Act 6.1 - Póster Argumentativo de Reflexión

Jesus Raul Jimenez Perez

A01666493

👨🏽‍💻 **Act1.1-Templates**

Los templates, o plantillas, en C++ permiten escribir código genérico y reutilizable. En lugar de escribir funciones y clases múltiples para cada tipo de dato, se puede escribir una sola plantilla que funcione con cualquier tipo de dato. Por ejemplo, una función para sumar dos números puede definirse como:

```
template <typename T>
T add(T a, T b) {
    return a + b;
}
```

Aquí, `T` es un parámetro de tipo que será reemplazado por el tipo real cuando se use la función. Las plantillas pueden ser usadas tanto para funciones como para clases, permitiendo una flexibilidad y reutilización significativa del código. Sin embargo, pueden complicar la depuración y el tiempo de compilación debido a su naturaleza genérica y expansiva.

👨🏽‍💻 **Act1.2-Recursion**

La recursión es una técnica de programación donde una función se llama a sí misma para resolver un problema. Un ejemplo clásico es el cálculo del factorial de un número:

```
int factorial(int n) {
    if (n <= 1) return 1; // caso base
    else return n * factorial(n - 1); // caso recursivo
}
```

Cada llamada recursiva descompone el problema en uno más pequeño, hasta alcanzar el caso base, que termina la recursión. La recursión es intuitiva para problemas que pueden dividirse en subproblemas idénticos más pequeños, como en la búsqueda binaria y el algoritmo de la Torre de Hanoi. Sin embargo, la recursión puede consumir más memoria

debido al uso de la pila de llamadas y puede ser ineficiente si no se maneja correctamente.

👨🏽‍💻 Act1.3-BigO

La notación Big O describe la eficiencia de un algoritmo en términos de tiempo y espacio a medida que el tamaño de la entrada crece. Es una medida del peor caso o comportamiento asintótico de un algoritmo. Algunas notaciones comunes incluyen:

- O(1): Tiempo constante, independiente del tamaño de la entrada.
- O(n): Tiempo lineal, crece proporcionalmente con el tamaño de la entrada.
- O(n^2): Tiempo cuadrático, crece con el cuadrado del tamaño de la entrada.
- O(log n): Tiempo logarítmico, crece con el logaritmo del tamaño de la entrada.

El análisis Big O ayuda a los desarrolladores a elegir el algoritmo más eficiente para una tarea específica, optimizando el rendimiento y el uso de recursos.

👨🏽‍💻 Act1.4-Search

Los algoritmos de búsqueda encuentran elementos dentro de estructuras de datos. Los dos algoritmos básicos son:

- **Búsqueda lineal**: Recorre cada elemento de una lista hasta encontrar el objetivo o llegar al final. Tiene una complejidad de O(n).

```
int linearSearch(int arr[], int n, int x) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == x) return i;
    }
    return -1;
}
```

- **Búsqueda binaria**: Requiere una lista ordenada. Divide la lista a la mitad repetidamente hasta encontrar el objetivo o reducir el intervalo a cero. Tiene una complejidad de O(log n).

```
int binarySearch(int arr[], int l, int r, int x) {
    while (l <= r) {
        int m = l + (r - l) / 2;
        if (arr[m] == x) return m;
        if (arr[m] < x) l = m + 1;
        else r = m - 1;
```

```
    }
    return -1;
}
```

👨🏽‍💻 **Act1.5-Sort**

Los algoritmos de ordenamiento organizan elementos de una lista. Algunos de los más comunes son:

- **Ordenamiento de burbuja**: Intercambia repetidamente elementos adyacentes si están en el orden incorrecto. Tiene una complejidad de O(n^2).

```
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n-1; i++)
        for (int j = 0; j < n-i-1; j++)
            if (arr[j] > arr[j+1]) swap(arr[j], arr[j+1]);
}
```

- **Ordenamiento por inserción**: Construye la lista ordenada de un elemento a la vez, insertando cada nuevo elemento en su posición correcta. Tiene una complejidad de O(n^2).

```
void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

- **Quicksort**: Divide y vencerás. Selecciona un pivote y reordena los elementos para que todos los menores estén antes y todos los mayores estén después. Recursivamente ordena las sublistas. Tiene una complejidad promedio de O(n log n), pero O(n^2) en el peor caso.

```
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);
```

```
    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

- **Mergesort**: Divide la lista en dos mitades, las ordena y las combina. Tiene una complejidad de O(n log n) y es estable.

```
void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;
    int L[n1], R[n2];
    for (int i = 0; i < n1; i++) L[i] = arr[l + i];
    for (int j = 0; j < n2; j++) R[j] = arr[m + 1 + j];
    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) arr[k] = L[i++];
        else arr[k] = R[j++];
        k++;
    }
    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];
}

void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
```

```
    }
}
```

👨🏽‍💻 Act1.6-Linked-lists

Las listas enlazadas son estructuras de datos donde cada nodo contiene un valor y una referencia (enlace) al siguiente nodo. Una lista simplemente enlazada tiene nodos conectados en una secuencia lineal.

```
struct Node {
    int data;
    Node* next;
};

void append(Node** head_ref, int new_data) {
    Node* new_node = new Node();
    Node *last = *head_ref;
    new_node->data = new_data;
    new_node->next = nullptr;
    if (*head_ref == nullptr) {
        *head_ref = new_node;
        return;
    }
    while (last->next != nullptr) last = last->next;
    last->next = new_node;
}
```

Las listas enlazadas permiten inserciones y eliminaciones eficientes, pero el acceso a los elementos es más lento que en los arreglos, ya que debe seguirse la referencia nodo por nodo.

👨🏽‍💻 Act2.2-Double-Linked-lists

Las listas doblemente enlazadas son una extensión de las listas simplemente enlazadas donde cada nodo tiene dos referencias: una al siguiente nodo y otra al nodo anterior. Esto permite una navegación bidireccional.

```
struct Node {
    int data;
    Node* next;
    Node* prev;
```

```
};

void append(Node** head_ref, int new_data) {
    Node* new_node = new Node();
    Node* last = *head_ref;
    new_node->data = new_data;
    new_node->next = nullptr;
    if (*head_ref == nullptr) {
        new_node->prev = nullptr;
        *head_ref = new_node;
        return;
    }
    while (last->next != nullptr) last = last->next;
    last->next = new_node;
    new_node->prev = last;
}
```

Las listas doblemente enlazadas facilitan las operaciones de inserción y eliminación en ambas direcciones, pero requieren más memoria debido a las referencias adicionales.

👨🏽‍💻 Act2.3-Queues

Las colas son estructuras de datos lineales que siguen el principio FIFO (First In, First Out). Las operaciones principales son:

- **Enqueue**: Añadir un elemento al final de la cola.
- **Dequeue**: Eliminar el elemento del frente de la cola.

```
struct Queue {
    int front, rear, size;
    unsigned capacity;
    int* array;
};

Queue* createQueue(unsigned capacity) {
    Queue* queue = new Queue();
    queue->capacity = capacity;
    queue->front = queue->size = 0;
    queue->rear = capacity - 1;
    queue->array = new int[queue->capacity];
    return queue;
}
```

```
void enqueue(Queue* queue, int item) {
    queue->rear = (queue->rear + 1) % queue->capacity;
    queue->array[queue->rear] = item;
    queue->size = queue->size + 1;
}

int dequeue(Queue* queue) {
    int item = queue->array[queue->front];
    queue->front = (queue->front + 1) % queue->capacity;
    queue->size = queue->size - 1;
    return item;
}
```

Las colas se utilizan en la gestión de tareas, simulaciones de sistemas, y en algoritmos como el de búsqueda en anchura (BFS) en grafos.

👨🏽‍💻 Act2.4-Stack

Las pilas son estructuras de datos lineales que siguen el principio LIFO (Last In, First Out). Las operaciones principales son:

- **Push**: Añadir un elemento al tope de la pila.
- **Pop**: Eliminar el elemento del tope de la pila.

```
struct Stack {
    int top;
    unsigned capacity;
    int* array;
};

Stack* createStack(unsigned capacity) {
    Stack* stack = new Stack();
    stack->capacity = capacity;
    stack->top = -1;
    stack->array = new int[stack->capacity];
    return stack;
}

void push(Stack* stack, int item) {
    stack->array[++stack->top] = item;
}

int pop(Stack* stack) {
```

```
    return stack->array[stack->top--];
}
```

Las pilas son fundamentales en la evaluación de expresiones aritméticas, la gestión de la memoria durante las llamadas a funciones, y la implementación de algoritmos de retroceso.

👨🏽‍💻 Act2.6-Arboles-binarios

Los árboles binarios son estructuras jerárquicas en las que cada nodo tiene como máximo dos hijos: izquierdo y derecho. Un árbol binario de búsqueda (BST) es un tipo especial donde para cada nodo, los valores de los nodos en su subárbol izquierdo son menores y en su subárbol derecho son mayores.

```
struct Node {
    int key;
    Node* left, *right;
};

Node* newNode(int item) {
    Node* temp = new Node();
    temp->key = item;
    temp->left = temp->right = nullptr;
    return temp;
}

Node* insert(Node* node, int key) {
    if (node == nullptr) return newNode(key);
    if (key < node->key) node->left = insert(node->left, key);
    else if (key > node->key) node->right = insert(node->right, ke
y);
    return node;
}
```

Los recorridos en un árbol binario incluyen:

- **Preorden**: Visitar el nodo, luego su subárbol izquierdo y luego su subárbol derecho.
- **Inorden**: Visitar el subárbol izquierdo, luego el nodo, y luego el subárbol derecho.
- **Postorden**: Visitar el subárbol izquierdo, luego el derecho, y luego el nodo.

👨🏽‍💻 Act2.7-Arbol-Heap

Un montículo es un árbol binario completo donde cada nodo cumple la propiedad del montículo. En un montículo máximo, el valor de cada nodo es mayor o igual que los valores de sus hijos; en un montículo mínimo, el valor de cada nodo es menor o igual que los valores de sus hijos.

```
void heapify(int arr[], int n, int i) {
    int largest = i;
    int l = 2 * i + 1;
    int r = 2 * i + 2;
    if (l < n && arr[l] > arr[largest]) largest = l;
    if (r < n && arr[r] > arr[largest]) largest = r;
    if (largest != i) {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}

void heapSort(int arr[], int n) {
    for (int i = n / 2 - 1; i >= 0; i--) heapify(arr, n, i);
    for (int i = n - 1; i > 0; i--) {
        swap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
}
```

Los montículos se utilizan principalmente para implementar colas de prioridad, donde la extracción del elemento máximo o mínimo es eficiente.

👨🏽‍💻 Act3.1-Grafos

Los grafos son estructuras de datos que consisten en un conjunto de vértices (nodos) y aristas (conexiones) que pueden ser dirigidas o no dirigidas. Los grafos pueden representarse mediante matrices de adyacencia o listas de adyacencia. Los algoritmos comunes en grafos incluyen:

- **Búsqueda en profundidad (DFS)**: Explora tan lejos como sea posible a lo largo de cada rama antes de retroceder.

  ```
  void DFSUtil(int v, bool visited[], list<int> adj[]) {
      visited[v] = true;
      for (auto i = adj[v].begin(); i != adj[v].end(); ++i)
          if (!visited[*i]) DFSUtil(*i, visited, adj);
  }
  ```

```
void DFS(int V, list<int> adj[]) {
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++) visited[i] = false;
    for (int i = 0; i < V; i++)
        if (visited[i] == false) DFSUtil(i, visited, adj);
}
```

- **Búsqueda en anchura (BFS)**: Explora todos los vecinos de un vértice antes de pasar a los vecinos de esos vecinos.

```
void BFS(int s, list<int> adj[], int V) {
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++) visited[i] = false;
    list<int> queue;
    visited[s] = true;
    queue.push_back(s);
    list<int>::iterator i;
    while(!queue.empty()) {
        s = queue.front();
        queue.pop_front();
        for (i = adj[s].begin(); i != adj[s].end(); ++i) {
            if (!visited[*i]) {
                visited[*i] = true;
                queue.push_back(*i);
            }
        }
    }
}
```

- **Algoritmo de Dijkstra**: Encuentra el camino más corto desde un vértice de origen a todos los demás vértices en un grafo ponderado no dirigido.

```
void dijkstra(int graph[V][V], int src) {
    int dist[V];
    bool sptSet[V];
    for (int i = 0; i < V; i++) {
        dist[i] = INT_MAX;
        sptSet[i] = false;
    }
    dist[src] = 0;
    for (int count = 0; count < V-1; count++) {
        int u = minDistance(dist, sptSet);
        sptSet[u] = true;
```

```
        for (int v = 0; v < V; v++)
            if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX &
& dist[u] + graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
    }
}
```

👨🏽‍💻 Act3.2-HashTable

Las tablas hash son estructuras de datos que implementan un mapeo de claves a valores utilizando una función hash. Permiten el acceso rápido a los datos, típicamente en O(1) tiempo. Una función hash convierte una clave en un índice en un arreglo. Las colisiones, cuando dos claves tienen el mismo índice, se manejan comúnmente mediante:

- **Encadenamiento**: Cada índice de la tabla apunta a una lista enlazada de elementos.

```
struct Node {
    int key;
    int value;
    Node* next;
};

void insert(Node** table, int key, int value, int table_size) {
    int index = hashFunction(key, table_size);
    Node* new_node = new Node();
    new_node->key = key;
    new_node->value = value;
    new_node->next = table[index];
    table[index] = new_node;
}
```

- **Exploración lineal**: Encuentra el siguiente índice libre en la tabla.

```
int hashFunction(int key, int table_size) {
    return key % table_size;
}

void insert(int table[], int key, int table_size) {
    int index = hashFunction(key, table_size);
    while (table[index] != -1) index = (index + 1) % table_size;
    table[index] = key;
}
```

Las tablas hash son ampliamente utilizadas en aplicaciones como bases de datos, sistemas de archivos y almacenamiento en caché.