**John Maurice Hortua**

**Santiago Arboleda Velasco**

**Samuel Viviescas Carrillo**

# **Problem Context**

Students enrolled in the Algorithms and Discrete Structures course at Icesi University are facing challenges in understanding and applying graph theory concepts. The difficulty arises from the abstract nature of these concepts, making their application less intuitive. Additionally, students are confronted with a task where they are required to design and implement a game with a minimum of 50 vertices and 50 edges. The interesting aspect is that students have the opportunity to propose the game themselves.

# **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

* The solution must ensure the use of 2 algorithms, whether they are Graph Traversals (BFS, DFS), Minimum Weight Paths (Dijkstra, Floyd-Warshall), or Minimum Spanning Tree (MST) (Prim, Kruskal).
* The solution to the problem must ensure the automatic switch to at least 2 different versions of algorithm implementations.
* The solution to the problem must ensure a user-friendly graphical interface.
* The solution should provide assistance and instructions to the user.

**Definición del problema**

Students in the Algorithms and Discrete Structures course at Icesi University must design and implement a game with a minimum of 50 vertices and 50 edges, with the added twist that students can propose the game themselves.

# **Step 2. Background research**

*Definitions*

# **Step 3. Search solutions**

**Alternative 1:**

**Alternative 2:**

**Alternative 3:**

Es por eso que el juego “Trap the Cat” presenta una situación problemática en la que un gato intenta escapar de un tablero y el jugador debe evitarlo. La solución al problema se logra mediante la estrategia y la anticipación de los movimientos del gato. Al hacer clic en los círculos grises, estos se vuelven oscuros y forman una barrera que el gato no puede cruzar. El desafío es prever por dónde intentará huir el gato y bloquear su camino antes de que alcance el borde del tablero y escape. Es un juego que requiere pensamiento crítico y habilidad para resolver problemas bajo presión, ya que cada movimiento del jugador puede facilitar o dificultar la captura del gato.

Todo esto lograra que el estudiante aterrice esos conceptos poniendolo en practica en un problema desafiante

**Step 4. Transition from Ideas to Initial designs**

**Alternative 2:**

**Alternative 3:**

# **Step 5. Evaluate and choose solution**

Criteria:

Evaluation:

| Alternatives | Criterion A | Criterion B | Criterion C | Criterion D | Total |
| --- | --- | --- | --- | --- | --- |
| **Alternative** | Approximate  1 | No  1 | More than one  2 | None  1 | 5 |
| **Alternative** | Exact  2 | Yes  2 | All  3 | High  2 | 9 |

**Solution selection:**

# **Step 6. Reports and specifications**

Problem Specification

**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 |

**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** Heap -> item * **size** Heap -> int * **add** Heap x item -> item * **poll** Heap -> item * **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)**

“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 -> void * **key** LinkedList x value -> entry * **values** LinkedList -> LinkedList * **getIndex** LinkedList x key -> int * **get** LinkedList x (bucket, key) -> entry * **get** V x key -> value * **size** int -> int * **peek** V -> value * **hash** int x key -> int * **getLoadFactor** double -> double |

**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.}

**getIndex(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).}

**get(LinkedList<Entry<K,V>> bucket, K key)**

“retrieve the entry associated with a given key”

{pre: key != null}

{post: If the key is found returns the associated value, else return null }

**get(K key)**

“retrieve the value associated to an entry given a key”

{pre: key != null}

{post: If the key is found returns the associated value, else return null }

**hash(K key)**

“hashes the given key”

{pre: key != null}

{post: returns the result of hashing the key }

# **Step 7. Implementation**

Solution implementation in Java:

**List of task to implement:**

* Create Task
* Create Reminder
* Edit task
* Delete task
* Undo

**Subroutine specifications**

| Name: | Create Task |
| --- | --- |
| Description: | allows to create a task |
| Input: | - title: String, title of the task  -description: String, what is the purpose of the task  -deadline:LocalDateTime, date to finish the task  -newHasPriority: Priority, if the task has priority or not |
| Output | Task , task with all parameters setted |

Implementation

| Name: | Create Reminder |
| --- | --- |
| Description: | allows to create a reminder |
| Input: | - title: String, title of the reminder  -description: String, what do you want to remind later  -deadline:LocalDateTime, date to remind  -hasPriority: Priority, if the reminder has priority or not |
| Output | Reminder, reminder with all parameters setted |

implementation

| Name: | Edit |
| --- | --- |
| Description: | allows to edit a task |
| Input: | - title: String, title of the task that is going to edit  -index: int, task index  -newTitle: String, new task title  -newDescription: String, new description  -newDeadline:LocalDateTime, new due date  -newHasPriority: Priority, if the task has priority or no |
| Output | Task , task with new parameters setted |

implementation

| Name: | Delete |
| --- | --- |
| Description: | allows to delete a task |
| Input: | -taskTitle: String, title of the task to delete  -index: int, index of the task |
| Output | nill , task deleted |

| Name: | Undo |
| --- | --- |
| Description: | allows to reverse the most recent action |
| Input: |  |
| Output | nil, restored last command |