Study Activity Planning and Monitoring System

Developed by skrmrcode

Github: https://github.com/skrmrcode/psd-project

1. Motivation for ADT Choice

This project is designed to manage and track the progress of study activities efficiently. The Abstract Data Types (ADTs) selected for this system are:

- *`activity`: to encapsulate all data related to a single study task, such as description, course, deadline, priority, and state.*

- *`heap`: a max-heap is used to maintain and access activities in order of priority. This ensures that high-priority tasks are handled first.*

These ADTs were chosen to enforce modularity and abstraction, enabling encapsulated logic for scheduling and managing academic tasks.

2. How to Use and Interact with the System

To use this program, follow the steps below:

1. Save all the provided source files (`.c` and `.h`) in the same folder on your computer.

2. Open a terminal and navigate to that folder.

3. Type `make` to compile the entire project.

4. After successful compilation:

   - Run the main program using `./main.exe`

   - Run the test suite using `./run_test_suite.exe`

5. To clean object files and temporary output, type `make clean`

Note: the `make clean` command will not delete the `activities.txt` file. This ensures that user data persists across sessions. To reset and delete all saved activities, manually remove the file `activities.txt`.

3. System Design

The system is structured into modular components:

- `activity.c/.h`: defines the structure and operations on study activities.

- `heap.c/.h`: maintains the activities in a priority heap structure.

- `menu.c/.h`: handles the user interface and menu interactions.

- `utils.c/.h`: provides auxiliary functions such as input validation and file management.

- `test.c/.h`: implements the test suite to validate system functionalities.

Each module hides its internal data using opaque pointers and exposes only essential functions through its header file, maintaining strong information hiding.

4. Syntactic and Semantic Specification

The complete specifications of all functions, including their syntax, semantics, preconditions, and postconditions, are provided in the next section of the document.

Activity Module Functions

newAct

Syntax: activity newAct(char *descr, char *course, char *deadline, int time, priority p);

Semantics: Creates and initializes a new activity with the given parameters.

Preconditions: descr, course, and deadline must be valid strings; time must be a positive integer; p must be a valid priority enum.

Postconditions: Returns a pointer to a newly allocated activity with the given data and default state as UNCOMPLETED.

updateProgress

Syntax: void updateProgress(activity a, int hours);

Semantics: Increments the time spent on the activity and updates its state accordingly.

Preconditions: a must be a valid activity pointer; hours must be a non-negative integer.

Postconditions: Updates the internal timeSpentAct field. The state becomes COMPLETED if the estimated time is reached or exceeded.

getDescr

Syntax: char* getDescr(activity a);

Semantics: Returns the description of the activity.

Preconditions: a must be a valid activity pointer.

Postconditions: Returns a string with the description.

getCourse

Syntax: char* getCourse(activity a);

Semantics: Returns the course name associated with the activity.

Preconditions: a must be a valid activity pointer.

Postconditions: Returns a string with the course name.

getDeadline

Syntax: char* getDeadline(activity a);

Semantics: Returns the deadline of the activity.

Preconditions: a must be a valid activity pointer.

Postconditions: Returns a string representing the deadline.

getCreatedDate

Syntax: char* getCreatedDate(activity a);

Semantics: Returns the creation date of the activity.

Preconditions: a must be a valid activity pointer.

Postconditions: Returns a string in the format 'YYYY-MM-DD'.

getWorkedHours

Syntax: int getWorkedHours(activity a);

Semantics: Returns the number of hours spent on the activity so far.

Preconditions: a must be a valid activity pointer.

Postconditions: Returns a non-negative integer.

getTimeReq

Syntax: int getTimeReq(activity a);

Semantics: Returns the estimated total time required for the activity.

Preconditions: a must be a valid activity pointer.

Postconditions: Returns a positive integer.

getP

Syntax: priority getP(activity a);

Semantics: Returns the priority of the activity.

Preconditions: a must be a valid activity pointer.

Postconditions: Returns a value of type priority.

getState

Syntax: c_state getState(activity a);

Semantics: Returns the current state of the activity.

Preconditions: a must be a valid activity pointer.

Postconditions: Returns a value of type c_state.

setCreatedDate

Syntax: void setCreatedDate(activity a, const char* date);

Semantics: Sets the creation date of the activity.

Preconditions: a must be a valid activity pointer; date must be a valid date string.

Postconditions: The internal created field is updated with the new date.

priorityToStr

Syntax: char* priorityToStr(int p);

Semantics: Converts a priority value to a string.

Preconditions: p should be 1 (Low), 2 (Medium), or 3 (High).

Postconditions: Returns a string representing the priority or '-' if invalid.

printAct

Syntax: void printAct(activity a);

Semantics: Prints a formatted summary of the activity to standard output.

Preconditions: a must be a valid activity pointer.

Postconditions: Outputs activity information to the terminal.

dataConvert

Syntax: time_t dataConvert(char* data_str);

Semantics: Converts a date string into a time_t object.

Preconditions: data_str must be a string in 'YYYY-MM-DD' format.

Postconditions: Returns the corresponding time_t value.

isLate

Syntax: int isLate(activity a);

Semantics: Determines whether the activity is overdue.

Preconditions: a must be a valid activity pointer.

Postconditions: Returns 1 if the current date is past the deadline and the activity is not completed; otherwise returns 0.

Heap Module Functions

swap

Syntax: void swap(activity *a, activity *b);

Semantics: Exchanges the values pointed to by two activity pointers.

Preconditions: a and b must be valid non-null pointers to activity elements.

Postconditions: The values pointed to by a and b are swapped.

heapifyUp

Syntax: void heapifyUp(heap h, int pos);

Semantics: Restores the heap property by moving the element at position pos upward.

Preconditions: h must be a valid heap; pos must be a valid index in the heap (0 ≤ pos < numElem).

Postconditions: The element at position pos is moved to its correct position to maintain heap order.

heapifyDown

Syntax: void heapifyDown(heap h, int pos);

Semantics: Restores the heap property by moving the element at position pos downward.

Preconditions: h must be a valid heap; pos must be a valid index in the heap.

Postconditions: The element is placed in a position that maintains the max-heap structure.

newHeap

Syntax: heap newHeap(int size);

Semantics: Creates a new empty heap with the given capacity.

Preconditions: size must be a positive integer.

Postconditions: Returns a pointer to a newly allocated empty heap with specified capacity and zero elements.

addAct

Syntax: void addAct(heap h, activity a);

Semantics: Inserts a new activity into the heap and maintains heap order.

Preconditions: h must be a valid heap; a must be a valid activity; h must not be full (numElem < capacity).

Postconditions: The activity a is added to the heap in the correct position. The number of elements is incremented.

extractMaxPrio

Syntax: activity extractMaxPrio(heap h);

Semantics: Removes and returns the activity with the highest priority.

Preconditions: h must be a valid heap and must not be empty (numElem > 0).

Postconditions: The top-priority activity is removed from the heap and returned. The heap remains valid.

deleteHeap

Syntax: void deleteHeap(heap h);

Semantics: Frees all memory associated with the heap and its stored activities.

Preconditions: h must be a valid heap pointer.

Postconditions: All activities and internal heap data are deallocated.

printHeap

Syntax: void printHeap(heap h);

Semantics: Prints all the activities in the heap to standard output.

Preconditions: h must be a valid heap.

Postconditions: Each activity is printed using printAct.

getNumElem

Syntax: int getNumElem(heap h);

Semantics: Returns the current number of activities in the heap.

Preconditions: h must be a valid heap.

Postconditions: Returns an integer ≥ 0.

getActivity

Syntax: activity getActivity(heap h, int index);

Semantics: Retrieves the activity at the specified index in the heap.

Preconditions: h must be a valid heap; index must satisfy 0 ≤ index < numElem.

Postconditions: Returns a pointer to the activity at the given index, or NULL if out of bounds.

5. Rationale Behind the Test Cases

The test suite is structured to validate the correctness and robustness of all major system functionalities. It covers the following aspects:

- **Activity Creation:** Verifies that all fields are properly initialized when creating a new activity.

- **Progress Update:** Ensures that activity state transitions correctly from UNCOMPLETED to ONGOING and to COMPLETED.

- **Priority Heap Operations:** Tests the correct insertion, ordering, and extraction of activities by priority.

- **Report Generation:** Validates that completed and overdue activities are correctly classified and formatted in the output report.

Each test is logged in `result.txt` with a [PASS]/[FAIL] outcome, ensuring traceability and reproducibility of results.