KMUTT CPE38 Project. 2025

TO DO LIST
APPLICATION

Presented By: Kulchaya Paipinij 3406
Chayanit Kuntanarumitkul 3408
Siripitch Chaiyabutra 3440

01

KMUTT CPE38 Project. 2025

PROBLEM STATEMENT

Traditional to-do lists lack:

- Dynamic priority management
- Efficient search capabilities
- Progress tracking
- Deadline awareness

Real-world Relevance

Students and professionals struggle with deadline management !

KMUTT CPE38 Project. 2025

02

OUR SOLUTION!

To create an comprehensive To-Do List with :

- Automated priority adjustment
- Advanced search capabilities
- Real-time progress tracking
- Edge case handling
- Memory efficiency

Target Users

- University students managing assignments
- Professionals handling multiple projects
- Anyone needing organized task management

03

Project.

KMUTT CPE38

FEATURES

INCLUDING IN 3 GROUPS

01 Task Management and Organization

02 Progress Tracking

03 Advanced Functionalities

01 Task Management and Organization

- Add, edit, and delete tasks
- Set and track due dates
- Tag tasks for categorization
- Assign priorities (High, Medium, Low)

02 Progress Tracking

- Display task statistics
- View completed vs pending tasks
- Generate progress reports
- Assign priorities (High, Medium, Low)

03 Advanced Functionalities

- Automatic priority adjustment
- Date simulation for testing
- Import/Export capabilities
- Undo completed tasks

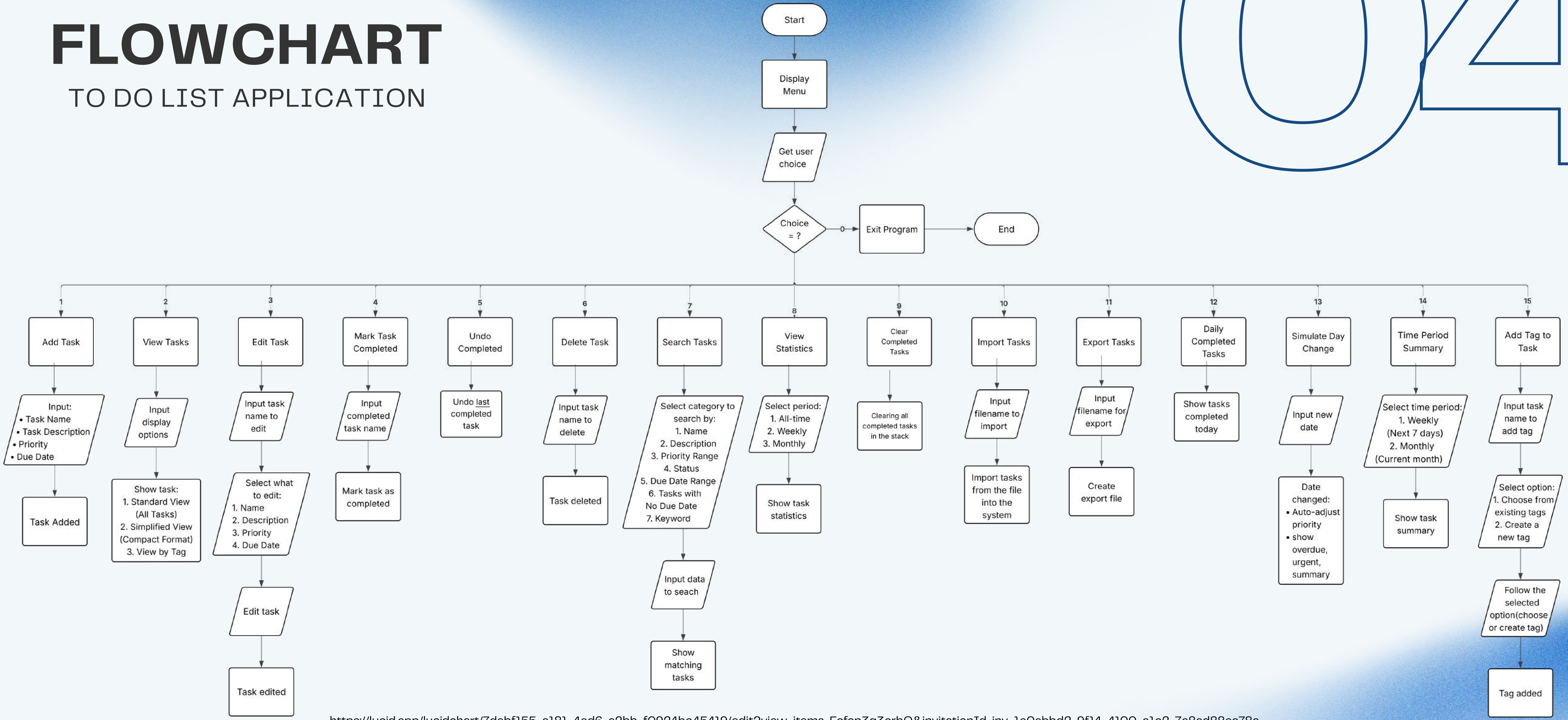
ARCHITECTURE OVERVIEW

```
ToDoList-DSA/
├── main.c                  # Entry point with menu system
├── task_management.c       # Core task operations
├── task_management.h       # Task structures and declarations
├── scheduler.c             # Date handling and scheduling logic
├── scheduler.h             # Date operations and status updates
├── searchandstat.c         # Search and statistics functions
├── searchandstat.h         # Search declarations
├── fileio.c               # Import/Export functionality
├── fileio.h               # File operations declarations
```

FLOWCHART

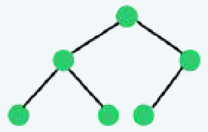
TO DO LIST APPLICATION

04



https://lucid.app/lucidchart/7debf155-a181-4ed6-a2bb-f0924bc45419/edit?view_items=Fcfcp3q3crbQ&invitationId=inv_1e0abbd2-9f14-4100-a1c2-7a8ad88ea78a

DATA STRUCT WE USE IN THE PROJECT



Linked-list (Primary Storage)

- Dynamic task storage
- O(1) insertion at head
- Efficient memory usage

Linked List vs Array: Dynamic vs Fixed size, O(1) vs O(n) insertion

Why ?

- Supports dynamic task management – no need to predefine size
- Efficient insertion and deletion – no shifting required
- Ideal for frequent updates (add/remove tasks anytime)
- Uses memory only when needed – better for unpredictable data size

DATA STRUCT WE USE IN THE PROJECT

Stack (Completed Tasks)

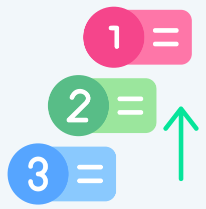
- LIFO for undo operations
- O(1) push/pop
- Perfect for "undo last completed"

Stack vs Queue for Undo: LIFO matches undo behavior

Why ?

- Follows Last-In, First-Out (LIFO) – ideal for reversing recent actions
- Easily tracks user actions – push when an action happens, pop to undo
- Efficient for single-step or multi-step undo
- Perfect for managing sequential edits (add, delete, edit)

DATA STRUCT WE USE IN THE PROJECT



Queue (Task Scheduling)

- FIFO for deadline processing
- O(1) enqueue/dequeue operations
- Manages time-sensitive tasks

Queue for Scheduling: Natural FIFO for processing tasks

Why ?

- Follows First-In, First-Out (FIFO) – tasks are handled in order
- Perfect for processing tasks chronologically (e.g., reminders or due tasks)
- Ensures fair scheduling – first task added is the first to be done
- Simple structure for managing time-based task execution

KMUTT CPE38

05

DATA STRUCT WE USE IN THE PROJECT

Bubble Sort (Task Organization)

- Sorts by priority and due date
- Simple implementation for small datasets
- In-place sorting (O(1) extra space)

Bubble Sort vs Quick Sort: Simplicity for our data size (<1000 tasks)

Why ?

- Much faster for large task lists – average time complexity O(n log n)
- Efficient for sorting tasks by due date, priority, etc.
- Recursively divides and conquers – better performance overall
- Suitable for real-world apps with many dynamic tasks

06

CORE CODE EXAMPLE

Project.

KMUTT CPE38

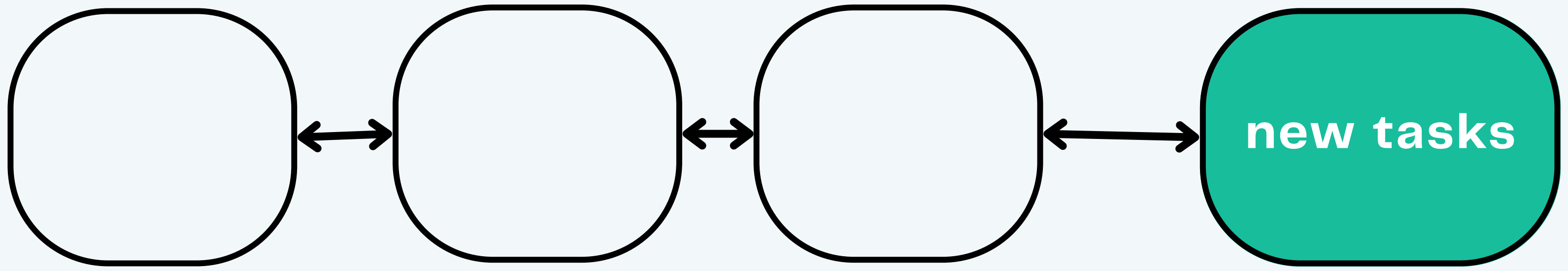
ADD TASKS

CODE

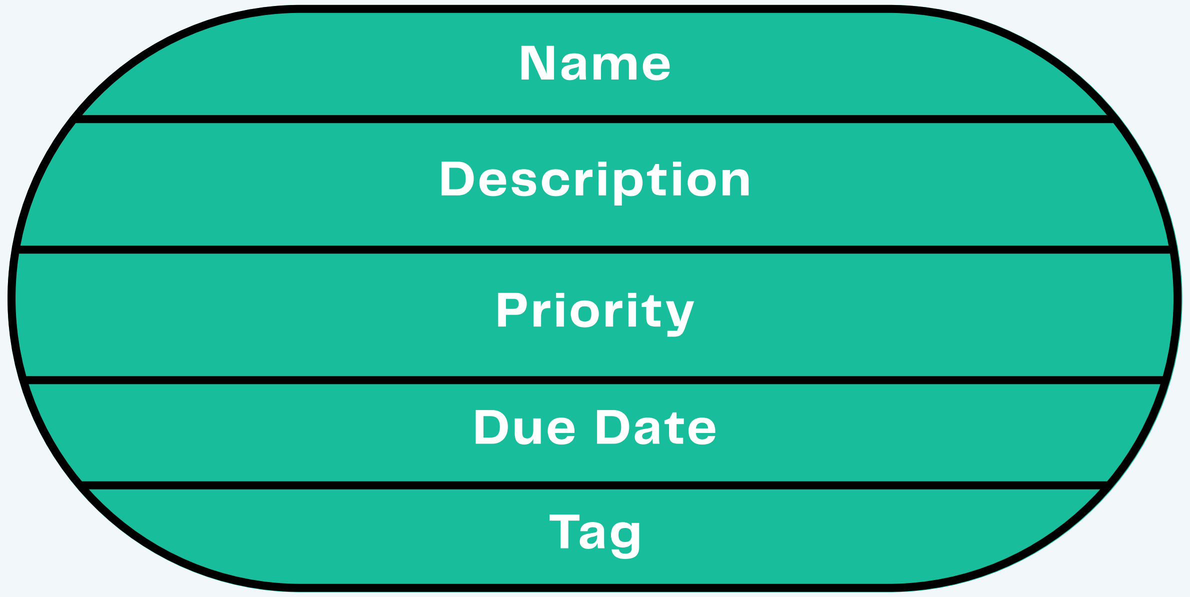
```c
void add(tasklist* list) {
    // 1. Allocate memory
    task* new_task = (task*)malloc(sizeof(task));

    // 2. Input validation
    // Check for duplicate names
    // Validate priority and date

    // 3. Insert at head of linked list
    new_task->next = list->head;
    list->head = new_task;
}
```



LINKED LISTS : STORE TASKS



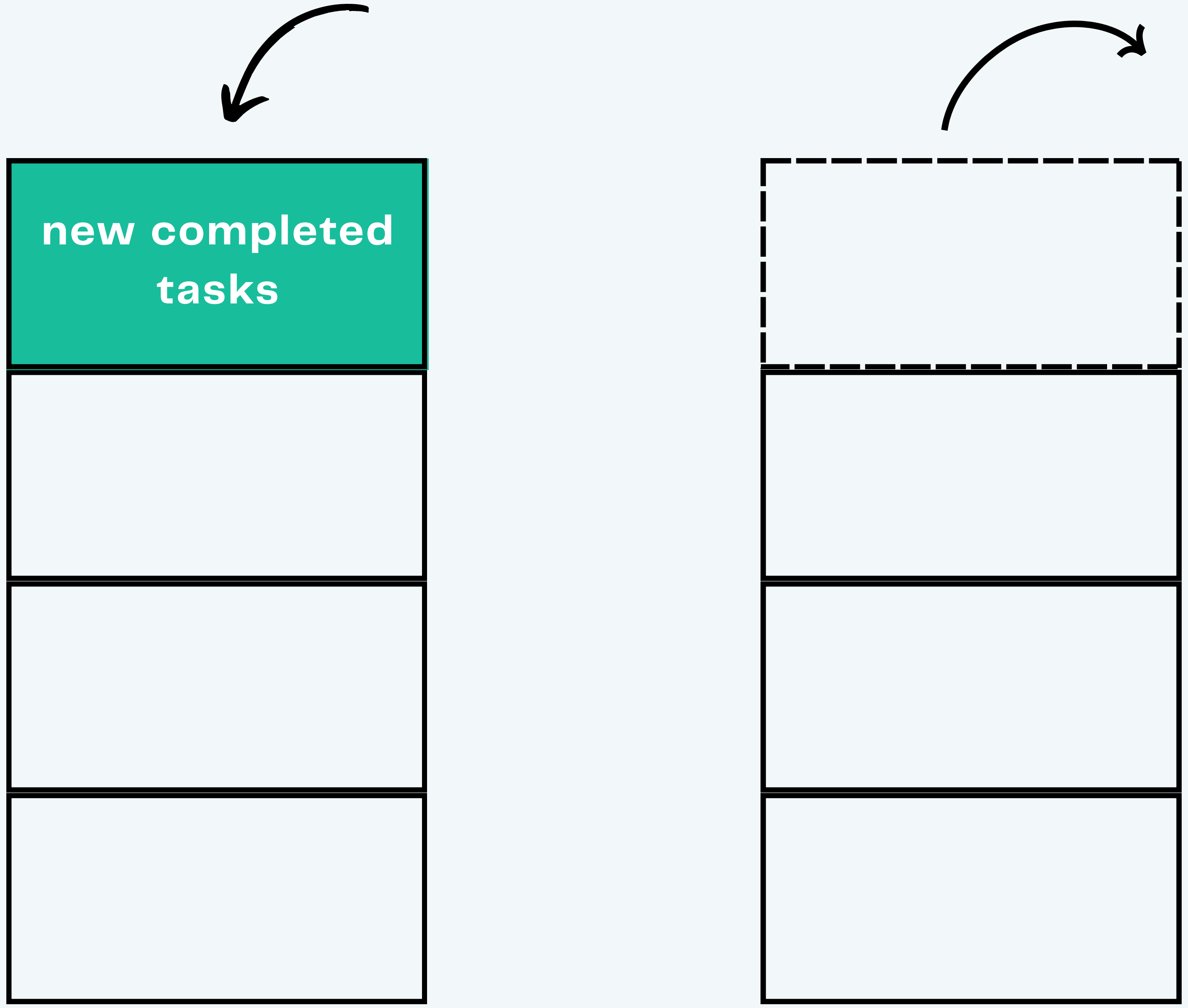
STRUCTURE OF TASKS

COMPLETE AND UNDO

CODE

```c
void complete(tasklist* list, completedstack* stack,
const char* name) {
    // 1. Find task in linked list
    // 2. Remove from active list
    // 3. Push onto completed stack
}

void undoCompleted(tasklist* list, completedstack*
stack) {
    // 1. Pop from stack
    // 2. Reinsert into active list
}
```



COMPLETE

UNDO

COMPLETE STACK

SEARCH (BY PRIORITY)

CODE

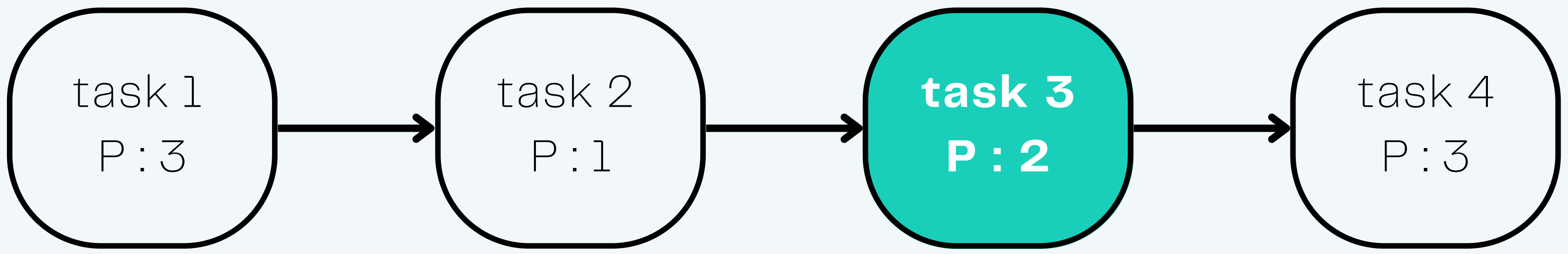
```c
void searchTasks(task* h, completedstack* s, const char* k) {
    int f = 0;

    // Search active tasks
    for (task* t = h; t; t = t->next)
        if (strstr(t->name, k)) { printTaskInfo(t); f = 1; }

    // Search completed tasks
    for (stacknode* n = s->top; n; n = n->next)
        if (strstr(n->task_data->name, k)) { printTaskInfo(n->task_data); f = 1; }

    if (!f) printf("No matching tasks found.\n");
}
```

INITIAL TASKS LISTS (UNSORTED)



EX : SEARCH FOR PRIORITY OF 2

CHECK

- **TASK 1 : PRIORITY 3**
- **TASK 2 : PRIORITY 1**
- **TASK 3 : PRIORITY 2 (FOUNDED)**
- **TASK 4 : PRIORITY 3**

RESULT : TASK 3

EDGE CASE HANDLING

Edge Case	Input	Expected Behavior
Empty name	""	Error message
Whitespace name	" "	Error message
Duplicate name	Existing name	Error message
Invalid date	32/13/2025	Date not set
Out-of-range priority	5	Default to 2 (Medium)
Undo with empty stack	N/A	"No tasks to undo"
Search with unexist name	ddkdoeo (unexist name)	“Tasks not found”

EXAMPLE (SEARCH)

CODE

```c
void searchTasks(task* head, completedstack* stack, const char* keyword) {
    // Edge cases
    if (!head && !stack->top) {
        printf("No tasks to search.\n");
        return;
    }

    if (!keyword || strlen(keyword) == 0) {
        printf("Error: Empty keyword.\n");
        return;
    }

    // Date range validation
    if (search_option == 5 && compareDates(start_date, end_date) > 0) {
        printf("Error: Invalid date range.\n");
        return;
    }

    // Search and handle no results
    int found = 0;
    // ... search code ...

    if (!found) {
        printf("No matching tasks found.\n");
    }
}
```

PROGRAM

```
Select an option: 7

=== Task Search ===
Search by:
1. Name
2. Description
3. Priority Range
4. Status (Pending/Completed/Overdue)
5. Due Date Range
6. Tasks with No Due Date
7. Keyword (search all fields)
Enter your choice (1-7): 1
Enter search keyword: Calculusreading

=== Search Results for 'Calculusreading' ===
--- Pending Tasks ---
--- Completed Tasks ---
No matching tasks found.

Press Enter to continue...
```

TIME COMPLEXITY

Time Complexity Analysis:

Operation	Time Complexity	Actual Time (1000 tasks)	Explanation
Add Task	O(1)	0.004 ms	Linked list head insertion
Search by Name	O(n)	0.022 ms	Linear traversal required
Delete Task	O(n)	0.245 ms	Search + pointer adjustment
Complete Task	O(n)	0.053 ms	Search + stack push
Undo Complete	O(1)	0.002 ms	Stack pop + list insertion
Sort by Priority	$O(n^2)$	2.838 ms	Bubble sort implementation

TIME COMPLEXITY

Data Structure Performance Comparison:

Operation	With Data Structures	Without Data Structures	Improvement
Task Access	O(n) linked list	O(n) array	Same
Undo Operation	O(1) stack	O(n) array shift	Significant
Priority Queue	$O(n^2)$ sort	$O(n^2)$ manual	Better Organization
Memory Usage	Dynamic allocation	Fixed array	More Efficient

08

KMUTT CPE38

Project.

2025

DEMO CODE WALKTHROUGH

TO DO LIST APPLICATION

CHALLENGE AND SOLUTION

09

Technical Challenges

- **Memory Management**
  Challenge: Preventing memory leaks
  Solution: Careful allocation/deallocation
  Implementation: Free functions for cleanup

- **Date Handling**
  Challenge: Complex date comparisons
  Solution: Custom date structure and comparison function
  Implementation: compareDates() function

- **Sorting Efficiency**
  Challenge: $O(n^2)$ bubble sort performance
  Solution: Limited sorting scope
  Future: Consider quicksort implementation

10

2025 Project. KMUTT CPE38

LEARNING OUTCOME

- **Data Structure Selection:**
  Matching structures to specific requirements
- **Memory Management:**
  Proper allocation/deallocation in C
- **Algorithm Efficiency:**
  Impact of $O(n^2)$ vs $O(n \log n)$ in practice
- **Error Handling:**
  Importance of input validation and edge cases

12 CURRENT ISSUES & LIMITATION AND FUTURE IMPROVEMENT

Current Issue	Description	Future Improvement	Expected Benefit
Sorting Performance	$O(n^2)$ bubble sort becomes slow with large datasets (152ms for 1000 tasks)	Replace with QuickSort algorithm	Improve to $O(n \log n)$, reducing time to ~8ms for 1000 tasks
Search Limitations	$O(n)$ search requires checking each task sequentially (0.250ms for 1000 tasks)	Implement hash table for name lookup	Achieve $O(1)$ constant-time search regardless of list size
Memory Management	Complex cleanup for nested structures increases risk of memory leaks	Implement systematic cleanup sequences	Guaranteed memory deallocation with reduced leak risk
Limited UI	Text-based interface limits visualization and user experience	Develop graphical user interface	Improved usability with visual task boards and progress charts

MEMBER ROLES AND CONTRIBUTES

KMUTT CPE38 Project. 2025

Kulchaya Paipinij 3406

- **Search, Statistics & File I/O**
- Report writing
- Presentation preparation

Chayanit Kuntanarumitkul 3408

- **Scheduling, Date Handling**
- Code Bug Fixing, GitHub
- Repository Management

Siripitch Chaiyabutra 3440

- **Task Management & UI**
- System Architecture
- Flowchart

Q&A

ABOUT THE PROJECT

THANK YOU

WRAPPING UP AND
PARTING THOUGHTS

present by eiei

CHALLENGE AND SOLUTION

Design Decisions

- **Why Linked List?**
  - Dynamic size
  - Efficient insertion/deletion
  - Memory efficiency
- **Why Stack for Completed Tasks?**
  - Natural LIFO behavior
  - Simple undo implementation
  - O(1) operations

TIME COMPLEXITY

- Add and Undo operations run in constant time O(1), showing stable performance regardless of task count. Searching shows O(n) behavior, increasing linearly with more tasks. Completing tasks is nearly constant but may vary slightly. Sorting is the most intensive operation, growing significantly with task count, consistent with O(n log n) time complexity.

```
=== PERFORMANCE ANALYSIS ===

Testing with 10 tasks:
----------------------
Add Task: 0.006 ms
Search Task: 0.004 ms
Complete Task: 0.002 ms
Undo Complete: 0.002 ms
Sort Tasks: 0.004 ms


Testing with 100 tasks:
----------------------
Add Task: 0.006 ms
Search Task: 0.004 ms
Complete Task: 0.004 ms
Undo Complete: 0.005 ms
Sort Tasks: 0.060 ms


Testing with 1000 tasks:
----------------------
Add Task: 0.004 ms
Search Task: 0.022 ms
Complete Task: 0.053 ms
Undo Complete: 0.002 ms
Sort Tasks: 2.838 ms


Press Enter to continue...
```