Engineering Method

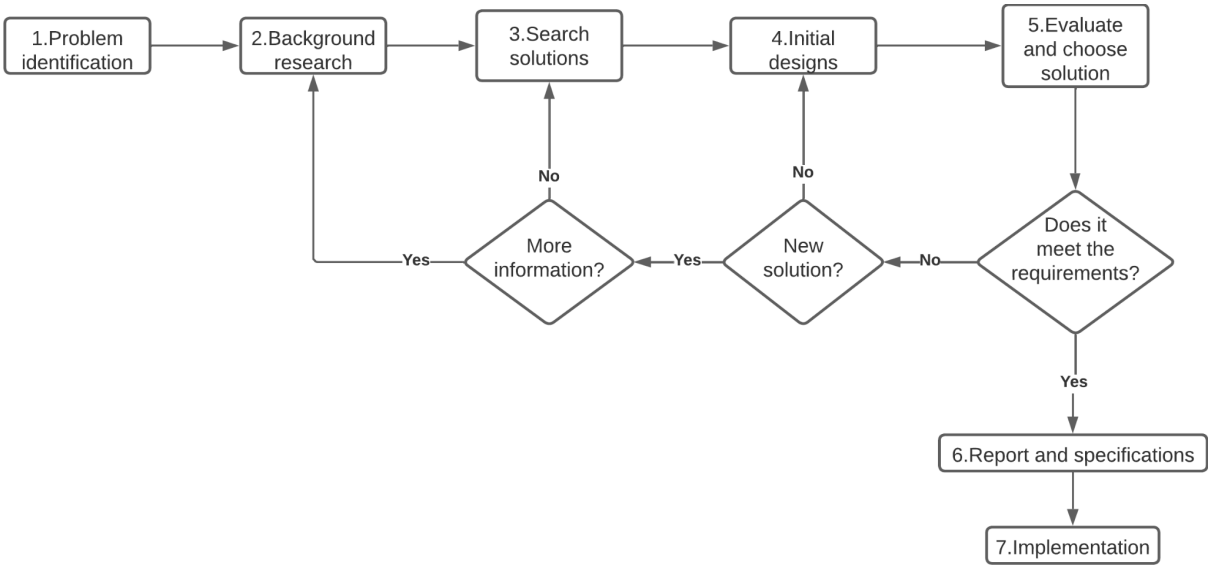
John Maurice Hortua
Santiago Arboleda Velasco
Samuel Viviescas Carrillo

Problem Context

Some students that are currently taking the computer science and discrete structures course at Icesi University, wish to improve their programming skills and their academic performance. With this in mind, they have realized that they lack effective decision-making skills when it comes to prioritizing their daily tasks and activities.

Solution development

In order to solve this problem, the engineering method was chosen for developing the solution, following a systematic approach that aligns with the stated problem. The following flowchart was defined, and we will follow its steps in the process of the development of the solution.



Step 1. Problem identification

Needs assessments

- Icesi's students wish to improve their academic performance and programming skills
- The solution must ensure the addition of a task or a reminder to address the issue.
- The solution to the problem must ensure the capability to edit a task or a reminder.

- The solution to the problem must ensure the ability to remove a task or a reminder.
- The solution must allow the user to assign whether a task is a priority or not.
- The solution must generate a list of tasks and reminders, sorted by deadline or priority.
- The solution must allow the user to undo an action.

Problem definition
Ineffectiveness in decision-making skills regarding the prioritization of daily tasks and activities.

Step 2. Background research

Definitions
Task
A piece of work to be done, especially one done regularly, unwillingly, or with difficulty:

Reminder
a written or spoken message that reminds someone to do something:

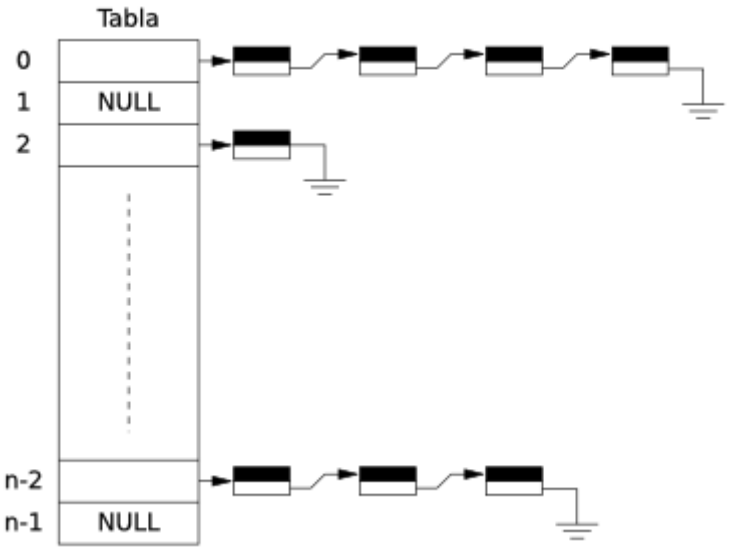
Hash table
Hash tables are data structures used to store large amounts of data requiring highly efficient search and insertion operations. A hash table stores a collection of key-value pairs. Each key is unique for every element in the table and serves as the data used to locate a specific value.

The implementation of a hash table relies on the following components:

1. A reasonably sized table to store the key-value pairs.
2. A 'hash' function that takes the key as input and returns an index for accessing a position in the table.
3. A procedure to handle cases where the aforementioned function returns the same index for two different keys. This situation is known as a collision."

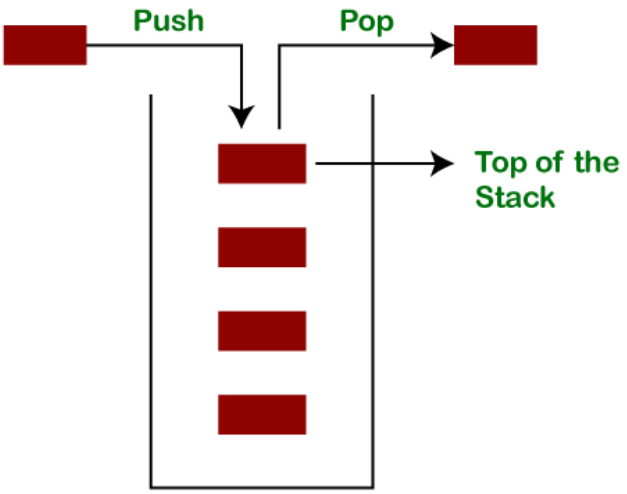
Collision handling

The proposed solution for implementing the hash table combines the table structure with a linked list. Each table position does not store a single element but rather the head of a linked list that contains all those elements whose hash function has returned an identical result. A table position in which no element has been inserted contains a NULL pointer.



Stack

The stack is a linear data structure that is used to store the collection of objects. It is based on Last-In-First-Out (LIFO). It has the two most important operations that are push and pop. The first operation inserts an element into the stack and the second one removes an element from the top of the stack.



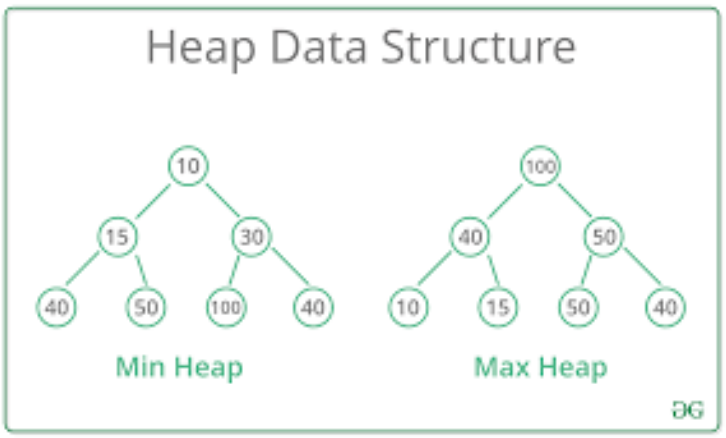
Queue

A queue is an object that represents a data structure designed to have the element inserted at the end of the queue, and the element removed from the beginning of the queue. The order of elements of the queue in Java is FIFO (first-in-first-out). It provides additional operations such as insertion, inspection, and deletion.

Heap

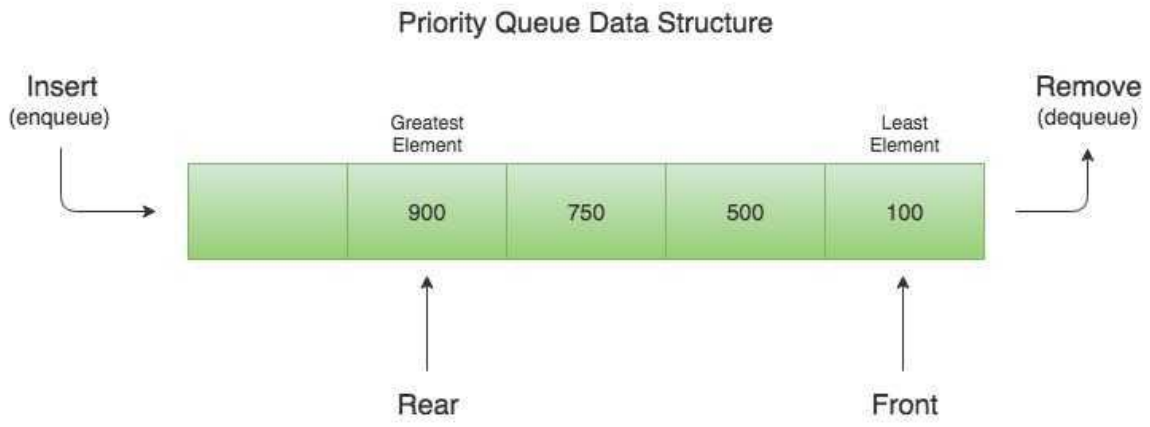
A heap data structure is a complete binary tree that satisfies the heap property, where any given node is

- Always greater than its child node/s and the key of the root node is the largest among all other nodes. This property is also called max heap property.
- Always smaller than the child node/s and the key of the root node is the smallest among all other nodes. This property is also called min heap property.



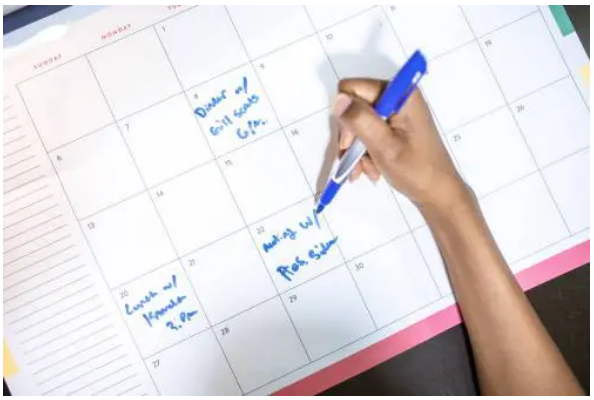
Priority Queue

Is a data structure that stores elements in a specific order based on their priority. It allows accessing the highest-priority element in the queue efficiently. The priority queue utilizes a heap to organize its elements. This indicates that the elements within the priority queue are arranged based on a comparator, which can be as basic as a regular number comparator.



Step 3. Search solutions

<u>Alternative 1: Paper Calendar</u>



It is a calendar in which you can write your activities, a physical tool used to organize and plan events, tasks, and commitments over a specific period of time, usually a year. This type of calendar consists of sheets printed with dates and days of the week, allowing individuals to note their activities, appointments, meetings, and other important events on the corresponding dates.

Alternative 2: Google Calendar

Google Calendar allows you to quickly schedule meetings and events, as well as receive reminders about upcoming activities, ensuring you always know what you need to do. Calendar is designed to collaborate, allowing you to share your schedule with others and create multiple calendars for joint use with your team.

Alternative 3: Create your own Task and Reminder Management System

Creating your own system would allow you to personalize the functionalities that you need, for this scenario you might need the program to add, organize and delete tasks and reminders. Also you might add deadlines and assign task priorities so you can know what to do first among all your tasks.

Step 4. Transition from Ideas to Initial designs

The fact that the Paper calendar is limited by the way of accessing data, the lack of advanced functionalities, it is not environmentally friendly and has a higher risk of loss or damage among others, it makes us discard the Alternative 1.

A careful review of the other alternatives leads us to:

Alternative 2: Google Calendar

- Features an intuitive and user-friendly interface. It allows you to add events with a few clicks.
- Allows you to set reminders and notifications for events.
- Allows you to search for past and future events

Alternative 3: Create your own Task and Reminder Management System

- Allows you to add, organize and delete tasks and reminders easily
- Allows you to assign priorities to the tasks

- Doing it by yourself will help you to improve your skills as a programmer
- It will guide through the process of completing your tasks from the highest-priority one to the lowest-priority.

Step 5. Evaluate and choose solution

Criteria:

We are going to define a series of criteria with which we will evaluate the solution alternatives. Based on the results obtained, we will choose the one that best satisfies the needs of the problem presented.

- Criterion A. Solution Accuracy. The alternative provides a solution:
  - [2] Exact
  - [1] Approximate
- Criterion B. Priority Assignment.
  - [2] Yes
  - [1] No
- Criterion C. Completeness.
  - [3] All
  - [2] More than one
  - [1] Only one or none
- Criterion D. Programming Skills Development.
  - [2] High
  - [1] None

Evaluation:

Alternatives	Criterion A	Criterion B	Criterion C	Criterion D	Total
Alternative 2: Google Calendar	Approximate 1	No 1	More than one 2	None 1	5
Alternative 3: Create your own Task and Reminder Management System	Exact 2	Yes 2	All 3	High 2	9

Solution selection:

According to the previous evaluation, Alternative 3 should be selected as it obtained the highest score based on the defined criteria.

Step 6. Reports and specifications

Problem Specification

Problem: Insufficient daily task prioritization skills
Input:
- Task: title, description, deadline, priority
- Reminder: title, description, deadline, priority

Output:
- List of all tasks and reminders organized by priority

ADTs

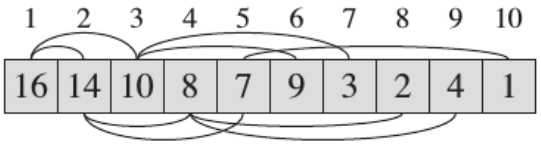
ADT Stack		
Stack = {item3, item2, item1}		
inv: Last in first out		
Basic operations: - **push**	Stack x Element	-> item
- **pop**	Element	-> item
- **peek**	Element	-> item
- **isEmpty**	Element	-> boolean

push(E item)
“Pushes an item onto the top of the stack.”
{pre: true}
{post: stack = {item}}

pop()
“Pops and returns the item on the top of the stack.”
{pre: stack.isEmpty() == false}
{post: stack.size() = stack.size() -1 }

peek()
“Returns the item at the top of the stack without removing it.”
{pre: stack.isEmpty() == false}
{post: true}

isEmpty()
“Returns true if the stack contains no elements.”
{pre: true}
{post: true }

ADT Heap		
		
inv: (valueParentNode >= valueChildrenNode) V **(**valueParentNode <= valueChildreNode)		
Basic operations:		
- **peek**	Heap	-> item
- **size**	Heap	-> int
- **add**	Heap x item	-> item
- **poll**	Heap	-> item
- **swap**	Heap x (int, int)	-> Heap
- **sort**	Heap	-> Heap
- **siftDown**	Heap x (BiPredicate, int, int)	-> Heap
- **siftUp**	Heap x int	-> Heap
- **isEmpty**	Heap	-> boolean

peek()
“ Returns the element at the top of the heap without removing it.”
{pre: heap.isEmpty() == false}
{post: true}

size()
“Returns the number of elements currently in the heap.”
{pre: heap.isEmpty() == false}
{post: true}

add(T item)
“Adds an element to the heap.”
{pre: true}
{post: heap invariant (max, min heap) is still valid}

poll()
“Removes and returns the root element of the heap.”
{pre: heap.isEmpty() == false}
{post: heap.get(0) is removed, heap invariant still valid }

swap(int i, int j)
“Swaps two elements in the heap array by index.”
{pre: indices must be valid}
{post: heap invariant still valid, values indices have been exchanged}

sort()
“Sorts the elements in the heap using heapsort.”
{pre: valid heap invariant }
{post: elements are rearranged}

siftDown(BiPredicate<T,T> comparator, int index, int size)
“Sift down operation to maintain heap order property.”
{pre: valid heap invariant}
{post: element has been moved to its correct position}

siftUp(int index)
“Elements are compared with their parent and swapped if necessary”
{pre: element is a part of the heap}
{post: element at given position is less than or equal to its parent}

isEmpty()
“Returns whether the heap is empty.”
{pre: true}
{post: true}

ADT PriorityQueue		
PriorityQueue = {(Task<title, descrp, …, priority>), (Task<title, descrp, …, priority>)}		
inv: (priority == 0 V priority == 1)		
Basic operations: - **peek** - **size** - **add** - **poll** - **isEmpty**	 Heap Heap Heap x item Heap Heap	 -> item -> int -> item -> item -> boolean

peek()
“ Returns the element at the top of the heap without removing it.”
{pre: heap.isEmpty() == false}
{post: true}

size()
“Returns the number of elements currently in the heap.”
{pre: heap.isEmpty() == false}
{post: true}

add(T item)
“Inserts an item into the priority queue.”
{pre: true}
{post:new element is added to the PriorityQueue in a position that maintains the priority order. }

poll()
“Retrieves and removes the head of this queue, or returns null if this queue is empty.”
{pre: true }
{post: element with the highest priority is removed from the PriorityQueue.}

isEmpty()
“Returns whether the heap is empty.”
{pre: true}
{post: true}

ADT Hash Table		
Set of keys: K = {k1, k2, k3} **hash function:** h: k->t, t = table position		
inv: size >= 0, values != null		
Basic operations:		
- **add**	LinkedList x (key, value)	-> entry
- **remove**	LinkedList x key	-> entry
- **key**	LinkedList x value	-> entry
- **values**	LinkedList	-> LinkedList
- **calculateIndex**	LinkedList x key	-> int
- **iterator**	LinkedList	-> entry
- **get**	LinkedList x key	-> entry

add(K key, V item)
“Adds the new key-value pair to the hash table”
{pre: true}
{post: key-value pair is added to the hash table.}

remove(K key)
“removes the entry from the linked list”
{pre: key is valid}
{post: If the key exists it has been removed, along with its associated value.}

key(V value)
“search for a key, based on a given value in the hashtable”
{pre: value != null}
{post: if value is in the hashtable return corresponding key, else null}

values()
“collects and returns all the values stored in the hash table”
{pre Each linked list within the hash table contains entries of type entry<K, V>:}
{post: returns a linked list that contains all the values from the hash table.}

calculateIndex(K key)
“calculates an index in a hash table based on the hash code of a given key.”
{pre: key != null}
{post: returns an index within the bounds of the hash table (0 <= index < table.length).}

iterator()
“allows to traverse all the values stored while efficiently skipping empty buckets. ”
{pre: true}
{post: The iterator allows traversal of all values stored in the hash table.}

get(K key)
“retrieve the value associated with a given key”
{pre: key != null}
{post: If the key is found returns the associated value, else return null }

Step 7. Implementation

Implementation of the solution in Java:

List of task to implement:

- Task creation
- Reminder creation
- Edit task
- Delete task
- Undo

Subroutine specifications

Name:	
Description:	
Input:	
Output	