

Parcial 3

Jaime Darley Angulo Tenorio

Facultad de Ingeniería

Departamento de Ingeniería de Sistemas e Industrial

Sistemas Operativos

Tarea 1

1.1 Cálculo PAPA

1.2 Probabilidad de Selección de Estudiante

Tarea 2

| **Estructura** | **¿Para qué sirve?** | **¿Cuándo utilizarla?** | **Operaciones clave** |
| --- | --- | --- | --- |
| **Static Array** | Colección indexada de elementos del mismo tipo con tamaño fijo | Cuando la cantidad exacta de elementos es conocida y se busca velocidad en acceso    Ej. Almacenar N notas mensuales de un estudiante | Insertar: .append(valor)  Eliminar: .remove(valor)  Acceder: .get(i) |
| **Dynamic Array** | Colección que puede adaptar su tamaño según se agreguen o eliminen elementos. | Cuando la cantidad de datos puede cambiar con el tiempo.    Ej. Gestionar un carrito de compras con productos que varían dinámicamente. | Insertar: .append(valor)  Eliminar: .remove(valor)  Acceder: .get(i) |
| **Singly-Linked list** | Secuencia de nodos donde cada uno apunta al siguiente. | Cuando necesitas insertar o eliminar elementos frecuentemente sin mover muchos datos.    Ej. Registrar tareas pendientes que cambian con frecuencia. | Insertar al inicio: .addFirst()  Insertar después de nodo: .insertAfter()  Eliminar: .remove(valor)  Recorrer: .traverse() |
| **Doubly-Linked list** | Secuencia de nodos donde cada uno apunta al anterior y al siguiente. | Para navegar hacia adelante y atrás.      Ej. Implementar historial de edición con opción de deshacer y rehacer. | Insertar al inicio/final: .addFirst(), .addLast()  Eliminar nodo: .remove()  Recorrer en ambos sentidos: .traverseForward(), .traverseBackward() |
| **Queue** | Estructura que procesa en el orden de llegada siguiendo el principio FIFO, el primer elemento en entrar es el primero en salir | Útil en procesamiento por turnos o tareas por orden de llegada.      Ej. Manejar una fila de solicitudes en soporte técnico, donde se atiende en orden de llegada. | Encolar: .enqueue(valor)  Desencolar: .dequeue()  Ver frente: .front() |
| **Stack** | Estructura que procesa en orden inverso de llegada siguiendo el principio LIFO, el último elemento en entrar es el primero en salir. | Ideal para manejar retrocesos, historial o resolver expresiones.    Ej. Implementar una función de deshacer cambios en un editor de texto. | Apilar: .push(valor)  Desapilar: .pop()  Ver tope: .top() |
| **Deque** | Cola que permite inserción y eliminación por ambos extremos. | Donde se requiere manejo dinámico en ambos extremos.      Ej. Manejar el historial de navegación de un navegador, permitiendo ir hacia adelante y atrás. | Insertar frente: .addFront(valor)  Insertar final: .addBack(valor)  Eliminar frente: .removeFront()  Eliminar final: .removeBack() |
| **Heap** | Cola que mantiene disponible el elemento con mayor o menor prioridad. | Para implementar agendas por prioridad.      Ej. Seleccionar al paciente más urgente en una sala de emergencias, priorizando gravedad sobre orden de llegada. | Insertar: .add(valor)  Ver mayor prioridad: .peek()  Eliminar prioridad: .poll() |
| **HashMap** | Mapa que almacena pares clave-valor con acceso rápido usando hash. | Cuando se necesita búsquedas rápidas sin importar el orden de los datos.    Ej. Consultar rápidamente el precio de un producto a partir de su código en un inventario. | Insertar: .put(clave, valor)  Buscar: .get(clave)  Eliminar: .remove(clave) |
| **TreeMap** | Mapa ordenado o por clave que usa un árbol binario balanceado. | Útil cuando se requiere mantener las claves ordenadas para consultas por rango.      Ej. Almacenar eventos con fechas clave y recorrerlos en orden cronológico. | Insertar: .put(clave, valor)  Buscar por rango: .firstKey(), .lastKey()  Eliminar: .remove(clave) |
| **HashSet** | Conjunto de no ordenado de elementos, con acceso rápido usando hash y sin duplicados. | Para validar existencia rápida sin repetir datos.      Ej. Guardar correos únicos de personas que se registran a una plataforma sin importar el orden. | Insertar: .add(valor)  Buscar: .contains(valor)  Eliminar: .remove(valor) |
| **TreeSet** | Conjunto ordenado usando un árbol binario balanceado, sin duplicados. | Cuando además de unicidad, se necesita mantener los elementos ordenados.      Ej. Mantener puntajes únicos en un ranking de juego y mostrar los mejores de forma ordenada. | Insertar: .add(valor)  Buscar mínimo/máximo: .first(), .last()  Buscar por rango: .subSet(a,b) |
| **MultiSet** | Colección que permite elementos repetidos y lleva el conteo de ocurrencias, combinando características de conjuntos y listas. | Cuando es importante saber cuántas veces aparece un elemento.    Ej. Contar palabras en un texto o frecuencias en un censo. | Insertar: .add(valor)  Contar ocurrencias: .count(valor)  Eliminar una ocurrencia: .remove(valor)  Ver todos: .elements() |