REPORT & DOCUMENTAZIONE PROGETTO (Traccia n.3):

Software per la Gestione dello Studio

Nome: Alessia Plaitano

Matricola: 0512123517

Corso di Laurea: Informatica

Corso: Programmazione e Strutture Dati

Docenti: Prof.ssa Carmen Bisogni, Prof.ssa Gemma Catolino

Data Consegna: 30 Maggio 2025

Anno Accademico: 2024-2025

INDICE

1. MOTIVAZIONE DELLA SCELTA E PRESENTAZIONE ADT "ATTIVITA'"

1.1 Motivazione della scelta dell'ADT

La traccia del progetto scelta (n.3) richiede un tipo di dato astratto capace di rappresentare e gestire le attività di studio inserite da uno studente. Le funzioni da implementare, quali la registrazione di nuove attività di studio e la modifica di queste, suggeriscono l'uso di una struttura a dimensione flessibile, che abbia una certa facilità per quel che riguarda l'inserimento e l'eliminazione di nuovi elementi e che risulti anche efficiente.

Nel corso di Programmazione e Strutture Dati, sono state analizzate strutture dati che seguono determinate politiche sull'inserimento e la rimozione dei propri elementi, quali *code* e *stack*, senza operazioni di ricerca efficienti e senza accesso rapido degli elementi, oppure strutture efficienti per la ricerca diretta dei dati ma con un'allocazione fissa di memoria e senza un supporto efficace per le operazioni di scansione ordinata, come le *tabelle hash.*

Il Data Abstract Tipe "Attività" è implementato attraverso una lista collegata. Questa scelta permette una gestione dinamica del numero di attività: a differenza di un array statico, l'aggiunta o la rimozione di un nodo è eseguita in modo rapido ed efficiente, senza la necessità di riallocazioni della memoria per quanto riguarda la sua lunghezza. L'architettura a nodi consente l'ordinamento degli elementi in maniera facilitata, grazie all'algoritmo di ordinamento MergeSort. Questa flessibilità ritorna utile anche per la funzione che stampa il report settimanale, richiesta dalla traccia del progetto. Inoltre, la scelta di quest'implementazione per l'ADT "Attivita" permette un approccio modulare: ogni nodo della lista è autosufficiente, contenendo tutti i dati relativi a una singola attività, cosicché le operazioni di inserimento, eliminazione e aggiornamento siano isolate in funzioni specifiche.

L'uso dell'ADT "Attività" e dell'insieme delle sue funzioni di base è progettato per rispecchiare l'obiettivo del progetto con traccia n.3: aiutare uno studente a pianificare e monitorare il proprio percorso di studio. La struttura dati memorizza e consente di modificare in modo efficiente le informazioni; permette di generare report settimanali che aiutano a valutare l'attività svolta, visualizzando chiaramente quali attività sono completate, quali sono in corso e quali richiedono ancora una revisione, favorendo una pianificazione strategica di studio.

1.2 Presentazione progettuale dell'ADT

```
struct attivita{
    char *nome;
    char *corso_appartenenza;
    char *descrizione;
    struct data data_scadenza;
    struct data data_inserimento;
    int priorita;
    int t_stimato;
    int t_trascorso;
    struct attivita *successivo;
};
```

SPECIFICA SINTATTICA:

-Tipo di riferimento: l'ADT è rappresentato attraverso la struttura “struct attivita”, la quale viene manipolata con dei puntatori per la gestione di una lista dinamica di attività.

-Tipi usati all'interno della struttura:

-Char *, per *nome, corso_appartenenza, descrizione*: questi campi rappresentano delle stringhe, che memorizzano rispettivamente il nome dell'attività, il corso a cui essa appartiene e una descrizione dettagliata;

-Struct data, per *data_scadenza, data_inserimento*: questi campi utilizzano un'altra struttura, composta dai campi giorno, mese, anno, definiti come int, per rappresentare la data entro cui l'attività deve essere completata e la data in cui l'attività viene inserita nel sistema;

-Int, per *priorita, t_stimato, t_trascorso*: campi che rappresentano rispettivamente l’importanza (in termini di urgenza) di un’attività, il tempo previsto per il completamento e il tempo effettivamente dedicato all'attività;

-Struct attivita*, per *successivo*: serve a creare il collegamento tra i nodi, permettendo la costruzione di una lista, dove ogni elemento punta all'attività successiva.

SPECIFICA SEMANTICA:

-*nome*: rappresenta l’identificativo dell’attività;

-*corso_appartenenza*: indica a quale corso è associata l’attività;

-*descrizione*: fornisce ulteriori dettagli sull’attività;

-*data_scadenza*: rappresenta la data entro cui l’attività deve essere completata;

-*data_inserimento:* registra la data in cui l’attività è stata aggiunta al sistema;

-*priorita*: determina il livello di importanza o urgenza dell’attività, varia da 1 a 3.

-t_*stimato*: indica il tempo previsto (espresso in ore) per il completamento dell’attività;

-t_*trascorso*: memorizza il tempo effettivamente impiegato per eseguire l’attività;

-*successivo**: è un puntatore al nodo della lista che rappresenta l’attività successiva;

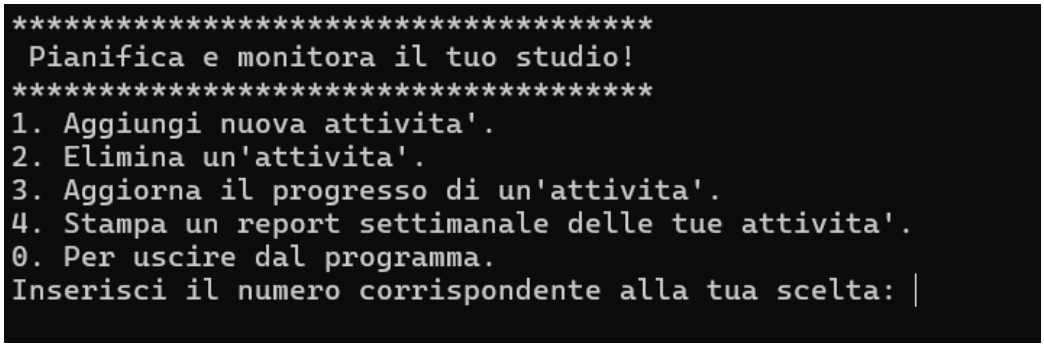
INSIEME DELLE FUNZIONI DI BASE:

nuova_lista_att, aggiungi_attivita, rimuovi_attivita, aggiornamento_progresso, ordina_per_priorita, scadenza_att, report_settimanale, libera_memoria.

2. GUIDA PRATICA PER L'UTENTE

Nella documentazione di un progetto per lo sviluppo di un software, non può di certo mancare una **guida pratica per l'utente**, utile per illustrare come interagire con il sistema. Questa risulta essenziale per scoprire a pieno le potenzialità del software che si sta utilizzando.

Il programma sviluppato è interattivo, ossia richiede un'interazione esterna per la ricezione di dati di input, quali scelte di opzioni o campi da compilare. Esso consente all'utente di pianificare e gestire le sue attività di studio, complete di tutti i dettagli necessari come nome, corso di appartenenza, descrizione, data di scadenza, eccetera….



L'interfaccia iniziale visualizza il menu principale, che illustra le principali possibili azioni da effettuare:

1- Inserimento di una nuova attività;

2- Rimozione di un'attività;

3- Aggiornamento del progresso di un'attività;

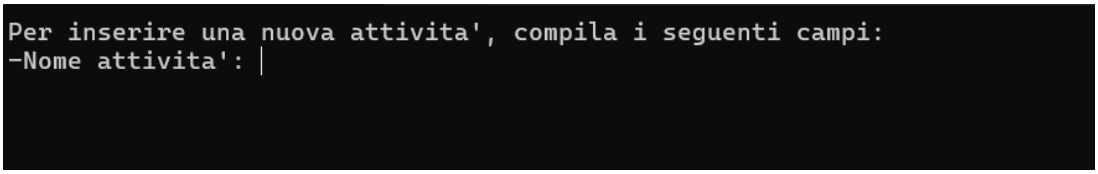
4- Stampa di un report settimanale;

5- Chiusura del programma.

Cosa succede se viene immesso un valore di scelta non riconosciuto? Il programma chiede all'utente di immettere un altro valore valido.

In questo capitolo, paragrafo per paragrafo, analizzeremo singolarmente tutte le scelte, per capirne al meglio l'uso corretto.

2.1 Inserimento di una nuova attività



Selezionando l'opzione n. 1, il programma pulirà lo schermo e chiederà all'utente di compilare la scheda relativa alla nuova attività. In particolare, verranno chiesti:

-il nome dell'attività;

-il corso di appartenenza;

-la descrizione;

-la data di scadenza, in formato gg-mm-aaaa;

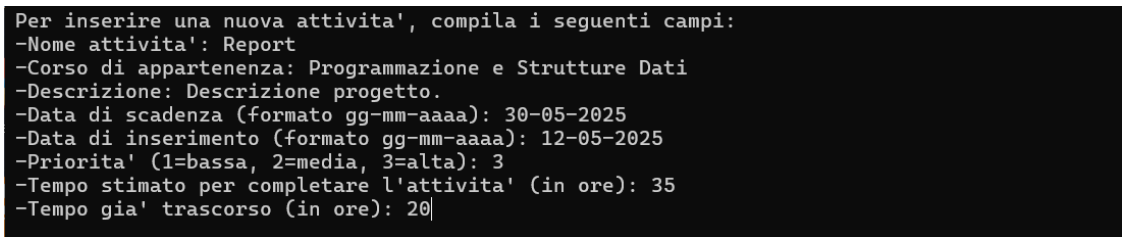
-la data di inserimento, in formato gg-mm-aaaa;

-la priorità, compresa tra 1 e 3, dove 1 è bassa, 2 è media e 3 è alta;

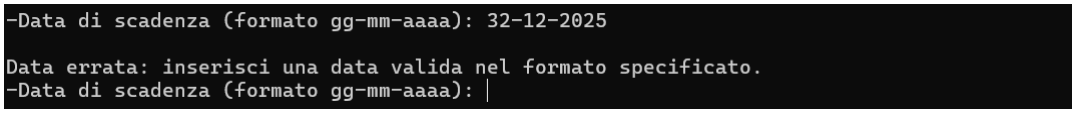
-il tempo stimato per il completamento dell'attività, con valore intero;

-il tempo di lavoro già trascorso, con valore intero.

Di seguito, viene riportato un possibile esempio di inserimento di dati di input:



Cosa succede se l'utente immettere una data errata in formato o una priorità maggiore di 3?
Il programma è dotato di un'azione di controllo sui campi data di scadenza, data di inserimento e priorità. Se riceve dei dati non consoni, stampa un messaggio di errore e chiede l'inserimento di un valore valido per quel campo.



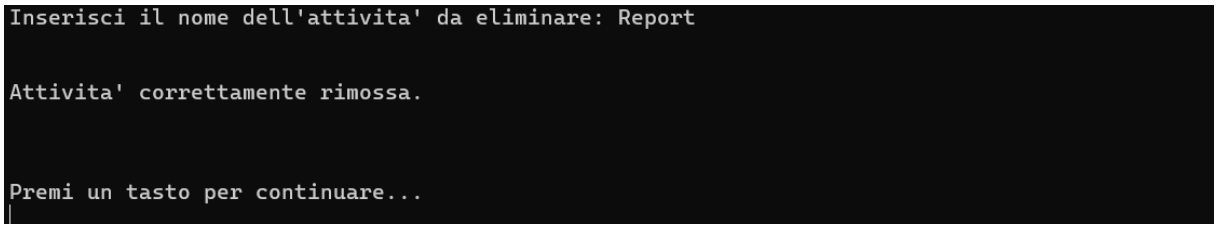
Dopo l'inserimento di tutti i dati validati e corretti, viene avvisato l'utente che la sua nuova attività è stata aggiunta con successo all'agenda. Lo schermo viene pulito autonomamente e viene stampato di nuovo il menu principale.

2.2 Rimozione di un'attività

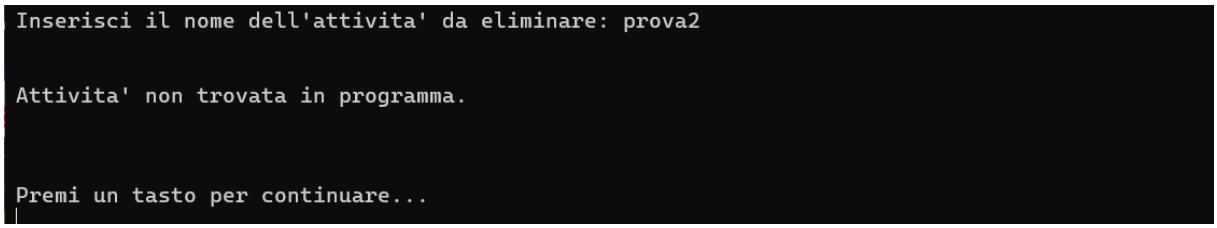
Il programma permette, attraverso l'opzione n.2, di poter rimuovere un'attività tra quelle precedentemente inserite.

Viene chiesto all'utente di immettere il nome dell'attività che desidera eliminare.

Dopodiché, verrà stampato un messaggio di successo, in caso che l'attività inserita è stata trovata ed eliminata:



Altrimenti, verrà stampato un messaggio che informa l'utente che non è stata trovata nessuna corrispondenza con le attività registrate:



Premendo un qualsiasi tasto, lo schermo viene pulito e viene stampato di nuovo il menu principale.

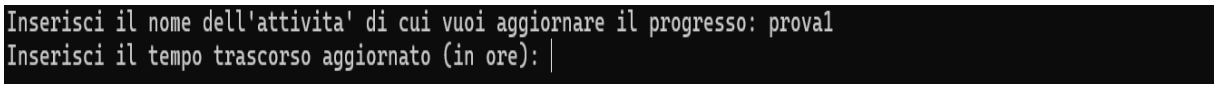
2.3 Aggiornamento del progresso di un'attività

Selezionando l'opzione n. 3, il programma permette all'utente di aggiornare il progresso di un'attività precedentemente inserita.

Nota: Con la dicitura “aggiornare il progresso” si intende l'aggiornamento del campo “Tempo trascorso” dell'attività.

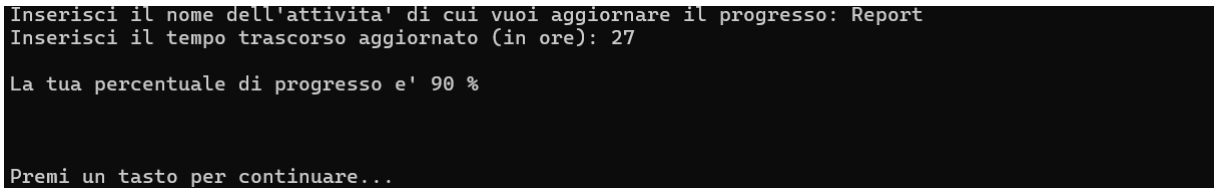
Viene chiesto all'utente di immettere il nome dell'attività di cui desidera aggiornare il progresso e, analogamente per la rimozione, questa attività viene ricercata tra quelle già presenti.

Se l'attività viene trovata, viene chiesto all'utente di inserire il nuovo valore del campo “tempo trascorso”:



Dopodiché, il programma confronta il nuovo valore inserito con il campo “tempo stimato” della medesima attività.

Se il valore inserito è strettamente minore di quest'ultimo, il programma calcola la percentuale di progresso dell'attività e la stampa con un messaggio di successo dell'operazione:



Se, viceversa, il valore risulta maggiore o uguale al campo “tempo stimato”, il programma chiede all'utente se vuole considerare l'attività completata o aggiornare il campo “tempo stimato”:

```
Inserisci il nome dell'attivita' di cui vuoi aggiornare il progresso: Report
Inserisci il tempo trascorso aggiornato (in ore): 35

Il tempo trascorso e' maggiore o uguale del tempo stimato.
Digita 1 se vuoi aggiornare il tempo stimato, altrimenti l'attivita' sara' considerata completata:
```

Inserendo il valore 1, verrà chiesto all'utente di inserire un nuovo valore per il campo “tempo stimato” e verrà stampata la nuova percentuale calcolata di progresso dell'attività selezionata:

```
Inserisci il nome dell'attivita' di cui vuoi aggiornare il progresso: Report
Inserisci il tempo trascorso aggiornato (in ore): 30

Il tempo trascorso e' maggiore o uguale del tempo stimato.
Digita 1 se vuoi aggiornare il tempo stimato, altrimenti l'attivita' sara' considerata completata:
1


Inserisci il nuovo tempo stimato (in ore):
45

Tempo stimato aggiornato!
La tua percentuale di progresso e' 66 %
```

Inserendo un qualsiasi valore, diverso da 1, il programma considererà l'attività completata e stamperà un messaggio di congratulazioni per l'utente:

```
Inserisci il nome dell'attivita' di cui vuoi aggiornare il progresso: Report
Inserisci il tempo trascorso aggiornato (in ore): 45

Il tempo trascorso e' maggiore o uguale del tempo stimato.
Digita 1 se vuoi aggiornare il tempo stimato, altrimenti l'attivita' sara' considerata completata:
3


Complimenti, hai completato l'attivita' con successo!
```

In caso l'attività non venga trovata tra quelle presenti, verrà stampato un messaggio per avvisare l'utente:

```
Inserisci il nome dell'attivita' di cui vuoi aggiornare il progresso: prova1
Inserisci il tempo trascorso aggiornato (in ore): 27

Attivita' non trovata.
```

Dopodiché, premendo un qualsiasi tasto, lo schermo viene pulito e viene stampato di nuovo il menu principale.

2.4 Stampa report settimanale

Attraverso la scelta dell'opzione n. 4, il programma è in grado di stampare un report settimanale delle attività presenti.

Osservazione: la funzione stamperà solo le attività inserite *prima dell’ultimo giorno della settimana,* precedentemente inserito. Inoltre, queste saranno inserite in *ordine di priorità decrescente*, dalla più importante alla meno, i.e. dal valore 3 al valore 1.

```
Inserisci il primo giorno della settimana (formato gg-mm-aaaa): 13-05-2025
Inserisci l'ultimo giorno della settimana (formato gg-mm-aaaa): 20-05-2025
```

In apertura, viene chiesto all’utente di inserire il primo e ultimo giorno della settimana considerata.

```
Inserisci il primo giorno della settimana (formato gg-mm-aaaa): 32-34-2332

Data non valida, inseriscine un'altra nel formato specificato.
Inserisci il primo giorno della settimana (formato gg-mm-aaaa):
```

Anche in questo caso, come ogni volta che viene chiesta l'immissione di una data all'utente, c'è un'azione di controllo riguardo la correttezza dei dati inseriti.

```
Inserisci il primo giorno della settimana (formato gg-mm-aaaa): 12-12-1212
Inserisci l'ultimo giorno della settimana (formato gg-mm-aaaa): 14-12-1212

Nessuna attivita' trovata per questa settimana!
Digita 1 se vuoi inserirne una adesso, altrimenti 0 per uscire:
```

Se non è presente nessuna attività, viene chiesto all'utente se desidera inserirne una, digitando 1. In tal caso, si ritroverà nella stessa situazione dell'opzione "Inserimento di una nuova attività":

```
Digita 1 se vuoi inserirne una adesso, altrimenti 0 per uscire:

1
Per inserire una nuova attivita', compila i seguenti campi:
-Nome attivita':
```

In caso siano presenti delle attività, queste saranno stampate, qualora rispettino l'inserimento entro la data di fine settimana. Inoltre, per ogni attività, verrà stampato il relativo stato di completamento (in corso/scaduta/completata), eventualmente con la percentuale di progresso.

2.5 Chiusura del programma

```
Chiusura programma in corso...

Premi un tasto per continuare...
```

Selezionando l'opzione n. 0, l'utente decide di chiudere il programma, terminandone l'esecuzione. Prima della chiusura, viene stampato un messaggio di avviso per l'utente.

3. PROGETTAZIONE DELLE FUNZIONI IMPLEMENTATE

La progettazione di un programma è il **processo** attraverso il quale *si definisce in modo strutturato e pianificato l'architettura, le funzionalità e le interazioni tra i vari componenti di un software*. Una buona progettazione consente di ottimizzare i tempi di sviluppo, facilitare il debugging e semplificare l'estensione del programma in caso di future esigenze, migliorando la qualità complessiva del prodotto che stiamo creando.

È fondamentale investire tempo nella fase di progettazione perché un approccio pianificato e strutturato previene l'insorgere di problematiche durante lo sviluppo e la manutenzione del software. Serve come guida per l'implementazione, riducendo il rischio di errori che potrebbero sorgere, altrimenti, in un'implementazione disordinata o non coerente.

L'analisi della progettazione verrà fatta riportando linee di codice e spiegandone le logiche di implementazione, date dalla codifica dei vari algoritmi situati dietro alle funzioni coinvolte. Verranno analizzate le funzioni usate nel Main del software principale, descritte nell'header file attivita.h, le funzioni presenti nell'header file mergesortList.h e, successivamente, le funzioni progettate per il software redatto al testing automatico, descritte nell'header file attivitatest.h.

3.1 FUNZIONI DEL FILE ATTIVITA.C:

3.1.1 Funzione nuova_lista_att:

La funzione nuova_lista_att è progettata per creare una nuova lista di attività vuota.

Essa non riceve parametri e, all'interno del corpo della funzione, l'unica operazione eseguita è la restituzione del valore NULL, senza allocazioni di memoria o altre operazioni, che rappresenta una lista priva di elementi. Questa funzione viene utilizzata per inizializzare la lista all'inizio dell'esecuzione del programma, nel Main, preparando la struttura dati per le future operazioni di inserimento.

```
lista_att nuova_lista_att(void){
    return NULL;
}
```

3.1.2 Funzione aggiungi_attivita:

La funzione crea una nuova attività, allocando dinamicamente memoria per un nodo della lista e popolandolo con dati inseriti dall'utente, dopo averne verificato la correttezza.

1. La funzione inizia l'esecuzione allocando dinamicamente memoria per un nuovo nodo della struttura attivita. Esegue subito dopo un controllo sulla sua allocazione, stampando eventualmente un messaggio di errore e terminando in caso:

```
struct attivita *nuovo;
nuovo=malloc(sizeof(struct attivita));
if(nuovo==NULL){
    printf("Allocazione fallita!\n\n");
    return NULL;
}
```

2. Vengono dichiarate le variabili per gestire le date (data_scadenza e d_inserimento), i tempi (t_stimato, t_trascorso), la priorità, e per memorizzare temporaneamente le stringhe nome, descrizione, corso_appartenenza:

```
struct data data_scadenza, d_inserimento;
int t_stimato, t_trascorso, priorita;
char nome[50], descrizione[150], corso_appartenenza[100];
```

3. Si procede con l'acquisizione degli input per il riempimento dei campi del nodo attivita:

```
printf("Per inserire una nuova attivita', compila i seguenti campi: \n");
printf("-Nome attivita': ");
fgets(nome, sizeof(nome), stdin);
nome[strcspn(nome, "\n")]=0;

printf("-Corso di appartenenza: ");
fgets(corso_appartenenza, sizeof(corso_appartenenza), stdin);
corso_appartenenza[strcspn(corso_appartenenza, "\n")]=0;

printf("-Descrizione: ");
fgets(descrizione, sizeof(descrizione), stdin);
descrizione[strcspn(descrizione, "\n")]=0;
```

4. Vengono eseguiti gli inserimenti che richiedono azioni di controllo sulla validità degli input per le strutture data data_scadenza e d_inserimento, poiché vincolate dalla durata e dal numero dei mesi, e per il campo priorita, poiché esso è definito da un range di valori che va da 1 a 3. Questi cicli while, che potrebbero apparentemente sembrare dei loop (la loro condizione è sempre soddisfatta), verificano che i vincoli sui campi descritti precedentemente sia soddisfatti, stampando dei messaggi in caso di errore di validità. Quando le condizioni vengono soddisfatte si esce dal ciclo e l'esecuzione del programma procede, grazie a break.

```
while(1){
    //Il ciclo termina solo se l'utente inserisce una data di scadenza valida.
    printf("-Data di scadenza (formato gg-mm-aaaa): ");
    if(scanf("%d-%d-%d", &data_scadenza.giorno, &data_scadenza.mese, &data_scadenza.anno)!= 3 ||
        data_scadenza.giorno<1 || data_scadenza.giorno>31 ||
        data_scadenza.mese<1 || data_scadenza.mese>12){
        printf("\nData errata: inserisci una data valida nel formato specificato.\n");
        while(getchar()!='\n');
    } else break;
}

while(1){
    //Il ciclo termina solo se l'utente inserisce una data di scadenza valida.
    printf("-Data di inserimento (formato gg-mm-aaaa): ");
    if(scanf("%d-%d-%d", &d_inserimento.giorno, &d_inserimento.mese, &d_inserimento.anno)!= 3 ||
        d_inserimento.giorno<1 || d_inserimento.giorno>31 ||
        d_inserimento.mese<1 || d_inserimento.mese>12){
        printf("\nData errata: inserisci una data valida nel formato specificato.\n");
        while(getchar()!='\n');
    } else break;
}
```

```
while(1){
    //Il ciclo termina solo se l'utente inserisce un valore valido per "priorità".
    printf("-Priorita' (1=bassa, 2=media, 3=alta): ");
    if(scanf("%d", &priorita)!= 1 || priorita<1 || priorita>3){
        printf("\nErrore: inserisci un numero valido compreso tra 1 e 3.\n");
        while(getchar()!='\n');
    } else break;
}
```

5. Sono richiesti all'utente gli input per riempire i restanti campi della struct attivita, ossia t_stimato e t_trascorso:

```
printf("-Tempo stimato per completare l'attivita' (in ore): ");
scanf("%d", &t_stimato);

printf("-Tempo gia' trascorso (in ore): ");
scanf("%d", &t_trascorso);
```

6. Prima di copiare i dati, la funzione alloca memoria (mediante malloc) per i campi stringa nome, descrizione e corso_appartenenza della struttura. Se l'allocazione di uno di questi campi fallisce, viene eseguito un controllo che libera la memoria già allocata e stampa un messaggio di errore:

```
nuovo->nome=malloc(strlen(nome)+1);
nuovo->descrizione=malloc(strlen(descrizione)+1);
nuovo->corso_appartenenza=malloc(strlen(corso_appartenenza)+1);

if(!nuovo->nome || !nuovo->descrizione || !nuovo->corso_appartenenza){
    //Controllo complessivo della memoria allocata dinamicamente.
    printf("\nSi è verificato un errore nell'inserimento.\n\n\n");
    free(nuovo->nome);
    free(nuovo->descrizione);
    free(nuovo->corso_appartenenza);
    free(nuovo);
    return NULL;
}
```

7. Con le funzioni strcpy, i dati presenti nei buffer locali vengono copiati negli appositi campi dinamici del nuovo nodo. Successivamente, vengono assegnati i valori relativi alle struct data, priorità e tempo:

```
strcpy(nuovo->nome,nome);
strcpy(nuovo->descrizione, descrizione);
strcpy(nuovo->corso_appartenenza, corso_appartenenza);
nuovo->data_scadenza=data_scadenza;
nuovo->priorita=priorita;
nuovo->t_stimato=t_stimato;
nuovo->t_trascorso=t_trascorso;
```

8. Il nuovo nodo viene inserito in testa alla lista e l'esecuzione termina stampando un messaggio di successo, restituendo il nuovo nodo, che diventa la nuova testa della lista:

```
nuovo->successivo=l;

printf("\n\nNuova attivita' aggiunta con successo!\n\n\n");
return nuovo;
```

3.1.3 Funzione rimuovi_attivita:

La funzione è progettata per eliminare un'attività specifica dalla lista, basandosi sul nome inserito dall'utente. Distingue i diversi casi:

-Se la lista è vuota, non viene effettuata alcuna operazione;

-Se l'attività da eliminare si trova in testa alla lista, ne viene aggiornata la testa;

-Se l'attività si trova in una posizione intermedia o in coda, il puntatore del nodo precedente viene aggiornato per saltare il nodo interessato.

Alla fine dell'operazione, la funzione libera la memoria associata al nodo, cioè all'attività, stampando un messaggio di conferma o, nel caso in cui l'attività non venga trovata, un messaggio di errore. Il risultato è la lista aggiornata.

1. All'inizio dell'esecuzione, vengono dichiarati due puntatori (corrente e precedente) per scorrere la lista. Viene richiesto e memorizzato il nome dell'attività da eliminare. La funzione fgets legge la stringa immessa e strcspn rimuove il carattere di newline residuo:

```
struct attivita *corrente=l, *precedente=NULL;
char nome[50];
printf("Inserisci il nome dell'attivita' da eliminare: ");
fgets(nome, sizeof(nome), stdin);
nome[strcspn(nome, "\n")]=0;
```

2. La funzione scorre la lista nodo per nodo, grazie al puntatore corrente, confrontando il campo nome di ciascun nodo con il nome immesso dall'utente, tramite la funzione strcmp:

```
while(corrente != NULL){
    if(strcmp(corrente->nome, nome)==0){
```

3. Se il nodo individuato è il primo della lista, il puntatore della lista (l) viene aggiornato per puntare al nodo successivo:

```
if(precedente == NULL){
    //Caso: l'attività da eliminare è la prima della lista.
    l=corrente->successivo;
```

4. Altrimenti, se l'elemento si trova in una posizione intermedia o in coda, il campo successivo del nodo precedente viene aggiornato per saltare il nodo da rimuovere:

```
} else{
    //Caso: l'attività da eliminare è in mezzo alla lista o è l'ultima (in tal caso, gestione implicita).
    //Se è l'ultima: corrente->successivo=NULL, quindi precedente->successivo=NULL.
    precedente->successivo=corrente->successivo;
}
```

5. Le chiamate alla funzione free rimuovono la memoria allocata per i campi dinamici del nodo ed, infine, per il nodo stesso, evitando memory leak. Dopo la deallocazione, viene stampato un messaggio di conferma e la funzione restituisce la lista aggiornata:

```
    free(corrente->nome);
    free(corrente->corso_appartenenza);
    free(corrente->descrizione);
    free(corrente);
    printf("\n\nAttivita' correttamente rimossa.\n\n\n");
    return l; //Lista aggiornata.
}
```

6. Nel caso in cui l'iterazione finisce senza trovare un nodo il cui nome corrisponda a quello inserito, la funzione stampa un messaggio di errore e restituisce la lista originale:

```
printf("\n\nAttivita' non trovata in programma.\n\n\n");
return l; //Lista originale.
```

3.1.4 Funzione aggiornamento_progresso:

La funzione aggiorna il campo t_trascorso di una specifica attività individuata dall'utente, nella lista delle attività. Dopo aver aggiornato questo campo, ne calcola e stampa la percentuale di progresso rispetto al tempo stimato. Se il tempo trascorso risulta maggiore o uguale al tempo stimato, la funzione chiede all'utente se desidera aggiornare il valore di t_stimato oppure considera l'attività completata. In base al risultato, viene fornito un riscontro per l'utente e restituito 1 se l'aggiornamento ha avuto successo, altrimenti 0 se l'attività non è stata trovata.

1. Vengono dichiarate le variabili locali per memorizzare il puntatore corrente, per scorrere la lista, gli interi per il tempo aggiornato (t_trascorso_agg), per la scelta dell'utente (val), per l'eventuale nuovo tempo stimato (t_stimato_agg) e per la percentuale di progresso, e un array di caratteri nome per salvare il nome dell'attività da cercare. Il nome dell'attività viene acquisito con fgets e "ripulito" rimuovendo il carattere di newline mediante strcspn. Successivamente, viene richiesto all'utente di immettere il nuovo tempo trascorso aggiornato:

```
struct attivita *corrente;
int t_trascorso_agg, val, t_stimato_agg, percentuale;
char nome[50];
printf("Inserisci il nome dell'attivita' di cui vuoi aggiornare il progresso: ");
fgets(nome, sizeof(nome), stdin);
nome[strcspn(nome, "\n")]=0;
printf("Inserisci il tempo trascorso aggiornato (in ore): ");
scanf("%d", &t_trascorso_agg);
```

2. Usando un ciclo for, assegnando a corrente il valore iniziale l e aggiornandolo ad ogni ciclo, si scorre la lista per intero. Per ogni nodo esaminato, viene confrontato il campo nome con la stringa immessa dall'utente tramite strcmp. Se il risultato di strcmp è zero (cioè se c'è corrispondenza), l'attività è stata trovata e il campo t_trascorso può essere aggiornato:

```
for(corrente=l; corrente!=NULL; corrente=corrente->successivo){
    if((strcmp(corrente->nome, nome)== 0)){
        corrente->t_trascorso=t_trascorso_agg;
```

3. Se t_stimato è maggiore di t_trascorso, si calcola la percentuale di completamento dell'attività (in maniera implicita all'interno della funzione) e ne viene stampato il suo valore:

```
if(corrente->t_stimato > corrente->t_trascorso){
   //Aggiorna con numeri positivi il valore t_stimato.
   percentuale=(corrente->t_trascorso*100)/corrente->t_stimato;
   printf("\nLa tua percentuale di progresso e' %d %%\n\n\n", percentuale);
}
```

4. Invece, se t_trascorso è maggiore o uguale a t_stimato, l'attività potrebbe essere stata già completata o potrebbe essere necessario aggiornare il tempo stimato. Viene informato l'utente

```
else {
    printf("\nIl tempo trascorso e' maggiore o uguale del tempo stimato.\n");
    printf("Digita 1 se vuoi aggiornare il tempo stimato, altrimenti l'attivita' sara' considerata completata: \n");
    scanf("%d", &val);
    if(val==1){
        printf("\n\nInserisci il nuovo tempo stimato (in ore): \n");
        scanf("%d", &t_stimato_agg);
        corrente->t_stimato=t_stimato_agg;
        printf("\nTempo stimato aggiornato!\n");
        if(corrente->t_stimato>0){
           percentuale=(corrente->t_trascorso*100)/corrente->t_stimato;
           printf("La tua percentuale di progresso e' %d %%\n\n\n", percentuale);
        }
    } else printf("\n\nComplimenti, hai completato l'attivita' con successo!\n\n\n");
}
```

e chiesto se si desidera aggiornare il tempo stimato. Digitando 1, viene aggiornato il campo t_stimato nel nodo e stampata la nuova percentuale. Se, invece, si sceglie di non aggiornare, viene stampato un messaggio di congratulazioni per il completamento dell'attività.

5. Prima della fine dell'esecuzione, la funzione restituisce 1 se l'aggiornamento dei campi è avvenuto correttamente, altrimenti 0, con un messaggio per l'utente.

3.1.5 Funzione ordina_per_priorita:

La funzione riordina una lista di attività in base al campo "priorità" in ordine decrescente (dalla priorità 3 alla priorità 1). Utilizza l'algoritmo MergeSort, delegando il compito alla funzione mergesort_lista, che è responsabile dell'ordinamento vero e proprio della lista. Infatti, la funzione si compone di una sola riga di codice, dove è presente la chiamata a mergesort_lista.

```
        return 1; //Attività trovata e modificata.
        }
    }
    printf("\nAttivita' non trovata.\n\n\n");
    return 0;
void ordina_per_priorita(lista_att *l){
    *l=mergesort_lista(*l);
}
```

3.1.6 Funzione scadenza_att:

La funzione determina se un'attività è scaduta confrontando due strutture di tipo struct data, ossia data_scadenza e fine_sett, che rappresenta la fine della settimana inserita nella funzione report_settimanale. Viene restituito 1, se la data di scadenza è precedente alla data di fine settimana (quindi l'attività è scaduta), oppure 0, in caso contrario.

1. Si compone di una serie di confronti effettuati con delle istruzioni if di selezione:

-Se data_scadenza.anno è inferiore a fine_sett.anno, ciò implica che la scadenza è avvenuta in un anno precedente; quindi, l'attività è considerata scaduta e la funzione restituisce 1;

-Se gli anni coincidono, il confronto passa al mese: se il mese di data_scadenza è inferiore a quello di fine_sett, l'attività è scaduta e la funzione restituisce 1;

-Se gli anni e i mesi sono uguali, il confronto si sposta ai giorni: se il giorno di data_scadenza è minore di quello di fine_sett, l'attività risulta scaduta e la funzione restituisce 1.

```
if(data_scadenza.anno < fine_sett.anno) return 1;
else if(data_scadenza.anno == fine_sett.anno){
        if(data_scadenza.mese<fine_sett.mese)
        return 1;
        else if(data_scadenza.mese == fine_sett.mese){
                if(data_scadenza.giorno < fine_sett.giorno)
                return 1;
```

2. Altrimenti significa che la data di scadenza non è antecedente a quella di fine settimana. Dunque, funzione restituisce 0, indicando che l'attività non è scaduta:

```
    }
return 0;
```

3.1.7 Funzione report_settimanale:

La funzione genera un report settimanale sulle attività presenti nella lista. Chiede all'utente di inserire due date di riferimento (il primo e l'ultimo giorno della settimana corrente). Se la lista è vuota, offre all'utente la possibilità di inserire una nuova attività. Viceversa, se la lista contiene almeno un'attività, la funzione scorre la lista e, per ogni attività registrata entro la data di fine settimana, ne stampa tutti i campi. Usando scadenza_att, determina se l'attività è scaduta, completata o in corso, e stampa il relativo messaggio in uscita.

1. La funzione inizia ordinando la lista con la funzione ordina_per_priorita, in base al campo "priorità" (in ordine decrescente). Vengono poi dichiarate le variabili per memorizzare le date di inizio e fine settimana, oltre ad una variabile di controllo val, una variabile per determinare la scadenza delle attività ed una per il calcolo della percentuale di completamento:

```
ordina_per_priorita(l);
struct attivita *corrente=*l;
struct data inizio_sett, fine_sett;
int val, scadenza, percentuale;
```

2. Vengono eseguiti gli inserimenti che richiedono azioni di controllo sulla validità degli input per le strutture data inizio_sett e fine_sett, poiché vincolate dalla durata e dal numero dei mesi. Questi cicli while, verificano che i vincoli sui campi descritti precedentemente sia soddisfatti, stampando dei messaggi in caso di errore di validità. Quando le condizioni vengono soddisfatte si esce dal ciclo e l'esecuzione del programma procede, grazie all'azione di break:

```
while(1){
    //Il ciclo termina solo se l'utente inserisce una data valida.
    printf("Inserisci il primo giorno della settimana (formato gg-mm-aaaa): ");
    if(scanf("%d-%d-%d", &inizio_sett.giorno, &inizio_sett.mese, &inizio_sett.anno)!= 3 ||
        inizio_sett.giorno<1 || inizio_sett.giorno>31 ||
        inizio_sett.mese<1 || inizio_sett.mese>12){
        printf("\nData non valida, inseriscine un'altra nel formato specificato.\n");
        while(getchar()!='\n');
    } else break;
}

while(1){
    //Il ciclo termina solo se l'utente inserisce una data valida.
    printf("Inserisci l'ultimo giorno della settimana (formato gg-mm-aaaa): ");
    if(scanf("%d-%d-%d", &fine_sett.giorno, &fine_sett.mese, &fine_sett.anno)!= 3 ||
        fine_sett.giorno<1 || fine_sett.giorno>31 ||
        fine_sett.mese<1 || fine_sett.mese>12){
        printf("\nData non valida, inseriscine un'altra nel formato specificato.\n");
        while(getchar()!='\n');
    } else break;
}
```

3. Se la lista risulta vuota, la funzione offre all'utente la possibilità di inserire una nuova attività, digitando il valore 1. In tal caso, la funzione aggiungi_attivita viene chiamata, aggiornando la lista e il puntatore corrente. Altrimenti, in caso contrario, la funzione termina l'esecuzione:

```c
if(corrente==NULL){ //Lista vuota.
    printf("\nNessuna attivita' trovata per questa settimana!\n");
    printf("Digita 1 se vuoi inserirne una adesso, altrimenti 0 per uscire: \n\n");
    scanf("%d", &val);
    while(getchar()!='\n');
    if(val==1){
       *l=aggiungi_attivita(*l);
       corrente=*l;
    } else{
       return;
    }
}
```

4. Ora avviene la vera e propria generazione del report settimanale: scorrendo la lista, per ogni attività inserita entro la fine della settimana, questa viene stampata con i relativi campi:

```c
printf("\n*****************************************************************************************************");
printf("\nEcco un report settimanale delle attivita' inserite entro questa settimana, ordinate per priorita' maggiore!");
printf("\n*****************************************************************************************************\n\n");
while(corrente!=NULL){
    //Se l'attività è stata aggiunta entro il giorno di fine settimana, stampala.
    if (corrente->data_inserimento.anno<fine_sett.anno ||
        (corrente->data_inserimento.anno==fine_sett.anno && corrente->data_inserimento.mese<fine_sett.mese) ||
        (corrente->data_inserimento.anno==fine_sett.anno && corrente->data_inserimento.mese==fine_sett.mese &&
        corrente->data_inserimento.giorno<=fine_sett.giorno)){

        printf("Nome: %s\n", corrente->nome);
        printf("Corso di appartenenza: %s\n", corrente->corso_appartenenza);
        printf("Descrizione: %s\n", corrente->descrizione);
        printf("Data di scadenza: %d-%d-%d\n", corrente->data_scadenza.giorno, corrente->data_scadenza.mese, corrente->data_scadenza.anno);
        printf("Data di inserimento: %d-%d-%d\n", corrente->data_inserimento.giorno, corrente->data_inserimento.mese, corrente->data_inser
        printf("Priorita': %d\n", corrente->priorita);
        printf("Tempo stimato: %d ore\n", corrente->t_stimato);
        printf("Tempo trascorso: %d ore\n", corrente->t_trascorso);
```

5. Usando scadenza_att, la funzione determina se l'attività è scaduta. Se scadenza ha valore 1, l'attività è scaduta. Se il tempo stimato è uguale a quello trascorso o se t_stimato è 0: viene indicato che l'attività è stata completata con successo. Altrimenti, viene calcolata e stampata la percentuale di completamento:

```c
    scadenza=scadenza_att(corrente->data_scadenza, fine_sett);
    //scadenza=0 se non è scaduta l'attività, altrimenti 1.

   if(scadenza==1)
        printf("L'attivita' e' scaduta!\n\n");
    else if(corrente->t_stimato==corrente->t_trascorso || corrente->t_stimato==0)
            printf("L'attivita' e' stata completata con successo!\n\n");
         else {
            percentuale=(corrente->t_trascorso*100)/corrente->t_stimato;
            printf("L'attivita' e' in corso, con una percentuale di completamento %d %%!\n\n", percentuale);
        }
```

3.1.8 Funzione libera_memoria:

La funzione dealloca tutta la memoria dinamica occupata dalla lista di attività, prevenendo così memory leak, prima della conclusione dell'esecuzione del programma. Questo processo è essenziale per una gestione corretta delle risorse che si servono di memoria dinamica.

1. Viene inizializzato il puntatore corrente con il valore attuale di *l, che rappresenta la testa della lista:

```c
struct attivita *corrente=*l;
```

2. Il ciclo while viene eseguito finché non si raggiunge la fine della lista e, al suo interno, il nodo corrente viene assegnato temporaneamente a temp per conservarne il riferimento, per permettere di accedere a tutte le informazioni del nodo corrente anche dopo aver aggiornato il puntatore di iterazione:

```
while(corrente!=NULL){
struct attivita *temp=corrente;
```

3. Il puntatore corrente viene aggiornato per puntare al nodo successivo, sostituendo il nodo attuale con quello seguente e viene liberata la memoria allocata per ciascun campo stringa dinamico nome, descrizione, corso_appartenenza appartenente al nodo. Successivamente, viene deallocato il nodo stesso:

```
corrente=corrente->successivo;
free(temp->nome);
free(temp->descrizione);
free(temp->corso_appartenenza);
free(temp);
}
```

4. Una volta completato il ciclo e liberata tutta la memoria relativa ai nodi della lista, il puntatore passato per riferimento viene impostato a NULL. Ciò impedisce possibili accessi futuri a memoria ormai liberata, evitando errori come i dangling pointer (i puntatori pendenti):

```
*l=NULL; //Evita futuri accessi non validi.
```

3.2 FUNZIONI DEL FILE MERGESORTLIST.C:

3.2.1 Funzione trova_meta:

La funzione individua il punto medio della lista, ossia l'elemento posto centralmente, e lo restituisce. Questo è il primo passo nell'algoritmo MergeSort per le liste, dove la lista viene divisa in due metà per poi essere ordinate ricorsivamente. La funzione è chiamata solo se la lista contiene più di un elemento; in caso contrario, restituisce direttamente head.

1. Inizialmente verifica se la lista è vuota oppure contiene un solo. Se la condizione è verificata, la funzione restituisce direttamente head, poiché non è necessario individuare un punto medio in una lista con meno di due attività:

```
if(head==NULL || head->successivo==NULL) return head;
```

2. Dichiara due puntatori per attraversare la lista a ritmi diversi: il puntatore "lento", inizializzato a head, che si sposta di un solo nodo per iterazione, e il puntatore "veloce", inizializzato a head->successivo, che si sposta di due nodi per ogni iterazione:

```
struct attivita *lento=head, *veloce=head->successivo;
```

3. Questo ciclo continua finché il puntatore veloce non raggiunge la fine della lista o non esiste un successivo di veloce->successivo, ossia finché il puntatore lento non si trova al nodo centrale della lista:

```
while(veloce && veloce->successivo){
    lento=lento->successivo;
    veloce=veloce->successivo->successivo;
}
```

4. Infine, viene ritornato il puntatore lento, ossia quello al nodo centrale della lista:

```
return lento;
```

3.2.2 Funzione unione_liste:

La funzione unisce due liste ordinate in ordine decrescente in base al campo priorita. Rappresenta il passo finale dell'algoritmo MergeSort per le liste, fondendo due sotto-liste ordinate in un'unica lista ordinata. La funzione usa la ricorsione per procedere nell'unione, modificando direttamente il campo puntatore successivo dei nodi, in modo da mantenere l'ordinamento.

1. Con due azioni di controllo, si verifica all'inizio se una delle due liste è vuota. In tal caso, la funzione restituisce l'altra lista:

```
if(lista1==NULL) return lista2;
if(lista2==NULL) return lista1;
```

2. Altrimenti, esistono 2 sotto casi:

-Se il nodo corrente di lista1 ha una priorità maggiore o uguale rispetto al nodo corrente di lista2, si unisce ricorsivamente la parte restante di lista1 (ossia lista1->successivo) con l'intera lista2 e si assegna il risultato al campo successivo di lista1. In questo modo, il nodo di lista1 mantiene la sua posizione come testa della nuova lista unita:

```
if(lista1->priorita >= lista2->priorita){
    lista1->successivo=unione_liste(lista1->successivo, lista2);
    return lista1;
```

3. -Se il nodo corrente di lista2 ha una priorità maggiore rispetto al nodo corrente di lista1, invece, si unisce ricorsivamente l'intera lista1 con la parte rimanente di lista2 (ossia lista2->successivo) e si assegna il risultato al campo successivo di lista2. In questo modo, il nodo di lista2 diventa la testa della nuova lista unita:

```
} else{
    lista2->successivo=unione_liste(lista1, lista2->successivo);
    return lista2;
}
```

Le chiamate ricorsive si ripetono fino a fondere completamente le due liste, modificando i puntatori successivo di ogni nodo coinvolto.

3.2.3 Funzione mergesort_lista:

La funzione implementa l'algoritmo di ordinamento MergeSort applicato a una lista collegata di attività, ordinandole in base al campo priorita in ordine decrescente. Se la lista contiene più di un elemento, la funzione la divide in due sottoliste, utilizzando la funzione trova_meta (che individua il punto medio), e procede ricorsivamente ad ordinare le due metà. Infine, le due sottoliste ordinate vengono fuse, mediante la funzione unione_liste, per ottenere la lista finale ordinata. Se la lista è vuota o contiene un solo elemento, la funzione la restituisce invariata.

1. Verificare se la lista è vuota o se contiene un solo nodo. In entrambi i casi, la lista viene considerata già ordinata e viene restituita senza ulteriori modifiche:

```
//se la lista è vuota o ha un solo elemento, è considerata già ordinata.
if(head==NULL || head->successivo==NULL) return head;
```

2. Altrimenti, viene chiamata la funzione trova_meta per individuare il nodo centrale della lista, assegnandolo al puntatore meta. Il puntatore seconda_meta viene impostato al nodo successivo rispetto a meta. La rimozione del collegamento meta->successivo = NULL separa effettivamente la lista in due sotto-liste:

```
struct attivita* meta=trova_meta(head);
struct attivita* seconda_meta=meta->successivo;
meta->successivo=NULL;
```

3. Le due sotto-liste vengono ordinate ricorsivamente chiamando mergesort_lista su ognuna. Il processo di divisione continua finché ciascuna contiene al massimo un elemento:

```
struct attivita* sinistra=mergesort_lista(head);
struct attivita* destra=mergesort_lista(seconda_meta);
```

4. A questo punto, viene chiamata la funzione unione_liste per fondere le due sotto-liste ordinate in una singola lista ordinata:

```
return unione_liste(sinistra, destra);
```

3.3 FUNZIONI DEL FILE ATTIVITATEST.C:

3.3.1 Funzione nuova_lista_att:

La funzione nuova_lista_att è progettata per creare una nuova lista di attività vuota.

Essa non riceve parametri e, all'interno del corpo della funzione, l'unica operazione eseguita è la restituzione del valore NULL, senza allocazioni di memoria o altre operazioni, che rappresenta una lista priva di elementi. Questa funzione viene utilizzata per inizializzare la lista all'inizio dell'esecuzione del programma, nel Main, preparando la struttura dati per le future operazioni di inserimento.

```
lista_att nuova_lista_att(void){
    return NULL;
}
```

3.3.2 Funzione test_aggiungi_attivita:

La funzione legge da un file di input tutti i dati necessari per l'inserimento di una nuova attività in una lista di attività. Durante il processo di lettura, la funzione valida ogni campo (nome, corso, descrizione, date, priorità, tempo stimato e tempo trascorso). Se in una qualsiasi fase si verifica un errore (ad esempio, la mancanza di dati, dati non validi o fallimenti nelle allocazioni dinamiche), viene stampato un messaggio d'errore sul file di output e tutte le risorse eventualmente allocate vengono liberate, restituendo la lista originale (senza modifiche).

1. Viene allocata dinamicamente la struttura per il nuovo nodo. Se l'allocazione fallisce, viene scritto un messaggio d'errore su tc_output e si restituisce la lista invariata:

```
struct attivita *nuovo=malloc(sizeof(struct attivita));
if(nuovo == NULL){
    fprintf(tc_output, "Allocazione fallita!\n\n");
    return l;  // l è lista originale.
}
```

2. Vengono dichiarate le variabili per memorizzare temporaneamente i dati letti da tc_input. La variabile buffer viene impiegata per leggere input intermedi (ad esempio, per le date e per i numeri interi) con fgets per garantire la corretta gestione del buffer e delle nuove linee:

```
struct data data_scadenza, d_inserimento;
int t_stimato, t_trascorso, priorita;
char nome[50], descrizione[150], corso_appartenenza[100];
char buffer[100];
```

3. Il nome viene letto dalla funzione usando fgets, eliminando però il carattere di newline. Se la lettura fallisce o il campo risulta vuoto, viene stampato un errore, liberata la memoria del nodo e la funzione restituisce la lista invariata:

```
fprintf(tc_output, "Per inserire una nuova attivita', compila i seguenti campi:\n");

if(fgets(nome, sizeof(nome), tc_input) == NULL){
    fprintf(tc_output, "Errore nella lettura del nome.\n");
    free(nuovo);
    return l;
}
nome[strcspn(nome, "\n")]='\0';
if(strlen(nome) == 0){
    fprintf(tc_output, "Il campo nome non può essere vuoto!\n");
    free(nuovo);
    return l;
}
fprintf(tc_output, "-Nome attivita': %s\n", nome);
```

4. Analogamente al campo nome, il corso viene letto e validato: in caso di errore o campo vuoto, la funzione segnala il problema, libera la memoria e restituisce la lista originale. Identicamente, avviene per il campo descrizione:

```c
if(fgets(corso_appartenenza, sizeof(corso_appartenenza), tc_input) == NULL){
    fprintf(tc_output, "Errore nella lettura del corso di appartenenza.\n");
    free(nuovo);
    return l;
}
corso_appartenenza[strcspn(corso_appartenenza, "\n")]='\0';
if(strlen(corso_appartenenza) == 0){
    fprintf(tc_output, "Il campo corso di appartenenza non può essere vuoto!\n");
    free(nuovo);
    return l;
}
fprintf(tc_output, "-Corso di appartenenza: %s\n", corso_appartenenza);


if(fgets(descrizione, sizeof(descrizione), tc_input) == NULL){
    fprintf(tc_output, "Errore nella lettura della descrizione.\n");
    free(nuovo);
    return l;
}
descrizione[strcspn(descrizione, "\n")]='\0';
if(strlen(descrizione) == 0){
    fprintf(tc_output, "La descrizione non può essere vuota!\n");
    free(nuovo);
    return l;
}
fprintf(tc_output, "-Descrizione: %s\n", descrizione);
```

5. Successivamente, viene usato un ciclo while(1) per assicurare che il dato inserito per il campo struct data (data_scadenza e d_inserimento) sia nel formato corretto, utilizzando fgets e sscanf per scansionare la stringa formattata. Se il dato è errato, viene segnalato l'errore e il ciclo continua fino all'immissione di un valore valido:

```c
while (1){
    //Ciclo iterato finchè non si immette una data valida.
    if(fgets(buffer, sizeof(buffer), tc_input) == NULL){
        fprintf(tc_output, "Errore nella lettura della data di scadenza.\n");
        free(nuovo);
        return l;
    }
    if(sscanf(buffer, "%d-%d-%d", &data_scadenza.giorno, &data_scadenza.mese, &data_scadenza.anno) != 3 ||
        data_scadenza.giorno<1 || data_scadenza.giorno>31 ||
        data_scadenza.mese<1 || data_scadenza.mese>12){
        fprintf(tc_output, "Inserisci una data valida nel formato specificato (gg-mm-aaaa).\n");
    } else {
        break;
    }
}
fprintf(tc_output, "-Data di scadenza: %02d-%02d-%04d\n", data_scadenza.giorno, data_scadenza.mese, data_scadenza
```

```c
while (1){
    //Ciclo iterato finchè non si immette una data valida.
    if(fgets(buffer, sizeof(buffer), tc_input) == NULL){
        fprintf(tc_output, "Errore nella lettura della data di inserimento.\n");
        free(nuovo);
        return l;
    }
    if(sscanf(buffer, "%d-%d-%d", &d_inserimento.giorno, &d_inserimento.mese, &d_inserimento.anno) != 3 ||
        d_inserimento.giorno<1 || d_inserimento.giorno>31 ||
        d_inserimento.mese<1 || d_inserimento.mese>12){
        fprintf(tc_output, "Inserisci una data valida nel formato specificato (gg-mm-aaaa).\n");
    } else {
        break;
    }
}
fprintf(tc_output, "-Data di inserimento: %02d-%02d-%04d\n", d_inserimento.giorno, d_inserimento.mese, d_inserime
nuovo->data_inserimento = d_inserimento;
```

6- La stessa azione di controllo viene eseguita per verificare la validità dell'input assegnato al campo priorita, che deve essere un intero compreso tra 1 e 3:

```
while (1){
    //Ciclo iterato finchè non si immette una valore valido.
    if(fgets(buffer, sizeof(buffer), tc_input) == NULL){
        fprintf(tc_output, "Errore nella lettura della priorità.\n");
        free(nuovo);
        return l;
    }
    if(sscanf(buffer, "%d", &priorita) != 1 || priorita < 1 || priorita > 3){
        fprintf(tc_output, "Inserisci un numero valido, compreso tra 1 e 3!\n");
    } else {
        break;
    }
}
fprintf(tc_output, "-Priorità: %d\n", priorita);
```

7- Viene poi letto il tempo stimato e il tempo già trascorso. Se uno dei due campi non viene letto correttamente, viene stampato un messaggio d'errore, la memoria allocata finora viene liberata e la funzione restituisce la lista invariata:

```
if(fgets(buffer, sizeof(buffer), tc_input) == NULL || sscanf(buffer, "%d", &t_stimato) != 1){
    fprintf(tc_output, "Errore nella lettura del tempo stimato.\n");
    free(nuovo);
    return l;
}
fprintf(tc_output, "-Tempo stimato per completare l'attivita': %d ore\n", t_stimato);


if(fgets(buffer, sizeof(buffer), tc_input) == NULL || sscanf(buffer, "%d", &t_trascorso) != 1){
    fprintf(tc_output, "Errore nella lettura del tempo gia' trascorso.\n");
    free(nuovo);
    return l;
}
fprintf(tc_output, "-Tempo gia' trascorso: %d ore\n", t_trascorso);
```

8- Dopo, si alloca dinamicamente memoria per le stringhe della struttura (nome, descrizione e corso_appartenenza). Se una di queste fallisce, il codice libera tutte le risorse allocate e restituisce la lista originale dopo aver stampato un messaggio di errore:

```
//Allocazione dinamica dei campi stringa.
nuovo->nome=malloc(strlen(nome) + 1);
nuovo->descrizione=malloc(strlen(descrizione) + 1);
nuovo->corso_appartenenza=malloc(strlen(corso_appartenenza) + 1);
if(!nuovo->nome || !nuovo->descrizione || !nuovo->corso_appartenenza){
    fprintf(tc_output, "Si è verificato un errore nell'inserimento.\n\n");
    free(nuovo->nome);
    free(nuovo->descrizione);
    free(nuovo->corso_appartenenza);
    free(nuovo);
    return l;
}
```

9- A questo punto, si popola il nuovo nodo e lo si inserisce all'interno della lista:

```
strcpy(nuovo->nome, nome);
strcpy(nuovo->descrizione, descrizione);
strcpy(nuovo->corso_appartenenza, corso_appartenenza);
nuovo->data_scadenza=data_scadenza;
nuovo->priorita=priorita;
nuovo->t_stimato=t_stimato;
nuovo->t_trascorso=t_trascorso;
nuovo->successivo=l;
```

10- Se il processo di lettura, validazione, allocazione e copia è andato a buon fine, la funzione scrive un messaggio di successo sul file di output e viene restituito il puntatore al nuovo nodo, che ora è la testa della lista aggiornata:

```
fprintf(tc_output, "\n\nNuova attivita' aggiunta con successo!\n\n");

return nuovo;
```

3.3.3 Funzione test_rimuovi_attivita:

La funzione legge dal file di input tc_input il nome di un'attività da rimuovere dalla lista collegata delle attività, scelta dall'utente. Partendo dalla testa della lista, scorre i nodi cercando quello il cui campo nome corrisponde alla stringa letta. Se l'attività viene trovata, la funzione rimuove il nodo dalla lista aggiornando i puntatori adeguatamente, libera la memoria allocata per il nodo e i relativi campi dinamici, e scrive un messaggio di successo sul file di output. Se invece non viene trovata alcuna attività con il nome specificato, la lista resta invariata e viene segnalato l'errore sul file di output.

1. Vengono dichiarati i puntatori corrente e precedente per scorrere la lista e un array per memorizzare il nome dell'attività da eliminare, letto dal file di input:

```
struct attivita *corrente=l, *precedente=NULL;
char nome[50];
```

2. Il nome dell'attività viene letto, rimuovendo il carattere di newline. Se la lettura fallisce, viene scritto un messaggio d'errore e la funzione restituisce la lista invariata:

```
//Legge il nome dell'attività da rimuovere dal file di input.
if(fgets(nome, sizeof(nome), tc_input) == NULL){
    fprintf(tc_output, "Errore nella lettura del nome dell'attività da eliminare.\n");
    return l;
}
// Rimuovere il carattere di newline, se presente.
nome[strcspn(nome, "\n")]=0;

fprintf(tc_output, "Tentativo di eliminare l'attività: %s\n", nome);
```

3. Con un ciclo viene controllato se il campo nome del nodo corrente corrisponde alla stringa letta. Se il nodo corrispondente viene trovato, si distingue:

-il caso in cui il nodo da eliminare è il primo della lista, dove la testa della lista viene aggiornata;

-il caso in cui il nodo si trova in mezzo alla lista o in ultima posizione, dove il puntatore successivo del nodo precedente viene aggiornato per saltare il nodo da rimuovere.

In entrambi i casi, viene scritto un messaggio di successo sul file di output:

```
while(corrente != NULL){
    if(strcmp(corrente->nome, nome) == 0){
        if(precedente == NULL){
            //L'attività da eliminare è la prima della lista.
            l=corrente->successivo;
        } else {
            //L'attività da eliminare è in mezzo alla lista o l'ultima.
            precedente->successivo=corrente->successivo;
        }

        free(corrente->nome);
        free(corrente->corso_appartenenza);
        free(corrente->descrizione);
        free(corrente);

        fprintf(tc_output, "\n\nAttività correttamente rimossa.\n\n\n");
        return l; //Lista aggiornata.
    }

    precedente=corrente;
    corrente=corrente->successivo;
}
```

4. Nel caso in cui il ciclo termini senza trovare l'attività, viene scritto un messaggio d'errore su tc_output e la lista originale viene restituita:

```
fprintf(tc_output, "\n\nAttività non trovata in programma.\n\n\n");
return l; //Lista originale.
```

3.3.4 Funzione test_aggiornamento_progresso:

La funzione aggiorna il progresso (cioè, t_trascorso) di un'attività presente nella lista, utilizzando i dati letti da un file di input. Se il tempo trascorso è maggiore o uguale al tempo stimato, la funzione richiede, tramite ulteriori letture da tc_input, se l'utente intende aggiornare il tempo stimato, digitando 1, oppure se considerare l'attività completata. In quest'ultimo caso, viene letto il nuovo tempo stimato, si aggiorna il campo t_stimato e si ricalcola la percentuale.

1. Vengono dichiarati un puntatore per scorrere la lista, delle variabili per memorizzare i nuovi dati letti dal file di input e, eventualmente, la scelta dell'utente ed un array di caratteri per memorizzare il nome dell'attività da aggiornare:

```
struct attivita *corrente;
int t_trascorso_agg, val, t_stimato_agg, percentuale;
char nome[50];
```

2. Si utilizza fgets per leggere il nome dell'attività da tc_input. Viene controllato il corretto risultato della lettura; in caso di errore la funzione stampa un messaggio su tc_output e restituisce 0. Subito dopo, viene effettuata la lettura di un intero da tc_input e il valore viene assegnato a t_trascorso_agg. Analogamente, se la lettura non va a buon fine, viene segnalato l'errore e la funzione restituisce 0:

```
// Legge il nome dell'attività dal file di input.
if(fgets(nome, sizeof(nome), tc_input) == NULL){
    fprintf(tc_output, "Errore nella lettura del nome dell'attività!\n");
    return 0;
}
nome[strcspn(nome, "\n")]=0;

// Legge il tempo trascorso aggiornato dal file di input.
if(fscanf(tc_input, "%d", &t_trascorso_agg) != 1){
    fprintf(tc_output, "Errore nella lettura del tempo trascorso!\n");
    return 0;
}
```

3. La funzione inizia il ciclo per la ricerca del nodo della lista con il nome corrispondente a quello letto dal file di input. Quando lo trova, aggiorna il campo t_trascorso:

```
for(corrente=l; corrente != NULL; corrente=corrente->successivo){
    if(strcmp(corrente->nome, nome) == 0){
        corrente->t_trascorso=t_trascorso_agg;
```

4. Se il tempo stimato (campo t_stimato) risulta maggiore del nuovo tempo trascorso, si calcola la percentuale di progresso e la si scrive su tc_output.

```
if(corrente->t_stimato > corrente->t_trascorso){
    percentuale= (corrente->t_trascorso * 100) / corrente->t_stimato;
    fprintf(tc_output, "\nLa tua percentuale di progresso è %d %%\n\n", percentuale);
```

5. Se il tempo trascorso è maggiore o uguale al tempo stimato, la funzione informa l'utente (tramite tc_output) e richiede di verificare se aggiornare il tempo stimato. Viene letta la scelta tramite fscanf e, se la scelta è 1, si procede alla lettura del nuovo tempo stimato (t_stimato_agg), aggiornando quindi anche il campo t_stimato e ricalcolando la percentuale. Se la scelta è diversa da 1, l'attività viene considerata completa e si stampa un messaggio di congratulazioni per il completamento:

```
} else {
    fprintf(tc_output, "\nIl tempo trascorso è maggiore o uguale al tempo stimato.\n");
    fprintf(tc_output, "Digita 1 se vuoi aggiornare il tempo stimato, altrimenti l'attività sarà considerata completata:\n");

    if(fscanf(tc_input, "%d", &val) != 1){
        fprintf(tc_output, "Errore nella lettura della scelta!\n");
        return 0;
    }

    if(val == 1){
        fprintf(tc_output, "\n\nInserisci il nuovo tempo stimato (in ore):\n");

        if(fscanf(tc_input, "%d", &t_stimato_agg) != 1){
            fprintf(tc_output, "Errore nella lettura del tempo stimato aggiornato!\n");
            return 0;
        }

        corrente->t_stimato=t_stimato_agg;
        fprintf(tc_output, "\nTempo stimato aggiornato!\n");

        if(corrente->t_stimato > 0){
            percentuale= (corrente->t_trascorso * 100) / corrente->t_stimato;
            fprintf(tc_output, "La tua percentuale di progresso è %d %%\n\n", percentuale);
        }
    } else {
        fprintf(tc_output, "\n\nComplimenti, hai completato l'attività con successo!\n\n");
```

In entrambi i casi riusciti, la funzione restituisce 1.

6. In caso l'attività non venga trovata, viene scritto un messaggio sul file di output e la funzione restituisce 0:

```
fprintf(tc_output, "\nAttività non trovata.\n\n");
return 0;
```

3.3.5 Funzione ordina_per_priorita:

La funzione riordina una lista di attività in base al campo "priorità" in ordine decrescente (dalla priorità 3 alla priorità 1). Utilizza l'algoritmo MergeSort, delegando il compito alla funzione mergesort_lista, che è responsabile dell'ordinamento vero e proprio della lista. Infatti, la funzione si compone di una sola riga di codice, dove è presente la chiamata a mergesort_lista.

```
void ordina_per_priorita(lista_att *l){
    *l=mergesort_lista(*l);
}
```

3.3.6 Funzione scadenza_att:

La funzione determina se un'attività è scaduta confrontando due strutture di tipo struct data, ossia data_scadenza e fine_sett, che rappresenta la fine della settimana inserita nella funzione report_settimanale. Viene restituito 1, se la data di scadenza è precedente alla data di fine settimana (quindi l'attività è scaduta), oppure 0, in caso contrario.

1. Si compone di una serie di confronti effettuati con delle istruzioni if di selezione:

-Se data_scadenza.anno è inferiore a fine_sett.anno, ciò implica che la scadenza è avvenuta in un anno precedente; quindi, l'attività è considerata scaduta e la funzione restituisce 1;

-Se gli anni coincidono, il confronto passa al mese: se il mese di data_scadenza è inferiore a quello di fine_sett, l'attività è scaduta e la funzione restituisce 1;

-Se gli anni e i mesi sono uguali, il confronto si sposta ai giorni: se il giorno di data_scadenza è minore di quello di fine_sett, l'attività risulta scaduta e la funzione restituisce 1.

```
if(data_scadenza.anno < fine_sett.anno) return 1;
else if(data_scadenza.anno == fine_sett.anno){
        if(data_scadenza.mese<fine_sett.mese)
        return 1;
        else if(data_scadenza.mese == fine_sett.mese){
                if(data_scadenza.giorno < fine_sett.giorno)
                return 1;
```

2. Altrimenti significa che la data di scadenza non è antecedente a quella di fine settimana. Dunque, funzione restituisce 0, indicando che l'attività non è scaduta:

```
    }
return 0;
```

3.3.7 Funzione test_report_settimanale:

La funzione genera un report settimanale dettagliato delle attività contenute in una lista, ordinandole per priorità in ordine decrescente, per garantire che le attività di maggiore priorità siano visualizzate per prime. Legge dal file di input la data di inizio settimana e la data di fine settimana, validandone l'inserimento (controllando che il giorno sia compreso tra 1 e 31 e il mese tra 1 e 12). Se la lista risulta vuota, viene chiesto tramite tc_input all'utente se desidera inserire una nuova attività; in caso positivo, viene invocata la funzione test_aggiungi_attivita, aggiornando la lista. Per ogni attività presente nella lista che è stata inserita entro il giorno di fine settimana, la funzione ne stampa i dettagli su tc_output e verifica lo stato dell'attività (scaduta\completa\in corso) tramite la funzione scadenza_att.

1. La funzione inizia ordinando la lista in modo che le attività con priorità maggiore (valore più alto) siano in testa, chiamando la funzione ordina_per_priorita. Il puntatore corrente viene inizializzato alla testa della lista, per servirsi nella successiva iterazione e vengono dichiarate le variabili locali necessarie:

```
ordina_per_priorita(l);
struct attivita *corrente=*l;
struct data inizio_sett, fine_sett;
int val, scadenza, percentuale;
```

2. Viene letta la data di inizio settimana dal file tc_input, scritta in formato gg-mm-aaaa. Se i valori non rispettano i range previsti o la lettura non va a buon fine, la funzione scrive un messaggio d'errore su tc_output e termina l'esecuzione del report. In maniera analoga, viene letta la data di fine settimana:

```
if(fscanf(tc_input, "%d-%d-%d", &inizio_sett.giorno, &inizio_sett.mese, &inizio_sett.anno) != 3 ||
    inizio_sett.giorno<1 || inizio_sett.giorno>31 ||
    inizio_sett.mese<1 || inizio_sett.mese>12){
    fprintf(tc_output, "Data di inizio settimana non valida!\n");
    return;
}


if(fscanf(tc_input, "%d-%d-%d", &fine_sett.giorno, &fine_sett.mese, &fine_sett.anno) != 3 ||
    fine_sett.giorno<1 || fine_sett.giorno>31 ||
    fine_sett.mese<1 || fine_sett.mese>12){
    fprintf(tc_output, "Data di fine settimana non valida!\n");
    return;
}
```

3. Se la lista è vuota, la funzione chiede all'utente se desidera aggiungerne una. L'utente immette la scelta tramite tc_input e, se la risposta è affermativa (= 1), viene invocata la funzione test_aggiungi_attivita che aggiorna la lista. Se l'utente non desidera aggiungere una nuova attività, la funzione termina:

```c
if(corrente == NULL){  //Se la lista è vuota.
    fprintf(tc_output, "\nNessuna attività trovata per questa settimana!\n");
    fprintf(tc_output, "Digita 1 se vuoi inserirne una adesso, altrimenti 0 per uscire:\n");

    if(fscanf(tc_input, "%d", &val) != 1){
        fprintf(tc_output, "Errore nella lettura della scelta.\n");
        return;
    }

    if(val == 1){
        *l=test_aggiungi_attivita(*l, tc_input, tc_output);
        corrente=*l;
    } else {
        return;
    }
}
```

4. Inizia la stampa del report settimanale, segnalata da un messaggio scritto sul file di output. Per ogni attività presente, la funzione controlla se la data di inserimento è antecedente o uguale alla data di fine settimana. In tal caso, vengono stampati su `tc_output tutti i dettagli dell'attività:

```c
fprintf(tc_output, "\n*****************************************************************************************************");
fprintf(tc_output, "\nEcco un report settimanale delle attività inserite entro questa settimana, ordinate per priorità maggiore:\n");
fprintf(tc_output, "*****************************************************************************************************\n\n");

while(corrente != NULL){
   // Se l'attività è stata aggiunta entro il giorno di fine settimana, stampala.
   if(corrente->data_inserimento.anno < fine_sett.anno ||
       (corrente->data_inserimento.anno == fine_sett.anno && corrente->data_inserimento.mese < fine_sett.mese) ||
       (corrente->data_inserimento.anno == fine_sett.anno && corrente->data_inserimento.mese == fine_sett.mese &&
        corrente->data_inserimento.giorno <= fine_sett.giorno)){


       fprintf(tc_output, "-Nome: %s\n", corrente->nome);
       fprintf(tc_output, "-Corso di appartenenza: %s\n", corrente->corso_appartenenza);
       fprintf(tc_output, "-Descrizione: %s\n", corrente->descrizione);
       fprintf(tc_output, "-Data di scadenza: %02d-%02d-%04d\n", corrente->data_scadenza.giorno, corrente->data_scadenza.mese, corrente->c
       fprintf(tc_output, "-Data di inserimento: %02d-%02d-%04d\n", corrente->data_inserimento.giorno, corrente->data_inserimento.mese, cc
       fprintf(tc_output, "-Priorità: %d\n", corrente->priorita);
       fprintf(tc_output, "-Tempo stimato: %d ore\n", corrente->t_stimato);
       fprintf(tc_output, "-Tempo trascorso: %d ore\n", corrente->t_trascorso);
```

5. Viene invocata la funzione scadenza_att per determinare se l'attività sia scaduta, completata o in corso. In base al valore ritornato da questa, vengono stampati su `tc_output messaggi specifici (scadenza, percentuale di completamento o conferma del completamento):

```c
scadenza=scadenza_att(corrente->data_scadenza, fine_sett);

if(scadenza == 1){
    fprintf(tc_output, "L'attività è scaduta!\n\n");
} else if(corrente->t_stimato == corrente->t_trascorso || corrente->t_stimato == 0){
    fprintf(tc_output, "L'attività è stata completata con successo!\n\n");
} else{
    percentuale= (corrente->t_trascorso * 100) / corrente->t_stimato;
    fprintf(tc_output, "L'attività è in corso, con una percentuale di completamento: %d %%!\n\n", percentuale);
}
```

3.3.8 Funzione test_rimuovi_attivita:

La funzione confronta il contenuto di due file, il file oracolo e il file di output, che viene prodotto dal programma. Il confronto avviene riga per riga, rimuovendo dai buffer i caratteri di fine linea (\r e \n). Se tutte le righe dei due file sono identiche, compresa l'assenza di righe in eccesso

in uno dei due file, il test viene considerato "SUCCESSO"; altrimenti, viene considerato "FALLIMENTO". Successivamente, la funzione estrae la parte iniziale del nome del file oracolo (fino al carattere di underscore “_”) e, in modalità append, scrive una riga nel file indicato dal parametro “risultato”, riportando il nome del test e il risultato (SUCCESSO o FALLIMENTO).

1. La funzione apre il file oracolo in modalità lettura, utilizzando il nome passato tramite il parametro file_oracolo ed apre il file di output generato dal programma in modalità lettura. Se uno dei due file non viene aperto correttamente, viene stampato un messaggio d’errore sullo standard output e l’esecuzione della funzione viene terminata:

```
FILE *f_oracolo=fopen(file_oracolo, "r");
FILE *f_output=fopen("TC_OUTPUT.txt", "r");

if(!f_oracolo || !f_output){
    printf("Errore nell'apertura dei file oracolo o output.\n");
    if (f_oracolo) fclose(f_oracolo);
    if (f_output) fclose(f_output);
    return;
}
```

2. La funzione utilizza un ciclo while(1) per leggere le righe dai due file. Per ogni iterazione, le righe vengono lette in buffer e vengono rimossi i caratteri di fine linea.

```
//Confronto riga per riga.
int successo=1;
char buf_oracolo[1024], buf_output[1024];
while (1){
    char *riga_oracolo=fgets(buf_oracolo, sizeof(buf_oracolo), f_oracolo);
    char *riga_output=fgets(buf_output, sizeof(buf_output), f_output);
```

3. Se le due righe non corrispondono, la variabile successo viene impostata a 0 e il ciclo viene interrotto. Se le due letture terminano contemporaneamente, il ciclo viene terminato con successo invariato. Altrimenti, se solo uno dei due file termina, il test fallisce:

```
//Se entrambe le letture hanno successo, confronta le stringhe.
if (riga_oracolo && riga_output){

    buf_oracolo[strcspn(buf_oracolo, "\r\n")]='\0';
    buf_output[strcspn(buf_output, "\r\n")]='\0';

    if(strcmp(buf_oracolo, buf_output) != 0){
        successo=0;
        break;
    }
}
//Se entrambe hanno finito contemporaneamente, esci.
else if(!riga_oracolo && !riga_output){
    break;
}
//Se uno solo dei due file è terminato, i file sono differenti.
else {
    successo=0;
    break;
}
```

4. Al termine del confronto, entrambi i file aperti in lettura vengono chiusi, liberando le risorse:

```
fclose(f_oracolo);
fclose(f_output);
```

5. Ora avvien l'estrazione della parte iniziale del nome del file oracolo: la funzione usa sscanf per estrarre dalla stringa file_oracolo la porzione iniziale fino al primo underscore "_", poiché questa parte rappresenta generalmente il nome del test e viene usata per scrivere il risultato nel file di output finale:

```
char tc[20];
sscanf(file_oracolo, "%[^_]", tc);
```

6. Il file specificato dal parametro "risultato" viene aperto in modalità append. Se l'apertura fallisce, viene stampato un messaggio d'errore sullo standard output e la funzione termina, senza modificare il file. In caso di apertura corretta, la funzione scrive su "risultato" una riga che contiene il nome del test (estratto dal file oracolo) seguito dal risultato del confronto, cioè "SUCCESSO" o "FALLIMENTO". Infine, il file viene chiuso:

```
FILE *f_risultato=fopen(risultato, "a");
if(f_risultato == NULL){
    printf("Errore nell'apertura del file!\n");
    return;
}

fprintf(f_risultato, "%s: %s\n", tc, successo ? "SUCCESSO" : "FALLIMENTO");
fclose(f_risultato);
```

3.3.9 Funzione libera_memoria:

La funzione dealloca tutta la memoria dinamica occupata dalla lista di attività, prevenendo così memory leak, prima della conclusione dell'esecuzione del programma. Questo processo è essenziale per una gestione corretta delle risorse che si servono di memoria dinamica.

1. Viene inizializzato il puntatore corrente con il valore attuale di *l, che rappresenta la testa della lista:

```
struct attivita *corrente=*l;
```

2. Il ciclo while viene eseguito finché non si raggiunge la fine della lista e, al suo interno, il nodo corrente viene assegnato temporaneamente a temp per conservarne il riferimento, per permettere di accedere a tutte le informazioni del nodo corrente anche dopo aver aggiornato il puntatore di iterazione:

```
while(corrente!=NULL){
struct attivita *temp=corrente;
```

3. Il puntatore corrente viene aggiornato per puntare al nodo successivo, sostituendo il nodo attuale con quello seguente e viene liberata la memoria allocata per ciascun campo stringa dinamico nome, descrizione, corso_appartenenza appartenente al nodo. Successivamente, viene deallocato il nodo stesso:

```
corrente=corrente->successivo;
free(temp->nome);
free(temp->descrizione);
free(temp->corso_appartenenza);
free(temp);
}
```

4. Una volta completato il ciclo e liberata tutta la memoria relativa ai nodi della lista, il puntatore passato per riferimento viene impostato a NULL. Ciò impedisce possibili accessi futuri a memoria ormai liberata, evitando errori come i dangling pointer (i puntatori pendenti):

```
*l=NULL; //Evita futuri accessi non validi.
```

3.4 FILE MAIN.C:

Il file main.c costituisce il cuore del programma che gestisce l'interazione con l'utente tramite un menu testuale. Attraverso questo, l'utente può aggiungere nuove attività, rimuovere attività esistenti, aggiornare il progresso di un'attività o generare report settimanali delle attività, tramite la lettura della sua scelta. Ad ogni scelta, viene invocata la funzione corrispondente, che opera sulla struttura dati della lista delle attività. Prima di terminare, infine, il programma libera la memoria allocata per la lista e termina.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "attivita.h"
```

All'inizio del file, vengono incluse le librerie standard necessarie per l'I/O, la gestione della memoria e le stringhe. Il file header "attivita.h" contiene le dichiarazioni delle funzioni insieme alle definizioni delle strutture dati lista_att e data, senza le quali il main non saprebbe come agire e segnalerebbe errore, qualora le incontrasse.

Osservazione: le funzioni che vengono chiamate nel main possono avere, nel loro corpo, delle chiamate a funzioni contenute nell'header file "mergesortList.h". Quest'ultimo non è incluso nel main poiché non è usato in maniera esplicita da esso, bensì da funzioni definite nel file "attivita.h", che contiene già l'inclusione per "mergesortList.h".

```
int main(void){
    lista_att l=nuova_lista_att();
    int scelta;
    char c;
```

La funzione main inizia dichiarando e creando una lista di attività, inizialmente vuota, grazie alla chiamata alla funzione nuova_lista_att. Vengono anche dichiarate una variabile per la memorizzazione della scelta dell'utente e un'altra usata per pulire il buffer di input dopo la lettura della scelta.

```c
do{

    printf("************************************");
    printf(" Pianifica e monitora il tuo studio!\n");
    printf("************************************\n");
    printf("1. Aggiungi nuova attivita'.\n");
    printf("2. Elimina un'attivita'.\n");
    printf("3. Aggiorna il progresso di un'attivita'.\n");
    printf("4. Stampa un report settimanale delle tue attivita'.\n");
    printf("0. Per uscire dal programma.\n");
    printf("Inserisci il numero corrispondente alla tua scelta: ");
    scanf("%d", &scelta);
```

Viene subito stampato a video il menu principale, elencando le opzioni disponibili (aggiunta, eliminazione, aggiornamento, report o uscita). La scelta dell'utente viene letta tramite scanf.

```c
while((c=getchar())!='\n');
#ifdef _WIN32
    system("cls");
#else
    system("clear");
#endif
```

Successivamente, un ciclo while svuota il buffer di input dai caratteri residui (come il carattere di newline) e viene effettuato un comando di sistema per pulire lo schermo, differenziando il comando tra Windows ("cls") e sistemi Linux/MacOS) ("clear"), mediante la direttiva #ifdef _WIN32. Questa è un'opzione che fa si che il programma sia portabile ed eseguibile su elaboratori con differenti sistemi operativi.

```c
switch(scelta){
    case 1: l=aggiungi_attivita(l);
            break;
    case 2: l=rimuovi_attivita(l);
            break;
    case 3: aggiornamento_progresso(l);
            break;
    case 4: report_settimanale(&l);
            break;
    case 0: printf("Chiusura programma in corso... \n");
            break;
    default: printf("Scelta non valida, riprova.\n");
}
```

Attraverso un costrutto switch, il valore di "scelta" viene confrontato con le varie etichette presenti: a seconda di esso, viene invocata la funzione corrispondente all'azione che l'utente ha scelto. Se è stato inserito un valore non presente nelle etichette, si applica il caso di default, ossia viene chiesto all'utente di inserire un nuovo valore, facendo ristampare il menu con le scelte.

Attenzione: ciò avviene fintanto che il valore di "scelta" è diverso da 0. Se risulta essere uguale a 0, il programma terminerà, come indicato dall'etichetta "case 0".

```
//Pulizia schermo prima della prossima scelta dell'utente.
printf("\nPremi un tasto per continuare...\n");
getchar();
printf("\033[H\033[J");
```

Dopo l'esecuzione del comando scelto, viene chiesto all'utente di premere un tasto per continuare. Successivamente, si pulisce lo schermo con il codice escape ANSI.

```
  } while(scelta!=0);

  libera_memoria(&l);
  return 0;
```

Quando l'utente sceglie 0, il ciclo termina, viene chiamata la funzione libera_memoria per deallocare tutta la memoria dinamica associata alla lista e, infine, il programma termina con return 0.

3.5 FILE MAINTEST.C:

Il file maintest.c rappresenta il fulcro per il testing automatico dell'applicazione di gestione dello studio. La funzione main in questo file riceve come argomenti della linea di comando i nomi dei file, di cui: uno di input (tc_input) contenente le istruzioni/test da eseguire, un file oracolo, contenente l'output atteso dal programma, ed un file di risultato, su cui verrà scritto l'esito finale del test.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "attivitatest.h"
```

All'inizio del file, analogamente al file main.c, vengono incluse le librerie standard necessarie per l'I/O, la gestione della memoria e le stringhe. Il file header "attivitatest.h" contiene le dichiarazioni delle funzioni insieme alle definizioni delle strutture dati lista_att e data, senza le quali il main non saprebbe come agire e segnalerebbe errore, qualora le incontrasse.

Osservazione: anche in questo caso le funzioni che vengono chiamate nel main possono avere, nel loro corpo, delle chiamate a funzioni contenute nell'header file "mergesortList.h". Quest'ultimo non è incluso nel main poiché non è usato in maniera esplicita da esso, bensì da funzioni definite nel file "attivitatest.h", che contiene già l'inclusione per "mergesortList.h".

```
int main(int argc, char *argv[]){
    if(argc < 4){
        printf("Errore: nessun file specificato!\n");
        return 1;
    }
```

La funzione main accetta gli argomenti della linea di comando. Se il numero di argomenti è inferiore a 4, viene stampato un messaggio d'errore e il programma termina, poiché i file necessari (tc_input, oracolo, risultato) non sono stati specificati.

```
FILE *tc_input = fopen(argv[1], "r");
FILE *oracolo = fopen(argv[2], "r");
FILE *risultato = fopen(argv[3], "r+");
FILE *tc_output = fopen("TC_OUTPUT.txt", "w"); //Creazione automatica del file di output.
```

Vengono aperti i file usando i nomi passati come argomenti:

-tc_input: file di input per i test;

-oracolo: file contenente l'output atteso;

-risultato: file in cui scrivere il risultato finale;

-tc_output: file temporaneo di output che verrà creato al momento dell'esecuzione.

```
if(tc_input == NULL){
    printf("Errore nell'apertura del file di input: %s\n", argv[1]);
    return 1;
}

if(oracolo == NULL){
    printf("Errore nell'apertura del file oracolo: %s\n", argv[2]);
    fclose(tc_input);
    return 1;
}

if(tc_output == NULL){
    printf("Errore nella creazione del file di output TC_OUTPUT.txt.\n");
    fclose(tc_input);
    fclose(oracolo);
    return 1;
}

if(risultato == NULL){
    printf("Errore nell'apertura del file %s.\n", argv[3]);
    fclose(tc_input);
    fclose(oracolo);
    fclose(tc_output);
    return 1;
}
```

Per ogni file, vengono eseguiti controlli di apertura e, in caso di errore, i file già aperti vengono chiusi e il programma termina.

```
lista_att l = nuova_lista_att();
int scelta;
```

Vengono dichiarate una variabile per memorizzare la scelta letta dal file di input e una lista, già inizializzata come vuota, grazie alla chiamata alla funzione nuova_lista_att.

```c
while(fscanf(tc_input, "%d", &scelta) == 1){

    int ch;
    while ((ch = fgetc(tc_input)) != '\n' && ch != EOF);
```

Il programma legge il valore della scelta dal file tc_input in un ciclo while. Dopo la lettura viene pulito il buffer corrente per rimuovere gli eventuali caratteri residui.

```c
    switch(scelta) {
        case 1: l = test_aggiungi_attivita(l, tc_input, tc_output);
                break;
        case 2: l = test_rimuovi_attivita(l, tc_input, tc_output);
                break;
        case 3: test_aggiornamento_progresso(l, tc_input, tc_output);
                break;
        case 4: test_report_settimanale(&l, tc_input, tc_output);
                break;
        case 0: fprintf(tc_output, "Chiusura programma in corso...\n");
                break;
        default: fprintf(tc_output, "Scelta non valida, riprova.\n");
    }
```

Proprio per emulare al meglio il file main.c, in base alla scelta, viene invocata la funzione corrispondente e, in caso di scelte non valide, viene stampato un messaggio di errore nel file di output.

```c
 // Chiudiamo il file di output per assicurarci che tutti i dati siano scritti.
fclose(tc_output);


verifica_test(argv[2], argv[3]);
```

Prima di confrontare i risultati, il file tc_output viene chiuso per assicurarsi che tutti i dati siano stati scritti. La funzione verifica_test viene invocata usando i nomi dei file oracolo e risultato (aperto in modalità "r+"): essa confronterà riga per riga il contenuto atteso con quello ottenuto e scriverà il risultato del confronto nel file risultato.

Questo rappresenta il punto più importante del nostro maintest.c, interpretando pienamente il significato di “testing automatico”.

```c
 fclose(tc_input);
 fclose(oracolo);
 fclose(risultato);

 libera_memoria(&l);

 return 0;
```

Successivamente, vengono chiusi i file tc_input, oracolo e risultato e, infine, viene liberata la memoria riservata per la lista di attività tramite la chiamata libera_memoria(&l).

Il programma termina con return 0.

4. SPECIFICA SINTATTICA E SEMANTICA DELLE FUNZIONI IMPLEMENTATE

Quando si sviluppa un software, la prima fase riguarda l'analisi e la specifica: qualsiasi sia l'uso al quale esso è destinato, occorre inserire nella sua documentazione le specifiche sintattiche e semantiche, con la relativa progettazione, dell'insieme di tutte le funzioni progettate affinché l'applicazione funzioni nella sua interezza.

La **specifica sintattica** si riferisce, come suggerito dal nome, alla struttura e alla forma del codice, definendo i tipi di dato accettati come parametro e restituiti da una determinata funzione, con annesso un elenco generale di essi. È importante individuare i possibili errori sintattici poiché potrebbero impedire al compilatore o all'interprete di eseguire il codice. Generalmente, la forma che ha una specifica sintattica è:

nome_funzione (tipo_parametro, tipo_parametro) -> tipo_restituito

Tipi: … (elenco dei tipi di dato presenti)

(Riferimento impostazione: libro "Programmazione in C" di King)

La **specifica semantica**, invece, fa riferimento al significato e al comportamento del codice. Essa si compone dell'elenco dei side-effect, ossia delle modifiche che vengono fatte sullo stato del programma, di precondizione, ossia delle condizioni definite sui dati di input che devono essere soddisfatte affinché sia possibile l'esecuzione, e di postcondizione, ossia delle condizioni definite sui dati di input in funzione dei dati di output, che devono essere soddisfatte al termine dell'esecuzione del programma. Se la precondizione è vera ed il programma è eseguito, allora la postcondizione deve essere valida. Gli errori semantici permettono comunque al codice di compilare ed eseguire ma, tuttavia, producono dei risultati errati o inaspettati. Una struttura d'esempio per la specifica semantica è:

nome_funzione (nome_parametro, nome_parametro) ->nome_ variabile_restituita

Spiegazione di che cosa fa la funzione

Side-effect: … (effetti collaterali)

Precondizione: … (descrizione condizione sui dati di input)

Postcondizione: (descrizione condizione sui dati di input in funzione di quelli di output)

(Riferimento impostazione: libro "Programmazione in C" di King)

Inoltre, buona norma consiste nell'usare un **dizionario dei dati**, ossia una tabella che descriva, per ogni variabile dichiara nella funzione, identificatore, tipo di dato e descrizione (eventualmente, con il fine di quest'ultima).

Identificatore	*Tipo*	*Descrizione*

Verranno analizzate le funzioni usate nel Main del software principale, descritte nell'header file attivita.h, le funzioni presenti nell'header file mergesortList.h e, successivamente, le funzioni progettate per il software redatto al testing automatico, descritte nell'header file attivitatest.h.

4.1 FUNZIONI DEL FILE ATTIVITA.C:

4.1.1 Funzione nuova_lista_att:

SPECIFICA SINTATTICA: nuova_lista_att(void) -> lista_att

Tipi: lista_att.

SPECIFICA SEMANTICA: nuova_lista_att () -> lista_att

La funzione crea e restituisce una nuova lista di attività vuota.

-*Side-effect*: Nessuno.

-*Precondizione*: Nessuna, la funzione può essere chiamata in qualsiasi momento;

-*Postcondizione*: Restituisce una lista vuota (NULL).

4.1.2 Funzione aggiungi_attivita:

SPECIFICA SINTATTICA: aggiungi_attivita(lista_att) -> lista_att

Tipi: lista_att.

SPECIFICA SEMANTICA: aggiungi_attivita(l) -> nuovo

La funzione alloca dinamicamente una nuova attività, leggendo i dati dall'utente e verificando la validità dei campi data_inserimento, data_scadenza e priorità. Inizialmente, controlla che sia stato allocato dinamicamente efficientemente il nuovo nodo della lista, che rappresenta la nuova attività che si sta inserendo. In caso di allocazione fallita per i campi nome, descrizione e corso_appartenenza, ne gestisce la deallocazione della memoria, stampando un messaggio di errore per l'utente; altrimenti aggiunge l'attività alla lista esistente.

-*Side-effect*: Allocazione dinamica della memoria per il nuovo nodo e per le stringhe dei campi nome, descrizione, corso_appartenenza, eventuale deallocazione in caso di errore, scrittura sul file di output dei messaggi di feedback, modifica della struttura della lista.

-*Precondizione*: l è una lista di attività valida (che può essere anche vuota). L'input dell'utente deve rispettare i vincoli richiesti per i vari campi: formato corretto delle date e valore di priorità compreso tra 1 e 3.

-*Postcondizione*: Se l'allocazione dinamica fallisce, la funzione deve restituire NULL e stampare un messaggio di errore. Se tutti gli input dell'utente sono validi, deve essere allocata una nuova attività che sarà collegata alla lista esistente e restituita come nuova testa della lista. Se l'input non è valido, deve essere richiesto nuovamente l'inserimento fino a ottenere un dato corretto, grazie all'iterazione dei cicli while.

DIZIONARIO DEI DATI:

Identificatore	*Tipo*	*Descrizione*

nuovo	Struct attivita*	Puntatore alla nuova attività allocata.
data_scadenza	Struct data	Data scadenza dell'attività.
d_inserimento	Struct data	Data inserimento dell'attività.
priorita	int	Valore priorità (1,2 o 3).
t_stimato	int	Tempo stimato per completare l'attività.
t_trascorso	int	Tempo già trascorso sull'attività.
nome	char[50]	Nome dell'attività.
descrizione	char[150]	Descrizione dell'attività.
corso_appartenenza	char[100]	Corso associato all'attività.
l	lista_att	Lista delle attività già esistenti.
successivo	Struct attivita*	Puntatore all'attività successiva della lista l.

4.1.3 Funzione rimuovi_attivita:

SPECIFICA SINTATTICA: rimuovi_attivita(lista_att) -> lista_att

Tipi: lista_att.

SPECIFICA SEMANTICA: rimuovi_attivita(l) -> l

La funzione chiede all'utente di immettere il nome dell'attività da rimuovere e la ricerca nella lista. Distingue i casi: lista vuota o piena (con i sotto casi: elemento da eliminare in prima posizione o in mezzo/in ultima posizione). A conclusione dell'operazione, stampa un messaggio di successo o insuccesso, a seconda del caso verificatosi. Restituisce la lista aggiornata, se è riuscita a trovare l'attività da eliminare e l'ha eliminata, vuota altrimenti.

-*Side-effect*: Modifica struttura della lista, deallocazione della memoria precedentemente allocata dinamicamente per il nodo da rimuovere.

-*Precondizioni*: l è una lista di attività valida (che può essere anche vuota). L'utente deve inserire un nome valido per la ricerca dell'attività.

-*Postcondizioni*: Se l'attività è presente, deve venire rimossa dalla lista e la lista aggiornata deve essere restituita. Se l'attività non è trovata, perché non presente o perché la lista è vuota, la lista originale deve essere restituita senza modifiche. In entrambi i casi, è stampato un messaggio per l'utente per informarlo riguardo l'esito dell'operazione di rimozione.

DIZIONARIO DEI DATI:

Identificatore	*Tipo*	*Descrizione*
l	lista_att	Lista di attività.
corrente	Struct attivita*	Puntatore all'elemento corrente della lista per poterla scorrere tutta.
precedente	Struct attivita*	Puntatore all'elemento precedente della lista.

nome	char[50]	Nome dell'attività da eliminare.

4.1.4 Funzione aggiornamento_progresso:

SPECIFICA SINTATTICA: aggiornamento_progresso(lista_att) -> int

Tipi: lista_att, int.

SPECIFICA SEMANTICA: aggiornamento_progresso(l)-> int

La funzione permette di aggiornare il campo t_trascorso di una specifica attività presente nella lista, cercata grazie all'input dell'utente, calcolando e stampando il progresso percentuale rispetto al campo t_stimato. Se il tempo trascorso aggiornato risulta maggiore o uguale al tempo stimato, la funzione fa scegliere all'utente se aggiornare il valore t_stimato oppure segnala il completamento dell'attività, stampando di conseguenza il nuovo valore percentuale.

-*Side-effect*: Modifica dei campi t_trascorso, ed eventualmente t_stimato, dell'attività selezionata.

-*Precondizione*: l è una lista di attività valida (che può essere anche vuota). L'utente deve immettere un nome valido per la ricerca dell'attività da aggiornare e valori int validi per t_trascorso_agg (e, se richiesto, per t_stimato_agg).

-*Postcondizione:* Se l'attività indicata viene trovata nella lista, il campo t_trascorso deve essere aggiornato e, se necessario, deve essere aggiornato anche il campo t_stimato, sempre su scelta dell'utente; in entrambi i casi, viene stampata la percentuale del progresso, calcolata implicitamente dalla funzione. Se l'attività viene trovata e ne è eseguito l'aggiornamento, la funzione deve restituire 1. Se l'attività non viene trovata, la funzione non deve modificare la lista e deve restituire 0.

DIZIONARIO DEI DATI:

Identificatore	*Tipo*	*Descrizione*
l	lista_att	Lista delle attività presenti.
corrente	Struct attivita *	Puntatore per scorrere la lista ed accedere alle attività.
nome	char[50]	Nome dell'attività da cercare.
t_trascorso_agg	int	Valore del tempo trascorso aggiornato immesso in input.
val	int	Valore della scelta dell'utente: aggiornamento di t_stimato o segnalazione del completamento di un'attività.
t_stimato_agg	int	Valore del tempo stimato aggiornato immesso in input.
percentuale	int	Percentuale del progresso di un'attività calcolata.

4.1.5 Funzione ordina_per_priorita:

SPECIFICA SINTATTICA: ordina_per_priorita(lista_att*) -> void

Tipi: lista_att*.

SPECIFICA SEMANTICA: ordina_per_priorita(l) -> void

La funzione riordina la lista di attività, puntata da l, sulla base del campo "priorità" in modo decrescente, ossia dalla priorità massima (=3) a quella minima (=1). Realizza l'ordinamento chiamando la funzione mergesort_lista, che implementa l'algoritmo MergeSort, presente nel file mergesortList.c.

-*Side-effect*: riordinamento della struttura della lista tramite l'algoritmo MergeSort.

-*Precondizione*: Il puntatore l deve essere inizializzato (anche se la lista è vuota). Le attività presenti nella lista, se esistenti, devono contenere un campo "priorità" valido.

-*Postcondizione:* La lista puntata da l deve essere modificata in modo tale da risultare ordinata in ordine decrescente secondo il campo "priorità". La funzione non deve restituisce alcun valore, poiché l'ordinamento è effettuato grazie alla sostituzione del puntatore alla lista con quello restituito dalla funzione mergesort_lista.

DIZIONARIO DEI DATI:

Identificatore	*Tipo*	*Descrizione*
l	lista_att*	Puntatore alla lista di attività da riordinare.
*l	lista_att	Lista delle attività che verrà riordinata.
Mergesort_lista	Funzione	Funzione che implementa l'algoritmo di ordinamento MergeSort.

4.1.6 Funzione scadenza_att:

SPECIFICA SINTATTICA: scadenza_att(struct data, struct data) -> int

Tipi: struct data, int.

SPECIFICA SEMANTICA: scadenza_att(data_scadenza, fine_sett) -> int

La funzione esegue una serie di confronti tra due strutture di tipo data per determinare se data_scadenza sia precedente ad una data limite, rappresentata dal campo fine_sett. Se l'attività risulta scaduta verrà restituito il valore 1, altrimenti 0.

-*Side-effect*: Nessuno.

-*Precondizione:* data_scadenza e fine_sett devono contenere valori validi (essere date corrette).

-*Postcondizione:* Se data_scadenza risulta precedente a fine_sett, l'attività è scaduta e la funzione dovrà restituire 1. Viceversa, l'attività sarà considerata in corso e la funzione restituirà 0.

DIZIONARIO DEI DATI:

Identificatore	*Tipo*	*Descrizione*
data_scadenza	struct data	Data di scadenza dell'attività da valutare.
fine_sett	struct data	Data limite per il confronto.
anno	int	Campo della struct data.
mese	int	Campo della struct data.
giorno	int	Campo della struct data.

4.1.7 Funzione report_settimanale:

SPECIFICA SINTATTICA: report_settimanale(lista_att*) -> void

Tipi: lista_att.

SPECIFICA SEMANTICA: report_settimanale(*l) -> void

La funzione chiama la funzione ordina_per_priorita per ordinarla in base al campo priorità. Successivamente, chiede all'utente di immettere, tramite due input, le date di inizio e di fine della settimana corrente, eseguendone il controllo di validità. Se la lista delle attività è vuota, viene data la possibilità di aggiungerne una nuova; in caso contrario, la funzione procede a stampare un report settimanale. Durante l'attraversamento della lista, per ogni attività viene verificato se è stata inserita entro la data di fine settimana. Per ogni attività stampata, ne vengono mostrati i campi (nome, corso di appartenenza, descrizione, date, priorità, tempo stimato e trascorso) e, usando il valore restituito dalla funzione scadenza_att, viene determinato se l'attività è scaduta, è stata completata o è in corso (con calcolo e stampa della percentuale di completamento).

-*Side-effect*: Modifica struttura della lista (ordinamento), eventuale aggiunta di un nodo su scelta dell'utente.

-*Precondizione*: Il puntatore l deve essere inizializzato (la lista passata può essere vuota oppure contenere una o più attività). Le strutture data_inserimento e data_scadenza devono contenere valori validi. L'utente deve immettere le date nel formato corretto (gg-mm-aaaa) con valori validi (giorno da 1 a 31 e mese da 1 a 12).

-*Postcondizione:* Se la lista è vuota, la funzione deve chiedere all'utente se vuole inserire una nuova attività, aggiornando eventualmente la lista. La funzione deve effettuare una serie di stampe a video, in ordine di priorità decrescente (grazie alla chiamata a ordina_per_priorita), delle attività inserite entro la data limite specificata (fine_sett). Per ogni attività stampata, deve esserne indicato lo stato tra scaduta, in corso o completata. Non deve restituire nessun valore.

DIZIONARIO DEI DATI:

Identificatore	*Tipo*	*Descrizione*

l	lista_att*	Puntatore alla lista delle attività.
corrente	Struct attivita*	Puntatore per scorrere gli elementi della lista.
inizio_sett	Struct data	Struttura per memorizzare il primo giorno della settimana.
fine_sett	Struct data	Struttura per memorizzare l'ultimo giorno settimana.
val	int	Valore per scelta dell'utente: aggiungere nuova attività in caso di lista vuota oppure uscire.
scadenza	int	Memorizza il valore restituito dalla funzione scadenza_att.
percentuale	int	Memorizza la percentuale di completamento di un'attività, calcolata implicitamente dalla funzione.

4.1.8 Funzione libera_memoria:

SPECIFICA SINTATTICA: libera_memoria(lista_att*) -> void

Tipi: lista_att*.

SPECIFICA SEMANTICA: libera_memoria(l) -> void

La funzione dealloca tutta la memoria dinamica occupata dagli elementi della lista di attività, prevenendo così possibili memory leak prima della fine della funzione main. Per ogni elemento nella lista, ne libera i campi dinamici (nome, descrizione, corso_appartenenza) e, infine, ne libera l'elemento stesso. Una volta completato il ciclo, il puntatore alla lista viene impostato a NULL per evitare futuri accessi non validi.

-*Side-effect*: Deallocazione memoria di ogni nodo della lista, incluse le stringhe allocate dinamicamente in ciascun nodo (nome, descrizione, corso di appartenenza), modifica puntatore della lista a NULL.

-*Precondizione*: Il puntatore l deve essere inizializzato (la lista può essere vuota o piena).

-*Postcondizione*: Tutta la memoria dinamica allocata per ogni attività della lista deve essere stata correttamente rilasciata. Il puntatore alla lista deve puntare a NULL, rendendo la lista non più accessibile.

DIZIONARIO DEI DATI:

Identificatore	*Tipo*	*Descrizione*
l	lista_att*	Puntatore alla lista da liberare.
corrente	struct attivita*	Puntatore per scorrere la lista.

temp	struct attivita*	Puntatore temporaneo per memorizzare l'attività corrente prima di deallocarla.
nome	char*	Campo dinamico della struct attivita da deallocare.
descrizione	char*	Campo dinamico della struct attivita da deallocare.
corso_appartenenza	char*	Campo dinamico della struct attivita da deallocare.

4.2 FUNZIONI DEL FILE MERGESORTLIST.C:

4.2.1 Funzione trova_meta:

SPECIFICA SINTATTICA: trova_meta(struct attivita*) -> struct attivita*

Tipi: struct attivita*.

SPECIFICA SEMANTICA: trova_meta(head) -> struct attivita*

La funzione trova e restituisce il punto medio della lista collegata. È il primo passo dell'algoritmo MergeSort (e delle sue chiamate ricorsive). Divide la lista in due metà: se la lista è vuota o contiene un solo elemento, la funzione restituisce immediatamente head; altrimenti, usa due puntatori (lento, che avanza di un nodo per volta, e veloce, che avanza di due nodi per volta) per individuare il nodo centrale.

-*Side-effect*: Nessuno.

-*Precondizione*: l è una lista di attività valida (che può essere anche vuota).

-*Postcondizione*: Se la lista contiene più di un elemento, la funzione deve restituire il puntatore all'elemento centrale (punto medio) della lista. Se la lista è vuota o contiene un solo elemento, deve essere restituito head.

DIZIONARIO DEI DATI:

Identificatore	*Tipo*	*Descrizione*
head	struct attivita*	Puntatore al primo elemento della lista.
lento	struct attivita*	Puntatore che avanza di un nodo alla volta, per individuare il punto medio della lista.
veloce	struct attivita*	Puntatore che avanza di due nodi alla volta, per individuare la fine della lista.

4.2.2 Funzione unione_liste:

SPECIFICA SINTATTICA: unione_liste(struct attivita*, struct attivita*) -> struct attivita*

Tipi: struct attivita*.

SPECIFICA SEMANTICA: unione_liste(lista1, lista2) -> struct attivita*

La funzione unisce ricorsivamente due liste ordinate in base al campo priorità, in ordine decrescente. Se una delle due liste è vuota, la funzione restituisce l'altra. Altrimenti, confronta il valore priorità dell'elemento iniziale di entrambe le liste e collega il nodo con la priorità maggiore al risultato della chiamata ricorsiva, ottenendo così infine una lista ordinata.

-*Side-effect*: Modifica dei campi puntatore “successivo” del nodo di lista1 oppure di lista2 con modifica consequenziale della struttura della lista.

-*Precondizione*: Le liste lista1 e lista2 devono essere state precedentemente ordinate in ordine decrescente rispetto al campo priorità. I puntatori delle liste possono anche essere NULL se la lista risulta vuota.

-*Postcondizione:* Deve essere restituita una lista contenente tutti gli elementi di lista1 e lista2, ordinati in maniera decrescente secondo priorità. L’unione deve avvenire utilizzando chiamate ricorsione di sé stessa sulle due sotto-liste.

DIZIONARIO DEI DATI:

Identificatore	*Tipo*	*Descrizione*
lista1	struct attivita*	Puntatore alla prima lista ordinata per priorità.
lista2	struct attivita*	Puntatore alla seconda lista ordinata per priorità.
priorita	int	Campo della struct attivita che rappresenta il valore di priorita.
successivo	struct attivita*	Puntatore nodo successivo nella lista collegata.

4.2.3 Funzione mergesort_lista:

SPECIFICA SINTATTICA: mergesort_lista(struct attivita*) -> struct attivita*

Tipi: struct attivita*.

SPECIFICA SEMANTICA: mergesort_lista(head) -> struct attività*

La funzione ordina ricorsivamente una lista collegata di elementi di tipo struct attivita in base al campo priorita, secondo l’algoritmo di ordinamento MergeSort. Se la lista è vuota o contiene un solo elemento, viene considerata già ordinata e la funzione restituisce immediatamente head. Altrimenti, la funzione individua il punto medio della lista tramite la funzione trova_meta e divide la lista in due metà. Queste due sotto liste vengono poi ordinate ricorsivamente tramite

due chiamate a mergesort_lista. Infine, le due liste ordinate vengono unite in una lista ordinata in maniera decrescente secondo priorita, tramite la funzione unione_liste.

-*Side-effect*: Modifica ricorsiva della struttura della lista in due sottoliste separate che vengono poi ricollegate per essere ordinate.

-*Precondizione*: La lista passata tramite il puntatore head deve essere inizializzata (può essere vuota) e, se contiene più di un elemento, gli elementi devono avere un valore valido per il campo priorita.

-*Postcondizione*: Deve essere restituita la lista head riordinata in modo tale che gli elementi siano disposti in ordine decrescente rispetto al campo priorita (con l'elemento con la priorità più alta in testa). Se la lista è vuota o contiene un solo elemento, allora deve essere restituita invariata.

DIZIONARIO DEI DATI:

Identificatore	*Tipo*	*Descrizione*
head	struct attivita*	Puntatore al primo elemento della lista di attività da ordinare.
meta	struct attivita*	Puntatore al nodo individuato come punto medio della lista, grazie alla funzione trova_meta.
seconda_meta	struct attivita*	Puntatore all'inizio della seconda metà della lista, visto come nodo successivo a quello identificato da meta.
sinistra	struct attivita*	Lista ordinata ottenuta dalla parte sx (dal primo elemento fino a meta).
destra	struct attivita*	Lista ordinata ottenuta dalla parte dx (a partire da seconda_meta).

4.3 FUNZIONI DEL FILE ATTIVITATEST.C:

4.3.1 Funzione nuova_lista_att:

SPECIFICA SINTATTICA: nuova_lista_att(void) -> lista_att

Tipi: lista_att.

SPECIFICA SEMANTICA: nuova_lista_att () -> lista_att

La funzione crea e restituisce una nuova lista di attività vuota.

-*Side-effect:* Nessuno.

-*Precondizione*: Nessuna, la funzione può essere chiamata in qualsiasi momento;

-*Postcondizione*: Restituisce una lista vuota (NULL).

4.3.2 Funzione test_aggiungi_attivita:

SPECIFICA SINTATTICA: test_aggiungi_attivita(lista_att, FILE*, FILE*) -> lista_att

Tipi: lista_att, FILE*.

SPECIFICA SEMANTICA: test_aggiungi_attivita(l, tc_input, tc_output) -> l

La funzione legge una serie di campi relativi a una nuova attività (nome, corso di appartenenza, descrizione, data di scadenza, data di inserimento, priorità, tempo stimato e tempo già trascorso) dal file di input tc_input. Ad ogni fase di lettura, se si verifica un errore (come la mancanza di dati o dati non validi) viene stampato un messaggio di errore sul file tc_output, la memoria eventualmente allocata viene liberata e la funzione restituisce la lista senza modifiche. Se invece tutti i dati vengono letti e validati correttamente, la funzione alloca dinamicamente memoria per un nuovo nodo della lista, che corrisponde ad una nuova attività, ne inizializza i campi (inclusa l'allocazione dinamica dei campi stringa) e lo inserisce in testa alla lista esistente. Infine, la funzione scrive un messaggio di successo sul file di output e restituisce la nuova lista aggiornata.

-*Side-effect*: Scrittura sul file tc_output, allocazione dinamica del nuovo nodo per l'attività e di stringhe, eventuale deallocazione in caso di errori, modifica struttura della lista se tutti i dati di input risultano validi.

-*Precondizione*: l deve rappresentare una lista (anche vuota) valida di attività. I puntatori tc_input e tc_output devono essere aperti correttamente, in lettura e scrittura rispettivamente. Il file tc_input deve contenere i dati nel formato atteso: nome (non vuoto), corso di appartenenza (non vuoto), descrizione (non vuota), due date formattate con il separatore "-" (per data di scadenza e data di inserimento) con valori validi (giorno da 1 a 31, mese da 1 a 12), priorità (intero compreso tra 1 e 3), tempo stimato e il tempo già trascorso (numeri interi).

-*Postcondizione*: Se tutti i dati sono validi e la memoria viene allocata correttamente, la funzione deve inserire la nuova attività in testa alla lista e restituire la lista aggiornata. In caso di errore di qualunque tipo (lettura, validazione o allocazione), deve essere stampato un messaggio di errore su tc_output, la memoria eventualmente allocata deve essere liberata e la funzione deve restituire la lista invariata.

DIZIONARIO DEI DATI:

Identificatore	*Tipo*	*Descrizione*
l	lista_att	Puntatore alla testa della lista delle attività esistente.
tc_input	FILE*	Puntatore al file di input.
tc_output	FILE*	Puntatore al file di output.

nuovo	Struct attivita*	Puntatore al nuovo nodo struct attivita allocato dinamicamente.
data_scadenza	struct data	Struttura che contiene i dati (giorno, mese, anno) della data in cui l'attività scade.
d_inserimento	struct data	Struttura che contiene i dati (giorno, mese, anno) della data in cui l'attività viene inserita.
t_stimato	int	Intero che rappresenta il tempo stimato per l'attività.
t_trascorso	int	Intero che rappresenta il tempo trascorso per l'attività.
priorita	int	Intero che indica la priorità dell'attività, compresa tra 1 e 3.
nome	char[50]	Array di caratteri per il nome della nuova attività (non vuoto).
descrizione	char[150]	Array di caratteri per la descrizione della nuova attività (non vuoto).
corso_appartenenza	char[100]	Array di caratteri per il corso di appartenenza della nuova attività (non vuoto).
buffer	char[100]	Buffer per lettura di stringhe e validazione degli input.
nuovo->nome	char *	Campo della struttura che memorizza il nome della nuova attività.
nuovo->descrizione	char *	Campo della struttura che memorizza la descrizione della nuova attività.
nuovo->corso_appartenenza	char*	Campo della struttura che memorizza il corso di appartenenza della nuova attività.
nuovo->data_scadenza	struct data	Campo della struttura che memorizza la data di scadenza dell'attività.
nuovo->data_inserimento	struct data	Campo della struttura che memorizza la data di inserimento dell'attività.
nuovo->priorita	int	Campo della struttura che memorizza la priorità dell'attività.
nuovo->t_stimato	int	Campo della struttura che memorizza il tempo stimato per completare l'attività.
nuovo->t_trascorso	int	Campo della struttura che memorizza il tempo già trascorso per l'attività.

nuovo->successivo	lista_att	Campo della struttura che punta all'elemento successivo nella lista, inserito in testa.

4.3.3 Funzione test_rimuovi_attivita:

SPECIFICA SINTATTICA: test_rimuovi_attivita(lista_att, FILE*, FILE*) -> lista_att

Tipi: lista_att, FILE*.

SPECIFICA SEMANTICA: test_rimuovi_attivita(l, tc_input, tc_output) -> lista_att

La funzione rimuove una determinata attività dalla lista, che legge dal file di input (tc_input). Attraversa la lista (partendo dalla testa l), cercando un nodo in cui il campo nome corrisponde al nome letto. Se trova il nodo, considera due casi: se l'attività da eliminare è la prima della lista oppure se è in una posizione intermedia o finale. Aggiorna opportunamente i puntatori, dealloca le eventuali aree di memoria dinamica relative ai campi stringa (nome, corso_appartenenza, descrizione) e libera il nodo. Al termine, scrive un messaggio di conferma (attività rimossa) su tc_output e restituisce la lista aggiornata. Se il nodo non viene trovato durante l'attraversamento, viene scritto un messaggio di errore su tc_output indicando che l'attività non è stata trovata e la lista originale viene restituita.

-*Side-effect*: Scrittura sul file tc_output, modifica struttura della lista con relativo aggiornamento dei puntatori, deallocazione memoria allocata precedentemente per il nodo eliminato.

-*Precondizione*: l deve essere una lista valida (anche se vuota). I puntatori tc_input e tc_output devono essere aperti correttamente ed essere validi. Il file tc_input deve contenere il nome (come stringa) di un'attività da eliminare.

-*Postcondizione*: Se l'attività con il nome corrispondente è presente, deve essere rimossa dalla lista e la memoria allocata per essa liberata; il file tc_output deve registrare un messaggio di conferma. Se nessuna attività con il nome fornito viene trovata, la lista deve rimanere invariata e deve essere segnalato l'errore sul file tc_output.

DIZIONARIO DEI DATI:

Identificatore	*Tipo*	*Descrizione*
l	lista_att	Testa della lista delle attività.
tc_input	FILE*	Puntatore al file di input.
tc_output	FILE*	Puntatore al file di output.
corrente	struct attivita*	Rappresenta il nodo corrente durante lo scorrimento della lista.
precedente	struct attivita*	Rappresenta il nodo precedente durante lo scorrimento della lista.
nome	char[50]	Array per memorizzare il nome dell'attività da cercare per rimuoverla.

4.3.4 Funzione test_aggiornamento_progresso:

SPECIFICA SINTATTICA: test_aggiornamento_progresso(lista_att, FILE*, FILE*) -> int

Tipi: lista_att, FILE*, int.

SPECIFICA SEMANTICA: test_aggiornamento_progresso(l, tc_input, tc_output) -> int

La funzione aggiorna il progresso (tempo già trascorso) di un'attività presente nella lista passata. Legge dal file tc_input il nome dell'attività, che servirà per cercare l'attività nella lista. Legge, successivamente, il nuovo valore del tempo trascorso (t_trascorso_agg) da utilizzare per l'aggiornamento. Se il tempo stimato (t_stimato) è maggiore del tempo trascorso aggiornato, viene calcolata e stampata la percentuale di progresso. Se il tempo trascorso è maggiore o uguale al tempo stimato, viene chiesto (sempre leggendo da tc_input) se si intende aggiornare il tempo stimato, digitando il valore 1 se sì, altrimenti un altro.
Se l'attività viene trovata e aggiornamenti eseguiti, la funzione restituisce 1. Se si verifica un errore di lettura o l'attività non viene individuata nella lista, viene restituito 0 e viene stampato un messaggio di errore sul file tc_output.

-*Side-effect*: Scrittura sul file tc_output, modifica della struttura lista_att relativa ai campi t_trascorso e t_stimato.

-*Precondizione*: l deve essere una lista valida (anche vuota). tc_input e tc_output devono essere stati aperti correttamente (rispettivamente in lettura e in scrittura). Il file di input tc_input deve contenere i dati formattati in modo tale che ci sia il nome dell'attività, un intero (t_trascorso_agg) e, se necessario, ulteriori interi (per la scelta dell'utente e, in caso, per il nuovo tempo stimato).

-*Postcondizione*: Se l'attività con il nome specificato viene trovata, il suo campo t_trascorso deve essere aggiornato con il nuovo valore. In caso di tempo trascorso maggiore o uguale a quello stimato, anche il campo t_stimato potrebbe dover essere aggiornato. La funzione deve stampare su tc_output la percentuale di progresso oppure messaggi di errore/conferma relativi all'aggiornamento. La funzione restituirà 1 se l'aggiornamento (o il controllo) è andato a buon fine, 0 in caso contrario.

DIZIONARIO DEI DATI:

Identificatore	*Tipo*	*Descrizione*
l	lista_att	Testa della lista delle attività.
tc_input	FILE *	Puntatore al file di input.
tc_output	FILE *	Puntatore al file di output.
corrente	struct attivita*	Puntatore per scorrere la lista.
nome	char[50]	Array di carattere con nome dell'attività da aggiornare.
t_trascorso_agg	int	Nuovo valore del tempo trascorso letto dal file di input.
val	int	Variabile per la scelta dell'utente.

t_stimato_agg	int	Nuovo valore del tempo stimato letto dal file di input.
percentuale	int	Variabile calcolata implicitamente che rappresenta il progresso dell'attività aggiornata.
corrente->nome	char*	Campo della struttura che contiene dinamicamente il nome dell'attività.
corrente->t_stimato	int	Campo della struttura che memorizza il tempo stimato per completare l'attività.
corrente->t_trascorso	int	Campo della struttura che memorizza il tempo già trascorso per completare l'attività.

4.3.5 Funzione ordina_per_priorita:

SPECIFICA SINTATTICA: ordina_per_priorita(lista_att*) -> void

Tipi: lista_att*.

SPECIFICA SEMANTICA: ordina_per_priorita(l) -> void

La funzione riordina la lista di attività, puntata da l, sulla base del campo "priorità" in modo decrescente, ossia dalla priorità massima (=3) a quella minima (=1). Realizza l'ordinamento chiamando la funzione mergesort_lista, che implementa l'algoritmo MergeSort, presente nel file mergesortList.c.

-*Side-effect*: riordinamento della struttura della lista tramite l'algoritmo MergeSort.

-*Precondizione*: Il puntatore l deve essere inizializzato (anche se la lista è vuota). Le attività presenti nella lista, se esistenti, devono contenere un campo "priorità" valido.

-*Postcondizione:* La lista puntata da l deve essere modificata in modo tale da risultare ordinata in ordine decrescente secondo il campo "priorità". La funzione non deve restituisce alcun valore, poiché l'ordinamento è effettuato grazie alla sostituzione del puntatore alla lista con quello restituito dalla funzione mergesort_lista.

DIZIONARIO DEI DATI:

Identificatore	*Tipo*	*Descrizione*
l	lista_att*	Puntatore alla lista di attività da riordinare.
*l	lista_att	Lista delle attività che verrà riordinata.
Mergesort_lista	Funzione	Funzione che implementa l'algoritmo di ordinamento MergeSort.

4.3.6 Funzione scadenza_att:

SPECIFICA SINTATTICA: scadenza_att(struct data, struct data) -> int

Tipi: struct data, int.

SPECIFICA SEMANTICA: scadenza_att(data_scadenza, fine_sett) -> int

La funzione esegue una serie di confronti tra due strutture di tipo data per determinare se data_scadenza sia precedente ad una data limite, rappresentata dal campo fine_sett. Se l'attività risulta scaduta verrà restituito il valore 1, altrimenti 0.

-*Side-effect*: Nessuno.

-*Precondizione:* data_scadenza e fine_sett devono contenere valori validi (essere date corrette).

-*Postcondizione:* Se data_scadenza risulta precedente a fine_sett, l'attività è scaduta e la funzione dovrà restituire 1. Viceversa, l'attività sarà considerata in corso e la funzione restituirà 0.

DIZIONARIO DEI DATI:

Identificatore	*Tipo*	*Descrizione*
data_scadenza	struct data	Data di scadenza dell'attività da valutare.
fine_sett	struct data	Data limite per il confronto.
anno	int	Campo della struct data.
mese	int	Campo della struct data.
giorno	int	Campo della struct data.

4.3.7 Funzione test_report_settimanale:

SPECIFICA SINTATTICA: test_report_settimanale(lista_att*, FILE*, FILE*) -> void

Tipi: lista_att*, FILE*.

SPECIFICA SEMANTICA: test_report_settimanale(l, tc_input, tc_output) -> void

La funzione genera un report settimanale delle attività contenute in una lista, ordinandole per priorità. Viene letta la data di inizio settimana da tc_input e validata (controllando che il giorno sia compreso fra 1 e 31 e il mese fra 1 e 12). Analogamente, viene letta e validata anche la data di fine settimana. In caso di dati non validi, viene stampato un messaggio di errore su tc_output e la funzione termina. Se la lista risulta vuota (cioè non sono presenti attività), la funzione chiede all'utente, tramite tc_input, se desidera inserire una nuova attività, invocando in caso la funzione test_aggiungi_attivita per aggiornare la lista. Per ogni attività presente nella lista e registrata non oltre il giorno di fine settimana vengono stampati su tc_output tutti i dettagli dell'attività (nome, corso di appartenenza, descrizione, data di scadenza, data di inserimento, priorità, tempo stimato e tempo trascorso) e ne viene verificato lo stato (completata/in corso/scaduta), grazie alla funzione scadenza_att.

-*Side-effect*: Scrittura sul file tc_output, eventuale allocazione dinamica di un nuovo nodo per la lista (su scelta dell'utente), riordinamento struttura della lista tramite la funzione ordina_per_priorita.

-*Precondizione*: l deve fare riferimento a una lista valida (che può essere vuota). I file puntati da tc_input e tc_output devono essere stati aperti correttamente in lettura e scrittura. I valori di input relativi alle date devono essere del formato previsto (ad es. "giorno-mese-anno") e con valori numerici validi.

-*Postcondizione*: Se la lista era vuota, deve essere chiesto all'utente se vuole inserire una nuova attività, tramite la funzione test_aggiungi_attivita. Deve essere stampato un report dettagliato su tc_output per le attività registrate entro la data di fine settimana, ordinate per priorità. La funzione non deve restituire un valore, ma il report e gli eventuali messaggi di errore o interazione devono essere scritti su tc_output.

DIZIONARIO DEI DATI:

Identificatore	*Tipo*	*Descrizione*
l	lista_att*	Puntatore al riferimento della lista delle attività.
tc_input	FILE *	Puntatore al file di input.
tc_output	FILE *	Puntatore al file di output.
corrente	struct attivita*	Puntatore usato per scorrere la lista.
inizio_sett	struct data	Struttura per memorizzare la data di inizio settimana.
fine_sett	struct data	Struttura per memorizzare la data di fine settimana.
val	int	Variabile per memorizzare la scelta dell'utente.
scadenza	int	Variabile che memorizza il valore restituito dalla funzione scadenza_att.
percentuale	int	Variabile che fa riferimento alla percentuale di completamento di un'attività.

4.3.8 Funzione verifica_test:

SPECIFICA SINTATTICA: verifica_test(char*, char*) -> void

Tipi: char*.

SPECIFICA SEMANTICA: verifica_test(file_oracolo, risultato) -> void

La funzione confronta il contenuto del file oracolo e del file di output. Il confronto avviene riga per riga, rimuovendo da ciascuna riga i caratteri di fine linea (\r e \n). Se tutte le righe risultano identiche (compresa l'assenza di righe extra in uno dei due file), il test viene considerato "SUCCESSO" altrimenti "FALLIMENTO". Successivamente, la funzione estrae la parte iniziale del nome del file oracolo (ad esempio, estrae "TC1" da "TC1_oracolo.txt") e scrive, in modalità append, una riga nel file "risultato".

-*Side-effect*: Scrittura sul file indicato da “risultato” ed eventuale scrittura sul file di output per messaggi di errore.

-*Precondizione*: Il file oracolo indicato deve esistere ed essere accessibile in lettura. Analogamente per il file "TC_OUTPUT.txt”. Il file indicato da “risultato” deve poter essere aperto in modalità append per scrittura.

-*Postcondizione*: Se il contenuto dei due file confrontati è identico, deve essere scritto su risultato il messaggio indicante il successo, altrimenti viene scritto un messaggio di fallimento. Tutti i file aperti dovranno essere correttamente chiusi alla fine della funzione. In caso di errore nell’apertura di uno dei file, viene stampato un messaggio di errore sullo standard output e la funzione termina senza apportare modifiche al file di “risultato”.

DIZIONARIO DEI DATI:

Identificatore	*Tipo*	*Descrizione*
file_oracolo	char *	Nome del file oracolo.
risultato	char *	Nome del file di output.
f_oracolo	FILE *	Puntatore al file aperto in lettura per il file oracolo.
f_output	FILE *	Puntatore al file aperto in lettura per il file di output.
successo	int	Variabile per memorizzare l’esito del confronto dei due file.
buf_oracolo	char[1024]	Buffer che contiene temporaneamente una riga letta dal file oracolo.
buf_output	char[1024]	Buffer che contiene temporaneamente una riga letta dal file di output.
riga_oracolo	char *	Puntatore usato per memorizzare la riga letta per il file oracolo.
riga_output	char *	Puntatore usato per memorizzare la riga letta per il file output.
tc	char[20]	Array di caratteri che contiene la porzione iniziale del nome del file oracolo.
f_risultato	FILE *	Puntatore al file di risultato aperto in modalità append.

4.3.9 Funzione libera_memoria:

SPECIFICA SINTATTICA: libera_memoria(lista_att*) -> void

Tipi: lista_att*.

SPECIFICA SEMANTICA: libera_memoria(l) -> void

La funzione dealloca tutta la memoria dinamica occupata dagli elementi della lista di attività, prevenendo così possibili memory leak prima della fine della funzione main. Per ogni elemento nella lista, ne libera i campi dinamici (nome, descrizione, corso_appartenenza) e, infine, ne libera l'elemento stesso. Una volta completato il ciclo, il puntatore alla lista viene impostato a NULL per evitare futuri accessi non validi.

-*Side-effect*: Deallocazione memoria di ogni nodo della lista, incluse le stringhe allocate dinamicamente in ciascun nodo (nome, descrizione, corso di appartenenza), modifica puntatore della lista a NULL.

-*Precondizione*: Il puntatore l deve essere inizializzato (la lista può essere vuota o piena).

-*Postcondizione*: Tutta la memoria dinamica allocata per ogni attività della lista deve essere stata correttamente rilasciata. Il puntatore alla lista deve puntare a NULL, rendendo la lista non più accessibile.

DIZIONARIO DEI DATI:

Identificatore	*Tipo*	*Descrizione*
l	lista_att*	Puntatore alla lista da liberare.
corrente	struct attivita*	Puntatore per scorrere la lista.
temp	struct attivita*	Puntatore temporaneo per memorizzare l'attività corrente prima di deallocarla.
nome	char*	Campo dinamico della struct attivita da deallocare.
descrizione	char*	Campo dinamico della struct attivita da deallocare.
corso_appartenenza	char*	Campo dinamico della struct attivita da deallocare.

5. FASE DI TESTING E RAZIONALE DEI CASI DI TEST

Il testing è una fase importante dello sviluppo di un software poiché permette di "stressare" l'algoritmo per verificare che il suo comportamento sia sempre fedele a quello definito nella specifica. Non è possibile verificare la correttezza assoluta di un programma, ma è possibile individuarne i malfunzionamenti, cioè quei casi in cui il comportamento del programma è diverso da quello atteso.

Ogni file TC_INPUT (file di input test case) è in corrispondenza con un file ORACOLO, che contiene la stampa degli atteggiamenti attesi dal programma. Il testing non avviene attraverso il programma generico, ma ne è stata sviluppata una versione per il testing automatico, che riporta il successo (o il fallimento) dei casi di test su un file, denominato "RISULTATI_TEST".

```
PS C:\Users\plait\Desktop\psd-0512123517> make -f makefileTest
gcc -c maintest.c
gcc -c attivitatest.c
gcc -c mergesortList.c
gcc maintest.o attivitatest.o mergesortList.o -o ProgrammaTesting
PS C:\Users\plait\Desktop\psd-0512123517> .\ProgrammaTesting TC1_INPUT.txt TC1_ORACOLO.txt RISULTATI_TEST.txt
```

Per ogni test di questa sezione, si considererà il valore 1 come scelta dell'utente.

TC1: Inserimento corretto con tutti i dati validi*:*

-*Scenario:* L'utente inserisce dati validi per ogni campo del nodo Attività.

-*Risultato atteso*: L'attività viene correttamente inserita nella lista con tutti i campi compilati come da input.

-*Esito*: SUCCESSO

TC2: Inserimento con data scadenza non valida inizialmente:

-*Scenario:* L'utente inserisce una data di scadenza non valida (con giorno=35 e mese=14), poi corregge l'input con una data valida.

-*Risultato atteso:* Viene stampato l'errore e, una volta corretta la data, la funzione procede e inserisce l'attività nella lista.

-*Esito:* SUCCESSO

TC3: Inserimento con valore di priorità non valida:

-*Scenario:* L'utente inserisce un valore di priorità fuori dal range e poi fornisce il valore corretto.

-*Risultato atteso:* Dopo l'errore, la funzione accetta il valore corretto e completa l'inserimento.

-*Esito:* SUCCESSO

TC4: Inserimento con stringhe vuote per campi nome o corso:

-*Scenario:* L’utente immette una stringa vuota per uno dei campi testuali.

-*Risultato atteso:* Viene stampato un messaggio di errore e la funzione termina.

-*Esito:* SUCCESSO

TC5: Verifica della corretta allocazione dinamica delle stringhe:

-*Scenario:* Si immettono input più lunghi per verificare che allocazioni per nome, descrizione e corso funzionino correttamente.

-*Risultato atteso:* Le stringhe vengono copiate correttamente, senza troncamenti o errori di memoria.

-*Esito:* SUCCESSO