# Report di progetto

# Introduzione

Partendo dal requisito che richiedeva di “creare un programma in C che aiuti uno studente a gestire e monitorare la sua attività di studio” l’ipotesi di partenza è stata che l’utente avrebbe passato molto più tempo ad aggiornare lo stato delle attività (ripetendo questa operazione molte volte per ogni attività), rispetto ad inserire o cancellare attività. Il flusso di lavoro immaginato è quindi il seguente:

- l’utente inserisce una attività (una volta sola)
- l’utente aggiorna una attività man mano che ci lavora (n volte) fino al completamento
- l’utente, normalmente, non cancella le attività

# Architettura del progetto

Il progetto in esame rappresenta il tentativo di creare un'applicazione C modulare per la gestione di attività, con funzionalità di inserimento, visualizzazione, modifica, cancellazione, salvataggio e reporting delle attività stesse. È strutturato attorno a una struttura dati principale ad albero AVL, che dovrebbe garantire efficienza nelle operazioni O(log n). Nella progettazione ho cercato di privilegiare oltre alla modularità anche il riuso, l'information hiding e l'incapsulamento delle informazioni.

## Motivazione della scelta degli ADT usati

Nel progetto sono stati utilizzati due tipi di Abstract Data Type (ADT):

- Albero AVL
- Lista concatenata

La scelta di questi ADT è stata motivata da esigenze funzionali e prestazionali specifiche che elenco di seguito.

### Albero AVL (AVL Tree)

#### *Motivazione della scelta*

L'albero AVL è un albero binario di ricerca bilanciato che garantisce operazioni di inserimento, cancellazione e ricerca in tempo logaritmico.
Questo permette di gestire un numero elevato di attività, in quanto consente un accesso rapido e ordinato in base all'ID dell'attività (usato come chiave primaria).
Una ulteriore motivazione è stata quella didattica: volevo capire meglio il funzionamento di questa struttura.

#### *Utilizzo nel progetto*

L'albero AVL, dove ogni nodo rappresenta un'attività, è usato come struttura principale all'interno di ActivitiesContainer.
L'inserimento automatico dell'ID e il mantenimento del bilanciamento dell'albero assicurano che la struttura rimanga efficiente anche con modifiche frequenti (anche se non attese, come da premessa).
Le stampe (in-order traversal) e le ricerche specifiche sfruttano le proprietà dell'albero (ordine) per restituire risultati coerenti e veloci.

#### *Vantaggi ipotizzati*

Ottime performance per operazioni di ricerca e aggiornamento.
Mantiene i dati ordinati in modo naturale, facilitando la stampa e l'esportazione ordinata.

## Lista Concatenata (Linked List)

### *Motivazione*

La lista concatenata è una struttura dati semplice ma flessibile, adatta alla costruzione dinamica di insiemi temporanei.
È ideale quando l'ordine di inserimento non è prioritario o quando si lavora con sottoinsiemi dinamici di dati, come nei report.

### *Utilizzo nel progetto*

Utilizzata in ActivitiesContainerSupportList, serve come contenitore di supporto per i report settimanali/periodici, dove le attività vengono temporaneamente classificate come completate, in corso, in ritardo o ancora da iniziare.
La lista supporta ordinamento personalizzato (tramite merge sort) su vari criteri, come data di completamento o percentuale di avanzamento, in modo da mostrare all'utente un'informazione coerente secondo il contesto in cui si trova.

### *Vantaggi ipotizzati*

Per la sua relativa semplicità di implementazione e gestione dinamica della memoria si adatta a operazioni di raccolta, filtro e ordinamento temporaneo, senza impattare sulla struttura dati principale.

## Riassunto motivazione scelta ADT

Albero AVL: scelto per gestire l'intero insieme delle attività in modo ordinato, efficiente e scalabile.
Lista concatenata: scelta per gestire insiemi temporanei di attività da ordinare o stampare in vari formati.

Questa combinazione fornisce, - a mio avviso - efficienza e flessibilità, mantenendo ben separati i compiti strutturali (AVL) da quelli di supporto e analisi (lista).

# Struttura del Sistema e interazione dei Componenti

La struttura del sistema è organizzata in moduli distinti, ognuno con responsabilità specifiche. La comunicazione tra i moduli avviene attraverso interfacce ben definite, cercando di non violare la divisione in moduli. Di seguito è illustrata l'architettura e le interazioni tra i componenti.

## Struttura del Sistema (moduli software)

### *Activity (Definizione di attività)*

Rappresenta i dati e lo stato di una singola attività.
Contiene la struttura 'struct activity', funzioni getter/setter, creazione ('newActivity') e distruzione ('deleteActivity').
Usato da tutti gli altri moduli, in particolare AVL e ActivitiesContainer.

### *AVL Tree*

Gestisce l'organizzazione efficiente delle attività in un AVL tree ordinato per ID.
Include inserimento ('insertNode'), cancellazione ('deleteNode'), bilanciamento, ricerca ('search').
Usato internamente e solo dal contenitore ActivitiesContainer. Dipende solo da Activity.

### *ActivitiesContainer (Contenitore ad alto livello)*

Fornisce un'interfaccia astratta che nasconde l'implementazione dell'albero AVL.
Gestisce inserimenti, rimozioni, salvataggi, caricamenti da file, stampa e report.
Contiene l'AVL e interagisce con tutti gli altri moduli.

### *Activity helper (file con insieme di funzioni)*

Funzioni ausiliarie su attività: calcolo percentuale completamento, comparazioni, stampa.
Usato dagli altri moduli per operazioni avanzate su attività. Dipende solo da Activity.

### *Activities container helper (file con insieme di funzioni)*

Esegue operazioni ricorsive su albero, stampa, salvataggio in-order, creazione da input utente.
Usato da ActivitiesContainer per operazioni avanzate sull'albero. Dipende solo dall'albero ALV.

### *ActivitiesContainerSupportList*

Implementazione della lista. Viene usato nella generazione dei report come liste temporanee categorizzate (completate, scadute, ecc.).
Supporta ordinamento (merge sort) e stampa.

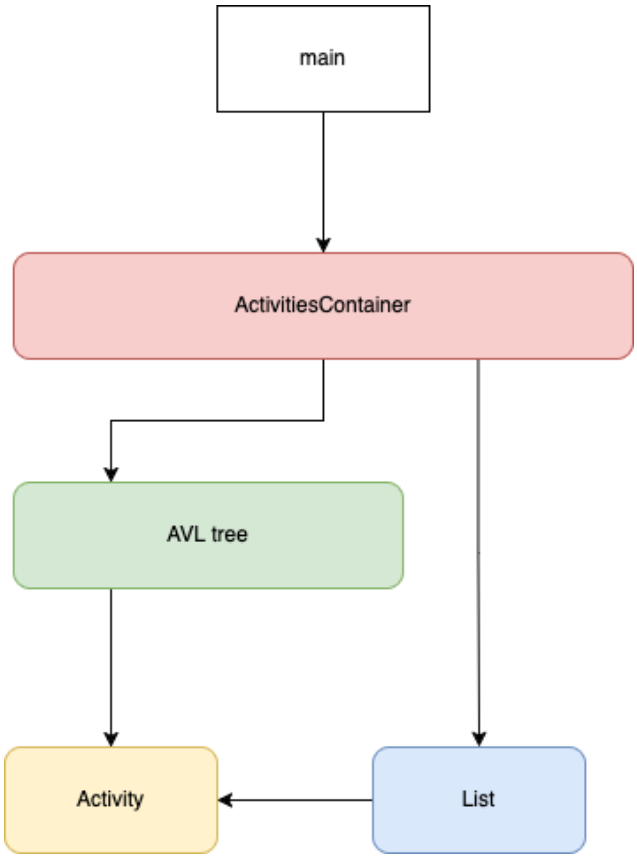
### *Utils (file con insieme di funzioni)*

Funzioni generiche di input/output, gestione menu, conversione date e tempo.
Utilizzato trasversalmente da tutto il progetto.

## File principali e loro contenuto

| File | Descrizione |
|---|---|
| **main.c** | Punto di ingresso, gestisce il menu e l'interazione utente. |
| **activity.h / activity.c** | Definizione e gestione dell'entità 'Activity'. |
| **activities_container.h/.c** | Interfaccia e implementazione del contenitore ad alto livello. |
| **activities_container_avl.h/.c** | Gestione dell'albero AVL. |
| **activity_helper.h/.c** | Funzioni ausiliarie sulle attività. |
| **activities_container_helper.h/.c** | Operazioni su albero, stampa/salvataggio, creazione attività. |
| **activities_container_support_list.h/.c** | Liste per report settimanale/periodico e ordinamento. |
| **utils.h/.c** | Utility generiche: input, menu, gestione stringhe, date. |
| **test_main.c** | Test del progetto. |

## Diagramma semplificato delle interazioni




## Caratteristiche Architetturali

Con questa architettura ho cercato di mantenere la separazione dei compiti in modo che ogni componente fosse isolato e modificabile indipendentemente. In particolare, credo di aver ottenuto le seguenti caratteristiche principali:

- *Modularità*: Ogni modulo ha responsabilità definita, sviluppo e manutenzione indipendente.
- *Riutilizzabilità*: Moduli come ‘utils’, ‘activity’, e ‘activity_helper’ sono riutilizzabili senza necessità di modifiche.
- *Efficienza*: AVL garantisce efficienza logaritmica in inserimento/ricerca/cancellazione.
- *Flessibilità*: L’AVL può essere sostituito senza impatti sul codice client grazie all’uso di ActivitiesContainer.
- *Information Hiding*: Tipi opachi e funzioni getter/setter nascondono l’implementazione.
- *Information Encapsulation*: Accessi ai dati solo tramite interfacce pubbliche.

# Istruzioni per la compilazione e l'esecuzione

Informazioni più dettagliate possono essere trovate nel README di progetto.

## Prerequisiti

- Compilatore GCC installato sul sistema
- Utilità Make disponibile

## Compilazione

Per compilare il programma principale, eseguire:

```
make
```

Questo comando genererà l'eseguibile ‘gestatt’ nella directory ‘bin/’

## Compilare ed eseguire i test

Per compilare ed eseguire la suite di test, eseguire:

```
make test
```

Questo comando eseguirà automaticamente i test

## Esecuzione

Dopo la compilazione, eseguire il programma con:

```
cd bin
./gestatt
```

## Eseguire solo i test

Se i test sono già stati compilati e si desidera eseguirli nuovamente:

```
cd test
./gestatt_test
```

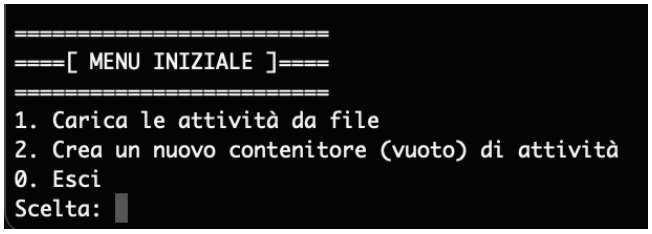
## Struttura dei file di progetto

Il progetto segue questa struttura di directory:

```
.
├── src/       # Codice sorgente
├── test/      # Directory di test ed eseguibile del test
├── bin/       # Directory principale dell'eseguibile (creata durante la compilazione)
├── docs/      # Documentazione
└── Makefile # Configurazione di compilazione
```

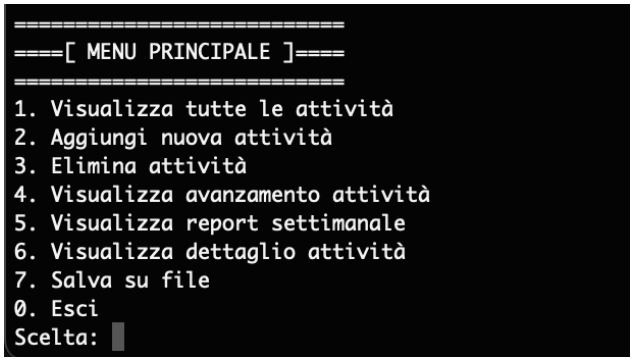
# Guida per l’utente

All’avvio l’applicazione *(**./gestatt**)* chiederà se si vogliono caricare le attività da un file esistente o se si vuole creare un nuovo contenitore (vuoto).




## Menu principale

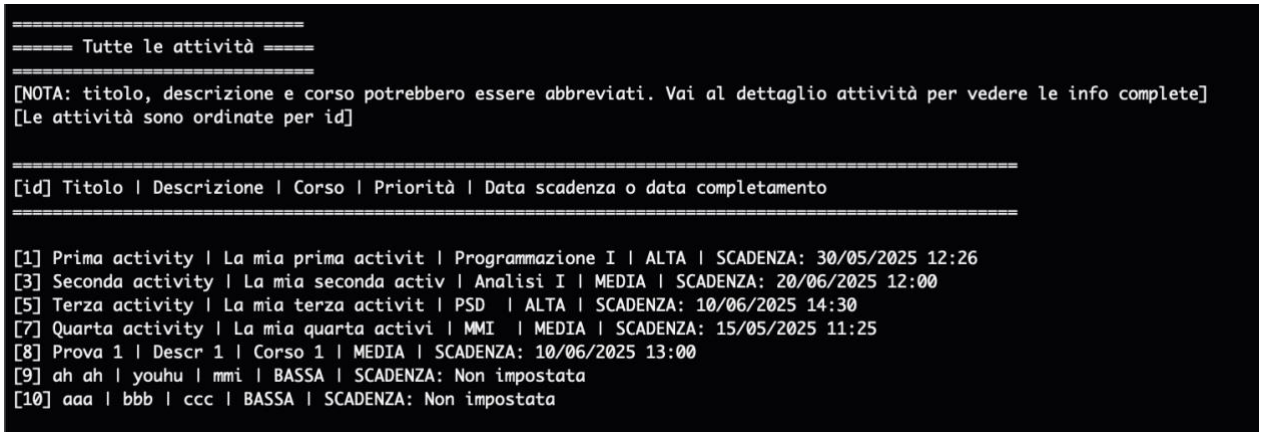
Fatta la propria scelta, si verrà portati al menu principale che è il cuore dell’applicazione.




Supponendo di aver caricato le attività da file (o di averne create alcune dopo aver scelto un nuovo contenitore vuoto) oltre alle opzioni per la creazione/cancellazione delle attività (2 e 3) e la possibilità di salvare la situazione corrente in un file (opzione 7), sono presenti le opzioni di visualizzazione che sono quelle che esploreremo brevemente in questo manuale.

## Visualizza tutte le attività (opzione 1)

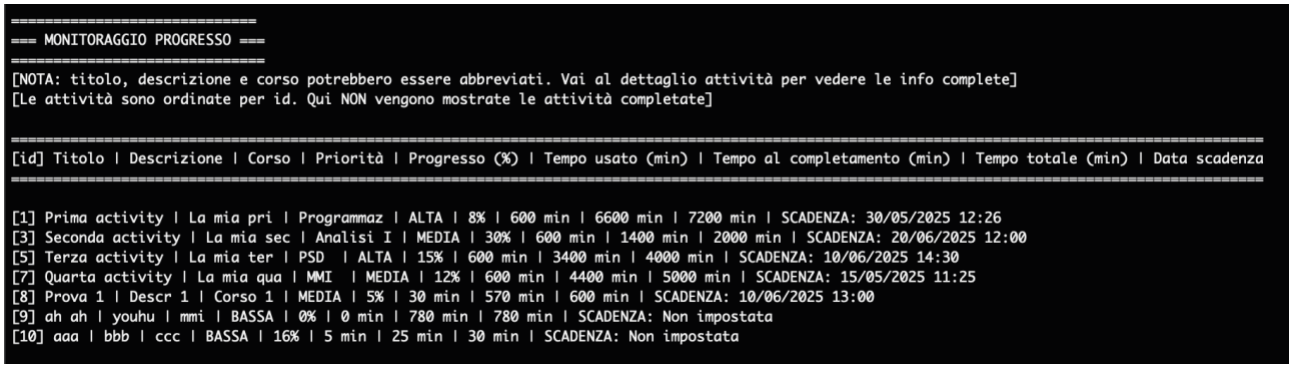
Permette di visualizzare tutte le attività, in tutti gli stati possibili, ordinate per id. Questa visualizzazione serve ad avere una visione generale delle attività nel sistema.



## Visualizza avanzamento attività (opzione 4)

In questa modalità vengono ignorate le attività già completate e viene mostrato un dettaglio dell'avanzamento (percentuale di completamento, tempo mancante al completamento, tempo già usato sull'attività, tempo totale) per tutte le altre attività.

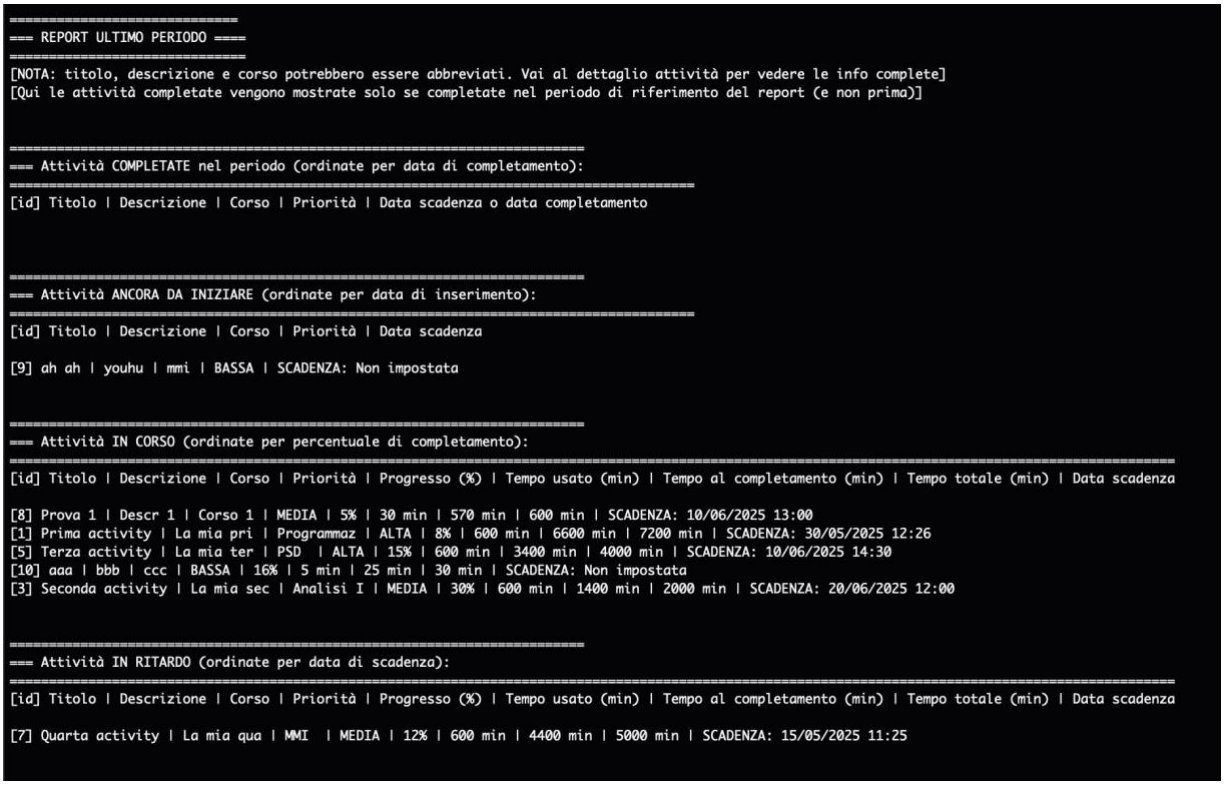
Lo scopo di questa visualizzazione è fornire uno stato di avanzamento generale delle attività ancora da completare.



## Visualizza report settimanale (opzione 5)

L’utente può scegliere se visualizzare l’ultima settimana (di default il sistema calcola una settimana dalla data attuale) o scegliere un’altra data dalla quale far partire il report (report periodico):


Una volta scelta la data (o confermata quella di default) si accede ad una visualizzazione dettagliata in cui sono evidenziate le attività completate nel periodo scelto (solo se completate nel periodo) e altri 3 gruppi di attività (non iniziate, in corso, in ritardo) ognuno con ordinamento coerente con il gruppo e indicato nell’interfaccia utente.
Lo spirito di questa modalità di visualizzazione è quello fornire il dettaglio dello stato riferito all’ultimo periodo in modo che l’utente possa avere evidenza dell’andamento delle attività e agire di conseguenza.



## Visualizza dettaglio attività (opzione 6)

In questa modalità l'utente ha la visualizzazione di dettaglio sulla singola attività e un menu che gli permette di aggiornare l'attività stessa. In questa sede si sottolinea che un'attività viene impostata come completata solo quando viene scelta esplicitamente dall'utente l'opzione 8 da questo menu di dettaglio.


Le altre interazioni e menu dovrebbero risultare intuitivi e autoesplicativi.

# Casi di Test

I casi di test implementati nel sistema sono progettati per validare il corretto funzionamento dei moduli principali e delle funzionalità più rilevanti, garantendo affidabilità, coerenza e correttezza delle operazioni. La validazione dell'input dell'utente viene fatta contestualmente all'input stesso e non ho perciò ritenuto particolarmente vantaggioso indirizzare i test su questo aspetto specifico. Nella modalità di esecuzione scelta ho cercato di privilegiare il riutilizzo della logica già esistente nel progetto e reindirizzare le print finali su file, invece che a video. per il confronto con il risultato atteso.

## TC_1 – Caricamento e salvataggio

*Obiettivo*: Verificare che un file contenente attività possa essere caricato correttamente e salvato senza alterazioni.
*Razionale*: Assicura l'integrità del ciclo I/O (input da file → gestione interna → salvataggio). Fondamentale perché molti altri test dipendono da una corretta gestione di questi aspetti. Il superamento di questo test mi assicura anche che la struct activity viene creata correttamente e che l'albero viene creato, aggiornato e visitato nel modo giusto.

## TC_2 – Visualizzazione dettaglio attività

*Obiettivo*: Testare che la funzione di stampa dettagliata di una singola attività funzioni come previsto.
*Razionale*: Il dettaglio attività è critico per l'interazione utente. Il test garantisce che i dati visualizzati siano coerenti con quelli salvati.

## TC_3 – TC_11 – Modifica attributi attività

Ogni test modifica un singolo attributo dell'attività e verifica che il cambiamento sia persistente e correttamente stampato:

| *Test* | *Campo Testato* | *Razionale* |
|---|---|---|
| TC_3 | Nome attività | Verifica la gestione delle stringhe e l'aggiornamento corretto |
| TC_4 | Descrizione | Come sopra |
| TC_5 | Corso | Come sopra |
| TC_6 | Data inserimento | Testa la gestione delle date |
| TC_7 | Data scadenza | Critico per i report e la logica temporale |
| TC_8 | Data completamento | Impatta il calcolo dei progressi e i report |
| TC_9 | Tempo totale | Coinvolge calcoli di completamento |
| TC_10 | Tempo usato | Come sopra |
| TC_11 | Priorità | Verifica la formattazione e l'ordinamento per priorità |

## TC_12 – Stampa di tutte le attività

*Obiettivo*: Testare l'output della stampa in-order dell'intero albero AVL.
*Razionale*: Verifica la gestione corretta del contenitore (ActivitiesContainer), l'ordinamento interno dell'albero, la sua visita e la corretta gestione di Acivity in modalità lista.

## TC_13 – Stampa avanzamento attività

*Obiettivo*: Testare la funzione che mostra il progresso delle attività non completate.
*Razionale*: Il progresso è basato su percentuali e richiede accuratezza nei calcoli e nei filtri. In particolare, verifica la gestione dell'albero e la corretta gestione di Acivity in modalità "lista con progresso".

## TC_14 – Report settimanale

*Obiettivo*: Verificare la corretta classificazione e stampa delle attività in base al loro stato.
*Razionale*: È una delle funzionalità più complesse, unendo filtro, ordinamento e formattazione. In particolare, verifica la gestione delle liste e il loro corretto ordinamento.

## TC_15 – Caricamento/salvataggio con più attività

*Obiettivo*: Test simile al TC_1 ma con un numero maggiore di attività.
*Razionale*: Verifica la correttezza della gestione della struttura AVL con più nodi.

## TC_16 – Report con modifica post-caricamento

*Obiettivo*: Testare che una modifica ad un'attività abbia effetto immediato nei report.
*Razionale*: Verifica l'integrazione tra aggiornamento dati e generazione dei report e la conseguente gestione delle liste e il loro corretto ordinamento.

## Conclusione sui casi di test

L'intera suite di test cerca di coprire l'integrità dei dati, la gestione delle strutture dati (AVL e liste), il corretto funzionamento delle funzionalità utente, la gestione delle date e progressi, e funzionalità complesse come i report.

# Specifiche

Le specifiche che seguono sono state copiate e tradotte partendo da quelle presenti nei file di progetto (.c e .h). In caso di dubbi si prega di controllare quelle presenti all'interno dei file sorgenti.

## File "activities_container"

### struct containerItem (ActivitiesContainer)

Specifica Sintattica:
```
struct containerItem {
    TreeNode avlTree;
    int nextId;
};
```

Specifica Semantica:
Definisce la struttura container delle attività che fornisce un livello di astrazione tra l'applicazione e la struttura dati sottostante AVL tree. Questo container incapsula la complessità delle operazioni sull'albero e la gestione degli ID, offrendo un'interfaccia semplificata per la gestione delle attività.

Campi:
- avlTree: Puntatore al nodo radice dell'AVL tree contenente tutte le attività
- nextId: Il prossimo ID univoco da assegnare a una nuova attività al momento dell'inserimento

Vantaggi dell'Astrazione:
- Indipendenza dalla struttura dati: il codice applicativo non ha bisogno di conoscere i dettagli implementativi dell'AVL tree, rotazioni o operazioni di bilanciamento
- Interfaccia semplificata: il codice client usa operazioni ad alto livello (insert, remove, search) senza gestire direttamente puntatori o strutture di nodi
- Gestione automatica degli ID: il container gestisce automaticamente la generazione di ID univoci, prevenendo conflitti e garantendo l'integrità dei dati
- Flessibilità di implementazione: la struttura dati sottostante può essere cambiata (es. in hash table, B-tree) senza influenzare il codice client
- Gestione degli errori: le funzioni del container forniscono una gestione coerente degli errori e validazioni senza esporre condizioni di errore specifiche dell'albero
- Gestione della memoria: allocazione e deallocazione centralizzata delle risorse dell'albero tramite funzioni a livello di container

Note:
- Quando il container è vuoto, avlTree è NULL e nextId è tipicamente 1
- nextId viene incrementato automaticamente quando si inseriscono nuove attività
- Le attività sono memorizzate nell'AVL tree ordinate per ID univoco
- Il container garantisce operazioni efficienti O(log n) nascondendo la complessità
- Questa astrazione segue il principio di information hiding e progettazione modulare

## getRootNode

Specifica Sintattica:
TreeNode getRootNode(ActivitiesContainer container);

Specifica Semantica:
Restituisce il nodo radice dell'AVL tree contenuto nel container.

Precondizioni:
- Nessuna

Postcondizioni:
- Se 'container == NULL', restituisce 'NULL'
- Altrimenti, restituisce 'container->avlTree'

Side Effects:
- Nessuno

## getNextId

Specifica Sintattica:
int getNextId(ActivitiesContainer container);

Specifica Semantica:
Restituisce il prossimo ID disponibile per una nuova attività.

Precondizioni:
- Nessuna

Postcondizioni:
- Se 'container == NULL', restituisce -1
- Altrimenti, restituisce 'container->nextId'

Side Effects:
- Nessuno

## getActivityWithId

Specifica Sintattica:
Activity getActivityWithId(ActivitiesContainer container, int activityId);

Specifica Semantica:
Restituisce l'attività con l'ID specificato.

Precondizioni:
- 'activityId' deve essere un ID valido

Postcondizioni:
- Se l'attività è trovata, viene restituita

- Altrimenti, restituisce 'NULL'

Side Effects:
- Nessuno

## insertActivity

Specifica Sintattica:
void insertActivity(ActivitiesContainer container, Activity activity);

Specifica Semantica:
Inserisce un'attività nel container (AVL tree), generando automaticamente un ID se necessario (id = 0).

Precondizioni:
- 'activity != NULL'

Postcondizioni:
- Se 'container == NULL', crea un nuovo container
- Se l'ID è 0, genera un nuovo ID univoco
- Inserisce l'attività nel container
- Aggiorna 'nextId'

Side Effects:
- Può allocare memoria per il container
- Modifica l'ID dell'attività se era 0
- Modifica la struttura dell'albero AVL

## removeActivity

Specifica Sintattica:
void removeActivity(ActivitiesContainer container, int activityId);

Specifica Semantica:
Rimuove un'attività dal container (AVL tree) dato il suo ID.

Precondizioni:
- 'container' può essere 'NULL'
- 'activityId' deve essere un ID valido

Postcondizioni:
- Se 'container == NULL' o 'container->avlTree == NULL', nessuna azione
- Altrimenti, rimuove l'attività

Side Effects:
- Modifica la struttura dell'albero
- Aggiorna 'avlTree' con la nuova radice
- Può deallocare memoria

## newActivityContainer

Specifica Sintattica:
ActivitiesContainer newActivityContainer(void);

Specifica Semantica:
Crea e inizializza un nuovo container vuoto.

Precondizioni:
- Nessuna

Postcondizioni:
- Restituisce container con 'avlTree = NULL' e 'nextId = 1'
- Se fallisce l’allocazione, restituisce 'NULL'

Side Effects:
- Alloca memoria per la struttura

## deleteActivityContainer

Specifica Sintattica:
void deleteActivityContainer(ActivitiesContainer container);

Specifica Semantica:
Elimina completamente un contenitore di attività e tutto il suo contenuto (albero e attività).

Precondizioni:
- Nessuna

Postcondizioni:
- Se 'container == NULL', nessuna azione
- Altrimenti, dealloca l'intero contenitore e le sue attività

Side Effects:
- Dealloca tutta la memoria del contenitore

## saveActivitiesToFile

Specifica Sintattica:
int saveActivitiesToFile(const char* filename, ActivitiesContainer container);

Specifica Semantica:
Salva tutte le attività dal contenitore in un file.

Precondizioni:
- Nessuna

Postcondizioni:
- Se 'container == NULL' o 'filename != NULL', restituisce 1 (errore)
- Se impossibile aprire il file, restituisce 1 (errore)
- Altrimenti, salva tutte le attività e restituisce 0 (successo)

Side Effects:
- Apertura e scrittura di file
- Output su stdout (messaggi informativi)

## printActivityWithId

Specifica Sintattica:
void printActivityWithId(ActivitiesContainer container, int activityId);

Specifica Semantica:
Stampa i dettagli dell'attività con l'ID specificato.

Precondizioni:
- 'container' può essere 'NULL'
- 'activityId' deve essere un ID valido

Postcondizioni:
- Se l'attività viene trovata, i suoi dettagli vengono stampati
- Altrimenti, nessuna azione

Side Effects:
- Output su stdout tramite 'printActivityDetailWithMenu()'

## printActivities

Specifica Sintattica:
void printActivities(ActivitiesContainer container);

Specifica Semantica:
Stampa tutte le attività (formato lista, una per riga) dal contenitore in ordine crescente di ID.

Precondizioni:
- Nessuna

Postcondizioni:
- Se 'container != NULL', stampa tutte le attività con intestazioni (vedi 'printAllActivities')

Side Effects:
- Output su stdout

## printActivitiesToFile

Specifica Sintattica:
void printActivitiesToFile(ActivitiesContainer container, FILE* file);

Specifica Semantica:
Stampa tutte le attività del contenitore su file (formato lista, una per riga).

Precondizioni:
- 'file' deve essere aperto per la scrittura

Postcondizioni:
- Se entrambi i parametri sono validi, stampa tutte le attività su file

Side Effects:
- Scrittura su file

## printActivitiesProgress

Specifica Sintattica:
void printActivitiesProgress(ActivitiesContainer container);

Specifica Semantica:
Stampa il progresso di tutte le attività del contenitore (formato lista progresso, una per riga) con intestazioni.

Precondizioni:
- Nessuna

Postcondizioni:
- Se 'container != NULL', stampa il progresso di tutte le attività

Side Effects:
- Output su stdout

## printActivitiesProgressToFile

Specifica Sintattica:
void printActivitiesProgressToFile(ActivitiesContainer container, FILE* file);

Specifica Semantica:
Stampa il progresso di tutte le attività del contenitore su file (formato lista progresso, una per riga).

Precondizioni:
- 'file' deve essere aperto per la scrittura

Postcondizioni:
- Se entrambi i parametri sono validi, stampa il progresso con intestazione

Side Effects:
- Scrittura su file

## printActivitiesReport

Specifica Sintattica:
void printActivitiesReport(ActivitiesContainer container);

Specifica Semantica:
Genera e stampa un rapporto dettagliato delle attività categorizzate per stato e periodo.

Precondizioni:
- Nessuna

Postcondizioni:
- Se 'container' è valido, stampa un rapporto completo con attività categorizzate
- Interagisce con l'utente per definire il periodo del rapporto

Side Effects:
- Alloca e dealloca liste di supporto temporanee
- Interazione con l'utente (input/output)
- Output su stdout
- Chiamate a funzioni di gestione del tempo

## printActivitiesReportToFile

Specifica Sintattica:
void printActivitiesReportToFile(ActivitiesContainer container, time_t beginDate, FILE* file);

Specifica Semantica:
Stampa un rapporto delle attività su file per un periodo specificato.

Precondizioni:
- 'file' deve essere aperto per la scrittura
- 'beginDate >= 0'

Postcondizioni:
- Se tutti i parametri sono validi, stampa il rapporto su file

Side Effects:
- Alloca e dealloca liste di supporto temporanee
- Scrittura su file

## readActivitiesFromFile

Specifica Sintattica:
ActivitiesContainer readActivitiesFromFile(const char* filename, int* count);

Specifica Semantica:
Legge le attività da un file e le inserisce in un nuovo contenitore.

Precondizioni:
- 'count != NULL'

Postcondizioni:
- Crea un nuovo contenitore
- Se il file non esiste, restituisce un contenitore vuoto e '*count = 0'
- Altrimenti, legge tutte le attività dal file e aggiorna '*count' (numero di attività lette)

Side Effects:
- Alloca memoria per il contenitore e le attività
- Modifica '*count' (numero di attività lette)
- Apertura e lettura di file
- Output su stdout (messaggi informativi)

## addNewActivityToContainer

Specifica Sintattica:
ActivitiesContainer addNewActivityToContainer(ActivitiesContainer container);

Specifica Semantica:
Interagisce con l'utente per creare e aggiungere una nuova attività al contenitore.

Precondizioni:
- 'container != NULL'

Postcondizioni:
- Se 'container == NULL', ritorna
- Altrimenti, chiede i dati all'utente e aggiunge la nuova attività

Side Effects:
- Modifica il contenitore aggiungendo una nuova attività
- Alloca memoria per la nuova attività
- Interazione con l'utente (input/output)

# File "activities_container_avl"

## struct node (AVL tree)

Specifica Sintattica:
typedef struct node {
    Activity activity;
    struct node* left;
    struct node* right;
    int height;
} Node;

Specifica Semantica:
Definisce la struttura del nodo per l'albero AVL. Ogni nodo contiene una struttura dati Activity (puntatore),
puntatori ai nodi figli sinistro e destro, e memorizza la propria altezza per operazioni efficienti di bilanciamento AVL.

Campi:
- activity: la struttura dati Activity (puntatore) memorizzata in questo nodo
- left: puntatore al nodo figlio sinistro (contiene attività con ID minori)
- right: puntatore al nodo figlio destro (contiene attività con ID maggiori)
- height: l'altezza di questo nodo nell'albero (lunghezza del percorso più lungo verso una foglia)

Note:
- Per ogni nodo, tutte le attività nel sottoalbero sinistro hanno ID minori dell'ID dell'attività del nodo
- Per ogni nodo, tutte le attività nel sottoalbero destro hanno ID maggiori dell'ID dell'attività del nodo
- Il campo height riflette accuratamente l'altezza del nodo nell'albero
- È mantenuta la proprietà di bilanciamento AVL: |height(left) - height(right)| <= 1

## getLeftNode

Specifica Sintattica:
TreeNode getLeftNode(TreeNode node);

Specifica Semantica:
Restituisce il figlio sinistro del nodo specificato.

Precondizioni:
- Nessuna

Postcondizioni:
- Se 'node == NULL', restituisce 'NULL'
- Altrimenti, restituisce 'node->left'

Side Effects:
- Nessuno

## getRightNode

Specifica Sintattica:
TreeNode getRightNode(TreeNode node);

Specifica Semantica:
Restituisce il figlio destro del nodo specificato.

Precondizioni:
- Nessuna

Postcondizioni:
- Se 'node == NULL', restituisce 'NULL'
- Altrimenti, restituisce 'node->right'

Side Effects:
- Nessuno

## getActivityFromNode

Specifica Sintattica:
Activity getActivityFromNode(TreeNode node);

Specifica Semantica:
Restituisce l'attività memorizzata nel nodo specificato.

Precondizioni:
- Nessuna

Postcondizioni:
- Se 'node == NULL', restituisce 'NULL'
- Altrimenti, restituisce 'node->activity'

Side Effects:
- Nessuno

## search

Specifica Sintattica:
TreeNode search(Node* root, int activityId);

Specifica Semantica:
Cerca un'attività con l'ID specificato (chiave dell'albero AVL) nell'albero con radice 'root'.

Precondizioni:
- 'activityId' deve essere un ID valido

Postcondizioni:
- Se 'root == NULL', restituisce 'NULL'
- Se trova l'attività con l'ID specificato, restituisce il nodo corrispondente
- Altrimenti, restituisce 'NULL'

Side Effects:
- Nessuno

## getHeight

Specifica Sintattica:
int getHeight(Node* n);

Specifica Semantica:
Restituisce l'altezza del nodo 'n'. Ricorda che l'altezza di un nodo è la lunghezza del percorso più lungo dal nodo a una foglia.
Negli alberi AVL è importante memorizzarla perché consente di calcolare in modo efficiente il fattore di bilanciamento
e verificare la necessità di ribilanciamento.

Precondizioni:
- Nessuna

Postcondizioni:
- Se 'n == NULL', restituisce 0
- Altrimenti, restituisce 'n->height'

Side Effects:
- Nessuno

## createNode

Specifica Sintattica:
Node* createNode(Activity activity);

Specifica Semantica:
Crea un nuovo nodo contenente l'attività specificata.

Precondizioni:
- 'activity != NULL'

Postcondizioni:
- Se 'activity == NULL', restituisce 'NULL'
- Altrimenti, alloca un nuovo nodo con l'attività specificata
- Il nuovo nodo ha figli 'NULL' e altezza 1

Side Effects:
- Alloca memoria per un nuovo nodo

## getBalanceFactor

Specifica Sintattica:
int getBalanceFactor(Node* n);

Specifica Semantica:
Calcola il fattore di bilanciamento del nodo (altezza sottoalbero sinistro - altezza sottoalbero destro).

Precondizioni:
- Nessuna

Postcondizioni:
- Se 'n == NULL', restituisce 0
- Altrimenti, restituisce la differenza tra le altezze dei figli sinistro e destro

Side Effects:
- Nessuno

## rightRotate

Specifica Sintattica:
Node* rightRotate(Node* y);

Specifica Semantica:
Esegue una rotazione a destra sul nodo 'y' per mantenere le proprietà dell'albero AVL.

Precondizioni:
- 'y != NULL'
- 'y->left != NULL'

Postcondizioni:
- Restituisce la nuova radice dopo la rotazione
- Le altezze dei nodi coinvolti vengono aggiornate
- Sono mantenute le proprietà AVL

Side Effects:
- Modifica la struttura dell'albero
- Aggiorna le altezze dei nodi

## leftRotate

Specifica Sintattica:
Node* leftRotate(Node* x);

Specifica Semantica:
Esegue una rotazione a sinistra sul nodo 'x' per mantenere le proprietà dell'albero AVL.

Precondizioni:
- 'x != NULL'
- 'x->right != NULL'

Postcondizioni:
- Restituisce la nuova radice dopo la rotazione
- Le altezze dei nodi coinvolti vengono aggiornate
- Sono mantenute le proprietà AVL

Side Effects:
- Modifica la struttura dell'albero
- Aggiorna le altezze dei nodi

## insertNode

Specifica Sintattica:
TreeNode insertNode(TreeNode node, Activity activity);

Specifica Semantica:
Inserisce una nuova attività nel sottoalbero con radice 'node' mantenendo le proprietà AVL.

Precondizioni:
- 'activity != NULL'

Postcondizioni:
- Se 'node == NULL', crea e restituisce un nuovo nodo
- Se l'attività esiste già (stesso ID), restituisce l'albero non modificato
- Altrimenti, inserisce l'attività e riequilibra l'albero se necessario
- L'albero risultante mantiene le proprietà AVL

Side Effects:
- Può allocare memoria per nuovi nodi

- Modifica la struttura dell'albero
- Aggiorna le altezze dei nodi

## minValueNode

Specifica Sintattica:
Node* minValueNode(Node* node);

Specifica Semantica:
Restituisce il nodo con il valore minimo della chiave (id) nell'albero con radice 'node'. Foglia più a sinistra.

Precondizioni:
- Nessuna (la funzione gestisce il caso 'node == NULL')

Postcondizioni:
- Se 'node == NULL', restituisce 'NULL'
- Altrimenti, restituisce il nodo più a sinistra dell'albero
- Il nodo restituito non ha figlio sinistro (se non è 'NULL')

Side Effects:
- Nessuno

## deleteNode

Specifica Sintattica:
TreeNode deleteNode(TreeNode root, int activityId);

Specifica Semantica:
Elimina il nodo con l'ID specificato dal sottoalbero con radice 'root' mantenendo le proprietà AVL.

Precondizioni:
- 'activityId' deve essere un ID valido

Postcondizioni:
- Se 'root == NULL', restituisce 'NULL'
- Se il nodo con 'activityId' non esiste, restituisce l'albero non modificato
- Altrimenti, elimina il nodo e riequilibra l'albero
- L'attività contenuta nel nodo eliminato viene deallocata

Side Effects:
- Dealloca la memoria del nodo eliminato
- Dealloca l'attività contenuta
- Modifica la struttura dell'albero
- Aggiorna le altezze dei nodi

## deleteSubtree

Specifica Sintattica:
void deleteSubtree(Node* root);

Specifica Semantica:
Elimina ricorsivamente (visita postorder) tutti i nodi di un sottoalbero e le relative attività.

Precondizioni:
- Nessuna

Postcondizioni:
- Tutti i nodi del sottoalbero vengono deallocati
- Tutte le attività contenute vengono deallocate

Side Effects:
- Dealloca tutta la memoria dell’albero

# File "activities_container_support_list"

## Struct nodelist (NodeList)

Definizione:
```
typedef struct nodelist {
    Activity activity;
    struct nodelist* next;
} NodeList;
```

Descrizione:
Rappresenta un singolo nodo di una semplice lista collegata che contiene un'attività e un puntatore al nodo successivo. Questa struttura implementa il pattern della linked list per memorizzare sequenze di attività.

Campi:
- Activity activity: Puntatore all'attività contenuta nel nodo. Può essere NULL se il nodo non contiene un'attività valida.
- struct nodelist* next: Puntatore al nodo successivo nella linked list. È NULL se questo è l'ultimo nodo della lista.

Utilizzo:
- Usata per creare catene di nodi che formano una linked list
- Ogni nodo mantiene un riferimento all'attività e al nodo successivo
- Consente l'attraversamento sequenziale della lista
- Supporta inserimenti e cancellazioni dinamici

Note:
- Se next è NULL, il nodo è l'ultimo della lista
- Il campo activity può essere NULL (nodo vuoto)
- La struttura è ricorsiva tramite il puntatore next

## Struct listItem (ActivitiesContainerSupportList)

Definizione:
```
struct listItem {
    NodeList* head;
};
```

Descrizione:
Rappresenta la struttura principale che gestisce una linked list di attività. Funziona da contenitore e punto di accesso alla lista, mantenendo un riferimento al primo nodo (head) della catena.

Campi:
- NodeList* head: Puntatore al primo nodo della linked list. È NULL se la lista è vuota. Rappresenta il punto di ingresso per tutte le operazioni sulla lista.

Utilizzo:
- Punto di accesso principale per tutte le operazioni sulla lista

- Mantiene l’integrità della struttura dati
- Consente l’identificazione rapida se la lista è vuota (head == NULL)
- Facilita le operazioni di inserimento in testa alla lista

Note:
- Se head è NULL, la lista è vuota
- Se head non è NULL, punta a un nodo NodeList valido
- La struttura deve essere allocata dinamicamente prima dell’uso
- Deve essere deallocata correttamente per evitare memory leak

Note Implementative:
- ActivitiesContainerSupportList è un typedef che punta a struct listItem*
- La lista supporta inserimenti in testa con efficienza O(1)
- Le operazioni di attraversamento iniziano sempre da head
- La deallocazione richiede la liberazione di tutti i nodi prima della struttura

## newSupportList

Specifica Sintattica:
```
ActivitiesContainerSupportList newSupportList(void);
```

Specifica Semantica:
Crea e inizializza una nuova lista vuota per contenere attività.
Alloca dinamicamente memoria per la struttura lista e
inizializza il puntatore head a NULL.

Precondizioni:
- Nessuna precondizione specifica

Postcondizioni:
- Se l’allocazione ha successo: restituisce un puntatore valido
  a una lista vuota con head = NULL
- Se l’allocazione fallisce: restituisce NULL

Side Effects:
- Alloca memoria dinamica per una struttura listItem
- Inizializza il campo head della lista a NULL

## deleteSupportList

Specifica Sintattica:
```
void deleteSupportList(ActivitiesContainerSupportList* list);
```

Specifica Semantica:
Dealloca completamente una lista di attività, liberando la memoria di
tutti i nodi e della struttura principale. Imposta il puntatore alla lista
a NULL per evitare riferimenti pendenti. Il parametro è un puntatore
passato per riferimento.

Precondizioni:

- list può essere NULL o puntare a un puntatore valido/NULL

Postcondizioni:
- Tutta la memoria allocata per la lista e i suoi nodi è liberata
- *list è impostato a NULL
- Se list o *list erano NULL, la funzione non ha effetto

Side Effects:
- Libera la memoria di tutti i nodi della lista
- Libera la memoria della struttura principale
- Modifica il valore del puntatore passato per riferimento

## isSupportListEmpty

Specifica Sintattica:
int isSupportListEmpty(ActivitiesContainerSupportList list);

Specifica Semantica:
Verifica se una lista di attività è vuota controllando se la lista è NULL o se il suo head è NULL.

Precondizioni:
- list può essere NULL o puntare a una struttura listItem valida

Postcondizioni:
- Restituisce 1 se la lista è vuota o NULL
- Restituisce 0 se la lista contiene almeno un elemento

Side Effects:
- Nessuno

## addActivityToSupportList

Specifica Sintattica:
void addActivityToSupportList(ActivitiesContainerSupportList list, Activity activity);

Specifica Semantica:
Aggiunge una nuova attività all'inizio della lista (inserimento in testa). Crea un nuovo nodo, copia l'attività (il puntatore all'attività) in esso e lo collega come nuovo head della lista.

Precondizioni:
- list deve essere un puntatore valido a una struttura listItem (non NULL)
- activity deve essere una struttura Activity valida

Postcondizioni:
- Se l'allocazione del nuovo nodo ha successo: l'attività è aggiunta in testa alla lista
- Se l'allocazione fallisce: la lista rimane invariata
- Il nuovo nodo diventa il nuovo head della lista

Side Effects:
- Alloca memoria per un nuovo nodo NodeList
- Copia l'attività nel nuovo nodo
- Aggiorna i puntatori per inserire il nodo in testa
- Modifica la struttura della lista

## splitSupportList

Specifica Sintattica:
NodeList* splitSupportList(NodeList* head);

Specifica Semantica:
Funzione di supporto per merge sort che divide una lista in due metà approssimativamente uguali usando la tecnica "tartaruga e lepre" (puntatori slow - la tartaruga - e fast - la lepre).

Precondizioni:
- head può essere NULL o puntare a un nodo valido

Postcondizioni:
- Se head è NULL: restituisce NULL
- Altrimenti: divide la lista e restituisce il puntatore alla seconda metà
- La prima metà rimane collegata a head
- La connessione tra le due metà viene interrotta

Side Effects:
- Modifica i puntatori next per dividere la lista
- Modifica la struttura originale della lista rompendo i collegamenti
- Non alloca né dealloca memoria

## mergeSupportLists

Specifica Sintattica:
NodeList* mergeSupportLists(NodeList* listA, NodeList* listB, int sortBy);

Specifica Semantica:
Funzione di supporto per merge sort che unisce due liste ordinate in un'unica lista ordinata secondo il criterio specificato da sortBy.

Precondizioni:
- listA e listB possono essere NULL o puntare a liste già ordinate
- sortBy deve essere un valore intero valido (0-11) che specifica il criterio di ordinamento

Postcondizioni:
- Restituisce una lista ordinata contenente tutti gli elementi di listA e listB
- Se entrambe le liste sono NULL: restituisce NULL
- Se una lista è NULL: restituisce l'altra lista
- L'ordinamento segue il criterio specificato da sortBy

Side Effects:
- Riorganizza i puntatori per unire le liste
- Modifica la struttura delle liste originali riconnettendo i nodi
- Non alloca nuova memoria per i nodi

## mergeSortSupportList

Specifica Sintattica:
NodeList* mergeSortSupportList(NodeList* head, int sortBy);

Specifica Semantica:
Implementa l'algoritmo ricorsivo di merge sort per ordinare una linked list di attività secondo il criterio specificato (sortBy).

Precondizioni:
- head può essere NULL o puntare a un nodo valido
- sortBy deve essere un valore intero valido (0-11)

Postcondizioni:
- Restituisce il puntatore al nuovo head della lista ordinata
- Se la lista è vuota o ha un solo elemento: restituisce head invariato
- La lista risultante è ordinata secondo il criterio sortBy

Side Effects:
- Divide ricorsivamente la lista in sottoliste
- Ordina e unisce le sottoliste
- Riorganizza tutti i puntatori della lista
- Modifica completamente la struttura originale della lista
- Usa lo stack per la ricorsione

## sortSupportList

Specifica Sintattica:
void sortSupportList(ActivitiesContainerSupportList list, int sortBy);

Specifica Semantica:
Ordina una lista di attività secondo il criterio specificato usando l'algoritmo di merge sort. Modifica la lista originale.

Precondizioni:
- list deve essere un puntatore valido a una struttura listItem (non NULL)
- sortBy deve essere un valore intero valido (0-11) che specifica il criterio di ordinamento

Postcondizioni:
- Se la lista è vuota: nessun effetto
- Se la lista non è vuota: è ordinata secondo il criterio sortBy
- Il campo head della lista punta al nuovo primo elemento ordinato

Side Effects:
- Aggiorna il puntatore head della struttura principale
- Modifica permanentemente l'ordine degli elementi nella lista
- Usa lo stack per la ricorsione di merge sort

## printActivitiesInSupportList

Specifica Sintattica:
void printActivitiesInSupportList(ActivitiesContainerSupportList list, int printType, FILE* file);

Specifica Semantica:
Stampa tutte le attività presenti nella lista secondo il tipo
di stampa specificato. Può stampare su stdout o su un file specificato.

Precondizioni:
- list può essere NULL o puntare a una struttura listItem valida
- printType: 0 per stampa normale, 1 per stampa con progressi, altri valori forzano la stampa normale (default)
- file può essere NULL (per stdout) o un puntatore FILE valido

Postcondizioni:
- Se la lista è vuota: non viene eseguita alcuna stampa
- Se la lista non è vuota: tutte le attività vengono stampate nell'ordine della lista
- La destinazione della stampa dipende dal valore di file: su file se non è NULL, altrimenti su stdout

Side Effects:
- Attraversa tutti i nodi della lista
- Output su stdout o file specificato
- Nessuna modifica alla struttura dati

# File "activity"

## Struct activity

Specifica Sintattica:
```
struct activity {
    int id;
    char* name;
    char* descr;
    char* course;
    time_t insertDate;
    time_t expiryDate;
    time_t completionDate;
    unsigned int totalTime;
    unsigned int usedTime;
    short unsigned int priority;
};
```

Specifica Semantica:
Definisce la struttura dati per la gestione di attività accademiche o professionali.
Ogni activity rappresenta un compito o incarico con vincoli temporali,
capacità di tracciamento del tempo e gestione delle priorità.

Campi:
- id: Identificatore univoco dell'activity (intero)
- name: Nome descrittivo dell'activity (stringa allocata dinamicamente)
- descr: Descrizione dettagliata dell'activity (stringa allocata dinamicamente)
- course: Corso o contesto associato (stringa allocata dinamicamente)
- insertDate: Timestamp di creazione/inserimento dell'activity (time_t)
- expiryDate: Scadenza o timestamp di fine validità dell'activity (time_t)
- completionDate: Timestamp di completamento dell'activity (time_t)
- totalTime: Tempo totale allocato per l'activity in minuti (unsigned int)
- usedTime: Tempo già speso sull'activity in minuti (unsigned int)
- priority: Livello di priorità dell'activity (short unsigned int)

Note:
- Tutti i campi stringa (name, descr, course) sono allocati dinamicamente e possono essere NULL
- I campi temporali usano time_t per la rappresentazione standard del timestamp
- Il tracciamento del tempo è misurato in minuti per un controllo più granulare
- La priorità usa short unsigned int per efficienza di memoria
- La struttura supporta il tracciamento completo del ciclo di vita, dalla creazione al completamento

## newActivity

Specifica Sintattica:
```
Activity newActivity(int id, char* name, char* descr, char* course,
                     time_t insertDate, time_t expiryDate, time_t completionDate,
                     unsigned int totalTime, unsigned int usedTime,
                     short unsigned int priority);
```

Specifica Semantica:
Crea una nuova activity con i parametri specificati.

Precondizioni:
- Memoria sufficiente disponibile per l'allocazione
- I parametri stringa possono essere NULL
- copyString deve essere definita e funzionante

Postcondizioni:
- Restituisce un puntatore a una Activity inizializzata con i valori forniti
- Restituisce NULL se l'allocazione fallisce
- Le stringhe sono copiate, non solo referenziate

Side Effects:
Alloca memoria dinamica per una nuova struttura Activity e per le sue stringhe.

## deleteActivity

Specifica Sintattica:
void deleteActivity(Activity a);

Specifica Semantica:
Libera la memoria allocata per un'activity.

Precondizioni:
- 'a' deve essere un puntatore valido o NULL

Postcondizioni:
- Tutta la memoria dinamica associata all'activity viene liberata

Side Effects:
Dealloca la memoria dinamica.

## copyActivity

Specifica Sintattica:
Activity copyActivity(Activity old);

Specifica Semantica:
Crea una copia completa di una activity esistente.

Precondizioni:
- Memoria sufficiente disponibile per l'allocazione

Postcondizioni:
- Restituisce un puntatore a una nuova Activity che è una copia di old
- Restituisce NULL se old è NULL o se l'allocazione fallisce

Side Effects:
Alloca memoria dinamica per una nuova struttura Activity e per le sue stringhe.

## Funzioni Getter (getActivityId, getActivityName, getActivityDescr, ecc.)

Specifica Sintattica:
int getActivityId(Activity a);
char* getActivityName(Activity a);
char* getActivityDescr(Activity a);
char* getActivityCourse(Activity a);
time_t getActivityInsertDate(Activity a);
time_t getActivityExpiryDate(Activity a);
time_t getActivityCompletionDate(Activity a);
unsigned int getActivityTotalTime(Activity a);
unsigned int getActivityUsedTime(Activity a);
short unsigned int getActivityPriority(Activity a);
char* getActivityPriorityFormatted(Activity a);

Specifica Semantica:
Restituisce i valori dei corrispondenti campi dell'activity.

Precondizioni:
Nessuna.

Postcondizioni:
- Restituisce il valore del campo se a non è NULL
- Restituisce NULL se a è NULL e il tipo di ritorno è char*
- Restituisce 0 se a è NULL e il tipo di ritorno non è char*
- getActivityPriorityFormatted restituisce una rappresentazione testuale della priorità

Side Effects:
Nessuno.

## Funzioni Setter (setActivityId, setActivityName, setActivityDescr, ecc.)

Specifica Sintattica:
void setActivityId(Activity a, int newId);
void setActivityName(Activity a, char* name);
void setActivityDescr(Activity a, char* descr);
void setActivityCourse(Activity a, char* course);
void setActivityInsertDate(Activity a, time_t insertDate);
void setActivityExpiryDate(Activity a, time_t expiryDate);
void setActivityCompletionDate(Activity a, time_t completionDate);
void setActivityTotalTime(Activity a, unsigned int totalTime);
void setActivityUsedTime(Activity a, unsigned int usedTime);
void setActivityPriority(Activity a, short unsigned int priority);

Specifica Semantica:
Imposta i valori dei corrispondenti campi dell'activity.

Precondizioni:
- Per le funzioni stringa: copyString deve essere definita e funzionante

Postcondizioni:
- Il campo corrispondente è aggiornato se a non è NULL
- Per le stringhe: la vecchia stringa è liberata e quella nuova è copiata
- Nessun effetto se a è NULL

Side Effects:
- Deallocazione e allocazione di memoria per le stringhe
- Modifica dello stato dell'oggetto Activity

# File "activity_helper"

## isActivityYetToBegin

Specifica Sintattica:
int isActivityYetToBegin(Activity a);

Specifica Semantica:
Verifica se un'attività deve ancora iniziare (used time = 0).

Precondizioni:
Nessuna.

Postcondizioni:
- Restituisce 1 se used time == 0
- Restituisce 0 altrimenti o se a è NULL (attività completata o in corso)

Side Effects:
Nessuno.

## isActivityCompleted

Specifica Sintattica:
int isActivityCompleted(Activity a);

Specifica Semantica:
Verifica se un'attività è completata (ha una data di completamento).

Precondizioni:
Nessuna.

Postcondizioni:
- Restituisce 1 se completion date != 0
- Restituisce 0 altrimenti o se a è NULL

Side Effects:
Nessuno.

## wasActivityCompletedAfterDate

Specifica Sintattica:
int wasActivityCompletedAfterDate(Activity a, time_t thresholdDate);

Specifica Semantica:
Verifica se un'attività è stata completata dopo una certa data di soglia.

Precondizioni:

- thresholdDate deve essere un valore valido di tipo time_t

Postcondizioni:
- Restituisce 1 se completion date > thresholdDate
- Restituisce 0 altrimenti o se a è NULL

Side Effects:
Nessuno.

## wasActivityExpiredBeforeDate

Specifica Sintattica:
int wasActivityExpiredBeforeDate(Activity a, time_t thresholdDate);

Specifica Semantica:
Verifica se un'attività è scaduta prima di una data di soglia.

Precondizioni:
- thresholdDate deve essere un valore valido di tipo time_t

Postcondizioni:
- Restituisce 1 se expiry date > 0 && expiry date < thresholdDate
- Restituisce 0 altrimenti o se a è NULL

Side Effects:
Nessuno.

## compareWithId

Specifica Sintattica:
int compareWithId(Activity a, int activityId);

Specifica Semantica:
Confronta l'ID di un'attività con un ID specificato.

Precondizioni:
- activityId deve essere un intero valido

Postcondizioni:
- Restituisce 0 se a_id == activityId
- Restituisce -1 se a_id < activityId
- Restituisce 1 se a_id > activityId
- Restituisce -2 se a è NULL

Side Effects:
Nessuno.

## activityCompletionPercentage

Specifica Sintattica:
int activityCompletionPercentage(Activity activity);

Specifica Semantica:
Calcola la percentuale (int) di completamento di un'attività basandosi sul tempo usato rispetto al tempo totale.

Precondizioni:
Nessuna.

Postcondizioni:
- Restituisce un intero che rappresenta la percentuale (0-100+)
- Restituisce 0 se activity è NULL o se il tempo totale è 0

Side Effects:
Nessuno.

## compareNullActivity

Specifica Sintattica:
int compareNullActivity(const Activity a, const Activity b);

Specifica Semantica:
Confronta due puntatori Activity per gestire i casi NULL.

Precondizioni:
Nessuna.

Postcondizioni:
- Restituisce 0 se entrambi sono NULL
- Restituisce 1 se solo a non è NULL
- Restituisce -1 se solo b non è NULL
- Restituisce 0 se entrambi non sono NULL

Side Effects:
Nessuno.

## compareNullString

Specifica Sintattica:
int compareNullString(const char* a, const char* b);

Specifica Semantica:
Confronta due puntatori a stringa per gestire i casi NULL.

Precondizioni:
Nessuna.

Postcondizioni:
- Restituisce 0 se entrambi sono NULL
- Restituisce 1 se solo a non è NULL
- Restituisce -1 se solo b non è NULL
- Restituisce 0 se entrambi non sono NULL

Side Effects:
Nessuno.

## compareActivityById

Specifica Sintattica:
int compareActivityById(const Activity a, const Activity b);

Specifica Semantica:
Confronta due attività per ID.

Precondizioni:
Nessuna.

Postcondizioni:
- Restituisce il risultato del confronto degli ID se entrambi non sono NULL
- Gestisce i casi NULL utilizzando compareNullActivity

Side Effects:
Nessuno.

## compareActivityByName

Specifica Sintattica:
int compareActivityByName(const Activity a, const Activity b);

Specifica Semantica:
Confronta due attività per nome utilizzando strcmp.

Precondizioni:
Nessuna.

Postcondizioni:
- Restituisce il risultato di strcmp se entrambi i nomi non sono NULL
- Gestisce i casi NULL utilizzando compareNullString

Side Effects:
Nessuno.

## compareActivityByDescr

Specifica Sintattica:
int compareActivityByDescr(const Activity a, const Activity b);

Specifica Semantica:
Confronta due attività per descrizione utilizzando strcmp.

Precondizioni:
Nessuna.

Postcondizioni:
- Restituisce il risultato di strcmp se entrambe le descrizioni non sono NULL
- Gestisce i casi NULL utilizzando compareNullString

Side Effects:
Nessuno.

## compareActivityByCourse

Specifica Sintattica:
int compareActivityByCourse(const Activity a, const Activity b);

Specifica Semantica:
Confronta due attività per corso utilizzando strcmp.

Precondizioni:
Nessuna.

Postcondizioni:
- Restituisce il risultato di strcmp se entrambi i corsi non sono NULL
- Gestisce i casi NULL utilizzando compareNullString

Side Effects:
Nessuno.

## compareActivityByInsertDate

Specifica Sintattica:
int compareActivityByInsertDate(const Activity a, const Activity b);

Specifica Semantica:
Confronta due attività per data di inserimento.

Precondizioni:
Nessuna.

Postcondizioni:
- Restituisce 0 se le date sono uguali
- Restituisce -1 se a->insertDate < b->insertDate
- Restituisce 1 se a->insertDate > b->insertDate
- Gestisce i casi NULL utilizzando compareNullActivity

Side Effects:
Nessuno.

## compareActivityByExpiryDate

Specifica Sintattica:
int compareActivityByExpiryDate(const Activity a, const Activity b);

Specifica Semantica:
Confronta due attività per data di scadenza.

Precondizioni:
Nessuna.

Postcondizioni:
- Restituisce 0 se le date sono uguali
- Restituisce -1 se a->expiryDate < b->expiryDate
- Restituisce 1 se a->expiryDate > b->expiryDate
- Gestisce i casi NULL utilizzando compareNullActivity

Side Effects:
Nessuno.

## compareActivityByCompletionDate

Specifica Sintattica:
int compareActivityByCompletionDate(const Activity a, const Activity b);

Specifica Semantica:
Confronta due attività per data di completamento.

Precondizioni:
Nessuna.

Postcondizioni:
- Restituisce 0 se le date sono uguali
- Restituisce -1 se a->completionDate < b->completionDate
- Restituisce 1 se a->completionDate > b->completionDate
- Gestisce i casi NULL utilizzando compareNullActivity

Side Effects:
Nessuno.

## compareActivityByTotalTime

Specifica Sintattica:
int compareActivityByTotalTime(const Activity a, const Activity b);

Specifica Semantica:
Confronta due attività per tempo totale.

Precondizioni:
Nessuna.

Postcondizioni:
- Restituisce 0 se i tempi sono uguali
- Restituisce -1 se a->totalTime < b->totalTime
- Restituisce 1 se a->totalTime > b->totalTime
- Gestisce i casi NULL utilizzando compareNullActivity

Side Effects:
Nessuno.

## compareActivityByUsedTime

Specifica Sintattica:
int compareActivityByUsedTime(const Activity a, const Activity b);

Specifica Semantica:
Confronta due attività per tempo utilizzato.

Precondizioni:
Nessuna.

Postcondizioni:
- Restituisce 0 se i tempi sono uguali
- Restituisce -1 se a->usedTime < b->usedTime
- Restituisce 1 se a->usedTime > b->usedTime
- Gestisce i casi NULL utilizzando compareNullActivity

Side Effects:
Nessuno.

## compareActivityByPriority

Specifica Sintattica:
int compareActivityByPriority(const Activity a, const Activity b);

Specifica Semantica:
Confronta due attività per priorità.

Precondizioni:
Nessuna.

Postcondizioni:
- Restituisce 0 se le priorità sono uguali
- Restituisce -1 se a->priority < b->priority
- Restituisce 1 se a->priority > b->priority
- Gestisce i casi NULL utilizzando compareNullActivity

Side Effects:
Nessuno.

## compareActivityByPercentCompletion

Specifica Sintattica:
int compareActivityByPercentCompletion(Activity a, Activity b);

Specifica Semantica:
Confronta due attività per percentuale di completamento.

Precondizioni:
Nessuna.

Postcondizioni:
- Restituisce 0 se le percentuali sono uguali
- Restituisce -1 se la percentuale di a < percentuale di b
- Restituisce 1 se la percentuale di a > percentuale di b
- Gestisce i casi NULL utilizzando compareNullActivity

Side Effects:
Nessuno.

## compareActivityByTimeToCompletion

Specifica Sintattica:
int compareActivityByTimeToCompletion(Activity a, Activity b);

Specifica Semantica:
Confronta due attività per tempo rimanente al completamento.

Precondizioni:

Nessuna.

Postcondizioni:
- Restituisce 0 se i tempi rimanenti sono uguali
- Restituisce -1 se il tempo rimanente di a < tempo rimanente di b
- Restituisce 1 se il tempo rimanente di a > tempo rimanente di b
- Gestisce i casi NULL utilizzando compareNullActivity

Side Effects:
Nessuno.

## compareActivityBy

Specifica Sintattica:
int compareActivityBy(Activity a, Activity b, int compareBy);

Specifica Semantica:
Confronta due attività utilizzando un criterio di confronto specificato. Il metodo di confronto è determinato dal parametro compareBy, che seleziona tra vari attributi dell'attività (ID, nome, descrizione, corso, date, tempi, priorità, ecc.).

CRITERI DI ORDINAMENTO (sortBy)

| Valore | Criterio |
|---|---|
| 0 | ID Attività |
| 1 | Nome Attività |
| 2 | Descrizione |
| 3 | Corso |
| 4 | Data di Inserimento |
| 5 | Data di Scadenza |
| 6 | Data di Completamento |
| 7 | Tempo Totale |
| 8 | Tempo Utilizzato |
| 9 | Priorità |
| 10 | Percentuale di Completamento |
| 11 | Tempo al Completamento |

Precondizioni:
- compareBy dovrebbe essere un intero valido (0-11 per confronti definiti)

Postcondizioni:
- Restituisce 0 se le attività sono uguali secondo il criterio specificato
- Restituisce -1 se l'attività a è minore dell'attività b secondo il criterio specificato
- Restituisce 1 se l'attività a è maggiore dell'attività b secondo il criterio specificato
- Utilizza il confronto per ID come predefinito per valori compareBy non validi
- Gestisce i casi NULL appropriatamente in base al metodo di confronto selezionato

Side Effects:
Nessuno.

## priorityToText

Specifica Sintattica:
char* priorityToText(int priority);

Specifica Semantica:
Restituisce una stringa che rappresenta la priorità.

Precondizioni:
Nessuna.

Postcondizioni:
- Restituisce "HIGH" se priority == 1
- Restituisce "MEDIUM" se priority == 2
- Restituisce "LOW" se priority == 3
- Restituisce "?" per altri valori

Side Effects:
Nessuno.

## getActivityPriorityText

Specifica Sintattica:
char* getActivityPriorityText(Activity a);

Specifica Semantica:
Restituisce una stringa che rappresenta il campo priorità dell'attività.

Precondizioni:
Nessuna.

Postcondizioni:
- Restituisce NULL se a è NULL
- Restituisce una rappresentazione testuale della priorità utilizzando la funzione 'priorityToText'

Side Effects:
Nessuno.

## displayActivityDetailMenu

Specifica Sintattica:
void displayActivityDetailMenu();

Specifica Semantica:

Visualizza il menu dei dettagli dell'attività. Funzione di interfaccia utente.

Precondizioni:
- stdout deve essere disponibile

Postcondizioni:
- Il menu viene stampato su stdout

Side Effects:
Output su stdout.

## handleActivityDetailMenu

Specifica Sintattica:
Activity handleActivityDetailMenu(Activity activity);

Specifica Semantica:
Gestisce l'interazione dell'utente con il menu dei dettagli dell'attività.

Precondizioni:
- activity deve essere un puntatore Activity valido

Postcondizioni:
- Restituisce activity se l'utente continua la modifica
- Restituisce NULL se l'utente esce dal menu
- L'attività può essere modificata in base alle scelte dell'utente (nome, descrizione, ecc.)

Side Effects:
- Input/output da console
- Modifica dello stato dell'oggetto Activity
- Allocazione/deallocazione di memoria per stringhe

## printActivity

Specifica Sintattica:
void printActivity(Activity activity);

Specifica Semantica:
Stampa tutti i dettagli di un'attività in un formato leggibile.

Precondizioni:
- stdout deve essere disponibile

Postcondizioni:
- I dettagli dell'attività vengono stampati su stdout se activity non è NULL
- Nessun output se activity è NULL

Side Effects:

Output su stdout.

## printActivityDetailWithMenu

Specifica Sintattica:
void printActivityDetailWithMenu(Activity activity);

Specifica Semantica:
Stampa i dettagli di un'attività e gestisce il menu interattivo.

Precondizioni:
- activity deve essere un puntatore valido a un'Activity

Postcondizioni:
- L'attività viene visualizzata e l'utente può interagire attraverso il menu

Side Effects:
- Input/output da console
- Potenziale modifica dello stato dell'oggetto Activity

## printActivityToFile

Specifica Sintattica:
void printActivityToFile(Activity activity, FILE* file);

Specifica Semantica:
Stampa tutti i dettagli di un'attività su un file in un formato leggibile. Il suo scopo principale è permettere il test delle modifiche apportate a un'attività.

Precondizioni:
- file deve essere un puntatore valido a un FILE aperto in modalità scrittura

Postcondizioni:
- I dettagli dell'attività vengono scritti nel file se entrambi i parametri non sono NULL

Side Effects:
Output su file.

## printActivityForListToScreenOrFile

Specifica Sintattica:
void printActivityForListToScreenOrFile(Activity activity, FILE* file);

Specifica Semantica:
Stampa un'attività in formato lista (singola riga) su schermo o file.

Precondizioni:
Nessuna.

Postcondizioni:
- Stampa su stdout se file è NULL
- Stampa su file se file non è NULL
- Nessun output se activity è NULL

Side Effects:
Output su stdout o scrittura su file.

## printActivityForList

Specifica Sintattica:
void printActivityForList(Activity activity);

Specifica Semantica:
Stampa un'attività in formato lista sullo schermo.

Precondizioni:
Nessuna.

Postcondizioni:
- L'attività viene stampata su stdout in formato lista

Side Effects:
Output su stdout.

## printActivityForListToFile

Specifica Sintattica:
void printActivityForListToFile(Activity activity, FILE* file);

Specifica Semantica:
Stampa un'attività in formato lista su un file.

Precondizioni:
- file deve essere un puntatore valido a un FILE aperto in modalità scrittura

Postcondizioni:
- L'attività viene scritta nel file in formato lista se file non è NULL

Side Effects:
Scrittura su file.

## printActivityProgressForListToScreenOrFile

Specifica Sintattica:
void printActivityProgressForListToScreenOrFile(Activity activity, FILE* file);

Specifica Semantica:
Stampa un'attività 'non completata' con informazioni sul progresso (come singola riga) su schermo o file.
Differisce da printActivityForListToScreenOrFile nella quantità di dettagli inclusi nella riga stampata.

Precondizioni:
Nessuna.

Postcondizioni:
- Stampa su stdout se file è NULL
- Stampa su file se file non è NULL
- Nessun output se activity è NULL o completata

Side Effects:
Output su stdout o scrittura su file.

## printActivityProgressForList

Specifica Sintattica:
void printActivityProgressForList(Activity activity);

Specifica Semantica:
Stampa un'attività 'non completata' con informazioni sul progresso sullo schermo.

Precondizioni:
Nessuna.

Postcondizioni:
- L'attività viene stampata su stdout con informazioni sul progresso

Side Effects:
Output su stdout.

## printActivityProgressForListToFile

Specifica Sintattica:
void printActivityProgressForListToFile(Activity activity, FILE* file);

Specifica Semantica:
Stampa un'attività 'non completata' con informazioni sul progresso su un file.

Precondizioni:
- file deve essere un puntatore valido a un FILE aperto in modalità scrittura

Postcondizioni:
- L'attività viene scritta nel file con informazioni sul progresso se file non è NULL

Side Effects:
Scrittura su file.

## readActivityFromFile

Specifica Sintattica:
Activity readActivityFromFile(FILE* file);

Specifica Semantica:
Legge un'attività da un file (formato: 10 righe per attività). Legge il formato scritto da saveActivityToFile.

Precondizioni:
- file deve essere un puntatore valido a un FILE aperto in modalità lettura
- Il file deve contenere almeno 10 righe nel formato corretto

Postcondizioni:
- Restituisce un puntatore all'Activity letta dal file
- Restituisce NULL se fallisce nel leggere 10 righe o se l'allocazione fallisce

Side Effects:
- Lettura da file
- Allocazione di memoria heap
- Avanza il puntatore del file

## saveActivityToFile

Specifica Sintattica:
void saveActivityToFile(FILE* file, Activity activity);

Specifica Semantica:
Salva un'attività su un file utilizzando un formato a 10 righe. Scrive nello stesso formato che readActivityFromFile legge.

Precondizioni:
- file deve essere un puntatore valido a un FILE aperto in modalità scrittura

Postcondizioni:
- L'attività viene scritta nel file in 10 righe se activity non è NULL
- Nessun output se activity è NULL

Side Effects:
Scrittura su file.

# File "activities_container_helper"

## inOrderSaveActivitiesToFile

Specifica Sintattica:
void inOrderSaveActivitiesToFile(FILE* file, TreeNode root);

Specifica Semantica:
Salva ricorsivamente tutte le attività dell'albero su file in ordine in-order.

Precondizioni:
- 'file' deve essere un file aperto in scrittura

Postcondizioni:
- Tutte le attività sono salvate su file in ordine crescente di ID

Side Effects:
- Scrittura su file tramite 'saveActivityToFile()'

## saveActivitiesFromTreeToFile

Specifica Sintattica:
int saveActivitiesFromTreeToFile(const char* filename, TreeNode root);

Specifica Semantica:
Salva tutte le attività dall'albero su un file.

Precondizioni:
- Nessuna

Postcondizioni:
- Se 'root == NULL' o 'filename != NULL', restituisce 1 (errore)
- Se non riesce ad aprire il file, restituisce 1 (errore)
- Altrimenti, salva tutte le attività e restituisce 0 (successo)

Side Effects:
- Apertura e scrittura del file
- Output su stdout (messaggi informativi)

## createNewActivityFromUserInput

Specifica Sintattica:
Activity createNewActivityFromUserInput();

Specifica Semantica:
Interagisce con l’utente per creare una nuova attività.

Precondizioni:
- Nessuna

Postcondizioni:
- Richiede i dati all’utente, crea e restituisce la nuova attività

Side Effects:
- Interazione con l’utente (input/output)
- Chiamate a funzioni di input utente
- Chiamata a 'time()' per timestamp
- Alloca memoria per la nuova attività

## printInOrder

Specifica Sintattica:
void printInOrder(TreeNode root);

Specifica Semantica:
Esegue una visita in-order dell’albero stampando le attività (una per riga).

Precondizioni:
- Nessuna

Postcondizioni:
- Tutte le attività nell’albero sono stampate in ordine crescente di ID

Side Effects:
- Output su stdout tramite 'printActivityForList()'

## printAllActivities

Specifica Sintattica:
void printAllActivities(TreeNode root);

Specifica Semantica:
Stampa tutte le attività (formato lista, una per riga) dall’albero in ordine crescente di ID.

Precondizioni:
- Nessuna

Postcondizioni:
- Se 'root != NULL', stampa tutte le attività con intestazioni

Side Effects:
- Output su stdout

## printInOrderToFile

Specifica Sintattica:
void printInOrderToFile(TreeNode root, FILE* file);

Specifica Semantica:
Stampa ricorsivamente le attività dell'albero su file (formato lista, una per riga) in ordine in-order.

Precondizioni:
- 'file' deve essere aperto in scrittura

Postcondizioni:
- Tutte le attività sono stampate su file in ordine crescente di ID

Side Effects:
- Scrittura su file

## printAllActivitiesToFile

Specifica Sintattica:
void printAllActivitiesToFile(TreeNode root, FILE* file);

Specifica Semantica:
Stampa tutte le attività del contenitore su file (formato lista, una per riga).

Precondizioni:
- 'file' deve essere aperto in scrittura

Postcondizioni:
- Se entrambi i parametri sono validi, stampa tutte le attività su file

Side Effects:
- Scrittura su file

## printInOrderProgress

Specifica Sintattica:
void printInOrderProgress(TreeNode root);

Specifica Semantica:
Stampa ricorsivamente il progresso delle attività dall'albero (formato lista con progresso, una per riga) in ordine in-order.

Precondizioni:
- Nessuna

Postcondizioni:
- Il progresso di tutte le attività è stampato in ordine crescente di ID

Side Effects:
- Output su stdout tramite 'printActivityProgressForList()'

## printTreeActivitiesProgress

Specifica Sintattica:
void printTreeActivitiesProgress(TreeNode root);

Specifica Semantica:
Stampa il progresso di tutte le attività del contenitore (formato lista con progresso, una per riga) con intestazioni.

Precondizioni:
- Nessuna

Postcondizioni:
- Se 'root != NULL', stampa il progresso di tutte le attività nell'albero

Side Effects:
- Output su stdout

## printInOrderProgressToFile

Specifica Sintattica:
void printInOrderProgressToFile(TreeNode root, FILE* file);

Specifica Semantica:
Stampa ricorsivamente il progresso delle attività su file (formato lista con progresso, una per riga) in ordine in-order.

Precondizioni:
- 'file' deve essere aperto in scrittura

Postcondizioni:
- Il progresso di tutte le attività è stampato su file

Side Effects:
- Scrittura su file

## printTreeActivitiesProgressToFile

Specifica Sintattica:
void printTreeActivitiesProgressToFile(TreeNode root, FILE* file);

Specifica Semantica:

Stampa il progresso di tutte le attività del contenitore su file (formato lista con progresso, una per riga).

Precondizioni:
- 'file' deve essere aperto in scrittura

Postcondizioni:
- Se entrambi i parametri sono validi, stampa il progresso con intestazioni

Side Effects:
- Scrittura su file

## buildInOrderSupportListsForActivitiesReport

Specifica Sintattica:
```
void buildInOrderSupportListsForActivitiesReport(Node* root,
    ActivitiesContainerSupportList completedList,
    ActivitiesContainerSupportList ongoingList,
    ActivitiesContainerSupportList expiredList,
    ActivitiesContainerSupportList yetToBeginList,
    time_t beginDate, time_t nowDate);
```

Specifica Semantica:
Costruisce liste di supporto categorizzando le attività per stato in un periodo dato.

Precondizioni:
- Tutte le liste di supporto devono essere inizializzate
- 'beginDate <= nowDate'

Postcondizioni:
- Le attività sono categorizzate in base al loro stato:
  - 'completedList': attività completate nel periodo specificato
  - 'ongoingList': attività in corso
  - 'expiredList': attività scadute (nessuna attività completata qui)
  - 'yetToBeginList': attività non ancora iniziate

Side Effects:
- Allocazione di memoria

## printTreeActivitiesReport

Specifica Sintattica:
```
void printTreeActivitiesReport(TreeNode root);
```

Specifica Semantica:
Genera e stampa un report dettagliato delle attività categorizzate per stato e periodo.

Precondizioni:
- Nessuna

Postcondizioni:
- Se 'root' è valido, stampa un report completo con attività categorizzate
- Interagisce con l’utente per definire il periodo del report

Side Effects:
- Interazione con l’utente (input/output)
- Output su stdout
- Chiamate a funzioni di gestione del tempo
- Alloca e dealloca liste di supporto temporanee

## printTreeActivitiesReportToFile

Specifica Sintattica:
void printTreeActivitiesReportToFile(TreeNode root, time_t beginDate, FILE* file);

Specifica Semantica:
Stampa un report delle attività su file per un periodo specificato.

Precondizioni:
- 'file' deve essere aperto in scrittura
- 'beginDate >= 0'

Postcondizioni:
- Se tutti i parametri sono validi, stampa il report su file

Side Effects:
- Scrittura su file
- Alloca e dealloca liste di supporto temporanee

# File "utils"

## max

Specifica Sintattica:
int max(int a, int b)

Specifica Semantica:
Restituisce il valore maggiore tra due interi.

Precondizioni:
- 'a' e 'b' devono essere valori interi validi

Postcondizioni:
- Il valore restituito è uguale ad 'a' se 'a > b', altrimenti è uguale a 'b'
- Il valore restituito è sempre maggiore o uguale a entrambi i parametri di input

Side Effects:
- Nessuno

## minToHoursAnMinutes

Specifica Sintattica:
void minToHoursAnMinutes(unsigned int minIn, unsigned int * hoursOut, unsigned int * minOut)

Specifica Semantica:
Converte un numero di minuti in ore e minuti (es.: minIn=80 ==> hoursOut=1, minOut=20).

Precondizioni:
- 'minIn' deve essere un valore unsigned int valido
- 'hoursOut' deve essere un puntatore valido a unsigned int
- 'minOut' deve essere un puntatore valido a unsigned int

Postcondizioni:
- '*hoursOut' contiene il numero di ore complete (minIn / 60)
- '*minOut' contiene i minuti restanti (minIn % 60)
- '*minOut' è sempre compreso tra 0 e 59

Side Effects:
- Modifica i valori puntati da 'hoursOut' e 'minOut'

## readLine

Specifica Sintattica:
char* readLine(FILE* file)

Specifica Semantica:

Legge una riga da un file e alloca dinamicamente memoria per memorizzare la stringa letta.

Precondizioni:
- 'file' deve essere un puntatore valido a FILE aperto in modalità lettura
- Il file deve essere accessibile in lettura

Postcondizioni:
- Restituisce un puntatore a una stringa allocata dinamicamente contenente la riga letta (senza il carattere newline)
- Restituisce NULL in caso di errore di lettura o di allocazione di memoria
- La stringa restituita è terminata da '\0'

Side Effects:
- Allocazione di memoria sper la stringa (deve essere liberata dal chiamante)
- Modifica della posizione del puntatore del file

## copyString

Specifica Sintattica:
char* copyString(const char* str)

Specifica Semantica:
Crea una copia di una stringa allocando dinamicamente la memoria necessaria.

Precondizioni:
- 'str' può essere NULL oppure un puntatore valido a una stringa terminata da null

Postcondizioni:
- Se 'str' è NULL, restituisce NULL
- Se 'str' è valido, restituisce un puntatore a una nuova stringa identica allocata dinamicamente
- Restituisce NULL se l'allocazione di memoria fallisce

Side Effects:
- Allocazione di memoria sul heap (deve essere liberata dal chiamante)

## getChoiceWithLimits

Specifica Sintattica:
unsigned int getChoiceWithLimits(unsigned int minLimit, unsigned int maxLimit)

Specifica Semantica:
Richiede all'utente di inserire un intero compreso tra due limiti specificati, ripetendo la richiesta finché l'input non è valido.

Precondizioni:
- 'minLimit' <= 'maxLimit'
- Lo standard input deve essere disponibile per la lettura

Postcondizioni:
- Restituisce un valore unsigned int compreso tra 'minLimit' e 'maxLimit' (inclusi)

- Il valore restituito è quello inserito dall'utente

Side Effects:
- Output su stdout (messaggi di errore)
- Lettura da stdin
- Possibile loop infinito se l'input continua a essere non valido

## getChoice

Specifica Sintattica:
unsigned int getChoice(unsigned int limit)

Specifica Semantica:
Richiede all'utente di inserire un intero tra 0 e un limite specificato. Usata principalmente nella gestione dei menu.

Precondizioni:
- 'limit' deve essere un valore unsigned int valido
- Lo standard input deve essere disponibile per la lettura

Postcondizioni:
- Restituisce un valore unsigned int tra 0 e 'limit' (inclusi)

Side Effects:
- Uguali a quelli di 'getChoiceWithLimits'

## getInfoFromUser

Specifica Sintattica:
char* getInfoFromUser(const char* prompt)

Specifica Semantica:
Mostra un prompt all'utente e legge una stringa di input, allocando dinamicamente memoria per memorizzarla.

Precondizioni:
- 'prompt' può essere NULL oppure un puntatore valido a una stringa
- Lo standard input deve essere disponibile per la lettura

Postcondizioni:
- Se l'utente inserisce una stringa non vuota, restituisce un puntatore alla stringa allocata dinamicamente
- Se l'utente inserisce una stringa vuota o si verifica un errore, restituisce NULL
- La stringa restituita non contiene il carattere newline

Side Effects:
- Output su stdout (prompt)
- Lettura da stdin
- Allocazione di memoria sul heap (deve essere liberata dal chiamante)

## dateToEpoch

Specifica Sintattica:
time_t dateToEpoch(int year, int month, int day, int hour, int min)

Specifica Semantica:
Converte una data e ora specificate nel formato timestamp Unix (epoch time).

Precondizioni:
- 'year' deve essere compreso tra 1900 e 2037
- 'month' tra 1 e 12
- 'day' tra 1 e 31
- 'hour' tra 0 e 23
- 'min' tra 0 e 59

Postcondizioni:
- Se tutti i parametri sono validi, restituisce il timestamp Unix corrispondente
- Se uno o più parametri non sono validi, restituisce 0
- Il timestamp rappresenta la data/ora specificata

Side Effects:
- Nessuno

## getDateFromUser

Specifica Sintattica:
time_t getDateFromUser()

Specifica Semantica:
Richiede interattivamente all'utente di inserire anno, mese, giorno, ora e minuti, validando ciascun input e restituendo il corrispondente timestamp Unix.

Precondizioni:
- Lo standard input deve essere disponibile per la lettura
- Lo standard output deve essere disponibile per la scrittura

Postcondizioni:
- Restituisce un timestamp Unix valido corrispondente alla data/ora inserita dall'utente
- Tutti i valori inseriti rispettano i limiti specificati (anno: 2000-2037, ecc.)
- Tiene conto degli anni bisestili per la validazione del giorno

Side Effects:
- Output su stdout (prompt e messaggi di errore)
- Lettura da stdin
- Possibili loop per la richiesta di input valido

## displayConfirmMenu

Specifica Sintattica:
void displayConfirmMenu(const char* confirmInfo)

Specifica Semantica:
Mostra un menu di conferma con opzioni "Continua" e "Annulla", preceduto da un messaggio informativo opzionale.

Precondizioni:
- 'confirmInfo' può essere NULL oppure un puntatore valido a una stringa
- Lo standard output deve essere disponibile per la scrittura

Postcondizioni:
- Mostra il menu di conferma formattato
- Se 'confirmInfo' non è NULL, mostra il messaggio di avviso

Side Effects:
- Output su stdout

## getConfirmMenuChoice

Specifica Sintattica:
int getConfirmMenuChoice(const char* confirmInfo)

Specifica Semantica:
Mostra un menu di conferma e richiede all'utente di scegliere tra "Continua" (1) e "Annulla" (0).

Precondizioni:
- 'confirmInfo' può essere NULL oppure un puntatore valido a una stringa
- Lo standard input e output devono essere disponibili

Postcondizioni:
- Restituisce 0 o 1 in base alla scelta dell'utente
- L'utente deve necessariamente scegliere una delle due opzioni valide

Side Effects:
- Output su stdout (menu)
- Lettura da stdin
- Possibili messaggi di errore per input non valido (tramite getChoice)

# File "main"

## displayStartMenu

Specifica Sintattica:
void displayStartMenu();

Specifica Semantica:
Mostra il menu iniziale dell'applicazione con le opzioni disponibili per l'avvio del programma.

Precondizioni:
- Nessuna precondizione particolare

Postcondizioni:
- Il menu iniziale è stato stampato sull'output standard
- Il cursore è posizionato dopo la richiesta di scelta

Side Effects:
- Output su stdout

## displayMainMenu

Specifica Sintattica:
void displayMainMenu();

Specifica Semantica:
Mostra il menu principale dell'applicazione con tutte le operazioni disponibili per la gestione delle attività.

Precondizioni:
- Nessuna precondizione particolare

Postcondizioni:
- Il menu principale è stato stampato sull'output standard
- Il cursore è posizionato dopo la richiesta di scelta

Side Effects:
- Output su stdout

## handleStartMenu

Specifica Sintattica:
ActivitiesContainer handleStartMenu();

Specifica Semantica:
Gestisce l'interazione con il menu iniziale, acquisisce la scelta dell'utente e inizializza

il contenitore delle attività di conseguenza.

Precondizioni:
- Nessuna

Postcondizioni:
- Restituisce un ActivitiesContainer valido se l'utente sceglie l'opzione 1 o 2
- Restituisce NULL se l'utente sceglie di uscire (opzione 0)
- La memoria allocata per userFile viene liberata se necessario

Side Effects:
- Input da stdin
- Output su stdout
- Allocazione/deallocazione dinamica della memoria
- Possibile lettura da file

## handleMainMenu

Specifica Sintattica:
int handleMainMenu(ActivitiesContainer container);

Specifica Semantica:
Gestisce l'interazione con il menu principale, esegue l'operazione scelta dall'utente sul contenitore delle attività.

Precondizioni:
- container != NULL
- container deve essere un contenitore di attività valido

Postcondizioni:
- Restituisce -1 se l'utente sceglie di uscire (opzione 0)
- Restituisce il numero dell'opzione scelta negli altri casi
- Il contenitore può essere modificato dalle operazioni 2 e 3
- Se l'opzione 0 è confermata, il contenitore viene deallocato e il programma termina

Side Effects:
- Input da stdin
- Output su stdout
- Possibile modifica del contenitore delle attività
- Possibile allocazione/deallocazione dinamica della memoria
- Possibile scrittura su file (opzione 7)
- Possibile deallocazione completa della memoria (opzione 0)

# File "test_main"

## compareFiles

Specifica Sintattica:
int compareFiles(const char* filenameA, const char* filenameB);

Specifica Semantica:
Confronta due file di testo riga per riga per determinare se sono identici.

Precondizioni:
- 'filenameA' e 'filenameB' devono essere puntatori validi a stringhe non NULL
- I file specificati devono esistere ed essere leggibili

Postcondizioni:
- Restituisce 0 se i file sono identici riga per riga
- Restituisce 1 se i file differiscono o si verifica un errore

Side Effects:
- Apre (e chiude) entrambi i file in modalità lettura
- Stampa messaggi di errore su stdout in caso di problemi di apertura file
- Alloca e dealloca dinamicamente memoria per ogni riga letta
- Modifica i puntatori di posizione dei file durante la lettura

## tc_1

Specifica Sintattica:
int tc_1();

Specifica Semantica:
Test case 1: verifica la corretta esecuzione delle operazioni di caricamento file, creazione dati e salvataggio.

Precondizioni:
- Il file "tc_1.txt" deve esistere ed essere accessibile
- Il file "tc_1_oracle.txt" deve esistere per il confronto
- Le funzioni del contenitore delle attività devono essere implementate

Postcondizioni:
- Restituisce 0 se il test è superato (file di output uguale all’oracolo)
- Restituisce 1 se il test fallisce

Side Effects:
- Crea il file "tc_1_output.txt"
- Alloca e dealloca memoria per il contenitore delle attività
- Modifica il filesystem creando file temporanei

## tc_2

Specifica Sintattica:
int tc_2();

Specifica Semantica:
Test case 2: carica le attività da file e stampa e verifica i dettagli di una specifica attività.

Precondizioni:
- Il file "tc_2.txt" deve esistere ed essere accessibile
- Il file "tc_2_oracle.txt" deve esistere per il confronto
- Deve esistere un'attività con ID 1 nel file di input

Postcondizioni:
- Restituisce 0 se il test è superato
- Restituisce 1 se il test fallisce o l'attività non viene trovata

Side Effects:
- Crea il file "tc_2_output.txt"
- Alloca e dealloca memoria per il contenitore
- Apre e chiude file per la scrittura

## tc_3_to_11

Specifica Sintattica:
int tc_3_to_11(int numTest, char* fileIn, char* fileOut, char* fileOracle);

Specifica Semantica:
Funzione generica per i test case da 3 a 11: carica un'attività, modifica un campo specifico e verifica l'output.

| Test Case | Descrizione del Test | Campo Coinvolto |
|---|---|---|
| TC 3 | Modifica e verifica il nome dell'attività | name |
| TC 4 | Modifica e verifica la descrizione dell'attività | descr |
| TC 5 | Modifica e verifica il corso associato | course |
| TC 6 | Modifica e verifica la data di inserimento | insertDate |
| TC 7 | Modifica e verifica la data di scadenza | expiryDate |
| TC 8 | Modifica e verifica la data di completamento | completionDate |
| TC 9 | Modifica e verifica il tempo totale previsto | totalTime |
| TC 10 | Modifica e verifica il tempo già utilizzato | usedTime |
| TC 11 | Modifica e verifica la priorità dell'attività | priority |

Precondizioni:
- 'numTest' deve essere un valore tra 3 e 11
- 'fileIn', 'fileOut' e 'fileOracle' devono essere puntatori a stringhe valide

- Il file di input deve esistere e contenere un’attività con ID 1
- Il file oracolo deve esistere per il confronto

Postcondizioni:
- Restituisce 0 se il test è superato
- Restituisce 1 se il test fallisce o si verificano errori

Side Effects:
- Crea file di output temporanei
- Modifica lo stato interno dell’oggetto Activity
- Stampa messaggi di errore per test non supportati
- Alloca e dealloca memoria per il contenitore

## tc_12

Specifica Sintattica:
int tc_12();

Specifica Semantica:
Test case 12: carica le attività da file, stampa e verifica la lista completa delle attività.

Precondizioni:
- Il file "tc_12.txt" deve esistere ed essere accessibile
- Il file "tc_12_oracle.txt" deve esistere per il confronto

Postcondizioni:
- Restituisce 0 se il test è superato
- Restituisce 1 se il test fallisce

Side Effects:
- Crea il file "tc_12_output.txt"
- Alloca e dealloca memoria per il contenitore
- Apre e chiude file per la scrittura

## tc_13

Specifica Sintattica:
int tc_13();

Specifica Semantica:
Test case 13: carica le attività da file, stampa e verifica il report di avanzamento.

Precondizioni:
- Il file "tc_13.txt" deve esistere ed essere accessibile
- Il file "tc_13_oracle.txt" deve esistere per il confronto

Postcondizioni:
- Restituisce 0 se il test è superato
- Restituisce 1 se il test fallisce

Side Effects:
- Crea il file "tc_13_output.txt"
- Alloca e dealloca memoria per il contenitore
- Apre e chiude file per la scrittura

## tc_14

Specifica Sintattica:
int tc_14();

Specifica Semantica:
Test case 14: carica le attività da file, stampa e verifica il report settimanale.
Include anche: categorizzazione delle attività in sezioni (completate, in corso, ecc.) e ordinamento.

Precondizioni:
- Il file "tc_14.txt" deve esistere ed essere accessibile
- Il file "tc_14_oracle.txt" deve esistere per il confronto

Postcondizioni:
- Restituisce 0 se il test è superato
- Restituisce 1 se il test fallisce

Side Effects:
- Crea il file "tc_14_output.txt"
- Alloca e dealloca memoria per il contenitore
- Apre e chiude file per la scrittura

## tc_15

Specifica Sintattica:
int tc_15();

Specifica Semantica:
Test case 15: verifica la corretta esecuzione del caricamento, creazione e salvataggio di più attività.

Precondizioni:
- Il file "tc_15.txt" deve esistere ed essere accessibile
- Il file "tc_15_oracle.txt" deve esistere per il confronto
- Le funzioni del contenitore delle attività devono essere implementate

Postcondizioni:
- Restituisce 0 se il test è superato (file di output uguale all'oracolo)
- Restituisce 1 se il test fallisce

Side Effects:
- Crea il file "tc_15_output.txt"
- Alloca e dealloca memoria per il contenitore delle attività
- Modifica il filesystem creando file temporanei

## tc_16

Specifica Sintattica:
int tc_16();

Specifica Semantica:
Test case 16: carica attività da file, modifica i dati e stampa/verifica il report settimanale.
Include anche: categorizzazione in sezioni (completate, in corso, ecc.) e ordinamento.

Precondizioni:
- Il file "tc_16.txt" deve esistere ed essere accessibile
- Il file "tc_16_oracle.txt" deve esistere per il confronto

Postcondizioni:
- Restituisce 0 se il test è superato
- Restituisce 1 se il test fallisce

Side Effects:
- Crea il file "tc_16_output.txt"
- Alloca e dealloca memoria per il contenitore
- Apre e chiude file per la scrittura

## execTest

Specifica Sintattica:
void execTest(int numTest, FILE* fileWithTestsResult);

Specifica Semantica:
Esegue un test specifico e registra i risultati su console e su file.

Precondizioni:
- 'numTest' deve essere un valore tra 1 e 16
- 'fileWithTestsResult' può essere NULL o un puntatore a file valido aperto in scrittura

Postcondizioni:
- Il test specificato viene eseguito
- I risultati sono stampati su console
- Se il file è valido, i risultati sono anche scritti su file

Side Effects:
- Stampa messaggi su stdout
- Scrive su file se il puntatore è valido
- Può stampare messaggi di errore per test non supportati

## main (di test_main)

Specifica Sintattica:
int main();

Specifica Semantica:
Funzione principale che coordina l'esecuzione di tutti i test case.

Precondizioni:
- Tutti i file di test e oracolo devono essere presenti nella directory corrente
- Il sistema deve avere i permessi di scrittura per creare i file di output

Postcondizioni:
- Restituisce 0 (implicito) al termine dell’esecuzione
- Tutti i test da 1 a 16 vengono eseguiti in sequenza

Side Effects:
- Crea il file "TESTS_RESULT.txt" se possibile
- Stampa messaggi informativi ed errori su stdout
- Apre e chiude file per la scrittura dei risultati
- Esegue un ciclo che modifica lo stato del sistema tramite vari test