista: visita in sequenza tutti i nodi, dalla testa alla coda. Comunemente utilizzata assieme alla stampa della stessa.

# La Ricerca di un elemento di una Lista: ricerca un elemento nella lista restituendone il riferimento.

# L’Inserimento di un elemento di una Lista: inserisce un nuovo elemento nella lista, preservandone l’ordinamento.

# La Cancellazione di un elemento di una Lista: elimina un elemento della lista, se presente, deallocando la memoria occupata.

# È spesso utile introdurre dei predicati, ovvero delle funzioni che restituiscono un valore booleano che hanno lo scopo di conoscere lo stato corrente della lista, come ad esempio:

# La Verifica se la lista sia vuota o meno: restituisce ad esempio il valore TRUE se la lista non contiene elementi.

Essendo la lista identificata dal riferimento alla sua testa, tutte le operazioni che comportano una variazione della testa, come l’inserimento o la cancellazione dell’elemento che occupa la prima posizione della lista, comportano che il riferimento “list” cambi valore; in queste circostanze è quindi necessario che le operazioni di inserimento e cancellazione restituiscano il valore aggiornato del riferimento di testa “list”.

Ne approfittiamo per rendere note ulteriori due funzioni di base definite sulle liste:

# La Creazione di un nodo della lista: crea un nodo, allocandone dinamicamente la memoria richiesta.

# La Distruzione di un nodo della lista: distrugge un nodo, deallocando la memoria occupata.

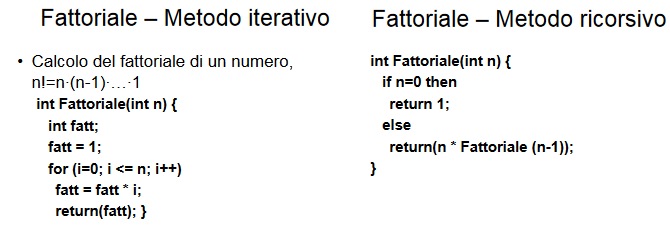
È interessante sottolineare che ogni algoritmo risolto con una tecnica iterativa, dove l’attenzione del Programmatore si concentra sull’individuazione della sequenza di azioni elementari che portano alla sua risoluzione, può essere risolto con la tecnica ricorsiva riconducendo il problema principale in problemi del medesimo tipo come richiesto dal paradigma *Divide et Impera.*

Entriamo ora nel vivo della relazione, l’analisi degli algoritmi iterativi e ricorsivi delle Liste Semplici!

Ma vediamo innanzitutto le differenze basilari tra Iterazione e Ricorsione:

L’Iterazione: si realizza quando si eseguono ripetutamente una o più righe di codice, in accordo al verificarsi di determinate condizioni.

La Ricorsione: è il processo che si verifica quando una routine richiama se stessa, di norma su porzioni di dati di volta in volta più piccole; in presenza di ricorsioni incontrollate, cioè che non prevedono una condizione di uscita, viene in genere visualizzato il messaggio di errore Spazio dello stack esaurito. Di solito, un algoritmo ricorsivo è più compatto dell’analogo iterativo, ma comporta un consumo di memoria maggiore.

Giusto per rendere la descrizione più comprensibile, qui di seguito verranno illustrati i due metodi applicati nell’Algoritmo del calcolo del Fattoriale.

# 02. Liste Dinamiche Semplici: Algoritmi Iterativi

Analizzeremo ora le funzioni di gestione di una lista precedentemente introdotte, sviluppate mediante l’utilizzo di una tecnica iterativa, in riferimento alle liste dinamiche semplici.

Le funzioni di gestione della lista usano le due funzioni di base per la creazione e la distruzione dei nodi.

/\* Crea e alloca un nodo

\* PRE: nessuna

\*/

TNode\* node\_create(TInfo value)

{

TNode \*node;

node = (TNode \*) malloc(sizeof(TNode));

// alloco un blocco di memoria di dimensione TNode ed assegno alla variabile node il puntatore a tale blocco

// malloc puo' restituire il puntatore nullo se la memoria non e' sufficiente (o se è stato richiesto un blocco di 0 bytes)

// ---> (il valore di ritorno va controllato sempre)

if (node==NULL)

{

printf("\n\n E R R O R E D I A L L O C A Z I O N E D E L L A M E M O R I A\n\n");

return NULL; // -> in tal caso la funzione termina

}

node->info = value;

node->link = NULL;

return node;

}

/\* Distrugge e dealloca un nodo

\* PRE: nessuna

\*/

void node\_destroy(TNode \*node)

{

if (node != NULL)

{

free(node);

}

}

02.1 Creazione e Distruzione di una Lista

La funzione di creazione di una lista, crea una lista vuota, restituendone il riferimento.

/\* Questa funzione crea e restituisce una lista vuota

\* PRE: nessuna

\*/

void list\_create()

{

return NULL;

}

La funzione di distruzione di una lista ha invece l’obiettivo di eliminare la lista deallocando tutti i nodi di cui si compone, richiamando la funzione node\_destroy(TNode \*node).

/\* Funzione iterativa per distruggere una lista, deallocandone tutti gli elementi

\* PRE: nessuna

\* NOTA: consuma il parametro list

\*/

TList list\_destroy (TList list) {

TNode \* curr, \*succ;

curr = list;

while (curr != NULL) {

succ = curr->link;

// la variabile succ viene utilizzata per riferirsi all’elemento successivo a quello corrente.

node\_destroy (curr);

curr = succ;

}

return NULL;

}

# Il relativo algoritmo impiega un ciclo nel quale al generico passo viene deallocato l’elemento corrente visitato. Dal codice osserviamo che, in assenza della variabile succ, impiegata per riferirsi all’elemento successivo a quello corrente, a seguito della sua deallocazione non avremmo avuto accesso al suo campo link e quindi al nodo successivo della lista.

# 02.2 Visita di una Lista

# La funzione di visita di una lista, consiste nello scorrimento sequenziale della stessa a partire dalla testa, applicando su ciascun nodo una preassegnata funzione f().

# Una delle operazioni più comuni affiancate a questo algoritmo è la stampa del valore del campo info degli elementi, o il conteggio degli stessi; nel nostro caso, possiamo osservare la procedura correlata alla funzione di stampa su schermo:

/\* Visita la lista list dalla testa alla coda stampando gli elementi

\* PRE: nessuna

\*/

void list\_print(TList list)

{

// ciclo di scansione non predeterminato:

// quando sono alla fine il puntatore list (link al successivo elemento) è NULL e il ciclo termina

while (list != NULL)

{

printf("%d", list->info); // visualizza l'informazione su schermo

printf(" ---> ");

list = list->link; // scorre di un elemento (come in list\_create punto al nodo successivo grazie al link)

}

printf("\n\n");

}

list è il riferimento che, come vediamo da codice, punta al link dell’elemento successivo. Lo scorrimento della Lista terminerà quando list si posiziona oltre l’ultimo elemento della lista, assumendo il valore NULL.

La complessità computazionale dell’algoritmo di visita si valuta nel seguente modo:

Ipotizziamo che la lista verrà interamente percorsa in ogni caso, il ciclo principale verrà attraversato n volte, con n il numero di elementi della lista. Considerando che nel ciclo vengono eseguite operazioni che richiedono una complessità θ(1), ed indicando Tw, Tb, Ta, rispettivamente la complessità nel caso peggiore, migliore ed in quello medio, si ottiene che:

Tw = Ta = Tb = n\*θ(1) = θ(n)

# Ricerca di un elemento in una Lista Ordinata

# L’obiettivo del seguente algoritmo è quello di cercare un elemento nella Lista segnalandone l’esito.

# In caso positivo, verrà restituito il riferimento al nodo che contiene il valore ricercato, in caso negativo invece verrà restituito un valore NULL ad indicare che la ricerca è stata infruttuosa.

/\* Ricerca Iterativamente l'elemento di valore info nella lista list, preservando l'ordinamento; restituisce la lista risultante.

\* PRE: list e' ordinata

\* NOTA: consuma il parametro list; inoltre se l'elemento da inserire e' gia' presente, esso viene duplicato.

\*/

TNode \*list\_search\_ordered(TList list, TInfo info)

{

/\* PRE: la lista list e' ordinata \*/

TNode \* curr;

curr = list;

//posizione di curr dopo il ciclo :

/\*P1: l'elemento da cercare ha un valore info inferiore a quello dell'elemento di testa della lista

\*P2: l'elemento da cercare ha un valore info compreso tra quello

\* della testa e quello della coda della lista (in tal caso può comunque esistere o non esistere nella lista)

\*P3: l'elemento da cercare ha un valore info maggiore di tutti quelli degli elementi della lista

\*/

// il ciclo scansiona tutti i nodi passando al successivo finché curr non è NULLO (ovvero si sono verificati i casi P1 e P3)

// e contemporaneamente il valore cercato non è maggiore di curr->info corrente (caso P2 con dato non trovato)

while ((curr != NULL) && greater(info, curr->info))

{

curr = curr->link;

}

/\* Analisi delle post-condizioni del ciclo

C1: valore da cercare piu' piccolo della Testa --> curr = NULL

C2: valore da cercare maggiore della Coda --> curr = NULL

C3: valore da cercare compreso tra quello di Testa e quello di Coda --> curr = puntatore ad un nodo

\*/

if ((curr != NULL) && equal(curr->info, info))

/\* Elemento trovato se curr non è nullo e se il puntatore trovato fa riferimento al valore info in esame \*/

return curr;

else

return NULL;

}

Nella versione iterativa, la ricerca si realizza mediante un ciclo, in cui al generico passo si confronta il valore di info dell’elemento da cercare con il valore info dell’elemento della lista correntemente analizzato, individuato da curr.

Come possiamo ben vedere, il ciclo terminerà quando l’elemento della lista ha un valore di info eguale a quello da cercare, concludendo peraltro con successo. Alternativamente, la ricerca si conclude quando si raggiunge la coda della Lista o quando l’elemento corrente ha un valore di info superiore a quello da ricercare: in tal caso, la condizione di ordinamento ci dà garanzia che gli elementi successivi avranno valori di info ancora maggiori di quello corrente, e dunque l’elemento ricercato non è presente.

Il calcolo della complessità computazionale nella ricerca di un elemento di una Lista Ordinata è particolare, in quanto bisogna tener conto che lo scorrimento può concludersi anticipatamente in alcune condizioni. Analizziamole:

* C1: l’elemento da cercare occupa la posizione di testa o ha un valore inferiore ad essa. In tal caso lo scorrimento si ferma al primo elemento e l’algoritmo termina con complessità computazionale θ(1)
* C2: l’elemento da cercare ha valore compreso tra quello di testa e quello di coda. Indipendentemente dal fatto che venga trovato o meno, lo scorrimento della lista ha complessità computazionale pari n\* θ(1) = θ(n) dove con n si intende il numero dei valori comparati dall’algoritmo.
* C3: l’elemento da cercare non esiste ed ha valore superiore a quello di coda. In questo caso, la Lista viene comunque percorsa interamente ed il ciclo principale viene percorso n volte con n il numero degli elementi della lista. Le operazioni richiedono complessità θ(1) e come nel caso precedente ci ritroveremo con un algoritmo di complessità computazionale n\* θ(1) = θ(n)

# Inserimento di un elemento in una Lista Ordinata

# L’inserimento di un elemento in una Lista Ordinata si articola in tre fasi:

# F1: Ricerca della posizione nella quale inserire il nuovo elemento;

# F2: Creazione di un nuovo nodo, destinato ad ospitare l’elemento da inserire;

# F3: Aggiornamento della catena dei collegamenti per l’inclusione del nodo creato.

# Con la prima fase si percorre la lista confrontando passo dopo passo il valore dell’elemento correntemente analizzato (riferito con curr) con il valore dell’elemento da inserire. Stiamo analizzando il caso di una lista ordinata, dunque il suo scorrimento termina quando si raggiunge un elemento il cui valore è immediatamente più grande di quello da inserire.

# Nel particolare caso in cui l’elemento da inserire è più piccolo del valore di testa, questo verrà inserito come nuovo elemento di testa della lista.

# La seconda fase consiste nell’allocazione del nuovo nodo e nell’inserimento nel relativo campo info del valore da inserire.

# Infine la terza fase comporta l’aggiornamento dei collegamenti del nodo o del collegamento del nodo nel caso l’inserimento sia in testa.

/\* Inserisce Iterativamente l'elemento di valore info nella lista list, preservando l'ordinamento; restituisce la lista risultante.

\* PRE: list e' ordinata

\* NOTA: consuma il parametro list; inoltre se l’elemento da inserire e’ gia’ presente, esso viene duplicato.

\*/

TList list\_insert\_iterative(TList list, TInfo info)

{

/\* PRE: la lista list e' ordinata \*/

TNode \* prec, \*succ, \*new;

prec = NULL;

curr = list;

/\* P1: prec==NULL e succ==NULL, la lista e’ vuota \*/

/\* P2: prec==NULL e succ!=NULL, l’elemento da inserire e’ non maggiore del primo elemento della lista \*/

/\* P3: prec!=NULL e succ==NULL, l’elemento da inserire e’ maggiore dell’ultimo elemento della lista \*/

/\* P4: prec!=NULL e succ!=NULL, l’elemento da inserire e’ maggiore del nodo di testa e minore di quello di coda. \*/

/\* F1: ricerca della posizione di inserimento \*/

while ((succ != NULL) && greater(info, succ->info))

{

curr = succ;

succ = succ->link;

}

/\* F2: allocazione del nuovo nodo \*/

new = node\_create(info);

if (new == NULL) {

printf("\n\n E R R O R E D I A L L O C A Z I O N E D E L L A M E M O R I A\n\n");

exit (1);

}

/\* F3: aggiornamento della catena dei collegamenti \*/

if (prec == NULL) {

/\*C1: inserimento in Testa \*/

new->link = list;

return new;

} else {

/\* C2: inserimento in posizione centrale o in coda \*/

new->link = succ;

prec->link = new;

return list;

}

}

# Cancellazione di un elemento in una Lista Ordinata

# La cancellazione di un elemento in una Lista Ordinata si articola in tre fasi:

# F1: Ricerca dell’elemento da cancellare;

# F2: Aggiornamento della catena dei collegamenti.

# F3: Deallocazione dell’elemento da cancellare;

# Durante la fase F1 di ricerca dell’elemento, si percorre la lista dal primo elemento fino a trovare l’elemento da cancellare; se durante la ricerca viene percorsa tutta la lista o si è raggiunto un elemento il cui valore è maggiore al valore dell’elemento da cancellare, avremo la certezza che quell’elemento non è presente nella lista.

# La fase F2 invece, viene detta *cancellazione logica*, trovato l’elemento da cancellare, viene aggiornata l’informazione strutturale degli elementi della lista, e l’elemento ricercato non sarà più accessibile in alcun modo. Tuttavia il valore rimarrà in memoria finché l’algoritmo non entrerà nella fase F3, dove il nodo eliminato (puntato mediante “alias”), verrà deallocato definitivamente.

/\* Cancella Iterativamente un elemento di valore info nella lista list, preservando l'ordinamento; restituisce la lista risultante.

\* PRE: list e' ordinata

\* NOTA: consuma il parametro list; inoltre se l’elemento da inserire e’ duplicato cancella la prima occorrenza.

\*/

TList list\_delete(TList list, TInfo info)

{

TNode \*prec, \*curr, \*alias;

prec = NULL;

curr = list;

/\*P1: curr==NULL o info!=curr->info, l’elemento da cancellare non è presente

/\*P2: la condizione P1 non è verificata, e prec==NULL va cancellato l’elemento di testa

/\*P3: la condizione P1 non è verificata, e prec!=NULL e curr!=NULL va cancellato un elemento in posizione

/\* intermedia della lista \*/

/\* F1: ricerca dell’elemento da cancellare \*/

while ((curr != NULL) && greater(info, curr->info))

{

prec = curr;

curr = curr->link;

}

/\* Analisi delle post-conduzioni \*/

if ((curr != NULL) && equal(curr->info, info))

{

/\* Elemento trovato \*/

/\* F2: aggiornamento della catena dei collegamenti \*/

if (prec == NULL)

{

/\* CASO C2: Cancellazione della Testa \*/

list = curr->link;

}

else

{

/\* CASO C3: Cancellazione da una posizione intermedia \*/

alias = curr->link;

prec->link = alias;

}

/\*F3: Deallocazione del Nodo cancellato logicamente \*/

node\_destroy(curr);

return list;

}

return list;

}

# 03. Liste Dinamiche Semplici: Algoritmi Ricorsivi

Diversamente dall’implementazione Iterativa, la realizzazione ricorsiva delle funzioni di visita, inserimento, ricerca e cancellazione di un elemento in una lista ordinata si ottengono mediante l’approccio del metodo del Divide et Impera (in inglese: Divide and Conquer).

La tecnica del Divide et Impera è un Paradigma che consiste nel risolvere un problema mediante un’accorta suddivisione di esso in vari sotto problemi, un programma sviluppato secondo questa tecnica è sostanzialmente diviso in tre parti:

* *Divide*: in questa parte si procede all’individuazione ed alla suddivisione del problema originario in problemi di dimensione minore che operano su strutture dati di dimensioni più piccole;
* *Impera*: nella seconda parte i problemi vengono risolti in modo ricorsivo. Quando i sotto problemi arrivano ad avere una dimensione sufficientemente piccola, essi vengono risolti direttamente tramite il caso base;
* *Combina*: l'ultima fase del paradigma prevede di ricombinare l'output ottenuto dalle precedenti chiamate ricorsive al fine di ottenere il risultato finale.



Il codice riguardante le funzioni inerenti alla creazione e distruzione dei nodi sono le stesse già indicate all’inizio del Capitolo 2; la stessa cosa vale per le funzioni di creazione e distruzione della lista, esposte nel Paragrafo 2.1

# 03.1 Visita di una Lista

# Vediamo come viene elaborata la programmazione con tecnica ricorsiva mediante il paradigma Divide et Impera:

* *Divide*: la divisione del problema deve essere realizzata in modo tale da raggiungere il caso base. Questo obiettivo è conseguito riducendo ricorsivamente la lista l alla lista l’ ottenuta eliminando da l il suo elemento di testa.
* *Caso Base*: il caso base è quello che si riferisce alla stampa di una lista vuota, in questo caso non c’è bisogno di effettuare alcuna operazione.
* *Impera*: si ipotizza che il problema figlio fornisca il risultato corretto nel visitare l’.
* *Combina*: la soluzione al problema padre è ottenuta visitando il primo elemento della lista l e poi visitando la lista l’, invocando quindi la soluzione del problema figlio, corretta l’ipotesi della fase precedente.

// Visita la lista list dalla testa alla coda stampando gli elementi

// PRE: nessuna

void list\_print\_recursive(TList list)

{

if (list != NULL) // non stampo se la lista è vuota o se sono arrivato alla coda (list->link punta a NULL)

{

printf("%d", list->info); // visualizza l'informazione su schermo

printf(" ---> ");

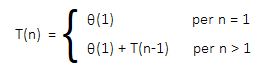
list\_print\_recursive(list->link); // chiamata ricorsiva alla funzione passando il puntamento all'elemento successivo

}

}

È importante evidenziare che nel caso degenere, ovvero in presenza della lista vuota ϕ la visita non deve effettuare alcuna operazione; questo caso è quindi assimilabile al caso base.

La formula di ricorrenza, identica per il caso migliore, peggiore e medio tra loro coincidenti è data da:

  
Dunque Tw = Ta = Tb = θ(n)

03.2 Ricerca di un elemento in una Lista Ordinata

I passi per la realizzazione ricorsiva della ricerca sono riassunti nel seguito:

* *Divide*: la divisione del problema deve essere realizzata in modo tale da raggiungere il caso base. Questo obiettivo è conseguito riducendo ricorsivamente la lista l alla lista l’ ottenuta eliminando da l il suo elemento di testa. Nei casi banali, il problema della ricerca di un elemento in l è decomposto nel più semplice problema della ricerca dell’elemento l’.
* *Caso Base*: il problema della ricerca è banale quando la lista è vuota, quando il primo elemento della lista è più grande di quello da cancellare o ancora quando l’elemento ricercato è proprio il primo elemento della lista. Nei primi due casi la ricerca può terminare in quanto l’elemento ricercato certamente non esiste. Anche nel terzo caso la ricerca termina, ma sta volta con successo.
* *Impera*: Si ritiene, per ipotesi induttiva, che il problema figlio consistente nella ricerca dell’elemento in l’, fornisca il risultato corretto.
* *Combina*: la soluzione al problema padre è ottenuta verificando che il primo elemento della lista l non sia quello cercato, in caso negativo, se l’elemento è presente in l’ lo sarà anche in l e quindi la soluzione del problema padre coincide con quella del problema figlio.

/\* Cerca l'elemento di valore info nella Lista list. Ritorna il riferimento all'elemento se e' presente, altrimenti ritorna NULL.

\* PRE: list e' ordinata

\*/

TNode \*list\_search\_recursive(TList list, TInfo info)

{

if (list == NULL || greater(list->info, info))

{

// casi base :

// 1) la lista è vuota, oppure

// 2) il primo elemento della lista è + grande di quello cercato

return NULL;

}

else

{

// casi ricorsivi:

if (equal(list->info, info))

{

// 3) l'elemento cercato è proprio il primo della lista

return list;

}

else

{

// 4) l'elemento cercato e` uno dei successivi

return list\_search\_recursive(list->link, info);

}

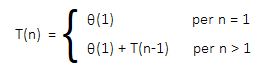
}

}

La complessità computazionale, nel caso migliore, che si verifica quando l’elemento da cercare è al primo posto, è semplicemente data da:

Tb = θ(1)

Viceversa, nel caso medio e peggiore si ottiene mediante la seguente formula di ricorrenza:

La cui soluzione è riportata da:

Tw=Ta= θ(n)

03.3 Inserimento di un elemento in una Lista Ordinata

I passi per la realizzazione ricorsiva della ricerca sono riassunti nel seguito:

* *Divide*: la divisione del problema è realizzata come nelle funzioni precedentemente illustrate.
* *Caso Base*: Premettiamo che il problema dell’inserimento di un elemento in una lista ordinata è banale quando la lista è vuota o quando il primo elemento della lista è maggiore dell’elemento da inserire; in queste circostanze l’inserimento va effettuato in testa alla lista, allocando opportunamente un nuovo nodo, che avrà come successore la lista iniziale.
* *Impera*: Per ipotesi induttiva il problema dell’inserimento dell’elemento in l’ è risolto correttamente.
* *Combina:* Se non ci troviamo nei casi base, vuol dire che l’elemento di testa di l, se esiste, è più piccolo dell’elemento da inserire e quindi la soluzione del problema padre è ricondotta all’inserimento nella lista l’; quest’ultimo problema, per ipotesi induttiva, è risolto correttamente. Bisogna osservare che l’inserimento in l’ può aver cambiato il puntatore di testa della lista, e pertanto la soluzione al problema padre consiste semplicemente nell’aggiornamento del campo link dell’elemento di testa di l.

/\* Inserisce l’elemento di valore info nella lista list, preservando

\* l’ordinamento; restituisce la lista risultante.

\* PRE: list è ordinate

\* NOTA: consuma il parametro list; inoltre se l’elemento da inserire è già presente, esso viene duplicato.

\*/

TList list\_insert\_recursive (TList list, TInfo info)

{

If ((list == NULL) || (greater(list->info, info)))

{

TNode\* newnode;

newnode = node\_create(info);

if (newnode == NULL)

{

printf (“ERRORE DI ALLOCAZIONE DELLA MEMORIA\n”);

exit(1);

}

newnode->link = list

return newnode;

}

else

{

TList l2;

l2 = list\_insert\_recursive(list->link, info);

list->link = l2;

return list;

}

}

La complessità computazionale di questa funzione è identica a quella di ricerca, sia nel caso migliore che medio e peggiore.

Tb = θ(1)

Tw = Ta = θ(n)

03.4 Cancellazione di un elemento in una Lista Ordinata

* *Divide*: la divisione del problema è realizzata come nelle funzioni precedentemente illustrate. Quest’obiettivo è conseguito riducendo ricorsivamente la list l alla lista l’ ottenuta eliminando il suo elemento di testa. Quindi, nei casi non banali, la cancellazione in l è decomposto nel più semplice problema della cancellazione l’.
* *Caso Base*: Il problema della cancellazione è banale quando la lista è vuota, oppure quando il primo elemento della lista è più grande di quello da cancellare o ancora quando l’elemento da cancellare è proprio il primo elemento della lista. Nei primi due casi la cancellazione non può essere effettuata in quanto l’elemento da cancellare non esiste, e di conseguenza la lista rimane inalterata. Nel terzo caso, invece la cancellazione viene effettuata in testa, prima logicamente e poi fisicamente.
* *Impera*: per ipotesi induttiva, il problema della cancellazione in l’ è risolto correttamente e sulla base della soluzione corretta a tale problema si ricava la soluzione al problema della cancellazione dell’elemento della nella lista l.
* *Combina:* Se non ci troviamo nei casi base, vuol dire che l’elemento di testa di l, se esiste, è più piccolo dell’elemento da cancellare e quindi la soluzione del problema padre è ricondotta alla cancellazione nella lista l’. Quest’ultimo problema è risolto correttamente nella fase di impera; l’elemento da cancellare, se esiste, è cancellato correttamente in l’. Bisogna solo osservare che la cancellazione in l’ può aver cambiato il puntatore di testa ad l’, e pertanto la soluzione al problema padre consiste semplicemente nell’aggiornamento del campo link dell’elemento di testa di l con il nuovo valore del puntatore a l’. La lista risultante della cancellazione è ovviamente l.

/\* Cancella un elemento di valore info dalla lista list, preservando l'ordinamento

\* ed i collegamenti tra nodi; restituisce la lista risultante.

\* PRE: list e' ordinata

\* NOTA: consuma il parametro list; se l'elemento da cancellare non e' presente, la lista resta inalterata.

\*/

TList list\_delete\_recursive(TList list, TInfo info)

{

if (list == NULL || greater(list->info, info))

{

// casi base non cancellazione:

// 1) lista vuota, oppure

// 2) primo elemento della lista piu`grande del valore cercato

// in entrambi i casi non devo effettuare cancellazioni (ritorno list)

return list;

}

else

{

// casi base cancellazione:

if (equal(list->info, info))

{

// 3) cancellazione dell'elemento di testa (prima logica e poi fisica)

TNode \*alias = list->link;

node\_destroy(list); // distruggo solamente il nodo corrente; i successivi sono ora "agganciati" ad alias

return alias;

}

else

{

// casi ricorsivi di cancellazione:

TList l2;

l2 = list\_delete\_recursive(list->link, info);

list->link = l2;

return list;

}

}

}

# La complessità computazionale ha la stessa formula ricorrente vista nell’inserimento:

# 

# E quindi anche in questo caso, il caso peggiore ha complessità: Tw = θ(n)

04. Confronto Codice Iterativo e Codice Ricorsivo

04.1 Codice Iterativo :

/\*

Il codice convergerà nella creazione di una lista ordinata con un ammontare massimo di elementi scelti dall'utente,

inseriti poi in modo randomico dalla macchina stessa.

Verranno applicate alla lista operazioni come:

1° Creazione della lista,

2° Inserimento di elementi,

3° Vista della lista e stampa di essa

4° Ricerca di elementi,

5° Cancellazione di un elemento

In questo programma, ciò avverrà prima in maniera ITERATIVA.

Il progetto convergerà nella trascrizione del codice seguito da un confronto tra le due metodologie,

con le relative funzioni precedentemente illustrate.

\*/

/\* //////////////////// \\\\\\\\\\\\\\\\\\\\

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ##### INIZIO CODICE ##### \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\\\\\\\\\\\\\\\\\\\\ //////////////////// \*/

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <ctype.h>

#include <time.h>

/\* ########## DEFINIZIONE STRUTTURA ########## \*/

typedef int TInfo;

struct SNode

{

TInfo info; // informazione

struct SNode \*link; // puntatore al successivo nodo

};

typedef struct SNode TNode; // elemento della lista

typedef TNode \*TList; // puntatore alla testa della lista

/\* ########## ELENCO FUNZIONI ########## \*/

TNode\* node\_create(TInfo value)

{

TNode \*node;

node = (TNode \*) malloc(sizeof(TNode)); // alloco un blocco di memoria di dimensione TNode ed assegno alla variabile new il puntatore a tale blocco

// malloc puo' restituire il puntatore nullo se la memoria non e' sufficiente (o se è stato richiesto un blocco di 0 bytes)

// ---> (il valore di ritorno va controllato sempre)

if (node==NULL)

{

printf("\n\n E R R O R E D I A L L O C A Z I O N E D E L L A M E M O R I A\n\n");

return NULL; // -> in tal caso la funzione termina

}

node->info = value;

node->link = NULL;

return node;

}

TList list\_create()

{

TNode \*new, \*punt; // new è il puntatore al primo elemento della lista, mentre punt e' un puntatore ausiliario che permette di

// scorrere la lista in modo sequenziale grazie ai collegamenti o link

int i, n; // i è l'indice del ciclo, mentre n serve a memorizzare il numero degli elementi che si intende inserire

int value; // memorizza input di utente

printf("Da quanti elementi deve essere composta la lista?\n");

scanf("%d", & n); // chiedo il numero n di elementi della lista (da tastiera)

srand(time(NULL)); // imposta la funzione random a time(NULL) per non avere sempre la stessa sequenza random

if(n<=0)

{

new = NULL; // lista vuota

}

else

{

/\* creazione primo elemento --- inserimento in TESTA \*/

// alloco un blocco di memoria di dimensione TNode ed assegno alla variabile new il puntatore a tale blocco

value = rand() % 1500 +1;

punt = node\_create(value);

new = punt; // inizializzo il puntatore ausiliario al primo nodo (TESTA)

/\* creazione elementi successivi --- inserimento in posizione centrale o in CODA \*/

for(i=2; i<=n; i++)

{

value = value + rand() % 1500 +1;

punt->link = node\_create(value);

punt = punt->link; // il puntatore ausiliario viene fatto puntare al successivo nodo (l'oggetto da lui puntato) grazie al link

}

punt->link = NULL; // marcatore fine lista -> il link dell'ultimo elemento (CODA) punta a NULL

}

return new;

}

bool equal(TInfo a, TInfo b) // a e b sono equivalenti

{

return a == b;

}

bool greater(TInfo a, TInfo b) // a segue (è più grande di) b

{

return a > b;

}

/\* ################## --- PROCEDURE ITERATIVE --- #################### \*/

/\* Visita la lista list dalla testa alla coda stampando gli elementi

\* PRE: nessuna

\*/

void list\_print(TList list)

{

// ciclo di scansione non predeterminato:

// quando sono alla fine il puntatore list (link al successivo elemento) è NULL e il ciclo termina

while (list != NULL)

{

printf("%d", list->info); // visualizza l'informazione su schermo

printf(" ---> ");

list = list->link; // scorre di un elemento (come in list\_create punto al nodo successivo grazie al link)

}

printf("\n1\n");

}

/\* Ricerca Iterativamente l'elemento di valore info nella lista list, preservando

\* l'ordinamento; restituisce la lista risultante.

\* PRE: list e' ordinata

\*/

TNode \*list\_search\_iterative(TList list, TInfo info)

{

/\* PRE: la lista list e' ordinata \*/

TNode \* curr;

curr = list;

//posizione di curr dopo il ciclo :

/\*P1: l'elemento da cercare ha un valore info inferiore a quello

\* dell'elemento di testa della lista

\*P2: l'elemento da cercare ha un valore info compreso tra quello

\* della testa e quello della coda della lista (in tal caso può comunque esistere o non esistere nella lista)

\*P3: l'elemento da cercare ha un valore info maggiore di tutti

\* quelli degli elementi della lista

\*/

// il ciclo scansiona tutti i nodi passando al successivo finché curr non è NULLO (ovvero si sono verificati i casi P1 e P3)

// e contemporaneamente il valore cercato non è maggiore di curr->info corrente (caso P2 con dato non trovato)

while ((curr != NULL) && greater(info, curr->info))

{

curr = curr->link;

}

/\* Analisi delle post-condizioni del ciclo

C1: valore da cercare piu' piccolo della Testa --> curr = NULL

C2: valore da cercare maggiore della Coda --> curr = NULL

C3: valore da cercare compreso tra quello di Testa e quello di

Coda --> curr = puntatore ad un nodo

\*/

if ((curr != NULL) && equal(curr->info, info))

/\* Elemento trovato se curr non è nullo e se il puntatore trovato fa riferimento al valore info in esame \*/

return curr;

else

return NULL;

}

/\* Cancella un elemento di valore info dalla lista list, preservando l'ordinamento

\* ed i collegamenti tra nodi; restituisce la lista risultante.

\* PRE: list e' ordinata

\* NOTA: consuma il parametro list; inoltre se l'elemento da cancellare

\* e' duplicato cancella solo la prima occorrenza.\*/

TList list\_delete(TList list, TInfo info)

{

TNode \*prec, \*curr;

prec = NULL;

curr = list;

// postcondizioni all'uscita del ciclo :

/\* P1: curr==NULL o info!=curr->info

\* --> l'elemento da cancellare non e' presente; lascia la lista inalterata

\* P2: la condizione P1 non e' verificata (l'elemento esiste!) e prec==NULL

\* --> va cancellato l'elemento di testa

\* P3: la condizione P1 non e' verificata (l'elemento esiste!), prec!=NULL e curr!=NULL

\* --> va cancellato un elemento in posizione intermedia o di coda \*/

/\* F1: ricerca dell'elemento da cancellare \*/

while ((curr != NULL) && greater(info, curr->info))

{

prec = curr; // mantieni un riferimento all'elemento precedente (prec)

curr = curr->link;

}

/\* Analisi delle post-condizioni \*/

if ((curr != NULL) && equal(curr->info, info))

{

/\* Elemento trovato \*/

/\* F2: aggiornamento della catena dei collegamenti \*/

if (prec == NULL)

{

/\* CASO C2: Cancellazione della Testa

---> il nuovo elemento della testa sarà il successore del nodo da cancellare \*/

list = curr->link;

}

else

{

/\* CASO C3: Cancellazione di nodo intermedio o di coda

---> il nodo precedente a quello da cancellare va collegato al successore del nodo da cancellare\*/

prec->link = curr->link;

}

/\* F3: Deallocazione del Nodo cancellato logicamente = cancellazione fisica \*/

node\_destroy(curr);

}

return list;

}

/\* Distrugge e dealloca un nodo

\* PRE: nessuna

\*/

void node\_destroy(TNode \*node)

{

if (node != NULL)

{

free(node);

}

}

/\* //////////////////// \\\\\\\\\\\\\\\\\\\\

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ##### MAIN CODE ##### \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\\\\\\\\\\\\\\\\\\\\ //////////////////// \*/

/\*

All'interno del main mettere un menù che favorisca un interazione Uomo/Macchina

Niente interfaccia grafica, solo terminale, ma con interazione mediante l'utilizzo della tastiera.

Ciò permetterà di testare l'algoritmo con un numero di elementi a scelta dall'utente,

Renderà il programma più facile da utilizzare, più chiaro all'utente e per finire, esteticamente migliore

\*/

int main()

{

TList lista; // definisce la lista

printf("Benvenuto! Questo programma effettuerà operazioni sulle Liste in Maniera Iterativa");

printf("\nPer iniziare, creeremo ora la lista.\n");

lista = list\_create();

printf("\*\*\*\*\* \*\*\*\*\*\n");

printf(" Menù\n\n");

printf("Scegliere un operazione da svolgere:\n1.Visita e Stampa della Lista\n2.Ricerca di un Elemento\n3.Cancellazione di un Elemento della Lista\n");

printf("\nDigita una delle opzioni:\t");

int risp;

scanf("%d", &risp);

switch (risp)

{

// Visita e Stampa della Lista

case 1:

printf("\nEcco la stampa della Lista:\n");

list\_print(lista);

break;

// Ricerca di un Elemento

case 2:

printf("\nEcco la stampa della Lista:\n");

list\_print(lista);

printf("\nIndicare l'elemento da ricercare:\n");

int search;

scanf("%d", &search);

if (list\_search\_iterative(lista, search) != NULL)

{

printf("\nElemento trovato!\n");

}

else

{

printf("\nElemento NON trovato!\n");

}

break;

// Distruzione della Lista

case 3:

printf("\nEcco la stampa della Lista:\n");

list\_print(lista);

printf("\nIndicare l'elemento da eliminare:\n");

int erase;

scanf("%d", &erase);

list\_delete(lista, erase);

printf("\nEcco la nuova lista:\n");

list\_print(lista);

break;

// Messaggio comando sbagliato

default:

printf("Errore.\nOpzione Inesistente\n");

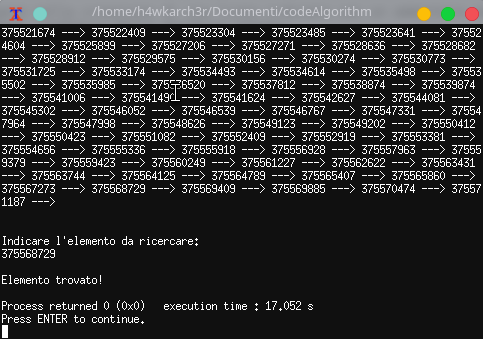
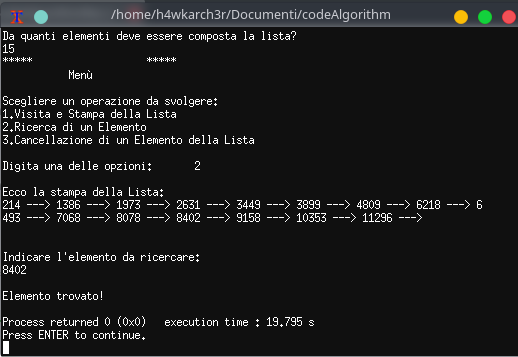
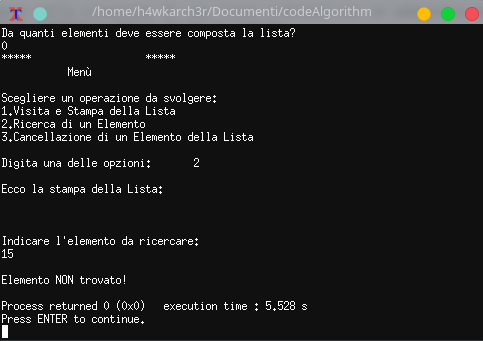
}

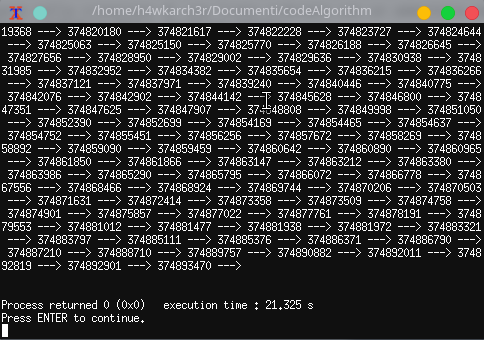
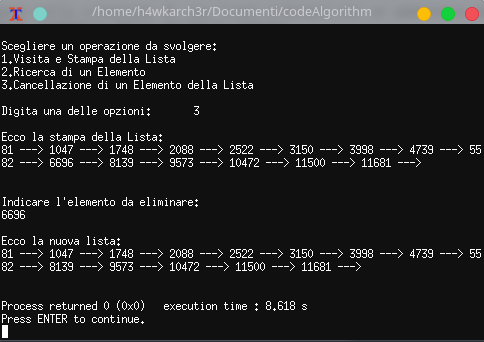
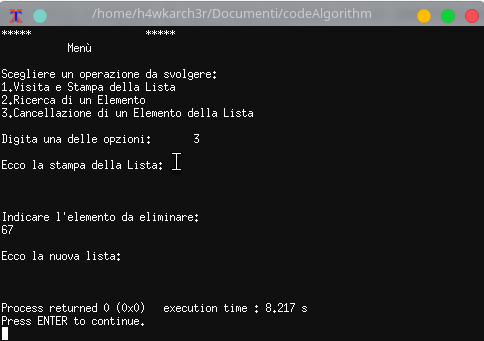
}

04.2 Commenti dell’Output

# Visita:

# 

Ricerca:

Cancellazione:

Come annunciato durante le singole spiegazioni del codice iterativo del capitolo 2 e considerando che abbiamo lavorato quasi sempre di situazioni di “worst” case, notiamo che effettivamente il tempo di esecuzione dipende prevalentemente dal numero di elementi inseriti, nel secondo screenshot inerente alla ricerca, il tempo di esecuzione maggiore nella lista composta da 15 elementi rispetto a quella di 500000 elementi è dovuto semplicemente al tempo di attesa prima della digitazione da tastiera dell’elemento da ricercare.

04.3 Codice Ricorsivo:

/\*

Il codice convergerà nella creazione di una lista ordinata con un ammontare massimo di elementi scelti dall'utente,

inseriti poi in modo randomico dalla macchina stessa.

Verranno applicate alla lista operazioni come:

1° Creazione della lista,

2° Inserimento di elementi,

3° Vista della lista e stampa di essa

4° Ricerca di elementi,

5° Cancellazione di un elemento

In questo programma, ciò avverrà prima in maniera RICORSIVA.

Il progetto convergerà nella trascrizione del codice seguito da un confronto tra le due metodologie,

con le relative funzioni precedentemente illustrate.

\*/

/\* //////////////////// \\\\\\\\\\\\\\\\\\\\

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ##### INIZIO CODICE ##### \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\\\\\\\\\\\\\\\\\\\\ //////////////////// \*/

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <ctype.h>

#include <time.h>

/\* ########## DEFINIZIONE STRUTTURA ########## \*/

typedef int TInfo;

struct SNode

{

TInfo info; // informazione

struct SNode \*link; // puntatore al successivo nodo

};

typedef struct SNode TNode; // elemento della lista

typedef TNode \*TList; // puntatore alla testa della lista

/\* ########## ELENCO FUNZIONI ########## \*/

TNode\* node\_create(TInfo value)

{

TNode \*node;

node = (TNode \*) malloc(sizeof(TNode)); // alloco un blocco di memoria di dimensione TNode ed assegno alla variabile new il puntatore a tale blocco

// malloc puo' restituire il puntatore nullo se la memoria non e' sufficiente (o se è stato richiesto un blocco di 0 bytes)

// ---> (il valore di ritorno va controllato sempre)

if (node==NULL)

{

printf("\n\n E R R O R E D I A L L O C A Z I O N E D E L L A M E M O R I A\n\n");

return NULL; // -> in tal caso la funzione termina

}

node->info = value;

node->link = NULL;

return node;

}

TList list\_create()

{

TNode \*new, \*punt; // new è il puntatore al primo elemento della lista, mentre punt e' un puntatore ausiliario che permette di

// scorrere la lista in modo sequenziale grazie ai collegamenti o link

int i, n; // i è l'indice del ciclo, mentre n serve a memorizzare il numero degli elementi che si intende inserire

int value; // memorizza input di utente

printf("Da quanti elementi deve essere composta la lista?\n");

scanf("%d", & n); // chiedo il numero n di elementi della lista (da tastiera)

srand(time(NULL)); // imposta la funzione random a time(NULL) per non avere sempre la stessa sequenza random

if(n<=0)

{

new = NULL; // lista vuota

}

else

{

/\* creazione primo elemento --- inserimento in TESTA \*/

// alloco un blocco di memoria di dimensione TNode ed assegno alla variabile new il puntatore a tale blocco

value = rand() % 1500 +1;

punt = node\_create(value);

new = punt; // inizializzo il puntatore ausiliario al primo nodo (TESTA)

/\* creazione elementi successivi --- inserimento in posizione centrale o in CODA \*/

for(i=2; i<=n; i++)

{

value = value + rand() % 1500 +1;

punt->link = node\_create(value);

punt = punt->link; // il puntatore ausiliario viene fatto puntare al successivo nodo (l'oggetto da lui puntato) grazie al link

}

punt->link = NULL; // marcatore fine lista -> il link dell'ultimo elemento (CODA) punta a NULL

}

return new;

}

bool equal(TInfo a, TInfo b) // a e b sono equivalenti

{

return a == b;

}

bool greater(TInfo a, TInfo b) // a segue (è più grande di) b

{

return a > b;

}

/\* ################## --- PROCEDURE RICORSIVE --- #################### \*/

// Visita la lista list dalla testa alla coda stampando gli elementi

// PRE: nessuna

void list\_print\_recursive(TList list)

{

if (list != NULL) // non stampo se la lista è vuota o se sono arrivato alla coda (list->link punta a NULL)

{

printf("%d", list->info); // visualizza l'informazione su schermo

printf(" ---> ");

list\_print\_recursive(list->link); // chiamata ricorsiva alla funzione passando il puntamento all'elemento successivo

}

}

/\* Cerca l'elemento di valore info nella Lista list. Ritorna il

\* riferimento all'elemento se e' presente, altrimenti ritorna NULL.

\* PRE: list e' ordinata

\*/

TNode \*list\_search\_recursive(TList list, TInfo info)

{

if (list == NULL || greater(list->info, info))

{

// casi base :

// 1) la lista è vuota, oppure

// 2) il primo elemento della lista è + grande di quello cercato

return NULL;

}

else

{

// casi ricorsivi:

if (equal(list->info, info))

{

// 3) l'elemento cercato è proprio il primo della lista

return list;

}

else

{

// 4) l'elemento cercato e` uno dei successivi

return list\_search\_recursive(list->link, info);

}

}

}

/\* Cancella un elemento di valore info dalla lista list, preservando l'ordinamento

\* ed i collegamenti tra nodi; restituisce la lista risultante.

\* PRE: list e' ordinata

\* NOTA: consuma il parametro list; se l'elemento da cancellare non e' presente, la lista resta inalterata.

\*/

TList list\_delete\_recursive(TList list, TInfo info)

{

if (list == NULL || greater(list->info, info))

{

// casi base non cancellazione:

// 1) lista vuota, oppure

// 2) primo elemento della lista piu` grande del valore cercato

// in entrambi i casi non devo effettuare cancellazioni (ritorno list)

return list;

}

else

{

// casi base cancellazione:

if (equal(list->info, info))

{

// 3) cancellazione dell'elemento di testa (prima logica e poi fisica)

TNode \*alias = list->link;

node\_destroy(list); // distruggo solamente il nodo corrente; i successivi sono ora "agganciati" ad alias

return alias;

}

else

{

// casi ricorsivi di cancellazione:

TList l2;

l2 = list\_delete\_recursive(list->link, info);

list->link = l2;

return list;

}

}

}

/\* Distrugge e dealloca un nodo

\* PRE: nessuna

\*/

void node\_destroy(TNode \*node)

{

if (node != NULL) {

free(node);

}

}

/\* //////////////////// \\\\\\\\\\\\\\\\\\\\

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ##### MAIN CODE ##### \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\\\\\\\\\\\\\\\\\\\\ //////////////////// \*/

/\*

All'interno del main mettere un menù che favorisca un interazione Uomo/Macchina

Niente interfaccia grafica, solo terminale, ma con interazione mediante l'utilizzo della tastiera.

Ciò permetterà di testare l'algoritmo con un numero di elementi a scelta dall'utente,

Renderà il programma più facile da utilizzare, più chiaro all'utente e per finire, esteticamente migliore

\*/

int main()

{

TList lista; // definisce la lista

printf("Benvenuto! Questo programma effettuerà operazioni sulle Liste in Maniera Ricorsiva");

printf("\nPer iniziare, creeremo ora la lista.\n");

lista = list\_create();

printf("\*\*\*\*\* \*\*\*\*\*\n");

printf(" Menù\n\n");

printf("Scegliere un operazione da svolgere:\n1.Visita e Stampa della Lista\n2.Ricerca di un Elemento\n3.Cancellazione di un Elemento della Lista\n");

printf("\nDigita una delle opzioni:\t");

int risp;

scanf("%d", &risp);

switch (risp)

{

// Visita e Stampa della Lista

case 1:

printf("\nEcco la stampa della Lista:\n");

list\_print\_recursive(lista);

break;

// Ricerca di un Elemento

case 2:

printf("\nEcco la stampa della Lista:\n");

list\_print\_recursive(lista);

printf("\nIndicare l'elemento da ricercare:\n");

int search;

scanf("%d", &search);

if (list\_search\_recursive(lista, search) != NULL)

{

printf("\nElemento trovato!\n");

}

else

{

printf("\nElemento NON trovato!\n");

}

break;

// Distruzione della Lista

case 3:

printf("\nEcco la stampa della Lista:\n");

list\_print\_recursive(lista);

printf("\nIndicare l'elemento da eliminare:\n");

int erase;

scanf("%d", &erase);

list\_delete\_recursive(lista, erase);

printf("\nEcco la nuova lista:\n");

list\_print\_recursive(lista);

break;

// Messaggio comando sbagliato

default:

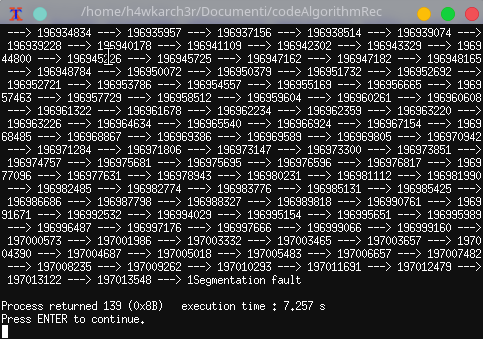
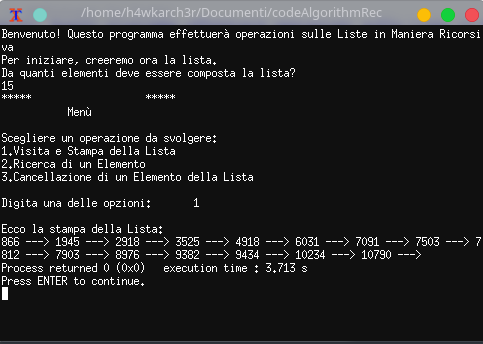
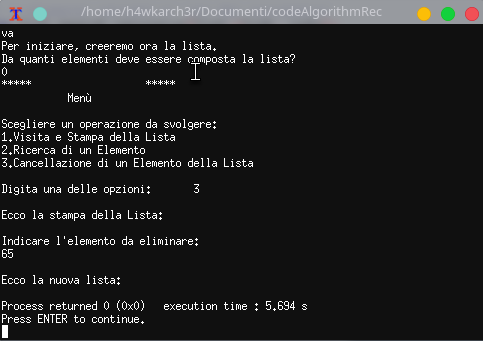
printf("Errore.\nOpzione Inesistente\n");

}

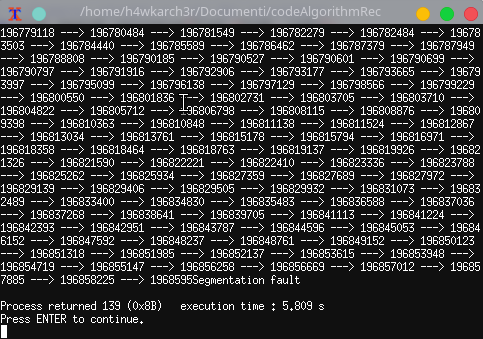
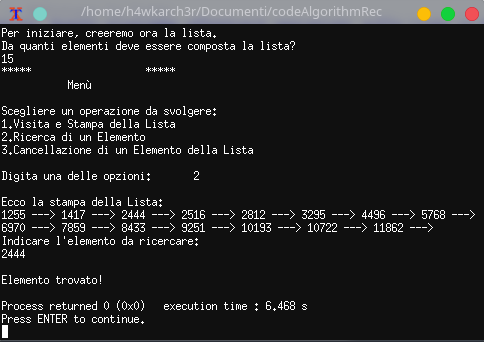
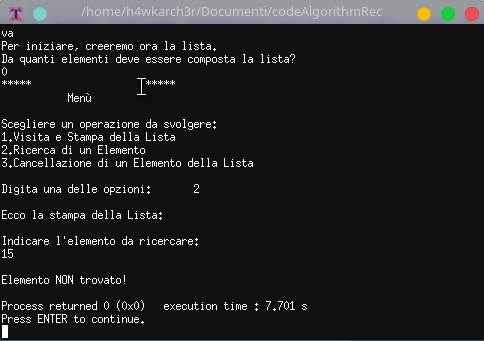
}

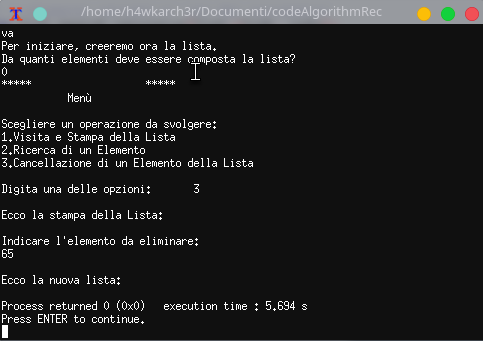
04.4 Commenti dell’Output

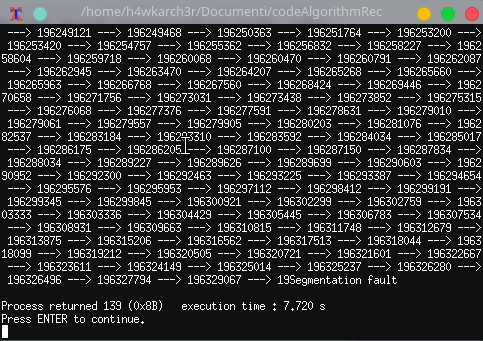
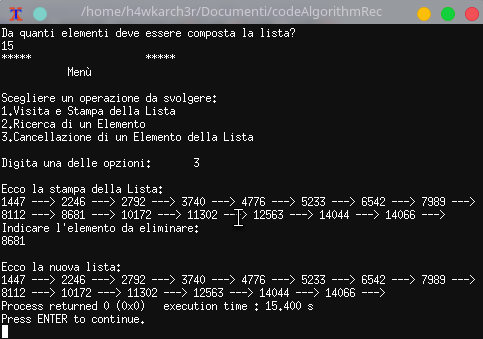
Visita:



Ricerca:

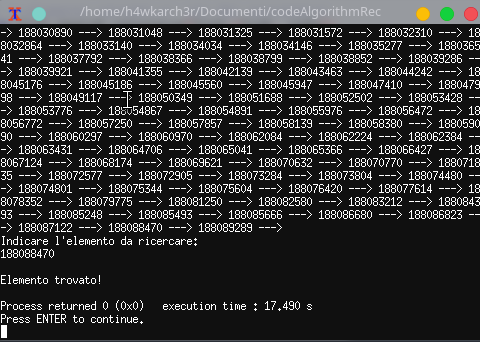


Cancellazione:



Dagli screenshot degli output, notiamo che i tempi di esecuzione sono del tutto paragonabili a quelli dei medesimi algoritmi di tipologia iterativa, tuttavia, durante l’esecuzione del programma riscontriamo un errore di Segmentation Fault ogni volta ci ritroviamo a lavorare con una lista di 500000 elementi.

Questo avviene perché, come ben sappiamo, gli algoritmi ricorsivi occupano più memoria delle controparti iterative, nel momento in cui il codice scrive una quantità di dati eccessiva, si finisce per sporcare lo stack.

È stata eseguita una prova con 250000 elementi, e l’esecuzione avviene senza problemi.

04.5 Confronto

Possiamo dunque concludere confrontando i due codici:

Come prima cosa, si osserva subito che il codice iterativo è più lungo della sua versione ricorsiva, e presenta più cicli for e while che rendono più complicata la comprensione e la possibile correzione/modifica del codice. Il codice ricorsivo è invece più breve e semplice da modificare/correggere, inoltre è decisamente più elegante del primo.

Per quello che riguarda i tempi di esecuzione, questi sono del tutto equivalenti tra le due tipologie di codice, dunque non vi è un vero vantaggio riguardo la complessità temporale, e spesso, dato il menù interattivo realizzato con comando switch, il tempo di esecuzione dipende per lo più dall’utente e dalla sua velocità di elaborazione delle funzioni disponibili del programma.

Ultimo ma non per questo meno importante, riguardando gli output della soluzione ricorsiva a pagina 25, e gli output della soluzione iterativa di pagina 20, è immediata la differenza di utilizzo della memoria delle due tipologie di algoritmi durante l’elaborazione con grandi quantità di dati; la modalità ricorsiva presenta infatti errori di Segmentation Fault dovuto all’utilizzo eccessivo di memoria che, una volta terminata quella disponibile, va a sporcare lo stack a lui non dedicato.

Dunque possiamo concludere che, per un programma dove è necessario lavorare con grandi moli di dati, è suggerita un implementazione iterativa per quanto questa poi diventerà più lunga e difficile da modificare, mentre l’implementazione ricorsiva è suggerita per quei programmi dove ci è permesso “sprecare” un po’ di memoria in più a beneficio di una miglior lettura e di un codice decisamente più breve e “bello” dal punto di vista sintattico ed estetico.

05. Bibliografia e Fonti.

* Algoritmi e strutture dati Astrazione, progetto e realizzazione – McGraw-Hill. Pasquale Foggia e Mario Vento
* C Corso completo di programmazione IV edizione – Apogeo. Paul Deitel & Harvey Deitel
* Slide del corso
* Codici relativi alle esercitazioni
* <https://it.wikipedia.org/wiki/Struttura_dati>

* <https://www.html.it/pag/15418/introduzione-alle-liste/>